Le développeur Java™ 2

édition 1999

Antoine Mirecourt

OSMAN EYROLLES MULTIMEDIA
1, rue Thénard 75005 Paris

Fax : 01 44 41 11 85
http://www.eyrolles.com

ISBN 2-212-25003-7

Le développeur Java™ 2

édition 1999

Antoine Mirecourt

OSMAN EYROLLES MULTIMEDIA

Dans la même collection:

- L'Ordinator, WINDOWS 98 édition 1999
- L'Explorator, INTERNET édition 1999
- L'Ordinator, PC édition 1999

Sommaire

Introduction xxi

Pourquoi Java ? xxi
Qu'est-ce que Java ? xxv
Java est un langage orienté objets xxv
Java est extensible à l'infini xxviii
Java est un langage à haute sécurité xxviii
Java est un langage simple à apprendre xxx
Java est un langage compilé xxxi
Les compilateurs natifs xxxi
Les compilateurs de bytecode xxxii
Les compilateurs JIT xxxii
Le premier langage du troisième millénaire ? xxxiii
Voir Java à l'œuvre xxxiv

Chapitre 1 : Installation de Java 1

Installation du JDK 3
Ce que contient le JDK 4
Configuration de l'environnement 8
Le chemin d'accès aux programmes exécutables 8
Le chemin d'accès aux classes Java 10
Améliorer le confort d'utilisation 10
Le "Java Runtime Environment" 12
Installation du JRE Windows 14
Installation de la documentation en ligne 15
Un navigateur HTML compatible Java 17
HotJava 17
Netscape Navigator 18
Internet Explorer 19
Un éditeur pour la programmation 20
Ce que doit offrir un éditeur 21
UltraEdit 21
Quelques petits fichiers batch qui vous simplifieront la vie 22
Un environnement de développement 23
Avantages 24
Inconvénients 26
Quel environnement ? 26
Où trouver les informations ? 27
Le site Web de Sun 27
Le JDC 28
Les autres sites Web dédiés à Java 28
Les Newsgroups 29

Chapitre 2 : Votre premier programme 31

Première version : un programme minimal 31
Analyse du premier programme 36
Première applet 46

Analyse de la première applet 49
Une façon plus simple d'exécuter les applets 55
Résumé 56
Exercice 56

Chapitre 3 : Les objets Java 57

Tout est objet 60
Les classes 61
Pertinence du modèle 62
Les instances de classes 65
Les membres de classes 66
Les membres statiques 67
Les constructeurs 69
Création d'un objet 72
La destruction des objets : le garbage collector 76
Comment retenir les objets : les handles 78
Création des handles 79
Modifier l'affectation d'un handle 79
Résumé 82
Exercice 82

Chapitre 4 : Les primitives et les handles 83

Les primitives 85
Utiliser les primitives 87
Valeurs par défaut des primitives 91
Différences entre les objets et les primitives 93
Les valeurs littérales 99
Le casting des primitives 101
Le casting explicite 102
Casting d'une valeur entière en valeur flottante 103

Casting d'une valeur flottante en valeur entière 104
Formation des identificateurs 104
Portée des identificateurs 106
Portée des identificateurs de méthodes 112
Les objets n'ont pas de portée 113
Les chaînes de caractères 113
Constantes 117
Utilisation de final avec des objets 118
Accessibilité 118
Retour sur les variables statiques 118
Masquage des identificateurs de type static 126
Java et les pointeurs 129
Exercices 131
Première question 131
Deuxième question 131
Résumé 132

Chapitre 5 : Créez vos propres classes 133

Tout est objet (bis) 134
L'héritage 135
Les constructeurs 138
Référence à la classe parente 141
La redéfinition des méthodes 141
Surcharger les méthodes 143
Signature d'une méthode 145
Optimisation 147
Surcharger les constructeurs 151
Les constructeurs par défaut 154
Les initialiseurs 162
Les initialiseurs de variables d'instances 162

Les initialiseurs d'instances 164
Ordre d'initialisation 165
Méthodes : les valeurs de retour 168
Surcharger une méthode avec une méthode de type différent 172
Distinction des méthodes surchargées par le type uniquement .. 173
Le retour 176
Résumé 177
Exercice 178

Chapitre 6 : Les opérateurs 183

Java et les autres langages 184
L'opérateur d'affectation 185
Raccourci 186
Les opérateurs arithmétiques à deux opérandes 187
Les opérateurs à deux opérandes et le sur-casting automatique . 188
Les raccourcis 191
Les opérateurs à un opérande 193
Les opérateurs relationnels 196
Les opérateurs logiques 204
Les opérateurs d'arithmétique binaire 207
Des conséquences du sur-casting automatique sur l'extension de zéro 214
Utilisation des opérateurs d'arithmétique binaire avec des valeurs logiques 218
Utilisation de masques binaires 219
L'opérateur à trois opérandes ?: 220
Les opérateurs de casting 221
Les opérateurs de casting et les valeurs littérales 223
L'opérateur new 223
L'opérateur instanceof 224

Priorité des opérateurs 224
Résumé 229

Chapitre 7 : Les structures de contrôle 231

La séquence 232
Le branchement par appel de méthode 232
L'instruction de branchement return 235
L'instruction conditionnelle if 235
L'instruction conditionnelle else 237
Les instructions conditionnelles et les opérateurs ++ et -- 239
Les instructions conditionnelles imbriquées 240
La boucle for 244
L'initialisation 245
Le test 246
L'incrémentation 248
Le bloc d'instructions 249
Modification des indices à l'intérieur de la boucle 249
Imbrication des boucles for 250
Type des indices 250
Portée des indices 252
Sortie d'une boucle par return 256
Branchement au moyen des instructions break et continue 256
Utilisation de break et continue avec des étiquettes 258
L'instruction while 259
L'instruction switch 265
L'instruction synchronized 272
Résumé 273

Chapitre 8 : L'accessibilité 275

Les packages (Où) 279

Chemins d'accès et packages 282
L'instruction package 283
Placement automatique des fichiers .class dans les répertoires correspondant aux packages 285
Chemin d'accès par défaut pour les packages 286
L'instruction import 287
Packages accessibles par défaut 292
Les fichiers .jar 293
Création de vos propres fichiers .jar ou .zip 293
Comment nommer vos packages ? 295
Ce qui peut être fait (Quoi) 296
static 297
Les variables static 297
Les méthodes static 300
Initialisation des variables static 301
Les initialiseurs statiques 302
final 304
Les variables final 304
Les variables final non initialisées 305
Les méthodes final 306
Les classes final 308
synchronized 308
native 309
transient 309
volatile 310
abstract 310
Les interfaces 312
Qui peut le faire (Qui) 313
public 313
protected 314
package 314
private 315
Autorisations d'accès aux constructeurs 316
Résumé 321

Chapitre 9 : Le polymorphisme 323

Le sur-casting des objets 325
Retour sur l'initialisation 326
Le sur-casting 328
Le sur-casting explicite 332
Le sur-casting implicite 334
Le sous-casting 334
Le "late binding" 337
Les interfaces 343
Utiliser les interfaces pour gérer des constantes 344
Un embryon d'héritage multiple 346
Le polymorphisme et les interfaces 347
Pas de vrai héritage multiple 352
Quand utiliser les interfaces ? 354
Héritage et composition 356
Résumé 360

Chapitre 10 : Les tableaux et les collections 363

Les tableaux 364
Déclaration 364
Initialisation 365
Initialisation automatique 366
Les tableaux littéraux 367
Les tableaux de primitives sont des objets 369
Le sur-casting implicite des tableaux 369
Les tableaux d'objets sont des tableaux de handles 374
La taille des tableaux 376
Les tableaux multidimensionnels 379
Les tableaux et le passage d'arguments 381
Copie de tableaux 384
Les vecteurs 388

Le type Stack 392
Le type BitSet 393
Les tables (Map) 394
Les tables ordonnées (SortedMap) 396
Le type Hashtable 396
Les collections 397
Les listes (List) 399
Les ensembles (Set) 401
Les ensembles ordonnés (SortedSet) 402
Les itérateurs 402
Les itérateurs de listes 404
Les comparateurs 405
Les méthodes de la classe Collections 413
Exemple : utilisation de la méthode sort() 416
Résumé 420

Chapitre 11 : Les objets meurent aussi 421

Certains objets deviennent inaccessibles 421
Que deviennent les objets inaccessibles ? 427
Le garbage collector 428
Principe du garbage collector 428
Optimiser le travail du garbage collector 430
Les finaliseurs 432
Contrôler le travail du garbage collector 437
Références et accessibilité 441
Les références faibles 442
SoftReference 445
WeakReference 445
PhantomReference 445
Les queues 446
Exemple d'utilisation de références faibles 446

Autres formes de contrôle du garbage collector 450
La finalisation et l'héritage 458
La finalisation et le traitement d'erreur 459
Contrôler le garbage collector
à l'aide des options de l'interpréteur 460
Résumé 461

Chapitre 12 : Les classes internes 463

Les classes imbriquées 463
Les classes membres 471
Instances externes anonymes 480
Classes membres et héritage 481
Remarque concernant les classes membres 485
Les classes locales 488
Les handles des instances de classes locales 491
Les classes anonymes 495
Comment sont nommées les classes anonymes 498
Résumé 500

Chapitre 13 : Les exceptions 501

Stratégies de traitement des erreurs 502
Signaler et stopper 503
Corriger et réessayer 503
Signaler et réessayer 503
La stratégie de Java 503
Les deux types d'erreurs de Java 504
Les exceptions 506
Attraper les exceptions 507
Dans quelle direction sont lancées les exceptions ? 509

Manipuler les exceptions 510
Modification de l'origine d'une exception 512
Créer ses propres exceptions 520
La clause finally 525
Organisation des handlers d'exceptions 529
Pour un véritable traitement des erreurs 530
D'autres objets jetables 532
Les exceptions dans les constructeurs 532
Exceptions et héritage 532
Résumé 533

Chapitre 14 : Les entrées/sorties 535

Principe des entrées/sorties 536
Les streams de données binaires 537
Les streams d'entrée 537
Streams de communication 537
Streams de traitement 537
Les streams de sortie 538
Streams de communication 538
Streams de traitement 539
Les streams de caractères 540
Les streams d'entrée 540
Streams de communication 541
Streams de traitement 541
Les streams de sortie 542
Streams de communication 542
Streams de traitement 543
Les streams de communication 543
Lecture et écriture d'un fichier 544
Les streams de traitement 546
Exemple de traitement : utilisation d'un tampon 552

Exemple de traitement : conversion des fins de lignes 553
Compression 554
Décompression 560
La sérialisation 561
Les fichiers à accès direct 564
Résumé 569

Chapitre 15 : Le passage des paramètres 571

Passage des paramètres par valeur 571
Passage des paramètres par référence 572
Passer les objets par valeur 575
Le clonage des objets 576
Clonage de surface et clonage en profondeur 579
Clonage en profondeur d'un objet de type inconnu 585
Clonabilité et héritage 585
Interdire le clonage d'une classe dérivée 589
Une alternative au clonage : la sérialisation 589
Une autre alternative au clonage 594
Résumé 596

Chapitre 16 : Exécuter plusieurs processus simultanément 599

Qu'est-ce qu'un processus ? 600
Avertissement 600
Créer explicitement un thread 601
Exécuter plusieurs threads simultanément 603
Caractéristiques des threads 608
Contrôler les threads 610

Utilisation des groupes pour référencer les threads 613
Gérer la répartition du temps entre les threads 618
La priorité 621
La synchronisation 623
Problèmes de synchronisation 630
Mise en œuvre d'un démon 641
Communication entre les threads : wait et notifyAll 644
Résumé 648
Exercice 648

Chapitre 17 : RTTI et réflexion 651

Le RTTI ou comment Java vérifie les sous-castings explicites 651
Connaître la classe d'un objet 654
Instancier une classe à l'aide de son objet Class 657
Connaître la classe exacte d'un objet 658
Utiliser la réflexion pour connaître le contenu d'une classe 659
Utilité de la réflexion 659
Utiliser la réflexion pour manipuler une classe interne 666
Utiliser la réflexion pour créer des instances 667
Conclusion : quand utiliser la réflexion ? 668

Chapitre 18 : Fenêtres et boutons 669

Les composants lourds et les composants légers 670
Les composants lourds 670
Les composants légers 671
Le look & feel 672
Les fenêtres 673
Hiérarchie des fenêtres Java 673

Structure d'une fenêtre 674
Les layout managers 677
Créer une application fenêtrée 677
Quel événement intercepter ? 679
Intercepter les événements de fenêtre 680
Les événements et les listeners 680
Utilisation des composants 691
Utilisation des layout managers 693
Philosophie des layout managers 694
Bien utiliser les layout managers 699
Utilisation du FlowLayout 703
Utilisation d'un layout à plusieurs niveaux 708
Utilisation du GridBagLayout 711
Rendre l'interface réellement indépendante du système 718
Affichage sans layout manager 721
Créer votre propre layout manager 723
Les éléments de l'interface utilisateur 726
Les boutons 727
Les événements de souris 734
Les menus 741
Les fenêtres internes 746
Le plaf (Pluggable Look And Feel) 752
Exemples de composants 760
Résumé 776

Chapitre 19 : Le graphisme 777

Les primitives graphiques 777
Les objets graphiques 778
Un composant pour les graphismes 778
Les différentes primitives 784
L'affichage du texte 799

Les polices de caractères ... 800
Les images ... 808
Obtenir une image ... 809
Surveiller le chargement d'une image ... 812
Conclusion ... 819

Chapitre 20 : Applets et réseaux ... 821

Les applets ... 821
Création d'une applet ... 822
Problèmes de compatibilité entre les versions ... 823
Avertissement ... 825
Deuxième avertissement ... 825
Fonctionnement d'une applet ... 826
Passer des paramètres à une applet ... 827
Agir sur le navigateur ... 832
Afficher un nouveau document ... 832
Afficher un message dans la barre d'état ... 835
Afficher des images ... 836
Les sons ... 837
Optimiser le chargement des applets à l'aide des fichiers d'archives ... 838
Les applets et la sécurité ... 839
Mettre en place les autorisations ... 840
Comment Java utilise les fichiers d'autorisations ... 844
Utiliser le security manager avec les applications ... 845
Accéder à un url à partir d'une application ... 845
Conclusion ... 849

Annexe A : Les classes String et StringBuffer ... 851

L'opérateur + ... 851

Les constructeurs de la classe String 852
Les méthodes de la classe String 855
La classe StringBuffer 859

Annexe B : Les mots réservés 861

Annexe C : Configuration d'UltraEdit 863

UltraEdit n'est pas gratuit ! 865
Installation et configuration d'UltraEdit 865

Index 871

Introduction

Java va avoir 4 ans ! C'est peu, dans l'absolu, mais c'est une éternité à l'échelle du développement de l'informatique. Qu'en est-il aujourd'hui des objectifs qui ont présidé à son lancement ?

Pourquoi Java ?

Java devait être un langage multi-plate-forme qui permettrait, selon l'adage[1] lancé par Sun Microsystems, son concepteur, d'écrire une fois pour toutes des applications capables de fonctionner dans tous les environnements. L'objectif était de taille, puisqu'il impliquait la définition d'une *machine virtuelle Java* (JVM) sur laquelle les programmes écrits devaient fonctionner, ainsi que la réalisation de cette machine virtuelle dans tous les environnements concernés. Sun Microsystems se chargeait par ailleurs de la réalisation d'une

1. "Write once, run everywhere" – Écrire une fois, utiliser partout.

machine virtuelle dans les environnements Unix et Windows, laissant à d'autres le soin d'en faire autant pour les autres environnements (et en particulier Mac OS, le système d'exploitation des Macintosh). Afin que le langage devienne un standard, Sun Microsystems promettait d'en publier les spécifications et de permettre à tous d'utiliser gratuitement son compilateur.

Dès l'annonce de Java, le monde de l'informatique se divisait en quatre camps : ceux qui pensaient, pour des raisons variées, que cela ne marcherait jamais, ceux qui pensaient, pour des raisons tout aussi variées, qu'il fallait absolument que ça marche, ceux qui ne voulaient absolument pas que cela marche, et ceux qui trouvaient qu'il était urgent d'attendre pour voir.

Ceux qui pensaient que ça ne marcherait jamais considéraient essentiellement qu'une machine virtuelle interprétant un langage intermédiaire (appelé *bytecode*) ne serait jamais suffisamment performante. Il faut dire que les exemples précédents avaient laissé des traces. D'autres considéraient qu'il serait impossible d'imposer un standard ouvert dans le monde des PC voué à l'hégémonie d'un seul éditeur détenant le quasi-monopole des systèmes d'exploitation pour PC. Java apparaissait effectivement comme une tentative de casser le monopole de Windows. En effet, celui-ci reposait (et repose encore !) sur un cercle vicieux : tous les PC (ou presque) doivent utiliser Windows parce que toutes les applications importantes sont développées pour cet environnement. Pour que cela change, il faudrait que les éditeurs portent leurs produits sous un autre environnement : cela ne serait pas du tout rentable, puisque très peu de PC utilisent un autre environnement. En revanche, une application développée en Java pourrait fonctionner (sans aucune modification, même pas une recompilation) dans n'importe quel environnement disposant d'une JVM.

Pour cette raison, un certain nombre d'éditeurs et d'utilisateurs voulaient absolument que cela fonctionne et imaginaient voir là la fin de la domination de Microsoft ! Et chacun de se mettre à rêver. Un éditeur important (Corel) annonça même qu'il allait réécrire toutes ses applications en Java.

La même raison entraînait Microsoft à prendre immédiatement le contre-pied de Java en proposant une technologie concurrente. Les concepteurs de Windows, en effet, arguaient qu'il était inefficace et inutile de développer des applications pour une machine virtuelle et qu'il suffisait de développer une interface commune pouvant être adaptée sous différents environnements. Ainsi, les développeurs n'étaient pas tenus d'utiliser un nouveau langage. Il

leur suffisait de programmer en fonction d'une nouvelle interface de programmation : ActiveX. Bien sûr, cette API ne serait ni ouverte (ce qui permettrait à certains de prétendre que Microsoft se réserve des fonctions pour son usage personnel afin d'avantager ses propres applications) ni gratuite.

La quatrième catégorie d'utilisateurs pensait probablement que les promesses de Java étaient tout à fait alléchantes, mais qu'on n'a jamais écrit des applications avec des promesses, et qu'il fallait donc rester à l'écoute pour voir ce que cela donnerait.

La version 1.0 de Java, bien qu'assez rassurante en ce qui concerne le problème, essentiel, des performances, confirmait certaines craintes dans le domaine de la richesse fonctionnelle. D'une part, si finalement la machine virtuelle se révélait suffisamment rapide pour la plupart des applications, elle restait cependant une solution inapplicable pour un certain nombre de traitements en temps réel. D'autre part, dans le but d'offrir une interface utilisateur unifiée pour tous les environnements, celle-ci était réduite au plus petit dénominateur commun, ce que de nombreux développeurs trouvaient notoirement insuffisant. Ajoutez à cela que certaines parties du langage, écrites par des équipes de programmation autonomes, présentaient un "look and feel" sensiblement différent du reste. Le tableau était loin d'être noir, mais il n'était pas franchement blanc non plus. Le côté positif était le ralliement généralisé au standard pour ce qui concerne les applications diffusées sur Internet. Les deux principaux navigateurs (Netscape Navigator 3 et Internet Explorer 3) disposaient en effet de JVM (machines virtuelles Java), même si celles-ci donnaient parfois l'impression d'un certain cafouillage. Il faut noter que, dans le cas d'Explorer, le ralliement au langage Java était tout à fait opportuniste, Microsoft continuant à promouvoir sa technologie concurrente, tout en essayant de détourner le standard Java en proposant une JVM étendant les spécifications de la norme, ce qui allait le conduire à une confrontation avec Sun Microsystems devant les tribunaux américains.

Avec l'apparition de la version 1.1 de Java, les choses se compliquaient quelque peu. En effet, cette version présentait de nombreuses améliorations. On pouvait même dire qu'il s'agissait de la première version réellement utile, même si certains aspects étaient encore inachevés (en particulier l'interface utilisateur toujours limitée). Cependant, le problème le plus grave était que les utilisateurs qui avaient fondé de réels espoirs sur l'adage "*Write once, run everywhere*" en étaient pour leurs frais. Pour profiter des améliorations, une grande partie des pro-

grammes devait être réécrite. L'inconvénient majeur n'était pas le travail de réécriture. Celui-ci était en effet assez limité, de nombreuses modifications n'ayant pour but que d'unifier la terminologie – par exemple, la méthode **reshape()**, permettant de modifier la taille et la position d'un composant, devenant **setBounds()**. Le réel problème était que les programmes conformes à la version 1.1 ne fonctionnaient pas avec les machines virtuelles Java 1.0.

Bien sûr, on peut arguer de ce que le langage étant encore jeune, Sun Microsystems devait corriger le tir au plus tôt, et en tout cas avant que les principaux systèmes d'exploitation soient livrés en standard avec une machine virtuelle. En effet, dans l'état actuel du marché, un éditeur souhaitant diffuser une application écrite en Java ne peut se reposer sur les JVM déjà installées. Il lui faut donc distribuer en même temps que son application la JVM correspondante. Dans le cas des PC et des machines fonctionnant sous Solaris, l'environnement Unix de Sun Microsystems, celle-ci est proposée gratuitement par cet éditeur sous la forme du JRE (*Java Runtime Environment*).

Le problème se complique dans le cas des autres environnements (cas du Macintosh, par exemple) et surtout dans le cas des applications distribuées sur Internet (les "applets"). Pour qu'une applet Java 1.0 puisse être utilisée par un navigateur, il faut que celui-ci possède une JVM, ce qui n'est pas toujours le cas, mais est tout de même assez fréquent. En revanche, pour qu'une applet Java 1.1 soit utilisable dans les mêmes conditions, il faut que la machine virtuelle Java soit compatible 1.1, ce qui est déjà plus rare. Au moment où ces lignes sont écrites, seuls Internet Explorer et HotJava disposent d'une telle JVM. Netscape Navigator dispose d'une JVM compatible 1.1 à partir de la version 4.05. Elle est cependant beaucoup moins compatible que celles des deux premiers cités.

Quant à Java 2, il n'existe simplement aucun navigateur capable de faire fonctionner les applets conformes à celle-ci. Cela est d'autant plus frustrant qu'elle apporte enfin une solution satisfaisante à certains problèmes sérieux, comme les limitations de l'interface utilisateur.

Pourquoi, alors, choisir de développer des applications en Java 2. D'une part, parce que cette version permet de résoudre de façon élégante et productive certains problèmes posés par les versions précédentes. D'autre part, parce que l'inconvénient cité précédemment ne concerne que les applets (diffusées sur Internet), et non les applications (qui peuvent être diffusées accompa-

gnées de la dernière version de la machine virtuelle Java). Enfin, parce que le langage Java a maintenant acquis une telle maturité et une telle notoriété qu'il ne fait nul doute que, dans un délai très court, tous les navigateurs seront équipés d'une machine virtuelle Java 2. En effet, ceux-ci sont conçus pour que la JVM puisse être mise à jour de façon simple par l'utilisateur, en téléchargeant la nouvelle version. (Ainsi, la version Macintosh d'Internet Explorer est livrée avec deux JVM différentes que l'utilisateur peut sélectionner.) Sun Microsystems propose d'ailleurs gratuitement un plug-in permettant à la plupart des navigateurs d'utiliser la machine virtuelle du JDK Java 2. Ce plug-in est inclus dans la version du JDK disponible sur le CD-ROM accompagnant ce livre.

Qu'est-ce que Java ?

Java a été développé dans le but d'augmenter la productivité des programmeurs. Pour cela, plusieurs axes ont été suivis.

Java est un langage orienté objets

Les premiers ordinateurs étaient programmés en langage machine, c'est-à-dire en utilisant directement les instructions comprises par le processeur. Ces instructions ne concernaient évidemment que les éléments du processeur : registres, opérations binaires sur le contenu des registres, manipulation du contenu d'adresses mémoire, branchement à des adresses mémoire, etc.

Le premier langage développé a été l'*assembleur*, traduisant d'une part mot à mot les instructions de la machine sous forme de *mnémoniques* plus facilement compréhensibles, et d'autre part masquant la complexité de certaines opérations en libérant le programmeur de l'obligation de décrire chaque opération dans le détail (par exemple, choisir explicitement un mode d'adressage).

L'assembleur, comme le langage machine, permet d'exprimer tous les problèmes qu'un ordinateur peut avoir à résoudre. La difficulté, pour le programmeur, réside dans ce que le programme (la "solution" du problème) doit être exprimé en termes des éléments du processeur (registres, adresses, etc.) plutôt qu'en termes des éléments du problème lui-même. Si vous avez à programmer un éditeur de texte, vous serez bien ennuyé, car l'assembleur ne connaît ni caractè-

res, ni mots, et encore moins les phrases ou les paragraphes. Il ne permet de manipuler que des suites plus ou moins longues de chiffres binaires.

Certains programmeurs pensèrent alors qu'il serait plus efficace d'exprimer les programmes en termes des éléments du problème à résoudre. Les premiers ordinateurs étaient utilisés uniquement pour les calculs numériques. Aussi, les premiers langages dits "de haut niveau" exprimaient tous les problèmes en termes de calcul numérique. En ce qui concerne le déroulement des programmes, ces langages reproduisaient purement et simplement les fonctions des processeurs : tests et branchements.

Est rapidement apparue la nécessité de mettre un frein à l'anarchie régnant au sein des programmes. Les données traitées par les ordinateurs étant en majorité numériques, c'est au déroulement des programmes que fut imposée une structuration forte. Ainsi, le déroulement des programmes devenait (en théorie) plus facilement lisible (et donc ceux-ci devenaient plus faciles à entretenir), mais les données était toujours essentiellement des nombres (mais de différents formats) et éventuellement des caractères, voire des chaînes de caractères, ou même des tableaux de nombres ou de chaînes, ou encore des fichiers (contenant, bien sûr, des nombres ou des chaînes de caractères). Le pionnier de ces langages dits "structurés" fut Pascal, que sa conception extrêmement rigide limitait essentiellement à l'enseignement. Le langage C, universellement employé par les programmeurs professionnels, marqua l'apogée de ce type de programmation.

Tous les problèmes se présentant à un programmeur ne concernent pas forcément des nombres ou des caractères uniquement. Cela paraît évident, mais cette évidence était probablement trop flagrante pour entrer dans le champ de vision des concepteurs de langages, jusqu'à l'apparition des premiers langages orientés objets. Ceux-ci, en commençant par le précurseur, Smalltalk, permettent d'exprimer la solution en termes des éléments du problème, plutôt qu'en termes des outils employés. Cela représente un énorme pas en avant pour la résolution de problèmes complexes, et donc pour la productivité des programmeurs. Bien sûr, toute médaille a son revers. En termes de performances pures, rien ne vaudra jamais un programme en assembleur. Toutefois, dans le cas de projets très complexes, il est probable que l'écriture et surtout la mise au point de programmes en assembleur, voire en langages "structurés", prendraient un temps tendant vers l'infini.

Le langage orienté objets le plus répandu est sans équivoque C++. Il s'agit d'une version de C adaptée au concept de la programmation par objets. Dans ce type de programmation, on ne manipule pas des fonctions et des procédures, mais des objets qui s'échangent des messages. Le principal avantage, outre le fait que l'on peut créer des objets de toutes natures représentant les véritables objets du problème à traiter, est que chaque objet puisse être mis au point séparément. En effet, une fois que l'on a défini le type de message auquel un objet doit répondre, celui-ci peut être considéré comme une boîte noire. Peu importe sa nature. Tout ce qu'on lui demande est de se comporter conformément au cahier des charges. Il devient ainsi extrêmement facile de mettre en œuvre un des concepts fondamentaux de la programmation efficace : d'abord écrire un programme de façon qu'il fonctionne. Ensuite, l'améliorer afin qu'il atteigne une rapidité suffisante.

Ce concept est particulièrement important, sinon le plus important de la programmation orientée objets. En effet, avec les langages des générations précédentes, cette approche, souvent utilisée, conduisait généralement à une solution médiocre, voire catastrophique. Il était d'usage de développer un prototype fonctionnel, c'est-à-dire un programme conçu rapidement dans le seul but de reproduire le fonctionnement souhaité pour le programme final. Les programmeurs les plus sérieux utilisaient pour cela un langage de prototypage. Une fois un prototype satisfaisant obtenu, ils réécrivaient le programme dans un langage plus performant. L'avantage était qu'ils avaient ainsi l'assurance de repartir sur des bases saines lors de l'écriture de la version finale. L'inconvénient était qu'il fallait faire deux fois le travail, puisque l'intégralité du programme était à réécrire. De plus, il était nécessaire de connaître deux langages différents, ce qui n'augmentait pas la productivité.

Une autre approche consistait à développer un prototype dans le même langage que la version finale, avec la ferme décision de réécrire ensuite, de façon "propre", les éléments qui auraient été mal conçus au départ. Satisfaisante en théorie, cette solution n'était pratiquement jamais réellement appliquée, la version finale étant presque toujours la version initiale agrémentée de nombreux replâtrages là où il aurait fallu une réécriture totale, celle-ci s'avérant impossible en raison des nombreuses connexions apparues en cours de développement entre des parties du code qui auraient dû rester indépendantes.

Les langages orientés objets, et en particulier Java, apportent une solution efficace et élégante à ce problème en définissant de manière très stricte la façon dont les objets communiquent entre eux. Il devient ainsi tout à fait possible de faire développer les parties d'une application par des équipes de programmeurs totalement indépendantes. C'est d'ailleurs le cas pour le langage Java lui-même, comme on peut s'en apercevoir en examinant les noms choisis pour les méthodes des différents objets du langage. Certains sont courts et obscurs, d'autres longs et explicites, en fonction de la personnalité du programmeur les ayant choisis. (Ces différences ont d'ailleurs tendance à disparaître dans les nouvelles versions du langage, marquant en cela une volonté d'unification appréciable.)

Java est extensible à l'infini

Java est écrit en Java. Idéalement, toutes les catégories d'objets (appelées *classes*) existant en Java devraient être définies par extension d'autres classes, en partant de la classe de base la plus générale : la classe *Object*. En réalité, cela n'est pas tout à fait possible et certaines classes doivent utiliser des *méthodes* (nom donné à ce que nous considérerons pour l'instant comme des procédures) natives, c'est-à-dire réalisées sous forme de sous-programmes écrits dans le langage correspondant à une machine ou un système donné.

Le but clairement annoncé des concepteurs du langage Java est de réduire les méthodes natives au strict minimum. On peut en voir un bon exemple avec les composants *Swing*, intégrés à la Java 2. Ceux-ci remplacent les composants d'interface utilisateur (boutons, listes déroulantes, menus, etc.) présents dans les versions précédentes et qui faisaient appel à des méthodes natives. Les composants Swing sont intégralement écrits en Java, ce qui simplifie la programmation et assure une portabilité maximale grâce à une indépendance totale par rapport au système. (Il est néanmoins tout à fait possible d'obtenir un aspect conforme à celui de l'interface du système utilisé. Ainsi, le même programme tournant sur une machine Unix, sur un PC et sur un Macintosh pourra, au choix du programmeur, disposer d'une interface strictement identique ou, au contraire, respecter le "look and feel" du système utilisé.)

Par ailleurs, Java est extensible à l'infini, sans aucune limitation. Pour étendre le langage, il suffit de développer de nouvelles classes. Ainsi, tous les composants écrits pour traiter un problème particulier peuvent être ajoutés au langage et utilisés pour résoudre de nouveaux problèmes comme s'il s'agissait d'objets standard.

Java est un langage à haute sécurité

Contrairement à C++, Java a été développé dans un souci de sécurité maximale. L'idée maîtresse est qu'un programme comportant des erreurs ne doit pas pouvoir être compilé. Ainsi, les erreurs ne risquent pas d'échapper au programmeur et de survivre aux procédures de tests. De la même façon, en détectant les erreurs à la source, on évite qu'elles se propagent en s'amplifiant.

Bien sûr, il n'est pas possible de détecter toutes les erreurs au moment de la compilation. Si on crée un tableau de données, il est impossible au compilateur de savoir si l'on ne va pas écrire dans le tableau avec un index supérieur à la valeur maximale. En C++, une telle situation conduit à l'écriture des données dans une zone de la mémoire n'appartenant pas au tableau, ce qui entraîne, au mieux, un dysfonctionnement du programme et, au pire (et le plus souvent), un plantage généralisé.

Avec Java, toute tentative d'écrire en dehors des limites d'un tableau ne conduit simplement qu'à l'exécution d'une procédure spéciale qui arrête le programme. Il n'y a ainsi aucun risque de plantage.

Un des problèmes les plus fréquents avec les programmes écrits en C++ est la "fuite de mémoire". Chaque objet créé utilise de la mémoire. Une fois qu'un objet n'est plus utilisé, cette mémoire doit être rendue au système, ce qui est de la responsabilité du programmeur. Contrairement au débordement d'un tableau, ou au débordement de la pile (autre problème évité par Java), la fuite de mémoire, si elle est de faible amplitude, n'empêche ni le programme ni le système de fonctionner. C'est pourquoi les programmeurs ne consacrent pas une énergie considérable à traquer ce genre de dysfonctionnement. C'est aussi pourquoi les ressources de certains systèmes diminuent au fur et à mesure de leur utilisation. Et voilà une des raisons pour lesquelles vous devez redémarrer votre système de temps en temps après un plantage. (C'est le cas de Windows. Ce n'est pas le cas d'autres systèmes qui se chargent de faire le ménage une fois que les applications sont terminées. Cependant, même dans ce cas, les fuites de mémoire demeurent un problème tant que les applications sont actives.)

Avec Java, vous n'aurez jamais à vous préoccuper de restituer la mémoire une fois la vie d'un objet terminée. Le *garbage collector* (littéralement "le ramasseur de déchets") s'en charge. Derrière ce nom barbare se cache en effet un programme qui, dès que les ressources mémoire descendent au-dessous d'un certain niveau, permet de récupérer la mémoire occupée par les objets qui ne

sont plus utiles. Un souci de moins pour le programmeur et surtout un risque de bug qui disparaît. Bien sûr, là encore, il y a un prix à payer. Vous ne savez jamais quand le garbage collector s'exécutera. Dans le cas de contrôle de processus en temps réel, cela peut poser un problème. (Il existe cependant d'ores et déjà des versions de Java dans lesquelles le fonctionnement du garbage collector est contrôlable, mais cela sort du cadre de ce livre.)

Java est un langage simple à apprendre

Java est un langage relativement simple à apprendre et à lire. Beaucoup plus, en tout cas, que C++. Cette simplicité a aussi un prix. Les concepts les plus complexes (et aussi les plus dangereux) de C++ ont été simplement éliminés. Est-ce à dire que Java est moins efficace ? Pas vraiment. Tout ce qui peut être fait en C++ (sauf certaines erreurs graves de programmation) peut l'être en Java. Il faut seulement parfois utiliser des outils différents, plus simples à manipuler, et pas forcément moins performants.

Contrairement à d'autres langages, la simplicité de Java est en fait directement liée à la simplicité du problème à résoudre. Ainsi, un problème de définition de l'interface utilisateur peut être un véritable casse-tête avec certains langages. Avec Java, créer des fenêtres, des boutons et des menus est toujours d'une simplicité extrême, ce qui laisse au programmeur la possibilité de se concentrer sur le véritable problème qu'il a à résoudre. Bien sûr, avec d'autres langages, il est possible d'utiliser des bibliothèques de fonctions ou de procédures pour traiter les problèmes d'intendance. L'inconvénient est double. Tout d'abord, ces bibliothèques ne sont pas forcément écrites dans le langage utilisé et, si elles le sont, elles ne sont généralement pas disponibles sous forme de sources, ce qui est facilement compréhensible, les programmeurs préférant préserver leurs produits des regards indiscrets. Il en résulte que les programmes utilisant ces bibliothèques ne sont pas portables dans un autre environnement. De plus, si elles sont disponibles dans les différents environnements pour lesquels vous développez, rien ne dit qu'elles le seront encore demain. Avec Java, pratiquement tout ce dont vous avez besoin est inclus dans le langage, ce qui, outre le problème de portabilité, résout également celui du coût. (Certaines bibliothèques coûtent fort cher.) De plus, si vous avez quand même besoin d'acquérir une bibliothèque d'objets, il vous suffit d'en sélectionner une conforme au label "*Pure Java*", ce qui vous assure qu'elle sera portable dans tous les environnements compatibles Java, présents ou à venir, et ce au niveau exécutable, ce qui garantit la pérennité de votre investissement.

Java est un langage compilé

Java est un langage compilé, c'est-à-dire qu'avant d'être exécuté, il doit être traduit dans le langage de la machine sur laquelle il doit fonctionner. Cependant, contrairement à de nombreux compilateurs, le compilateur Java traduit le code source dans le langage d'une machine virtuelle, appelée JVM *(Java Virtual Machine*). Le code produit, appelé *bytecode*, ne peut pas être exécuté directement par le processeur de votre ordinateur. (Cependant, rien n'interdit de fabriquer un processeur qui exécuterait directement le bytecode Java.) Le bytecode est ensuite confié à un *interpréteur*, qui le lit et l'exécute. En principe, l'interprétation du bytecode est un processus plus lent que l'exécution d'un programme compilé dans le langage du processeur. Cependant, dans la plupart des cas, la différence est relativement minime. Elle pose toutefois un problème pour les applications dont la vitesse d'exécution est un élément critique, et en particulier pour les applications "temps réel" nécessitant de nombreux calculs comme la simulation 3D animée. Il existe à cela plusieurs solutions

Les compilateurs natifs

Une des solutions consiste à utiliser un compilateur natif, traduisant directement le langage Java en un programme exécutable par le processeur de la machine. Le programme ainsi produit n'est plus portable dans un environnement différent. Cependant le programme source reste portable en théorie, du moment qu'il existe également un compilateur natif dans les autres environnements. Cette solution présente toutefois deux inconvénients.

Le premier est que, le code produit par le compilateur n'étant plus portable, il faut distribuer autant de versions différentes de l'application qu'il y a de systèmes cibles. En clair, si l'application doit tourner sur PC et sur Macintosh, il faudra que le support employé pour la distribution (par exemple un CD-ROM) contienne les deux versions.

Le deuxième inconvénient n'est pas inhérent à l'utilisation d'un compilateur natif, mais il est beaucoup plus grave. En effet, il est tentant pour le concepteur du compilateur, de se simplifier la tâche en demandant à l'utilisateur d'apporter quelques légères modifications à son programme. Dans ce cas, c'est tout simplement la portabilité des sources qui est remise en cause. Donc, avant de vous décider à choisir un compilateur natif, vérifiez s'il est compatible avec un programme "Pure Java" sans aucune modification.

Les compilateurs de bytecode

Une autre approche, pour améliorer la vitesse d'exécution des programmes Java, consiste à compiler le bytecode dans le langage du processeur de la machine cible. De cette façon, la portabilité est préservée jusqu'au niveau du bytecode, ce qui peut être suffisant dans bien des cas. Il reste alors à assurer la diffusion du code compilé en autant de versions qu'il y a de systèmes cibles. Cette approche est satisfaisante pour la diffusion d'applications vers des systèmes cibles connus. En particulier, la diffusion sur support physique (CD-ROM, par exemple) se prête bien à ce type de traitement. En effet, choisir un support physique implique généralement de connaître le système cible, car il faut s'être assuré que celui-ci est bien capable de lire le support physique. Ainsi, la diffusion d'une application pour PC et Macintosh implique d'utiliser un format de CD-ROM hybride pouvant être lu dans ces deux environnements. Si un compilateur de bytecode existe pour chacun d'eux, cela peut constituer une solution satisfaisante. Il peut s'agir d'un même compilateur, fonctionnant dans l'un des environnements, ou même dans un environnement totalement différent, et produisant du code natif pour l'environnement cible. On parle alors de *cross-compilation*. Un compilateur fonctionnant sur PC peut ainsi produire du code natif PC et Macintosh, au choix de l'utilisateur. Ce type de solution est cependant très rare, les cross-compilateurs servant le plus souvent à produire du code natif pour des machines ne disposant pas de système de développement propre (consoles de jeu, par exemple).

Les compilateurs JIT

La solution précédente présente un inconvénient majeur si l'ensemble des environnements cibles n'est pas connu. Par exemple, si vous créez une application devant être diffusée sur Internet, vous souhaiterez que celle-ci fonctionne sur tous les environnements compatibles Java, présents et à venir. La compilation du bytecode à la source (par le programmeur) n'est pas, dans ce cas, une solution envisageable. Les compilateurs JIT (*Just In Time*) sont supposés résoudre ce problème. Un compilateur JIT fonctionne sur la machine de l'utilisateur. Il compile le bytecode "à la volée", d'où l'appellation *Just In Time — Juste à Temps*. Généralement, l'utilisateur peut choisir d'activer ou non le compilateur JIT. Lorsque celui-ci est inactif, le programme est exécuté par l'interpréteur. Dans le cas contraire, le programme en bytecode est compilé dans le langage du processeur de la machine hôte, puis exécuté directement par celui-ci. Le gain de temps d'exécution est évidemment compensé en partie par le temps de com-

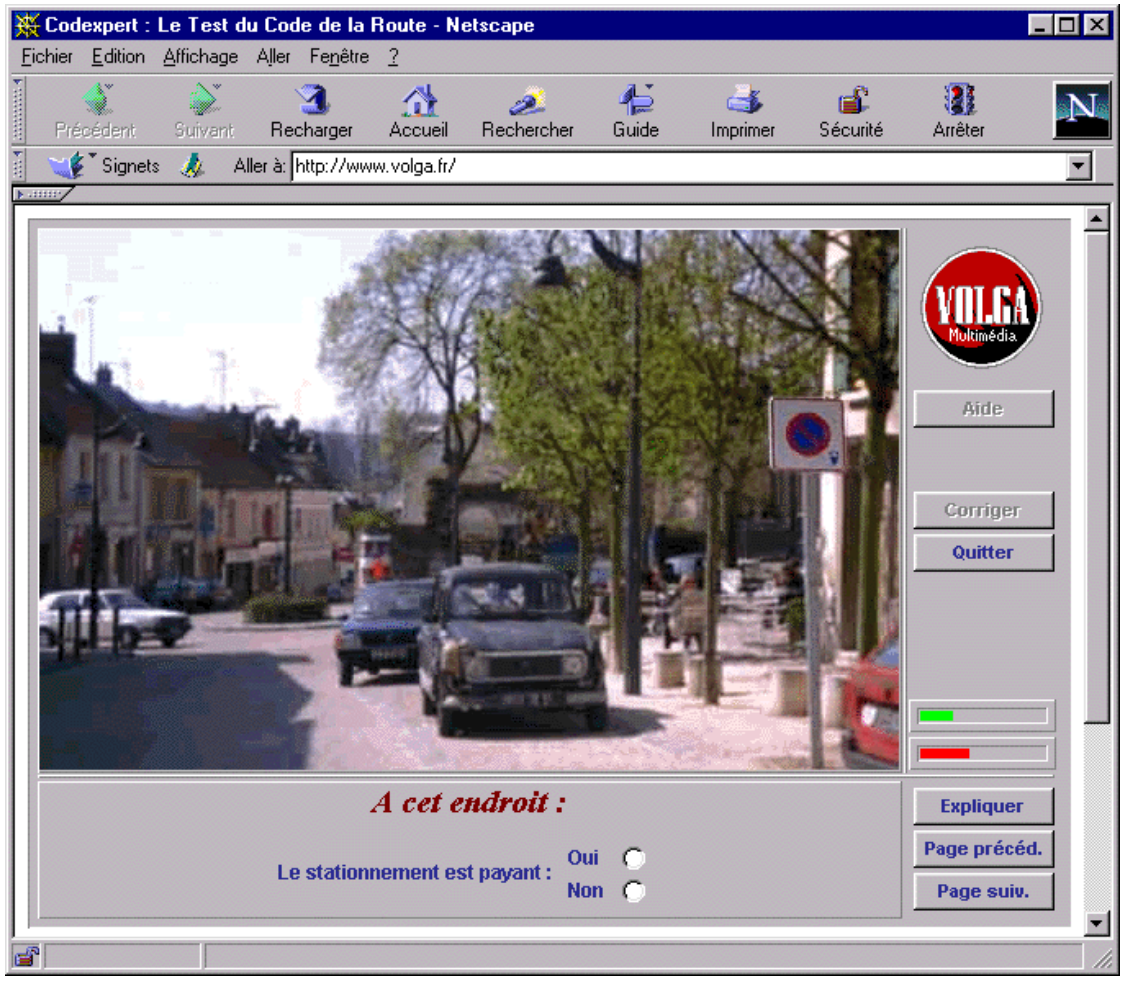
pilation. Pour un programme s'exécutant de façon totalement indépendante de l'utilisateur, le gain peut être nul. En revanche, pour un programme interactif, c'est-à-dire attendant les actions de l'utilisateur pour effectuer diverses opérations, le gain peut être substantiel, car le compilateur peut fonctionner pendant que l'utilisateur consulte l'affichage. Pour un gain maximal, il faut cependant que le programme soit conçu de telle façon qu'il soit chargé dans sa totalité dès le début de l'utilisation afin de pouvoir être compilé en *arrière-plan*. En effet, Java est conçu pour que les *classes* nécessaires au fonctionnement du programme soient chargées selon les besoins, afin que seules celles qui seront réellement utilisées soient transmises, cela, bien sûr, afin d'optimiser les temps de chargement sur Internet. En revanche, pour bénéficier au maximum des avantages d'un compilateur JIT, il faut autant que possible télécharger les classes à l'avance afin qu'elles soient compilées, et cela sans savoir si elles seront réellement utilisées. L'usage d'un compilateur JIT lorsque les classes sont téléchargées à la demande se traduit le plus souvent par un faible gain de vitesse, voire parfois un ralentissement de l'exécution.

Le premier langage du troisième millénaire ?

Java s'annonce comme une des évolutions majeures de la programmation. Pour la première fois, un langage efficace, performant, standard et facile à apprendre (et, de plus, gratuit) est disponible. Il satisfait aux besoins de l'immense majorité des développeurs et représente une opportunité de se libérer un peu de la tour de Babel des langages actuels (ainsi que de la domination de certains monopoles). Pour que Java soit un succès, il faut qu'il soit dès aujourd'hui appris comme premier langage de programmation par les programmeurs en cours de formation ; c'est dans cet esprit que ce livre a été conçu, non pas pour s'adresser aux programmeurs rompus aux secrets de C++, qui ont plutôt besoin d'un manuel de référence et d'une description des différences entre ces deux langages, mais à ceux qui veulent s'imprégner directement des concepts fondamentaux de Java. Tout comme on ne maîtrise pas une langue étrangère en traduisant sa pensée depuis sa langue maternelle, il n'est pas efficace d'essayer de "traduire" en Java des programmes élaborés dans un autre langage. Pour que Java devienne le premier langage du troisième millénaire, il faut qu'apparaisse une génération de programmeurs "pensant" directement en Java. Puissions-nous y contribuer modestement.

Voir Java à l'œuvre

Si vous souhaitez voir Java à l'œuvre en situation réelle, vous pouvez visiter le site de Volga Multimédia, à l'adresse **http://www.volga.fr/**. Vous y trouverez *Codexpert*, une applet proposant un test du Code de la Route simulant l'examen théorique du permis de conduire, qui vous montrera un exemple de ce que Java permet de réaliser. L'illustration ci-dessous reproduit l'affichage de *Codexpert* :



Chapitre 1

Installation de Java

Écrire des programmes en Java nécessite que vous disposiez d'un certain nombre d'outils. Bien entendu, il vous faut un ordinateur équipé d'un disque dur et d'une quantité de mémoire suffisante, ainsi que d'un système d'exploitation. Vous pouvez utiliser n'importe quel ordinateur, à condition que vous soyez en mesure de trouver le logiciel indispensable. Sun Microsystems diffuse gratuitement le compilateur et les autres outils nécessaires (mis à part l'éditeur) pour les environnements Windows et Solaris. En revanche, si vous utilisez un Macintosh, il vous faudra vous procurer les outils nécessaires auprès d'un autre fournisseur. Dans ce premier chapitre, nous supposerons que vous disposez d'un PC sous Windows 98. Les manipulations à effectuer sont cependant pratiquement identiques dans le cas des autres environnements. L'écriture des programmes, elle, est strictement identique. En revanche, la façon de les exécuter est différente sur un Macintosh, car son système d'exploitation (Mac OS) ne dispose pas d'un mode "ligne de commande". Vous devez donc utiliser un environnement de

programmation qui fournisse une fenêtre de commande dans laquelle vous pourrez taper les commandes nécessaires.

Sun Microsystems diffuse sur son site Web les dernières versions de ses logiciels. Les trois plus importants sont :

- Le JDK (*Java Development Kit*), qui contient *javac*, le compilateur qui transforme votre programme source en bytecode, *java*, l'interpréteur de bytecode, pour exécuter les applications, l'*AppletViewer*, pour exécuter les applets, *javadoc*, un programme permettant de créer automatiquement la documentation de vos programmes au format HTML, et d'autres utilitaires.

- La documentation, qui contient la liste de toutes les classes Java. Cette documentation est absolument indispensable au programmeur. Elle est au format HTML et doit être consultée à l'aide d'un navigateur.

- Le JRE (*Java Runtime Environment*), qui contient un interpréteur de bytecode et tout ce qui est nécessaire pour diffuser vos applications aux utilisateurs (y compris un compilateur JIT).

Pour être certain de disposer des dernières versions, vous pouvez vous connecter à l'adresse :

```
http://java.sun.com/
```

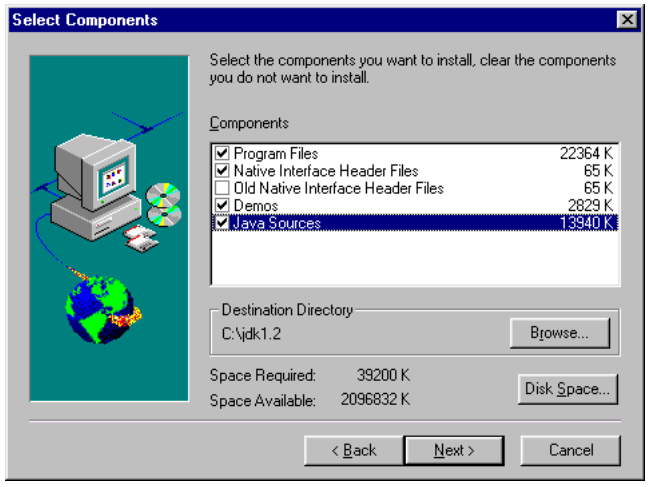
et télécharger ces trois éléments (ainsi que bien d'autres !). Cependant, étant donné le volume de données que représentent ces éléments (20 Mo pour le JDK, 16 Mo pour la documentation et 12 Mo pour le JRE), le téléchargement peut durer de nombreuses heures. Aussi, nous avons inclus le JDK, sa documentation et le JRE sur le CD-ROM accompagnant ce livre. Si vous souhaitez vérifier qu'il s'agit bien de la dernière version, connectez-vous à l'adresse ci-dessus. De toute façon, les versions fournies sur le CD-ROM sont suffisantes pour tous les exemples du livre. Attention : Le JDK et le JRE Java 2 portent encore le numéro de version 1.2. Cette nouvelle version a en effet été développé sous l'appellation Java 1.2, avant d'être renommée Java 2. En revanche, le JDK et le JRE n'ont pour l'instant pas encore été renommés. Lorsque nous parlerons du JDK 1.2, il s'agira bien du kit de développement Java 2.

Installation du JDK

Pour installer le JDK, exécutez le programme :

```
jdk12-win32.exe
```

se trouvant dans le dossier **Java2** sur le CD-ROM. Suivez les instructions affichées. Lorsque le programme vous demande quels sont les composants que vous souhaitez installer, cochez ceux qui vous intéressent, décochez les autres.



L'option **Old native interface header files** n'est absolument pas nécessaire. Les options **Demos** et **Java sources** sont très intéressantes si vous disposez d'un espace suffisant sur votre disque dur. (Les démos occupent près de 4 Mo et les sources plus de 16 Mo sous forme compressée.) Il est très instructif d'examiner les programmes de démonstration afin de voir comment les problèmes y sont résolus. Quant aux sources de Java (écrites en Java), leur étude est la meilleure façon de se familiariser avec les aspects les plus intimes du langage, mais cela demande toutefois d'avoir déjà de bonnes notions de base de ce langage !

Le programme d'installation place les fichiers dans le dossier **`jdk1.2`**, sur votre disque dur.

Le JDK est l'ensemble minimal que vous devez installer pour compiler et exécuter des programmes Java. Cependant, avant de pouvoir utiliser le compilateur, il est nécessaire de configurer l'environnement.

Ce que contient le JDK

A l'installation, le JDK est placé dans un dossier qui comporte plusieurs sous-dossiers. Dans la version acuelle, ce dossier est nommé **jdk1.2**. Vous trouverez ci-après une description succincte du contenu de chacun, avec quelques commentaires.

`jdk1.2\bin`

Ce dossier contient essentiellement les fichiers exécutables du JDK, c'est-à-dire :

`java.exe`

L'interpréteur de bytecode Java. C'est lui qui permettra d'exécuter les programmes que vous écrirez.

`javac.exe`

Le compilateur Java. Il permet de traduire vos programmes Java en bytecode exécutable par l'interpréteur.

`appletviewer.exe`

Ce programme permet de tester les applets Java, prévues pour être intégrées dans des pages HTML.

`jdb.exe`

Le débogueur Java. Il facilite la mise au point des programmes grâce à de nombreuses options permettant de surveiller leur exécution. Il est cependant beaucoup plus facile à utiliser lorsqu'il est intégré à un IDE (*Integrated Development Environment*).

`javap.exe`

Ce programme désassemble les fichiers compilés et permet donc d'examiner le bytecode.

`javadoc.exe`

Javadoc est un utilitaire capable de générer automatiquement la documentation de vos programmes. Attention, il ne s'agit pas de la documentation destinée aux utilisateurs finaux ! Ce programme est prévu pour documenter les bibliothèques de classes qui sont destinées à être utilisées par les programmeurs. L'ensemble de la documentation de Java a été réalisée à l'aide de cet utilitaire, qui produit un résultat sous forme de pages HTML.

`javah.exe`

Ce programme permet de lier des programmes Java avec des méthodes natives, écrites dans un autre langage et dépendant du système. (Ce type de programme n'est pas portable.)

`jar.exe`

Un utilitaire permettant de compresser les classes Java ainsi que tous les fichiers nécessaires à l'exécution d'un programme (graphiques, sons, etc.). Il permet en particulier d'optimiser le chargement des applets sur Internet.

`jarsigner.exe`

Un utilitaire permettant de signer les fichiers archives produits par **jar.exe**, ainsi que de vérifier les signatures. C'est un élément important de la sécurité, et également de l'efficacité. Nous verrons en effet que l'utilisation d'applets signées (dont on peut identifier l'auteur) permet de se débarrasser des contraintes et limitations très restrictives imposées à ce type de programme.

Le dossier **bin** contient encore une foule d'autres choses dont nous ne nous préoccuperons pas pour le moment.

jdk1.2/demo

Ce dossier comporte deux sous-dossiers :

applets

Le dossier **applets** contient un ensemble de programmes de démonstration sous forme d'applets. Chaque exemple comprend le fichier source (**.java**), le fichier compilé (**.class**) et un fichier HTML permettant de faire fonctionner l'applet dans un navigateur. Certains exemples comportent également des données telles que sons ou images.

Ces exemples sont intéressants pour avoir un aperçu des possibilités offertes par Java, en particulier dans le domaine graphique.

jfc

Les exemples figurant dans le dossier **jfc** concernent essentiellement les nouveaux composants Swing de l'interface utilisateur. Si vous voulez en avoir un catalogue complet, exécutez le programme **swingset**.

jdk1.2/docs

Ce dossier n'existe que si vous avez installé la documentation de Java (ce qui est absolument indispensable à tout programmeur sérieux). La quasi-totalité de la documentation se trouve dans le sous-dossier **api**.

jdk1.2/include

Ce dossier contient des fichiers header C qui ne nous concernent pas.

jdk1.2/lib

Ce dossier contient divers éléments utilitaires mais ne contient plus les classes standard Java comme dans les versions précédentes. Celles-ci ont été reportées dans un nouveau dossier nommé **jdk1.2/jre/lib**.

jdk1.2/jre

Ce dossier contient les éléments nécessaires à l'exécution des programmes. Il est également utilisé par le compilateur puisqu'il contient les classes standard Java.

jdk1.2/jre/bin

Dans ce sous-dossier se trouvent les exécutables utilisés par le JRE, et en particulier :

jre.exe

Un autre interpréteur de bytecode, semblable à **java.exe** et destiné aux utilisateurs de vos programmes. Il est dépourvu des options nécessaires à la mise au point des programmes.

jdk1.2/jre/lib

Ce dossier contient les éléments nécessaires au fonctionnement de l'interpréteur et du compilateur Java, et en particulier l'ensemble des classes de base compilées, contenues dans le fichier **rt.jar**. Vous ne devez surtout pas décompresser ce fichier qui est utilisé sous cette forme.

On y trouve également toutes sortes d'autres ressources nécessaires au fonctionnement de Java.

jdk1.2/src.jar

Ce fichier contient la quasi-intégralité des sources de Java, c'est-à-dire les programmes Java ayant servi à produire les classes publiques compilées figurant dans le fichier **rt.jar**. Il y manque toutefois les classes privées **java.*** et **sun.***. Vous ne pouvez donc pas les recompiler. (Pour ceux que cela aurait pu amuser !) Il est cependant très instructif, une fois que vous avez acquis une certaine connaissance du langage, d'étudier ces fichiers afin de comprendre comment les concepteurs du langage ont résolu les problèmes qui se posaient à

eux. Si vous voulez examiner ces sources, vous pouvez les décompresser à l'aide du programme **jar.exe** ou encore de Winzip.

Configuration de l'environnement

La configuration de l'environnement comporte deux aspects :

- Le chemin d'accès aux programmes exécutables.
- Le chemin d'accès aux classes Java.

Le chemin d'accès aux programmes exécutables

Pour configurer le chemin d'accès, ouvrez, à l'aide d'un éditeur de texte, le fichier **autoexec.bat** se trouvant dans la racine de votre disque dur. Ce fichier doit contenir quelques lignes semblables à celles-ci :

```
@IF ERRORLEVEL 1 PAUSE
@ECHO OFF
MODE CON CODEPAGE PREPARE=((850) C:\WINDOWS\COMMAND...
MODE CON CODEPAGE SELECT=850
KEYB FR,,C:\WINDOWS\COMMAND\KEYBOARD.SYS
SET BLASTER=A220 I5  D1 T4
SET PATH=C:\WINDOWS;C:\PROGRA~1\ULTRAEDT
```

Localisez la ligne commençant par **`SET PATH`** et ajoutez à la fin de celle-ci la commande suivante :

```
;C:\JDK1.2\BIN
```

Note 1 : Si la ligne se terminait déjà par un point-virgule, n'en ajoutez pas un second.

Note 2 : Vous pouvez taper en majuscules ou en minuscules, cela n'a pas d'importance.

Si votre fichier **autoexec.bat** ne comportait pas de ligne commençant par **SET PATH**, ajoutez simplement la ligne suivante à la fin du fichier :

```
SET PATH=C:\JDK1.2\BIN
```

La variable d'environnement **PATH** indique à Windows les chemins d'accès qu'il doit utiliser pour trouver les programmes exécutables. Les programmes exécutables du JDK se trouvent dans le sous-dossier **bin** du dossier dans lequel le JDK est installé. Ce dossier est normalement **c:\jdk1.2**. Si vous avez installé le JDK dans un autre dossier, vous devez évidemment modifier la ligne en conséquence.

Attention : Windows cherche les programmes exécutables tout d'abord dans le dossier à partir duquel la commande est tapée, puis dans les dossiers dont les chemins d'accès sont indiqués par la variable **PATH**, dans l'ordre où ils figurent sur la ligne **SET PATH**. Ainsi, si vous avez installé le JDK 1.1.5, votre fichier **autoexec.bat** contient (par exemple) la ligne :

```
SET PATH=C:\WINDOWS\;C:\JDK1.1.5\BIN
```

Si vous ajoutez le chemin d'accès au répertoire **bin** du JDK 1.2 de la façon suivante :

```
SET PATH=C:\WINDOWS\;C:\JDK1.1.5\BIN;C:\JDK1.2\BIN
```

le JDK 1.2 ne sera jamais exécuté car les fichiers exécutables portent les mêmes noms que dans la version 1.1.5. Le chemin d'accès à cette version se trouvant avant celui de la nouvelle version dans la ligne **SET PATH**, ce sont les anciennes versions qui seront exécutées.

Note : Pour que les modifications apportées au fichier autoexec.bat soient prises en compte, vous devez redémarrer Windows. Le plus simple est de redémarrer votre PC.

Le chemin d'accès aux classes Java

Le chemin d'accès aux classes Java doit être configuré exactement de la même façon à l'aide de la variable d'environnement **CLASSPATH**, en ajoutant au fichier **autoexec.bat** une ligne comme :

```
SET CLASSPATH=...
```

Cependant, vous n'avez pas besoin de configurer cette variable pour l'instant. En effet, nous n'utiliserons dans un premier temps que les classes standard de Java (en plus de celles que nous créerons directement dans nos programmes) et le compilateur saura les trouver !

Améliorer le confort d'utilisation

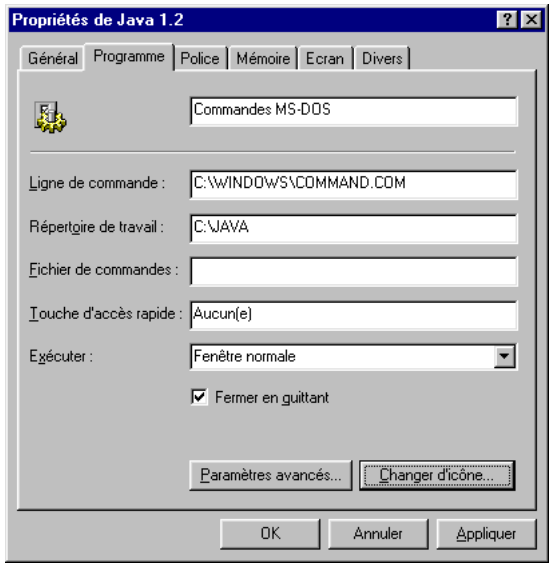
Pour utiliser le JDK, vous devrez ouvrir une fenêtre MS-DOS. Si vous utilisez la commande **Commandes MS-DOS** du sous-menu **Programmes** du menu **Démarrer** de la barre de tâches, la fenêtre s'ouvrira par défaut dans le dossier **c:\windows**, alors que vous préféreriez sûrement qu'elle s'ouvre dans un dossier spécifique. Par exemple, nous utiliserons le dossier **c:\java** pour y placer nos programmes. Pour simplifier le travail, vous pouvez créer sur votre bureau une icône qui ouvrira une fenêtre MS-DOS dans le dossier **c:\java**. Pour créer une telle icône, procédez de la façon suivante :

1. Ouvrez le dossier **c:\windows**, localisez le fichier **command.com** et faites-le glisser sur votre bureau. Windows ne déplace pas le fichier mais crée un raccourci :

2. Renommez ce raccourci pour lui donner un nom plus significatif, par exemple **Java 2**.

3. Cliquez à l'aide du bouton droit de la souris sur l'icône créée. Un menu contextuel est affiché. Sélectionnez l'option **Propriétés**, pour afficher la fenêtre des propriétés du raccourci.

4. Cliquez sur l'onglet **Programme** de la fenêtre affichée et modifiez l'option **Répertoire de travail** pour indiquer le dossier dans lequel vous allez placer vos programmes Java, par exemple **c:\java**. Vous pouvez également, si vous le souhaitez, modifier l'icône du raccourci en cliquant sur le bouton **Changer d'icône**.



5. Cliquez sur l'onglet **Police** et sélectionnez la police de caractères qui vous convient. La police idéale dépend de vos goûts et de la configuration de votre écran. Nous avons choisi la police **10 x 20**.

6. Refermez la fenêtre. Vous pourrez maintenant ouvrir une fenêtre MS-DOS directement dans le dossier **c:\java** en faisant simplement un double clic sur l'icône.

Le "Java Runtime Environment"

Le JDK contient tout ce qui est nécessaire pour développer des programmes Java. (Ou, plus exactement, presque tout le minimum nécessaire. D'une part il manque, en effet, un éditeur, et, d'autre part, on peut ajouter une multitude d'outils pour améliorer la productivité du programmeur.) En particulier, le JDK contient une machine virtuelle Java permettant d'exécuter les programmes. Cependant, les programmes que vous développerez ne seront pas forcément conçus pour n'être utilisés que par vous-même. Une phase importante de la vie d'un programme est ce que l'on appelle le *déploiement*, terme qui englobe toutes les façons dont un programme est diffusé vers ses utilisateurs. La façon traditionnelle de déployer une application consiste à faire une copie du code exécutable sur un support magnétique et à diffuser ce support vers les utilisateurs. Le support magnétique peut contenir simplement le code de l'application, que les utilisateurs devront exécuter depuis le support. C'est le cas, par exemple, de certains CD-ROM dits *écologiques*. Ce terme signifie que l'exécution du programme ne nécessite aucune modification du système de l'utilisateur. Dans la plupart des cas, cependant, l'application ne peut être utilisée qu'après une phase d'*installation*, qui inclut la copie de certains éléments sur le disque dur de l'utilisateur, et la modification du système de celui-ci. Dans le cas d'une application multimédia fonctionnant sous Windows, l'installation comporte généralement la création d'un dossier, la copie dans ce dossier d'une partie du code exécutable, la création dans ce dossier de fichiers de configuration, la copie dans le dossier système de Windows d'autres parties du code exécutable, la création d'un ou de plusieurs raccourcis permettant de lancer l'exécution à partir du bureau ou du menu **Démarrer** de la barre de tâches, la création de nouvelles entrées ou la modification d'entrées existantes dans la base de registres, etc. Les données utilisées par l'application peuvent également être copiées sur le disque dur, ou être utilisées depuis le CD-ROM, ce qui peut constituer un moyen (limité) de protection de l'application contre la copie.

La procédure qui vient d'être décrite concerne les applications dont le code est directement exécutable par le processeur de l'ordinateur de l'utilisateur. Dans le cas d'une application Java, ce n'est pas le cas, puisque le programme est compilé en *bytecode*, c'est-à-dire dans le langage d'un pseudo-processeur appelé JVM, ou *machine virtuelle Java*. Si le développeur sait que

chaque utilisateur dispose d'une machine équipée d'une JVM, nous sommes ramenés au cas précédent : celui-ci peut se présenter dans le cadre d'un déploiement interne, chez des utilisateurs appartenant tous à une même entreprise et disposant tous d'une machine dont la configuration est connue.

Cela peut être le cas, d'une certaine façon, lorsque l'application est diffusée sous forme d'*applet*, c'est-à-dire sous la forme d'un programme intégré à une page HTML diffusée, par exemple, sur Internet ou sur un réseau d'entreprise. Chaque utilisateur est alors supposé disposer d'un navigateur équipé d'une JVM. Si ce n'est pas le cas, c'est à lui de se débrouiller. En particulier, il lui revient de mettre à jour son navigateur afin qu'il soit équipé d'une JVM correspondant à la version de Java utilisée pour votre application. Ainsi, une JVM 1.0 ne pourra faire fonctionner les programmes écrits en Java 1.1 ou Java 2. (En revanche, les nouvelles JVM sont compatibles avec les anciens programmes.)

S'il s'agit d'une application devant être diffusée au public sur un CD-ROM, vous ne pouvez pas supposer que chaque utilisateur sera équipé d'une JVM. Cela viendra peut-être un jour, du moins on peut l'espérer. C'est seulement alors que Java sera devenu véritablement un langage universel. En attendant ce jour, il est nécessaire d'accompagner chaque application d'une machine virtuelle. C'est à cette fin que Sun Microsystems met à la disposition de tous, gratuitement, le JRE, ou *Java Runtime Environment*. Celui-ci contient tous les éléments nécessaires pour faire fonctionner une application Java, par opposition au JDK, qui contient tout ce qui est nécessaire pour développer une application. Le JRE peut être diffusé librement sans payer aucun droit à Sun Microsystems. Vous trouverez sur le site de Sun, à l'adresse :

```
http://java.sun.com/products/jdk/1.2/jre/download-windows.html
```

les versions Solaris et Windows du JRE. La version Windows est également présente sur le CD-ROM accompagnant ce livre.

Note : Il existe deux versions du JRE, avec ou sans support de l'internationalisation. La version "sans" est plus compacte mais ne convient que pour les applications utilisant exclusivement l'anglais. Vous préférerez proba-

blement la version internationale. C'est celle qui a été choisie pour figurer sur le CD-ROM accompagnant ce livre.

Bien sûr, si vous voulez que votre application soit également utilisable sur d'autres systèmes (Macintosh, par exemple), il faudra vous procurer l'équivalent pour ces systèmes. Pour trouver où vous procurer ces éléments, vous pouvez vous connecter à l'adresse :

```
http://java.sun.com/cgi-bin/java-ports.cgi
```

Installation du JRE Windows

L'installation du JRE est inutile pour le développement de programmes Java. Elle n'est pas non plus nécessaire pour le déploiement des applications. En effet, le JRE pour Windows est diffusé sous la forme d'un programme exécutable permettant de réaliser automatiquement son installation. Ce programme, nommé **jre12-win32.exe**, peut être appelé par votre programme d'installation. Vous pouvez également demander à l'utilisateur de le lancer lui-même. (Si vous diffusez plusieurs applications Java, l'utilisateur devra probablement avoir le choix d'installer ou non le JRE selon qu'il aura déjà installé ou non d'autres applications comportant cet élément.

Pourquoi, alors, installer le JRE ? Il y a deux raisons à cela. La première est qu'il est préférable de tester votre application avec le JRE avant de la diffuser. La deuxième est qu'il est possible de ne pas diffuser intégralement le JRE, mais seulement les parties qui sont indispensables pour l'exécution de votre application. La liste des fichiers indispensables figure dans le fichier **Readme** installé dans le même dossier que le JRE.

Si vous décidez d'installer vous-même le JRE comme une partie de votre application (sans utiliser le programme d'installation fourni), vous ne devez pas essayer de reproduire la procédure d'installation, mais installer le JRE dans un sous-dossier de votre application. En ne procédant pas de cette façon, vous risqueriez, chez certains utilisateurs, d'écraser une installation existante, ce qui pourrait empêcher le fonctionnement des applications Java précédemment installées. L'inconvénient de ne pas utiliser l'installation stan-

dard est que le disque de l'utilisateur se trouve encombré par autant d'exemplaires du JRE qu'il y a d'applications Java.

Sous Windows, si vous utilisez le programme d'installation fourni, le JRE est installé par défaut dans le dossier **c:\Program Files\ JavaSoft\Jre\1.2**. Le programme d'installation se charge d'effectuer les modifications nécessaires dans la base de registre de Windows. En revanche, aucune modification n'est apportée au fichier **autoexec.bat**. Vous pouvez toujours modifier la variable **PATH** dans votre fichier **autoexec.bat** pour y inclure le chemin d'accès au programme **jre.exe** (l'interpréteur de bytecode), mais cela n'est pas la façon normale de procéder. En effet, les applications sont rarement conçues pour être lancées depuis une fenêtre DOS. La façon la plus courante de procéder consiste à créer un raccourci (une icône) comme nous l'avons fait précédemment pour le lancement de **command.com**, en indiquant le chemin d'accès complet au programme ainsi que tous les paramètres nécessaires. Ce raccourci peut être créé par la procédure d'installation. Votre application est ainsi utilisable de façon habituelle, en faisant un double clic sur une icône.

Note : Le programme **jre.exe** n'utilise pas la variable d'environnement **CLASSPATH**. Si un chemin d'accès doit être indiqué pour les classes de votre application, cela doit être fait à l'aide d'une option de la ligne de commande. (Cela n'est pas un inconvénient, puisque la ligne de commande est cachée à l'utilisateur par l'utilisation du raccourci.)

Installation de la documentation en ligne

La documentation en ligne est un élément absolument indispensable pour le développeur d'applications Java. Celle-ci peut être consultée directement sur le site de Sun Microsystems, mais cela implique des heures de connexions Internet qui peuvent coûter fort cher. Il est beaucoup plus économique de télécharger cette documentation. Elle est disponible sous la forme d'un fichier compressé de type **zip** sur le site de Sun Microsystems.

Afin de vous éviter une longue attente, nous avons également inclus cette documentation sur le CD-ROM accompagnant ce livre. Vous la trouverez sous la forme d'un fichier nommé **jdk12-doc.zip**. Ce fichier doit être

décompacté à l'aide d'un utilitaire tel que **Winzip.exe**. (**Winzip.exe** est un programme en shareware, c'est-à-dire que vous pouvez l'utiliser gratuitement pour l'essayer. S'il convient à l'usage que vous voulez en faire et si vous décidez de le garder, vous devrez le payer. C'est un investissement indispensable ! Winzip est disponible partout sur Internet. La version française est disponible exclusivement à l'adresse :

```
http://www.absoft.fr/
```

Vous trouverez également une copie de ce programme sur le CD-ROM accompagnant ce livre.)

La documentation, une fois décompactée, doit être placée dans le sous-dossier **docs** du dossier d'installation du JDK. Le chemin d'accès complet est donc généralement :

```
c:\jdk12\docs
```

Pour consulter la documentation, vous devrez utiliser un navigateur. N'importe quel navigateur convient pour cet usage. Cependant, si vous développez des applets, vous aurez besoin d'un navigateur compatible avec Java 2. Autant faire d'une pierre deux coups en utilisant le même.

La plupart des navigateurs ne sont pas d'une ergonomie époustouflante lorsqu'il s'agit d'accéder à un fichier local. Pour simplifier l'accès à la documentation, il est donc conseillé d'utiliser une des possibilités offertes par les navigateurs. La solution la plus simple consiste à placer sur votre bureau un raccourci vers le document index de la documentation. Ce document est, en principe, contenu dans fichier **index.html** se trouvant dans le sous-dossier **docs** du dossier d'installation du JDK. Ce document comporte plusieurs liens vers divers documents accompagnant le JDK (fichier "readme", copyright, licence d'utilisation, etc.), ainsi que vers les applets de démonstration et vers plusieurs pages du site de Sun Microsystems consacré à Java. Le lien le plus important pointe vers la documentation de l'API. Il ouvre le document **index.html** du dossier **docs/api**. C'est le plus souvent à ce document que vous accéderez. Il est donc utile de créer un

raccourci pour celui-ci, ce que vous pouvez faire très simplement de la façon suivante :

1. Localisez le document **`index.html`** en ouvrant le dossier **`jdk1.2`** puis le dossier *docs* et enfin le dossier **`api`**.

2. Cliquez sur ce document avec le bouton droit de la souris et faites-le glisser sur votre bureau. Lorsque vous relâchez le bouton de la souris, un menu contextuel est affiché.

3. Sélectionnez l'option *Créer un ou des raccourci(s) ici*. Windows crée un raccourci sur votre bureau. Vous n'aurez qu'à faire un double clic sur cette icône pour ouvrir votre navigateur et accéder directement à la "table des matières" de la documentation.

Vous pouvez également ajouter ce document à la liste des *signets* ou des *favoris* (selon le navigateur que vous utilisez).

Un navigateur HTML compatible Java

Nous avons vu que vous aurez besoin d'un navigateur HTML pour consulter la documentation de Java, ainsi que pour vous connecter aux différents sites Web sur lesquels vous trouverez toutes les informations concernant les dernières évolutions du langage. N'importe quel navigateur peut convenir pour cet usage. En revanche, si vous souhaitez développer des applets (c'est-à-dire des applications pouvant être déployées sur Internet pour fonctionner à l'intérieur de pages HTML), vous devrez les tester dans un navigateur compatible Java. Lequel choisir ? La réponse est différente selon le public auquel s'adressent vos applets.

HotJava

Si vos applets sont destinées à un usage interne, par exemple au sein d'un réseau d'entreprise, vous avez peut-être la maîtrise (ou au moins voix au chapitre) du choix de l'équipement. Dans ce cas, vous devez choisir un navigateur qui vous simplifie la tâche. Si ce navigateur est destiné unique-

ment à afficher des pages HTML contenant des applets Java, et si tous les utilisateurs disposent de PC sous Windows ou d'ordinateurs Sun sous Solaris, le navigateur de Sun Microsystems, HotJava, est un bon candidat. Il est disponible gratuitement à l'adresse suivante :

```
http://java.sun.com/products/hotjava/
```

Il est également présent sur le CD-ROM accompagnant ce livre.

Le navigateur HotJava est entièrement écrit en Java. Il peut d'ailleurs être intégré à certaines applications. C'est celui qui vous garantira la compatibilité maximale avec le standard Java. En revanche, pour accéder à des sites Web sur Internet, il présente certains inconvénients. La plupart des sites Web sont testés sous Netscape Navigator et Internet Explorer. Si vous accédez à ces sites avec HotJava, attendez-vous à ce que le résultat affiché soit un peu différent de ce que les concepteurs avaient souhaité. Par ailleurs, HotJava n'est compatible ni avec les plug-ins Netscape, ni avec les composants ActiveX, ni avec le langage de script JavaScript (très utilisé sur les sites Web).

Netscape Navigator

Si vous souhaitez tester vos applets dans l'environnement le plus répandu, vous devez posséder une copie de Netscape Navigator. Ce navigateur est en effet le plus utilisé dans le monde. Cela ne devrait toutefois pas durer en raison des efforts acharnés de Microsoft. (En France, la situation est un peu différente.) Il est gratuit, et son code source est diffusé librement. De plus, il peut être mis à jour automatiquement en ligne pour s'adapter aux nouvelles versions de la machine virtuelle Java (JVM) de Sun. Il est compatible avec le JDK 1.1 depuis la version 4 (à condition, pour les versions 4.0 à 4.05, d'effectuer une mise à jour). Il peut être téléchargé gratuitement depuis le site de Netscape, à l'adresse :

```
http://www.netscape.com/download/
```

Malheureusement, Navigator comporte de nombreux inconvénients. En effet, et paradoxalement si l'on tient compte du fait que Netscape défend avec acharnement le standard Java aux côtés de Sun et d'IBM contre Microsoft, il est beaucoup moins compatible que son concurrent ! On peut citer quelques désagréments rencontrés par ses utilisateurs :

- Il ne permet d'utiliser qu'un seul fichier d'archive pour une applet, alors que la norme prévoit la possibilité d'en spécifier un nombre quelconque.
- Les fenêtres ouvertes par des applets ont une hauteur inférieure de 18 pixels par rapport à la hauteur spécifiée.
- Il restreint plus que nécessaire l'accès aux paramètres du système pour les applets (ce qui interdit de prendre en compte de façon dynamique le problème précédent).
- Sa JVM 1.1 ne permet pas l'exécution des programmes compatibles 1.1 mais compilés avec la version 1.2 (Java 2) du compilateur.
- Sa gestion de la mise en cache des classes est pour le moins spécifique, ce qui conduit à un ralentissement global des applets.

Nous voulons bien croire que Netscape soit le chevalier blanc qui lutte contre la volonté hégémonique de Microsoft, mais il faudrait tout de même commencer par balayer devant sa propre porte.

Internet Explorer

De nombreux utilisateurs, surtout en France, utilisent le navigateur Internet Explorer. Celui-ci est également gratuit, mais son code source n'est pas disponible. Le but à peine voilé de son éditeur est de lutter contre la diffusion du standard Java, dans lequel il voit un risque de contestation de son hégémonie sur le marché du logiciel pour PC. La stratégie adoptée consiste à diffuser gratuitement en très grosse quantité (par exemple, si possible, en l'intégrant à Windows) un produit qui étend le standard Java en proposant des fonctionnalités supplémentaires. Le but de la manœuvre est de rendre les utilisateurs dépendant de ces fonctions qui constituent des améliora-

tions, mais qui ne seraient en fait, d'après certains, que des pièges pour rendre les utilisateurs captifs. La justice américaine a d'ailleurs tranché, à la suite d'une action intentée par Sun Microsystems : Internet Explorer, pas plus que les autres produits du même éditeur, ne peut porter la marque ou le logo Java, à moins d'être mis en conformité avec le standard.

Il n'en reste pas moins que Internet Explorer 4 est beaucoup plus compatible avec le standard Java que Netscape Navigator ! Toutes les applets que nous avons pu écrire et tester avec les outils de Sun Microsystems ont toujours fonctionné sans problème avec le navigateur de Microsoft. En revanche, nous avons passé des nuits blanches à chercher pourquoi ces mêmes applets écrites en pur Java ne fonctionnaient pas avec Navigator. La solution a toujours été un "bidouillage" très peu satisfaisant et certainement pas conforme à l'esprit de Java. Dans ces conditions, qui dévoie le standard ?

Notez que Internet Explorer peut être modifié en remplaçant la machine virtuelle Java d'origine par une autre 100 % conforme au standard[1] .

Un éditeur pour la programmation

Un outil important absolument nécessaire pour développer des programmes en Java, comme dans la plupart des autres langages, est l'*éditeur de texte*. Dans le cas de Java, vous pouvez utiliser n'importe quel éditeur. Le bloc-notes de Windows peut tout à fait convenir, même si ce n'est vraiment pas l'idéal, tant ses fonctions sont limitées. Vous pouvez également employer un programme de traitement de texte, pour autant qu'il dispose d'une

1. Ne confondez pas *conforme* et *compatible*. Internet Explorer n'est pas conforme au standard car il y ajoute un certain nombre d'extensions. Pour autant, il n'en est pas moins compatible. Il est même, à l'heure actuelle, plus compatible que Netscape Navigator. Le problème vient de ce que, pour le développeur, une JVM doit être capable de faire tourner *tous* les programmes conformes au standard, mais *rien d'autre* que ceux-ci. Sinon, le programmeur peut, presque à son insu, utiliser des fonctions qui seront incompatibles avec d'autres navigateurs. A l'heure actuelle, la seule solution consiste à tester les applets avec tous les navigateurs présents sur le marché, ce qui n'est pas une mince affaire.

fonction permettant d'enregistrer en mode *texte seulement*, c'est-à-dire sans les enrichissements concernant les attributs de mise en page. Là aussi, c'est loin d'être la solution idéale, étant donné la lourdeur de ce type d'application.

Ce que doit offrir un éditeur

La meilleure solution consiste donc à utiliser un éditeur de texte, plus sophistiqué que le bloc-notes, mais débarrassé de la plupart des fonctions de mise en page d'un traitement de texte. En fait, la fonction la plus importante d'un éditeur de texte destiné au développement de programmes est la capacité de numéroter les lignes. En effet, les programmes que vous écrirez fonctionneront rarement du premier coup. Selon la philosophie de Java, un nombre maximal d'erreurs seront détectées par le compilateur, qui affichera une série de messages indiquant, pour chaque erreur, sa nature et le numéro de la ligne où elle a été trouvée. Si votre éditeur n'affiche pas les numéros de lignes, vous risquez d'avoir beaucoup de mal à vous y retrouver.

Il existe de nombreux éditeurs offrant des fonctions plus ou moins sophistiquées d'aide au développement de programmes. L'une d'elles consiste à afficher les mots clés du langage dans une couleur différente, ce qui facilite la lecture du code et aide à détecter les erreurs. En effet, si vous tapez un mot clé, supposé s'afficher, par exemple, en rouge, et que celui-ci reste affiché en noir, vous saurez immédiatement que vous avez fait une faute de frappe.

UltraEdit

Un des éditeurs les plus simples à utiliser et les plus performants est UltraEdit, développé par la société IDM. Vous pouvez télécharger la dernière version disponible à l'adresse :

```
http://www.idmcomp.com
```

Il s'agit d'un produit en shareware, que vous pouvez donc tester gratuitement pendant 30 jours. Si vous décidez de le conserver au-delà, vous de-

vrez le payer. Le prix en est très raisonnable et vous pouvez payer en ligne sur le site d'IDM.

Vous trouverez une copie de cet éditeur sur le CD-ROM accompagnant ce livre. Il est accompagné d'un dictionnaire Java, nécessaire pour l'affichage en couleur des mots clés du langage. D'autres dictionnaires (pour d'autres langages de programmation) sont disponibles sur le site d'IDM.

Quelques petits fichiers batch qui vous simplifieront la vie

Si vous n'installez pas *UltraEdit*, vous pouvez toutefois vous simplifier la vie en créant trois petits fichiers batch qui faciliteront la compilation et l'exécution de vos programmes[1] . Le premier sera appelé **ca.bat** (pour **c**ompiler **a**pplication ou **c**ompiler **a**pplet) et contiendra la ligne suivante :

```
javac %1.java
```

Créez ce fichier et enregistrez-le dans le dossier dans lequel vous allez stocker vos programmes (**c:\java**, par exemple). Ainsi, vous pourrez compiler un programme en tapant simplement :

```
ca prog
```

au lieu de :

```
javac prog.java
```

Ce n'est pas grand-chose, mais lorsque vous aurez compilé 20 fois le même programme, vous apprécierez l'économie.

De la même façon, créez le fichier **va.bat** contenant la ligne :

---

1. Si vous décidez d'installer UltraEdit, vous vous simplifierez encore beaucoup plus la vie, comme vous pouvez le constater en lisant l'Annexe C. Attention, cependant. Si vous l'essayez, vous ne pourrez plus vous en passer !

```
appletviewer %1.java
```

ainsi que le fichier **ra.bat** contenant la ligne :

```
java %1
```

Vous pourrez ainsi lancer une application en tapant :

```
ra prog
```

(l'économie, ici, n'est que de deux caractères, car il n'est pas nécessaire de taper l'extension **.class**) et visualiser une applet dans l'AppletViewer en tapant :

```
va prog
```

au lieu de :

```
appletviewer prog.java
```

Si vous n'êtes pas convaincu de l'intérêt, tapez 50 fois le mot **appletviewer** et on en reparle juste après... OK ?

Note : Vous êtes peut-être surpris de voir l'utilisation du programme AppletViewer, supposé afficher une page HTML, avec un fichier source Java. Cela est simplement dû au fait que nous utiliserons une petite astuce, comme nous le verrons au chapitre suivant, pour éviter d'avoir à créer un fichier HTML pour chacune des applets que nous testerons.

Un environnement de développement

Développer des applications à l'aide d'un simple éditeur de texte et d'un compilateur, cela peut paraître complètement dépassé à l'heure du RAD

(*Rapid Application Development* - Développement rapide d'applications). Il existe de nombreux outils pour améliorer la productivité des programmeurs. C'est le cas, en particulier, des environnements de développement, encore appelés *IDE* (*Integrated Development Environment*).

Avantages

Un environnement de développement peut améliorer la productivité de deux façons totalement différentes. La première consiste à aider le programmeur dans les tâches "ménagères", par exemple en numérotant les lignes. L'aide fournie peut aller beaucoup plus loin. Ainsi, il peut être possible de lancer la compilation depuis l'éditeur, et d'afficher le résultat dans une fenêtre. Cela constitue un énorme avantage car la logique impérative du développement de programme consiste à corriger les erreurs une par une (pas plus d'une à la fois) en commençant par la première. En effet, dans de nombreux cas, une première erreur de syntaxe perturbe tellement le compilateur qu'il détecte toutes sortes d'autres erreurs qui ne sont que le fruit de son imagination. (Pourquoi un compilateur ne pourrait-il pas avoir de l'imagination ?)

Par exemple, si vous oubliez un point-virgule à la fin d'une ligne, le compilateur "pense" que l'instruction en cours n'est pas terminée. Il lit donc la ligne suivante comme s'il s'agissait de la suite de cette instruction et trouve généralement une erreur. Puis il continue avec la suite de la ligne, et trouve alors une autre erreur, puisque le premier mot de la ligne a été "absorbé" par la première erreur. L'erreur se propage ainsi jusqu'à la fin de la ligne. Si vous oubliez une fin de bloc, l'erreur peut se propager très loin dans le programme. La règle (du moins jusqu'à ce que vous ayez acquis une bonne expérience de l'utilisation du compilateur Java) est donc de ne corriger qu'une seule erreur à la fois. Malheureusement, si votre programme comporte plusieurs erreurs (et nous venons de voir que c'est très souvent le cas), celles-ci sont affichées les unes derrière les autres en faisant défiler l'affichage. Les fenêtres DOS n'étant pas extensibles (elles ne comportent que 25 lignes), si le texte affiché défile au-delà du haut de la fenêtre, vous ne pouvez plus y accéder et vous ne pouvez donc plus prendre connaissance de la première erreur !

Une solution à ce problème consiste à rediriger la sortie générée par le compilateur vers un fichier, puis à ouvrir ce fichier à l'aide de l'éditeur de texte. Malheureusement, cela n'est pas possible. En effet, pour une raison inconnue, mais vraiment étrange, le compilateur de Sun n'utilise pas, pour l'affichage, la sortie standard (standard output), mais la sortie d'erreurs du DOS (standard error). Cela ne serait pas grave sous Unix, mais il se trouve que, sous DOS, *standard error* ne peut être redirigé !

Un environnement de développement présente donc le grand avantage d'afficher les erreurs dans une fenêtre que vous pouvez faire défiler.

D'autres fonctions plus ou moins utiles sont la faculté de lancer la compilation du programme en cours (ou son exécution) en cliquant sur un bouton, la capacité de l'éditeur à détecter les erreurs de syntaxe, ou encore la possibilité d'accéder à la documentation concernant un mot clé en sélectionnant celui-ci dans votre programme.

Mais l'intérêt des systèmes de développement va beaucoup plus loin, grâce à l'utilisation des *JavaBeans*. Ces "grains de café" (au fait, pour ceux qui ne le savaient pas, Java veut dire "café", d'où le logo représentant une tasse fumante !) sont des composants Java de nature un peu spéciale. Leur particularité est qu'ils respectent une norme permettant à un programme d'analyser leur contenu à partir de la version compilée. Il est ainsi possible aux environnements de développement de les intégrer aux programmes que vous écrivez. Au lieu de programmer un bouton pour une interface, vous pouvez alors simplement utiliser le JavaBean correspondant en sélectionnant l'icône d'un bouton dans une barre d'outils et en la faisant glisser sur la fenêtre représentant l'interface de votre programme. Le développement peut être ainsi considérablement accéléré. (Vous pouvez, bien sûr, développer vos propres JavaBeans.)

Un autre avantage certain des environnements de développement est l'intégration des fonctions de débogage qui, sans eux, sont disponibles sous la forme du débogueur livré avec le JDK.

Inconvénients

Les systèmes de développement présentent toutefois certains inconvénients. Tout d'abord, ils sont payants. Par ailleurs, vous subissez les inconvénients liés aux avantages : utiliser un bouton sous forme de JavaBean est certainement plus rapide que de le programmer soi-même, encore que la programmation d'un bouton en Java soit extrêmement simple. Mais si vous souhaitez un bouton d'un type particulier, vous ne serez guère avancé. Par ailleurs, pouvoir utiliser des boutons, ou tous autres composants, sans savoir comment ils fonctionnent n'est probablement pas votre but si vous lisez ce livre.

Toutefois, les avantages des systèmes de développement en termes d'ergonomie sont tout à fait indéniables. A vous de voir si cela justifie l'investissement dans un premier temps. A terme, une fois que vous aurez appris à programmer en Java, il ne fait nul doute que vous utiliserez un tel environnement. Cependant, vous aurez appris à écrire du code de façon pure et dure, et vous serez mieux à même de choisir l'environnement qui vous convient. Sachez qu'il est parfaitement possible d'écrire des applications Java sans ce type d'outils si vous connaissez le langage. En revanche, et contrairement à ce que certains voudraient faire croire, il est parfaitement impossible de développer des applications sérieuses avec ce type d'outils si vous n'avez pas une bonne connaissance du langage.

Quel environnement ?

Le choix d'un environnement est une affaire de goût. Il en existe de nombreux, plus ou moins sophistiqués, tels que *Java Studio* (pour l'assemblage de composants JavaBeans) et *Java WorkShop* (pour l'écriture de programmes) de Sun Microsystems, *JBuilder*, de *InPrise* (ex *Borland*), *Visual Café*, de *Symantec*, ou *PowerJ*, de *Sybase*, dont la version *Learning Edition*, disponible en téléchargement sur Internet, est gratuite. Tous ces produits utilisent le compilateur, l'interpréteur de bytecode, le débogueur et la documentation du JDK. Certains ajoutent leur propre compilateur JIT. (Il faut noter que le compilateur JIT livré avec la version Windows du JDK est tout simplement celui de Symantec.)

Un environnement mérite une mention séparée : il s'agit de *Visual J++*. Comme d'habitude, la volonté de son éditeur n'est pas de supporter le stan-

dard Java, mais au contraire de lutter contre lui en le dévoyant. Ainsi, ce système n'est pas compatible avec l'utilisation des JavaBeans, concurrents directs des technologies *ActiveX* et *COM*. De plus, il mélange allègrement la technologie Java avec des technologies propriétaires liées à Windows. Ainsi, les applications créées ne sont plus portables. Le but est clairement de générer une base d'applications suffisantes utilisant les technologies propriétaires pour inciter les éditeurs d'autres systèmes à les implémenter. Si vous êtes intéressé par Java pour les avantages que peut apporter une portabilité totale des programmes au niveau exécutable sur tous les systèmes, fuyez donc autant que possible ce type de produit. (Pour être tout à fait honnête, *Visual J++* peut fonctionner dans un mode 100 % compatible Java. Dans ce cas, il ne présente aucun avantage particulier par rapport à ses concurrents.)

Appréciation personnelle : Le choix d'un environnement est quelque chose de tout à fait personnel. Notre avis est que le plus simple est souvent le plus efficace. Nous avons choisi pour tous nos développements d'utiliser UltraEdit en créant simplement trois commandes personnalisées pour compiler et exécuter un programme, ou charger l'AppletViewer. Nous vous conseillons fortement d'en faire autant. Pour plus de détails, reportez-vous à l'Annexe C.

Où trouver les informations ?

Java est un langage jeune. (Il aura quatre ans le 23 mai 1999.) Il y a autour de ce langage un perpétuel bouillonnement de nouveautés. Il est donc particulièrement important de vous tenir au courant des nouvelles. Pour cela, il existe plusieurs sources privilégiées.

Le site Web de Sun

La première source officielle est le site de Sun Microsystems dédié à Java. Vous le trouverez à l'adresse :

```
http://java.sun.com
```

Il est libre d'accès et comporte toutes les informations généralistes concernant le langage, et en particulier toutes les annonces de nouveaux produits et développements.

Le JDC

Ces initiales signifient *Java Developer Connection* et désignent un site encore plus important pour les programmeurs. Il est également proposé par Sun Microsystems et concerne cette fois les aspects techniques du langage. Son adresse est :

```
http://developer.javasoft.com
```

C'est ici, en particulier, que sont proposées en téléchargement les différentes versions du JDK. Ce site est accessible uniquement aux utilisateurs enregistrés, mais l'enregistrement est gratuit. Vous y trouverez de très nombreux articles sur Java et les sujets annexes. Vous pourrez également vous abonner au *Java(sm) Developer Connection(sm) Tech Tips*, qui est une lettre mensuelle expédiée par courrier électronique et concernant diverses astuces de programmation en Java. Une visite de ce site est *absolument indispensable* à toute personne désirant programmer en Java. Il donne entre autres une liste des ressources disponibles concernant Java, à l'adresse :

```
http://developer.java.sun.com/developer/techDocs/resources.html
```

Les autres sites Web dédiés à Java

Il existe de nombreux sites consacrés à la programmation en Java. Vous y trouverez de tout, et en particulier toutes sortes d'applets prêtes à l'emploi. Pour trouver ces sites, vous pouvez utiliser un moteur de recherche comme Yahoo.

Parmi les sites les plus intéressants, on peut citer le *Java Developers Journal*, à l'adresse :

```
http://www.sys-con.com/java/
```

qui contient de nombreux articles, plutôt généralistes, et surtout de très nombreux liens vers les sociétés éditant des produits relatifs à Java. Le principal reproche que l'on puisse faire à ce site est qu'il est littéralement bourré d'images animées, ce qui rend le chargement particulièrement long et pénible.

Le site *Java Boutique*, à l'adresse :

```
http://javaboutique.internet.com/
```

offre la particularité de proposer en téléchargement 16 IDE. Vous y trouverez également une liste de liens vers les sites des éditeurs de tous les IDE existants (41 au moment où ces lignes sont écrites).

Le site *JavaWorld* est également un des plus intéressants pour ses articles techniques. Vous le trouverez à l'adresse :

```
http://www.javaworld.com/
```

Enfin, nous vous conseillons de visiter également le site :

```
http://sunsite.unc.edu/javafaq.html
```

Les Newsgroups

Plus de 40 groupes de nouvelles sont consacrés au langage Java, certains étant dédiés à des IDE d'éditeurs tels que *InPrise* (*Borland*). La plupart sont en anglais. Il existe cependant un groupe de nouvelles en français dont le nom est :

```
fr.comp.java
```

Sur ces groupes de nouvelles, les utilisateurs peuvent donner leur avis ou poser des questions concernant Java. La qualité des réponses est variable. N'oubliez pas, avant de poser une question, de vérifier s'il n'a pas été répondu quelques jours auparavant à une question similaire.

Chapitre 2

Votre premier programme

Dans ce chapitre, nous écrirons un premier programme en Java. (Nous en écrirons en fait plusieurs versions.) Nous aurons ainsi l'occasion de tester l'installation du JDK et de nous familiariser avec l'écriture du code Java. Il est possible que vous ne compreniez pas immédiatement toutes les subtilités des éléments du langage que nous emploierons. C'est normal. Nous y reviendrons en détail au cours des chapitres suivants. Pour l'instant, il ne s'agit que d'une séance d'imprégnation. Laissez-vous porter, même si vous vous sentez un peu désorienté au début. Tout cela s'éclaircira en temps utile.

Première version : un programme minimal

Démarrez votre éditeur de texte, si cela n'est pas déjà fait, et saisissez le programme suivant, en respectant scrupuleusement la mise en forme :

```
// Mon premier programme en Java
class PremierProgramme {
   public static void main(String[] argv) {
      System.out.println("Ca marche !");
   }
}
```

Enregistrez ce programme sous le nom **PremierProgramme.java**, dans le dossier que vous avez créé spécialement pour cet usage au chapitre précédent (normalement, **c:\java**).

Note : Si vous utilisez l'éditeur UltraEdit, vous verrez certains mots s'afficher en couleur au moment où vous enregistrerez le programme. En effet, UltraEdit se sert de l'extension du fichier (**.java**) pour déterminer son contenu et utiliser le dictionnaire adéquat. L'utilisation des couleurs vous facilite la lecture et vous permet d'identifier immédiatement certaines fautes de frappe. Ainsi, normalement, la première ligne est affichée en vert pâle, les mots **class**, **public**, **static** et **void** en bleu, et le mot **String** en rouge.

Ouvrez maintenant une fenêtre DOS, par exemple en faisant un double clic sur l'icône du raccourci que nous avons créé au chapitre précédent. L'indicatif affiché doit correspondre au chemin d'accès du dossier contenant votre programme (**c:\java**). Si ce n'est pas le cas, changez de dossier en utilisant les commandes du DOS (**cd..** pour remonter d'un niveau dans l'arborescence des dossiers, et **cd <*nom_de_dossier*>** pour ouvrir le dossier choisi. Vous pouvez également utiliser la commande **cd** avec le chemin d'accès complet du dossier à atteindre, par exemple **cd c:\java**.)

Une fois dans le dossier contenant votre programme, tapez la commande :

```
javac PremierProgramme.java
```

puis la touche *Entrée*, ce qui a pour effet de lancer la compilation de votre programme. Si vous avez créé les fichiers batch décrits au chapitre précédent, vous pouvez également taper :

```
ca PremierProgramme
```

Normalement, votre PC doit s'activer pendant un certain temps, puis vous rendre la main en affichant de nouveau l'indicatif du DOS.

Si vous obtenez le message :

```
Commande ou nom de fichier incorrect
```

c'est soit que vous avez fait une faute de frappe, dans le mot **javac**, soit que votre variable d'environnement n'est pas correctement configurée, soit que le JDK n'est pas correctement installé. Commencez par vérifier si vous n'avez pas fait de faute de frappe. La commande **javac** doit être séparée du reste de la ligne par un espace. Si vous avez utilisé le fichier batch, essayez sans. (La faute de frappe est peut-être dans ce fichier.)

Si vous n'avez pas fait de faute et que vous obtenez toujours le même message, vérifiez si la variable **PATH** est correctement configurée en tapant la commande :

```
path
```

(puis la touche *Entrée*). Vous devez obtenir le résultat suivant :

```
PATH=C:\JDK1.2\BIN
```

La ligne peut comporter d'autres chemins d'accès, mais celui-ci doit y figurer, éventuellement séparé des autres par des points-virgules. Si le chemin d'accès n'est pas correct, ou si vous avez un doute, tapez la commande :

```
PATH C:\JDK1.2\BIN
```

Cette commande est identique au résultat que vous auriez dû obtenir précédemment, à la différence que le signe égal est remplacé par un espace. Essayez ensuite de compiler de nouveau le programme.

Note : La commande **PATH** utilisée ainsi modifie la variable **PATH** pour la vie de la fenêtre DOS. Si vous fermez cette fenêtre et en ouvrez une autre, il faudra recommencer. Vous devrez donc, de toutes les façons, corriger votre fichier **autoexec.bat**.

Si vous obtenez toujours le même message, c'est que le programme **javac.exe** n'est pas installé dans le dossier **c:\jdk1.2\bin**. Modifiez alors le chemin d'accès en conséquence ou procédez à une nouvelle installation.

Si vous obtenez le message :

```
Error: Can't read: PremierPrograme.java
```

vérifiez si vous n'avez pas fait une faute de frappe dans le nom du fichier (comme ici, où nous avons volontairement omis un *m*). Si le nom du fichier vous semble correct, c'est peut-être que vous avez fait une faute de frappe en tapant son nom lors de l'enregistrement. Dans ce cas, renommez-le. Enfin, assurez-vous que vous avez bien enregistré le fichier dans le dossier dans lequel vous essayez de lancer le compilateur. Il est déconseillé d'effectuer cette vérification à l'aide de la commande **dir** du DOS, car celle-ci n'affiche pas les noms de fichiers longs. Il vaut donc mieux le faire sous Windows.

Note : On pourrait penser qu'il vaut mieux s'en tenir à des noms courts, mais cela ne résout pas le problème. En effet, le compilateur Java exige que l'extension du fichier soit **.java**, ce qui oblige le DOS à utiliser un nom long, même si celui-ci ne dépasse pas huit caractères.

Enfin, si vous obtenez un message du genre :

```
PremierProgramme.java:3: '{' expected.
          public static void main(String[] argv)
                                                  ^
1 error
```

c'est que vous avez fait une erreur de frappe à l'intérieur du programme. Essayez de la corriger et recommencez. (Ici, le compilateur indique que l'erreur se trouve à la ligne 3.)

Une fois que le programme est compilé correctement, exécutez-le en tapant la commande :

```
java PremierProgramme
```

puis la touche *Entrée* ou, plus simplement, si vous avez créé les fichiers batch du chapitre précédent :

```
r
```

puis les touches *F3* et *Entrée*, ce qui est tout de même beaucoup plus court. (La touche *F3* recopie la fin de la ligne précédente, ce qui est rendu possible parce que les commandes **ca** et **ra** font toutes deux la même longueur, ce qui n'est pas le cas des commandes **javac** et **java**.)

Votre programme doit être exécuté et afficher la ligne :

```
Ca marche !
```

Si vous obtenez un message d'erreur du type :

```
java.lang.ClassFormatError: PremierPrograme (wrong name)
```

c'est que vous avez fait une faute de frappe (ici, nous avons oublié volontairement un *m*.) Si vous utilisez les fichiers batch, cela ne devrait pas arriver, sauf dans un cas particulier. En effet, si vous ne respectez pas les majuscules lors de la compilation, le compilateur s'en moque. En revanche, l'interpréteur tient à ce que le nom du fichier soit tapé avec les majuscules. Nous verrons pourquoi dans une prochaine section.

Note : Si vous utilisez UltraEdit et si vous l'avez configuré comme nous l'indiquons à l'Annexe C, vous pouvez compiler le programme en tapant simplement la touche F11et l'exécuter en tapant F12.

Normalement, vous ne devriez pas obtenir une erreur d'exécution pour ce programme. En effet, Java est conçu pour qu'un maximum d'erreurs soient détectées lors de la compilation. Comme ce programme ne fait quasiment rien, il devrait s'exécuter sans problème.

Analyse du premier programme

Nous allons maintenant analyser étape par étape tout ce que nous avons fait jusqu'ici afin de bien comprendre comment tout cela fonctionne.

```
// Mon premier programme en Java
class PremierProgramme {
   public static void main(String[] argv) {
      System.out.println("Ca marche !");
   }
}
```

La première ligne de notre programme est une ligne de commentaire. Elle commence par //. Tout ce qui est compris entre ces deux caractères et la fin de la ligne est ignoré par le compilateur. (En revanche, d'autres outils peuvent utiliser le texte contenu dans ce type de commentaire.)

```
// Mon premier programme en Java
class PremierProgramme {
   public static void main(String[] argv) {
      System.out.println("Ca marche !");
   }
}
```

La ligne suivante commence par le mot **class**, qui veut dire que nous allons définir une nouvelle classe Java, suivi du nom de cette classe, qui est aussi le nom de notre programme. Nous verrons au chapitre suivant ce que sont exactement les classes. Sachez simplement pour l'instant les choses suivantes :

- Un programme comporte au moins une classe.

- Cette classe doit obligatoirement porter le même nom que le fichier dans lequel le programme est enregistré. Le fichier possède, en plus, l'extension **.java**.

- En Java, les majuscules et les minuscules sont considérées comme des caractères différents. Ainsi, **PremierProgramme** n'est pas identique à **premierProgramme** ou à **PREMIERprogramme**.

- En Java, les fins de lignes n'ont généralement pas de signification particulière. Elles sont simplement des séparateurs, au même titre que les espaces ou les tabulations. On peut le plus souvent utiliser comme séparateur entre deux mots Java n'importe quelle combinaison d'espace, de tabulations et de fins de lignes. Cette particularité est d'ailleurs mise à profit pour mettre en forme le code d'une façon qui facilite sa lecture, comme nous le verrons plus loin. Cependant, la première ligne de notre programme montre un exemple contraire. Dans ce type de commentaire, le début est marqué par deux barres obliques (pas forcément en début de ligne, d'ailleurs) et la fin de ligne marque la fin du commentaire. C'est pratique, mais pas très logique.

- La deuxième ligne se termine par le caractère **{**, qui marque le début d'un bloc. Nous verrons plus loin ce que représente exactement un bloc. Sachez seulement pour l'instant que la *déclaration* d'une nouvelle classe est suivie de sa *définition*, et que celle-ci est incluse dans un bloc, délimité par les caractères **{** et **}**. Ici le caractère marquant le début du bloc se trouve à la fin de la deuxième ligne. Le caractère marquant la fin du bloc se trouve sur la dernière ligne du programme. En effet, celui se trouvant sur l'avant-dernière ligne correspond à l'ouverture d'un autre bloc, à la fin de la troisième ligne.

Il existe de nombreuses façons de formater le code Java. Aucune n'est obligatoire. Il est cependant indispensable de respecter une certaine logique. Nous avons pour habitude (comme à peu près tout le monde) de passer à la ligne suivante immédiatement après l'ouverture d'un bloc, sauf si la totalité du bloc peut tenir sur la ligne. Par ailleurs, nous indenterons d'une tabulation le code se trouvant à l'intérieur du bloc. Enfin, le caractère de fermeture du bloc se trouvera toujours sur une ligne isolée (sauf dans le cas où le bloc tient sur une seule ligne) et sera aligné sur le début de la ligne comportant le caractère d'ouverture du bloc. De cette façon, la structure du code est parfaitement lisible. Vous pouvez adopter une autre habitude si vous le souhaitez. Cependant, il est conseillé de s'en tenir rigoureusement à une seule manière.

```
// Mon premier programme en Java
class PremierProgramme {
   public static void main(String[] argv) {
      System.out.println("Ca marche !");
   }
}
```

La troisième ligne est donc décalée d'une tabulation car elle se trouve à l'intérieur du bloc qui vient d'être ouvert. C'est la première ligne de la définition de la classe que nous sommes en train de créer.

Cette ligne est un peu plus complexe. La partie fondamentale est :

```
void main() {
```

main() { signifie que nous allons définir une méthode, ce qui est indiqué entre autres, par les parenthèses (). Une méthode est une sorte de procédure appartenant à une classe. Lorsqu'elle est exécutée, elle prend des paramètres de types précis et renvoie une valeur de type tout aussi précis. Le mot **main** est le nom de la méthode que nous allons définir. Le mot **void** désigne le type de valeur renvoyé par la méthode. Et justement, **void** signifie "rien", c'est-à-dire aucune valeur. Notre méthode ne renverra aucune valeur. Elle ne sera donc pas utilisée pour la valeur qu'elle renvoie, mais pour l'interaction qu'elle pourra avoir avec l'environnement (ce que l'on appelle les effets

de bord). Les paramètres que prend la méthode sont placés entre les parenthèses, chaque paramètre ayant un nom, précédé de son type. Dans l'exemple simplifié, la méthode ne prend pas de paramètres. Dans notre programme réel, elle prend un paramètre, dont le nom est **argv** et qui est de type **String[]**. Nous verrons plus loin ce que cela signifie. Sachez seulement pour l'instant que notre méthode n'utilisera pas ce paramètre, mais que nous sommes obligés de le faire figurer ici. Sans ce paramètre, le programme sera correctement compilé, mais ne pourra pas être exécuté.

Les mots **public** et **static** décrivent chacun une caractéristique de notre méthode. **public** indique que notre méthode peut être utilisée par n'importe qui, c'est-à-dire par d'autres classes, ou par l'interpréteur Java. Oublier le mot **public** n'empêcherait pas le programme d'être compilé normalement, mais cela empêcherait l'interpréteur de l'utiliser. En effet, l'interpréteur a pour fonction principale d'exécuter la méthode **main** du programme qu'on lui soumet.

Le mot **static** a une signification un peu plus complexe, qui s'éclaircira dans un prochain chapitre. Sachez simplement pour l'instant que certaines méthodes sont de type **static** et d'autres non.

Pour résumer, retenez qu'une application Java doit contenir une classe :

- portant le même nom que le fichier dans lequel elle est enregistrée,
- comportant (au moins) une méthode :
    - de type **public** et **static**
    - appelée **main**,
    - ayant un argument de type **String[]**.

Note 1 : Vous trouverez le plus souvent dans la littérature consacrée à Java le paramètre de la méthode **main** sous la forme **main(String argv[])**. Sachez que les deux notations :

```
String argv[]
String[] argv
```

sont acceptables. **String argv[]** est pratiquement toujours employé, probablement parce que le message d'erreur affiché par l'interpréteur Java lorsque ce paramètre manque est :

```
void main(String argv[]) is not defined
```

Cependant, nous verrons au chapitre traitant des tableaux que la notation :

```
String[] argv
```

est beaucoup plus logique. Par ailleurs, le nom du paramètre n'est pas obligatoirement **argv**. Vous pouvez utiliser n'importe quel nom, à condition de n'utiliser que des lettres, des chiffres et le caractère _, et de commencer par une lettre ou par _. Vous pouvez utiliser pratiquement autant de caractères que vous le souhaitez. (Nous avons testé un programme utilisant des noms de plus de quatre mille caractères sans problème.)

Note 2 : Nous avons dit que si vous oubliez le paramètre de la méthode **main**, le programme est compilé sans erreur mais ne s'exécute pas. Il en est de même, d'ailleurs, si votre programme ne comporte pas de méthode **main**. Cela peut sembler en contradiction avec la philosophie de Java, qui consiste à détecter un maximum d'erreurs lors de la compilation. Il n'en est rien. L'erreur n'est pas en effet d'omettre la méthode **main** ou son paramètre, mais de tenter d'exécuter le programme. Il est parfaitement normal de vouloir compiler une classe qui ne constitue pas un programme exécutable mais simplement un élément qui sera utilisé par une application ou une applet.

```
// Mon premier programme en Java
class PremierProgramme {
   public static void main(String[] argv) {
      System.out.println("Ca marche !");
   }
}
```

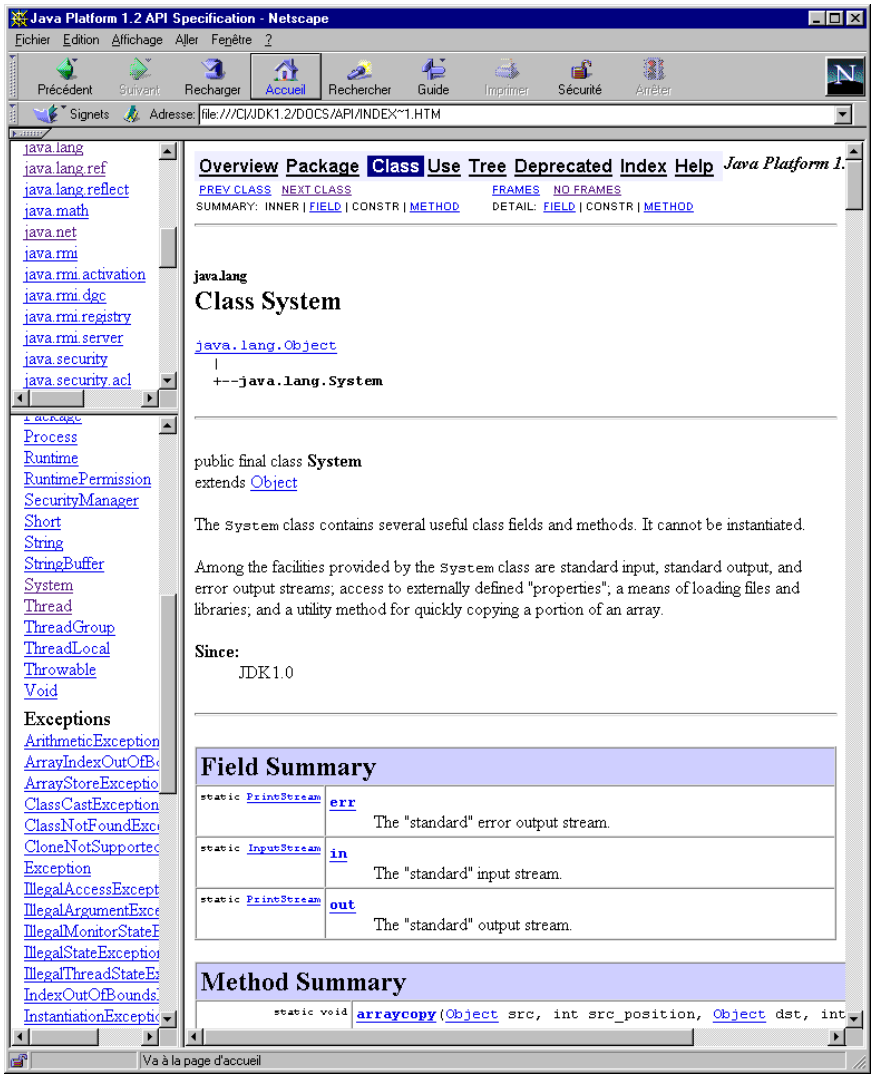
La quatrième ligne du programme constitue le corps de la méthode **main**. Elle comporte un appel à une méthode et un paramètre. La méthode appe-

lée est **println**. Cette méthode prend un paramètre de type *chaîne de caractères* et affiche sa valeur sur la sortie standard, en la faisant suivre d'un saut de ligne. Le paramètre est placé entre parenthèse. Ici, il s'agit d'une *valeur littérale*, c'est-à-dire que la chaîne de caractères est écrite en toutes lettres entre les parenthèses. Pour indiquer qu'il s'agit d'une valeur littérale, la chaîne de caractères est placée entre guillemets.

La méthode **println** ne tombe pas du ciel lorsque l'interpréteur Java en a besoin. Pour qu'il puisse l'utiliser, il faut lui indiquer où il peut la trouver. Les méthodes Java n'existent que dans des objets, qui peuvent être des classes, ou des *instances de classes*. Nous verrons au prochain chapitre la différence (fondamentale) qui existe entre une classe et une instance de classe. (Sachez simplement pour l'instant que ce qui est **static** appartient à une classe et ce qui ne l'est pas à une instance.) La méthode **println** à laquelle nous faisons référence ici appartient à l'objet **out**. Cet objet appartient lui-même à la classe **System**.

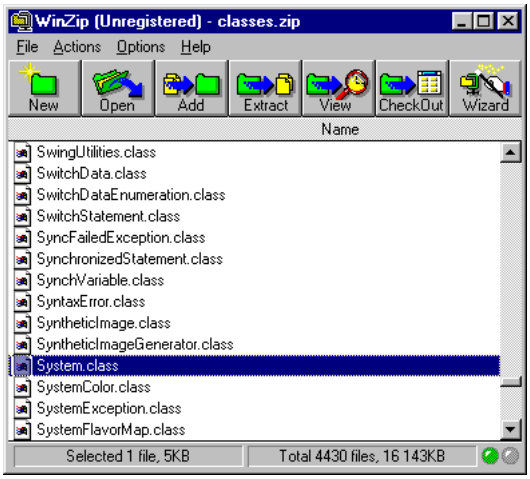
Si vous voulez obtenir des informations sur une classe standard du langage Java, vous devez utiliser la documentation fournie. Si vous ouvrez celle-ci à l'aide de votre navigateur Web en faisant un double clic sur l'icône du raccourci que nous avons créé au chapitre précédent, vous accédez directement au document **index.html** se trouvant dans le dossier **c:\jdk1.2\docs\api**. En haut du cadre principal, cliquez sur le lien *Index*, puis sur la lettre *S*, et faites défiler la page jusqu'à l'entrée **System**. Une fois l'entrée **System** localisée, cliquez sur le lien correspondant. (Pour trouver plus rapidement une entrée, vous pouvez utiliser la fonction de recherche de mot de votre navigateur.)

Vous accédez alors à la description de la classe **System** (page suivante). Le haut de la page montre sa hiérarchie. Nous verrons dans un prochain chapitre ce que cela signifie. Notez cependant une chose importante : le nom complet de la classe est **java.lang.System**. Vous trouvez ensuite une description de la classe, puis une liste des ses *champs* (*fields*), et enfin, une liste de ses méthodes. Vous pouvez constater que la classe **System** possède un champ **out**, qui est un objet de type **PrintStream**. De plus, cet objet est **static**. En cliquant sur le lien **PrintStream**, vous obtenez l'affichage de la documentation de cette classe. Vous pouvez alors constater qu'elle contient la méthode **println(String x)**, celle que nous utilisons dans notre programme.



Peut-être vous posez-vous la question suivante : Comment se fait-il que nous pouvons utiliser la classe **java.lang.System** en y faisant référence uniquement à l'aide du nom **System** ? En fait, le nom de la classe est simplement **System**. L'expression **java.lang** est en quelque sorte le chemin d'accès à cette classe. Si vous décompressez le contenu du fichier **src.jar** (se trouvant dans le dossier d'installation du JDK), vous pourrez constater

que vous obtenez un sous-dossier nommé **java**, que celui-ci contient à son tour un sous-dossier nommé **lang**, et qu'enfin celui-ci contient un fichier nommé **System.java**. Bien sûr, il s'agit là des sources de la classe. La classe **System** compilée est le fichier **System.class**. Vous ne trouverez pas ce fichier car il est inclus dans le fichier **rt.jar**, comme vous pouvez le voir dans l'illustration ci-après.



Cependant, son chemin d'accès, à l'intérieur du fichier compressé, est identique à celui du fichier source.

Cela ne répond toutefois pas à la question : Comment fait l'interpréteur pour savoir qu'il doit chercher la classe **System** avec le chemin d'accès **c:\jdk1.2\lib\java\lang** ? (A partir de maintenant, nous considérerons comme tout à fait transparente l'utilisation du fichier de classes compressé **rt.jar** et raisonnerons comme si celui-ci était décompressé.)

En fait, la première partie du chemin d'accès est indiquée par la variable d'environnement **CLASSPATH**, configurée dans le fichier de démarrage **autoexec.bat**. Cette variable devrait prendre ici la valeur **c:\jdk1.2\lib**. Cela n'est toutefois pas nécessaire, car Java utilise par défaut (c'est-à-dire en l'absence de toute autre indication) le sous-dossier **lib** du dossier d'installation du JDK.

La deuxième partie du chemin d'accès n'est pas nécessaire non plus, car Java utilise également, mais dans un processus totalement différent, le chemin d'accès **java.lang** comme chemin par défaut. Toutes les classes se trouvant dans **java.lang** sont donc directement accessibles à tous les programmes Java.

Note : La distinction entre les deux chemins d'accès par défaut est importante. En effet, le premier est extérieur à Java et ne dépend que du système. Sa syntaxe peut être différente d'un système à l'autre, particulièrement en ce qui concerne les séparateurs de niveaux (\ pour MS-DOS et Windows). En revanche, pour assurer la portabilité des applications, la deuxième partie du chemin d'accès est traduite par Java d'une façon identique sur tous les systèmes. C'est ainsi que le séparateur de niveaux employé est le point. Vous pouvez parfaitement remplacer la ligne :

```
System.out.println("Ca marche !");
```

par :

```
java.lang.System.out.println("Ca marche !");
```

mais pas par :

```
java\lang\System.out.println("Ca marche !");
```

ce qui entraînerait trois erreurs de compilation (bien que le programme ne comporte que deux erreurs !).

La méthode **println** envoie son argument (ici la chaîne de caractères) à la *sortie standard*. Généralement, il s'agit de l'écran, mais cela n'est pas vraiment du ressort de Java. Le système est libre d'en faire ce qu'il veut. (Sous MS-DOS, la sortie standard - *standard output* - peut être redirigée, contrairement à la sortie d'erreurs - *standard error*.)

Note : Si vous avez consulté la documentation de Java comme indiqué précédemment, vous avez certainement remarqué qu'il existe également une

méthode **print**. La plupart des livres sur Java indiquent que la différence entre ces deux méthodes est l'ajout d'un caractère *fin de ligne* à la fin de la chaîne affichée. Il existe cependant une différence plus importante. En effet, l'écriture vers la sortie standard peut faire appel à un *tampon*, qui est une zone de mémoire recevant les caractères à afficher. Normalement, le contrôle de ce tampon est dévolu au système. Java peut néanmoins provoquer le vidage du tampon afin d'être sûr que les caractères soient réellement affichés. Sous MS-DOS, le système affiche les caractères au fur et à mesure de leur entrée dans le tampon. Une chaîne de caractères affichée par la méthode **print** sera donc visible immédiatement à l'écran, comme dans le cas de la méthode **println**. La seule différence sera bien le passage à la ligne dans le second cas. En revanche, rien ne vous garantit qu'il en sera de même avec d'autres systèmes. Un autre système peut très bien garder les caractères dans le tampon et ne les afficher que lorsque l'impression est terminée, ou lorsque le tampon est plein, ou encore lorsqu'un caractère spécial est reçu (le plus souvent, une fin de ligne). La méthode **println**, en plus d'ajouter une fin de ligne, provoque le vidage du tampon, et donc assure que l'affichage des caractères est effectivement réalisé. (En tout état de cause, le tampon est vidé et tous les caractères sont affichés lorsque la sortie standard est refermée.)

Notez enfin que la ligne contenant l'appel à la méthode **println** est terminée par un point-virgule. Le point-virgule est un séparateur très important en Java. En effet, nous avons vu que les fins de ligne n'ont pas de signification particulière, sauf à l'intérieur des commentaires commençant par //. Vous pouvez donc ajouter des sauts de ligne entre les mots du langage sans modifier la syntaxe. La contrepartie est que vous devez indiquer à Java la terminaison des instructions.

Attention : En Java, contrairement à ce qui est pratiqué par d'autres langages, le point-virgule est toujours obligatoire après une instruction, même si celle-ci est la dernière d'un bloc. Dans notre programme, la quatrième ligne est la dernière d'un bloc. Bien que le point-virgule soit redondant dans ce cas (l'instruction se termine forcément puisque le bloc qui la contient se termine), son omission entraîne automatiquement une erreur de compilation. Cette légère contrainte vous évitera bien des soucis lorsque vous ajouterez des lignes à la fin d'un bloc !

```
// Mon premier programme en Java
class PremierProgramme {
   public static void main(String[] argv) {
      System.out.println("Ca marche !");
   }
}
```

Les deux dernières lignes de notre programme comportent les caractères de fin de bloc. Le premier correspond la fermeture du bloc contenant la définition de la méthode **main**. Il est donc aligné sur le début de la ligne contenant le caractère d'ouverture de bloc correspondant. Ce type d'alignement n'est pas du tout obligatoire. Il est cependant très utile pour assurer la lisibilité du code.

Le deuxième caractère de fin de bloc ferme le bloc contenant la définition de la classe **PremierProgramme.** Il est aligné de la même façon sur le début de la ligne contenant le caractère d'ouverture.

Première applet

Le programme que nous venons de créer était une *application*. C'est un bien grand mot pour un si petit programme, mais c'est là la façon de désigner les programmes qui doivent être exécutés depuis une ligne de commande. Les applets sont un autre type de programmes conçus pour être exécutés à l'intérieur d'une page HTML, dans une surface rectangulaire réservée à cet effet.

Les différences entre applications et applets sont multiples, mais on peut les classer en deux catégories :

- Les différences qui découlent de la nature des applets (le fait qu'elles doivent être exécutées dans une page HTML).

- Les différences qui ont été introduites volontairement, en général pour des raisons de sécurité.

Dans la première catégorie, on trouve un certain nombre de différences dues à la nature obligatoirement graphique des applets. Une applet peut être totalement invisible sur la page HTML qui la contient et peut effectuer un traitement ne nécessitant aucun affichage. Pour autant, elle est quand même construite à partir d'un élément de nature graphique.

Dans la deuxième catégorie, on trouve toutes les limitations imposées aux applets pour les empêcher d'attenter à la sécurité des ordinateurs. Les applets étant essentiellement destinées à être diffusées sur Internet, il leur est normalement interdit d'écrire ou de lire sur le disque de l'utilisateur. Elles ne peuvent (en principe) accéder qu'au disque du serveur sur lequel elles résident. Toute autre possibilité doit être considérée comme un bug qui risque de ne plus exister dans les prochaines versions du langage. De la même façon, les applets ne devraient pas, en principe, ouvrir des fenêtres sur l'écran. En effet, une fenêtre peut être utilisée pour faire croire à l'utilisateur qu'il s'agit d'une application et en profiter pour lui demander son numéro de carte bancaire, et le transmettre à un vilain pirate qui en fait collection. En fait, il n'est pas interdit à une applet d'ouvrir une fenêtre, mais celle-ci est clairement identifiée comme étant une fenêtre d'applet, à l'aide d'un gros bandeau qui occupe une partie de l'espace disponible (et complique singulièrement la tâche des programmeurs).

Il existe cependant une exception à ces limitations : il s'agit des applets signées, dont l'auteur peut être identifié, et pour lesquelles ces limitations sont levées.

Pour notre première applet, nous serons donc soumis à certaines contraintes. Pas celles qui empêchent d'accéder au disque, puisque notre programme ne le fait pas. En revanche, nous ne pourrons pas utiliser la sortie standard. De plus, la structure imposée aux applets est assez différente de celle imposée aux applications. Voici le code de notre première applet :

```
// Ma première applet Java
public class PremièreApplet extends java.applet.Applet {
   public void init() {
      add(new java.awt.Label("Ça marche !"));
   }
}
```

Tapez ce programme et enregistrez-le sous le nom **PremièreApplet.java**. Compilez-le comme nous l'avons fait pour le programme précédent. Corrigez les erreurs éventuelles. Une fois le programme compilé sans erreur, tapez le code HTML suivant :

```
<html>
<body>
<applet code="PremièreApplet" width="200" height="150">
</applet>
</body>
</html>
```

et enregistrez-le dans le fichier **PremièreApplet.htm**.

Pour exécuter l'applet, tapez la commande :

```
appletviewer PremièreApplet.htm
```

L'applet est exécutée par l'AppletViewer et vous devez obtenir le résultat suivant :

Vous pouvez également ouvrir le document **PremièreApplet.htm** avec un navigateur compatible Java.

Analyse de la première applet

Cette première applet mérite quelques commentaires. Nous commencerons par l'examen du code, ligne par ligne.

```
// Ma première applet Java
public class PremièreApplet extends java.applet.Applet {
   public void init() {
      add(new java.awt.Label("Ça marche !"));
   }
}
```

Nous passerons sur la première ligne, qui est identique à celle du programme précédent. La deuxième ligne, en revanche, présente deux différences. La première est que la classe que nous créons est précédée du mot **public**. Cela est nécessaire pour que l'AppletViewer ou un navigateur HTML puisse l'utiliser. Par ailleurs, le nom de la classe est suivi des mots **extends java.applet.Applet**, qui signifient que notre classe est définie par extension d'une classe existante, la classe **Applet**. Celle-ci peut être atteinte en utilisant le chemin d'accès **java.applet**. De cette façon, toutes les fonctionnalités de cette classe seront disponibles dans la nôtre. En particulier, la méthode **init** qui nous intéresse plus particulièrement.

Si vous consultez le fichier **Applet.java**, qui contient les sources de la classe **Applet** et qui se trouve dans le dossier dont le chemin d'accès est **\java\applet** une fois le fichier **scr.jar** décompressé, comme vous pouvez le déduire de ce qui est indiqué sur deuxième ligne du programme, vous constaterez que la méthode **init** ne prend aucun paramètre et ne fait rien :

```
/**
 * Called by the browser or applet viewer to inform
 * this applet that it has been loaded into the system. It is always
 * called before the first time that the <code>start</code> method is
 * called.
```

```
* <p>
* A subclass of <code>Applet</code> should override this method if
* it has initialization to perform. For example, an applet with
* threads would use the <code>init</code> method to create the
* threads and the <code>destroy</code> method to kill them.
* <p>
* The implementation of this method provided by the
* <code>Applet</code> class does nothing.
*
* @see     java.applet.Applet#destroy()
* @see     java.applet.Applet#start()
* @see     java.applet.Applet#stop()
*/
public void init() {
}
```

Les dix-huit premières lignes sont des lignes de commentaires. Il s'agit d'une forme de commentaire différente de celle que nous avons déjà vue. Le début en est marqué par les caractères **/*** et le compilateur Java ignore tout ce qui se trouve entre ces deux caractères et les caractères ***/**, qui marquent la fin des commentaires. (En revanche, le programme Javadoc utilise les commandes qui sont insérées dans ces commentaires pour générer automatiquement la documentation du programme. Nous y reviendrons.)

Les deux lignes qui nous intéressent sont les deux dernières, qui définissent la méthode **init**. Cette définition est vide, ce qui signifie que la méthode **init** ne fait rien.

En fait, ce qui nous intéresse n'est pas ce que fait cette méthode, mais le fait qu'elle soit automatiquement appelée lorsque l'applet est chargée par le navigateur. Ainsi, il nous suffit de redéfinir cette méthode pour lui faire faire ce que nous voulons. Nous reviendrons dans un prochain chapitre sur cette possibilité de redéfinir des méthodes existantes. Considérez seulement, pour le moment, que nous utilisons non pas ce que fait la méthode, mais le fait qu'elle soit exécutée automatiquement.

```
// Ma première applet Java
public class PremièreApplet extends java.applet.Applet {
   public void init() {
      add(new java.awt.Label("Ça marche !"));
   }
}
```

A la troisième ligne de notre applet, nous déclarons la méthode **init**. Elle est de type **void**, c'est-à-dire qu'elle ne retourne aucune valeur. Elle est **public**, ce qui est obligatoire parce que la méthode d'origine, dans la classe **Applet**, l'est. (Nous y reviendrons.) Elle n'utilise aucun paramètre. (Cependant, les parenthèses sont obligatoires.)

```
// Ma première applet Java
public class PremièreApplet extends java.applet.Applet {
   public void init() {
      add(new java.awt.Label("Ça marche !"));
   }
}
```

La ligne suivante définit ce que fait la méthode **init**. Elle utilise la méthode **add** pour ajouter quelque chose. Si vous consultez la documentation de Java, vous constaterez que la classe **Applet** ne comporte pas de méthode **add**. D'où vient celle-ci alors ? Nous avons vu précédemment que le compilateur était capable d'accéder par défaut aux classes se trouvant dans **java.lang**. Ici, cependant, nous n'avons pas indiqué de classe. Ce mécanisme ne peut donc pas être mis en œuvre. De plus, lorsque l'on invoque une méthode sans indiquer autre chose, cette méthode est recherchée dans la classe dans laquelle se trouve l'invocation, c'est-à-dire **Applet**.

C'est encore une fois la documentation de Java qui nous donne la réponse. En effet, la page consacrée à la classe Applet nous indique toute la hiérarchie de celle-ci :

```
java.lang.Object
    |
    +----java.awt.Component
            |
            +----java.awt.Container
                    |
                    +----java.awt.Panel
                            |
                            +----java.applet.Applet
```

Nous voyons ici que la classe **java.awt.Applet** dérive de la classe **java.awt.Panel**. Pour respecter la terminologie de Java, on dit que la classe **java.awt.Applet** constitue une *extension* de la classe **java.awt.Panel** qui est elle-même une extension de la classe **java.awt.Container**. Cette classe se trouve elle-même une extension de la classe **java.awt.Component**, à son tour extension de la classe **java.lang.Object**. De cette façon, la classe **java.awt.Applet** *hérite* de toutes les méthodes des classes parentes. La documentation de la classe **java.awt.Applet** donne d'ailleurs la liste des méthodes héritées. En faisant défiler la page vers le bas, vous constaterez que la méthode **add** est héritée de la classe **java.awt.Component**. Comme notre classe **PremièreApplet** est une extension de la classe **java.awt.Applet** (comme nous l'avons indiqué à la deuxième ligne du programme), elle hérite également de cette méthode.

Note : Vous vous demandez peut-être de quoi la classe que nous avons appelée **PremierProgramme** était l'extension. De rien, étant donné que nous n'avions pas utilisé le mot **extends** ? Non, car en fait toute classe créée sans mentionner explicitement qu'elle est l'extension d'une classe particulière est automatiquement l'extension de la classe la plus générale, **java.lang.Object**.

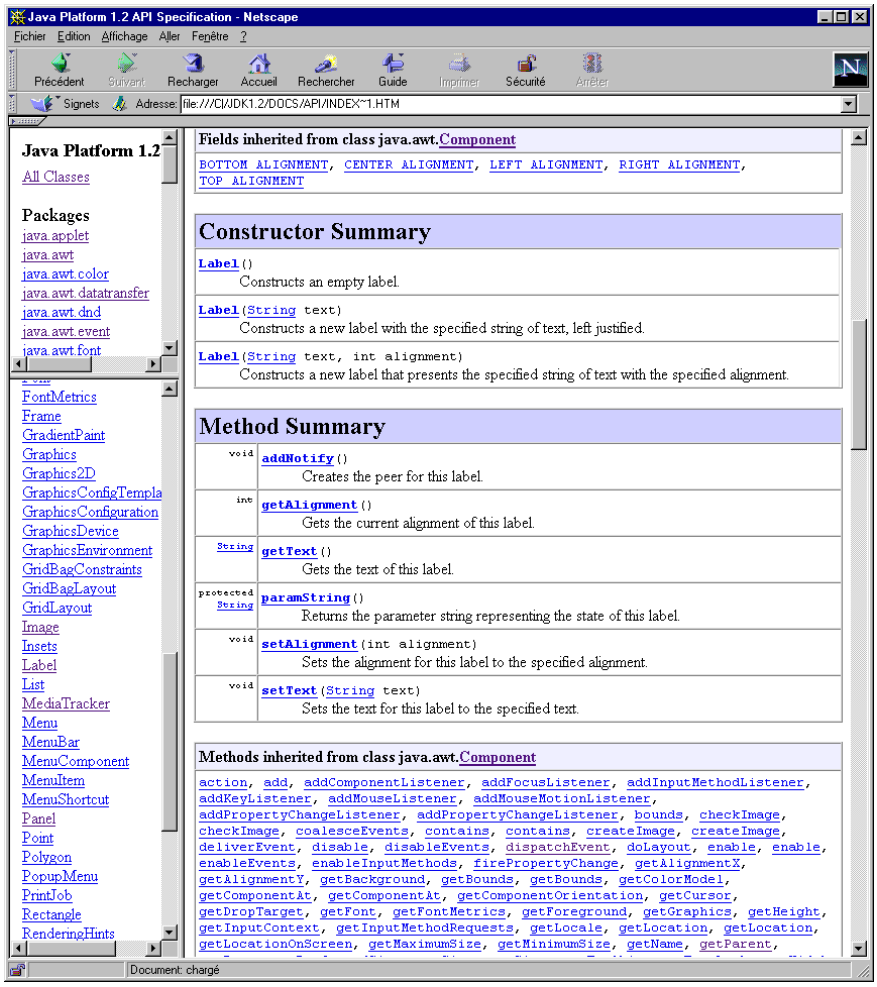
Ce que la méthode **add** ajoute à notre applet est tout simplement son argument, placé entre parenthèses. Nous devons ajouter un élément susceptible d'afficher du texte. Il existe plusieurs possibilités. La plus simple est de créer un objet de la classe **java.awt.Label**. Cet objet affiche simplement une chaîne de caractères.

Pour créer un tel objet, nous utilisons le mot clé **new**, suivi du nom de la classe dont l'objet est une *instance*, et des éventuels paramètres.

Attention : Il ne s'agit plus ici de créer une classe dérivant d'une autre classe, mais un objet qui est un représentant d'une classe existante. Ce concept, absolument fondamental, sera étudié en détail au chapitre suivant.

La syntaxe à utiliser pour créer une instance d'une classe consiste en le nom de la classe, suivi d'un ou de plusieurs paramètres placés entre parenthèses. (Cette syntaxe est en fait identique à celle utilisée pour l'invocation d'une méthode. Cela est dû au fait que l'on utilise, pour créer une instance, une méthode particulière qui porte le même nom que la classe. Cette méthode est appelée *constructeur*.) Pour déterminer les paramètres à fournir, reportez-vous une fois de plus à la documentation de Java. Cliquez sur le lien *Index*, puis sur *L* et trouvez la ligne *Label*. Cliquez sur le lien correspondant. Vous pouvez également sélectionner **java.awt** dans le cadre supérieur gauche pour afficher la liste des classes correspondantes. Sélectionnez ensuite **Label** dans le cadre inférieur droit. Vous devez obtenir la page consacrée à la classe **Label** : en faisant défiler cette page, vous trouverez la liste des *constructeurs* de cette classe comme indiqué sur la figure de la page suivante.

Vous pouvez constater qu'il en existe trois. (Si le fait qu'il puisse exister trois méthodes portant le même nom vous perturbe, ne vous inquiétez pas. Nous expliquerons cela bientôt.) Nous avons utilisé celui prenant pour paramètre une chaîne de caractères. Nous avons par ailleurs, comme dans l'exemple précédent, choisi d'utiliser une chaîne sous forme littérale, c'est-à-dire incluse entre guillemets. Notez, au passage, l'utilisation d'un *Ç*, qui est possible ici car nous utilisons le jeu de caractères de Windows (ANSI) alors que cela ne l'était pas pour notre programme précédent. (Plus exactement, il aurait été possible de produire un tel caractère, mais de façon beaucoup plus complexe, notre éditeur de texte sous Windows n'utilisant pas le même jeu de caractères que notre programme Java.) Si vous avez du mal à trouver le *Ç* sur votre clavier, sachez que vous pouvez l'obtenir en maintenant la touche *Alt* enfoncée et en tapant *0199* sur le pavé numérique.




```
// Ma première applet Java
public class PremièreApplet extends java.applet.Applet {
   public void init() {
      add(new java.awt.Label("Ça marche !"));
   }
}
```

Les deux dernières lignes contiennent les caractères de fermeture de blocs, alignés comme dans l'exemple précédent.

Une façon plus simple d'exécuter les applets

Si vous procédez comme nous l'avons fait, vous devrez, pour chaque applet que vous écrirez, créer un fichier HTML. C'est un peu pénible et cela encombre inutilement votre disque. Si vous utilisez essentiellement l'AppletViewer, vous pouvez profiter d'une de ses particularités pour vous simplifier le travail. En effet, ce programme n'a que faire du code HTML, il ne s'intéresse qu'au *tag* (c'est le nom que l'on donne aux mots clés HTML) <applet> et ignore tout le reste. Il vous suffit d'ajouter au début de votre programme une ligne de commentaire contenant les tags nécessaires, de la façon suivante :

```
//<applet code="PremièreApplet" width="200" height="150"></applet>
public class PremièreApplet extends java.applet.Applet {
   public void init() {
      add(new java.awt.Label("Ça marche !"));
   }
}
```

pour que vous puissiez ouvrir directement le fichier source Java à l'aide de l'AppletViewer. Si vous avez créé les fichiers batch du chapitre précédent, vous pourrez alors compiler votre applet en tapant :

```
ca PremièreApplet
```

et l'exécuter ensuite en tapant :

```
va
```

puis la touche *F3*.

Résumé

Ce chapitre est maintenant terminé. Tout ce qui a été abordé n'a pas été expliqué en détail car les implications sont trop profondes et trop ramifiées. Il faudrait pouvoir tout expliquer d'un seul coup. Ne vous inquiétez pas si certains points restent encore très obscurs. C'est normal. Tout s'éclaircira par la suite.

Par ailleurs, ne prenez pas les deux programmes que nous avons écrits dans ce chapitre comme exemple de la meilleure façon de programmer en Java. Nous avons employé ici certaines techniques que nous éviterons à l'avenir, lorsque nous aurons pu approfondir certains points.

Pour l'instant, vous devez être capable de comprendre les principes généraux qui gouvernent la création d'une applet et d'une application. Vous êtes également capable de compiler et de tester un programme. De plus, et c'est peut-être le plus important, vous commencez à avoir un aperçu de la façon d'utiliser non seulement la documentation de Java, pour connaître les caractéristiques des différentes classes standard et les paramètres à employer avec leurs méthodes et constructeurs, mais également les sources de Java pour améliorer votre compréhension des principes mis en œuvre.

Exercice

A titre d'exercice, nous vous suggérons de refermer ce livre et de réécrire, compiler et tester les deux programmes que nous avons étudiés. Pour que l'exercice soit profitable, vous ne devez absolument pas consulter ce livre. En revanche, vous pouvez consulter autant que vous le souhaitez la documentation ou les sources de Java.

Chapitre 3

Les structures de contrôle

Dans ce chapitre, nous allons commencer une étude plus théorique qui permettra d'éclaircir les points restés obscurs dans les explications du chapitre précédent. Il est très important, pour écrire de façon efficace des programmes Java, de bien comprendre la philosophie du langage. Nous essaierons de rendre les choses aussi claires que possible, et utiliserons pour cela de nombreuses comparaisons. N'oubliez pas que ces comparaisons ne sont généralement valables que sous un point de vue limité. Elles ne servent qu'à illustrer les points abordés, et vous ne devez pas les pousser plus loin que nous le faisons (ou alors, sous votre propre responsabilité !).

Java est un langage *orienté objet*. C'est là plus qu'un concept à la mode. C'est une véritable philosophie de la programmation. Un programmeur écrit généralement un programme pour exprimer la solution d'un problème. Il faut entendre problème au sens large. Il peut s'agir d'un problème de calcul,

mais également du traitement d'une image, de la gestion de la comptabilité d'une entreprise, ou de la défense de la planète contre des extra-terrestres tout gluants.

Imaginer que le programme, en s'exécutant, permet de trouver la solution du problème est une erreur. Le programme EST la solution. L'exécuter, ce n'est qu'exprimer cette solution sous une forme utilisable, le plus souvent en la traduisant sous une forme graphique (au sens large : le texte blanc affiché sur un écran noir, c'est aussi une forme d'expression graphique), ou sonore, voire même tactile. (On ne connaît pas encore de périphérique de sortie olfactive.) Un autre élément important à prendre en compte pour l'expression de la solution au problème traité est le temps. Tout programme peut être exécuté manuellement, sans le secours d'un ordinateur. Le temps nécessaire, dans ce cas, à l'expression de la solution sous une forme utilisable est généralement rédhibitoire.

Ecrire un programme consiste donc à exprimer la solution d'un problème dans un langage qui pourra être "traduit" d'une façon ou d'une autre en quelque chose de compréhensible pour un utilisateur, humain ou non. (Il est fréquent que la sortie d'un programme soit destinée à servir d'entrée à un autre programme.)

Le langage utilisé pour décrire la solution du problème peut être quelconque. On peut très bien écrire n'importe quel programme en langage naturel. Le seul problème est qu'il n'existe aucun moyen, dans ce cas, de le faire exécuter par un ordinateur. Il nous faut donc utiliser un langage spécialement conçu dans ce but.

Chaque processeur contient son propre langage, constitué des instructions qu'il est capable d'exécuter. On peut exprimer la solution de n'importe quel problème dans ce langage. On produit alors un programme en langage machine. La difficulté vient de ce qu'il est difficile de représenter certains éléments du problème dans ce langage, toujours très limité. On a donc mis au point des langages dit "évolués", censés améliorer la situation. Les premiers langages de ce type, comme FORTRAN, étaient très orientés vers un certain type de problème : le calcul numérique. (FORTRAN signifie d'ailleurs FORmula TRANslator, tout un... programme !) Ce type de lan-

gage ne fait d'ailleurs que reproduire le fonctionnement d'un processeur, en un peu plus évolué.

L'usage des ordinateurs se répandant, on a été amenés rapidement à traiter des problèmes autres que le calcul numérique. L'élément fondamental des problèmes à traiter était l'algorithme, c'est-à-dire la méthode de traitement. On a donc créé des langages spécialement conçus dans cette optique. Cependant, il est rapidement apparu que cette approche était mal adaptée à certains problèmes. En effet, l'écriture d'un programme consiste à modéliser la solution du problème, et donc à modéliser par la même occasion les éléments de l'univers du problème. Tant qu'il s'agissait de nombre, tout allait bien... et encore ! Le traitement des très grands nombres entiers était très difficile. Aucun langage ne permettait de représenter un nombre de 200 chiffres, tout simplement parce qu'aucun processeur ne disposait des moyens de les manipuler. La structure de l'univers du processeur s'imposait donc au programmeur pour représenter la solution de son problème. Ecrire un programme de traitement de personnel, par exemple, posait un problème difficile : si à chaque personne était attribués un nom, un prénom, un âge, une photo, une adresse, etc., comment manipuler simplement les données correspondant à chaque personne ? Il était possible d'utiliser des structures de données représentant chacun de ces éléments, mais il était impossible de les regrouper simplement. Aucun langage ne possédait le type de données "personne", et de toute façon, cela n'aurait servi à rien, car ce type recouvrait forcément une réalité très différente d'un programme à un autre.

Un certain nombre de langages ont été mis au point pour tenter d'exprimer la solution de problèmes concernant des structures très différentes des nombres manipulés par les langages précédents. Par exemple, Lisp a été créé pour traiter des listes. Au départ, il était surtout destiné au traitement du langage naturel, et aux problèmes d'intelligence artificielle. Le langage était donc conçu comme un outil spécifique, sur le modèle des problèmes à traiter. Cette approche était tout à fait justifiée jusqu'à ce que les programmeurs se mettent en tête d'utiliser leur langage pour exprimer d'autres types de problèmes.

Les Lispiens ont donc prétendu (à raison) que tout problème pouvait être modélisé à l'aide de listes. D'autres, pendant ce temps, démontraient que tout problème peut être modélisé sous forme de graphe. D'autres encore

montraient que la structure de pile pouvait servir à tout. Tous avaient raison, tout comme on peut démontrer que l'on peut tout faire avec un simple couteau. On peut même tout faire sans aucun outil ; par exemple construire un super-ordinateur à main nue, en partant de rien. C'est d'ailleurs ce que l'homme a réalisé, ce qui prouve bien que c'est possible. Question productivité, le tableau est moins gai. Il a tout de même fallu plusieurs millions d'années pour en arriver là.

Dès que l'on accorde de l'importance à la productivité, toutes les belles démonstrations ne valent plus rien. Avec la généralisation de l'usage des ordinateurs, est apparue la nécessité d'un langage permettant d'exprimer de façon simple et efficace n'importe quel problème. Les langages orientés objet répondent à cet objectif.

Tout est objet

Le principe fondamental est que le langage doit permettre d'exprimer la solution d'un problème à l'aide des éléments de ce problème. Un programme traitant des images doit manipuler des structures de données représentant des images, et non leur traduction sous forme de suite de 0 et de 1. Les structures de données manipulées par un programme de gestion de personnel doivent représenter des personnes, avec toutes les informations pertinentes, qu'il s'agisse de texte, de date, de nombre, d'images ou autres.

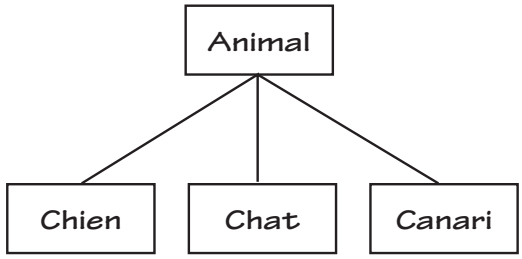
Java est l'aboutissement (pour le moment, en tout cas) de ce concept. Pour Java, l'univers du problème à traiter est constitué d'objets. Cette approche est la plus naturelle, car elle correspond également à notre façon d'appréhender notre univers. La modélisation des problèmes en est donc facilitée à l'extrême.

Tout est donc objet. Le terme d'objet est peut-être mal choisi. Il ne fait pas référence aux objets par opposition aux êtres animés. *Objet* signifie simplement "élément de l'univers".

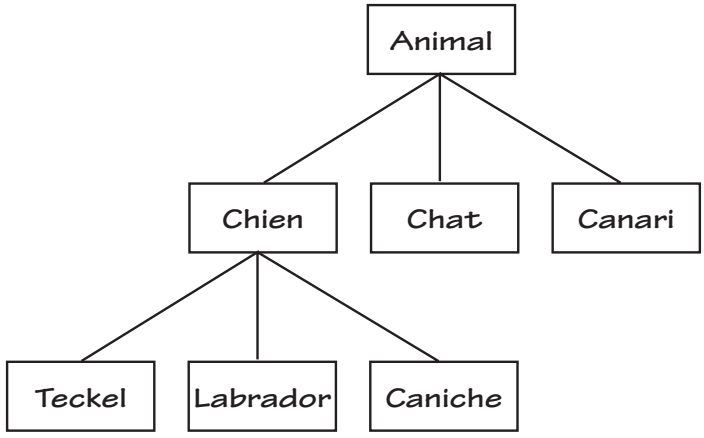
Les classes

Les objets appartiennent à des catégories appelées *classes*. L'ensemble des classes découpe l'univers. (Par univers, il faut entendre évidemment *univers du problème à traiter*.)

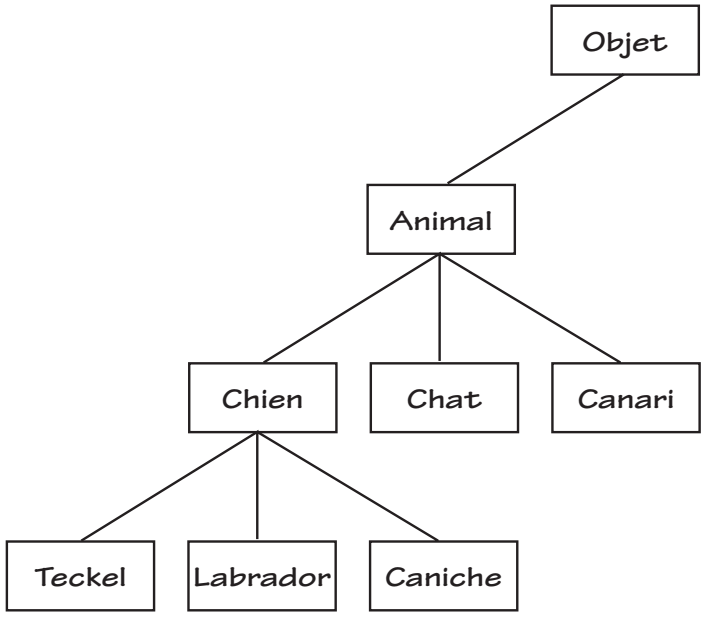
Si notre problème concerne les animaux, nous pouvons créer une classe que nous appellerons *animal*. Si nous devons considérer les chiens, les chats et les canaris, nous créerons trois classes dérivées de la classe *Animal* : les classes *Chien*, *Chat* et *Canari*. Peu importe que cette division ne soit pas pertinente dans l'univers réel. Il suffit qu'elle le soit dans celui du problème à traiter. Nous représenterons cette division de l'univers de la façon suivante :



Nous pouvons, si nécessaire, créer de nouvelles classes dérivées, par exemple :



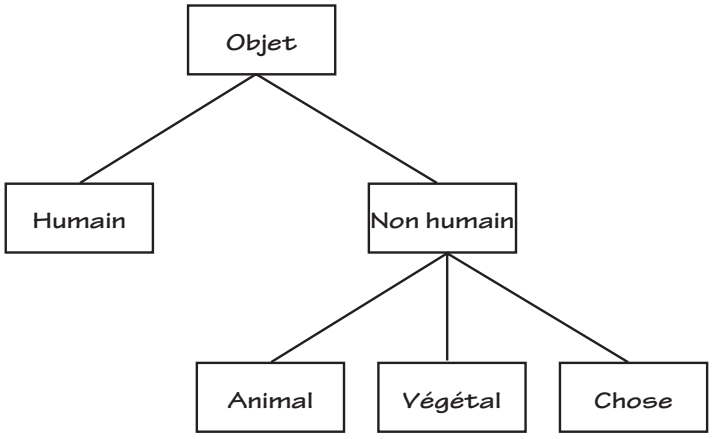
Il est d'usage de représenter ainsi sous forme d'arbre inversé, la hiérarchie des classes. La classe la plus générale se trouve à la racine (en haut, puisque l'arbre est inversé). Ces deux exemples sont incomplets. En effet, toutes les classes dérivent d'une même classe, la plus générale : la classe *Objet*. L'arbre précédent doit donc être complété de la façon suivante :

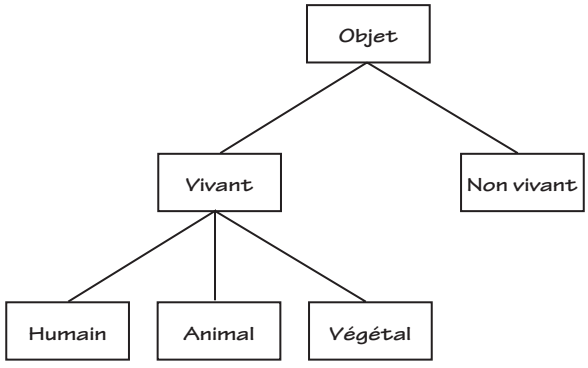


Pertinence du modèle

La structure que nous construisons ainsi représente l'univers du problème à traiter. S'il s'agit d'un problème de l'univers "réel", on peut être tenté d'essayer de construire un modèle du monde réel. C'est là un piège dans lequel il faut éviter de tomber, et ce pour deux raisons. La première est qu'un modèle du monde réel prendra en compte de très nombreux critères non pertinents en ce qui concerne notre problème. La deuxième est que ce que nous croyons être la structure du monde réel est en réalité la structure d'un modèle perceptif qui dépend de très nombreux facteurs (culturels et psychologiques en particulier) qui sont le plus souvent totalement parasites en ce qui concerne les problèmes que nous avons à traiter.

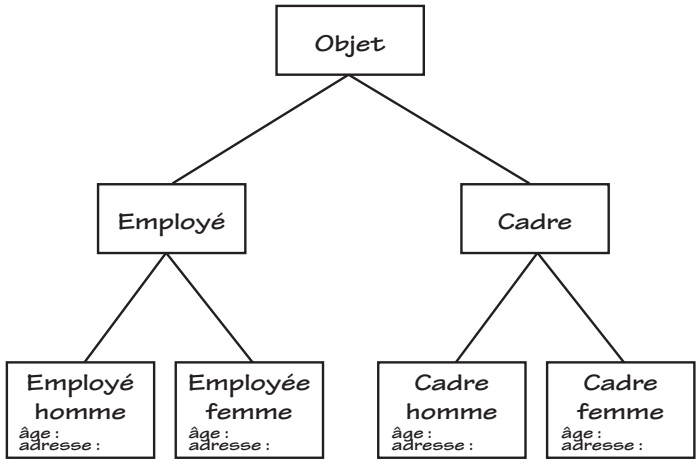
Par exemple, selon que nous sommes plutôt portés vers le créationnisme ou l'évolutionnisme, nous préférerons probablement l'un ou l'autre des modèles suivants :



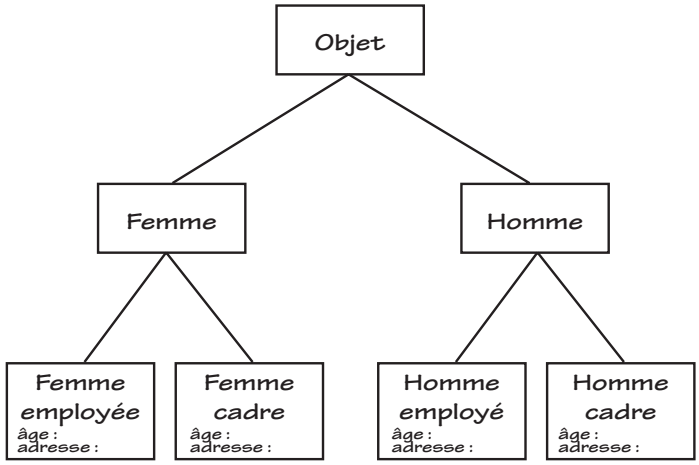


Peu importe lequel de ces modèles correspond le mieux au monde "réel". La seule chose qui compte véritablement est de savoir lequel est le plus efficace pour la représentation de notre problème.

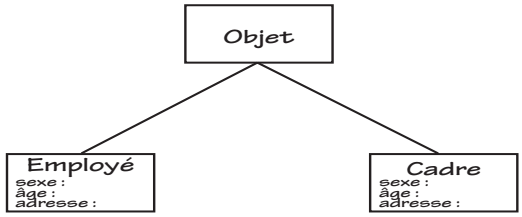
De la même façon, certaines caractéristiques des objets peuvent justifier ou non la création de classes distinctes. Ainsi, si vous écrivez un programme de gestion de personnel, vous créez probablement une classe *Employé* et une classe *Cadre*. Si vous voulez effectuer des traitements tenant compte du sexe, de l'âge et de l'adresse des membres du personnel, vous pourrez opter pour différentes structures, par exemple :



ou encore :

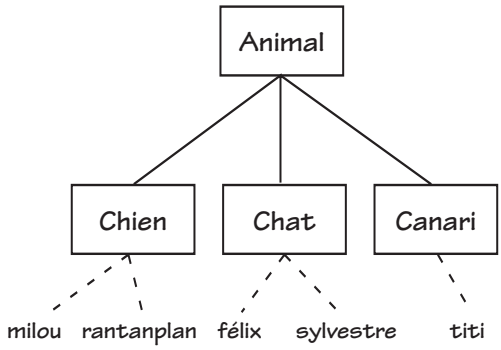


Ces deux structures peuvent sembler pertinentes, puisque si l'âge et l'adresse peuvent prendre n'importe quelles valeurs et sont susceptibles de changer (hélas), le sexe est une donnée qui détermine bien deux classes distinctes. Cependant, dans le cadre du problème qui nous concerne, il est probablement beaucoup plus simple et efficace d'utiliser un modèle comme celui-ci :



Les instances de classes

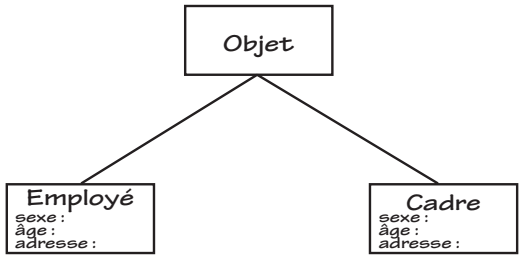
Considérez le modèle suivant :



Il est évident que le rapport entre *milou* et *Chien* n'est pas de même nature que le rapport entre *Chien* et *Animal*. *Chien* est une sous-classe de *Animal*. Dans la terminologie de Java, on dit que la classe *Chien étend* la classe *Animal*. En revanche, *milou* n'est pas une classe. C'est un représentant de la classe *Chien*. Selon la terminologie Java, on dit que *milou* est une *instance* de *Chien*.

Les membres de classes

Attention : piège ! Nous allons maintenant parler des *membres* d'une classe. Contrairement à ce que l'on pourrait croire, ce terme ne désigne pas les représentants de la classe (nous venons de voir qu'il s'agissait des *instances*), mais, en quelque sorte, ses caractéristiques. Dans l'exemple de gestion du personnel :



sexe, *âge* et *adresse* sont des *membres* de la classe *Employé*. Tous trois sont membres d'un même type : il s'agit de *variables*, c'est-à-dire d'éléments qui peuvent prendre une *valeur*. Chaque employé a un âge, un sexe et une adresse. (Il en est de même, dans cet exemple, pour chaque cadre.)

Les employés et les cadres ne se contentent pas forcément de posséder des variables. Ils peuvent également être capables de faire quelque chose ! Pour cela, il faut qu'ils possèdent des *méthodes*. Les *méthodes* décrivent simplement des procédures qui peuvent être exécutées pour :

- renvoyer une valeur,
- modifier l'environnement.

Par exemple, nous pouvons définir une classe *Employé* possédant une variable *matricule*, et une méthode *afficherMatricule()*.

Note : Nous respecterons ici l'usage typographique de Java :

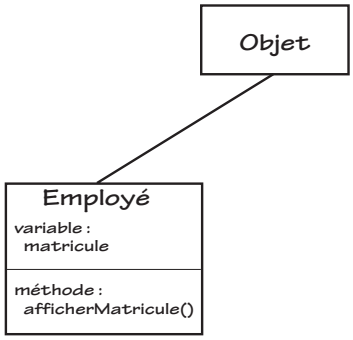
- Les noms de classes commencent par une majuscule.

- Les noms d'instances, de variables ou de méthodes commencent par une minuscule.

- Lorsqu'un nom est composé de plusieurs mots, les mots sont accolés et chacun d'eux (sauf le premier, pour lequel les règles ci-dessus s'appliquent) commence par une majuscule.

Les noms de méthodes sont faciles à distinguer des noms de variables car ils sont suivis de parenthèses encadrant les éventuels paramètres. Si une méthode ne prend pas de paramètres, les parenthèses sont vides mais doivent être présentes.

Notre nouvelle classe *Employé* se présente donc de la façon suivante :



Il faut noter que si la méthode *afficherMatricule()* se contente d'afficher la valeur de la variable *matricule*, d'autres méthodes comme, par exemple, la méthode *afficherAge()* devraient effectuer un certain nombre d'opérations pour pouvoir mener à bien leur tâche. Il s'agirait probablement (suivant le modèle choisi pour le reste de notre programme) d'interroger un objet pour connaître la date du jour, effectuer une soustraction (date du jour moins date de naissance), puis une série de conversions afin d'afficher l'âge.

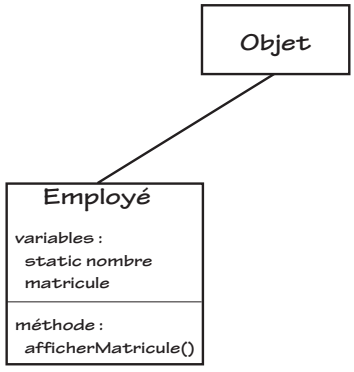
Les membres statiques

Il semble aller de soi que le matricule d'un employé soit une caractéristique de cet employé, et non une caractéristique de sa classe. Cependant, la diffé-

rence n'est pas toujours aussi évidente. Si nous créons une classe *Chien* étendant la classe *Animaux*, nous pouvons attribuer à la classe *Chien* une variable *nombreDePattes*. Il est cependant probable que cette variable aura la valeur 4 pour toutes les instances de *Chien*. En effet, tous les animaux n'ont pas quatre pattes, mais tous les chiens ont quatre pattes. Il serait donc inutile que chaque instance de *Chien* possède cette variable. Il suffit qu'il partage l'usage d'une variable commune à toute la classe. Ce type de variable est dit *statique*. Le terme correspondant en Java est **static**.

Note : Si le problème à traiter vous oblige à prendre en compte le fait qu'un chien puisse n'avoir que trois, deux, une seule ou même aucune patte, il vous suffira de créer la variable statique *nombreNormalDePattes* et une variable non statique *nombreDePattes*. Il vous sera alors possible de définir une méthode *estropié()* qui retournera la valeur *faux* si les variables *nombreDePattes* et *nombreNormalDePattes* ont des valeurs différentes, et *vrai* dans le cas contraire.

Dans le cas de notre classe *Employé*, nous supposerons que nous utiliserons comme matricule le numéro d'ordre de l'employé. Le premier employé créé aura le numéro 1, le deuxième le numéro 2, etc. Le problème qui se pose est alors de garder en mémoire le numéro du dernier employé créé. La solution consiste à ajouter une variable statique. Notre classe *Employé* devient alors :



Le matricule est propre à chaque employé (chaque *instance* d'*Employé*). La variable statique *nombre* est propre à la classe *Employé*. (Nous l'avons ap-

pelée *nombre* au lieu de *numéro*, car, dans le cadre de notre problème, le numéro attribué correspond au nombre d'employés. Bien sûr, dans la vie réelle, cela se passe rarement de cette façon, car des employés peuvent être supprimés sans que leurs numéros soient réaffectés.)

Les constructeurs

Vous vous demandez peut-être comment on fait pour créer une instance à partir d'une classe ? C'est très simple. Il suffit, pour cela, de faire appel à un *constructeur*.

Un constructeur est une *méthode* qui a pour particularité de posséder le même nom que la classe à laquelle il appartient. Si l'usage typographique de Java est respecté, il est donc très facile de distinguer un constructeur d'une méthode ordinaire : sa première lettre est en majuscule. (Il est également facile de le distinguer du nom de la classe à laquelle il appartient, puisqu'il est suivi de parenthèses encadrant ses éventuels arguments.)

Voyons comment pourrait se présenter un constructeur pour notre classe *Employé*. Pour créer un employé, il faut lui attribuer un matricule. Pour cela, il suffira d'interroger la classe pour connaître le nombre d'employés, d'augmenter ce nombre d'une unité et d'utiliser le résultat comme matricule. Voici à peu près ce que cela pourrait donner :

```
class Employé extends Object {
   int matricule;
   static int nombre;

   Employé() {
      matricule = ++nombre;
   }

   void afficherMatricule() {
      System.out.println(matricule);
   }
}
```

Nous avons ici écrit explicitement que la classe **Employé** étend la classe **Object**, ce qui n'était pas nécessaire, puisque nous avons vu précédemment que toutes les classes étendent par défaut la classe **Object**. Nous aurions pu tout aussi bien écrire :

```
class Employé {
```

Penchons-nous maintenant sur la deuxième ligne. Dans celle-ci, nous déclarons notre première variable. Ce sera un nombre entier (**int**). Le nom que nous lui donnons est *matricule*.

```
int matricule;
```

La troisième ligne fait exactement la même chose en déclarant une variable entière appelée **nombre**.

```
static int nombre;
```

Comme nous avons employé le mot **static**, il n'existera qu'un seul exemplaire de cette variable, qui appartiendra à la classe **Employé** et non à ses instances.

Attention : Tout ce que nous venons de décrire est très particulier. En effet, les variables de type **int** ne sont pas des objets ordinaires. Nous ne pouvons en dire plus pour le moment. Tout cela deviendra clair au prochain chapitre.

On trouve ensuite la définition du constructeur de la classe **Employé** :

```
Employé() {
   matricule = ++nombre;
}
```

A la première ligne, nous *déclarons* une méthode. Cette déclaration comporte le nom de la méthode et les paramètres qu'elle emploie. Le nom étant identique à celui de la classe à laquelle la méthode appartient, nous savons qu'il s'agit d'un constructeur. Cette méthode sera donc exécutée automatiquement par Java lorsque nous créerons une instance (un objet) de la classe

Employé. De plus, s'agissant d'un constructeur, cette méthode ne possède pas d'indication de type de valeur de retour. Entre les parenthèses se trouve la liste des *arguments*, ou *paramètres*, de la méthode. Chaque argument est composé d'un nom de classe et d'un nom d'instance, séparés par un espace. S'il y a plusieurs arguments, ils sont séparés par des virgules. Ici, nous n'utilisons aucun argument.

La déclaration de la méthode se termine par une accolade ouvrante qui ouvre un bloc. Ce bloc contient la *définition* de la méthode. Il se termine par une accolade fermante, à la quatrième ligne.

La déclaration ne comporte qu'une seule ligne :

```
matricule = ++nombre;
```

Sa signification (simplifiée) est : "Augmenter d'une unité la valeur de **nombre** et attribuer cette valeur à **matricule**."

Il faut bien comprendre que cette ligne fait deux choses différentes :

- Elle augmente la valeur de **nombre** ;
- Elle attribue la valeur de **nombre** à **matricule**.

Cela est tout à fait différent de la ligne suivante :

```
matricule = (nombre + 1);
```

qui attribuerait bien la valeur de **nombre** + 1 à **matricule**, mais laisserait **nombre** inchangé.

Nous avons ajouté ensuite une méthode qui affiche le matricule à l'aide de ce que nous avons étudié au chapitre précédent. Cette méthode n'étant pas un constructeur, elle doit être précédée de l'indication du type de valeur qu'elle fournit. Cette valeur est appelée *valeur de retour*. Cette méthode ne retourne aucune valeur. Il existe pour cela, comme nous l'avons vu au chapitre précédent, un type particulier : **void**, qui signifie "rien" ou "aucune valeur".

Création d'un objet

Voyons maintenant ce qui se passe lorsque nous créons un objet, c'est-à-dire une instance d'une classe. Nous avons vu que la création d'une instance mettait en jeu l'exécution du constructeur de sa classe. Ce qui nous manque, c'est la syntaxe à utiliser pour déclencher cette création.

Pour créer un objet en Java, il faut utiliser le mot clé **new**, suivi du nom du constructeur de la classe, avec les parenthèses et les arguments éventuels. Par exemple, pour créer une instance de la classe **Employé** (ou, en d'autres termes un objet de type **Employé**), nous utiliserons la syntaxe suivante :

```
new Employé();
```

Les questions qui se posent immédiatement sont : Que devient l'objet que nous avons créé, et comment pouvons-nous l'utiliser ? Pour répondre à ces questions, nous allons tester le programme suivant, version légèrement modifiée du précédent :

```
public class Test {
   public static void main(String[] argv) {
      new Employe();
   }
}

class Employe {
   int matricule;
   static int nombre;

   Employe() {
      matricule = ++nombre;
   }
   void afficherMatricule() {
      System.out.println(matricule);
   }
}
```

Les modifications que nous avons apportées au programme sont les suivantes :

- Notre classe a été renommée **Employe**, sans accent, pour éviter les problèmes.

- Nous avons créé une classe **Test** contenant une méthode **main()** de façon à pouvoir exécuter notre programme.

Saisissez ce programme, enregistrez-le dans un fichier que vous nommerez **Test.java**, compilez-le et exécutez-le. Si vous obtenez des messages d'erreurs, corrigez celles-ci et recommencez jusqu'à ce que le programme soit compilé et exécuté sans erreur. (N'oubliez pas de respecter la majuscule. Le nom du fichier doit être exactement identique au nom de la classe comportant la méthode **main()**.)

Note : Le programme se trouve également sur le CD-ROM accompagnant ce livre. Cependant, nous vous recommandons avec la plus haute insistance de saisir les programmes plutôt que de les copier depuis le CD-ROM. Votre apprentissage sera bien plus efficace si vous faites cet effort. En particulier, chaque erreur que vous ferez et corrigerez augmentera votre expérience plus que tout autre exercice.

Que fait ce programme ? Pour la partie qui nous intéresse, il crée une instance de **Employe**, ce qui est réalisé par la ligne :

```
new Employe();
```

Lorsque cette ligne est exécutée, Java cherche la classe **Employe** et la trouve dans le même dossier que la classe **Test**, qui constitue notre programme. En effet, lors de la compilation, le compilateur Java crée autant de fichiers **.class** qu'il y a de classes créées dans le programme compilé. (En fait, ce n'est pas tout à fait exact, mais considérez cela comme juste pour le moment.) Vous pouvez vérifier que le dossier Java contient bien les fichiers **Test.class** et **Employe.class**. Java vérifie si la classe **Employe** possède un constructeur ayant un argument correspondant à l'objet que nous lui passons, c'est-à-dire, dans notre cas, pas d'argument du tout. Comme c'est bien

le cas, ce constructeur est exécuté. Une instance d'**Employe** est donc créée dans la mémoire de l'ordinateur. Que pouvons-nous faire de cette instance ? Dans l'état actuel des choses, rien.

Avec certains langages, comme l'assembleur, lorsque vous créez une structure de données quelconque, vous devez lui allouer de la mémoire. Par exemple, si vous devez créer une variable entière capable de représenter des valeurs comprises entre - 30 000 et + 30 000, vous devrez réserver une zone de mémoire constituée de deux octets. Une fois cette zone réservée, vous pouvez y écrire ce que vous voulez, quand vous le voulez. Vous pouvez même écrire une valeur s'étendant sur trois octets ou même plus. Ce faisant, vous écrirez dans une zone non réservée, ce qui pourra avoir les conséquences les plus imprévisibles, allant de la simple erreur dans le résultat du programme au blocage total du système.

Avec Java, rien de tout cela n'est possible. Lorsque vous créez un objet, vous ne savez pas du tout où Java le place en mémoire. Tant que vous le tenez, vous pouvez l'utiliser. Si vous le lâchez, Java ne vous le rendra que si vous êtes capable de lui indiquer de façon non équivoque de quel objet il s'agit. Vous ne pouvez pas dire : "Je voudrais afficher le numéro matricule de l'employé que j'ai créé il y a cinq minutes à peine." En revanche, vous pouvez le faire au moment de la création, lorsque vous tenez en quelque sorte l'objet en main.

Pour faire référence à un membre d'un objet, il suffit d'indiquer le nom de l'objet suivi du nom du membre, en les séparant à l'aide d'un point. (Rappelons que nous connaissons pour l'instant deux sortes de membres : les *variables* et les *méthodes*.)

Modifiez le programme **Test.java** de la façon suivante :

```
public class Test2 {
   public static void main(String[] argv) {
      (new Employe()).afficherMatricule();
   }
}
```

```
class Employe {
   int matricule;
   static int nombre;

   Employe() {
      matricule = ++nombre;
   }

   void afficherMatricule() {
      System.out.println(matricule);
   }
}
```

et enregistrez-le dans le fichier **Test2.java**. Compilez-le et exécutez-le. Cette fois, vous devez obtenir le résultat suivant :

```
1
```

C'est-à-dire l'affichage du matricule de l'employé créé par votre programme. C'est plutôt rassurant. Nous voilà maintenant certains qu'un objet, instance de la classe **Employe**, a bien été créé. Sommes-nous, pour autant, plus avancés ? Pas vraiment. En effet, dès que l'exécution de la ligne :

```
(new Employe()).afficherMatricule();
```

est terminée, l'objet n'est plus accessible. Il se trouve toujours dans la mémoire, mais vous l'avez lâché et n'avez plus aucun moyen de le rattraper.

Il arrive souvent que l'on crée des objets de cette façon, que nous appellerons *anonyme*. Par exemple, si vous disposez d'un objet appelé **rectangle** et que cet objet possède une méthode **setColor** prenant pour paramètre une couleur, afin de s'afficher dans cette couleur, vous pourrez employer la syntaxe suivante :

```
rectangle.setColor(new Color(255, 0, 0));
```

pour afficher le rectangle en rouge. L'instruction **new Color(255, 0, 0)** crée un objet, instance de la classe **Color**, avec pour composante Rouge Vert et Bleu les valeurs 255, 0 et 0, et passe cet objet à la méthode **setColor** de l'objet rectangle. L'objet peut rester accessible ou non, suivant ce qu'en fait l'objet **rectangle**.

Dans le cas de notre programme **Test2**, une fois la méthode **afficherMatricule()** exécutée, l'objet **Employe** n'est plus disponible. Il continue d'exister un certain temps en mémoire avant d'être éliminé. Comment ? C'est là une des particularités de Java.

La destruction des objets : le *garbage collector*

Avec certains langages, le programmeur doit s'occuper lui-même de libérer la mémoire en supprimant les objets devenus inutiles. C'est une des principales sources de mauvais fonctionnement des programmes. En effet, il est fréquent que des parties de la mémoire restent occupées par des objets dont la vie est terminée. La tâche qui incombe au programmeur pour s'assurer que ce n'est jamais le cas est très importante, alors que les conséquences du problème paraissent minimes. Quelques octets de plus ou de moins, ce n'est pas ça qui va faire une grosse différence. Malheureusement, il y a là un phénomène cumulatif. Petit à petit, le nombre d'objets inutilisés restant en mémoire augmente et des troubles commencent à apparaître. Ce phénomène (appelé *memory leaks* ou *fuites de mémoire*) est bien connu des utilisateurs car il concerne en tout premier lieu le programme qu'il utilise le plus souvent : le système d'exploitation. Imaginez un programme de quelque quinze millions de lignes de code ! Combien de structures de données resteront, à un moment ou un autre, à occuper inutilement de la mémoire alors qu'elles ne sont plus utilisées ? Il est donc nécessaire de tout remettre à zéro régulièrement en redémarrant le système. Il en est de même avec certains programmes d'application, par exemple le traitement de texte que vous utilisez. Laissez-le en mémoire en permanence et travaillez chaque jour sur de gros documents. Il finira irrémédiablement par se "planter". Les programmeurs qui l'ont conçu ont probablement estimé que la grande ma-

jorité des utilisateurs quitteraient le programme chaque soir pour le recharger le lendemain, ce qui aurait pour effet de libérer la mémoire.

Avec Java, le problème est résolu de façon très simple : un programme, appelé *garbage collector*, ce qui signifie littéralement "ramasseur d'ordures", est exécuté automatiquement dès que la mémoire disponible devient inférieure à un certain seuil. De cette façon, vous pouvez être assuré qu'aucun objet inutilisé n'encombrera la mémoire au point d'être une cause de problèmes.

Certains langages plus anciens possédaient déjà un garbage collector. Cependant, il présentait généralement un inconvénient majeur. Le processeur ne pouvant exécuter qu'un seul programme à la fois, le programme principal devait être arrêté pendant l'opération de nettoyage de la mémoire. Java étant un langage *multithread*, c'est-à-dire capable d'exécuter plusieurs processus simultanément, la mise en action du garbage collector n'arrête pas le programme principal mais entraîne simplement un ralentissement.

Cela supprime-t-il tous les problèmes ? Malheureusement pas tous, mais une grande partie. La plupart des programmes modernes étant interactifs, c'est-à-dire attendant des actions de l'utilisateur, le garbage collector peut s'exécuter en tâche de fond, sans que l'utilisateur ne remarque quoi que ce soit. En effet, à la vitesse à laquelle celui-ci utilise la souris, ou même le clavier, il reste au processeur suffisamment de temps entre chaque bouton cliqué ou chaque touche pressée pour faire autre chose.

En revanche, pour les programmes devant s'exécuter en *temps réel*, un ralentissement peut poser un problème. Le garbage collector étant un processus asynchrone (c'est-à-dire dont il n'est pas possible de contrôler précisément la synchronisation avec d'autres processus), Java n'est normalement pas adapté à ce type de programme. Cependant, il existe déjà des versions spéciales de Java équipées d'un garbage collector synchrone qui peuvent résoudre ce type de problème au prix, évidemment, d'une plus grande complexité, puisque le programmeur doit prendre en charge la synchronisation. Nous reviendrons, dans un chapitre ultérieur, sur le fonctionnement du garbage collector.

Comment retenir les objets : les *handles*

Nous avons donc vu que nous pouvions créer un objet facilement à l'aide de l'opérateur **new** et du constructeur de l'objet. Nous pouvons utiliser immédiatement l'objet ainsi créé. Si nous voulons pouvoir l'utiliser plus tard, il faut, avant de confier cet objet à Java, s'assurer que nous pourrons le récupérer. Pour cela, il suffit de pouvoir l'identifier. Dans le langage courant, il est fréquent d'utiliser des artifices pour identifier les objets. Si vous avez un chien, vous pouvez le désigner par "mon chien". Si vous avez un chien noir et un chien blanc, vous pourrez les identifier par "mon chien noir" et "mon chien blanc". Mais parviendrez-vous à vous faire obéir en disant "Mon chien blanc, couché !". Il est beaucoup plus simple de leur donner à chacun un identificateur unique. Dans ce cas précis, il s'agira d'un *nom*. Ainsi vous pourrez, par exemple, appeler votre chien noir "Médor" et votre chien blanc "Azor" et leur donner des ordres individuels. (Toute comparaison a ses limites, et vous objecterez peut-être qu'il est parfaitement possible de dresser un chien pour qu'il obéisse à un ordre tel que "Mon chien blanc, couché !" et pas à "Mon chien noir, couché !". C'est tout simplement que votre chien aura appris que son nom est "Mon chien blanc".)

Dans le langage courant, les personnes, les animaux domestiques et certains objets (les bateaux, les villes, les pays, les fleuves, etc.) sont identifiés à l'aide d'un nom propre. D'autres objets sont identifiés par un moyen différent : numéro d'immatriculation, titre (pour une œuvre littéraire ou musicale), etc. En Java, tous les objets sont logés à la même enseigne : ils sont identifiés à l'aide d'un *handle*. Le mot *handle* signifie *poignée*, en français, ce qui est tout à fait évocateur, puisqu'il sert à manipuler l'objet correspondant. Cependant, il n'est pas entré dans le langage courant des programmeurs, et nous utiliserons donc le terme d'origine, au masculin, comme l'usage s'est établi, au risque de froisser les puristes de la langue française.

On assimile parfois le handle au nom de l'objet. Cette assimilation est incorrecte. En effet, la notion de nom évoque généralement l'unicité et la spécificité. On dit "mon nom" et non "mes noms", alors qu'il est évident que nous en avons plusieurs. Par ailleurs, nous pensons comme si notre nom nous était spécifique, ce qui n'est pas le cas (nous avons des homonymes). De plus, nous avons du mal à imaginer que chaque objet (une cuiller, une aiguille à coudre, une pomme, une idée, etc.) puisse avoir un nom.

En Java, les objets peuvent avoir plusieurs handles. Autant que vous le souhaitez. En revanche, un handle ne peut correspondre qu'à un seul objet simultanément. Il n'y a pas d'homonyme, ou plutôt, il n'y a pas de situation dans laquelle l'homonymie crée une ambiguité (sauf, évidemment, dans l'esprit du programmeur !). Un handle peut toutefois parfaitement changer d'objet. Dans la terminologie de Java, on dit que le handle *pointe* vers l'objet correspondant. Il est donc possible de modifier l'affectation d'un handle pour le faire pointer vers un autre objet. Il faut simplement que le type du nouvel objet soit compatible avec le handle. En effet, chaque fois que vous créerez un handle, vous devrez indiquer vers quel type d'objet il peut pointer.

Création des handles

Rien n'est plus simple que de créer un handle d'objet. Il suffit pour cela de le déclarer en indiquant vers quelle classe d'objets il est susceptible de pointer. Par exemple, pour créer le handle **alpha** pouvant pointer vers une instance d'**Employe**, nous écrirons :

```
Employe alpha;
```

Vers quoi pointe ce handle ? Pour l'instant, vers rien. En effet, nous avons *déclaré* le handle, ce qui suffit à le faire exister. En revanche, nous ne l'avons pas *initialisé*, c'est-à-dire que nous ne lui avons pas attribué sa valeur initiale. Pour l'instant, il ne pointe donc vers rien.

Attention : Notre apprentissage se fait petit à petit. Il faut donc commencer par des approximations. Nous verrons un peu plus loin que, dans certains cas, les handles sont automatiquement initialisés par Java.

Modifier l'affectation d'un handle

Nous allons aborder ici un des pièges de Java. Pour modifier l'affectation d'un handle, c'est-à-dire pour le faire pointer vers un objet (ou vers un autre objet, s'il était déjà *initialisé*), il faut utiliser un opérateur. Il aurait été naturel d'utiliser un opérateur tel que **pointe vers**, ou **-->**, mais malheureusement, les concepteurs de Java, qui étaient évidemment des programmeurs, ont préféré propager l'usage établi du signe égal. Nous allons voir que cet

usage est totalement incohérent. De plus, il nous oblige à utiliser un autre opérateur pour la vraie égalité, ce qui est tout de même un comble !

Nous avons vu qu'un objet pouvait être créé à l'aide de son constructeur et de l'opérateur **new**. Nous ne pouvons affecter un premier handle à un objet nouvellement créé qu'au moment de sa création. En effet, si nous écrivons les lignes :

```
Employe alpha;
new Employe();
```

il n'y a ensuite aucun moyen d'établir un lien entre le handle créé à la première ligne et l'objet créé à la seconde. Il faut donc effectuer l'affectation en même temps que la création, de la façon suivante :

```
Employe alpha;
alpha = new Employe();
```

Ce qui ne signifie en aucun cas que le handle **alpha** soit égal à l'objet créé, mais simplement qu'il pointe vers cet objet. Il pourra donc servir à manipuler cet objet.

Java nous permet même de condenser ces deux lignes en une seule sous la forme :

```
Employe alpha = new Employe();
```

Si nous utilisons cette technique (sous l'une de ces deux formes) dans notre programme, nous pouvons obtenir le même résultat tout en conservant la possibilité de réutiliser notre objet :

```
public class Test3 {
   public static void main(String[] argv) {
      Employe alpha = new Employe();
      alpha.afficherMatricule();
```

```
      alpha.afficherMatricule();
   }
}

class Employe {
   int matricule;
   static int nombre;

   Employe() {
      matricule = ++nombre;
   }

   void afficherMatricule() {
      System.out.println("Matricule " + matricule);
   }
}
```

Modifiez le programme **Test2.java** comme indiqué dans le listing ci-dessus et enregistrez-le dans le fichier **Test3.java**. (Nous avons introduit d'autres modifications pour améliorer la sortie. Nous reviendrons plus loin sur leur signification.)

Compilez et exécutez ce programme. Vous pouvez constater qu'il nous a été possible d'afficher deux fois le matricule pour un même objet, ce qui aurait été tout à fait impossible sans l'utilisation d'un handle. Il nous aurait évidemment été possible d'utiliser deux fois la ligne :

```
(new Employe()).afficherMatricule();
(new Employe()).afficherMatricule();
```

mais le résultat n'aurait pas du tout été le même. Si vous n'êtes pas convaincu, essayez ! Vous obtiendrez le résultat suivant :

```
Matricule 1
Matricule 2
```

En effet, la deuxième ligne crée un deuxième objet, différent du premier, et qui reçoit donc le numéro matricule suivant.

Résumé

Dans ce chapitre, vous avez appris comment les objets sont créés et comment on peut utiliser des handles pour les manipuler. Au chapitre suivant, nous approfondirons notre étude des handles, après avoir présenté les *primitives*, que nous avons d'ailleurs déjà utilisées dans nos exemples. Vous devez vous rappeler les choses suivantes :

- Les objets sont des instances de classes.
- Les objets peuvent être créés à volonté à l'aide de l'opérateur **new** et des constructeurs.
- Un handle peut être créé en le déclarant, c'est-à-dire en indiquant son nom, précédé du nom de la classe des objets auxquels il peut être affecté.
- Un handle qui n'a pas reçu d'affectation pointe vers **null**.
- Pour faire pointer un handle vers un objet, on utilise l'opérateur = en plaçant, à gauche, le handle, et à droite du signe = une référence à l'objet vers lequel il doit pointer.

Exercice

A titre d'exercice, essayez de reconstruire le programme **Test3.java** sans consulter le livre. Modifiez la méthode **main()** pour créer deux instances d'**Employe** ayant chacune un handle différent. Appelez la méthode **afficherMatricule()** pour afficher le matricule de chaque instance. Affectez ensuite le handle du deuxième objet au premier, puis essayez de nouveau d'afficher les matricules des deux objets. Est-ce possible ? Qu'est devenu le deuxième objet ? Les réponses sont dans le chapitre suivant.

Chapitre 4

Les primitives et les handles

Au chapitre précédent, nous avons commencé l'étude des handles. Nous avons également utilisé sans les étudier en détail des éléments d'un type nouveau permettant de manipuler des nombres entiers. Avant de poursuivre notre étude des handles, nous devons nous arrêter sur ces éléments différents que l'on appelle *primitives*. Mais, pour comprendre la nécessité des primitives, il faut dire quelques mots de la façon dont les données sont conservées en mémoire.

Les données manipulées par un programme peuvent résider dans les *registres* du processeur. Avec certains langages (l'assembleur, par exemple), il est possible de manipuler les registres du processeur. Ce n'est pas le cas en Java.

Les données peuvent résider dans une zone de mémoire spéciale que l'on appelle *pile (stack)*. Il s'agit d'une structure de données dans laquelle le dernier élément entré est le premier disponible. Cette structure fonctionne un peu comme un distributeur de bonbons, constitué d'un tube dans lequel se trouve un ressort. On charge le distributeur en plaçant les bonbons dans le tube, ce qui a pour effet de comprimer le ressort. Le dernier bonbon entré se trouve au niveau de l'ouverture du tube. Si on enlève un bonbon, le ressort pousse ceux qui restent vers le haut afin que le bonbon suivant soit disponible. Dans un ordinateur, la pile fonctionne de façon semblable, excepté qu'au lieu de déplacer tous les éléments vers le bas lorsque l'on ajoute un élément, c'est le haut de la pile qui se déplace vers le haut. (Cette façon de faire est beaucoup plus efficace car on n'a pas à déplacer tous les éléments.) Certains langages de programmation permettent au programmeur de manipuler la pile, et en particulier de la faire déborder dans d'autres structures de données, ce qui peut avoir des conséquences graves. Ce n'est pas le cas de Java. La pile est une structure performante, mais qui présente l'inconvénient de ne pouvoir être utilisée sans en connaître la taille.

Les données peuvent également résider dans une autre structure, que l'on appelle *tas (heap)*. Le tas est moins performant que la pile, mais sa taille est dynamique. Il n'est pas nécessaire de la connaître pour l'utiliser. Elle s'étend au fur et à mesure des besoins tant que la mémoire est disponible. Tous les objets Java sont placés dans cette structure, de façon transparente pour l'utilisateur. Leur adresse n'a pas à être connue du programmeur puisque le système des handles permet de manipuler les objets. Le programmeur n'a donc pas à s'occuper de l'allocation de la mémoire, ce qui réduit considérablement les risques d'erreurs. Si un objet voit sa taille croître, il peut être déplacé à volonté par Java sans que cela change quoi que ce soit pour le programmeur. C'est donc une structure souple, sûre, mais peu performante.

Les données peuvent également être placées dans des structures de données moins transitoires, voire quasi permanentes. Il s'agit, par exemple, des fichiers enregistrés sur disque, des zones de mémoire morte, etc. Contrairement aux autres structures de données, celles-ci persistent après l'arrêt du programme.

Les primitives

Nous avons dit au chapitre précédent que tout, en Java, était objet, c'est-à-dire instance de classes. En fait, ce n'est pas tout à fait vrai. Comme nous venons de le voir, les objets sont toujours créés dans une zone de mémoire dont les performances ne sont pas optimales. Java possède une classe permettant de créer des objets de type "nombre entier". Il s'agit de la classe **Integer**. Si nous voulons effectuer une même opération 1000 fois, par exemple afficher un compteur comptant le nombre de fois que l'opération est effectuée, nous pouvons le faire de la façon suivante :

- Créer un objet nombre entier et lui donner la valeur 0.
- Tant que la valeur du nombre entier est différente de 1000, augmenter cette valeur de 1 et l'afficher.

D'une part, utiliser pour cela un objet de type **Integer** (la classe Java utilisée pour représenter les nombres entiers) serait particulièrement peu efficace, car il faut accéder 1 000 fois à cet objet. D'autre part, cet objet ne sert à rien d'autre qu'à compter le nombre d'itérations et à afficher sa valeur. Et encore s'agit-il là d'un exemple dans lequel la valeur de la variable est utilisée pour l'affichage. Dans la plupart des exemples réels, ce type de variable ne sert que de compteur. Aucune des caractéristiques spécifiques d'un objet n'est utilisée ici. Il en est de même pour la plupart des calculs numériques.

Les concepteurs de Java ont donc doté ce langage d'une série d'éléments particuliers appelés *primitives*. Ces éléments ressemblent à des objets, mais ne sont pas des objets. Ils sont créés de façon différente, et sont également manipulés en mémoire de façon différente. Cependant, ils peuvent être *enveloppés* dans des objets spécialement conçus à cet effet, et appelés *enveloppeurs (wrappers)*.

Java dispose des primitives suivantes :

Primitive	*Étendue*	*Taille*
char	0 à 65 535	16 bits
byte	-128 à +127	8 bits
short	-32 768 à +32 767	16 bits
int	- 2 147 483 648 à + 2 147 483 647	32 bits
long	de - 2^{63} à (+ 2^{63} - 1), soit de - 9 223 372 036 854 775 808 à + 9 223 372 036 854 775 807	64 bits
float	de ± 1.4E-45 à ± 3.40282347E38	32 bits
double	de ± 4.9E-324 à ± 1.7976931348623157E308	64 bits
boolean	true ou false	1 bit
void	-	0 bit

Java dispose également des classes suivantes pour envelopper les primitives :

Classe	*Primitive*
Character	**char**
Byte	**byte**
Short	**short**
Integer	**int**
Long	**long**

Classe	*Primitive*
Float	**float**
Double	**double**
Boolean	**boolean**
Void	**void**
BigInteger	**-**
BigDecimal	**-**

Les classes **BigInteger** et **BigDecimal** sont utilisées pour représenter respectivement des valeurs entières et décimales de précision quelconque. Il n'existe pas de primitives équivalentes.

Notez que le type **char**, servant à représenter les caractères, est un type non signé sur 16 bits, conformément au standard UNICODE. (Il n'existe pas d'autre type numérique non signé.) En revanche, contrairement à l'usage dans d'autres langages, le type **boolean** n'est pas un type numérique.

Note : A l'inverse de ce qui se passe avec les autres langages, la taille des différentes primitives est toujours la même en Java, quel que soit l'environnement. Sur tous les ordinateurs, du plus petit PC au super-calculateur, un **int** ou un **float** feront toujours 32 bits et un **long** ou un **double** feront toujours 64 bits. C'est une des façons d'assurer la portabilité des programmes.

Utiliser les primitives

Les primitives sont utilisées de façon très simple. Elles doivent être déclarées, tout comme les handles d'objets, avec une syntaxe similaire, par exemple :

```
int i;
char c;
double valeurFlottanteEnDoublePrécision;
boolean fini;
```

Comme les handles d'objets, il n'est pas nécessaire de leur affecter une valeur avant de les utiliser. Elles doivent donc être *initialisées*. Si vous n'initialisez pas une primitive, vous obtiendrez un message d'erreur. Par exemple, la compilation du programme suivant :

```
public class primitives {

   public static void main(String[] argv) {
      int i;
      char c;
      double valeurFlottanteEnDoublePrécision;
      boolean fini;
      System.out.println(i);
      System.out.println(c);
      System.out.println(valeurFlottanteEnDoublePrécision);
      System.out.println(fini);
   }
}
```

produit quatre messages d'erreur :

```
primitives.java:7: Variable i may not have been initialized.
               System.out.println(i);
                                  ^
primitives.java:8: Variable c may not have been initialized.
               System.out.println(c);
                                  ^
primitives.java:9: Variable valeurFlottanteEnDoublePrécision may
                                       not have been initialized.
            System.out.println(valeurFlottanteEnDoublePrécision);
                                  ^
```

```
primitives.java:10: Variable fini may not have been initialized.
               System.out.println(fini);
                                  ^
4 errors
```

Pour initialiser une primitive, on utilise le même opérateur que pour les objets, c'est-à-dire le signe =. Cette utilisation du signe égal est moins choquante que dans le cas des objets, car il s'agit là de rendre la variable égale à une valeur, par exemple :

```
int i;
char c;
double valeurFlottanteEnDoublePrécision;
boolean fini;
i = 12;
c = "a";
valeurFlottanteEnDoublePrécision = 23456.3456;
fini = true;
```

Comme dans le cas des handles d'objets, il est possible d'effectuer la déclaration et l'initialisation sur la même ligne :

```
int i = 12;
char c = "a";
double valeurFlottanteEnDoublePrécision = 23456.3456;
boolean fini = true;
```

Il faut noter que le compilateur peut parfois déduire du code une valeur d'initialisation, mais c'est assez rare. Considérez l'exemple suivant :

```
class Initialisation {
   public static void main(String[] args) {
      int b;
      if (true) {
         b = 5;
```

```
      }
      System.out.println(b);
   }
}
```

Nous n'avons pas encore étudié l'instruction **if**, mais sachez simplement qu'elle prend un argument placé entre parenthèses. Si cet argument est vrai, le bloc suivant est exécuté. Dans le cas contraire, il est ignoré. Ici, si **true** est vrai, la variable **b** prend la valeur **5**. Or, **true** est toujours vrai. (**true** *signifie* vrai.)

Le compilateur est suffisamment intelligent pour s'en apercevoir. Ce code est donc compilé sans erreur. En revanche, si vous modifiez le programme de la façon suivante :

```
class Initialisation {
   public static void main(String[] args) {
      int b;
      boolean a = true;
      if (a) {
         b = 5;
      }
      System.out.println(b);

   }
}
```

le compilateur ne s'y retrouve plus et produit une erreur. Certains auteurs recommandent de toujours initialiser les variables au moment de leur déclaration afin d'éviter les erreurs. Ce n'est pas, à notre avis, un bon conseil. Il est évident qu'il faut initialiser les variables si leur valeur d'initialisation est connue. En revanche, si elle doit être le résultat d'un calcul, il est préférable de ne pas les initialiser (à 0, par exemple, pour les valeurs numériques) avant que le calcul soit effectué. En effet, si pour une raison ou une autre, vous oubliez d'initialiser une variable qui n'a pas été initialisée à 0, l'erreur sera détectée à la compilation. En revanche, si vous initialisez la variable à 0, le programme sera compilé sans erreur. Il est également tout à

fait possible qu'il ne produise pas d'erreur d'exécution. Simplement, le programme risque de donner un résultat incohérent (c'est un moindre mal) ou simplement faux. Ce type d'erreur peut parfaitement échapper à votre vigilance. Si, par chance, vous effectuez les vérifications qu'il faut pour vous apercevoir qu'il y a une erreur, rien ne vous indiquera d'où elle provient. Vous pouvez passer de longues heures de débogage, parfois après que votre programme aura été distribué à des millions d'exemplaires (on peut rêver), alors qu'en ayant pris soin de ne pas initialiser la variable, l'erreur aurait été détectée immédiatement.

Valeurs par défaut des primitives

Nous venons de voir que les primitives devaient être initialisées. Cela semble indiquer qu'elles n'ont pas de valeur par défaut. En fait, elles en reçoivent une dans certains cas. Lorsque des primitives sont utilisées comme membres d'une classe, et dans ce cas seulement, elles reçoivent une valeur par défaut au moment de leur déclaration. Pour le vérifier, vous pouvez compiler et exécuter le programme suivant :

```
public class primitives2 {
   public static void main(String[] argv) {
      PrimitivesInitialiseesParDefaut pipd =
           new PrimitivesInitialiseesParDefaut();
      System.out.println(pipd.i);
      System.out.println((int)pipd.c);
      System.out.println(pipd.valeurFlottanteEnDoublePrécision);
      System.out.println(pipd.fini);
   }
}

class PrimitivesInitialiseesParDefaut {
   int i;
   char c;
   double valeurFlottanteEnDoublePrécision;
   boolean fini;
}
```

Vous pouvez remarquer trois choses intéressantes dans ce programme. La première est que, pour accéder à un membre d'un objet, nous utilisons le nom de cet objet, suivi d'un point et du nom du membre. Il arrive souvent qu'un objet possède des membres qui sont eux-mêmes des objets possédant des membres, etc. Il suffit alors d'ajouter à la suite les noms des différents objets en les séparant à l'aide d'un point pour accéder à un membre. Par exemple, si l'objet **rectangle** contient un membre appelé **dimensions** qui est lui-même un objet contenant les membres **hauteur** et **largeur** qui sont des primitives, on peut accéder à ces primitives en utilisant la syntaxe :

```
rectangle.dimensions.hauteur
```

En revanche, si l'objet **rectangle** contient directement deux membres de type primitives appelés **hauteur** et **largeur**, on pourra y accéder de la façon suivante :

```
rectangle.hauteur
```

Note : Dans ce type d'expression, les références sont évaluées de gauche à droite, c'est-à-dire que :

```
alpha.bêta.gamma.delta
```

est équivalent à :

```
((alpha.bêta).gamma).delta
```

Il est parfaitement possible d'utiliser des parenthèses pour modifier l'ordre d'évaluation des références.

La deuxième chose qu'il est intéressant de noter est l'utilisation de **(int)** à la sixième ligne du programme. Nous reviendrons sous peu sur cette technique très importante, qui sert ici à transformer la valeur de type caractère en type numérique afin de pouvoir l'afficher. (Nous avons dit que le type char était un type numérique, ce qui est vrai, mais il est affiché par

System.out.println sous la forme du caractère UNICODE correspondant à sa valeur. Le code 0 ne correspond pas à un caractère affichable. Le résultat obtenu n'est donc pas significatif. C'est pourquoi nous affichons ici le code du caractère et non le caractère lui-même.)

La troisième chose intéressante dans ce programme est que nous n'avons pas fourni de constructeur pour la classe **PrimitivesInitialiseesParDefaut**. Ce n'est pas un problème, car dans ce cas, Java utilise un constructeur par défaut, qui ne fait rien.

Vous pouvez constater que notre programme est compilé sans erreur. Son exécution produit le résultat suivant :

```
0
0
0.0
false
```

Nos primitives ont donc bien été initialisées par défaut ! Toutes les primitives de type numérique utilisées comme membres d'un objet sont initialisées à la valeur **0**. Le type boolean est initialisé à la valeur **false** qui, rappelons-le, ne correspond à aucune valeur numérique.

Différences entre les objets et les primitives

Nous allons maintenant pouvoir approfondir les différences qui existent entre les objets et les primitives. Nous utiliserons pour cela un programme manipulant des primitives de type **int** et des objets de type **Entier**. **Entier** est un enveloppeur, c'est-à-dire une classe que nous créerons pour *envelopper* une primitive dans un objet. Saisissez le programme suivant :

```
public class primitives3 {
   public static void main(String[] argv) {
      System.out.println("Primitives :");
      int intA = 12;
      System.out.println(intA);
```

```
      int intB = intA;
      System.out.println(intB);
      intA = 48;
      System.out.println(intA);
      System.out.println(intB);

      System.out.println("Objets :");
      Entier entierA = new Entier(12);
      System.out.println(entierA);
      Entier entierB = entierA;
      System.out.println(entierB);
      entierA.valeur = 48;
      System.out.println(entierA);
      System.out.println(entierB);
   }
}

class Entier {
   int valeur;

   Entier(int v){
      valeur = v;
   }

   public String toString(){
      return ("" + valeur);
   }
}
```

Compilez et exécutez ce programme. Vous devez obtenir le résultat suivant :

```
Primitives :
12
12
48
12
```

```
Objets :
12
12
48
48
```

Examinons tout d'abord la classe **Entier**. Celle-ci comporte une variable de type **int**. Il s'agit de la primitive à envelopper. Elle comporte également un constructeur, qui prend pour paramètre une valeur de type **int** et initialise le champ **valeur** à l'aide de ce paramètre. Elle comporte enfin une méthode de type **String** qui ne prend pas de paramètres et retourne une représentation du champ **valeur** sous forme de chaîne de caractères, afin qu'il soit possible de l'afficher. La syntaxe utilisée ici est un peu particulière. Elle consiste à utiliser l'opérateur + avec une chaîne de caractères de longueur nulle d'un côté, et le champ valeur de l'autre, ce qui a pour conséquence de forcer Java à convertir **valeur** en chaîne de caractères et de l'ajouter à la suite de la chaîne de longueur nulle. Le résultat est donc une chaîne représentant **valeur**. Nous reviendrons en détail sur cette opération dans la section consacrée aux chaînes de caractères.

Venons-en maintenant à la procédure **main** de la classe **primitives3**. Tout d'abord, nous affichons un message indiquant que nous traitons des primitives :

```
System.out.println("Primitives :");
```

puis nous créons une variable de type **int**, que nous initialisons avec la valeur littérale 12 :

```
int intA = 12;
```

Nous affichons immédiatement sa valeur à la ligne suivante :

```
System.out.println(intA);
```

Le programme affiche donc 12.

Nous créons ensuite une nouvelle variable de type **int** et nous l'initialisons à l'aide de la ligne :

```
int intB = intA;
```

ce qui attribue à **intB** la valeur de **intA**. Nous affichons ensuite la valeur de intB :

```
System.out.println(intB);
```

et obtenons naturellement 12. Puis nous modifions la valeur de **intA** en lui attribuant une nouvelle valeur littérale, 48. Nous affichons ensuite **intA** et **intB** :

```
intA = 48;
System.out.println(intA);
System.out.println(intB);
```

Nous constatons que **intA** vaut bien maintenant 48 et que **intB** n'a pas changé et vaut toujours 12.

Ensuite, nous faisons la même chose avec des objets de type **Entier**. Nous affichons tout d'abord un message indiquant qu'à partir de maintenant, nous traitons des objets :

```
System.out.println("Objets :");
```

puis nous créons un objet, instance de la classe **Entier** :

```
Entier entierA = new Entier(12);
```

Le handle **entierA** est déclaré de type **Entier**, un nouvel objet est instancié par l'opérateur **new** et il est initialisé par le constructeur **Entier()** en utili-

sant la valeur littérale 12 pour paramètre. Le champ **valeur** de cet objet contient donc la valeur entière 12.

Nous le vérifions immédiatement à la ligne suivante en affichant l'objet **entierA** :

```
System.out.println(entierA);
```

Lorsque l'on essaie d'afficher un objet, Java exécute simplement la méthode **toString()** de cet objet et affiche le résultat. Ici, le programme affiche donc 12.

Nous créons ensuite un nouveau handle d'objet de type **Entier** et nous l'initialisons à l'aide de la ligne :

```
Entier entierB = entierA;
```

Puis nous affichons l'objet **entierB** :

```
System.out.println(entierB);
```

ce qui affiche 12.

A la ligne suivante, nous modifions la valeur de **entierA**, puis nous affichons les valeurs de **entierA** et **entierB**.

```
entierA.valeur = 48;
System.out.println(entierA);
System.out.println(entierB);
```

Nous constatons alors que **entierA** a bien pris la valeur 48, mais que **entierB** a également pris cette valeur.

Au point où nous en sommes, cela ne devrait pas vous surprendre. En effet, la différence entre objet et primitive prend toute sa signification dans les lignes :

```
int intB = intA;
```

et :

```
Entier entierB = entierA;
```

La première signifie :

> "Créer une nouvelle primitive de type **int** appelée **intB** et l'initialiser avec la valeur de **intA**."

alors que la deuxième signifie :

> "Créer un nouveau handle de type **Entier** appelé **entierB** et le faire pointer vers le même objet que le handle **entierA**."

Dans le premier cas, nous créons une nouvelle primitive, alors que dans le second, nous ne créons pas un nouvel objet, mais seulement un nouveau "nom" qui désigne le même objet. Puisqu'il n'y a qu'un seul objet, il ne peut avoir qu'une seule valeur. Par conséquent, si nous changeons la valeur de l'objet vers lequel pointe le handle **entierA**, nous changeons aussi la valeur de l'objet pointé par **entierB**, puisqu'il s'agit du même !

Pour simplifier, et bien que cela ne soit pas la réalité, vous pouvez considérer que le handle n'est qu'une étiquette pointant vers un objet qui peut avoir une valeur (c'est vrai), alors que la primitive **est** une valeur (en fait, c'est faux, mais cela peut être considéré comme vrai dans une certaine mesure du point de vue du programmeur Java).

A partir de maintenant, nous utiliserons de façon générique le terme de *variable* pour faire référence à des primitives ou à des handles d'objets. Vous devrez bien garder à l'esprit le fait que deux variables différentes auront des valeurs différentes, alors que deux handles différents pourront éventuellement pointer vers le même objet.

Les valeurs littérales

Nous avons vu au chapitre précédent qu'il pouvait exister des objets anonymes, c'est-à-dire vers lesquels ne pointe aucun handle. Existe-t-il également des primitives anonymes ? Pas exactement, mais l'équivalent (très approximatif) : les valeurs littérales. Alors que les primitives sont créées et stockées en mémoire au moment de l'exécution du programme (leur valeur n'est donc pas connue au moment de la compilation), les valeurs littérales sont écrites en toutes lettres dans le code source du programme. Elles sont donc traduites en bytecode au moment de la compilation et stockées dans le fichier **.class** produit par le compilateur. Elles ne peuvent évidemment être utilisées qu'à l'endroit de leur création, puisqu'il n'existe aucun moyen d'y faire référence. (Elles présentent une différence fondamentale avec les objets anonymes qui, eux, n'existent que lors de l'exécution du programme, et non lors de la compilation.)

Des valeurs littérales correspondent à tous les types de primitives. Pour les distinguer, on utilise cependant un procédé totalement différent. Le tableau ci-dessous donne la liste des syntaxes à utiliser :

Primitive	*Syntaxe*	
char	'x'	
	'\--'	où -- représente un code spécifique :
		\b arrière (backspace)
		\f saut de page (form feed)
		\n saut de ligne (new line)
		\r retour chariot (carriage return)
		\t tabulation horizontale (h tab)
		\\ \ (backslash)
		\' guillemet simple
		\" guillemet double
		\0oo caractère dont le code est oo en octal (Le 0 est facultatif.)
		\uxxxx caractère dont le code Unicode est xxxx (en hexadécimal)

Primitive	*Syntaxe*
int	5 (décimal), 05 (octal), 0x5 (hexadécimal)
long	5L, 05L, 0x5L
float	5.5f ou 5f ou 5.5E5f
double	5.5 ou 5.5d ou 5.5E5 ou 5.5E5
boolean	**false** ou **true**

Notes : Les valeurs flottantes ne peuvent être exprimées qu'en décimal.

Ces syntaxes ne présentent un intérêt que lorsqu'il y a une ambiguïté possible. Ainsi, si vous écrivez :

```
long i = 55;
```

il n'y a aucune ambiguïté et il n'est pas utile d'ajouter le suffixe **L**. Par ailleurs, vous pouvez écrire :

```
byte i = 5;
short j = 8;
```

Bien que les valeurs littérales utilisées ici soient de type **int**, cela ne pose pas de problème au compilateur, qui effectue automatiquement la conversion. Nous verrons plus loin par quel moyen. (Notez, au passage, que les littéraux sont évalués lors de la compilation, de façon à éviter au maximum les erreurs lors de l'exécution.)

Vous ne pouvez cependant pas écrire :

```
byte i = 300;
```

car la valeur littérale utilisée ici est supérieure à la valeur maximale qui peut être représentée par le type **byte**.

Vous pouvez écrire :

```
float i = 5
```

car 5 est de type **int** et le compilateur est capable de le convertir automatiquement en type **float**. En revanche, vous ne pouvez pas écrire :

```
float j = 5.5
```

car 5.5 est de type **double**. Dans ce cas, Java n'effectue pas automatiquement la conversion, pour une raison que nous verrons plus loin.

Le *casting* des primitives

Le *casting* (mot anglais qui signifie *moulage*), également appelé *cast* ou, parfois, *transtypage*, consiste à effectuer une conversion d'un type vers un autre type. Le casting peut être effectué dans deux conditions différentes :

- Vers un type plus général. On parle alors de *sur-casting* ou de *sur-typage*.
- Vers un type plus particulier. On parle alors de *sous-casting* ou de *sous-typage*.

Dans le cas des primitives, le sur-casting consiste à convertir vers un type dont la précision est supérieure. C'est le cas, par exemple, de la conversion d'un **byte** en **short**, d'un **int** en **long** ou d'un **float** en **double**. On parlera en revanche de sous-casting lors de la conversion d'un type vers un autre de précision inférieure, par exemple de **double** en **float**, de **long** en **int** ou de **short** en **byte**. A l'inverse du sur-casting, le sous-casting présente un risque de perte d'informations. C'est pourquoi Java est autorisé à effectuer de fa-

çon *implicite* un sur-casting, alors qu'un sous-casting doit être demandé *explicitement* par le programmeur.

Le cas du type **char** est particulier. En effet, il s'agit d'un type numérique sur 16 bits non signé. C'est pourquoi il peut être l'objet d'un casting vers un autre type numérique. Cependant, du fait qu'il est non signé, le casting d'un **char** en **short** constitue un sous-casting, bien que les deux valeurs soient représentées sur 16 bits. Il doit donc être demandé explicitement. Ainsi, vous pouvez écrire :

```
int i = 'x';
```

mais pas :

```
short i = 'x';
```

Le casting explicite

Un casting explicite peut être effectué simplement en faisant précéder la valeur par l'indication du nouveau type entre parenthèses, par exemple :

```
1: float a = (float)5.5;
2: short b = (short)'x';
3: double c = (double)5;
4: float d = (float)5;
5: byte f = (byte)300;
```

A la première ligne, la valeur littérale 5.5, de type **double**, est sous-castée (excusez le barbarisme !) vers le type **float**. Il n'y a pas de perte de données, car la valeur 5.5 peut être valablement représentée par un **float**.

A la ligne suivante, le caractère **'x'** est sous-casté vers le type **short**. Il n'y pas de perte de données. Il n'y en a jamais avec les caractères ANSI utilisés dans les pays occidentaux, car leurs codes sont représentés sur 8 bits. Leur valeur est donc limitée à 255, alors que le type **char** peut représenter des valeurs de 0 à 65 535 et le type short de -32 768 à +32 767.

A la troisième ligne, la valeur 5 (de type **int**) est sur-castée vers le type **double**. Ici, le casting explicite n'était donc pas obligatoire. (On peut toujours effectuer un casting explicite, même lorsqu'un casting implicite suffit.)

A la quatrième ligne, la valeur 5, de type **int**, est castée vers le type **float**. Ici non plus, un casting explicite n'était pas obligatoire, mais pour une raison que nous allons développer dans un instant.

A la cinquième ligne, la valeur de type **int** 300 est castée explicitement vers un **byte**. La valeur maximale que peut contenir le type **byte** étant de +127, il y a perte de données. Après ce cast, si vous affichez la valeur de **f**, vous obtiendrez 44. La raison en est que le nombre 300, exprimé en binaire, est tronqué à gauche au huitième bit. De plus, le huitième bit indique le signe (en notation dite *complément à 2*). Le résultat, quoique calculable, est assez aléatoire !

Casting d'une valeur entière en valeur flottante

Le cas du casting d'une valeur entière en valeur flottante est particulier. En effet, ce n'est pas la valeur maximale qui est prise en considération ici, mais la précision. De façon surprenante, Java traite le problème un peu à la légère en autorisant les casts implicites d'une valeur entière vers une valeur flottante. Cela peut se comprendre du fait que la variation de valeur est limitée à la précision de la représentation. (Il ne peut y avoir de surprise comme dans le cas du casting d'**int** en **byte**, où 128 devient -128.) Cependant, il y a quand même perte de données ! Après la ligne :

```
float i = 214748364794564L;
```

la variable **i** vaut :

```
2.14748365E14
```

soit :

```
214748365000000
```

ce qui constitue tout de même une perte d'information. En revanche, l'ordre de grandeur est conservé.

Attention : Si vous utilisez une valeur littérale trop élevée pour son type, vous obtiendrez un message d'erreur **Numeric overflow**. C'est le cas, par exemple, si vous essayez de compiler un programme contenant la ligne :

```
short i = 2147483640;
```

car la valeur **2147483640** dépasse la valeur maximale qui peut être représentée par un **short**. Par ailleurs, la ligne suivante produira la même erreur, bien que la valeur utilisée ici soit parfaitement représentable par un **long** :

```
long i = 21474836409;
```

Pouvez-vous deviner pourquoi ? La réponse se trouve à la fin de ce chapitre.

Casting d'une valeur flottante en valeur entière

Le casting d'une valeur flottante en valeur entière est exécuté par troncage (ou troncature) et non par arrondi. Pour obtenir une valeur arrondie à l'entier le plus proche, il faut utiliser la méthode **round()** de la classe **java.lang.Math**.

Formation des identificateurs

Les identificateurs Java sont formés d'un nombre quelconque de caractères. Le premier caractère doit être un *Identifier Start* (début d'identificateur). Les autres caractères doivent faire partie des *Identifier Parts*. Les majuscules et les minuscules sont distinguées. (**Longueur**, **longueur** et **LONGUEUR** sont trois identificateurs différents.) Vous ne pouvez pas utiliser comme identificateur un mot clé réservé de Java. (La liste des mots clés réservés est donnée à l'Annexe B.) Les caractères autorisés pour le début et la suite des identificateurs sont décrits dans le tableau ci-après :

Codes (décimal)	Caractères (en Times)	Début d'identificateur	Corps d'identificateur
0-8	Caractères de contrôle	Non	Oui
14-27	Caractères de contrôle	Non	Oui
36	$	Oui	Oui
48-57	0-9	Non	Oui
65-90	A-Z	Oui	Oui
95	_ (trait de soulignement)	Oui	Oui
97-122	a-z	Oui	Oui
127-159	• • • , *f* „ … † ‡ ˆ ‰ Š ‹ Œ • • • • ‘ ’ “ ” • – — ˜ ™ š › œ • • Ÿ	Non	Oui
162-165	¢ £ ¤ ¥	Oui	Oui
170	ª	Oui	Oui
181	µ	Oui	Oui
186	º	Oui	Oui
192-214	À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö	Oui	Oui
216-246	Ø Ù Ú Û Ü Ý Þ ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö	Oui	Oui
248-255	ø ù ú û ü ý þ ÿ	Oui	Oui

Note : Les ronds noirs (•) représentent des caractères non imprimables dans la police *Times*.

Attention : Les caractères de code 0-8 et 14-27 sont souvent impossibles à saisir dans les éditeurs de texte. Si vous voulez vérifier s'ils sont utilisables, il peut être nécessaire d'employer un programme spécial (éditeur hexadécimal). De toute façon, cela ne présente aucun intérêt pratique !

En plus des règles ci-dessus, il est conseillé de respecter les directives suivantes :

- Évitez les caractères accentués dans les noms de classes. Les noms de classes sont en effet utilisés de façon externe (en relation avec le système d'exploitation), ce qui peut poser des problèmes.

- Évitez de donner le même nom à des identificateurs d'éléments différents, bien que cela soit autorisé. Par exemple, évitez de donner à une méthode le même nom qu'une classe.

- N'utilisez pas le caractère $, qui est employé automatiquement par Java dans les noms de classes pour nommer les classes anonymes et les classe internes. (Au moins, ne l'utilisez pas dans les noms de classes.) Les classes internes et les classes anonymes seront étudiées dans un prochain chapitre.

Note : Si vous voulez savoir si un caractère est un *Indentifier Start* ou un *Identifier Part*, vous pouvez utiliser les méthodes statiques **isJavaIdentifierStart(char ch)** et **isJavaIdentifierPart(char ch)** de la classe **java.lang.Character**.

Portée des identificateurs

Pour faire référence collectivement aux noms de primitives et aux handles d'objets, ainsi qu'à tout ce qui permet d'identifier quelque chose (méthode, classe, etc.), on utilise le terme d'*identificateur*. A première vue, on peut considérer un identificateur comme le nom de l'élément qu'il identifie, mais on préfère le terme d'identificateur pour éviter la confusion avec l'usage courant dans le langage naturel du mot *nom*. (Et en particulier, le fait que le mot *nom* évoque l'unicité - on n’a souvent qu'un seul nom - alors que les objets peuvent avoir un nombre quelconque d'identificateurs.)

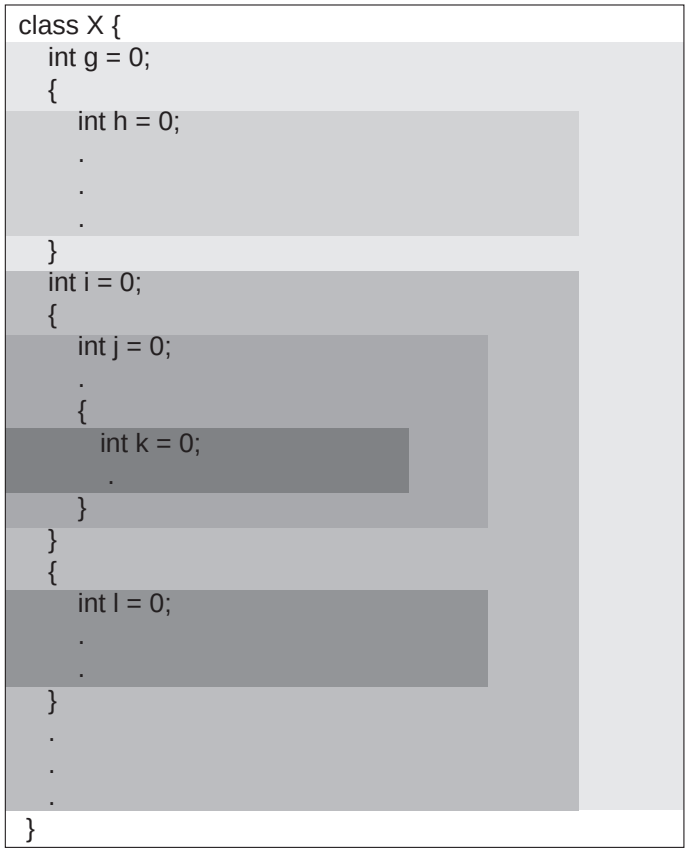
Il est utile de s'intéresser en détail aux conditions de vie des identificateurs. La vie d'un objet se conçoit souvent en termes de durée, ce qui semble pertinent. Un objet existe depuis le moment où il est créé, jusqu'au moment où le garbage collector le supprime. En revanche, ses identificateurs ont une vie tout à fait indépendante. Ils peuvent être créés avant ou après la création de l'objet, et être supprimés avant ou après la suppression de l'objet. La vie d'un identificateur ne se conçoit pas en termes de durée, mais en termes de visibilité dans le programme. Un identificateur est visible dans certaines parties du programme, et invisible dans d'autres. Lorsqu'il n'est pas visible, il peut être masqué, ou ne pas exister. L'étendue de programme dans laquelle un identificateur existe est appelée *portée* (*scope* en anglais). Le fait qu'il soit ou non visible est appelé tout simplement *visibilité*.

Nous avons vu précédemment que la quasi-totalité d'un programme Java se trouve incluse dans une structure faite de blocs emboîtés, délimités par les caractères **{** et **}**. La portée d'un identificateur (du moins en ce qui concerne les primitives et les instances d'objets) s'étend de l'endroit où il est créé à la fin du bloc dans lequel cette création a eu lieu, en incluant tous les blocs imbriqués.

Par exemple, dans le squelette de programme de la page suivante, les zones de différentes nuances de gris indiquent la portée des identificateurs.

Portée n'est pas synonyme de visibilité. En effet, considérez le programme suivant :

```
public class Visibilite {
   static char i = 'x';
   public static void main(String[] argv) {
      System.out.println(i);
      boolean i = false;
      System.out.println(i);
      test();
      System.out.println(i);
      System.out.println(Visibilite.i);
   }
```



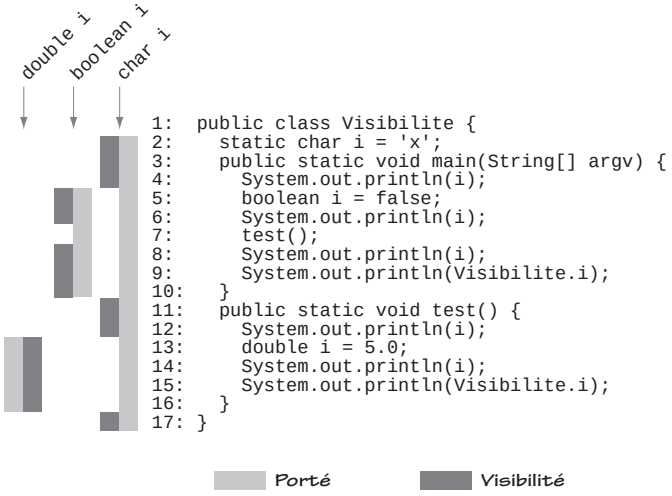

```
   public static void test() {
      System.out.println(i);
      double i = 5.0;
      System.out.println(i);
      System.out.println(Visibilite.i);
   }
}
```

Si vous compilez et exécutez ce programme, vous devez obtenir le résultat suivant :

```
x
false
x
5.0
x
false
x
```

L'identificateur **char i** et déclaré **static** de façon que l'on puisse y accéder en faisant référence à la classe (sous la forme **Visibilite.i**) et non en faisant référence à une instance de cette classe. La méthode **test()** est déclarée **static** pour la même raison.

La figure ci-après met en évidence la portée et la visibilité des identificateurs dans ce programme :



La portée de l'identificateur **char i** s'étend de la ligne 2 à la ligne 17. Nous le vérifions aux lignes 4, 9, 12 et 15.

La portée de l'identificateur **boolean i** s'étend de la ligne 5 à la ligne 10. Nous le vérifions aux lignes 6 et 8.

La portée de l'identificateur **double i** s'étend de la ligne 13 à la ligne 16. Nous le vérifions à la ligne 14.

La visibilité de l'identificateur **char i** est suspendue lorsque celui-ci est masqué par l'identificateur **boolean i**, à la ligne 5, et ce jusqu'à la ligne 10. L'identificateur **char i** est de nouveau visible aux lignes 11 et 12, puis il est masqué encore une fois de la ligne 13 à la ligne 16 par l'identificateur **double i**.

La visibilité de l'identificateur **boolean i** est suspendue à la ligne 11. En fait, la ligne 11 correspond à l'appel de la méthode **test()**. La visibilité de l'identificateur **boolean i** est suspendue sur toute l'étendue de la méthode **test()**.

La visibilité de l'identificateur **double i** s'étend sur toute sa portée. (Il n'est jamais masqué.)

Note 1 : *Visibilité* n'est pas synonyme d'*accessibilité*. Nous pouvons le constater aux lignes 9 et 15. Dans ces lignes, l'identificateur **char i** n'est pas visible, mais il est quand même accessible en faisant référence à la classe **Visibilite**. (Rappelons que cet identificateur est **static**, c'est-à-dire qu'il appartient à la classe et non à une instance de cette classe.) Nous reviendrons en détail au chapitre suivant sur la notion d'accessibilité.

Note 2 : Le masquage d'un identificateur n'est possible que :

- Dans des méthodes différentes. Ici, **char i** appartient à la classe **Visibilite**, **boolean i** à la méthode **main()** et **double i** à la méthode **test()**.

- Dans des blocs différents, à l'extérieur de toute méthode.

Il est possible de masquer un identificateur de la façon suivante :

```
char i = "x"
{
   boolean i = false;
}
```

à condition que ce morceau de code n'appartienne pas à une méthode. Ainsi, le programme suivant est compilé sans erreur :

```
public class Visibilite2 {
   static char i = 'x';
   {
      boolean i = false;
   }
   public static void main(String[] argv) {
      System.out.println(i);
   }
}
```

et son exécution affiche :

```
x
```

En revanche, la compilation du programme suivant :

```
public class Visibilite3 {
   public static void main(String[] argv) {
      char i = 'x';
      {
         boolean i = false;
      }
      System.out.println(i);
   }
}
```

produit le message d'erreur :

```
Visibilite3.java:5: Variable 'i' is already defined in this method.
               boolean i = false;
                       ^
1 error
```

Portée des identificateurs de méthodes

Contrairement aux handles et aux noms de primitives, la portée des identificateurs de méthodes s'étend à la totalité du bloc qui les contient. Ainsi, dans la classe **Visibilite**, la déclaration de la méthode **test()** est placée après son appel dans la méthode **main()**, ce qui ne produit pas d'erreur :

```
public class Visibilite {
   static char i = 'x';
   public static void main(String[] argv) {
      System.out.println(i);
      boolean i = false;
      System.out.println(i);
      test();
      System.out.println(i);
      System.out.println(Visibilite.i);
   }
   public static void test() {
      System.out.println(i);
      double i = 5.0;
      System.out.println(i);
      System.out.println(Visibilite.i);
   }
}
```

Cette écriture et la suivante sont parfaitement équivalentes :

```
public class Visibilite {
   public static void test() {
      System.out.println(i);
      double i = 5.0;
      System.out.println(i);
      System.out.println(Visibilite.i);
   }
   static char i = 'x';
   public static void main(String[] argv) {
      System.out.println(i);
```

```
        boolean i = false;
        System.out.println(i);
        test();
        System.out.println(i);
        System.out.println(Visibilite.i);
    }
}
```

Les objets n'ont pas de portée

Il faut bien garder à l'esprit le fait que seuls les identificateurs ont une portée. Dans le cas des primitives, l'identificateur pouvant être assimilé à l'objet identifié, cela ne fait pas beaucoup de différence pour le programmeur. En revanche, en ce qui concerne les objets, il en va différemment. Lorsqu'un handle est hors de portée, l'objet correspondant continue d'exister, et peut éventuellement être accessible grâce à un autre handle.

Les chaînes de caractères

En Java, les chaînes de caractères sont des objets. Ce sont des instances de la classe **String**. Cependant, leur nature nous oblige à en parler en même temps que des primitives.

Depuis les premiers langages de programmation, les chaînes de caractères ont posé des problèmes. En effet, s'il est facile de définir différents types numériques de format fixe, les chaînes de caractères ne peuvent être représentées dans un format fixe car leur longueur peut varier de 0 à un nombre quelconque de caractères. Plusieurs approches sont possibles :

- Obliger le programmeur à déclarer la longueur de chaque chaîne. Il est alors possible de réserver une zone de mémoire correspondant au nombre de caractères déclaré. Le programmeur est ensuite libre de modifier comme il le souhaite le contenu d'une chaîne, du moment qu'il ne la fait pas déborder. En cas de débordement, deux options sont possibles :

- Les données sont tronquées (avec ou sans message d'erreur).
- Les données débordent librement, ce qui conduit généralement à un plantage (au mieux, du programme, au pire, du système).

Cette méthode donne de bonnes performances, mais elle est peu souple, et dangereuse dans certains cas.

- Utiliser des chaînes de longueur dynamique. Les chaînes sont créées avec la longueur correspondant à leur valeur d'initialisation. Si leur valeur est modifiée, leur longueur est automatiquement adaptée. Cette technique est très souple, mais peu efficace en termes de performances.

Java utilise une approche particulière. Les chaînes de caractères peuvent être initialisées à une valeur quelconque. Leur longueur est choisie en fonction de leur valeur d'initialisation. En revanche, leur contenu ne peut plus être modifié. Cette formule peut paraître restrictive, mais elle présente plusieurs avantages. En effet, la longueur des chaînes étant assurée de ne jamais varier, leur utilisation est très efficace en termes de performances. Par ailleurs, la restriction concernant l'immuabilité de leur contenu ne tient pas pour trois raisons :

- Dans la plupart des programmes, de très nombreuses chaînes de caractères sont utilisées, entre autres pour l'interface utilisateur, comme des constantes. (Java ne possède pas de constantes à proprement parler, comme nous le verrons plus loin.) Les messages qui doivent être affichés par le programme sont le plus souvent manipulés sous forme de chaînes de caractères, et la plupart de ces messages ne changent jamais.
- Java dispose d'une autre classe, appelée **StringBuffer**, qui permet de gérer des chaînes dynamiques.
- Si le programmeur veut traiter des instances de la classe **String** comme des chaînes dynamiques, il le peut. Il y a simplement une petite précaution à prendre.

Les chaînes de caractères existent aussi sous forme littérale. Pour utiliser une chaîne littérale, il suffit de la placer entre guillemets, comme dans l'exemple suivant :

```
System.out.println("Message à afficher");
```

Attention : Vous possédez un PC, vous utilisez certainement un éditeur sous Windows pour écrire vos programmes, alors que l'interpréteur Java fonctionne dans une fenêtre DOS. L'éditeur utilise donc le code ANSI alors que l'interpréteur utilise le code ASCII "étendu". Les caractères accentués que vous saisirez dans votre éditeur ne seront donc probablement pas affichés de manière correcte. (Rappelons que le code ANSI est englobé dans le code UNICODE utilisé par Java pour représenter les caractères. Le code ANSI constitue les 256 premiers caractères (codes 0 à 255) du code UNICODE. En revanche, le code ASCII ne comporte que 128 caractères (codes 0 à 127) identiques au code ANSI ; les codes 128 à 255 font partie des *extensions* qui peuvent différer selon les pays ou les environnements. Il est toutefois possible de sélectionner une extension particulière grâce aux *pages de code* du DOS.)

Les chaînes de caractères littérales peuvent contenir tous les caractères spéciaux que nous avons décrits à propos du type **char** :

\b	arrière (backspace)
\f	saut de page (form feed)
\n	saut de ligne (new line)
\r	retour chariot (carriage return)
\t	tabulation horizontale (h tab)
\\	\ (backslash)
\'	guillemet simple
\"	guillemet double
\0oo	caractère dont le code est oo en octal (le 0 est facultatif)
\uxxxx	caractère dont le code Unicode est xxxx (en hexadécimal)

Il est intéressant d'adapter notre programme **Primitive3** pour tester le comportement des chaînes de caractères. Voici le programme modifié, que nous avons appelé **Chaines.java** :

```
public class Chaines {
   public static void main(String[] argv) {
      String chaineA = new String("Chaine 1");
      System.out.println(chaineA);
      String chaineB = chaineA;
      System.out.println(chaineB);
      chaineA = "Chaine 2";
      System.out.println(chaineA);
      System.out.println(chaineB);
   }
}
```

et voici le résultat obtenu :

```
Chaine 1
Chaine 1
Chaine 2
Chaine 1
```

Nous voyons que, bien qu'il s'agisse d'objets, les chaînes se comportent comme des primitives. Cela est dû au fait que les chaînes ne peuvent être modifiées. Ainsi, la ligne :

```
chaineA = "Chaine 2";
```

ne modifie pas la chaîne pointée par le handle **chaineA**, mais crée une nouvelle chaîne et lui attribue ce handle. Si, à ce moment, il n'existe pas d'autre handle pointant vers la chaîne d'origine, celle-ci devient inaccessible et disparaîtra de la mémoire au prochain démarrage du garbage collector. Dans notre cas, le handle **chaine2** pointe encore vers la chaîne et celle-ci continue donc d'exister normalement, en gardant sa valeur originale.

Note : Les chaînes littérales sont créées par Java sous forme d'instances de la classe **String**. Voilà pourquoi il est possible d'exécuter une instruction comme celle ci-dessus. En fait, il ne s'agit pas d'affecter à l'objet **chaineA** une nouvelle valeur comme on le ferait pour une primitive, mais de le faire

pointer vers l'objet **"Chaine 2"**. Chaque fois que vous placez une chaîne littérale dans votre programme, vous créez un objet instance de la classe **String**. Il s'agit, là encore, d'un objet anonyme, qui peut être utilisé immédiatement et devenir ensuite indisponible et futur client du garbage collector, comme dans l'instruction :

```
System.out.println("Message à afficher");
```

ou se voir affecté un handle, comme dans l'instruction :

```
String message = "Message à afficher";
```

Dans notre programme **Chaines**, il était donc tout à fait inutile d'écrire :

```
String chaineA = new String("Chaine 1");
```

ce qui entraîne la création de deux objets dont l'un, anonyme, est immédiatement abandonné. Il aurait été plus efficace d'écrire directement :

```
String chaineA = "Chaine 1";
```

Constantes

Java ne comporte pas de constantes à proprement parler. Il est cependant possible de simuler l'utilisation de constantes à l'aide du mot clé **final**. Une variable déclarée **final** ne peut plus être modifiée une fois qu'elle a été initialisée.

L'usage de constantes pour représenter des valeurs qui ne doivent pas être modifiées présente deux avantages par rapport aux variables. Tout d'abord, c'est un bon moyen d'éviter les erreurs. Si, par inadvertance, vous modifiez la valeur d'une primitive déclarée **final**, le compilateur produira un message

d'erreur. Cela vous évitera de futures erreurs d'exécution pas toujours faciles à détecter.

Par ailleurs, lorsque vous déclarez un élément **final**, le compilateur est à même d'optimiser le code compilé afin d'améliorer sa vitesse d'exécution.

Note : Il convient de remarquer que les variables déclarées **final** peuvent être initialisées lors de l'exécution, et non seulement lors de leur déclaration. Ainsi, elles ne sont constantes que pour une exécution et peuvent prendre une valeur différente à chaque exécution du programme (contrairement à ce qui était le cas pour la première version de Java, dans laquelle les variables finales devaient être initialisées lors de leur déclaration).

Utilisation de final *avec des objets*

L'utilisation de **final** n'est pas réservée aux primitives. Un handle d'objet peut parfaitement être déclaré **final**. Cependant, la contrainte ne s'applique alors qu'au handle, qui ne peut plus voir son affectation modifiée. L'objet, lui, reste modifiable.

Accessibilité

Les variables, comme les objets, les méthodes et les classes, ont également des accessibilités différentes. L'accessibilité définit dans quelles conditions on peut accéder à un élément ou, plus exactement, qui peut y accéder. Un objet peut être déclaré de façon à n'être accessible que depuis la classe qui le contient. On utilise pour cela le mot clé **private**. Les autres modes d'accessibilité faisant appel à des notions que nous n'avons pas encore étudiées, nous les laisserons de côté pour l'instant.

Retour sur les variables statiques

Nous avons défini au chapitre précédent ce qu'étaient les membres statiques en disant qu'ils appartenaient à une classe et non à une de ses instances.

Nous allons maintenant revenir sur cette notion à la lumière de ce que nous savons.

Lorsque, dans la définition d'une classe, une primitive est déclarée statique, il n'en existe qu'un seul exemplaire, quel que soit le nombre d'instances de cette classe créées (même si ce nombre est 0). Pour prendre une analogie, considérons la définition d'une classe comme le plan d'une voiture. Le nombre de sièges est une variable statique. En effet, il s'agit d'une caractéristique du modèle, et non de chaque exemplaire. Vous pouvez connaître la valeur de cette variable en regardant le plan. En revanche, la quantité d'essence restant dans le réservoir n'est pas une variable statique. Vous ne pouvez pas la mesurer sur le plan, bien que son existence ait un sens. En effet, vous pouvez savoir, en consultant le plan, si vous pourrez interroger chaque exemplaire pour connaître la valeur de cette variable. Vous le pourrez si le plan indique que ce modèle de voiture possède une jauge de carburant.

La distinction *variable de classe* / *variable d'instance* devrait maintenant être claire. Nous pouvons donc la compliquer un peu.

Supposons que vous vouliez connaître la quantité d'essence restant dans le réservoir d'une voiture. Inutile de regarder sur le plan. Cette valeur ne concerne que les exemplaires, ou *instances*.

En revanche, si vous voulez connaître le nombre de sièges, vous pouvez consulter le plan, mais vous pouvez également consulter un exemplaire quelconque.

Nous avons vu qu'en Java, pour accéder à un membre d'un objet, il faut indiquer le nom de l'objet suivi du nom du membre, en séparant les deux à l'aide d'un point. Considérons maintenant l'exemple suivant :

```
public class VariablesStatiques {
   public static void main(String[] argv) {
      Voiture maVoiture = new Voiture();
      Voiture taVoiture = new Voiture();
      System.out.println (maVoiture.puissanceMaxi);
      System.out.println (taVoiture.puissanceMaxi);
      System.out.println (Voiture.puissanceMaxi);
```

```
        Voiture.puissanceMaxi = 300;
        System.out.println (maVoiture.puissanceMaxi);
        System.out.println (taVoiture.puissanceMaxi);
        System.out.println (Voiture.puissanceMaxi);
        maVoiture.puissanceMaxi = 200;
        System.out.println (maVoiture.puissanceMaxi);
        System.out.println (taVoiture.puissanceMaxi);
        System.out.println (Voiture.puissanceMaxi);
    }
}
class Voiture {
        static int puissanceMaxi = 250;
}
```

Ce programme affiche le résultat suivant :

```
250
250
250
300
300
300
200
200
200
```

Nous voyons ainsi que :

- On peut accéder à un membre **static** par l'intermédiaire de la classe ou d'une instance. Il est néanmoins conseillé, pour optimiser la lisibilité des programmes, de s'en tenir à l'accès par la classe.

- Si on modifie la valeur d'une variable statique, elle est modifiée pour toutes les instances, même celles créées avant la modification. Cela est impliqué par le fait que la variable n'existe qu'à un seul exemplaire, partagé par la classe et toutes les instances.

- En modifiant une variable statique par l'intermédiaire d'une instance, (ce qui n'améliore pas la lisibilité du programme), on obtient le même résultat qu'en le faisant par l'intermédiaire de la classe.

Pour observer encore plus en détail ce principe, on peut modifier le programme précédent de la manière suivante :

```
public class VariablesStatiques2 {
   public static void main(String[] argv) {
      Voiture maVoiture = new Voiture();
      Voiture taVoiture = new Voiture();
      System.out.print (maVoiture.moteur + "   ");
      System.out.println (maVoiture.moteur.puissanceMaxi);
      System.out.print (taVoiture.moteur + "   ");
      System.out.println (taVoiture.moteur.puissanceMaxi);
      System.out.print (Voiture.moteur + "   ");
      System.out.println (Voiture.moteur.puissanceMaxi);
      Voiture.moteur.puissanceMaxi = 300;
      System.out.print (Voiture.moteur + "   ");
      System.out.println (Voiture.moteur.puissanceMaxi);
      System.out.print (taVoiture.moteur + "   ");
      System.out.println (taVoiture.moteur.puissanceMaxi);
      System.out.print (Voiture.moteur + "   ");
      System.out.println (Voiture.moteur.puissanceMaxi);
      maVoiture.moteur.puissanceMaxi = 200;
      System.out.print (maVoiture.moteur + "   ");
      System.out.println (maVoiture.moteur.puissanceMaxi);
      System.out.print (taVoiture.moteur + "   ");
      System.out.println (taVoiture.moteur.puissanceMaxi);
      System.out.print (Voiture.moteur + "   ");
      System.out.println (Voiture.moteur.puissanceMaxi);
      Moteur.puissanceMaxi = 150;
      System.out.print (maVoiture.moteur + "   ");
      System.out.println (maVoiture.moteur.puissanceMaxi);
      System.out.print (taVoiture.moteur + "   ");
      System.out.println (taVoiture.moteur.puissanceMaxi);
      System.out.print (Voiture.moteur + "   ");
```

```
      System.out.println (Voiture.moteur.puissanceMaxi);
      System.out.println (maVoiture);
      System.out.println (taVoiture);
   }
}
class Voiture {
   static Moteur moteur = new Moteur();
   }

class Moteur {
   static int puissanceMaxi = 250;
}
```

Ce programme ne présente d'autre intérêt que de nous permettre de vérifier ce que nous avons appris jusqu'ici. Nous avons maintenant deux classes en plus de notre programme principal : **Voiture** et **Moteur**.

Dans le programme principal, nous créons deux instances de **Voiture** : **maVoiture** et **taVoiture**. La classe **Voiture** possède cette fois un membre statique qui n'est plus une primitive, mais un objet. Cet objet possède lui-même un membre statique qui est une primitive de type **int**.

Note : Aucune de ces classes ne possède de constructeur explicite. (Rappelons qu'un constructeur est une méthode qui porte le même nom que la classe et qui est exécutée automatiquement lorsqu'une instance est créée.) Java utilise donc le constructeur par défaut pour créer les instances de **Voiture** et de **Moteur**.

Après avoir créé les deux instances de **Voiture**, nous affichons l'objet **maVoiture.moteur** :

```
System.out.print (maVoiture.moteur + "   ");
```

Comme nous l'avons dit précédemment, afficher un objet consiste simplement à exécuter sa méthode **toString()** et à afficher le résultat. En l'absence de méthode **toString()** dans la classe **Moteur**, Java affiche par défaut (par un mécanisme qui sera expliqué en détail au prochain chapitre) le nom de la

classe et l'adresse en mémoire de l'objet. (Nous utilisons ici la méthode **System.out.print()** qui n'ajoute pas de fin de ligne, et nous ajoutons une chaîne littérale composée de trois espaces.)

La ligne suivante affiche la valeur de la primitive **puissanceMaxi** du membre **moteur** de l'objet **maVoiture**.

```
System.out.println (maVoiture.moteur.puissanceMaxi);
```

Le programme fait ensuite la même chose pour l'objet **taVoiture** ainsi que pour la classe **Voiture**.

Les mêmes opérations sont ensuite répétées après avoir effectué les opérations suivantes :

```
Voiture.moteur.puissanceMaxi = 300;
```

puis :

```
maVoiture.moteur.puissanceMaxi = 200;
```

et enfin :

```
Moteur.puissanceMaxi = 150;
```

La première de ces opérations modifie l'instance **moteur** dans la classe **Voiture**. La deuxième modifie l'instance **moteur** dans l'instance **maVoiture**. Enfin, la troisième modifie directement la valeur de **puissanceMaxi** dans la classe **Moteur**.

Enfin, à la fin du programme, nous affichons les objets **maVoiture** et **taVoiture** :

```
        System.out.println (maVoiture);
        System.out.println (taVoiture);
```

Si nous exécutons ce programme, nous obtenons le résultat suivant :

```
Moteur@10f8011b   250
Moteur@10f8011b   250
Moteur@10f8011b   250
Moteur@10f8011b   300
Moteur@10f8011b   300
Moteur@10f8011b   300
Moteur@10f8011b   200
Moteur@10f8011b   200
Moteur@10f8011b   200
Moteur@10f8011b   150
Moteur@10f8011b   150
Moteur@10f8011b   150
Voiture@10f4011b
Voiture@10f0011b
```

Nous voyons immédiatement qu'il n'existe, à tous moments du déroulement du programme, qu'un seul objet instance de la classe **Moteur**, à l'adresse 10f8011b. En revanche, il existe bien deux instances différentes de la classe **Voiture**, l'une à l'adresse 10f4011b et l'autre à l'adresse 10f0011b.

Note : Toutes ces adresses changent à chaque exécution du programme et seront forcément différentes dans votre cas.

Attention : Nous venons de voir qu'il est possible d'utiliser des références différentes telles que :

```
maVoiture.moteur.puissanceMaxi
taVoiture.moteur.puissanceMaxi
Voiture.moteur.puissanceMaxi
```

En revanche, les références suivantes sont impossibles :

```
maVoiture.Moteur.puissanceMaxi
Voiture.Moteur.puissanceMaxi
```

En effet, **Moteur** fait référence à une classe. Or, la notation utilisant le point comme séparateur indique une hiérarchie d'inclusion. L'expression :

```
maVoiture.moteur.puissanceMaxi
```

signifie que l'objet **maVoiture** possède un membre appelé **moteur** qui possède un membre appelé **puissanceMaxi**. De la même façon, la référence :

```
maVoiture.Moteur.puissanceMaxi
```

signifierait que l'objet **maVoiture** possède un membre **Moteur** qui possède un membre **puissanceMaxi**, ce qui est manifestement faux, puisque **Moteur** est une classe et que cette classe n'est pas un membre de l'objet **maVoiture**.

De la même façon, la référence :

```
Voiture.Moteur.puissanceMaxi
```

signifie que la classe **Voiture** possède un membre **Moteur** (qui est lui-même une classe) qui possède un membre **puissanceMaxi**, ce qui ne correspond pas non plus à la hiérarchie créée par notre programme.

Note : Il faut distinguer les différents types de hiérarchies. Ici, il s'agit d'une hiérarchie basée sur la possession d'un membre, ou *composition*. Cela n'a rien à voir avec la hiérarchie liée à la généalogie des classes, que nous étudierons au chapitre suivant, pas plus qu'avec celle de la propagation des événements, qui fera l'objet d'une étude ultérieure. Le type de relation impliquée par la hiérarchie décrite ici est parfois appelé "*a un*" (en anglais, "*has a*") par opposition à la relation mise en œuvre dans la hiérarchie généalogique ou *héritage*, appelée "*est un*" (en anglais "*is a*").

Ainsi, on peut dire que les objets de la classe **Voiture** ont un membre de la classe **Moteur**, ce qui permet d'écrire **Voiture.moteur** ou **maVoiture.moteur** mais ni les objets instances de la classe **Voiture** ni la classe **Voiture** elle-même ne contiennent la classe **Moteur**, ce qui interdit d'écrire **Voiture.Moteur** ou **maVoiture.Moteur**. (Nous verrons au chapitre suivant qu'une classe peut, d'une certaine façon, contenir une autre classe, mais ce n'est pas le cas ici.)

En revanche, l'écriture suivante est possible :

```
Voiture.(Moteur.puissanceMaxi)
```

car (**Moteur.puissanceMaxi**) est une référence à une primitive statique, alors que :

```
Voiture.Moteur.puissanceMaxi
```

est l'équivalent de :

```
(Voiture.Moteur).puissanceMaxi
```

Toutefois, il n'est pas possible d'écrire :

```
maVoiture.(Moteur.puissanceMaxi);
```

ce qui produit une erreur de compilation.

Masquage des identificateurs de type static

Nous avons vu précédemment que des identificateurs pouvaient être masqués. Les identificateurs de type **static** peuvent être masqués comme les autres. Examinez le programme suivant :

```
public class VariablesStatiques4 {
   public static void main(String[] argv) {
```

```
      Voiture maVoiture = new Voiture(350);
      System.out.println (maVoiture.puissance);
   }
}
class Voiture {
   static int puissance = 300;

   Voiture (int puissance) {
      System.out.println (puissance);
      System.out.println (Voiture.puissance);
   }
}
```

Si vous exécutez ce programme, vous obtiendrez le résultat suivant :

```
350
300
300
```

Ici, la classe **Voiture** est dotée d'un constructeur. La première ligne de ce constructeur affiche la valeur du paramètre **puissance**, passé lors de la création de l'instance. Ce paramètre ayant masqué le membre statique qui porte le même nom, comme on peut le voir sur la première ligne de l'affichage. En revanche, le membre statique est toujours accessible en y faisant référence à l'aide du nom de la classe, comme le montre la ligne suivante de l'affichage. Dès que l'on est hors de la portée du paramètre qui masquait le membre statique, celui-ci redevient disponible normalement comme le montre la troisième ligne de l'affichage, provoquée par la ligne :

```
System.out.println (maVoiture.puissance);
```

de la méthode **main()**.

Vous vous demandez peut-être ce que donnerait cette ligne si elle était exécutée à l'intérieur du constructeur, comme dans l'exemple ci-dessous :

```
public class VariablesStatiques4 {
   public static void main(String[] argv) {
      Voiture maVoiture = new Voiture(350);
      System.out.println (maVoiture.puissance);
   }
}

class Voiture {
   static int puissance = 300;
   Voiture (int puissance) {
      System.out.println (puissance);
      System.out.println (Voiture.puissance);
      // La ligne suivante produit une erreur :
      System.out.println (maVoiture.puissance);
   }
}
```

Ce programme ne peut pas être compilé. En effet, il produit une erreur car la référence à l'instance **maVoiture** est impossible puisque la construction de cet objet n'est pas terminée. Pourtant, l'objet en question est parfaitement accessible. Seule sa référence à l'aide du handle **maVoiture** ne peut être résolue. En revanche, vous pouvez y faire référence à l'aide du mot clé **this**, qui sert à désigner l'objet dans lequel on se trouve. Ainsi modifié, le programme devient :

```
public class VariablesStatiques4 {
   public static void main(String[] argv) {
      Voiture maVoiture = new Voiture(350);
      System.out.println (maVoiture.puissance);
   }
}

class Voiture {
   static int puissance = 300;
   Voiture (int puissance) {
      System.out.println (puissance);
      System.out.println (Voiture.puissance);
```

```
        System.out.println (this.puissance);
    }
}
```

et peut être compilé sans erreur. Son exécution produit le résultat suivant :

```
350
300
300
300
```

ce qui confirme le masquage du membre statique.

Note : Vous ne devez pas croire tout ce que l'on vous dit sur parole. Il vaut toujours mieux vérifier par vous-même. Pouvez-vous imaginer un moyen de vérifier que **this** fait bien référence à l'objet **maVoiture** ? (Réponse à la fin de ce chapitre.)

Java et les pointeurs

Un des sujets de controverse à propos de Java est de savoir si ce langage possède des pointeurs. Certains auteurs ont écrit que non, puisque le langage ne permet pas de manipuler le type *pointeur*, comme d'autres langages.

Un pointeur est tout simplement une variable qui, au lieu de contenir une valeur, contient l'adresse en mémoire de cette valeur. Avant les langages orientés objet, les pointeurs étaient le seul outil qui permettait de créer efficacement des structures de données composites, comportant par exemple des chaînes de caractères, des valeurs numériques, des valeurs booléennes, etc. Il suffisait au programmeur de réserver une partie de la mémoire pour y stocker ce qu'il voulait, et d'utiliser une variable pour pointer vers l'adresse de cette zone de mémoire. Un certain nombre d'opérateurs permettaient d'effectuer des opérations sur les pointeurs pour optimiser l'accès aux données. Les pointeurs sont considérés, à juste titre, comme l'élément le plus

dangereux de ce type de langage. Une simple erreur sur l'utilisation d'un pointeur permet au programme d'écrire dans n'importe quelle partie de la mémoire et, dans certains environnements peu sûrs comme Windows, de planter tout le système.

Un langage tel que Java rend les pointeurs inutiles puisque l'on peut créer des objets de toutes natures pour représenter les données. Cependant, certains prétendent que Java possède des pointeurs (les handles) mais que le programmeur ne dispose pas des opérateurs pour manipuler ces pointeurs. Cet avis demande à être nuancé. En effet, qu'est-ce qu'un pointeur ? Si on appelle pointeur tout ce qui pointe vers quelque chose, alors l'identificateur d'une primitive (son nom) est aussi un pointeur, puisque ce nom pointe vers une zone de mémoire. Mais ici, le pointage est tout à fait virtuel. L'identificateur n'est qu'un mot dans le programme. Il n'a pas d'existence propre, indépendante de la zone de mémoire correspondante. En revanche, un handle pointe également vers une zone de mémoire, mais il a lui-même une existence propre en mémoire. On pourrait donc dire qu'il s'agit d'un pointeur.

Pas tout à fait. En effet, nul ne peut savoir vers quoi un handle pointe exactement. Virtuellement, pour le programmeur, il pointe vers l'objet auquel il a été affecté. Mais réellement, lors de l'exécution du programme par la JVM (Machine Virtuelle Java), bien malin qui peut dire vers quoi pointe le handle. Cela dépend de la façon dont la JVM est conçue. Si l'on veut optimiser la JVM pour l'accès aux objets, il est préférable que les handles pointent directement vers les objets. En revanche, si l'on veut optimiser la manipulation des objets, c'est une autre histoire. En effet, il est parfois nécessaire de déplacer les objets en mémoire. Si plusieurs handles pointent vers le même objet, il faudra alors mettre à jour tous les handles, ce qui peut être long. Pour palier ce problème, certaines JVM (celle de Sun, en particulier) font pointer les handles vers un pointeur qui pointe lui-même vers l'objet. Ainsi, si cinq handles pointent vers le même objet et si l'objet est déplacé, il suffit de modifier le pointeur intermédiaire, ce qui est plus rapide. Reste à savoir si l'on peut encore dans ce cas considérer que les handles sont des pointeurs, mais cela reste un débat purement terminologique.

Exercices

Pas d'exercice, ici, mais la réponse aux deux questions posées dans ce chapitre.

Première question (page 104)

La première question était : "Pourquoi la ligne suivante produit-elle une erreur de compilation" :

```
long i = 21474836409;
```

alors que la valeur littérale **21474836409** est tout à fait représentable par un **long**. Pour comprendre ce qui se passe ici, il faut se souvenir qu'en Java, les valeurs littérales entières sont, par défaut, de type **int**. Ainsi, pour compiler la ligne ci-dessus, Java tente d'abord d'affecter la valeur littérale à un **int** avant d'effectuer un casting vers un **long**. C'est l'affectation de la valeur à un **int** qui produit l'erreur de compilation, car la valeur littérale est trop grande pour un **int**. Il aurait fallu écrire :

```
long i = 21474836409L;
```

Deuxième question (page 129)

Pour vérifier que le mot clé **this** fait bien référence à l'instance de **Voiture** en cours de création, il faut, tout d'abord, considérer que, dans ce programme, il ne peut exister qu'un seul objet de ce type, puisque nous n'en créons qu'un seul. Il suffit donc de montrer que **this** fait référence à un objet instance de la classe **Voiture**. Si c'est le cas, c'est forcément le nôtre. Cela peut être mis en évidence très simplement en appelant la méthode **System.out.println()** avec **this** pour paramètre :

```
public class VariablesStatiques4 {
   public static void main(String[] argv) {
```

```
      Voiture maVoiture = new Voiture(350);
      System.out.println (maVoiture.puissance);
   }
}
class Voiture {
   static int puissance = 300;
   Voiture (int puissance) {
      System.out.println (puissance);
      System.out.println (Voiture.puissance);
      System.out.println (this.puissance);
      System.out.println (this);
   }
}
```

Le résultat obtenu est le suivant :

```
350
300
300
Voiture@10f723ec
300
```

ce qui nous prouve que **this** fait référence à un objet instance de la classe **Voiture** se trouvant à l'adresse mémoire 10f723ec. (Cette adresse sera différente dans votre cas.)

Résumé

Dans ce chapitre, nous avons étudié deux éléments fondamentaux du langage Java : les handles et les primitives. La discussion présentée ici peut parfois sembler couper les cheveux en quatre. Il est cependant nécessaire de maîtriser parfaitement ces notions avant de passer à la suite. Dans le chapitre suivant, nous verrons comment il est possible de créer de nouvelles classes d'objets, ce qui constitue l'activité fondamentale d'un programmeur en Java.

Chapitre 5

Créez vos propres classes

Dans les exemples que nous avons utilisés dans les chapitres précédents, nous avons été amenés à créer des classes. Nous allons maintenant étudier cet aspect de la programmation en Java plus en détail. Programmer en Java consiste uniquement à créer des classes d'objets adaptées aux problèmes à résoudre. Lorsque les classes nécessaires au traitement d'un problème ont été déterminées et définies, le développement est terminé.

Le développement d'une application Java peut se décomposer en deux phases. La première, que nous appellerons *conception*, consiste à représenter l'univers du problème à l'aide d'un ensemble de classes. La seconde, que nous appellerons *implémentation*, consiste à construire ces classes. La phase

de conception est la plus importante car elle conditionne le bon déroulement de la phase d'implémentation. Si la conception est mauvaise, aussi bonne que soit la qualité de l'implémentation, l'application sera peu efficace, voire ne fonctionnera pas. De plus, il faudra reprendre le problème à la base pour améliorer les choses. Il est donc important de consacrer les ressources suffisantes à cette phase.

La première chose à faire pour développer une application consiste donc à faire la liste des classes qui seront nécessaires, ce qui consiste à découper l'univers du problème en catégories d'objets (les classes), en déterminant leurs fonctionnalités, c'est-à-dire :

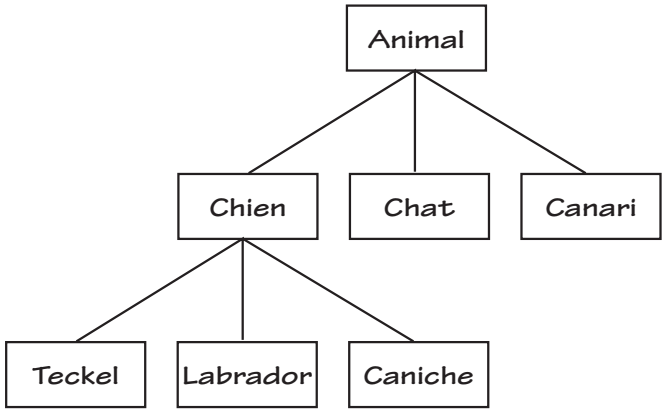
- les états qu'ils pourront prendre ;
- les messages qu'ils pourront recevoir et la façon dont ils y réagiront.

Cependant, avant de pouvoir concevoir un modèle efficace, il est nécessaire de comprendre le système hiérarchique qui régit les relations entre les classes Java.

Tout est objet (bis)

Comme nous l'avons déjà dit, tous les éléments que manipule Java sont des objets, à l'exception des primitives. Cependant, les primitives peuvent être traitées comme des objets grâce aux *enveloppeurs*.

Tout objet Java est une *instance* d'une *classe*. De plus, chaque classe dérive d'une classe de niveau supérieur, parfois appelée *classe parente*. Cela est vrai pour toutes les classes, sauf une. Il s'agit de la classe **Object**, qui est l'ancêtre de toutes les classes. C'est là le premier point fondamental que vous ne devez jamais oublier. Le deuxième point tout aussi fondamental est que toute instance d'une classe est un objet du type correspondant, mais aussi du type de toutes ses classes ancêtres. C'est en fait ce que nous exprimons en disant que tout est objet. Si l'on reprend la hiérarchie suivante :



il apparaît immédiatement qu'elle est incomplète. En effet, la classe **Animal** est elle-même dérivée de la classe **Object**, ou d'une autre classe descendant de la classe **Object**.

La hiérarchie décrite ici est de type "*est un*", contrairement à la hiérarchie "*a un*" étudiée au chapitre précédent. Un **Teckel** *est un* **Chien**, un **Chien** *est un* **Animal**, un **Animal** *est un* **Object**. On peut en déduire qu'un **Teckel** est un **Animal**, et qu'un **Teckel** est aussi un **Object**.

L'héritage

Lorsque nous disons qu'un **Chien** est un **Animal** (en insistant sur le fait qu'il s'agit de classes Java, et non d'objets réels) cela signifie entre autres qu'un **Chien** *hérite* de toutes les caractéristiques d'un animal. De la même façon, un **Teckel** hérite de toutes les caractéristiques d'un **Chien**, et donc, par transitivité, d'un **Animal**. Définissons la classe **Animal** de la façon suivante :

Animal :
    vivant
    âge
    crie()
    vieillit()
    meurt()

ce qui signifie qu'un animal possède un état (**vivant**, qui peut être vrai ou faux et **âge**, qui est une valeur numérique) et sait faire plusieurs choses (**crie()**, **vieillit()** et **meurt()**, qui sont des *méthodes*).

Nous pouvons en déduire immédiatement qu'un **Caniche** ou un **Canari** possèdent les mêmes caractéristiques. Nous pouvons parfaitement imaginer que, pour un **Animal**, la caractéristique *vivant* soit représentée par une primitive de type **boolean**. Nous pouvons donc commencer à écrire notre classe de la façon suivante :

```
class Animal {
   boolean vivant;
   int âge;
}
```

Les méthodes **vieillit()** et **meurt()** peuvent être écrites très simplement de la façon suivante :

```
class Animal {
   boolean vivant;
   int âge;

   void vieillit() {
      ++âge;
   }

   void meurt() {
      vivant = false;
   }
}
```

(La syntaxe **++âge** n'a pas encore été étudiée. Sachez que cela signifie simplement *augmenter de 1 la valeur de* **âge**.)

En revanche, il est plus difficile de représenter la méthode **crie()**. En effet, nous savons que tout animal crie (dans l'univers de notre problème, pas

dans la réalité !), mais nous ne pouvons pas déterminer ce qu'est le cri d'un animal. Aussi, nous allons définir une méthode générique de la façon suivante :

```
class Animal {
   boolean vivant;
   int âge;

   void vieillit() {
      ++âge;
   }

   void meurt() {
      vivant = false;
   }

   void crie() {
   }
}
```

Cette méthode ne fait rien, mais sa présence indique qu'un **Animal** est capable de réagir au message **crie()**. En l'absence de cette méthode, ce message produirait une erreur.

Notre méthode n'est pas complète. En effet, si nous voulons créer une instance d'**Animal**, il faut que cette instance soit *initialisée*. Cette initialisation peut comporter diverses tâches, par exemple l'affichage d'un message indiquant la création d'une nouvelle instance. Cela est laissé à l'appréciation du programmeur, mais un certain nombre de tâches doivent obligatoirement être effectuées. Il s'agit, par exemple, de l'initialisation des variables. Si une instance d'**Animal** est créée, il faut lui donner un âge et indiquer s'il est vivant ou mort. (Cela peut paraître bizarre mais, encore une fois, il s'agit de l'univers du problème et non de l'univers réel. Si nous écrivons un programme de gestion de stock pour un revendeur d'animaux domestiques, il faut pouvoir faire entrer dans le stock un animal de n'importe quel âge.)

Il serait possible d'initialiser la variable **vivant** de la façon suivante :

```
boolean vivant = true;
```

De cette façon, un animal créé est toujours vivant. Cependant, cette technique ne peut pas être utilisée pour l'âge, ni pour afficher un message.

En règle générale, et hormis le cas des primitives initialisées lors de leur déclaration comme dans l'exemple ci-dessus, on utilise en Java deux techniques différentes pour initialiser les objets : les *constructeurs* et les *initialiseurs*.

Les constructeurs

Comme nous l'avons déjà évoqué, les constructeurs sont des méthodes particulières en ce qu'elles portent le même nom que la classe à laquelle elles appartiennent. Elles sont automatiquement exécutées lors de la création d'un objet. Nous pourrions écrire pour notre classe **Animal** le constructeur suivant :

```
class Animal {
  boolean vivant;
  int âge;

  Animal (int a) {
     âge = a;
     vivant = true;
  }

  void vieillit() {
     ++âge;
  }

  void meurt() {
     vivant = false;
  }
```

```
    void crie() {
    }
}
```

Dans ce cas, la création d'un **Animal** se ferait à l'aide de l'instruction suivante :

```
Animal nouvelAnimal = new Animal(3);
```

(Notez que, dans cet exemple, un animal créé est toujours vivant.)

Le constructeur initialise l'objet au moyen du paramètre qui lui est passé. Ce paramètre est utilisé pour initialiser la variable d'instance **âge**. Cela est possible parce que cette variable a une portée qui s'étend de sa déclaration jusqu'à la fin du bloc, c'est-à-dire, dans ce cas, la fin de la classe. Par ailleurs, la visibilité de cette variable dans le constructeur est assurée par le fait que nous avons appelé le paramètre **a** et non **âge**. Cela nous permet d'éviter le masquage de la variable d'instance **âge**. Si nous avions donné au paramètre le même nom que celui de la variable d'instance, il aurait fallu accéder à celle-ci de la façon suivante :

```
Animal (int âge) {
   this.âge = âge;
   vivant = true;
}
```

Comme il ne saute pas aux yeux qu'**âge** (le paramètre) et **âge** (la variable d'instance) sont deux primitives différentes, il vaut mieux leur donner des noms différents. Dans le cas de l'initialisation d'une variable d'instance à l'aide d'un paramètre, on utilise souvent pour le nom du paramètre la première (ou les premières) lettre(s) du nom de la variable d'instance.

Si nous voulons qu'un message soit affiché lors de la création d'un **Animal**, nous pouvons ajouter au constructeur le code suivant :

```
Animal (int âge) {
   this.âge = âge;
   vivant = true;
   System.out.println("Un animal de " + a
               + " an(s) vient d'être créé");
}
```

ou encore :

```
Animal (int âge) {
   this.âge = âge;
   vivant = true;
   System.out.println("Un animal de " + âge
               + " an(s) vient d'être créé");
}
```

Note : De ces deux solutions, laquelle faut-il préférer ? L'accès à un paramètre à l'intérieur d'une méthode est toujours plus rapide que l'accès à une variable d'instance. Le gain de performance est variable selon la JVM utilisée. Avec la version 1.2 bêta 3 de Sun, le gain est de 20 %. (Avec la version 1.1.5, il n'est que de 10 %, ce qui est plus proche de ce que vous pouvez espérer en général.) La réponse est donc : "Utilisez les paramètres chaque fois que c'est possible." En particulier, si des calculs doivent être effectués, il faut les effectuer sur les paramètres et affecter le résultat aux variables d'instances, plutôt que d'initialiser les variables d'instances puis effectuer les calculs sur elles.

Nous pouvons maintenant créer les classes dérivées de la classe **Animal**. Nous commencerons par la classe **Canari**.

Tout d'abord, nous indiquerons que cette classe dérive de la classe **Animal** de la façon suivante :

```
class Canari extends Animal {
}
```

Référence à la classe parente

Notre nouvelle classe a besoin d'un constructeur. Ce constructeur prendra pour paramètre un entier indiquant l'âge du canari. Ce paramètre sera tout simplement passé au constructeur de la classe parente au moyen de l'identificateur **super**.

La seule chose, dans notre modèle, qui distingue un **Canari** d'un autre **Animal** est son cri. Pour mettre en œuvre cette différence, nous devons utiliser une technique appelée *method overriding*, en anglais, que l'on peut traduire par *redéfinition de méthode*. Dans les livres publiés en français, on trouve souvent le terme *outrepasser* une méthode. L'inconvénient de ce verbe est qu'il est difficile d'en dériver un substantif (*outrepassement ?*). Nous préférerons donc *redéfinition*.

La redéfinition des méthodes

Au titre de l'*héritage*, la classe **Canari** dispose de tous les membres de la classe **Animal**, et donc, en particulier, de ses variables d'instances et de ses méthodes. Toutefois, si une variable d'instance est initialisée dans la classe parente, nous pouvons non seulement l'utiliser en consultant sa valeur, mais également lui affecter une nouvelle valeur. De la même façon, nous pouvons utiliser les méthodes de la classe parente, mais nous pouvons également les redéfinir. C'est ce que nous ferons pour la méthode **crie()** :

```
class Canari extends Animal {
   void crie() {
      System.out.println("Cui-cui !");
   }
}
```

Pour utiliser ces classes, nous les placerons, pour l'instant, dans le même fichier que le programme principal, que nous appellerons **Animaux**. En voici le listing complet :

```
class Animaux {
   public static void main(String[] args) {
      Canari titi = new Canari(3);
      titi.vieillit();
      titi.crie();
   }
}
class Animal {
   boolean vivant;
   int âge;

   Animal (int a) {
      âge = a;
      vivant = true;
      System.out.println("Un animal de " + a
              + " an(s) vient d'être créé");
   }
   void vieillit() {
      ++âge;
      System.out.println("C'est l'anniversaire de cet animal.");
      System.out.println("Il a maintenant " + âge + " an(s).");
   }

   void meurt() {
      vivant = false;
      System.out.println("Cet animal est mort");
   }
   void crie() {
   }
}
class Canari extends Animal {
   Canari(int a) {
      super(a);
   }
   void crie() {
      System.out.println("Cui-cui !");
   }
}
```

Nous avons apporté quelques modifications aux méthodes afin de les rendre plus attractives. Si vous compilez et exécutez ce programme, vous obtenez le résultat suivant :

```
Un animal de 3 an(s) vient d'être créé.
C'est l'anniversaire de cet animal.
Il a maintenant 4 an(s).
Cui-cui !
```

La classe **Animaux** possède uniquement une méthode main() qui crée un nouveau handle pour un objet de type **Canari** et lui affecte un nouvel objet de cette classe créé à l'aide de l'opérateur **new**. Le paramètre 3 (de type **int**, comme nous l'avons vu au chapitre précédent) est passé au constructeur de la classe **Canari**.

Le constructeur de la classe **Canari** appelle simplement le constructeur de la classe parente (**super()**) en lui transmettant le paramètre reçu. C'est là un point particulièrement remarquable. En effet, c'est le constructeur de la classe **Animal** qui est exécuté, mais il construit bien un objet de la classe **Canari**. Les variables d'instances **âge** et **vivant** sont initialisées, et le message correspondant est affiché.

La procédure **main()** appelle ensuite les méthodes **vieillit()** et **crie()**. La méthode **vieillit()** n'est pas redéfinie dans la classe **Canari**. Elle y est cependant disponible en vertu de l'héritage. L'**âge** de **titi** est donc augmenté d'une unité, puis deux messages sont affichés.

La méthode **crie()**, en revanche, est redéfinie dans la classe **Canari**. C'est donc cette version qui est exécutée et affiche le cri du canari.

Surcharger les méthodes

Une des modifications que nous pourrions apporter à notre programme concerne la possibilité de ne souhaiter l'anniversaire des animaux qu'une fois de temps en temps, mais pas tous les ans. (Ne me demandez pas pourquoi !

Disons que c'est une exigence du client pour lequel nous devons développer ce programme.)

Nous pourrions modifier la méthode **vieillit()** afin qu'elle prenne un paramètre indiquant le nombre d'années à ajouter à l'âge. Cependant, nous voulons conserver la possibilité d'utiliser la méthode sans paramètre si le vieillissement est de 1. Pour cela, il nous suffit de créer dans la classe **Animal** une deuxième version de la méthode avec le même nom :

```
class Animal {
   boolean vivant;
   int âge;

   Animal (int a) {
      âge = a;
      vivant = true;
      System.out.println("Un animal de " + a
              + " an(s) vient d'être créé");
   }
   void vieillit() {
      ++âge;
      System.out.println("C'est l'anniversaire de cet animal.");
      System.out.println("Il a maintenant " + âge + " an(s).");
   }
   void vieillit(int a) {
      âge += a;
      System.out.println("C'est l'anniversaire de cet animal.");
      System.out.println("Il a maintenant " + âge + " an(s).");
   }
   void meurt() {
      vivant = false;
      System.out.println("Cet animal est mort");
   }
   void crie() {
   }
}
```

(admettez, pour l'instant, que la syntaxe **âge += a** signifie *ajouter la valeur de* **a** *à* **âge**.)

Nous avons utilisé ici une des fonctionnalités très importantes du langage Java, qui consiste à *surcharger* les méthodes. Surcharger une méthode consiste simplement à la définir plusieurs fois avec le même nom. Comment fait l'interpréteur pour savoir quelle version il doit exécuter ? Il utilise tout simplement pour cela la *signature* de la méthode.

Signature d'une méthode

La signature d'une méthode désigne la liste de ses paramètres avec leur type. Ici, les deux méthodes ont des signatures différentes. L'une prend un paramètre de type **int**, l'autre ne prend aucun paramètre. Si la méthode **viellit()** est appelée avec un paramètre de type **int** ou sans paramètre, le compilateur sait quelle version il doit choisir. En revanche, un appel avec un paramètre d'un autre type, ou plusieurs paramètres, produit une erreur.

Note : Pour identifier les signatures des méthodes, Java utilise également l'ordre des paramètres. Ainsi, les deux méthodes suivantes :

```
méthode(int a, long b)
méthode(long b, int a)
```

ont des signatures différentes. En revanche, le nom des paramètres ne distingue pas les signatures. Une même classe ne pourra donc contenir les deux méthodes suivantes :

```
méthode(int a, long b)
méthode(int c, long d)
```

ce qui produirait une erreur de compilation.

Notre programme complet modifié est donc maintenant le suivant :

```
class Animaux {
   public static void main(String[] args) {
      Canari titi = new Canari(3);
      titi.vieillit();
      titi.vieillit(2);
      titi.crie();
   }
}
class Animal {
   boolean vivant;
   int âge;

   Animal (int a) {
      âge = a;
      vivant = true;
      System.out.println("Un animal de " + a
              + " an(s) vient d'être créé");
   }

   void vieillit() {
      ++âge;
      System.out.println("C'est l'anniversaire de cet animal.");
      System.out.println("Il a maintenant " + âge + " an(s)");
   }

   void vieillit(int a) {
      âge += a;
      System.out.println("C'est l'anniversaire de cet animal.");
      System.out.println("Il a maintenant " + âge + " an(s)");
   }

   void meurt() {
      vivant = false;
      System.out.println("Cet animal est mort");
   }
   void crie() {
   }
}
```

```
class Canari extends Animal {
   Canari(int a) {
      super(a);
   }
   void crie() {
      System.out.println("Cui-cui !");
   }
}
```

Si nous exécutons ce programme, nous obtenons le résultat suivant :

```
Un animal de 3 an(s) vient d'être créé.
C'est l'anniversaire de cet animal.
Il a maintenant 4 an(s).
C'est l'anniversaire de cet animal.
Il a maintenant 6 an(s).
Cui-cui !
```

Nous voyons que Java a bien exécuté chaque fois la méthode correspondant à la signature utilisée.

Optimisation

Il apparaît immédiatement que notre programme n'est pas du tout optimisé du point de vue de la maintenance. Si nous devons un jour modifier les messages affichés, il nous faudra effectuer deux fois la modification. En effet, chaque version de la méthode **vieillit()** comporte deux lignes identiques, susceptibles d'être modifiées.

Il y a plusieurs façons de résoudre le problème. L'une pourrait être de créer une méthode séparée pour l'affichage de l'âge. Elle conviendra si cette opération est réellement distincte de l'augmentation de l'âge. Cela pourrait en effet être le cas pour l'affichage. La classe **Animal** pourrait alors être réécrite de la façon suivante :

```
class Animal {
   boolean vivant;
   int âge;

   Animal (int a) {
      âge = a;
      vivant = true;
      System.out.println("Un animal de " + a
              + " an(s) vient d'être créé");
   }

   void vieillit() {
      ++âge;
      afficheAge();
   }

   void vieillit(int a) {
      âge += a;
      afficheAge();
   }

   void afficheAge() {
      System.out.println("C'est l'anniversaire de cet animal.");
      System.out.println("Il a maintenant " + âge + " an(s)");
   }

   void meurt() {
      vivant = false;
      System.out.println("Cet animal est mort");
   }

   void crie() {
   }
}
```

En revanche, imaginons que nous voulions vérifier si l'âge ne dépasse pas une valeur maximale, ce qui pourrait être réalisé de la façon suivante :

```
class Animal {
   boolean vivant;
   int âge;
   int âgeMaxi = 10;

   Animal (int a) {
      âge = a;
      vivant = true;
      System.out.println("Un animal de " + a
              + " an(s) vient d'être créé");
   }
   void vieillit() {
      ++âge;
      afficheAge();
      if (âge > âgeMaxi) {
         meurt();
      }
   }
   void vieillit(int a) {
      âge += a;
      afficheAge();
      if (âge > âgeMaxi) {
         meurt();
      }
   }
   void afficheAge() {
      System.out.println("C'est l'anniversaire de cet animal.");
      System.out.println("Il a maintenant " + âge + " an(s)");
   }
   void meurt() {
      vivant = false;
      System.out.println("Cet animal est mort");
   }
   void crie() {
   }
}
```

Nous retrouvons cette fois encore des lignes de code dupliquées. (Admettez, pour l'instant, que ces lignes signifient *si la valeur de* **âge** *est supérieure à la valeur de* **âgeMaxi**, *exécuter la méthode* **meurt()**.)

Nous n'avons aucune raison d'ajouter ici une nouvelle méthode pour tester l'âge, car celui-ci n'est testé que lorsque l'âge est modifié. Nous allons donc utiliser une autre approche :

```
class Animal {
   boolean vivant;
   int âge;
   int âgeMaxi = 10;

   Animal (int a) {
      âge = a;
      vivant = true;
      System.out.println("Un animal de " + a
              + " an(s) vient d'être créé");
   }
   void vieillit() {
      vieillit(1);
   }
   void vieillit(int a) {
      âge += a;
      if (âge > âgeMaxi) {
         meurt();
      System.out.println("C'est l'anniversaire de cet animal.");
      System.out.println("Il a maintenant " + âge + " an(s)");
      }
   }
   void meurt() {
      vivant = false;
      System.out.println("Cet animal est mort");
   }
   void crie() {
   }
}
```

Cette fois, il n'y a pas de lignes dupliquées. L'intégralité du traitement est effectuée en un seul endroit. La méthode qui gère le cas particulier appelle simplement la méthode générale en lui fournissant la valeur particulière du paramètre. Cette approche doit être employée chaque fois que possible pour améliorer la lisibilité du code. (Ici, nous avons réintégré l'affichage dans la méthode **vieillit()**. Bien sûr, l'opportunité de cette modification dépend de ce que vous voulez obtenir. Nous aurions tout aussi bien pu conserver la méthode **afficheAge()**.)

Surcharger les constructeurs

Les constructeurs peuvent également être surchargés. Notre classe **Animal** possède un constructeur qui prend pour paramètre une valeur de type **int**. Cependant, si la plupart des instances sont créées avec la même valeur initiale, nous pouvons souhaiter utiliser un constructeur sans paramètre.

Supposons, par exemple, que la plupart des instances soient créées avec 1 pour valeur initiale de **âge**. Nous pouvons alors réécrire la classe **Animal** de la façon suivante :

```
class Animal {
   boolean vivant;
   int âge;
   int âgeMaxi = 10;
   Animal() {
      âge = 1;
      vivant = true;
      System.out.println
              ("Un animal de 1 an(s) vient d'être créé");
   }
   Animal(int a) {
      âge = a;
      vivant = true;
      System.out.println("Un animal de " + a
              + " an(s) vient d'être créé");
   }
```

```
   void vieillit() {
      vieillit(1);
   }

   void vieillit(int a) {
      âge += a;
      if (âge > âgeMaxi) {
         meurt();
      System.out.println("C'est l'anniversaire de cet animal.");
      System.out.println("Il a maintenant " + âge + " an(s)");
      }
   }

   void meurt() {
      vivant = false;
      System.out.println("Cet animal est mort");
   }

   void crie() {
   }
}
```

Ici, les deux constructeurs ont des signatures différentes. Le constructeur sans paramètre traite les cas où l'âge vaut 1 à la création de l'instance. Une nouvelle instance peut donc être créée sans indiquer l'âge, de la façon suivante :

```
Animal milou = new Animal();
```

Cependant, nous avons le même problème d'optimisation de la lisibilité du code que dans le cas des méthodes. Bien sûr, la solution est maintenant évidente. Il suffit de modifier le constructeur traitant la valeur particulière (1) pour lui faire appeler le constructeur traitant le cas général. Dans le cas des méthodes, il suffisait d'appeler la méthode par son nom, avec le ou les paramètres correspondant à sa signature. Dans le cas d'un constructeur, l'équivalent pourrait être :

```
Animal() {
   Animal(1);
}
```

Ce type de référence n'est pas possible. Pour obtenir le résultat souhaité, vous devez utiliser le mot clé **this**, de la façon suivante :

```
Animal() {
   this(1);
}
```

La raison pour laquelle vous ne pouvez pas utiliser la première syntaxe n'est pas évidente à priori. Pouvez-vous essayer de l'imaginer ?

En fait, la réponse tient au fait que les constructeurs ne sont pas des méthodes. Par conséquent, une invocation de type :

```
Animal();
```

appelle une méthode et non un constructeur. Or, notre programme ne contient pas de méthode de ce nom. Cependant, il est tout à fait possible d'en créer une. Une classe peut parfaitement (et c'est bien dommage !) posséder une méthode portant le même nom que son constructeur. Bien entendu, il est impératif d'éviter cela, principalement en raison des risques de confusion entre la méthode et le constructeur. Par ailleurs, il est préférable de respecter l'usage consistant à faire commencer le nom des méthodes par une minuscule. Les constructeurs, ayant le même nom que leur classe, commenceront par une majuscule.

De plus, un autre usage consiste à utiliser des noms (du langage naturel) pour les classes, et donc les constructeurs, et des verbes pour les méthodes. En effet, les classes correspondent à des objets et les méthodes à des actions. La signification d'une méthode nommée **vieillit()** est assez évidente. En revanche, que fait une méthode nommée **Animal()** ?

Les constructeurs par défaut

Notre programme ainsi modifié ne fonctionne plus car nous avons introduit dans la classe **Canari** un bug que le compilateur n'est pas capable de détecter. Examinez le listing complet et essayez de le trouver :

```
class Animaux {
   public static void main(String[] args) {
      Canari titi = new Canari(3);
      titi.vieillit();
      titi.vieillit(2);
      titi.crie();
   }
}
class Animal {
   boolean vivant;
   int âge;
   int âgeMaxi = 10;

   Animal() {
      this(1);
   }
   Animal (int a) {
      âge = a;
      vivant = true;
      System.out.println("Un animal de " + a
              + " an(s) vient d'être créé");
   }

   void vieillit() {
      vieillit(1);
   }

   void vieillit(int a) {
      âge += a;
      afficheAge();
```

```
      if (âge > âgeMaxi) {
         meurt();
      }
   }

   void afficheAge() {
      System.out.println("C'est l'anniversaire de cet animal.");
      System.out.println("Il a maintenant " + âge + " an(s)");
   }

   void meurt() {
      vivant = false;
      System.out.println("Cet animal est mort");
   }

   void crie() {
   }
}

class Canari extends Animal {
   Canari(int a) {
      super(a);
   }
   void crie() {
      System.out.println("Cui-cui !");
   }
}
```

Vous avez trouvé ? Si vous avez essayé de compiler le programme et de l'exécuter, vous avez pu constater que tout fonctionne correctement. Cependant, modifiez la troisième ligne de la façon suivante :

```
Canari titi = new Canari();
```

La compilation n'est plus possible. Cette situation est délicate et potentiellement dangereuse. En effet, le bug se trouve dans la classe **Canari**. Nous l'y avons introduit en modifiant la classe parente **Animal**. De plus, il n'ap-

paraît pas si l'on compile ces deux classes séparément, ou même ensemble, sans le programme principal (la classe **Animaux**).

La raison pour laquelle le programme ne fonctionne pas apparaît clairement lorsque vous compilez la version ainsi modifiée. En effet, la tentative de compilation produit le message d'erreur suivant :

```
Animaux.java:3: No Constructor matching Canari() found in class Ca-
nari.
                Canari titi = new Canari();
                              ^
1 error
```

En effet, la classe parente **Animal** comporte bien un constructeur sans paramètre, mais la classe **Canari** n'en comporte pas !

Modifiez maintenant la classe **Canari** en supprimant simplement son constructeur, de la façon suivante :

```
class Canari extends Animal {
   void crie() {
      System.out.println("Cui-cui !");
   }
}
```

Miracle ! Le programme est maintenant compilé sans erreur ! Demi-miracle seulement car, si vous tentez maintenant de créer une instance de **Canari** avec un paramètre, vous obtenez de nouveau un message d'erreur !

Que s'est-il passé ? Pour que notre programme fonctionne, il faut que la classe **Canari** possède un constructeur dont la signature corresponde à la syntaxe que nous utilisons pour créer une instance. Lorsque nous supprimons le constructeur de cette classe, Java fournit automatiquement un *constructeur par défaut*. Le constructeur par défaut est le suivant :

```
NomDeLaClasse() {}
```

c'est-à-dire qu'il ne comporte aucun paramètre et qu'il ne fait rien... en apparence ! En fait, lors de la création d'une instance d'une classe, Java appelle automatiquement le constructeur de la classe parente. Cependant cet appel se fait sans passer le ou les paramètres correspondants. Nous pouvons le vérifier en modifiant le programme de la façon suivante :

```
class Animaux {
   public static void main(String[] args) {
      Canari titi = new Canari(3);
      titi.vieillit();
      titi.vieillit(2);
      titi.crie();
   }
}
class Animal {
   boolean vivant;
   int âge;
   int âgeMaxi = 10;


   Animal() {
      this(1);
   }

   Animal (int a) {
      âge = a;
      vivant = true;
      System.out.println("Un animal de " + a
           + " an(s) vient d'être créé");
   }

   void vieillit() {
      vieillit(1);
   }

   void vieillit(int a) {
      âge += a;
```

```
      afficheAge();
      if (âge > âgeMaxi) {
         meurt();
      }
   }

   void afficheAge() {
      System.out.println("C'est l'anniversaire de cet animal.");
      System.out.println("Il a maintenant " + âge + " an(s)");
   }

   void meurt() {
      vivant = false;
      System.out.println("Cet animal est mort");
   }

   void crie() {
   }
}

class Canari extends Animal {
   Canari(int a) {
   }
   void crie() {
      System.out.println("Cui-cui !");
   }
}
```

(Nous avons supprimé la troisième ligne de la classe **Canari**.)

Ce programme est compilé sans erreurs, et produit le résultat suivant :

```
Un animal de 1 an(s) vient d'être créé.
C'est l'anniversaire de cet animal.
Il a maintenant 2 an(s).
C'est l'anniversaire de cet animal.
Il a maintenant 4 an(s).
```

```
Cui-cui !
```

ce qui n'est évidemment pas le résultat escompté. Ce type de bug est le plus dangereux. Ici, dans un programme de soixante-dix lignes, il ne saute déjà pas aux yeux, alors imaginez le cas d'un programme de plusieurs dizaines de milliers de lignes ! La principale difficulté est de localiser l'erreur. Toutes les classes fonctionnent, mais le résultat obtenu est correct pour certaines valeurs, et pas pour d'autres.

La raison de ce comportement est que, lorsqu'un constructeur ne commence pas par un appel d'un autre constructeur au moyen de :

```
this(arguments); // Autre constructeur de la même classe
```

ou :

```
super(arguments); // constructeur de la classe parente
```

Java invoque automatiquement le constructeur sans argument de la classe parente. Ce qui est équivalent à :

```
super();
```

Note : Bien entendu, cela n'est pas valable pour la classe **Object**, qui n'a pas de classe parente.

Retenez donc cette règle fondamentale, dont la connaissance vous évitera bien des ennuis :

Tout constructeur dont la définition ne commence pas explicitement par this(ou super(commence implicitement par super().

Cette règle a un corollaire encore plus important :

Si vous voulez invoquer explicitement un autre constructeur à l'aide de this ou super, vous devez le faire sur la première ligne de la définition.

Dans le cas contraire, l'invocation implicite aura déjà eu lieu.

Il est un autre point important que vous ne devez pas oublier. En l'absence de tout constructeur, Java fournit un constructeur par défaut sans argument. Cependant, si un constructeur avec argument(s) existe, Java ne fournit pas de constructeur par défaut sans argument.

Nous pouvons maintenant modifier le programme pour que tout fonctionne correctement.

```
class Animaux {
   public static void main(String[] args) {
      Canari titi = new Canari(3);
      titi.vieillit();
      titi.vieillit(2);
      titi.crie();
   }
}
class Animal {
   boolean vivant;
   int âge;
   int âgeMaxi = 10;

   Animal() {
      this(1);
   }

   Animal (int a) {
      âge = a;
      vivant = true;
      System.out.println("Un animal de " + a
            + " an(s) vient d'être créé");
   }

   void vieillit() {
      vieillit(1);
   }
```

```
   void vieillit(int a) {
      âge += a;
      afficheAge();
      if (âge > âgeMaxi) {
         meurt();
      }
   }

   void afficheAge() {
      System.out.println("C'est l'anniversaire de cet animal.");
      System.out.println("Il a maintenant " + âge + " an(s)");
   }

   void meurt() {
      vivant = false;
      System.out.println("Cet animal est mort");
   }

   void crie() {
   }
}

class Canari extends Animal {
   Canari() {
   }

   Canari(int a) {
      super(a);
   }

   void crie() {
      System.out.println("Cui-cui !");
   }
}
```

Note : Nous verrons dans un prochain chapitre que Java permet le traitement des erreurs d'exécution au moyen des *exceptions*. Les exceptions sont

des objets Java qui sont *lancés* en cas d'erreur et peuvent être *attrapés* à l'aide d'une structure particulière. Cela permet au programmeur de traiter des erreurs sans que le programme soit interrompu. Par exemple, si le programme tente d'ouvrir un fichier inexistant, le programme peut afficher un message d'erreur pour demander à l'utilisateur d'y remédier, au lieu de se terminer brutalement. Les instructions donnant lieu au traitement d'erreurs doivent être insérées dans un bloc commençant par :

```
try {
```

Par conséquent, aucun traitement d'erreur n'est possible dans un constructeur invoqué à l'aide de **this** ou **super**.

Une autre conséquence de la façon dont fonctionnent les constructeurs est que le constructeur de la classe parente est toujours exécuté avant celui de la classe instanciée. De cette façon, on devrait pouvoir être assuré que si une classe initialise des variables de façon dépendante de variables de la classe parente, il n'y aura aucun risque d'utiliser des variables non encore initialisées. En fait, il existe plusieurs moyens de contourner cette sécurité, en particulier en invoquant dans un constructeur des méthodes manipulant des variables d'une classe dérivée. Bien sûr, nous éviterons ce type de manipulation.

Les initialiseurs

Les constructeurs ne sont pas le seul moyen d'initialiser les objets. En fait, il existe en Java trois éléments pouvant servir à l'initialisation : les constructeurs, que nous venons d'étudier, les *initialiseurs de variables d'instances*, que nous avons déjà utilisés sans les étudier explicitement, et les *initialiseurs d'instances*.

Les initialiseurs de variables d'instances

Nous avons vu que nous pouvions déclarer une variable de la façon suivante :

```
int a;
```

Nous créons ainsi une variable de type **int** dont le nom est **a**. Cette variable a-t-elle une valeur ? Comme nous l'avons déjà dit, cela dépend. Si cette déclaration se trouve dans une méthode, la variable n'a pas de valeur. Toute tentative d'y faire référence produit une erreur de compilation.

En revanche, s'il s'agit d'une *variable d'instance*, c'est-à-dire si cette déclaration se trouve en dehors de toute méthode, Java l'initialise automatiquement avec une valeur par défaut, comme nous l'avons vu au Chapitre 4. Nous pouvons cependant initialiser nous-mêmes les variables de la façon suivante :

```
int a = 5;
int b = a * 7;
float c = (b - a) / 3;
boolean d = (a < b);
float e = (b != 0) ? (float)a / b : 0;
```

Sur chaque ligne, on trouve à gauche du premier signe = la déclaration d'une variable, et à droite un *initialiseur de variable.* Il peut s'agir de la forme la plus simple (une valeur littérale, comme à la première ligne) ou d'une formule plus complexe, comme celle de la cinquième ligne, qui comporte un test logique, une division et un cast d'un **int** en **float**. (La signification de cette expression sera étudiée dans un prochain chapitre. Si vous êtes curieux, sachez que sa valeur est *a/b* si *b* est différent de 0 et 0 dans le cas contraire.)

Les initialiseurs de variables permettent d'effectuer des opérations d'une certaine complexité, mais celle-ci est tout de même limitée. En effet, ils doivent tenir sur une seule ligne. Pour effecteur des initialisations plus complexes, nous savons que nous pouvons exécuter celles-ci dans un constructeur. Une autre possibilité consiste à utiliser un *initialiseur d'instance*.

Les initialiseurs d'instances

Un initialiseur d'instance est tout simplement un bloc placé, comme les variables d'instances, à l'extérieur de toute méthode ou constructeur. Par exemple, nous pouvons initialiser la variable **e** de l'exemple précédent de la façon suivante :

```
class Initialiseurs {
   public static void main(String[] args) {
   Init init = new Init();
   System.out.println("a = " + init.a);
   System.out.println("b = " + init.b);
   System.out.println("c = " + init.c);
   System.out.println("d = " + init.d);
   System.out.println("e = " + init.e);
   }
}

class Init {
   int a = 5;
   int b = a * 7;
   float c = (b - a) / 3;
   boolean d = (a < b);
   float e;
   {
      if (b != 0) {
         e = (float)a/b;
      }
   }
}
```

(Nous avons intégré ces initialisations à une classe de façon à pouvoir compiler le programme sans erreurs.)

Ici, la classe **Init** ne fait rien d'autre qu'initialiser ses variables. La variable **e** est initialisée à la même valeur que dans l'exemple précédent. En fait, le cas où **b** vaut **0** n'est pas traité car, s'agissant d'une variable d'instance, **e** est

d'abord initialisée à **0**. Si **b** est égal à **0**, aucune autre initialisation de **e** n'est alors nécessaire.

Les initialiseurs d'instances permettent d'effectuer des initialisations plus complexes que les initialiseurs de variables. Ils offrent en outre l'avantage par rapport aux constructeurs d'être beaucoup plus rapides. Invoquer un constructeur est une opération plus longue, surtout si la classe a une ascendance complexe. Dans ce cas, en effet, les constructeurs de tous les ancêtres doivent être invoqués. De plus, les initialiseurs permettent d'utiliser les exceptions pour traiter les conditions d'erreurs.

Un autre avantage des initialiseurs est qu'ils permettent d'effectuer un traitement quelle que soit la signature du constructeur utilisé. Il est ainsi possible de placer le code d'initialisation commun à tous les constructeurs dans un initialiseur et de ne traiter dans les différents constructeurs que les opérations spécifiques.

Un dernier avantage des initialiseurs est qu'ils permettent d'effectuer des initialisations dans des classes qui ne peuvent pas comporter de constructeurs. En effet, tout comme il est possible de créer des objets anonymes, qui sont utilisés seulement au moment de leur création, Java permet de créer "au vol" des classes anonymes. Les constructeurs devant porter le même nom que leur classe, les classes anonymes ne peuvent pas comporter de constructeurs. (Nous reviendrons plus loin sur les classes anonymes, lorsque nous aurons introduit les classes internes.)

Les initialiseurs comportent cependant des limitations. Il n'est pas possible de leur passer des paramètres comme dans le cas des constructeurs. De plus, ils sont exécutés avant les constructeurs et ne peuvent donc utiliser les paramètres de ceux-ci.

Ordre d'initialisation

L'exécution des initialiseurs a lieu dans l'ordre dans lequel ils apparaissent dans le code. Normalement, il n'est pas possible de faire référence à une variable qui n'a pas encore été initialisée. Par exemple, le code suivant :

```
class OrdreInitialisation {
   public static void main(String[] args) {
   Init init = new Init();
   System.out.println("a = " + init.a);
   System.out.println("b = " + init.b);
   }
}

class Init {
   int b = a;
   int a = 5;
}
```

provoque une erreur de compilation. Le compilateur voit qu'il s'agit d'une *référence en avant* et refuse d'exécuter l'initialisation. Cependant, il est possible, dans certains cas, d'arriver au même résultat sans provoquer d'erreur de compilation. Le programme suivant, par exemple, est compilé sans erreur :

```
class OrdreInitialisation {
   public static void main(String[] args) {
   Init init = new Init();
   System.out.println("a = " + init.a);
   System.out.println("b = " + init.b);
   }
}

class Init {
   int b = calcul();
   int a = 5;

   int calcul() {
      return a;
   }
}
```

Cependant, son exécution produit le résultat suivant :

```
a = 5
b = 0
```

En effet, Java effectue d'abord toutes les déclarations avant d'effectuer les initialisations. En d'autres termes, lorsque le programme comporte les instructions :

```
int a = 5;
int b = 8;
```

Java effectue d'abord la déclaration de la variable **a**, c'est-à-dire qu'il réserve l'espace mémoire nécessaire pour le type de donnée déclaré (ici un **int**) et met en place l'identificateur correspondant (**a**). A ce moment, la zone de mémoire considérée contient une valeur quelconque. Laisser cette valeur telle quelle pourrait conduire, dans certains cas, à un bug. Pour éviter cela, Java dispose de deux techniques. La première consiste à empêcher le programme d'être exécuté, en produisant une erreur de compilation. Cette technique est utilisée pour les variables qui ne sont pas des variables d'instances. La seconde technique consiste à initialiser immédiatement cette zone de mémoire. C'est ce que fait Java pour les variables d'instances, et ce même si la déclaration est suivie d'un initialiseur. Toutes les variables d'instances sont donc ainsi créées et initialisées à leurs valeurs par défaut avant que la première initialisation explicite soit exécutée. En d'autres termes, les deux lignes ci-dessus sont équivalentes à :

```
int a;
a = 0;
int b;
b = 0;
a = 5;
b = 8;
```

L'exemple précédent était donc équivalent à :

```
class Init {
   int b;
   b = 0;
   int a;
   a = 0;
   b = calcul();
   a = 5;

   int calcul() {
      return a;
   }
}
```

On voit immédiatement que la variable **a** existe et a bien une valeur au moment où la méthode **calcul()** est appelée pour affecter une valeur à **b**. Cette valeur est 0. Le compilateur n'a pas été capable de détecter la référence en avant, mais Java parvient quand même à éviter un problème grave grâce à l'initialisation systématique et immédiate des variables d'instances. Ce programme ne produira pas d'erreur, mais ne s'exécutera pas correctement, en ce sens que la variable **b** ne sera pas initialisée à la valeur escomptée. Cependant, du fait que **b** aura pris la valeur par défaut 0, le dysfonctionnement sera plus facile à localiser que si elle avait pris une valeur aléatoire en fonction du contenu antérieur de la zone de mémoire concernée.

Méthodes : les valeurs de retour

Jusqu'ici, nous avons utilisé des méthodes et des constructeurs en précisant qu'il s'agissait de deux éléments différents. Une des différences est que les méthodes peuvent avoir une valeur de retour. La valeur de retour d'une méthode est la valeur fournie par cette méthode lorsqu'elle *retourne*, c'est-à-dire lorsqu'elle se termine.

Le type de la valeur de retour d'une méthode est indiqué avant son nom dans sa déclaration. Les méthodes que nous avons employées jusqu'ici possédaient presque toutes une valeur de retour de type **void**, ce qui n'équivaut à aucune valeur de retour. De plus, ces méthodes retournaient toutes à la fin

du code. Lorsque ces deux conditions sont réunies, il n'est pas obligatoire que la méthode retourne explicitement. En d'autres termes, la méthode :

```
void crie() {
   System.out.println("Cui-cui !");
}
```

devrait être écrite :

```
void crie() {
   System.out.println("Cui-cui !");
   return;
}
```

mais l'utilisation de **return** est facultative car, d'une part, le code est terminé et, d'autre part, il n'y a aucune valeur à retourner. Dans l'exemple suivant :

```
void crie() {
   System.out.println("Cui-cui !");
   return;
   System.out.println("Aie !");
}
```

le deuxième message ne sera jamais affiché car la méthode retourne explicitement à la troisième ligne.

L'instruction **return** est souvent utilisée pour un retour conditionnel, du type :

```
si (condition) {
   return;
}
suite du traitement
```

Dans ce cas, si la condition est remplie, la méthode retourne (c'est-à-dire s'arrête). Dans le cas contraire, le traitement continue.

L'autre fonction de **return** est de renvoyer la valeur de retour, comme dans la méthode suivante :

```
int calcul() {
   return a;
}
```

Ici, la méthode ne fait rien. Elle retourne simplement la valeur d'une variable. En général, les méthodes qui retournent des valeurs effectuent un certain nombre de calculs avant de retourner. Par exemple, la méthode suivante calcule le carré d'un nombre :

```
long carré(int i) {
   return i * i;
}
```

Tout comme les constructeurs, les méthodes peuvent être surchargées. Ainsi, une méthode élevant une valeur à une puissance pourra être surchargée de la façon suivante :

```
double puissance(float i) {
   return i * i;
}

double puissance(float i, int j)
   double résultat = 1;
   for (int k = 1; k <= j; k++) {
      résultat *= i;
   }
   return résultat;
}
```

Dans cet exemple, la méthode **puissance()** retourne son premier argument élevé à la puissance de son second argument si elle est appelée avec un **float** et un **int**. Si elle est appelée avec pour seul argument un **float**, elle retourne cette valeur au carré. La valeur de retour est un **double** dans les deux cas.

Notez que si vous appelez la méthode **puissance()** avec un **int** et un **float**, le compilateur protestera qu'il ne trouve pas de version de la méthode **puissance()** correspondant à cette signature. Parfois, il peut être intéressant, si vous écrivez une méthode devant être utilisée par d'autres programmeurs, de la surcharger avec des versions utilisant les mêmes arguments dans un ordre différent. Par exemple, la méthode suivante prend pour argument une chaîne de caractères et l'affiche une fois :

```
void affiche(String s) {
   System.out.println (s);
}
```

Vous pouvez la surcharger avec la version suivante, qui affiche plusieurs fois la chaîne :

```
void affiche(String s) {
   System.out.println (s);
}
void affiche(String s, int i) {
   for (int j = 0; j < i; j++) {
      System.out.println (s);
   }
}
```

Si la méthode est appelée avec une chaîne pour argument, la chaîne est affichée une fois. Si elle est appelée avec une chaîne et un **int** (dans cet ordre), la chaîne est affichée le nombre de fois indiqué par la valeur entière. Si vous voulez que l'utilisateur de la méthode n'ait pas à mémoriser l'ordre des paramètres, il vous suffit de surcharger encore une fois la méthode de la façon suivante :

```
void affiche(String s) {
   System.out.println (s);
}
void affiche(String s, int i) {
   for (int j = 0; j < i; j++) {
      System.out.println (s);
   }
}
void affiche(int i, String s) {
   for (int j = 0; j < i; j++) {
      System.out.println (s);
   }
}
```

La méthode pourra alors être appelée des différentes façons suivantes :

```
affiche("chaîne 1");
affiche("chaîne 2", 3);
affiche(6, "chaîne 3");
```

Surcharger une méthode avec une méthode de type différent

Pour surcharger une méthode, il n'est pas obligatoire d'employer une méthode du même type. Il est tout à fait possible de surcharger une méthode de type **int** (par exemple) à l'aide d'une méthode de type **float**.

Dans l'exemple suivant, nous créons une classe **CalculCarre** qui contient la méthode **carré()**, calculant le carré d'une valeur numérique. La première version est de type **int** et prend un paramètre de type **short**. La deuxième est de type **float** et prend un paramètre de type **int**. La troisième est de type **double** et prend un paramètre de type **long**.

```
class Surcharge {
   public static void main(String[] args) {
   short a = 32767;
   int b = 2147483;
```

```
      long c = 2147483647214748364L;
      System.out.println(CalculCarre.carré(a));
      System.out.println(CalculCarre.carré(b));
      System.out.println(CalculCarre.carré(c));
      }
   }
   class CalculCarre {
      static int carré(short i) {
         System.out.println("int carré(short i)");
         return i * i;
      }
      static float carré(int i) {
         System.out.println("Version float carré(int i)");
         return i * i;
      }
      static double carré(long i) {
         System.out.println("Version double carré(long i)");
         return i * i;
      }
   }
```

Chaque version affiche un message afin de montrer quelle version est exécutée.

Note : La classe est nommée **CalculCarre**, sans accent, afin d'éviter les erreurs de compilation. Rappelons que les noms de variables et de méthodes peuvent être accentués, mais pas les noms de classes.

Attention : Cet exemple ne sert qu'à démontrer la possibilité de surcharge par des méthodes de types différents. Si vous avez besoin d'une méthode pour élever une valeur quelconque au carré, ne suivez surtout pas ce modèle !

Distinction des méthodes surchargées par le type uniquement

Dans l'exemple précédent, les méthodes surchargées diffèrent par leur type **et** par leur signature. Vous vous demandez peut-être s'il est possible de

distinguer les méthodes surchargées uniquement par leur type, comme dans l'exemple suivant :

```
class Surcharge2 {
   public static void main(String[] args) {
   short a = 5;
   int b = Calcul.calculer(a);
   long c = Calcul.calculer(a);
   System.out.println(a);
   System.out.println(b);
   System.out.println(c);
   }
}

class Calcul {
   static int calculer(int i) {
      return i * 2;
   }
   static long calculer(int i) {
      return i * i;
   }
}
```

La réponse est non. En effet, ici, Java pourrait déterminer quelle méthode doit être employée en fonction du type utilisé pour l'appel de la méthode. Ainsi, à la ligne :

```
int b = Calcul.calculer(a);
```

la méthode :

```
static int calculer(int i) {
   return i * 2;
}
```

serait utilisée puisque son résultat est affecté à un **int**. Cependant, cela entrerait en conflit avec le fait qu'il est tout à fait possible d'écrire et de compiler sans erreur le progamme suivant :

```
class Surcharge2 {
   public static void main(String[] args) {
   short a = 5;
   int b = Calcul.calculer(a);
   long c = Calcul.calculer(a);
   System.out.println(a);
   System.out.println(b);
   System.out.println(c);
   }
}

class Calcul {
   static int calculer(int i) {
      return i * 2;
   }
}
```

Dans ce cas, la ligne :

```
long c = Calcul.calculer(a);
```

appelle la méthode **calculer()**, qui renvoie un **int**, mais le résultat est affecté à un **long**. Il s'agit, comme nous l'avons indiqué dans un précédent chapitre, d'un cast implicite.

Par ailleurs, dans notre classe **Surcharge**, nous avions effectué des appels de méthodes sans affecter les résultats à des variables typées, comme dans l'exemple :

```
System.out.println(CalculCarre.carré(a));
```

Ici, il serait impossible au compilateur de sélectionner une méthode sur la base du type utilisé pour stocker le résultat, puisque le résultat n'est pas stocké. En fait, comme vous pouvez le supposer, la méthode **System.out.println()** est surchargée pour traiter des arguments de toutes natures. C'est donc la valeur de retour fournie par **CalculCarre.carre(a)** qui indique à Java quelle version de **System.out.println()** doit être utilisée, et non l'inverse.

Le retour

Le retour d'une méthode ne consiste pas seulement à fournir une valeur de retour. Lorsqu'une méthode retourne, son traitement est terminé. Une méthode retourne dans deux conditions :

- Son code a été exécuté en entier.
- L'instruction **return** a été exécutée.

Ainsi, la méthode suivante :

```
void affiche(String s, int i) {
   for (int j = 0; j < i; j++) {
      System.out.println(s);
   }
   System.out.println("C'est fini !");
}
```

affiche **i** fois la chaîne **s**, puis le message **"C'est fini !"**, et retourne, exactement comme si on avait écrit :

```
void affiche(String s, int i) {
   for (int j = 0; j < i; j++) {
      System.out.println(s);
   }
   System.out.println("C'est fini !");
   return;
}
```

En revanche, la méthode suivante n'affichera jamais le message "C'est fini !" :

```
void affiche(String s, int i) {
   for (int j = 0; j < i; j++) {
      System.out.println (s);
   }
   return;
   System.out.println("C'est fini !");
}
```

car l'instruction **return** est exécutée et fait retourner la méthode avant que sa dernière ligne soit exécutée. Dans l'exemple suivant, la chaîne **s** n'est affichée qu'une seule fois :

```
void affiche(String s, int i) {
   for (int j = 0; j < i; j++) {
      System.out.println (s);
      return;
   }
   System.out.println("C'est fini !");
}
```

car l'instruction **return** est exécutée immédiatement après le premier affichage de la chaîne, ce qui a pour effet de faire retourner la méthode immédiatement.

Résumé

Dans ce chapitre, nous avons commencé à étudier la façon dont les classes sont construites en dérivant les unes des autres. Nous nous sommes intéressés en particulier aux mécanismes servant à contrôler la création des instances de classes (les initialiseurs et les constructeurs) et nous avons vu en quoi ils sont fondamentalement différents des méthodes.

La différence entre constructeurs et méthodes est particulièrement importante, car la façon dont ils sont construits offre bien des similitudes. Cependant, du point de vue du programmeur Java, ce sont des éléments très différents, comme l'attestent les particularités suivantes :

- Les constructeurs n'ont pas de type.
- Les constructeurs ne retournent pas.
- Les constructeurs ne sont pas hérités par les classes dérivées.
- Lorsqu'un constructeur est exécuté, les constructeurs des classes parentes le sont également.
- Une méthode peut porter le même nom qu'un constructeur (ce qui est toutefois formellement déconseillé).

Les constructeurs et les initialiseurs sont des éléments très importants car ils déterminent la façon dont les objets Java commencent leur existence. Nous verrons dans un prochain chapitre que Java propose également divers moyens de contrôler comment les objets finissent leur existence, ce qui est un problème tout aussi important.

Exercice

A titre d'exercice, examinez le programme suivant et déterminez :

- S'il est correctement écrit ;
- S'il peut être compilé sans erreur.

Dans la négative, comment pouvez-vous corriger ce programme sans modifier la classe **Animal** ?

```
class Animaux2 {
   public static void main(String[] args) {
```

```
                Canari titi = new Canari(3);
                titi.vieillit();
      titi.vieillit(2);
      titi.crie();
   }
}
class Animal {
   boolean vivant;
   int âge;
   int âgeMaxi = 10;

   Animal (int a) {
      âge = a;
      vivant = true;
      System.out.println("Un animal de " + a
           + " an(s) vient d'être créé");
   }
   void vieillit() {
      vieillit(1);
   }
   void vieillit(int a) {
      âge += a;
      afficheAge();
      if (âge > âgeMaxi) {
         meurt();
      }
   }
   void afficheAge() {
      System.out.println("C'est l'anniversaire de cet animal.");
      System.out.println("Il a maintenant " + âge + " an(s)");
   }
   void meurt() {
      vivant = false;
      System.out.println("Cet animal est mort");
   }
   void crie() {
   }
}
```

```
class Canari extends Animal {
   Canari(int a) {
      âge = a + 2;
      vivant = true;
      System.out.println("Un canari de " + âge
            + " an(s) vient d'être créé");
   }
   void crie() {
      System.out.println("Cui-cui !");
   }
}
```

Une fois que vous avez répondu aux questions, essayez de compiler ce programme. Vous obtiendrez un message d'erreur vous indiquant qu'aucun constructeur sans argument n'a été trouvé pour la classe **Animal**, et ce, bien qu'à aucun moment le programme ne tente de créer une instance d'animal sans argument. La raison de ce problème vient de ce que le constructeur de la classe **Canari** ne commence ni par **this** ni par **super**. Java invoque donc implicitement le constructeur sans argument de la classe parente **Animal**. Or, cette classe n'en possède pas. De plus, elle possède un constructeur avec argument(s), ce qui empêche Java de créer un constructeur par défaut. Vous ne pouvez pas corriger ce programme sans modifier la classe **Animal**. En effet, vous pouvez modifier le constructeur de la classe **Canari** de la façon suivante :

```
Canari(int a) {
   super(a + 2);
   System.out.println("Un canari de " + âge
         + " an(s) vient d'être créé");
}
```

Toutefois, dans ce cas, vous obtiendrez l'affichage de deux messages : celui affiché par le constructeur de la classe **Animal** :

```
Un animal de 5 an(s) vient d'être créé.
```

et celui affiché par le constructeur de la classe **Canari** :

```
Un canari de 5 an(s) vient d'être créé.
```

Un bon compromis pourrait être :

```
Canari(int a) {
   super(a + 2);
   System.out.println("C'est un canari.");
}
```

ce qui permettrait d'obtenir le résultat suivant :

```
Un animal de 5 an(s) vient d'être créé.
C'est un canari.
```

Chapitre 6

Les opérateurs

Le langage Java peut être décomposé en trois parties : la syntaxe, qui regroupe l'ensemble des mots clés, les opérateurs, ainsi que les règles gouvernant leur utilisation, la "sémantique" des classes et des objets, qui regroupe tous les aspects des règles concernant l'organisation des classes et leur manipulation, ainsi que celle des objets, et les packages standard, qui ne font pas vraiment partie intégrante du langage, mais sans lesquels celui-ci serait très peu efficace, puisqu'il faudrait au programmeur créer toutes les classes dont il a besoin pour réaliser une application. Si l'on examine le problème d'un peu plus près, on s'aperçoit rapidement qu'un certain nombre de classes sont indispensables au programmeur Java. En effet, un programme Java ne tourne pas (encore) dans un environnement 100 % Java. Il faut donc gérer l'interaction des applications écrites en Java avec l'environnement (l'ordinateur sur lequel l'application est utilisée et son système d'exploitation.) Ainsi, pour afficher une chaîne de caractères à l'écran, l'application doit dialoguer avec le système. Ce type de dialogue peut être pris en

charge par l'interpréteur de *bytecode*. Cependant, le nombre de cas différents à traiter est tel qu'il est beaucoup plus rentable d'utiliser des classes toutes faites, dont certaines font éventuellement appel directement au système, par l'intermédiaire de méthodes dites *natives*. Java est ainsi fourni avec un ensemble de classes regroupées en *packages*, selon leur fonction. Les packages en général feront l'objet d'un prochain chapitre. Certains packages, comme ceux concernant l'interface utilisateur (les boutons, les menus, les fenêtres, etc.) feront l'objet d'une étude approfondie dans les chapitres suivants. L'étude de la sémantique des classes et des objets a déjà été entamée dans les chapitres précédents, et nous y reviendrons car il reste beaucoup à dire. C'est dans ce domaine que s'exprime l'essence même des langages orientés objets en général, et de Java en particulier.

La syntaxe est un sujet beaucoup moins passionnant, mais dont l'étude est tout de même indispensable. Nous l'avons déjà employée sans l'expliquer en détail (on ne peut pas écrire une ligne de Java sans en utiliser la syntaxe). Nous allons maintenant essayer d'en faire le tour. C'est un sujet beaucoup moins intéressant et plus rébarbatif que ce que nous avons étudié jusqu'ici, mais il n'y a malheureusement pas moyen d'y échapper.

Java et les autres langages

On prétend souvent que Java dérive de tel ou tel langage. Du point de vue sémantique, on peut clairement établir une filiation avec des langages objets tels que C++ ou Smalltalk. En ce qui concerne la syntaxe, Java ressemble à beaucoup de langages. En effet, la grande majorité des langages utilisent la même syntaxe. Quelques instructions peuvent manquer à certains langages, certains raccourcis d'écriture peuvent être possibles avec les uns et pas avec les autres, mais ces différences sont très superficielles. (Il est tout de même des langages plus exotiques que d'autres en ce qui concerne la syntaxe, comme Forth ou PostScript, qui utilisent la métaphore de la pile.)

La syntaxe de Java est clairement calquée sur celle de C, et donc également de C++. Cependant, ne vous inquiétez pas si vous ne connaissez pas ces langages. La syntaxe des langages de programmation est une chose extrêmement facile à maîtriser. Celle de Java, en tout cas, est d'une grande sim-

plicité, comparée aux concepts mis en jeu par les aspects sémantiques du langage.

Les opérateurs Java s'utilisent pratiquement de la même façon que dans la vie courante, du moins pour les opérateurs binaires, c'est-à-dire ceux prenant deux opérandes, qui utilisent la notation dite *infixée*, dans laquelle un opérateur est placé entre les opérandes. C'est la notation que vous avez apprise à l'école.

L'opérateur d'affectation

Java utilise pour l'affectation le signe =, ce qui est une vieille habitude en programmation. Pour être vieille, cette habitude n'en est pas moins déplorable car elle entraîne un risque de confusion certain entre l'affectation :

```
x = 3
```

et la condition :

```
if x = 3
```

Cette confusion n'existe toutefois pas en Java, car la syntaxe de la condition a été remplacée par :

```
if (x == 3)
```

(Les parenthèses sont obligatoires en Java.)

Souvenez-vous donc simplement qu'en Java :

```
x = 3;
```

est une affectation, qui signifie "donner à l'élément se trouvant à gauche, la valeur de l'élément se trouvant à droite".

Notez que, dans le cas des primitives, l'affectation n'entraîne pas une égalité au sens courant. L'égalité au sens courant est plutôt une identité. Ici, nous avons à gauche un conteneur qui reçoit la valeur d'un autre conteneur. Ainsi :

```
y = 3 * 2;
x = y;
```

signifie, pour la première ligne, *mettre dans y le résultat de l'opération 3 * 2* et, pour la deuxième ligne, *copier dans x la valeur de y*. Les deux conteneurs *x* et *y* existent simultanément de façon indépendante. Ils ont chacun un contenu différent, dont les valeurs sont momentanément égales.

Dans le cas des objets, la situation est différente. Dans l'exemple suivant :

```
x = new Animal();
y = x;
```

la première ligne signifie *établir une relation entre le handle x et le résultat de l'instruction placée à droite du signe égal,* c'est-à-dire le nouvel objet créé. En revanche, la deuxième ligne signifie *établir une relation entre le handle y et l'élément en relation avec le handle x*. Dans ce cas, aucun objet nouveau n'est créé. Le "contenu" (c'est un abus de langage) de *x* est non seulement égal mais identique à celui de *y* (c'est le même !).

Raccourci

Java permet également une écriture raccourcie consistant à effectuer plusieurs affectations sur une même ligne, de la façon suivante :

```
x = y = z = w = v + 5;
```

Bien entendu, il faut que toutes les variables aient été déclarées, et que l'élément le plus à droite soit une expression évaluable. Dans cet exemple, il faut par conséquent que la variable *v* ait été initialisée.

Cette écriture est incompatible avec l'initialisation effectuée sur la même ligne que la déclaration. En revanche, dans ce cas, deux écritures raccourcies sont possibles :

```
int x = 5, y = 5, z = 5;
```

ou :

```
int x,y,z;
x = y = z = 5;
```

Bien sûr, la deuxième écriture n'est possible que si les variables sont initialisées avec la même valeur, alors que la première peut être employée quelles que soient les valeurs d'initialisation.

Les opérateurs arithmétiques à deux opérandes

Note : Nous n'utiliserons pas, comme certains auteurs, les termes d'opérateurs *binaires* et *unaires* pour désigner les opérateurs à deux opérandes et ceux à un opérande, de façon à éviter la confusion avec les opérateurs d'arithmétique binaire (manipulant des bits).

Les opérateurs arithmétiques à deux opérandes sont les suivants :

+ addition

- soustraction

* multiplication

/ division

% modulo (reste de la division)

Il faut noter que la division d'entiers produit un résultat tronqué, et non arrondi. La raison à cela est la cohérence des opérateurs appliqués aux entiers. En effet, la cohérence impose que pour toutes valeurs de *a* et *b* (sauf *b* = 0) :

*((a / b) * b) + (a % b) = a*

En d'autres termes, si *quotient* est le résultat et *reste* le reste de la division de *dividende* par *diviseur*, on doit avoir :

*(quotient * diviseur) + reste = dividende*

Si le quotient était arrondi, cette formule ne se vérifierait que dans 50 % des cas.

Java dispose cependant de moyens permettant d'effectuer des divisions avec des résultats arrondis.

Notez qu'il n'existe pas en Java d'opérateur d'exponentiation. Pour effectuer une exponentiation, vous devez utiliser la fonction **pow(double a, double b)** de la classe **java.lang.Math**.

Les opérateurs à deux opérandes et le sur-casting automatique

Java effectue automatiquement un sur-casting sur les données utilisées dans les expressions contenant des opérateurs à deux opérandes. Cela peut sembler normal dans un cas comme le suivant :

```
byte x = (byte)127;
System.out.println(x + 300);
```

Dans ce cas, Java sur-caste automatiquement la valeur **x** (du type **byte**) en type **int**. Le résultat produit est de type **int**. En revanche, cela paraît moins naturel dans le cas suivant :

```
byte x = (byte)12;
byte y = (byte)13;
System.out.println(x + y);
```

Ici, bien que les deux arguments soient de type **byte** et que le résultat ne dépasse pas les limites du type **byte**, Java effectue tout de même un surcasting. Le résultat **x + y** est en effet de type **int**. Pour le prouver, nous utiliserons le programme suivant :

```
class SurCast {
   public static void main(String[] args) {
      byte x = (byte)12;
      byte y = (byte)13;
      afficheType(x + y);
   }
   static void afficheType (byte x) {
      System.out.print("L'argument est de type byte.");
   }
   static void afficheType (short x) {
      System.out.print("L'argument est de type short.");
   }
   static void afficheType (char x) {
      System.out.print("L'argument est de type char.");
   }
   static void afficheType (int x) {
      System.out.print("L'argument est de type int.");
   }
   static void afficheType (long x) {
      System.out.print("L'argument est de type long.");
   }
   static void afficheType (float x) {
      System.out.print("L'argument est de type float.");
   }
   static void afficheType (double x) {
      System.out.print("L'argument est de type double.");
   }
}
```

Ce programme affiche :

```
L'argument est de type int.
```

Vous pouvez utiliser ce programme pour tester tous les cas. Vous constaterez que les opérations mettant en jeu des données de type **byte**, **char**, **short** ou **int** produisent un résultat de type **int**. Si un des opérandes est d'un autre type numérique, le résultat sera du type ayant la priorité la plus élevée, dans l'ordre suivant :

Priorité la plus faible : **int**
long
float
Priorité la plus élevée : **double**

En revanche, Java ne tient pas compte du débordement qui peut résulter de l'opération. Par exemple, le programme suivant :

```
class Debordement {
   public static void main(String[] args) {
      int x = 2147483645;
      int y = 2147483645;
      System.out.println(x + y);
   }
}
```

affiche gaillardement le résultat suivant :

```
-6
```

sans provoquer le moindre message d'erreur !

Ce comportement, outre les risques de bug difficile à déceler, induit un autre problème. Vous ne pouvez pas écrire :

```
byte x = (byte)12;
x = x * 2;
```

car le résultat de **x * 2** est un **int** et vous ne pouvez pas affecter un **int** à un **byte** sans effectuer un cast explicite, de la façon suivante :

```
byte x = (byte)12;
x = (byte)(x * 2);
```

Notez qu'ici, les parenthèses sont obligatoires autour de x * 2, car les opérateurs de cast ont la priorité sur les opérateurs arithmétiques. De ce fait :

```
(byte)x * 2
```

équivaut à :

```
((byte)x) * 2
```

ce qui ne produit évidemment pas le résultat escompté.

Vous trouvez peut-être tout cela bien gênant. Malheureusement, cela n'est pas tout, comme nous allons le voir dans la section suivante.

Les raccourcis

Java propose un raccourci permettant de condenser l'utilisation d'un opérateur binaire et de l'opérateur d'affectation. En effet, on utilise souvent ces deux types d'opérateurs sur une même ligne, de la façon suivante :

```
x = x + 4;
z = z * y;
v = v % w;
```

Chaque fois qu'une ligne commence par une affectation du type :

```
x = x opérateur opérande;
```

vous pouvez la remplacer par :

```
x opérateur= opérande;
```

soit, dans notre exemple précédent :

```
x += 4;
z *= y;
v %= w;
```

On ne peut pas dire que la lisibilité en soit grandement améliorée ! Alors, où est l'avantage ? Cela fait plus sérieux, car cela ressemble à du C. Des tests portant sur 10 millions d'opérations font apparaître des résultats inégaux en matière de performances, par exemple un gain de 0,006 % pour l'addition, rien du tout pour la multiplication. Autant dire qu'en ce qui nous concerne, la différence n'est guère significative. Vous choisirez donc l'une ou l'autre notation en fonction de vos goûts. Vous devrez toutefois tenir compte d'un aspect important concernant la fiabilité des programmes. En effet, les combinaisons *opérateurs* + *affectation* présentent l'inconvénient (ou l'avantage, c'est selon) de procéder à un sous-casting implicite sans en avertir le moins du monde le programmeur. Souvenez-vous que le code suivant :

```
byte x = (byte)12;
x = x * 2;
```

produit un message d'erreur puisque l'on tente d'affecter le résultat de l'opération **x * 2**, de type **int**, à un **byte**. En revanche, le code suivant est compilé sans erreur :

```
byte x = (byte)12;
x *= 2;
```

Ici, cela ne pose pas de problème. En revanche, avec d'autres valeurs, il n'en est pas de même. Le code suivant :

```
byte x = (byte)120;
x *= 2;
```

ne produit pas d'erreur non plus. Nous sommes pourtant en présence d'un débordement, comme dans le cas mis en évidence précédemment avec le type **int**.

Pour résumer, Java ne traite pas les problèmes de débordement. Il se contente de réduire les risques en transformant les **byte**, **char** et **short** en **int**. Malheureusement, cela est doublement insuffisant, d'une part parce que les débordements d'**int**, de **long**, de **float** et de **double** ne sont pas contrôlés, d'autre part parce que les raccourcis court-circuitent le mécanisme de cast automatique. Il revient donc au programmeur de traiter lui-même les risques de débordement.

Les opérateurs à un opérande

Java possède plusieurs types d'opérateurs à un opérande. Les plus simples sont les opérateurs + et -. Ils sont toujours préfixés. L'opérateur + ne fait rien. L'opérateur - change le signe de son opérande.

Attention : L'utilisation d'espaces pour séparer les opérateurs des opérandes n'est pas obligatoire. Cependant, l'absence d'espaces augmente le risque de confusion. Ainsi la notation suivante :

```
x=-5;
x-=4;
```

est moins facile à lire que :

```
x = -5;
x -= 4;
```

Ici, un bon usage des espaces améliore la lisibilité. Pas autant, toutefois, que l'utilisation de la notation suivante :

```
x = -5;
x = x - 4;
```

Java possède également des opérateurs d'auto-incrémentation et d'auto-décrémentation. Ils sont particulièrement utiles pour la réalisation de boucles de comptage. Il en existe deux versions : préfixée et postfixée.

```
x++;
++x;
x--;
--x;
```

x++ et **++x** ont tous deux pour effet d'augmenter la valeur de **x** de une unité, tout comme **x--** et **--x** ont l'effet inverse, c'est-à-dire qu'ils diminuent la valeur de **x** d'une unité. La différence entre la version préfixée et la version postfixée n'apparaît que lorsque l'opération est associée à une affectation ou une utilisation du résultat.

Considérez l'exemple suivant :

```
class Incr {
   public static void main(String[] args) {
      int x = 5;
      int y = x++;
      System.out.println("x = " + x);
      System.out.println("y = " + y);
      y = ++x;
      System.out.println("x = " + x);
      System.out.println("y = " + y);
   }
}
```

Ce programme affiche le résultat suivant :

```
x = 6
y = 5
x = 7
y = 7
```

On voit que la ligne **y = x++** affecte à **y** la valeur de **x** puis incrémente **x**. **y** vaut donc 5 (la valeur de **x**) et **x** voit sa valeur augmentée de 1 et vaut donc 6. En revanche, la ligne **y = ++x** augmente d'abord la valeur de **x** (qui vaut donc maintenant 7) puis affecte cette valeur à **y**.

Le même résultat est obtenu lorsque la valeur est fournie à une méthode au lieu d'être affectée à une variable. Par exemple, le programme suivant :

```
class Decr {
   public static void main(String[] args) {
      int x = 5;
      System.out.println(x);
      System.out.println(x--);
      System.out.println(x);
      System.out.println(--x);
      System.out.println(x);
   }
}
```

produit le résultat suivant :

```
5
5
4
3
3
```

ce qui met en évidence le fait que dans l'instruction **System.out. println(x--)**, **x** est passé à la méthode **System.out.println()** *avant* d'être décrémenté, alors que dans l'instruction **System.out. println(--x)**, il lui est passé *après*.

Note : Les opérateurs à un opérande ne produisent pas de sur-casting automatique, à l'inverse des opérateurs à deux opérandes.

Les opérateurs relationnels

Les opérateurs relationnels permettent de tester une relation entre deux opérandes et fournissent un résultat de type **boolean**, dont la valeur peut être **true** (vrai) ou **false** (faux). Java dispose des opérateurs relationnels suivants :

== équivalent

< plus petit que

> plus grand que

<= plus petit ou égal

>= plus grand ou égal

!= non équivalent

Les notions de *plus petit que* ou *plus grand que* s'appliquent aux valeurs numériques. Les notions d'équivalence et de non-équivalence s'appliquent à toutes les primitives ainsi qu'aux handles d'objets.

Il faut noter (et ne jamais oublier) que l'équivalence appliquée aux handles d'objets concerne les handles, et non les objets eux-mêmes. Deux handles sont équivalents s'ils pointent vers le même objet. Il ne s'agit donc pas d'objets égaux, mais d'un seul objet, ce qui correspond à la définition de l'identité. Considérez, par exemple, l'exemple suivant :

```
class Egal1 {
   public static void main(String[] args) {
   Integer a = new Integer(100);
   Integer b = new Integer(100);
   System.out.println(a == b);
   }
}
```

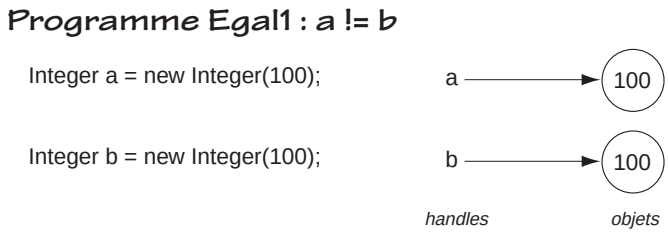
Ce programme affiche :

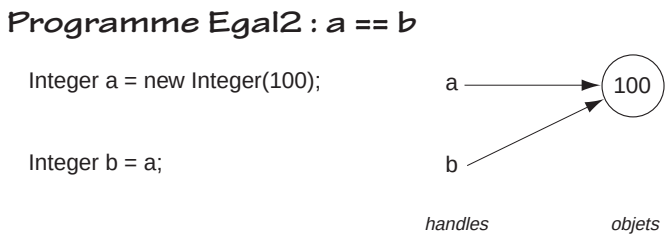
```
false
```

Bien que les deux objets de type **Integer** contiennent la même valeur, ils ne sont pas identiques, car il s'agit bien de deux objets différents. Pour que la valeur de la relation **b == c** soit **true**, il faudrait qu'il s'agisse d'un seul et même objet, comme dans l'exemple suivant :

```
class Egal2 {
   public static void main(String[] args) {
   Integer a = new Integer(100);
   Integer b = a;
   System.out.println(a == b);
   }
}
```

La figure ci-après montre clairement la différence :





Pour tester l'égalité de la valeur entière contenue dans les objets de type **Integer**, il faut utiliser la méthode **equals()**, de la façon suivante :

```
class Egal3 {
   public static void main(String[] args) {
   Integer a = new Integer(100);
   Integer b = new Integer(100);
   System.out.println(a.equals(b));
   }
}
```

Ce programme affiche :

```
true
```

Ici, nous avons utilisé la méthode **equals()** de l'objet **a**. Nous aurions tout aussi bien pu utiliser celle de l'objet **b** de la façon suivante :

```
System.out.println(b.equals(a));
```

Vous pouvez assimiler cela au fait qu'il est équivalent d'écrire **a == b** ou **b == a**. Ce n'est pourtant pas la même chose. Lorsque vous écrivez **a == b**, l'égalité est évaluée par un mécanisme construit dans l'interpréteur Java. C'est exactement le même mécanisme qui est utilisé pour évaluer **b == a**. Dans le cas de la méthode **equals()**, il s'agit d'une version différente dans chaque cas. A première vue, cela ne change rien, car la méthode **equals()** de l'objet **a** est identique à celle de l'objet **b**. Mais cela n'est pas toujours le cas, comme nous allons le voir plus loin.

En fait, la méthode **equals()** n'est pas propre aux objets de type **Integer**. Elle appartient à la classe **Object**. Tous les objets Java étant implicitement dérivés de cette classe, ils héritent tous de cette méthode, dont voici la définition :

```
public boolean equals(Object obj) {
   return (this == obj);
}
```

Cette méthode retourne **true** uniquement si son argument est un handle qui pointe vers le même objet que celui depuis lequel la méthode est appelée. Cela signifie qu'*a priori*, pour tous les objets de la classe **Object**, ces quatre expressions sont équivalentes :

```
a == b
b == a
a.equals(b)
b.equals(a)
```

La méthode **equals()** ne nous est donc guère utile. En fait, elle est conçue pour être redéfinie dans les classes dérivées qui nécessitent un autre type d'égalité. C'est le cas des classes telles que **Integer**, **Long**, **Float**, etc. Toutes ces classes redéfinissent la méthode **equals()**.

Si vous redéfinissez cette méthode dans une de vos classes, vous êtes supposé respecter les indications suivantes :

- La méthode doit être réflexive, c'est-à-dire que **x.equals(x)** doit retourner la valeur **true**.
- Elle doit être symétrique, c'est-à-dire que **x.equals(y)** doit retourner la même valeur que **y.equals(x)**.
- Elle doit être transitive, c'est-à-dire que si **x.equals(y)** et **y.equals(z)** retournent la valeur **true**, alors **x.equals(z)** doit retourner la valeur **true**.
- Elle doit être cohérente, c'est-à-dire que, quelles que soient les valeurs de **x** et **y**, **x.equals(y)** doit toujours retourner la même valeur.
- Quelle que soit **x**, **x.equals(null)** doit retourner la valeur **false**.

La classe **Integer** redéfinit la méthode **equals()** de la façon suivante :

```
public boolean equals(Object obj) {
   if ((obj != null) && (obj instanceof Integer)) {
      return value == ((Integer)obj).intValue();
   }
   return false;
}
```

ce qui respecte bien les critères définis. Cependant, vous pouvez parfaitement redéfinir cette méthode dans vos classes. En particulier, vous pouvez redéfinir cette méthode afin qu'elle renvoie **true** pour des objets de classes différentes, comme dans l'exemple suivant :

```
class MyEquals {
   public static void main(String[] args) {
   MyInteger a = new MyInteger(100);
   MyLong b = new MyLong(100);
   System.out.println(a.equals(b));
   System.out.println(b.equals(a));
   }
}

class MyInteger {
   int value;
   MyInteger(int i) {
      value = i;
   }

   public int value() {
      return value;
   }

   public long longValue() {
      return (long)value;
   }

   public boolean equals(Object obj) {
      System.out.print("Classe MyInt ");
```

```
      if (obj != null) {
         if (obj instanceof MyInteger)
            return value() == ((MyInteger)obj).longValue();
         else if (obj instanceof MyLong)
            return value() == ((MyLong)obj).value();
      }
      return false;
   }
}

class MyLong {
   long value;
   MyLong(long l) {
      value = l;
   }

   public long value() {
      return value;
   }

   public boolean equals(Object obj) {
      System.out.print("Classe MyLong ");
      if (obj != null) {
         if (obj instanceof MyInteger)
            return value == ((MyInteger)obj).longValue();
         else if (obj instanceof MyLong)
            return value == ((MyLong)obj).value();
      }
      return false;
   }
}
```

Dans ce programme, nous créons deux classes nommées **MyInteger** et **MyLong**. Il s'agit d'*enveloppeurs* pour des valeurs de type **int** et **long**. Chacune de ces classes comporte une méthode **equals()** qui respecte parfaitement les critères définis plus haut. Ces méthodes permettent de tester l'égalité des valeurs enveloppées, bien qu'elles soient de types différents. (Dans

la pratique, il existe des moyens beaucoup plus simples d'arriver au même résultat.) Malgré le respect des critères (qui ne concernent que la valeur de retour), les méthodes peuvent avoir des *effets de bord* différents (symbolisés ici par la présence des instructions **System.out.print("Classe MyLong ");** et **System.out.print("Classe MyInteger ");**. Ce programme affiche le résultat suivant :

```
Classe MyInt true
Classe MyLong true
```

Pour éviter ce type de problème, il est préférable, bien que cela ne figure pas parmi les recommandations officielles, de ne pas implémenter de méthodes **equals()** fonctionnant sur des objets de types différents. On pourra alors créer une classe intermédiaire parente des classes à tester et implémenter la méthode **equals()** dans la classe parente, par exemple de la façon suivante :

```
class MyEquals2 {
   public static void main(String[] args) {
   MyInteger a = new MyInteger(100);
   MyLong b = new MyLong(100);
   System.out.println(a.equals(b));
   System.out.println(b.equals(a));
   }
}
class MyNumber {
   public boolean equals(Object obj) {
      if (obj != null) {
         if (obj instanceof MyInteger) {
            if (this instanceof MyInteger)
               return ((MyInteger)this).value()
                                 == ((MyInteger)obj).value();
            if (this instanceof MyLong)
               return ((MyLong)this).value()
                                 == ((MyInteger)obj).longValue();
         }
         else if (obj instanceof MyLong) {
```

```
            if (this instanceof MyInteger)
               return ((MyInteger)this).longValue()
                              == ((MyLong)obj).value();
            if (this instanceof MyLong)
               return ((MyLong)this).value()
                              == ((MyLong)obj).value();
         }
      }
      return false;
   }
}
class MyInteger extends MyNumber {
   int value;
   MyInteger(int i) {
      value = i;
   }

   public int value() {
      return value;
   }

   public long longValue() {
      return (long)value;
   }
}

class MyLong extends MyNumber {
   long value;
   MyLong(long l) {
      value = l;
   }
   public long value() {
      return value;
   }
}
```

Dans ce programme, la méthode **equals()** est fournie par la classe parente **MyNumber**. Cela assure que ce sera bien toujours la même méthode qui sera appelée, quel que soit l'ordre des paramètres. Bien entendu, il est toujours possible de faire en sorte que les effets de bord soient différents dans les deux cas, mais les risques sont tout de même beaucoup plus limités. (Il faudrait quasiment le faire exprès !) De plus, le risque majeur avec la première approche est de modifier une des méthodes en oubliant l'existence de l'autre. Il est toujours préférable de grouper en un seul endroit le code correspondant à une même opération. Faire en sorte que les évaluations de **x.equals(y)** et **y.equals(x)** reposent sur deux méthodes différentes est la meilleure façon de s'attirer des ennuis futurs.

Note 1 : Nous sommes bien conscients que nous avons utilisé dans ces exemples des notions que nous n'avons pas encore présentées. Nous nous en excusons, mais il est vraiment difficile de faire autrement. Tout ce qui concerne les instructions conditionnelles **if {} else {}** sera traité dans la suite de ce chapitre, ainsi que l'opérateur **instanceof**. Les techniques de cast ont déjà été évoquées. Elles sont employées ici d'une façon systématique très courante en Java. Nous y reviendrons.

Note 2 : Pour ne pas ajouter aux inconvénients décrits dans la note 1, nous avons volontairement omis d'employer un certain nombre de techniques qui devraient l'être dans ce type d'exemple. Ne considérez donc pas ce programme comme une version définitive représentant la façon optimale de traiter le problème.

Les opérateurs logiques

Java dispose des opérateurs logiques qui produisent un résultat de type **boolean**, pouvant prendre les valeurs **true** ou **false**. Il s'agit des opérateurs suivants :

&&	Et	(deux opérandes)
||	Ou	(deux opérandes)
!	Non	(un seul opérande)

On représente couramment le fonctionnement de ces opérateurs sous la forme de *tables de vérité* :

&&	true	false
true	true	false
false	false	false

||	true	false
true	true	true
false	true	false

!	
true	false
false	true

Il faut noter une particularité dans la manière dont Java évalue les expressions contenant des opérateurs logiques : afin d'améliorer les performances, l'évaluation cesse dès que le résultat peut être déterminé. Ainsi, dans l'exemple suivant :

```
int x = 5;
int y = 10;
boolean z = (x > 10) && (y == 15);
```

Java commence par évaluer le premier terme de l'expression, **x > 10**. Sa valeur est **false**, puisque **x** vaut 5. Cette expression faisant partie d'une autre expression avec l'opérateur **&&**, un simple coup d'œil à la table de vérité de cet opérateur nous montre que l'expression globale sera fausse, quelle que soit la valeur de **y == 15**. Il est donc inutile d'évaluer la suite de l'expression. De la même façon, dans l'exemple suivant :

```
int x = 5;
int y = 10;
boolean z = (x < 10) || (y == 15);
```

il est inutile d'évaluer **y == 15** puisque l'évaluation de **x < 10** donne **true**, ce qui, d'après la table de vérité de l'opérateur ||, implique que l'expression globale vaudra **true** dans tous les cas.

Dans ces exemples, cela ne prête pas à conséquence. En revanche, dans l'exemple suivant :

```
int x = 5;
int y = 10;
boolean z = (x < 10) || (y == calcul(x));
```

la méthode **calcul** ne sera pas exécutée. Cela peut avoir ou non une importance, selon que cette méthode a ou non des effets de bord utiles. En effet, les méthodes Java peuvent servir de *fonctions* (qui retournent une valeur), de *procédures* (qui ne retournent pas de valeur mais exécutent un traitement), ou des deux à la fois.

Note : Une fonction pure n'existe pas. Toutes les fonctions modifient l'environnement, au moins de façon passagère. Idéalement, une fonction pure devrait laisser l'environnement tel qu'elle l'a trouvé. Ce n'est pratiquement jamais le cas en Java. En effet, si une fonction crée des objets, elle n'a aucun moyen de les détruire avant de retourner. Par ailleurs, les arguments passés aux fonctions sont le plus souvent des handles pointant vers des objets. Toute modification effectuée sur un de ses arguments objets par une méthode persiste après que la méthode a retourné. Toutes les méthodes Java ont donc des effets de bord, désirés ou non.

Dans l'exemple précédent, si vous comptez que la méthode sera appelée à chaque évaluation de l'expression, vous risquez d'avoir des surprises !

Le problème ne se pose pas seulement avec les méthodes, mais également avec certains opérateurs :

```
boolean z = (x < 10) || (y == x++);
System.out.println(x);
```

Dans cet exemple (on suppose que **x** et **y** ont été déclarées et initialisées ailleurs), Java affichera 5 si **x** est plus petit que 10 et 6 dans le cas contraire !

Vous devez faire très attention à ce point. Si vous voulez être certain que la totalité de l'expression sera évaluée systématiquement, vous pouvez utiliser une astuce consistant à employer les opérateurs d'arithmétique binaire, qui sont décrits dans la prochaine section.

Les opérateurs d'arithmétique binaire

Les opérateurs d'arithmétique binaire agissent au niveau des bits de données, sans tenir compte de ce qu'ils représentent. La présence de ces opérateurs en Java est curieuse, non pas parce qu'ils ne sont pas utiles ; ils sont même pratiquement indispensables pour certains traitements si l'on veut obtenir des performances correctes. Ce qui est étrange, c'est l'absence d'éléments fondamentalement complémentaires que sont les types de données non signés. Toutes les données sont représentées en mémoire par des séries de bits, généralement groupés par multiples de 8. L'élément fondamental est donc l'octet, qui comporte 8 bits. Java dispose bien d'un type de données correspondant, mais il est *signé*, c'est-à-dire qu'un de ses bits (le petit dernier au fond à gauche) représente le signe (positif ou négatif). Bien sûr, il est possible de ne pas en tenir compte lorsque l'on effectue des manipulations binaires, mais cela empêche d'effectuer facilement des conversions qui simplifient l'écriture. Nous examinerons tout d'abord les opérateurs disponibles avant de revenir sur ce problème.

Les opérateurs binaires disponibles en Java sont les suivants :

&	Et	(deux opérandes)
|	Ou	(deux opérandess)
^	Ou exclusif	(deux opérandes)
~	Non	(un opérande)
<<	Décalage à gauche	(deux opérandes)
>>	Décalage à droite avec extension du signe	(deux opérandes)
>>>	Décalage à droite sans extension du signe	(deux opérandes)

L'opérateur ~ inverse la valeur de chacun des bits de son opérande unique. Les opérateurs &, |, et ^ prennent deux opérandes et produisent un résultat, en fonction des tables suivantes :

&	1	0
1	1	0
0	0	0

|	1	0
1	1	1
0	1	0

^	1	0
1	0	1
0	1	0

~	
1	0
0	1

Ces tables donnent les résultats pour chaque bit des opérandes. Ces opérateurs peuvent être associés à l'opérateur d'affectation = pour produire les versions suivantes :

&=

|=

^=

<<=

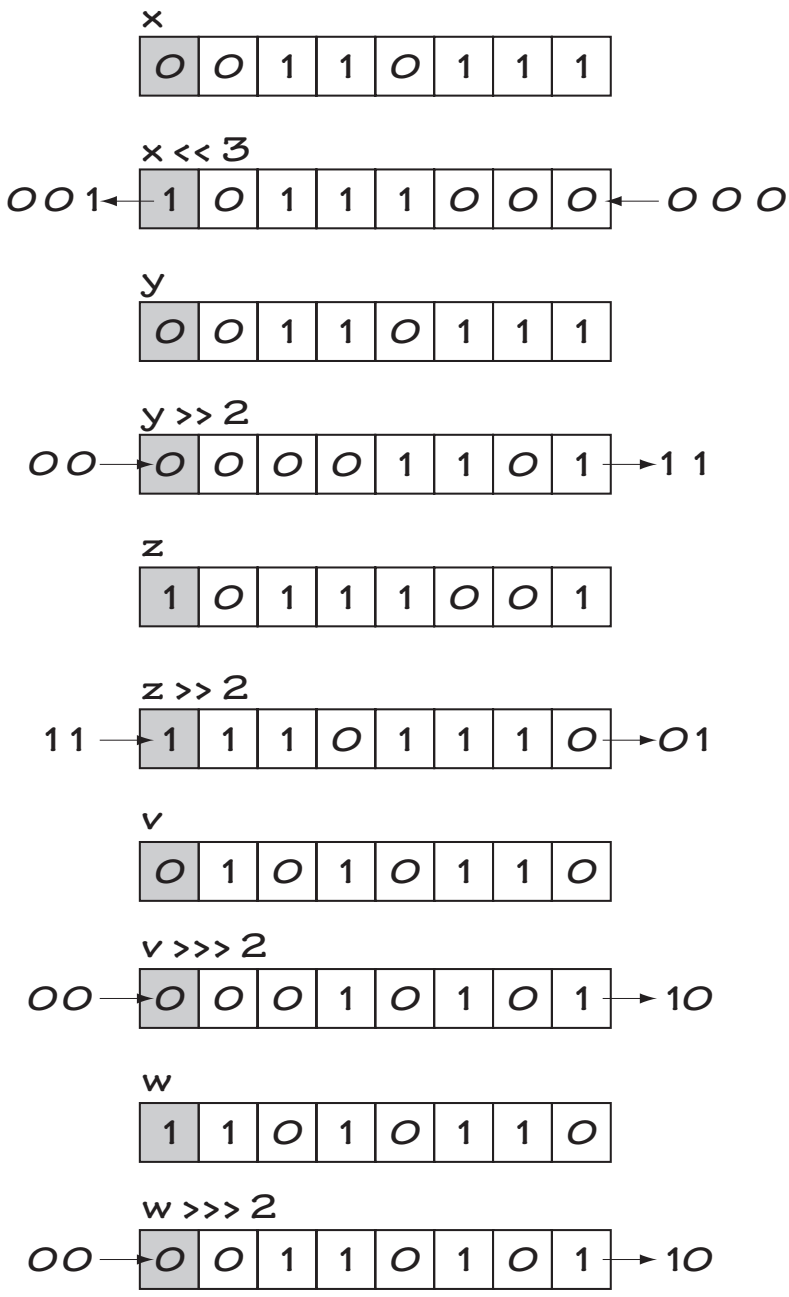
>>=

>>>=

avec lesquelles le résultat de l'opération est affecté à l'opérande de gauche. Dans ce cas, un sous-casting est automatiquement effectué, avec les risques de perte de données que nous avons déjà décrits pour les opérateurs arithmétiques.

Les opérateurs de décalage déplacent les bits du premier opérande vers la gauche ou vers la droite d'un nombre de rangs correspondant au second opérande, comme le montre la figure de la page suivante.

Les bits qui sortent à gauche ou à droite sont perdus. Les bits qui rentrent à droite, dans le cas d'un décalage à gauche (voir exemple *x*) valent toujours 0. On parle alors parfois d'*extension de 0.*


x
0 0 1 1 0 1 1 1
x << 3
0 0 1
1 0 1 1 1 0 0 0
0 0 0
y
0 0 1 1 0 1 1 1
y >> 2
0 0
0 0 0 0 1 1 0 1
1 1
z
1 0 1 1 1 0 0 1
z >> 2
1 1
1 1 1 0 1 1 1 0
0 1
v
0 1 0 1 0 1 1 0
v >>> 2
0 0
0 0 0 1 0 1 0 1
1 0
w
1 1 0 1 0 1 1 0
w >>> 2
0 0
0 0 1 1 0 1 0 1
1 0

Dans le cas d'un décalage à droite sans extension de signe, les bits qui entrent à gauche valent toujours 0 (exemples *v* et *w*). Dans le cas d'un décalage à droite avec extension de signe, les bits qui entrent à gauche ont la même valeur que le bit le plus à gauche (exemples *y* et *z*).

Comme nous l'avons dit précédemment, il est utile de disposer de types de données non signés pour l'arithmétique binaire. En effet, le décalage à gauche correspond à la multiplication par deux et le décalage à droite à la division par deux (dans la limite où le décalage est inférieur au nombre de bits, sinon le résultat vaut 0). On utilise fréquemment la notation hexadécimale pour représenter les valeurs binaires, car la conversion est très aisée (avec un peu d'habitude, elle se fait mentalement). Mais si l'on opère un décalage sur une valeur signée, il arrive fréquemment que le signe change à chaque décalage.

Il devient difficile de faire correspondre la représentation binaire et la représentation décimale ou hexadécimale. Par exemple, pour un décalage de 1 bit, la valeur :

01010101 (85 en décimal)

devient :

10101010 (-41 en décimal)

Alors qu'en représentation non signée, nous aurions :

01010101 (85 en décimal)

devient :

10101010 (170 en décimal)

Cependant, Java tourne la difficulté d'une autre façon. En fait, tous les décalages sont effectués sur 4 octets, c'est-à-dire que si vous décalez une va-

leur de type **byte** ou **short**, Java effectue d'abord un cast pour la convertir en **int**. Le résultat de l'opération est un **int**. (Dans la figure, tous les décalages sont représentés sur 8 bits pour des raisons d'espace disponible !) Un décalage sur un **long** produit un **long**. Un décalage sur un **char** produit un **char**.

Note : Vous pouvez effectuer des opérations binaires sur les types **byte**, **short**, **int**, **long**, et **char** (qui est le seul type numérique non signé de Java). Les opérateurs &, | et ^ peuvent également être appliqués aux valeurs de type **boolean** (qui sont des valeurs à 1 bit), mais pas l'opérateur ~ (ni évidemment, les décalages !).

Il résulte de ce que nous venons de dire que Java refusera de compiler le programme suivant :

```
class Decalage {
   public static void main(String[] args) {
   short a = 5;
   short b = a << 1;
}
```

car **a << 1** produit un **int**. Il faut donc un cast explicite pour affecter le résultat à un **short** :

```
class Decalage {
   public static void main(String[] args) {
   short a = 5;
   short b = (short)(a << 1);
}
```

Une source d'erreur fréquente est l'oubli des parenthèses autour de l'expression **a << 1**. En effet, si vous écrivez :

```
short b = (short)a << 1;
```

Java cast **a** en **short** (ce qu'il est déjà), puis en **int** pour effectuer l'opération.

Lorsque Java effectue un décalage sur un **int** (directement ou après un cast d'un **byte** ou d'un **short**), le décalage est limité à 31 bits, c'est-à-dire le nombre de bits correspondant à un **int**. De la même façon, un décalage effectué sur un **long** est limité à 63 bits, soit le nombre de bits d'un **long**. Dans les deux cas, il s'agit du nombre de bits en excluant le bit de signe.

Que se passe-t-il si vous tentez d'effectuer un décalage d'un nombre de bits supérieur à ces valeurs ? Le programme suivant permet de le savoir :

```
class Decalage2 {
   public static void main(String[] args) {
      int x = 150;
      System.out.println(x);
      System.out.println(x >>> 1);
      System.out.println(x >>> 33);
      System.out.println(x >>> 97);
      System.out.println(x >>> -31);
      System.out.println(x >>> -95);
   }
}
```

Ce programme produit le résultat suivant :

```
150
75
75
75
75
75
```

Nous constatons qu'un décalage d'un **int** avec un opérande égal à 33, 97, -95 ou -31 produit le même résultat qu'un décalage d'un bit. Pour comprendre pourquoi, il faut considérer la façon dont les nombres signés sont repré-

sentés. Java utilise la notation en *complément à 2*. Avec cette notation, un nombre négatif est représenté en inversant tous ses bits (ce qui produit le *complément à 1*) puis en ajoutant 1. Ce qui peut s'exprimer de la façon suivante :

$$-x = \sim x + 1$$

et peut être vérifié à l'aide du programme :

```
class Essai {
   public static void main(String[] args) {
   byte x = 66;
   System.out.println(x);
   System.out.println(~x + 1);
   }
}
```

qui affiche :

```
66
-66
```

Les valeurs décimales 1, 33, 97, -31 et -95 sont donc représentées respectivement par les valeurs binaires suivantes :

```
  1       0000 0000  0000 0000  0000 0000  0000 0001

 33       0000 0000  0000 0000  0000 0000  0010 0001

 97       0000 0000  0000 0000  0000 0000  0110 0001

-31       1111 1111  1111 1111  1111 1111  1110 0001

-95       1111 1111  1111 1111  1111 1111  1100 0001
```

Ces cinq valeurs ont en commun leurs cinq bits de *poids faible*, c'est-à-dire placés le plus à droite. Pour effectuer un décalage d'un **int**, Java ne tient

compte que de ces cinq bits. Utiliser une valeur supérieure produit un résultat tout à fait prévisible, mais souvent imprévu, ce qui peut parfaitement servir de définition au mot *bug* !

Lorsqu'il s'agit du décalage d'un **long**, Java prend en compte les six bits les plus à droite, ce qui limite le décalage à 63 bits. (Avec 6 bits, on peut exprimer de façon *non signée* des valeurs comprises entre 0 et 63.) Notez au passage que si Java ne fournit pas au programmeur des types non signés, ils sont toutefois utilisés par le langage, ce qui est d'autant plus frustrant.

Des conséquences du sur-casting automatique sur l'extension de zéro

Nous savons maintenant comment fonctionne réellement le décalage. Nous savons également que les valeurs de type **char**, **byte** ou **short** sont sur-castées en **int** avant que le décalage soit effectué. Le sur-casting n'introduit en principe aucune perte de données. Par conséquent, le même décalage appliqué à un **char**, un **byte**, un **short** ou un **int**, tous initialisés à la même valeur, devrait produire le même résultat. Nous pouvons le vérifier à l'aide du programme suivant :

```
class Probleme {
   public static void main(String[] args) {
      byte w = (byte)66;
      short x = (short)66;
      char y = (char)66;
      int z = 66;
      for (int i = 0; i < 3; i++) {
         System.out.print(
               (w >>> i) + "\t" +
               (x >>> i) + "\t" +
               (y >>> i) + "\t" +
               (z >>> i) + "\n");
      }
   }
}
```

Nous n'avons pas encore étudié l'instruction **for**, mais sachez que, dans ce programme, elle a pour effet d'exécuter le contenu du bloc suivant (l'instruction **System.out.print**) une fois pour chaque valeur de i comprise entre 0 (inclus) et 3 (exclu). L'instruction **System.out.print** a été coupée en plusieurs lignes pour des questions de mise en page. (Il est possible de la couper n'importe où sauf au milieu d'un mot ou entre les guillemets.) Les quatre variables subissent donc successivement un décalage de 0, 1 et 2 bits. Le résultat obtenu est le suivant :

```
66      66      66      66
33      33      33      33
16      16      16      16
```

Pour l'instant, tout va bien. Essayons la même chose avec des valeurs négatives, en remplaçant les lignes 3, 4, 5 et 6 du programme par :

```
byte w = (byte)-66;
char y = (char)-66;
int z = -66;
```

et en modifiant l'instruction d'affichage de la façon suivante :

```
System.out.print(
             (w >>> i) + "\t" +
             (x >>> i) + "\t" +
             (z >>> i) + "\n");
```

(Le type **char** étant non signé, lui attribuer une valeur négative n'a pas de sens, bien que cela soit parfaitement possible.) Nous obtenons maintenant le résultat suivant :

```
-66        -66        -66
2147483615      2147483615        2147483615
1073741807      1073741807        1073741807
```

Tout semble continuer d'aller parfaitement. Vraiment ? Pourtant, on peut déceler ici la trace d'un problème. Essayez de le trouver vous-même.

Cela saute aux yeux, non ? Le problème est lié au sur-casting. L'extension de 0 se produit dans le bit de poids fort, c'est-à-dire le bit le plus à gauche. Dans un **int**, il s'agit du 32e bit en commençant par la droite. Cependant, dans un **short**, le bit de signe est le 16e, et dans un **byte**, il s'agit du 8e. Java devrait en tenir compte lors des sous-castings. En fait, Java se contente de tronquer les bits excédentaires. En raison de l'utilisation de la notation en complément à deux, cela fonctionne parfaitement, dans la plupart des cas. En revanche, si le bit de signe est modifié avant le sous-casting, cela peut entraîner un problème. C'est le cas de l'extension de 0, qui, appliquée à un nombre négatif, le transforme en nombre positif. Exemple :

11111111 11111111 11111111 10111110 = - 66

après décalage avec extension de 0 :

01111111 11111111 11111111 11011111 = 2 147 483 615

Le résultat est correct dans le cas d'un **int**, mais pas dans celui d'un **short** ou d'un **byte** sur-castés. En effet, le décalage avec extension de 0 d'un **byte** de la même valeur est représenté ci-dessous :

10111110 = - 66

après décalage :

01011111 = 95

Dans le cas d'un **byte** sur-casté, le processus complet est :

10111110 = - 66

après sur-casting :

11111111 11111111 11111111 10111110 = - 66

après décalage :

01111111 11111111 11111111 11011111 = 2 147 483 615

après sous-casting :

11011111 = -33

L'extension de 0 ne s'est pas produite au bon endroit. Nous pouvons mettre cela en évidence dans la pratique en modifiant le programme précédent de la façon suivante :

```
class Probleme2 {
   public static void main(String[] args) {
      byte w = (byte)-66;
      short x = (short)-66;
      int z = -66;
      System.out.print(w + "\t" + x + "\t" + z + "\n");
      for (int i = 1; i < 3; i++) {
         w >>>= 1;
         x >>>= 1;
         z >>>= 1;
         System.out.print(w + "\t" + x + "\t" + z + "\n");
      }
   }
}
```

Ce programme affiche le résultat suivant :

```
-66        -66        -66
-33        -33        2147483815
-17        -17        1073741807
```

Cette particularité fait dire à certains que le décalage à droite avec extension de 0 comporte un bug. Ce n'est pas le cas. On est plutôt ici en présence d'une idiosyncrasie : l'incompatibilité du décalage tel qu'il est implémenté avec le sous-casting. Le bug n'est ni dans le décalage, ni dans le sous-casting. Il est induit par l'utilisation du sur-casting automatique. Moralité, pour l'arithmétique binaire, il vaut mieux utiliser le type **int**.

Utilisation des opérateurs d'arithmétique binaire avec des valeurs logiques

Nous avons vu que les opérateurs &, | et ^ peuvent être employés avec des valeurs logiques, qui sont des valeurs sur 1 bit. L'intérêt de cette possibilité est que, s'agissant d'opérateurs arithmétiques, ils sont toujours évalués. Comme nous l'avons dit précédemment, dans l'exemple suivant :

```
int x = 5;
int y = 10;
boolean z = (x < 10) || (y == calcul(x));
```

la méthode calcul(x) n'est jamais exécutée en raison du court-circuit provoqué par Java : le premier terme de l'expression étant vrai, Java peut déterminer que l'expression entière est vraie sans évaluer le second terme. Si vous voulez forcer Java à évaluer les deux termes dans tous les cas, il vous suffit de remplacer l'opérateur logique || par l'opérateur d'arithmétique binaire | :

```
int x = 5;
int y = 10;
boolean z = (x < 10) | (y == calcul(x));
```

Dans ce cas, les deux termes seront évalués quelle que soit la valeur du premier. De même, dans l'exemple suivant :

```
boolean z = (x < 10) $$ (y == x++);
```

la valeur de **x** sera augmentée de 1 seulement si **x** est plus petit que 10, alors que dans l'expression ci-après :

```
boolean z = (x < 10) $ (y == x++);
```

elle le sera quoi qu'il arrive.

Utilisation de masques binaires

L'arithmétique binaire est souvent mise en œuvre à travers la création et l'utilisation de masques. Supposons, par exemple, que vous souhaitiez écrire un programme de test de connaissances comportant vingt questions. Chaque fois que l'utilisateur répond à une question, le programme détermine si la réponse est juste ou fausse. Vous souhaitez stocker dans le programme une valeur indiquant si la réponse est juste ou fausse. Vous pouvez procéder de plusieurs façons. L'une consiste à utiliser une série de vingt valeurs de type **boolean** qui vaudront, par exemple, **true** si la réponse est juste et **false** dans le cas contraire. Ces valeurs pourront être placées, par exemple, dans un tableau. (Les tableaux seront étudiés dans un prochain chapitre.)

Cette façon de procéder n'est pas très performante, car la manipulation de tableau est une technique plutôt lourde. Une autre façon, plus efficace, consiste à employer une valeur de 20 bits et de mettre chaque bit à 1 ou à 0 selon que la réponse est juste ou fausse. Cette technique peut être mise en œuvre en employant des opérateurs arithmétiques. En effet, en partant d'une valeur de 0, mettre le bit de rang *n* à 1 équivaut à ajouter 2^n à la valeur. (Pour que cela fonctionne, il faut, comme c'est l'usage, compter les rangs de 0 à 19 et non de 1 à 20.)

Cette technique présente cependant un inconvénient certain : elle fonctionne parfaitement si le bit considéré vaut 0, mais produit un débordement sur le bit suivant s'il vaut déjà 1, ce qui peut être le cas si l'utilisateur a la possibilité de revenir en arrière pour modifier ses réponses. La prise en compte de cette possibilité avec les moyens de l'arithmétique décimale devient complexe.

Les opérateurs d'arithmétique binaire permettent de réaliser cela très facilement. En effet, pour mettre à 1 la valeur d'un bit quelle que soit sa valeur d'origine, il suffit d'effectuer l'opération OU avec une valeur dont le bit de rang considéré vaille 1 et tous les autres 0. Par exemple, pour mettre à 1 le bit de rang *n* (en commençant par la droite) d'une valeur quelconque *x* sans changer les autres bits, on utilisera la formule suivante :

$$x = x \mid 2^n$$

Pour faire l'opération inverse, c'est-à-dire mettre le bit de rang *n* à 0, ont utilisera la formule :

$$x = x \ \& \ (\sim 2^n)$$

en remarquant que le masque à utiliser est l'inverse binaire du précédent.

L'opérateur à trois opérandes ?:

Java dispose d'un opérateur un peu particulier permettant de tester une condition et d'utiliser une expression différente pour fournir le résultat, selon que cette condition est vraie ou fausse. La syntaxe de cet opérateur est la suivante :

condition ? *expression_si_vrai* : *expression_si_faux*

Voici un exemple de son utilisation :

```
x = y < 5 ? 4 * y : 2 * y;
```

Dans cet exemple, **x** prend une valeur égale à **4 * Y** si **y** est plus petit que 5 et à **2 * y** dans le cas contraire. Le même résultat peut être obtenu avec l'instruction conditionnelle **if...else** (que nous étudierons bientôt) de la façon suivante :

```
if (y < 5)
   x = 4 * y;
else
   x = 2 * y;
```

ce qui est beaucoup plus lisible.

Notez que seule l'expression utilisée pour le résultat est évaluée. Ainsi, dans l'exemple suivant :

```
int x = y < 5 ? y++ : y--;
```

Dans tous les cas, **x** prend la valeur de **y**. En revanche, si **y** est plus petit que 5, sa valeur est augmentée de 1. Dans le cas contraire, sa valeur est diminuée de 1. (**x** prend la valeur de **y** *avant* l'augmentation ou la diminution en raison de l'utilisation de la version postfixée des opérateurs ++ et --.) Exemples :

```
int y = 5;
int x = y < 5 ? y++ : y--;
System.out.println(x);
System.out.println(y);
```

Dans ce cas, le programme affiche :

```
5
4
```

En revanche, les lignes :

```
int y = 4;
int x = y < 5 ? y++ : y--;
System.out.println(x);
System.out.println(y);
```

affichent :

```
4
5
```

Les opérateurs de casting

Les opérateurs de casting existent pour toutes les primitives comme pour tous les objets. Cependant, tous les casts ne sont pas autorisés. Nous reviendrons sur le cas des objets. En ce qui concerne les primitives, les casts sont autorisés

entre tous les types sauf les **boolean**. Bien entendu, certains casts peuvent conduire à des pertes de données, mais cela reste sous la responsabilité du programmeur. Java ne se préoccupe que des casts implicites. Les opérateurs de casts sont simplement les noms des types de destination placés entre parenthèses avant la valeur à caster. Exemple :

```
int x = 5;
short y = (short)x;
```

Un cast d'un type d'entier vers un type d'entier de taille inférieure est effectué par simple troncature de la partie de poids fort (partie gauche). S'agissant de types signés utilisant la représentation en complément à 2, le résultat est cohérent dans la mesure où :

- La valeur sous-castée provient d'une valeur sur-castée à partir du même type.
- Aucune manipulation n'a été faite sur les bits se trouvant à gauche du bit de signe d'origine entre le moment où la valeur a été sur-castée et le moment ou elle est sous-castée.

Par exemple, si un **byte** (7 bits de données plus un bit de signe) est sur-casté en **short** (15 bits de données et un bit de signe), puis sous-casté en **byte**, le résultat sera cohérent si les bits 8 à 15 n'ont pas été modifiés. Dans le cas contraire, le résultat sera incohérent (et non pas seulement tronqué). Exemple :

```
byte x = (byte)127;
short y = (short)x;
y++;
System.out.println((byte)y);
```

Ce programme affiche le résultat suivant :

```
-128
```

Note 1 : Nous avons utilisé ici l'opérateur ++ pour incrémenter **y** car cet opérateur n'effectue pas un cast automatique en **int**.

Note 2 : Les casts s'appliquent aux valeurs et non aux variables. Lorsqu'une variable est déclarée avec un type, il n'est plus possible d'en changer car une variable ne peut être redéfinie dans sa zone de visibilité. En revanche, il est possible de masquer une variable avec une autre d'un autre type car les deux variables coexistent dans ce cas.

Les opérateurs de casting et les valeurs littérales

Les opérateurs de cast peuvent être employés avec des valeurs littérales. Pour une explication détaillée, reportez-vous au Chapitre 4. Rappelons seulement ici que Java effectue automatiquement un sous-casting de **int** en **short**, **char** ou **byte** lorsque cela est nécessaire, à condition que l'opération ne présente pas de risques de perte de données. Dans le cas contraire, un sous-casting explicite est nécessaire. En d'autres termes, l'expression :

```
byte i = 127;
```

est correcte bien que 127 soit un **int**. En revanche, l'expression suivante produit une erreur de compilation :

```
byte i = 128;
```

car la valeur maximale qui peut être représentée par un **byte** est 127.

L'opérateur *new*

L'opérateur **new** permet d'instancier une classe, c'est-à-dire de créer un objet instance de cette classe. Nous l'avons étudié dans le précédent chapitre. Nous ne le mentionnons ici que parce qu'il occupe une place dans l'ordre des priorités des opérateurs.

L'opérateur *instanceof*

L'opérateur **instanceof** produit une valeur de type **boolean**. Il prend deux opérandes, dont l'un (à gauche) est un handle et l'autre (à droite) une classe, renvoie **true** si l'identificateur pointe vers un objet de la classe indiquée, et **false** dans le cas contraire :

```
int x = 5;
String y = "y";
boolean a = x instanceof String;
boolean b = y instanceof String;
```

Dans cet exemple, **a** vaut **false** et **b** vaut **true**.

Note : Il existe également une version dynamique de l'opérateur **instanceof**, sous la forme d'une méthode de la classe **Class**. Nous reviendrons sur ce sujet dans un prochain chapitre.

L'opérateur **instanceof** ne permet pas de tester le type d'une primitive. Il ne peut même pas déterminer si un identificateur correspond à une primitive ou à un objet, car son utilisation avec une primitive produit une erreur de compilation.

Déterminer le type d'une primitive est cependant assez facile en utilisant la redéfinition de méthode, comme nous le verrons dans un exemple à la fin du chapitre suivant.

Priorité des opérateurs

Dans une expression, les opérateurs sont évalués dans un certain ordre, selon leur priorité. Les opérateurs de priorité haute sont évalués avant ceux de priorité basse. Lorsque plusieurs opérateurs d'un même niveau de priorité sont présents, ils sont évalués dans l'ordre propre à ce niveau de priorité (de droite à gauche ou de gauche à droite). Les parenthèses, les appels de

méthodes ou les crochets ou le connecteur "." peuvent modifier l'ordre d'évaluation.

L'ordre d'évaluation des opérateurs Java est le suivant :

Priorité la plus haute	
Opérateurs	***Ordre d'évaluation***
appel de méthode ( ) [ ] .	de gauche à droite
! ~ ++ -- + (un seul opérande) - (un seul opérande) (cast) new	de droite à gauche
* / %	de gauche à droite
+ (deux opérandes) - (deux opérandes)	de gauche à droite
<< >> >>>	de gauche à droite
< <= > >= instanceof	de gauche à droite
= !=	de gauche à droite
&	de gauche à droite
^	de gauche à droite
|	de gauche à droite
&&	de gauche à droite
||	de gauche à droite
?:	de gauche à droite
= += -= *= /= %= &= |= ^= <<= >>= >>>=	de droite à gauche
Priorité la plus basse	

Les parenthèses peuvent être utilisées pour modifier l'ordre d'évaluation de certains opérateurs. Cependant il est des cas où cela est impossible. Par exemple :

```
x *= y += 20;
```

est équivalent à :

```
x *= (y += 20);
```

mais il est impossible d'écrire :

```
(x *= y) += 20;
```

Pour obtenir le résultat souhaité (multiplier **x** par **y**, ajouter **3** et placer le résultat dans **x**), il faut écrire :

```
x *= y; x += 20;
```

ou encore, dans certains cas :

```
x *= y + 20 / x;
```

Pourquoi dans certains cas ? Si cela ne vous paraît pas évident, essayez l'exemple suivant :

```
class MonCalcul {
   public static void main(String[] args) {
   double x = 5;
   double y = 10;
   x *= y + 20 / x;
   System.out.println(x);
   }
}
```

Cela semble fonctionner. Essayez maintenant :

```
class MonCalcul {
   public static void main(String[] args) {
   double x = 3;
   double y = 10;
   x *= y + 20 / x;
   System.out.println(x);
   }
}
```

Cela semble fonctionner aussi. Essayez ensuite :

```
class MonCalcul {
   public static void main(String[] args) {
   int x = 3;
   int y = 10;
   x *= y + 20 / x;
   System.out.println(x);
   }
}
```

Tout va bien. Est-ce la peine de faire un essai supplémentaire ? Peut-être, comme le montre l'exemple suivant :

```
class MonCalcul {
   public static void main(String[] args) {
   int x = 5;
   int y = 10;
   x *= y + 20 / x;
   System.out.println(x);
   }
}
```

Cette fois, cela ne fonctionne plus du tout. Que s'est-il passé ? L'expression :

```
x *= y + 20 / x;
```

est équivalente, en tenant compte de l'ordre de priorité des opérateurs, de :

```
x = x * (y + (3 / x));
```

ce qui, pour nous, semble équivalent à :

```
x = (x * y) + (x * (20 / x))
```

soit :

```
x = (x * y) + 20
```

ce qui est bien le résultat souhaité. Eh bien, il n'en est rien ! En effet, Java ne simplifie pas ce type d'expression mais la calcule bêtement. Au passage, **20 / x** produit éventuellement un résultat tronqué, si **x** n'est pas un diviseur de 20, en raison du cast du résultat en **int**.

Si vous pensez que l'utilisation de primitives de type **float** ou **double** résout le problème, sachez qu'il n'en est rien. Pour le prouver, exécutez le programme suivant :

```
class Precision {
   public static void main(String[] args) {
   double x = 9.00000000000001;
   double y = 10;
   double z = 8;
   boolean a;
   a = ((x * y + z) == (x *= (y+z/x)));
   System.out.println(a);
   }
}
```

Ce programme affiche :

```
false
```

alors que, d'un point de vue mathématique, les deux expressions sont égales. En fait, le résultat de la première est 98,00000000000011 alors que celui de la seconde est 98,0000000000001. Bien sûr, l'erreur numérique est négligeable dans pratiquement tous les cas. En revanche, elle peut prendre une importance considérable si vous effectuez, comme ici, un test d'égalité.

Résumé

Dans ce chapitre, nous avons étudié en détail les opérateurs de Java. Nous avons en particulier mis en lumière quelques pièges liés au cast automatique effectué par Java lors des opérations, ainsi que quelques problèmes concernant les conséquences du manque de précision des calculs. Le sujet était un peu ardu. Aussi, il n'y aura pas d'exercice pour cette fois. Profitez-en pour vous reposer avant d'aborder le chapitre suivant, qui sera consacré à l'étude des structures de contrôle.

Chapitre 7

Les structures de contrôle

Dans ce chapitre, nous allons étudier les différentes structures qui permettent de contrôler le déroulement des programmes. Contrairement à d'autres langages, ces structures ne permettent pas de contrôler intégralement l'exécution. En effet, Java est basé sur un modèle contrôlé par des événements. La particularité de ce type de langage peut être décrite simplement par un exemple. Supposons que le programme doive réagir à la frappe d'une certaine touche. Avec les langages d'anciennes générations, le programmeur doit réaliser une boucle dans laquelle le programme teste le clavier pour savoir si une touche a été frappée. Tant qu'aucune touche n'est frappée, la boucle s'exécute indéfiniment. Au contraire, avec un langage basé sur la gestion d'événements, le programme n'a pas à se préoccuper du clavier. Un gestionnaire d'événements préviendra un objet de l'application

que l'événement "frappe d'une touche" s'est produit. Le programme pourra alors lire le clavier pour savoir si la touche frappée est celle à laquelle (ou une de celles auxquelles) il doit réagir. Dans un modèle différent, le message envoyé à l'objet pourra contenir également l'indication de la touche frappée, ce qui permettra à l'objet destinataire du message de savoir immédiatement s'il doit réagir ou non.

Java est un langage basé sur la gestion des événements. Cela dit, tout ne peut pas se ramener à la gestion événements et à l'envoi de messages entre objets. Des traitements sont effectués à l'intérieur des objets. Pour ces traitements, nous avons besoin de structures de contrôle identiques à celles que l'on trouve dans les langages traditionnels.

La séquence

La structure la plus simple est la séquence. Une séquence est simplement composée d'une ou de plusieurs instructions qui se suivent et sont exécutées les unes après les autres, dans l'ordre dans lequel elles figurent dans le programme. Cela peut paraître trivial, mais il s'agit de la structure la plus fréquemment rencontrée. En Java, grâce à l'opérateur à trois opérandes **?:**, il est possible d'effectuer des traitements relativement complexes uniquement à l'aide de séquences. Cependant, il s'agit là d'un exercice de style. Cette pratique n'est pas à conseiller car ce type de programme devient vite totalement illisible.

En fait, on peut traiter pratiquement n'importe quel problème à l'aide de l'opérateur **?:**, de séquences et d'appels de méthodes, à condition que la longueur des séquences ne soit pas limitée. Cela dit, ça ne nous avance pas beaucoup !

Le branchement par appel de méthode

Une façon de détourner le flux d’exécution d'un programme (opération parfois appelée *branchement*) consiste à appeler une méthode. Nous avons vu

dans les chapitres précédents qu'une méthode peut être utilisée comme une fonction, c'est-à-dire pour la valeur qu'elle retourne, ou comme une procédure, pour la modification qu'elle apporte à son environnement, ou encore pour les deux à la fois. Cette caractéristique est souvent mise à profit pour détourner l'exécution d'une séquence. Dans l'exemple suivant :

```
1   class Control1 {
2     public static void main(String[] args) {
3       int j = 10;
4       j = printDec(j);
5       j = printDec(j);
6       j = printDec(j);
7       j = printDec(j);
8       j = printDec(j);
9       j = printDec(j);
10    }
11
12    static int printDec(int i) {
13      System.out.println(i--);
14      return i;
15    }
16 }
```

le flux d'exécution se déroule de la façon suivante :

```
ligne 3
ligne 4
ligne 13
ligne 14
ligne 5
ligne 13
ligne 14
ligne 6
ligne 13
ligne 14
ligne 7
ligne 13
```

```
ligne 14
ligne 8
ligne 13
ligne 14
ligne 9
ligne 13
ligne 14
```

Une méthode utilisée comme procédure permet donc de contrôler le cours de l'exécution du programme. Par exemple, à l'aide de l'opérateur **?:**, il est possible de modifier ce programme afin qu'il décompte la valeur de **j** jusqu'à 0 :

```
class Control2 {
   public static void main(String[] args) {
      int j = 10;
      j = (j == 0)?0:printDec(j);
      j = (j == 0)?0:printDec(j);
      j = (j == 0)?0:printDec(j);
      j = (j == 0)?0:printDec(j);
      j = (j == 0)?0:printDec(j);
      j = (j == 0)?0:printDec(j);
      j = (j == 0)?0:printDec(j);
      j = (j == 0)?0:printDec(j);
      j = (j == 0)?0:printDec(j);
   }

   static int printDec(int i) {
      System.out.println(i--);
      return i;
   }
}
```

Nous avons volontairement recopié la ligne contenant l'opérateur **?:** plus de fois que nécessaire pour montrer que le programme fonctionne correctement. Vous objecterez peut-être qu'il ne fonctionne que pour des valeurs de **j** inférieures à 9. Qu'à cela ne tienne. Il suffit de multiplier à volonté la ligne

en question pour que le programme fonctionne pour n'importe quelle valeur. (Aucun langage ne pouvant traiter des valeurs infinies, il est possible de traiter tous les cas.) Ici, il suffit de recopier la ligne 2 147 483 647 fois (la plus grande valeur positive pouvant être représentée par un **int**) pour pouvoir traiter toutes les valeurs de **j**. Bien sûr, ce n'est pas très efficace, mais c'est possible !

L'instruction de branchement *return*

Le retour d'une méthode se fait à l'instruction **return** si elle est présente, ou à la fin du bloc dans le cas contraire. Cependant, l'instruction **return** peut être utilisée n'importe où dans le corps de la méthode. Celle-ci peut d'ailleurs contenir plusieurs instructions **return** qui seront exécutées en fonction de la réalisation de différentes conditions testées au moyen d'instructions telles que **if**, que nous étudierons dans la section suivante.

L'instruction **return** doit toujours être suivie d'une valeur correspondant au type de la méthode, sauf si le type est **void**. Cette valeur est appelée *valeur de retour* de la méthode.

L'instruction conditionnelle *if*

Un programme ne peut rien faire s'il n'est pas capable de prendre des décisions, c'est-à-dire d'effectuer ou non un traitement en fonction de l'évaluation d'une expression logique. En plus de l'opérateur **?:**, Java dispose, comme la plupart des langages de programmation, de l'instruction **if**.

L'instruction **if** prend pour paramètre une expression de type **boolean** placée entre parenthèses. Cette expression est évaluée et produit un résultat valant **true** ou **false**. Si le résultat est **true**, l'instruction ou le bloc suivant est exécuté. Si le résultat est **false**, l'instruction ou le bloc suivant est ignoré. La syntaxe de l'instruction **if** peut donc être décrite de la façon suivante :

```
if (expression) instruction;
```

ou :

```
if (expression) {
   instruction1;
   instruction2;
   .
   .
   .
   instructinon;
}
```

L'expression booléenne doit toujours être placée entre parenthèses. Si elle est suivie d'une seule instruction, celle-ci peut être placée sur la même ligne ou sur la ligne suivante. (En Java, les sauts de ligne ont presque toujours valeur de simples séparateurs.) Cependant, si vous préférez la version sur deux lignes :

```
if (expression)
   instruction;
```

il faut faire attention de ne pas mettre un point-virgule après l'expression.

Note : Lorsqu'il y a une seule instruction, il est parfaitement possible d'utiliser tout de même un bloc, ce qui est souvent plus lisible. Ainsi, les deux versions suivantes sont équivalentes à la précédente :

```
if (expression) {
   instruction;
}

if (expression) {instruction;}
```

Attention : N'oubliez pas qu'en Java, contrairement à l'usage avec d'autres langages, le point-virgule est obligatoire même en fin de bloc (avant le caractère **}**).

Quoi qu'il arrive, l'expression logique est toujours évaluée en respectant les règles de Java. Ainsi, dans l'exemple suivant :

```
int i = 0;
if (i++ > 0) {
   i += 2;
}
```

la variable **i** est initialisée à 0. Lorsque l'expression logique de la deuxième ligne est évaluée, **i** vaut encore 0. L'expression vaut donc **false** et l'instruction **i += 2** n'est donc pas exécutée. En revanche, la post-incrémentation de i (**i++**) est effectuée lors de l'évaluation de l'expression logique. A la fin du programme, **i** vaut donc 1.

Attention : Une source d'erreur très fréquente consiste à utiliser l'opérateur d'affectation dans l'expression logique :

```
if (i = 0) {
   i += 2;
}
```

La forme correcte est :

```
if (i == 0) {
   i += 2;
}
```

Heureusement, et contrairement à ce qui se passe avec d'autres langages, le compilateur Java signale immédiatement cette erreur.

L'instruction conditionnelle *else*

Comme nous l'avons dit précédemment, pratiquement tous les problèmes de programmation peuvent être traités en utilisant une instruction de bran-

chement (par exemple l'appel de méthode) et une instruction conditionnelle. Les premiers langages de programmation ne comportaient d'ailleurs que l'instruction **if** et une instruction de branchement beaucoup plus sommaire que l'appel de méthode. Cependant, il est utile de disposer d'instructions plus sophistiquées de façon à rendre les programmes plus lisibles.

Une des structures que l'on rencontre le plus souvent en programmation est du type :

```
si(expression logique)
   bloc d'instructions
sinon
   autre bloc d'instructions
```

Cette structure peut être réalisée à l'aide de deux instructions **if** utilisant des expressions logiques contraires, par exemple :

```
if (i == 0) {
   instruction1;
}
if (i != 0) {
   instruction2;
}
```

L'instruction **else** permet de simplifier cette écriture sous la forme :

```
if (i == 0) {
   instruction1;
}
else {
   instruction2;
}
```

Ces deux écritures ne sont toutefois pas du tout équivalentes. En effet, l'instruction **else** n'évalue pas l'expression contraire de celle de l'instruction **if**. Elle utilise simplement le résultat de l'évaluation de l'expression de l'ins-

truction **if** pour exécuter son bloc d'instruction si ce résultat est faux. La différence apparaîtra plus clairement dans l'exemple suivant.

Avec **if else** :

```
int i = 6;
if (i < 10) {
   i += 5;
}
else {
   i += 10;
}
```

Avec deux instructions **if** :

```
int i = 4;
if (i < 10) {
   i += 5;
}
if (i >= 10) {
   i += 10;
}
```

Dans le premier cas, l'expression **i < 10** est évaluée. Elle vaut **true**. Le premier bloc est donc exécuté et le bloc suivant l'instruction **else** est ignoré. A la fin du programme, **i** vaut 11. Dans le second cas, l'expression est testée de la même façon. Le bloc suivant l'instruction **if** est donc exécuté. Le problème vient de ce que la variable testée est modifiée avant la deuxième instruction **if**. Lorsque celle-ci est exécutée, la valeur de l'expression logique **i >= 10** est de nouveau **true**. Le second bloc est donc également exécuté. A la fin du programme, **i** vaut donc 21.

Les instructions conditionnelles et les opérateurs ++ et --

Vous pouvez parfaitement utiliser les opérateurs d'auto-incrémentation et d'auto-décrémentation dans les expressions logiques, bien que cela soit une source de confusion. Ainsi, dans l'exemple suivant :

```
int i = 4;
if (i++ > 4) {
   i += 2;
}
else {
   i -= 2;
}
```

le bloc **else** est exécuté alors que dans celui-ci :

```
int i = 4;
if (++i > 4) {
   i += 2;
}
else {
   i -= 2;
}
```

c'est le bloc **if** qui l'est.

Les instructions conditionnelles imbriquées

Dans certains langages, l'imbrication d'instructions **if else** conduit à des structures de hiérarchie complexes telles que :

```
if (expression1) {
   bloc1;
}
else {
   if (expression2) {
      bloc2;
   }
   else {
      if (expression3) {
         bloc3;
      }
```

```
      else {
         if (expression4) {
            bloc 4;
         }
         else {
            bloc 5;
         }
      }
   }
}
```

Java permet d'écrire ce type de structure plus simplement sous la forme :

```
if (expression1) {
   bloc1;
}
else if (expression2) {
   bloc2;
}
else if (expression3) {
   bloc3;
}
else if (expression4) {
   bloc 4;
}
else {
   bloc5;
}
```

De cette façon, tous les blocs se trouvent sur le même niveau de structure, ce qui est plus lisible et évite d'avoir à compter les parenthèses fermantes.

Attention : En fait, il ne s'agit pas d'une façon différente d'utiliser les instructions **else** et **if**, mais d'une instruction particulière, qui s'écrit en deux mots en Java, alors qu'il serait plus logique de lui donner un autre nom, comme dans de nombreux autres langages (par exemple **elseif**).

De toute façon, ces deux structures ne sont pas du tout équivalentes comme le montre l'exemple suivant, impossible à exprimer avec la structure **else if** :

```
if (i == 0) {
   System.out.println("0");
}
else {
   if (i == 1) {
      System.out.println("1");
   }
   else {
      if (i == 2) {
         System.out.println("2");
      }
      else {
         if (i == 3) {
            System.out.println("3");
         }
         else {
            System.out.println("plus grand que 3");
         }
      }
      System.out.println("Il suffit de 2.");
   }
}
```

Dans cet exemple, un bloc différent est exécuté pour chaque valeur de **i** comprise entre 1 et 4, et un bloc différent pour les valeurs supérieures à 4. De plus, un bloc spécial doit être exécuté pour les valeurs supérieures à 2. Seule la structure ci-dessus permet de réaliser cela sans dupliquer le bloc, ici représenté par l'instruction :

```
System.out.println("Il suffit de 2.");
```

La difficulté consiste à ne pas se tromper en plaçant ce bloc dans la hiérarchie. Pour réaliser la même chose avec la structure **else if**, il faut dupliquer le bloc de la façon suivante :

```
if (i == 0) {
   System.out.println("0");
}
else if (i == 1) {
   System.out.println("1");
}
else if (i == 2) {
   System.out.println("2");
   System.out.println("Il suffit de 2.");
}
else if (i == 3) {
   System.out.println("3");
   System.out.println("Il suffit de 2.");
}
else {
   System.out.println("plus grand que 3");
   System.out.println("Il suffit de 2.");
}
```

ce qui est contraire à un des principes de la programmation efficace qui veut que le code soit dupliqué le moins possible. (La raison en est évidente : si vous voulez modifier les messages de votre programme, par exemple pour les traduire, vous avez trois modifications à faire au lieu d'une.)

Une autre façon de procéder serait la suivante :

```
if (i == 0) {
   System.out.println("0");
}
else if (i == 1) {
   System.out.println("1");
}
else if (i == 2) {
   System.out.println("2");
}
else if (i == 3) {
   System.out.println("3");
}
```

```
else {
   System.out.println("plus grand que 3");
}
if (i >= 2) {
   System.out.println("Il suffit de 2.");
}
```

Cependant, le code est dupliqué ici aussi. En effet, le test **i >= 2** est une autre façon d'écrire que l'un des trois derniers tests est vrai. (On suppose pour les besoins de l'exemple que **i** a une valeur entière positive ou nulle.)

Comme vous le voyez, il faut parfois choisir entre l'élégance et la lisibilité.

Note : Il est tout à fait possible d'utiliser l'instruction **if else** avec une clause **if** vide. Cela permet d'indiquer une condition au lieu de son contraire, ce qui est parfois plus parlant. Ainsi, vous pouvez écrire :

```
if (i == 5);
else System.out.println(i);
```

ce qui est équivalent à :

```
if (i != 5) System.out.println(i);
```

Ici, l'avantage ne paraît pas évident. En revanche, en cas d'utilisation d'expressions logiques complexes, il peut être avantageux, du point de vue de la lisibilité, de ne pas utiliser leur négation.

La boucle *for*

La boucle **for** est une structure employée pour exécuter un bloc d'instructions un nombre de fois en principe connu à l'avance. Elle utilise la syntaxe suivante :

```
for (initialisation;test;incrémentation) {
   instructions;
}
```

Exemple :

```
int i = 0;
for (i = 2; i < 10;i++) {
   System.out.println("Vive Java !");
}
```

Lors de la première exécution de la boucle, **i** se voit affecter la valeur 2. L'expression logique **i < 10** est ensuite testée. Sa valeur étant **true**, la boucle est exécutée et le programme affiche la chaîne de caractères **"Vive Java !"**. La partie *incrémentation* est alors exécutée. L'expression logique est de nouveau évaluée, et le processus continue jusqu'à ce que la valeur de cette expression soit **false**. A ce moment, l'exécution se poursuit par la ligne suivant le bloc d'instruction de la boucle **for**.

Note : A la sortie de la boucle, la valeur de la variable servant d'indice (ici, **i**) est donc différente de la dernière valeur utilisée dans la boucle. Dans l'exemple dessus, la chaîne de caractères sera affichée 8 fois, pour les valeurs de **i** allant de 2 (valeur initiale) à 9. A la sortie de la boucle, **i** vaudra 10 (première valeur pour laquelle l'expression logique **i < 10** vaut **false**.

L'initialisation

La partie initialisation consiste en l'initialisation d'une ou de plusieurs variables, par exemple :

```
for (i = 1; ....
```

ou encore :

```
for (i = 1, j = 5, k = 12;....
```

Notez qu'avec ces syntaxes, la ou les variables doivent avoir été déclarées préalablement. En contrepartie, il est possible d'utiliser des variables de types différents, par exemple :

```
int i;
long j;
char k;
for (i = 1, j = 5, k = 'a';....
```

En revanche, il est possible d'utiliser la syntaxe suivante pour déclarer et initialiser les variables simultanément :

```
for (int i = 1, j = 5, k = 12;...
```

mais, dans ce cas, toutes les variables doivent être du même type.

Si vous utilisez une variable qui a été préalablement déclarée *et* initialisée, vous n'êtes pas obligé de l'initialiser une seconde fois. Il suffit de laisser vide la partie initialisation. Le point-virgule doit cependant être présent :

```
int i = 0;
for (;...
```

Le test

La partie *test* est composée d'une expression logique qui est évaluée et prend la valeur **true** ou **false**. Si la valeur est **true**, le bloc d'instructions suivant est exécuté. Dans le cas contraire, la boucle est terminée et l'exécution se poursuit à la ligne suivant le bloc.

La boucle **for** ne peut comporter qu'une seule expression logique. En revanche, celle-ci peut comprendre plusieurs expressions combinées à l'aide des opérateurs logiques, par exemple :

```
int i;
long j;
char k;
for (i = 0, j = 2, k = 'a'; i <= 10 || j <= 12 || k <= 'x';...
```

L'expression logique utilisée pour le test peut également comporter des opérateurs d'auto-incrémentation ou d'auto-décrémentation, bien que cela ne soit pas du tout conseillé. En effet, cela augmente considérablement les risques d'erreur, alors qu'il est pratiquement toujours possible de l'éviter. Évidemment, la différence entre pré et post-incrémentation/décrémentation ajoute encore à la confusion. Avouez qu'il n'est pas évident de comprendre immédiatement ce que fait le programme suivant :

```
int i;
long j;
char k;
for (i = 0, j = 2, k = 'a';
        --i <= 10 || j++ <= 12 || k-- <= 'x';
        i++, j += 2, k += 3) {
   System.out.println("i : " + i);
   System.out.println("j : " + j);
   System.out.println("k : " + k);
   }
```

La réponse est simple : il plante l'interpréteur Java ! En fait, cette boucle ne trouve aucune condition de sortie car **i** est pré-décrémenté dans le test et vaut donc -1 à la première itération. L'expression logique est donc vraie puisque son premier terme (**--i <= 10**) est vrai. En raison du "court-circuit", le reste de l'expression n'est pas évalué. La boucle est donc exécutée une première fois, puis **i** est incrémenté et vaut donc 0, **j** se voit augmenté de 2 et **k** de 3. Lorsque le test est effectué la fois suivante, **i** prend de nouveau la valeur -1 et tout recommence. Il s'agit donc d'une boucle infinie. Par ailleurs, à chaque boucle, **k** est augmenté de 3. Lorsque la valeur de **k** dépasse 56 319, l'interpréteur Java se "plante" ! (Bon, d'accord, il est rare d'avoir à imprimer des caractères ayant un code aussi élevé, mais c'est tout de même bon à savoir. Ce bug existe depuis la version 1.0 jusqu'à la version 1.2 bêta 3 sous les environnements Solaris, Windows 95 et Windows NT. Il est possible qu'il soit corrigé dans les versions ultérieures.)

Note : Pour réaliser une boucle infinie, il est plus simple d'utiliser **true** comme expression logique. Ainsi, dans l'exemple :

```
for (byte i = 0; true; i++) {
   System.out.println(i);
}
```

la boucle est effectuée indéfiniment puisque l'expression logique évaluée vaut toujours **true** !

Note 1 : Remarquez que ce programme ne produit pas de message d'erreur. Le débordement de la valeur de **i** lorsque celle-ci dépasse 127 lui fait prendre automatiquement la valeur -128 et la boucle continue de s'exécuter.

Note 2 : Si votre programme entre dans une boucle infinie, vous pouvez l'arrêter en tapant les touches Ctrl+C. (Du moins s'il s'agit d'une application. S'il s'agit d'une applet, il sera nécessaire de fermer le document qui la contient.)

L'incrémentation

Incrémentation est utilisé ici dans un sens générique signifiant "modification du ou des indices". L'incrémentation est l'opération la plus fréquente, mais vous pourrez effectuer toutes les opérations que vous souhaitez.

Cette partie peut d'ailleurs comporter un nombre quelconque d'instructions, séparées par des virgules. Elle peut, en particulier, ne comporter aucune instruction. (C'est la seule des trois parties qui puisse être vide.)

Les instructions présentes dans cette partie peuvent être utilisées pour leurs effets sur les variables indices, ou pour tout autre effet, par exemple :

```
for (int i = 1; i<10; i++, System.out.println(i)) {
   .
   .
   .
}
```

Notez l'absence de points-virgules à la fin des instructions.

Le bloc d'instructions

Les instructions exécutées par la boucle sont placées dans un bloc délimité par les caractères **{** et **}**. Si une seule instruction doit être exécutée, l'utilisation d'un bloc est facultative. Ainsi, les quatre formes suivantes sont équivalentes :

```
for (int i = 1; i<10; i++) {
   System.out.println(i);
}

for (int i = 1; i<10; i++) {System.out.println(i);}

for (int i = 1; i<10; i++)
   System.out.println(i);

for (int i = 1; i<10; i++) System.out.println(i);
```

La première est toutefois la plus lisible et nous vous conseillons de l'utiliser systématiquement (surtout si vous êtes un programmeur professionnel payé à la ligne de code).

Modification des indices à l'intérieur de la boucle

Il est tout à fait possible de modifier les indices utilisés pour contrôler la boucle à l'intérieur de celle-ci. Ainsi, le programme suivant fonctionne parfaitement mais ne s'arrête jamais :

```
for (int i = 1; i<10; i++) {
   System.out.println(i);
   i--;
}
```

La manipulation des indices à l'intérieur de la boucle est fortement déconseillée si vous souhaitez simplifier la maintenance de votre programme.

Comme vous pouvez le voir, la syntaxe des boucles **for** est très peu contraignante. Le revers de la médaille est qu'il est possible d'utiliser de nombreuses astuces qui rendent la maintenance des programmes hasardeuse. Considérez, par exemple, une boucle affichant les valeurs entières comprises entre 1 et une valeur quelconque **x**. Voici trois façons d'arriver à ce résultat :

```
for (int i = 1; i <= x; i++) System.out.println(i);
for (int i = 0; i++ < x;) System.out.println(i);
for (int i = 1; i <= x;) System.out.println(i++);
```

On peut trouver d'autres façons encore plus obscures. Inutile de préciser que nous vous conseillons fortement de vous en tenir à la première.

Imbrication des boucles for

Les boucles **for** peuvent être imbriquées. Par exemple, si vous souhaitez initialiser avec la valeur **z** les éléments d'un tableau d'entiers de dimensions **x** par **y** (nous étudierons les tableaux dans un prochain chapitre), vous pouvez utiliser deux boucles **for** imbriquées, de la façon suivante :

```
int z = 5;
int x = 8;
int y = 10;
for (int i = 0; i < x; i++) {
   for (int j = 0; j < y; j++) {
      tableau[i][j] = z;
   }
}
```

Type des indices

On peut se poser la question de savoir quel type de données il est préférable d'utiliser pour les indices des boucles. Il peut sembler, en effet, qu'il soit plus efficace d'utiliser le type **byte** pour les indices dont la valeur ne dépasse pas 127, et le type **short** pour les indices limités à 32 767. On peut même utiliser le type **char**, qui permet de compter jusqu'à 65 535 avec seulement 16 bits.

En fait, il n'en est rien. Les boucles fonctionnent beaucoup plus rapidement avec des indices de type **int** qu'avec tout autre type. Cela est dû au fait que Java convertit les **byte**, **short** et **char** en **int** avant d'effectuer des calculs, puis effectue la conversion inverse une fois le calcul effectué. Pour le vérifier, vous pouvez utiliser le programme suivant :

```
class TestFor {
   public static void main(String[] args) {
      int h = 0;
      long t = System.currentTimeMillis();
      for (byte i = 0; i < 127;i++) {
         for (byte j = 0; j < 127;j++) {
            for (byte k = 0; k < 127;k++) {
               for (byte l = 0; l < 127;l++) {
                  h = 2;
               }
            }
         }
      }
      t = System.currentTimeMillis() - t;
      System.out.println(t);
   }
}
```

qui affiche le temps mis pour effectuer 260 144 661 fois l'affectation de la valeur littérale 2 à la variable **h**. Ce programme affiche :

```
68770
```

Si vous utilisez des indices de type **int** :

```
      for (byte i = 0; i < 127;i++) {
         for (byte j = 0; j < 127;j++) {
            for (byte k = 0; k < 127;k++) {
               for (byte l = 0; l < 127;l++) {
```

le programme affiche :

```
43500
```

soit un gain de 36,7 % en performance.

Note : Si vous essayez ce programme, vous obtiendrez forcément des résultats différents. La vitesse d'exécution du programme dépend de nombreux facteurs, comme la puissance du processeur ou son occupation par d'autres tâches. Ce qui compte ici, c'est le rapport entre les deux valeurs. Notez que l'on peut constater ici une grande amélioration dans les casts de **byte** depuis la version 1.1 de Java. En effet, avec celle-ci, l'écart en performance était de 45,5 %.

Portée des indices

La portée des indices est un problème à ne pas négliger. Celle-ci dépend du choix que vous ferez d'initialiser les indices dans la boucle ou hors de celle-ci. En effet, les boucles **for** présentent une exception à la règle qui veut que les variables aient une portée limitée au bloc dans lequel elles sont déclarées. En effet, les variables d'indices déclarées dans la partie initialisation de l'instruction **for** ont une portée limitée au bloc exécuté par la boucle. En d'autres termes, le programme suivant fonctionne sans problème :

```
int i;
for (i = 0; i < 10; i++) {
   System.out.println(i);
}
System.out.println(i);
```

alors que celui-ci produit une erreur de compilation :

```
for (int i = 0; i < 10; i++) {
   System.out.println(i);
}
System.out.println(i);
```

car la variable **i** a une portée limitée au bloc d'instructions exécuté par la boucle. Elle n'existe donc plus lorsque la boucle est terminée.

Dans une prochaine section, nous étudierons un moyen permettant de sortir d'une boucle de façon anticipée, lorsqu'une condition est remplie, à l'aide de l'instruction **break**. Dans ce cas, il peut être nécessaire de connaître la valeur de l'indice au moment de la sortie. La façon politiquement correcte de procéder consiste à utiliser une autre variable et à lui affecter la valeur de l'indice au moment de la sortie. Par exemple :

```
int x = 0;
for (int i = 0; i < 10; i++) {
   if (tableau[i] = 5) {
      x = i;
      break;
}
System.out.println(x);
```

Ce programme parcourt les 10 éléments d'un tableau pour trouver le premier (s'il existe) dont la valeur est 5. Si cet élément est trouvé, la boucle est interrompue au moyen de l'instruction **break** et l'exécution continue à la ligne suivant le bloc d'instructions de la boucle. La valeur de l'indice a été préalablement affectée à la variable **x**, déclarée avant la boucle, et peut donc être utilisée après celle-ci.

Note : A la première ligne du programme, la variable **x** est déclarée et initialisée. Cette initialisation est obligatoire (dans la mesure où **x** n'est pas une variable d'instance) car le compilateur ne peut pas être sûr que **x** sera initialisée avant la dernière ligne et refuse donc de compiler le programme. Nous pouvons en être sûrs car nous voyons que les paramètres de la boucle **for** garantissent qu'elle sera exécutée au moins une fois. Le compilateur, lui, n'est pas assez "intelligent" pour le comprendre. Il est toutefois prudent puisque le message d'erreur affiché est :

```
Variable x may not have been iniatilized.
```

(*Il se pourrait que la variable x ne soit pas initialisée.*)

Une autre solution consiste à utiliser directement l'indice de la boucle, en le déclarant avant celle-ci, de la façon suivante :

```
int i;
for (i = 0; i < 10; i++) {
   if (tableau[i] = 5) {
      break;
}
System.out.println(i);
```

Cette méthode n'est pas conseillée. En effet, il est toujours préférable de réserver les indices pour leur fonction primaire, qui est de compter les itérations de la boucle. Une utilisation à l'extérieur de la boucle peut conduire à des problèmes difficiles à détecter. Comparez, par exemple, les deux programmes suivants :

```
class For2 {
   public static void main(String[] args) {
      int x = 0;
      for (i = 5; i < 10; i = i + 2) {
         x = i;
      }
      System.out.println(x);
   }
}
```

et :

```
class For1 {
   public static void main(String[] args) {
      int i;
      for (i = 5; i < 10; i = i + 2) {
      }
      System.out.println(i);
   }
}
```

Le premier programme, construit de la manière recommandée, affiche la valeur de l'indice lors de la dernière itération de la boucle, c'est-à-dire 9. Le deuxième programme, pour sa part, affiche la valeur qui a provoqué la sortie de la boucle, c'est-à-dire 11. Cela paraît évident parce que les indices sont initialisés à 5 et incrémenté de 2 à chaque boucle. En revanche si, comme c'est souvent le cas, on utilise une valeur initiale de 0 et un incrément de 1, on arrive à une situation confuse. Le premier programme :

```
class For2 {
   public static void main(String[] args) {
      int x = 0;
      for (i = 0; i < 10; i++) {
         x = i;
      }
      System.out.println(x);
   }
}
```

affiche 9, ce qui est clairement la valeur de l'indice lors de la dernière itération de la boucle, alors que le deuxième :

```
class For1 {
   public static void main(String[] args) {
      int i;
      for (i = 0; i < 10; i++) {
      }
      System.out.println(i);
   }
}
```

affiche 10, ce qui peut facilement être confondu avec la valeur de la limite ou, plus grave, avec le nombre d'itérations, qui se trouve, mais c'est un cas particulier, être aussi égal à 10.

Sortie d'une boucle par return

Nous avons déjà évoqué l'utilisation de l'instruction **return** pour modifier le cours de l'exécution du programme. Lorsqu'une boucle **for** est employée dans une méthode, il est possible de sortir de la boucle au moyen de l'instruction **return**, comme dans l'exemple suivant :

```
for (int i = 0; i < 10; i++) {
   if (tableau[i] = 5) {
      return i;
}
```

Ce programme parcourt les 10 éléments d'un tableau pour trouver le premier (s'il existe) dont la valeur est 5. Si cet élément est trouvé, la méthode retourne l'indice de l'élément correspondant.

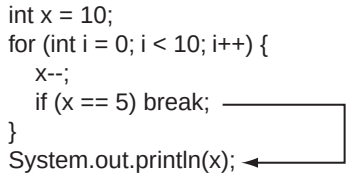
Notez qu'ici, il est parfaitement sûr d'utiliser la variable indice avec l'instruction **return** car, en Java, les primitives sont passées *par valeur* et non *par référence*, ce qui signifie que ce n'est pas une référence à la variable **i** qui est retournée, mais uniquement sa valeur.

Branchement au moyen des instructions *break* et *continue*

Comme nous l'avons vu précédemment, il est possible d'interrompre une boucle au moyen de l'instruction **break**. Celle-ci a deux effets distincts :

- Interruption de l'itération en cours.
- Passage à l'instruction suivant la boucle.

L'effet de l'instruction **break** est illustré sur le schéma suivant :



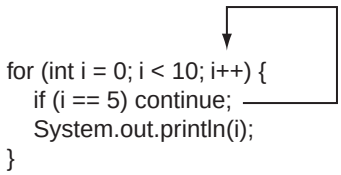
Note : L'instruction **break** ne modifie pas la valeur de la variable indice (ici **i**), contrairement à l'exemple suivant (fortement déconseillé), qui simule le précédent :

```
int x = 10;
for (int i = 0; i < 10; i++) {
   x--;
   if (x == 5) i = 10;
}
System.out.println(x);
```

L'instruction **continue** permet également de sortir d'une boucle. Elle a les deux effets suivants :

- Interruption de l'itération en cours.
- Retour au début de la boucle avec exécution de la partie *incrémentation*.

En d'autres termes, l'instruction **continue** interrompt l'itération en cours et passe directement à l'itération suivante. Son fonctionnement est illustré par le schéma suivant :



Ce programme a pour effet d'afficher toutes les valeurs de l'indice, de 0 à 9, à l'exception de 5. Il peut être simulé de deux façons :

```
for (int i = 0; i < 10; i++) {
   if (i == 5) i++;
   System.out.println(i);
}
```

ou :

```
for (int i = 0; i < 10; i++) {
   if (i == 5);
   else {
   System.out.println(i);
   }
}
```

La première façon, comme toutes les manipulations de l'indice dans la boucle, est fortement déconseillée.

Utilisation de break *et* continue *avec des étiquettes*

Dans le cas de boucles imbriquées que nous avons étudié précédemment, les instructions **break** et **continue** n'agissent qu'au niveau de la boucle interne. Ainsi, dans l'exemple suivant :

```
int z = 5;
int x = 8;
int y = 10;
for (int i = 0; i < x; i++) {
   for (int j = 0; j < y; j++) {
      if (tableau[i][j] != 0) continue;
      tableau[i][j] = z;
   }
}
```

si l'élément de tableau a une valeur non nulle, l'exécution se poursuit à la prochaine itération de la boucle interne. Si vous voulez que l'exécution se poursuive à la prochaine itération de la boucle externe, vous devez utiliser une *étiquette*. Les étiquettes sont des identificateurs Java se terminant par le signe :. Pour obtenir l'effet recherché dans l'exemple précédent, vous pouvez écrire :

```
int z = 5;
int x = 8;
int y = 10;
etiquette:
for (int i = 0; i < x; i++) {
   for (int j = 0; j < y; j++) {
      if (tableau[i][j] != 0) continue etiquette;
      tableau[i][j] = z;
   }
}
```

Il ne doit jamais y avoir d'instruction entre l'étiquette et la boucle concernée. Dans le cas contraire, Java est incapable de trouver l'étiquette lorsqu'il rencontre l'instruction **continue** ou **break**. Par ailleurs, il est impossible d'interrompre une itération de boucle extérieure pour effectuer un branchement vers une étiquette de boucle intérieure. Le branchement se fait toujours en "remontant vers la surface".

L'instruction *while*

Il arrive fréquemment qu'une boucle doive être exécutée tant qu'une condition est ou n'est pas remplie, sans que cette condition porte sur une forme d'indice. L'instruction **while** est prévue pour le cas où la condition doit être testée avant toute exécution. Le programme suivant recherche le premier élément non nul d'un tableau de valeur de type **int**.

```
int i = 0, x = 0;
while (x == 0) {
     x = tableau[i++];
}
```

L'instruction **while** est utile lorsque le nombre maximal d'itérations n'est pas connu à l'avance, et lorsque la condition de sortie est indépendante de celui-ci. Notez qu'ici, ce nombre pourrait être connu en déterminant la longueur du tableau. Nous pourrions donc utiliser une boucle **for**. (Il serait

même plus prudent de le faire, car la méthode employée ici risque de nous faire dépasser les limites du tableau si aucun élément satisfaisant la condition n'est trouvé.) Cependant, la condition de sortie étant déterminée par la valeur de l'élément de tableau, il nous faudrait utiliser également une instruction **break** pour sortir de la boucle :

```
int x = 0;
int y = tableau.length;
for (int i = 0; i < y;i++) {
     x = tableau[i];
     if (x != 0) break;
}
```

Notez que **tableau.length()** donne le nombre d'éléments du tableau. Les éléments étant numérotés à partir de 0, nous utilisons l'opérateur < et non <=. Tout cela sera plus clair lorsque nous étudierons les tableaux.

Ce code est beaucoup moins compact que la version précédente. (En revanche, il ne risque pas de provoquer d'erreur si on dépasse les limites du tableau !) Pour ce qui est des performances, le temps d'exécution est supérieur de plus de 57 %. Notez que l'on peut améliorer la situation en utilisant la forme suivante :

```
int x = 0, y = tableau.length;
for (int i = 0; i < y;i++) {
     if (tableau[i] != 0) {
        x = tableau[i];
        break;
     }
}
```

Le temps d'exécution est alors supérieur de seulement 12 % à celui de la version avec **while**. Notez qu'il est possible de réduire cet écart à 0 en utilisant la forme suivante :

```
int x = 0, i = 0, y = tableau.length;
for (; i < y;i++) {
   if (tableau[i] == 0) continue;
      x = tableau[i];
      break;
}
```

Le moins qu'on puisse dire est que l'on ne gagne pas en lisibilité !

Il existe une solution qui permet d'allier lisibilité, sécurité et performance. Il suffit de prévoir un tableau dont le dernier élément satisfait toujours la condition. De cette façon, on ne risque pas de dépasser les limites du tableau. Il suffit de tester l'indice à la sortie de la boucle pour savoir si l'élément trouvé est le dernier du tableau :

```
int[] tableau = new int[101];
tableau[100] = 1;
.
.
. // initialisation des éléments du tableau
.
.
int i = 0, x = 0;
while (x == 0) {
      x = tableau[i++];
}
if (i < 100) {...
   // Aucune valeur trouvée
}
```

Un autre possibilité consiste à utiliser une valeur particulière pour l'indicateur de fin de tableau, par exemple une valeur négative si on sait que toutes les valeurs seront positives ou nulles :

```
int[] tableau = new int[100];
tableau[100] = -1;
```

```
.
.
. // initialisation des éléments du tableau
.
.
int i = 0, x = 0;
while (x == 0) {
      x = tableau[i++];
}
if (x < 0) {...
   // Aucune valeur trouvée
}
```

Dans ces deux cas, le test permettant de savoir si on a atteint les limites du tableau n'est effectué qu'une seule fois à la fin du traitement, ce qui n'a qu'une influence négligeable sur les performances. (Tout cela sera évidemment plus clair lorsque nous aurons étudié les tableaux !)

L'instruction **do...while** correspond au cas où la condition testée dépend de l'issue d'un traitement. En d'autres termes, la boucle doit être effectuée au moins une fois pour que le test puisse avoir lieu. Par exemple, si vous comptez les occurrences d'un caractère dans une chaîne, l'algorithme sera (en supposant qu'une recherche qui aboutit est signalée par la valeur **true** de la variable de type **boolean trouvé**) :

```
do
   rechercher le caractère
   if (trouvé) augmenter le compte d'une unité
while (trouvé)
```

Dans un cas comme celui-ci, il serait impossible d'effectuer le test au début de la boucle, puisque la valeur testée dépend de l'exécution de la boucle.

Il est parfois possible de choisir presque indifféremment la forme **while** ou la forme **do...while**. Ainsi, notre exemple précédent peut être réécrit de la façon suivante :

```
int i = 0;
int x;
do {
   x = tableau[i++];
}
while (x == 0);
```

(Attention de ne pas oublier le point-virgule après le test.)

La différence entre les deux boucles apparaît clairement ici : avec la version **do...while**, il n'est pas nécessaire que la variable **x** soit initialisée avant la boucle puisque le test a lieu à la fin de celle-ci et que la variable reçoit une valeur à l'intérieur de la boucle.

Les instructions **break** et **continue** peuvent naturellement être utilisées avec les boucles **while** et **do...while** comme avec les boucles **for**. L'instruction **break** interrompt l'itération en cours et fait continuer l'exécution après la fin du bloc dans le cas de l'instruction **while**, et après le test dans le cas de **do...while**.

L'instruction **continue** présente une petite particularité. En effet, elle interrompt l'itération en cours et fait continuer l'exécution au début de la boucle dans le cas de **do...while**, ou juste avant le test dans le cas de **while**. Cependant, si **continue** est utilisée avant la partie de la boucle qui modifie la condition à tester, le programme tombe dans une boucle infinie, comme dans l'exemple suivant :

```
class TestWhile {
   public static void main(String[] args) {
      int i = 0;
      do {
         if (i == 10) continue;
         System.out.println(i++);
      }
      while (i <= 20) ;
   }
}
```

Ce programme est supposé afficher les valeurs de **i** de 0 à 20, à l'exception de 10. Cependant, lorsque **continue** est exécutée, **i** n'a pas été incrémenté. La solution consiste à traiter la modification de la condition avant d'exécuter **continue** :

```
class TestWhile2 {
   public static void main(String[] args) {
      int i = 0;
      do {
         if (i == 10) {
            i++;
            continue;
         }
         System.out.println(i++);
      }
      while (i <= 20) ;
   }
}
```

Il peut venir à certains l'idée de traiter la modification de la condition en même temps que le test déterminant l'exécution de l'instruction **continue** :

```
class TestWhile3 {
   public static void main(String[] args) {
      int i = 0;
      do {
         if (++i == 10) continue;
         System.out.println(i);
      }
      while (i <= 20) ;
   }
}
```

Cependant, ce programme ne donne pas le même résultat. En effet, il imprime les valeurs de 1 à 21 et non de 0 à 20. Le résultat correct serait obtenu

en initialisant **i** à -1 et en remplaçant le test par **i <= 21**, mais le programme deviendrait difficile à lire !

L'instruction *switch*

Nous avons vu que l'instruction **if...else if** permet de traiter plusieurs cas sous la forme :

```
if (condition1) {
}
else if (condition2) {
}
else if (condition3) {
}
etc.
```

Il arrive souvent que les différentes conditions correspondent à différentes valeurs d'une variable. Si cette variable (appelée *sélecteur*) est entière, Java permet d'utiliser l'instruction **switch**, dont la syntaxe est la suivante :

```
switch(variable) {
   case valeur1: instructions1;
   case valeur2: instructions2;
   case valeur3: instructions2;
      .
      .
      .
   case valeurN: instructionsN;
   default: instructions;
}
```

Le fonctionnement de cette structure est le suivant :

- La valeur de la variable est comparée avec *valeur1*. En cas d'inégalité, la variable est comparée avec *valeur2*, et ainsi de suite jusqu'à ce qu'une égalité soit trouvée.

- Lorsqu'une égalité est trouvée, le bloc d'instructions correspondant est exécuté. **Tous les blocs d'instructions suivants sont ensuite exécutés**. Il est très important de bien réaliser cette particularité. En effet, il pourrait sembler naturel que seul le bloc correspondant à la condition d'égalité soit exécuté. Il n'en est rien. (Ici, *bloc* désigne toutes les instructions comprises entre deux instructions **case**, que celles-ci soient ou non encadrées par les symboles **{** et **}**.)

- Si aucune égalité n'est trouvée, le bloc correspondant à l'étiquette **default:** est exécuté.

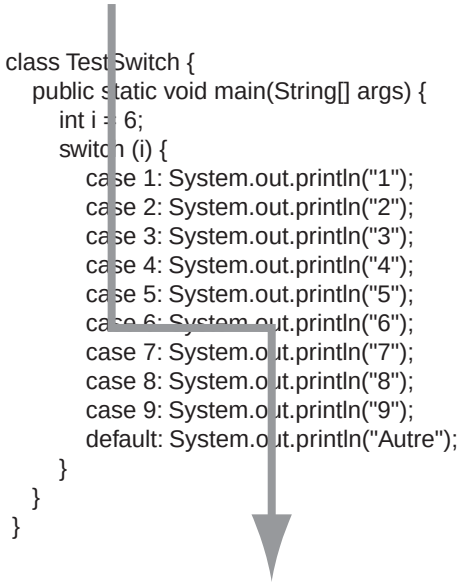
Par exemple le programme suivant :

```
class TestSwitch {
   public static void main(String[] args) {
      int i = 6;
      switch (i) {
         case 1: System.out.println("1");
         case 2: System.out.println("2");
         case 3: System.out.println("3");
         case 4: System.out.println("4");
         case 5: System.out.println("5");
         case 6: System.out.println("6");
         case 7: System.out.println("7");
         case 8: System.out.println("8");
         case 9: System.out.println("9");
         default: System.out.println("Autre");
      }
   }
}
```

affiche :

```
6
7
8
9
Autre
```

Le déroulement du programme peut être représenté de la façon suivante :



Si vous souhaitez que seul le bloc d'instructions correspondant à la valeur de la variable sélecteur soit exécuté, il suffit de terminer le bloc par une instruction **break**. L'exemple précédent devient :

```
class TestSwitch2 {

   public static void main(String[] args) {
      int i = 6;
      switch (i) {
         case 1:
            System.out.println("1");
            break;
         case 2:
            System.out.println("2");
            break;
         case 3:
            System.out.println("3");
            break;
```

```
            case 4:
               System.out.println("4");
               break;
            case 5:
               System.out.println("5");
               break;
            case 6:
               System.out.println("6");
               break;
            case 7:
               System.out.println("7");
               break;
            case 8:
               System.out.println("8");
               break;
            case 9:
               System.out.println("9");
               break;
            default:
               System.out.println("Autre");
         }
      }
   }
```

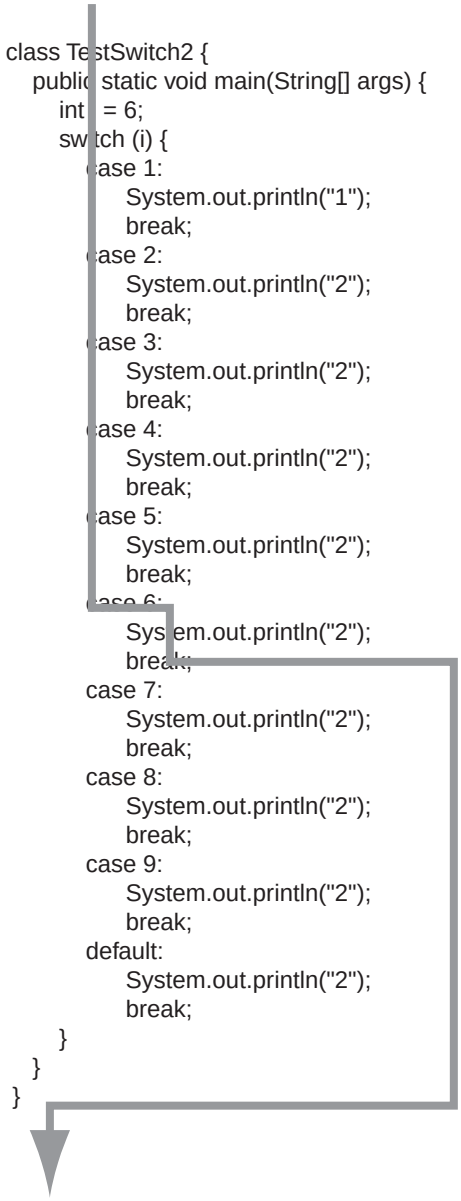
Ce programme n'affiche plus que :

```
6
```

Son déroulement est représenté sur la figure de la page ci-contre.

Il n'est pas du tout nécessaire que les valeurs de la variable sélecteur soient placées dans l'ordre croissant. Ainsi le programme suivant fonctionne parfaitement :

```
class TestSwitch3 {
   public static void main(String[] args) {
      int i = 3;
      switch (i) {
```


class TestSwitch2 {
public static void main(String[] args) {
int i = 6;
switch (i) {
case 1:
System.out.println("1");
break;
case 2:
System.out.println("2");
break;
case 3:
System.out.println("2");
break;
case 4:
System.out.println("2");
break;
case 5:
System.out.println("2");
break;
case 6:
System.out.println("2");
break;
case 7:
System.out.println("2");
break;
case 8:
System.out.println("2");
break;
case 9:
System.out.println("2");
break;
default:
System.out.println("2");
break;
}
}
}

```
            case 1:
               System.out.println("1");
               break;
            case 4:
               System.out.println("4");
               break;
            default:
               System.out.println("Autre");
               break;
            case 3:
               System.out.println("3");
               break;
            case 2:
               System.out.println("2");
         }
      }
   }
```

Notez cependant que si un seul bloc doit être exécuté pour chaque valeur du sélecteur, seul le dernier bloc peut être dispensé de l'instruction **break**.

Par ailleurs, un bloc d'instruction peut être vide, ou comporter seulement une instruction **break** si aucun traitement ne doit être effectué pour la valeur correspondante. De plus, le bloc **default:** est facultatif.

Toutes ces particularités peuvent être mises à profit pour créer un exemple convertissant les minuscules en majuscules (nous supposerons qu'il existe un morceau de programme par ailleurs pour vérifier si les valeurs soumises correspondent bien à des caractères) :

```
class Conversion {

   public static void main(String[] args) {
      System.out.println(conv('a'));
      System.out.println(conv('î'));
      System.out.println(conv('ç'));
```

```
      System.out.println(conv('c'));
      System.out.println(conv('e'));
      System.out.println(conv('ê'));
      System.out.println(conv('ö'));
      System.out.println(conv('o'));
   }

   static char conv (char c) {
      switch (c) {
         case 'à':
         case 'â':
         case 'ä':
            return 'A';
         case 'é':
         case 'è':
         case 'ë':
         case 'ê':
            return 'E';
         case 'ï':
         case 'î':
            return 'I';
         case 'ô':
         case 'ö':
            return 'O';
         case 'ü':
         case 'û':
         case 'ù':
            return 'U';
         case 'ç':
            return 'C';
         default:
            return (char)(c - 32);
      }
   }
}
```

Ce programme affiche :

```
A
I
C
C
E
E
O
O
```

Notez qu'ici, nous n'avons pas utilisé l'instruction **break** car l'instruction **return** permet d'obtenir le même résultat.

L'instruction **switch** voit son intérêt limité par le fait qu'il n'est possible de tester que des valeurs numériques entières, et non des expressions logiques comme dans la plupart des autres langages possédant ce type d'instruction. (Ici, les caractères peuvent être testés car le type **char** est en fait un type numérique entier non signé sur 16 bits.) Si vous devez tester des expressions logiques, vous devrez vous contenter d'une succession d'instructions **if** et **else if**.

L'instruction *synchronized*

L'instruction **synchronized** est une sorte d'instruction conditionnelle liée à l'utilisation d'une fonction particulière de Java appelée *multithreading*, qui permet à un programme d'exécuter plusieurs processus simultanément. L'instruction **synchronized** prend pour paramètre une référence d'objet (handle ou expression) et exécute le bloc de code suivant seulement si l'objet a pu être verrouillé afin d'en interdire l'accès aux autres processus. Sa syntaxe est la suivante :

```
synchronized (objet) {
   bloc d'instructions;
}
```

Nous reviendrons sur ce sujet au cours des prochains chapitres.

Résumé

Dans ce chapitre, nous avons étudié la syntaxe des instructions Java permettant de contrôler le flux d'exécution du programme. Cette partie plutôt rébarbative de l'apprentissage était cependant nécessaire à la suite de notre étude. Le prochain chapitre sera consacré à un sujet tout à fait différent : les conditions d'accessibilité aux éléments de Java.

Chapitre 8

L'accessibilité

Dans les chapitres précédents, tous les identificateurs que nous avons utilisés étaient accessibles pratiquement sans restriction. Les variables pouvaient être lues et modifiées sans contraintes. Les classes pouvaient être instanciées à volonté. En usage normal, cette situation est dangereuse. Par exemple, si vous concevez une classe pour la mettre à la disposition de programmeurs, il est souhaitable que vous puissiez exercer un certain contrôle sur la façon dont ceux-ci pourront l'utiliser. Vous pourrez, en particulier, limiter l'accès à certaines variables membres, autoriser ou interdire l'instanciation, limiter l'utilisation de certaines méthodes, restreindre les possibilités de création de nouvelles classes par extension de la vôtre ou, au contraire, n'autoriser que ce type d'utilisation.

Un des avantages essentiels de la restriction d'accès à certains éléments est que cette technique permet de se réserver la possibilité de modifier le

fonctionnement d'un programme sans que cela change quoi que se soit pour l'utilisateur. Par exemple, supposons que nous créions une classe représentant une voiture et possédant une variable membre statique **capacité** représentant la capacité du réservoir d'essence et une variable d'instance **carburant** représentant la quantité de carburant se trouvant dans le réservoir. Il est clair que nous ne devons pas autoriser l'utilisateur à modifier la valeur de **capacité**. En revanche, il pourrait sembler naturel de l'autoriser à lire cette valeur, ainsi qu'à lire et modifier la valeur de **carburant**. Cependant, si nous le laissons "faire le plein" en modifiant la valeur de **carburant**, rien ne nous permet de nous assurer qu'il ne va pas faire déborder le réservoir en affectant à cette variable une valeur supérieure à celle de **capacité**. Par ailleurs, si nous décidons un jour d'améliorer notre voiture en la dotant d'un double réservoir, nous nous trouverons dans une situation très délicate. En effet, tous les utilisateurs de notre classe devront modifier leurs programmes pour tenir compte de ce qui est, pour nous, une amélioration, mais risque d'être perçu par eux comme une lourde contrainte. La solution consiste à définir très précisément la façon dont les utilisateurs peuvent interférer avec notre classe.

Par exemple, nous pourrons définir deux méthodes, **capacité()** et **carburant()**, permettant respectivement de connaître les valeurs de **capacité** et de **carburant**, ainsi que de modifier cette dernière valeur. Dans la première version de la classe (un seul réservoir), la méthode **capacité()** retournera la valeur de la variable **capacité**. La méthode **carburant()** sans argument retournera la valeur de **carburant** alors que la version surchargée **capacité(float c)** ajoutera la valeur de **c** à **carburant** tout en limitant le total à la valeur de **capacité** et en retournant **capacité - (carburant + c)**, c'est-à-dire une valeur représentant le volume restant libre dans le réservoir si la valeur est positive, et la quantité de carburant qui a débordé dans le cas contraire.

De cette façon, si nous modifions la classe pour créer une version à double réservoir, il nous suffira de modifier les méthodes pour en tenir compte. Du point de vue de l'utilisateur, rien n'aura changé. Cette possibilité de "cacher" la mécanique interne est parfois appelée *encapsulation*. Dans d'autres langages, l'ensemble des éléments accessibles à l'utilisateur est parfois appelé *interface*. Cependant, en Java, ce terme signifie tout à fait autre chose.

Les méthodes qui permettent de lire les données contenues dans les objets sont appelées *accesseurs (accessors)*, alors que celles qui permettent de les modifier sont appelées *mutateurs (mutators)*. Une méthode peut, bien sûr, modifier *et* lire des données, mais cela est normalement déconseillé si votre classe doit être utilisée par d'autres programmeurs. L'usage est même de faire commencer le nom des accesseurs par **get** et celui des mutateurs par **set**. Bien sûr, avec des noms de méthodes en français, cela peut paraître bizarre. Ainsi, dans l'exemple précédent, la méthode (statique) permettant de connaître la capacité du réservoir serait nommée **getCapacité()**, celle permettant de savoir combien d'essence il reste **getCarburant()** et celle permettant de faire le plein **setCarburant()**. Normalement, cette dernière méthode ne devrait pas renvoyer une valeur mais, en cas de débordement, placer la quantité excédentaire dans une variable et renvoyer une condition d'erreur au moyen d'un mécanisme que nous n'avons pas encore étudié.

Écrire des applications en Java consiste à créer des classes (au moins une). Une classe peut contenir des primitives, des blocs d'instructions, des constructeurs, des méthodes, des objets (nous avons déjà utilisé toutes ces possibilités), ainsi que d'autres classes et d'autres éléments tels que des interfaces. (Les interfaces seront présentées plus loin. Les classes incluses dans d'autres classes feront l'objet d'un prochain chapitre.)

Lorsque nous créons un objet en instanciant une classe, ou lorsque nous utilisons un membre statique d'une autre classe (méthode ou variable), nous devons disposer d'un moyen d'y faire référence. Jusqu'ici, nous avons utilisé sans trop y faire attention deux façons d'accéder à des classes. La première façon consistait à créer dans notre application la classe principale (celle comportant la méthode **main**) ainsi que les autres classes dont nous avions besoin. Par exemple, au Chapitre 3, nous avons créé le programme suivant :

```
public class Test {
   public static void main(String[] argv) {
      new Employe();
   }
}
class Employe {
```

```
    int matricule;
    static int nombre;
    Employe() {
       matricule = ++nombre;
       afficherMatricule();
    }
    void afficherMatricule() {
       System.out.println(matricule);
    }
}
```

(Notez que nous avons ici ajouté une ligne.)

Ce programme, stocké dans le fichier **Test.java**, contient la définition de deux classes : **Test** (la classe principale contenant la méthode **main**) et **Employe**. Lorsque ce programme a été compilé, le compilateur a créé deux fichiers dans le dossier où se trouvait le fichier source **Test.java** : **Test.class** et **Employe.class**. Pour utiliser la classe **Employe** afin d'en créer une instance (ce qui est fait dans la méthode **main** de la classe **Test**) à l'aide de l'instruction :

```
new Employe();
```

était-il nécessaire que la classe **Employe** soit définie dans le même fichier que la classe **Test** ? Oui et non. Non, car le fichier **Employe.class** aurait très bien pu être créé à la suite de la compilation d'un fichier source indépendant, **Employe.class**. Cependant, dans ce cas, la méthode **main** de la classe **Test** aurait dû disposer de deux éléments pour l'utiliser :

- Son adresse, c'est-à-dire l'endroit où se trouve la classe, sur le disque dur ou sur un serveur relié à un réseau.
- Une autorisation d'accès.

Placer le code source de la classe **Employe** dans le même fichier que le code source de la classe qui l'utilise (**Test**) revient à :

- Créer le fichier **Employe.class** dans le même dossier que le fichier **Test.class** ;

- Donner à la classe **Test** l'autorisation d'utiliser la classe **Employe**.

Si vous compilez séparément les deux classes, vous pourrez facilement remplir la première condition. En revanche, la deuxième ne sera pas remplie par défaut. Il vous faudra donc utiliser un modificateur d'accès pour donner explicitement une autorisation.

Une troisième condition doit être remplie pour que la classe **Test** puisse utiliser la classe **Employe** : il faut que le type d'utilisation souhaité (ici l'instanciation) soit possible. Pour résumer, les trois critères permettant d'utiliser une classe sont *Qui*, *Quoi*, *Où*. Il faut donc :

- Que l'utilisateur soit autorisé (*Qui*).

- Que le type d'utilisation souhaité soit autorisé (*Quoi*).

- Que l'adresse de la classe soit connue (*Où*).

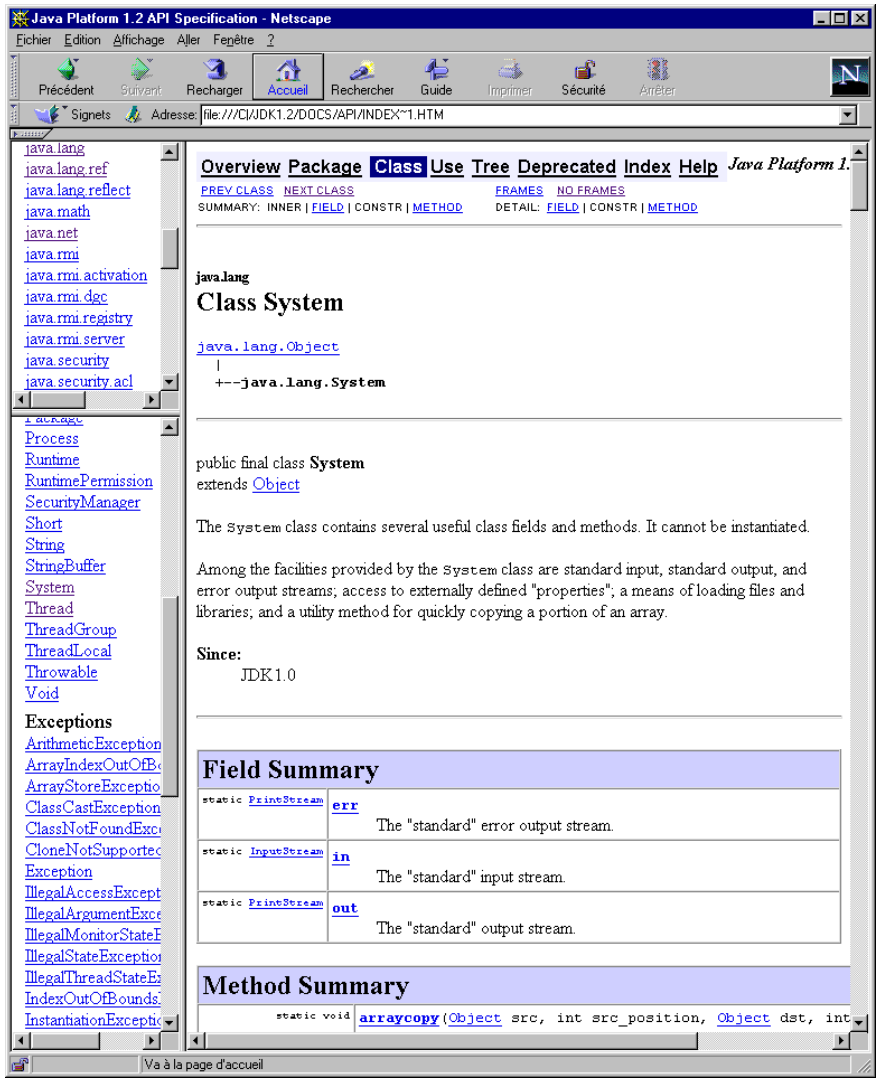
Nous commencerons notre étude par le troisième critère : Où.

Les packages (Où)

Dans l'exemple de programme précédent, la méthode **afficherMatricule** utilise une classe appelée **System** :

```
void afficherMatricule() {
   System.out.println(matricule);
}
```

Cette classe n'est pas instanciée. Il n'y a donc pas de tentative de création d'un nouvel objet instance de la classe **System**. En revanche, la deuxième ligne du code ci-dessus invoque la méthode **println** du membre **out** de la classe **System**, ce que nous pouvons vérifier en consultant la documentation de Java comme nous avons appris à le faire au Chapitre 2 :



La documentation nous informe également de l'endroit où se trouve cette classe : le chemin d'accès complet à la classe **System** est **java.lang.System**. Comme nous l'avons déjà dit au Chapitre 2, **java** et **lang** sont des dossiers. **java.lang.System** est donc un chemin d'accès, qui présente la particularité, par rapport aux chemins d'accès de Windows, d'utiliser le point comme séparateur de niveaux au lieu de la barre oblique inverse. En effet, les sépa-

rateurs de niveaux varient selon les systèmes (\ sous Windows, / sous Unix) et peuvent même parfois être configurés au gré de l'utilisateur. Pour que les programmes Java puissent être exécutés sur toutes les plates-formes sans modification, Java utilise son propre séparateur et effectue automatiquement la conversion lorsque c'est nécessaire. L'utilisation du point n'entre pas en conflit avec celle qui est faite par Windows car les noms de classes ne comportent pas d'extension en Java. (En revanche, les fichiers Windows qui les contiennent portent l'extension **.class**.)

Le "dossier" qui contient des classes est appelé *package*. La classe **System** fait donc partie du package **java.lang**.

Les chemins d'accès Windows commencent au *poste de travail*, mais il est possible de n'en spécifier qu'une partie. On parle alors de *chemin d'accès relatif*. Si vous vous trouvez dans le répertoire **c:\programmation\java**, et que vous vouliez accéder au fichier **c:\programmation\java\util\dates\vacances.java**, vous pouvez le faire en indiquant simplement **util\dates\vacances.java**. Windows reconstitue le chemin d'accès complet en mettant bout à bout le chemin d'accès courant (**c:\programmation\java**) et le chemin d'accès relatif (**util\dates\vacances.java**). De plus, vous pouvez également spécifier des chemins d'accès par défaut, grâce à la commande **path**, généralement utilisée au démarrage d'une session DOS. (Voir Chapitre 1.) Ainsi, si vous avez utilisé la commande :

```
path c:\windows;c:\windows\system;c:\jdk1.2\bin
```

et si vous vous trouvez dans le répertoire **c:\programmation\java**, Windows cherchera le fichier **util\dates\vacances.java** en utilisant successivement les chemins d'accès suivants :

```
c:\programmation\java\util\dates\vacances.java
c:\windows\util\dates\vacances.java
c:\windows\system\util\dates\vacances.java
c:\jdk1.2\bin\util\dates\vacances.java
```

Java dispose d'un mécanisme équivalent pour la recherche des classes. Java recherche les classes en priorité :

- Dans le répertoire courant, c'est-à-dire celui où se trouve la classe appelante, si la variable d'environnement **CLASSPATH** n'est pas définie ;
- Dans les chemins spécifiés par la variable d'environnement **CLASSPATH** si celle-ci est définie.

Sous Windows, la variable d'environnement **CLASSPATH** peut être définie à l'aide de la commande :

```
set classpath=.;c:\programmation\java;c:\appli\java\util
```

Sous Unix, la syntaxe à utiliser est :

```
setenv classpath .:/home/programmation/java;/home/appli/java/util
```

Les différents chemins d'accès sont séparés par le caractère : sous Unix et ; sous Windows. Notez l'utilisation du point comme premier chemin d'accès. Le point désigne le répertoire courant. En effet, il faut se souvenir que le répertoire courant est utilisé par défaut uniquement si la variable **CLASSPATH** n'est pas définie.

Note : Si vous souhaitez supprimer toute définition de la variable d'environnement, utilisez la syntaxe :

```
set classpath=
```

Chemins d'accès et packages

Tout cela signifie-t-il que package = chemin d'accès ? Pas du tout ! Un package est une unité logique regroupant un certain nombre de classes, au gré de leur concepteur. (Les raisons qui conduisent à placer des classes dans un package commun seront abordées dans une prochaine section.) Une classe

peut être définie sans indiquer à quel package elle appartient. Elle est alors considérée comme faisant partie du package par défaut. Celui-ci ne correspond à aucun répertoire particulier. Ses classes peuvent être placées n'importe où, du moment que le chemin d'accès par défaut ou un de ceux définis par la variable d'environnement **CLASSPATH** permet de les atteindre. On voit donc que cette variable d'environnement spécifie un chemin d'accès aux fichiers contenant les classes, et non les packages les contenant. La syntaxe est similaire, mais les deux concepts sont légèrement différents.

L'instruction package

Si vous souhaitez qu'une classe que vous avez créée appartienne à un package particulier, vous devez le spécifier explicitement au moyen de l'instruction **package**, suivie du nom du package. Cette instruction doit être la première du fichier. Elle concerne toutes les classes définies dans ce fichier.

L'indication du package ne suffit pas. En effet, les fichiers **.class** résultant de la compilation ne sont pas placés automatiquement dans le répertoire correspondant. En revanche, leur appartenance à un package est codée dans le fichier compilé, qui ne peut plus être utilisé autrement que depuis le chemin d'accès correspondant. En d'autres termes, si vous compilez les deux fichiers suivants :

```
package monpackage;
public class Employe {
   int matricule;
   static int nombre;

   Employe() {
      matricule = ++nombre;
      afficherMatricule();
   }
   void afficherMatricule() {
      System.out.println(matricule);
   }
}
```

et :

```
package monpackage;
public class Test {
   public static void main(String[] argv) {
      new monpackage.Employe();
   }
}
```

tout se passera bien à la compilation, mais vous obtiendrez un message d'erreur à la compilation du second :

```
Test.Java:3: Class monpackage.Employe not found in type declaration.
              new monpackage.Employe();
                             ^
1 error
```

Cela est tout simplement dû au fait que, la classe **Employe** étant déclarée appartenir au package **monpackage**, le fichier **Employe.class** doit se trouver dans le répertoire correspondant, alors qu'il se trouve pour l'instant dans le répertoire courant, tout comme le fichier **Test.class**.

Pour que la classe **Test** puisse être compilée, il suffit de déplacer le fichier **Employe.class** dans un répertoire appelé **monpackage**, que vous devez créer pour l'occasion. Où ça ? Dans un des répertoires accessibles par défaut, c'est-à-dire dans le répertoire courant si la variable d'environnement n'est pas définie, ou dans un des répertoires définis par cette variable dans le cas contraire.

Note 1 : Dans cet exemple, nous avons inclus la ligne :

```
package monpackage;
```

au début du fichier **Test.java**. Cette instruction est obligatoire pour une raison que nous étudierons dans une prochaine section, mais qui n'a rien à voir, cette fois, avec l'emplacement du fichier résultant **Test.class**. D'ailleurs,

le programme fonctionne parfaitement sans que nous ayons à déplacer le fichier dans le répertoire **monpackage**.

Note 2 : Contrairement à ce qui se passait lorsque les deux classes étaient définies dans le même fichier, où l'ordre d'apparition des classes était indifférent, il faut ici compiler d'abord la classe **Employe** puis la classe **Test**. Si vous procédez en ordre inverse, vous risquez deux types de problèmes :

- S'il s'agit de la première compilation, vous obtiendrez un message d'erreur car le compilateur ne pourra trouver la classe **Employe** dont il a besoin.

- Si la classe **Employe** a déjà été compilée et placée dans le répertoire correct, le compilateur utilisera la version existante. Cela peut ne poser aucun problème si la classe **Test** n'a pas été modifiée. Dans le cas contraire, une erreur peut produire. Pour l'éviter, le compilateur recompile automatiquement toutes les classes référencées dans le ou les fichiers sources indiqués sur la ligne de commande et qui ont été modifiées depuis leur dernière compilation. (Il est possible de compiler plusieurs fichiers en même temps en plaçant leurs noms les uns à la suite des autres à la fin de la ligne de commande.) Si les classes recompilées de cette façon font elles-mêmes appel à d'autres classes, celles-ci ne sont pas vérifiées automatiquement. (La vérification n'est pas *récursive*.) En revanche, vous pouvez demander explicitement une vérification récursive de toutes les classes utilisées grâce à l'option de compilation **-depend** :

```
javac -depend Test.java
```

Si le fichier source d'une classe n'est pas trouvé, la vérification n'est pas effectuée. Le compilateur ne produit dans ce cas aucun message d'erreur ou d'avertissement.

Placement automatique des fichiers **.class** *dans les répertoires correspondant aux packages*

Plutôt que de déplacer manuellement les fichiers **.class**, vous pouvez demander au compilateur de le faire pour vous, grâce à l'option de compilation **-d *répertoire***. Lorsque cette option est utilisée, le compilateur place les

fichiers **.class** sans indication de package dans le répertoire indiqué, et crée automatiquement les sous-répertoires nécessaires pour stocker les classes appartenant à un package. Pour indiquer le répertoire courant, vous devez taper un point. Par exemple :

```
javac -depend -d . Test.java
```

compile le fichier **Test.java** ainsi que le fichier **Employe.java** si celui-ci porte une date de dernière modification postérieure à la date de création du fichier **Employe.class**, et place les fichiers **Test.class** et éventuellement **Employe.class** dans le sous-répertoire **monpackage** du répertoire courant. Si ce sous-répertoire n'existe pas, il est automatiquement créé.

Note 1 : L'ordre des options du compilateur n'a pas d'importance, mais les noms des fichiers à compiler doivent être placés en dernier.

Note 2 : Le compilateur cherche le fichier **Employe.class** dans le sous-répertoire **monpackage** afin de déterminer s'il est ou non à jour. Il ne tient pas compte d'un éventuel fichier **Employe.class** se trouvant ailleurs (dans le répertoire courant, par exemple).

Note 3 : Pour vérifier si les classes utilisées sont à jour, les fichiers sources doivent se trouver dans le même répertoire que les fichiers **.class**, ce qui réduit beaucoup l'intérêt de l'option **-d** puisque le compilateur place par défaut les fichiers **.class** dans le même répertoire que les fichiers sources.

Chemin d'accès par défaut pour les packages

Par défaut, c'est-à-dire en l'absence de toute définition de la variable **CLASSPATH**, deux chemins d'accès sont utilisés pour rechercher les packages :

- Le répertoire courant (pour toutes les classes).
- Le chemin d'accès des classes standard Java.

Il est important de noter que les classes sont recherchées en utilisant ces deux possibilités dans l'ordre ci-dessus. Il est donc possible d'intercepter

les appels à des classes Java en créant des classes de même nom. Par exemple, si vous créez une classe **System** contenant un objet **out** possédant une méthode **println()** et si vous placez cette classe dans le répertoire courant, l'instruction :

```
System.out.println("message");
```

fera référence à votre classe et non à la classe standard. En revanche, vous pourrez toujours faire référence à la classe standard en utilisant l'instruction :

```
java.lang.System.out.println("message");
```

(sauf si vous avez défini votre classe **System** dans un package **java.lang** que vous avez créé dans le répertoire courant, mais là, vous cherchez vraiment les ennuis !).

Si la variable d'environnement **CLASSPATH** est définie, Java utilise les chemins d'accès suivants pour trouver les packages :

- Le(s) répertoire(s) indiqué(s) dans la variable (pour toutes les classes).
- Le chemin d'accès des classes standard Java.

Il est donc là aussi possible d'intercepter les appels aux classes standard Java. Par ailleurs, notez que le répertoire courant ne figure plus implicitement dans la liste des chemins d'accès utilisés. Il est donc d'usage de commencer la définition de **CLASSPATH** par une référence au répertoire courant :

```
set classpath=.;<chemin d'accès>;<chemin d'accès>...
```

L'instruction import

Dans un programme, il peut être pénible de systématiquement faire référence à une classe en indiquant explicitement le package auquel elle appar-

tient. Ainsi, si votre programme crée une instance de la classe **Button**, vous devez utiliser la syntaxe :

```
java.awt.Button bouton = new java.awt.Button();
```

ce qui est long et fastidieux. Une autre façon de procéder consiste à rendre la classe **Button** disponible dans votre programme au moyen de l'instruction :

```
import java.awt.Button;
```

Cette instruction n'a pas pour effet de charger la classe **Button**, mais simplement de la rendre disponible de façon plus simple. Vous pouvez alors y faire référence sous la forme :

```
Button bouton = new Button();
```

Si vous voulez "importer" toutes les classes d'un package, vous pouvez utiliser la syntaxe :

```
import java.awt.*;
```

(Encore une fois, "importer" a ici un sens virtuel. Rien n'est réellement importé par cette instruction. Vous pouvez utiliser autant d'instructions **import** que vous le souhaitez sans diminuer en quoi que ce soit les performances de votre programme, ni augmenter sa taille.)

Ainsi, toutes les classes du package **java.awt** seront accessibles sans qu'il soit besoin de spécifier leur chemin d'accès. En revanche, vous ne pouvez pas importer plusieurs packages à la fois. L'instruction :

```
import java.awt.*;
```

fait référence à toutes les classes du package **java.awt** et non à tous les packages dont le nom commence par **java.awt**. Une confusion est toutefois possible car il existe onze packages dont le nom commence par **java.awt** parmi les packages standard de Java. Il est difficile de comprendre pourquoi les concepteurs de Java ont choisi d'entretenir cette confusion. Rappelons donc la signification des références suivantes :

```
java.awt.Button
```

Ici, le nom le plus à droite est un nom de classe, ce qui est reconnaissable au fait qu'il commence par une majuscule. (Ce n'est toutefois qu'une convention, et non une obligation.) Tout ce qui précède le nom de classe est donc un nom de package : il s'agit du package **java.awt**.

```
java.awt.event.ActionEvent
```

ActionEvent est un nom de classe. (Il commence par une majuscule.) Le nom du package est donc ici **java.awt.event**.

```
java.awt.*
```

L'astérisque ne peut représenter que des noms de classes. Ce qui précède est donc un nom de package. Il s'agit ici du package **java.awt**.

Il est parfaitement possible que deux packages comportent des classes de même nom. Dans ce cas, vous ne pouvez "importer" qu'une seule de ces classes. Si vous importez explicitement les deux classes, par exemple :

```
import monpackage.Test;
import ancienpackage.Test;
```

le compilateur produit un message d'erreur. En revanche, il est possible d'importer implicitement les deux classes. Dans ce cas, le résultat varie selon les conditions. Dans le cas le plus général :

```
import monpackage.*;
import ancienpackage.*;
```

le compilateur ne produit aucun message d'erreur tant que votre programme n'utilise pas une classe existant dans les deux packages. Dans le cas contraire, il produit un message d'erreur peu explicite :

```
Class Test not found in type declaration.
```

qui pourrait faire croire que la classe n'a pas été trouvée, alors que le problème vient au contraire de ce que le compilateur en a trouvé deux.

Dans le cas où une des deux importations est explicite, celle-ci l'emporte :

```
import monpackage.Test;
import ancienpackage.*;
```

Ici, la classe **Test** du package **monpackage** sera utilisée car elle est importée explicitement, contrairement à la classe **Test** du package **ancienpackage**, qui est importée implicitement à l'aide du caractère générique *****.

Attention : D'autres situations d'importation implicite peuvent se produire, causant des erreurs dont l'origine n'est pas toujours évidente. Ainsi, si le répertoire courant contient un fichier **Test.class**, nous nous trouvons en situation d'importation implicite de cette classe. Aussi, si notre programme commence par :

```
import monpackage.Test;
```

la classe **Test** du package **monpackage** est utilisée car l'importation explicite l'emporte. En revanche, si nous utilisons l'instruction :

```
import monpackage.*;
```

il s'agit d'une importation implicite. Devant les deux possibilités d'importation implicite (la classe **Test** du répertoire courant et celle du package **monpackage**), Java choisit le répertoire courant. Si les deux versions du fichier **Test.class** sont différentes, cela peut poser des problèmes. Le cas est fréquent si vous avez d'abord développé une version du programme dans laquelle les différentes classes étaient définies dans le même fichier. Le compilateur a donc créé un fichier **Test.class** dans le répertoire courant, mais vous n'y avez pas prêté attention, n'ayant jamais créé de fichier **Test.java**. Lorsque vous créez ensuite une version de la classe **Test** dans le package **monpackage**, vous placez le fichier source dans le répertoire **monpackage**, afin que Java puisse le recompiler à la demande lors de la compilation du programme principal. Cependant, Java trouve alors le fichier **Test.class** du répertoire courant. En l'absence d'un fichier **Test.java**, il ne peut déterminer si le fichier **.class** est à jour et l'utilise donc tel quel.

Une autre situation intrigante peut se produire si vous oubliez que Java recompile automatiquement les classes non à jour lorsqu'il en a la possibilité. Si le répertoire courant contient un fichier **Test.java** mais pas de fichier **Test.class**, et si ce fichier contient l'instruction :

```
package monpackage;
```

ce qui est somme toute assez normal, Java le recompile automatiquement. S'il se trouve que vous n'avez pas utilisé l'option **-d** du compilateur (puisque vous êtes en train de compiler le programme principal), le fichier **Test.class** est créé dans le répertoire courant. C'est donc lui qui est utilisé en priorité. Si le fichier **Test.java** n'était pas à jour, le fonctionnement du programme ne sera pas correct. Trouver l'origine de l'erreur ne sera pas toujours évident !

En conclusion, si votre programme ne semble pas fonctionner correctement, et si vous soupçonnez qu'une ancienne version d'une classe est utilisée, ou si vous obtenez un message d'erreur du type :

```
Error: File: xxxxx does not contain type yyyyy as
expected
```

ou :

```
Class zzzz not found un type declaration
```

vérifiez les problèmes d'importation implicite et explicite. En particulier, dans le premier message, **xxxxx** est probablement le nom du fichier (**.class** ou **.java**) non à jour se trouvant dans le répertoire courant.

Packages accessibles par défaut

Deux packages sont accessibles par défaut. Ce sont, par ordre de priorité :

- Le "package par défaut", qui contient toutes les classes pour lesquelles aucun package n'a été indiqué, et qui se trouve dans un des répertoires accessibles par défaut.
- Le package **java.lang**, qui contient un certain nombre de classes d'usage général comme **System**, que nous avons déjà utilisée fréquemment, ou **Math**, qui contient des méthodes permettant d'effectuer des calculs mathématiques.

Attention : Nous avons vu que la spécification d'un chemin d'accès pour les classes à l'aide de la variable d'environnement **CLASSPATH** rend le répertoire courant inaccessible par défaut (obligeant à ajouter explicitement une référence à ce répertoire dans la variable). En revanche, le chemin d'accès au package **java.lang**, ainsi qu'aux autres packages standard de Java n'est pas affecté par la définition de **CLASSPATH**.

Il est également possible d'indiquer un chemin d'accès pour les classes au moyen de l'option **-classpath** du compilateur. Mais, dans ce cas, tous les chemins d'accès par défaut sont supprimés, y compris ceux des packages standard. Vous devez donc indiquer explicitement en paramètre de l'option **-classpath** tous les chemins d'accès que vous souhaitez utiliser, plus le chemin d'accès au répertoire courant si nécessaire, plus le chemin d'accès aux packages standard (en les séparant à l'aide de points-virgules). Pour cela, il faut connaître le chemin d'accès aux packages standard, ce qui nécessite d'aborder le cas particulier des fichiers **.jar**.

Les fichiers .jar

Les fichiers **.jar** sont des fichiers compressés comme les fichiers **.zip** selon un algorithme particulier devenu un standard de fait dans le monde des PC. Ils sont parfois appelés *fichiers d'archives* ou, plus simplement, *archives*. Ces fichiers sont produits par des outils de compression tels que Pkzip (sous DOS) ou Winzip (sous Windows), ou encore par **jar.exe**. Les fichiers **.jar** peuvent contenir une multitude de fichiers compressés avec l'indication de leur chemin d'accès. Par exemple, si vous placez vos fichiers dans un répertoire **programmes** contenant les sous-répertoires **appli** et **util**, vous pouvez recréer la même arborescence à l'intérieur d'un fichier **.zip** ou **.jar**. Vous pouvez donc spécifier un chemin d'accès à un fichier compressé en indiquant le chemin d'accès au fichier **.jar** suivi du chemin d'accès au fichier compressé à l'intérieur de l'archive. Les packages standard de Java sont organisés de cette manière, dans un fichier nommé **rt.jar** placé dans le sous-répertoire **lib** du répertoire où est installé le JDK. Dans le cas d'une installation standard de Java 2 sur le disque C:, le chemin d'accès complet à la classe **System** est donc :

```
c:\jdk1.2\jre\lib\rt.jar\java\lang\System
```

(Ce chemin d'accès est théorique, car le système d'exploitation est incapable de l'utiliser.)

La référence à la classe **System** peut être limitée à :

```
java.lang.System
```

parce que Java connaît le chemin d'accès par défaut aux packages standard.

Création de vos propres fichiers .jar ou .zip

Vous pouvez, vous aussi, placer vos packages dans des fichiers **.jar** ou **.zip**. Le principal intérêt n'est pas de gagner de l'espace, mais d'améliorer la transportabilité (à ne pas confondre avec la portabilité !) de vos packages. Il est plus simple de n'avoir qu'un seul fichier à transporter ou à archiver. De

plus, il est beaucoup plus rare d'effacer, de déplacer ou de remplacer par erreur un fichier dans une archive que dans un répertoire. La maintenance des versions successives de votre package est ainsi facilitée.

Pour créer un fichier **.zip** incluant l'arborescence des répertoires à l'aide de Winzip, il suffit d'ouvrir cette application, et de faire glisser un à un les répertoires contenant vos packages dans la fenêtre affichée. Lorsque vous faites glisser le premier répertoire, Winzip affiche une fenêtre pour vous demander le nom du fichier **.zip** à créer. La boîte de dialogue contient également deux options importantes :

- **Recurse folders** permet d'inclure dans l'archive les sous-répertoires éventuels de celui que vous faites glisser.

- **Save extra folder info** permet d'ajouter les fichiers à l'archive en incluant l'arborescence des répertoires.

La première option doit être active. Dans le cas contraire, seuls les fichiers du premier niveau sont archivés. Pour la deuxième option, c'est un peu plus complexe. Cette option ajoute le chemin d'accès depuis la racine du disque dur. Si le répertoire contenant vos packages se trouve dans la racine du disque dur, elle doit donc être active. En revanche, si, comme c'est souvent le cas, il se trouve plus profond dans l'arborescence de votre disque, vous devez :

- Ne pas activer cette option.

- Archiver le niveau supérieur à celui de vos packages. En effet, le premier niveau d'arborescence n'est pas conservé.

Exemples :

1) Vous avez créé trois packages **package1**, **package2** et **package3** correspondant aux répertoires **c:\package1**, **c:\package2** et **c:\package3**. Vous devez alors activer l'option **Save extra folder info** et faire glisser les trois répertoires dans la fenêtre de Winzip.

2) Vous avez créé trois packages **package1**, **package2** et **package3** correspondant aux répertoires **c:\programmes\util\package1**, **c:\program-**

mes\util\package2 et **c:\programmes\util\package3**. Vous devez alors désactiver l'option **Save extra folder info** et faire glisser le répertoire **c:\programmes\util** dans la fenêtre de Winzip. Bien sûr, ce répertoire ne doit rien contenir d'autre que vos packages. (Bien que cela ne pose pas de problème que le fichier **.zip** contienne également les photos de la communion de votre petit-neveu.)

Note 1 : La procédure ci-dessus n'est valable que si vos packages sont **package1**, **package2** et **package3** (**c:\programmes\util** étant le chemin d'accès). Si vous avez créé les packages **util.package1**, **util.package2** et **util.package3**, c'est le répertoire **c:\programmes** qu'il faut faire glisser dans la fenêtre de Winzip (et qui ne doit rien contenir d'autre que vos packages).

Note 2 : L'archivage utilisant le format **.jar** sera décrit au Chapitre 20.

Comment nommer vos packages ?

Vous pouvez nommer vos packages comme vous le souhaitez tant que vous en êtes le seul utilisateur. En revanche, si vous concevez des packages pour d'autres programmeurs, vous devez choisir des noms en vous assurant qu'ils n'entreront jamais en conflit avec ceux des autres packages qu'ils seraient susceptibles d'utiliser. Si tous vos "clients" sont des membres de votre entreprise, vous pouvez peut-être contrôler les packages qu'ils utilisent ; mais si vous destinez votre production à une diffusion externe, il faut respecter une convention permettant d'assurer l'unicité des noms de packages. Cette convention consiste à faire commencer les noms de vos packages par votre nom de domaine Internet en commençant par la fin. Ainsi, si votre domaine Internet est **volga.fr**, tous vos packages auront un nom commençant par **fr.volga**. La suite du nom est libre. Si vous souhaitez personnaliser vos packages, il vous suffit d'ajouter le pseudonyme que vous utilisez pour votre adresse Email. Si votre adresse est **anatole@volga.fr**, tous vos packages commenceront par **fr.volga.anatole.** Comme vous êtes assuré que votre adresse Email est unique au monde, vous êtes sûr du même coup que vos packages pourront être utilisés partout sans conflit.

Note : Si vous n'avez pas de domaine Internet, vous pouvez utiliser tout de même cette convention. (Il suffit d'une adresse Email.) Cependant, il n'est

pas sûr que des noms de packages commençant par **fr.wanadoo.marcel.durand.** soient du meilleur effet ! (Si vos packages sont destinés à une diffusion publique, il n'est même pas certain que votre fournisseur d'accès Internet voie d'un très bon œil l'utilisation de son nom, mais normalement, vous devriez dans ce cas avoir votre propre domaine.)

Ce qui peut être fait (Quoi)

Nous avons maintenant fait le tour de la question *Où ?* Pour qu'une classe puisse être utilisée (directement ou par l'intermédiaire d'un de ses membres), il faut non seulement être capable de la trouver, mais aussi qu'elle soit adaptée à l'usage que l'on veut en faire.

Une classe peut servir à plusieurs choses :

- Créer des objets, en étant *instanciée*.
- Créer de nouvelles classes, en étant *étendue*.
- On peut utiliser directement ses membres statiques (sans qu'elle soit instanciée.)
- On peut utiliser les membres de ses instances.

(On peut trouver d'autres utilisations, mais nous nous contenterons de celles-là pour l'instant).

Les classes peuvent contenir des primitives, des objets, des classes, des méthodes, des constructeurs, et d'autres choses encore que nous n'avons pas étudiées, comme les *finaliseurs*.

Java permet d'imposer à tous ces éléments des restrictions d'utilisation. Il ne s'agit pas à proprement parler d'un problème d'accès, mais, pour l'utilisateur, le résultat est du même ordre. C'est pourquoi nous traitons cet aspect dans le chapitre consacré à l'accessibilité.

Java dispose de mots clés permettant de modifier la façon dont les éléments peuvent être utilisés.

static

Un élément déclaré **static** appartient à une classe et non à ses instances. Dans l'exemple cité au début de ce chapitre, la primitive **capacité** aurait été déclarée **static**. Les objets instanciés à partir d'une classe ne possèdent pas les éléments de cette classe qui ont été déclarés **static**. Un seul élément existe pour la classe et il est partagé par toutes les instances. Cependant, cela ne limite pas l'accessibilité, mais conditionne simplement le résultat obtenu lors des accès.

Les variables *static*

Si la classe **Voiture** contient la primitive statique **capacité** et la primitive non statique **carburant**, et si nous instancions cette classe pour créer l'objet **maVoiture**, cet objet possède sa propre variable **carburant**, à laquelle il est possible d'accéder grâce à la syntaxe :

```
maVoiture.carburant
```

(Nous verrons que ce type d'accès est le plus souvent rendu impossible par une modification des droits d'accès. Considérons pour l'instant qu'il n'existe aucune restriction de droit.)

L'objet **maVoiture** ne possède pas de variable **capacité**. Normalement, **capacité** appartient à la classe **Voiture**, et il est possible d'y accéder en utilisant la syntaxe :

```
Voiture.capacité
```

Cependant, Java nous permet également d'utiliser la syntaxe :

```
maVoiture.capacité
```

Il faut bien comprendre que les deux expressions font référence à la même variable. Ainsi le code suivant :

```
maVoiture.capacité = 120;
System.out.println(Voiture.capacité);
```

affichera 120, valeur qui vient d'être affectée à la variable statique en y accédant à l'aide du nom de l'instance. (Encore une fois, ce type de manipulation est normalement impossible dans un programme correctement écrit, l'impossibilité ne venant pas de Java, mais des droits d'accès.)

En revanche, il sera impossible d'utiliser la référence suivante :

```
Voiture.carburant
```

En effet, la quantité de carburant présente dans le réservoir est une caractéristique d'une instance et n'a aucun sens appliquée à la classe. Une tentative d'utilisation de cette référence produit le message d'erreur :

```
Can't make a static reference to nonstatic variable car-
burant in class Voiture
```

Si la classe **Voiture** n'a pas été instanciée, ses variables statiques ne sont évidemment accessibles qu'avec le nom de la classe. Il existe cependant un cas dans lequel il est possible de faire référence à une variable **static** sans utiliser le nom de la classe ni le nom d'une instance : lors de la définition même de la classe. (Aucun nom d'instance ne peut évidemment être employé à ce moment.) Considérez l'exemple suivant :

```
public class Automobile {
   public static void main(String[] argv) {
      Voiture maVoiture = new Voiture(80);
      System.out.println (Voiture.capacité);
   }
}
```

```
class Voiture {
   static int capacité;
   int carburant = 0;

   Voiture (int c) {
      capacité = c;
   }
}
```

Dans le constructeur de la classe **Voiture**, nous utilisons la référence **capacité** sans indiquer le nom de la classe.

Cet exemple peut paraître étrange car la variable statique **capacité** est susceptible d'être modifiée chaque fois qu'un objet est instancié. Cela ne pose aucun problème. Tel que l'exemple est écrit, elle peut même être modifiée à tout moment en lui affectant une valeur quelconque. Ainsi, si nous ajoutons la ligne :

```
public class Automobile {
   public static void main(String[] argv) {
      Voiture maVoiture = new Voiture(80);
      maVoiture.capacité = 90;
      System.out.println (Voiture.capacité);
```

le programme affichera 90 au lieu de 80. Ce qu'il faut comprendre, c'est que **capacité** sera modifiée même pour les instances créées avant, puisqu'il n'en existe qu'un seul exemplaire partagé par toutes les instances. (Cet exemple est un peu tiré par les cheveux, mais ce n'est qu'un exemple !)

Note : N'oubliez pas que Java initialise automatiquement les variables membres. Nous y reviendrons dans très peu de temps.

Les primitives ne sont pas les seuls éléments à pouvoir être déclarés statiques. Il en est exactement de même pour les objets et les méthodes.

Les méthodes *static*

Les méthodes peuvent également être statiques. Supposons que nous souhaitions écrire un *accesseur* pour la variable **capacité**. Nous pouvons le faire de la façon suivante :

```
public class Automobile {
   public static void main(String[] argv) {
      Voiture maVoiture = new Voiture(80);
      System.out.println (maVoiture.getCapacité());
   }
}

class Voiture {
   static int capacité;
   int carburant = 0;

   Voiture (int c) {
      capacité = c;
   }

   static int getCapacité() {
      return capacité;
   }
}
```

La méthode **getCapacité()** peut être déclarée **static** car elle ne fait référence qu'à des membres **static** (en l'occurrence la variable **capacité**). Ce n'est pas une obligation. Le programme fonctionne aussi si la méthode n'est pas déclarée **static**. Dans ce cas, cependant, la méthode est dupliquée chaque fois qu'une instance est créée, ce qui n'est pas très efficace.

Comme dans le cas des variables, les méthodes **static** peuvent être référencées à l'aide du nom de la classe ou du nom de l'instance. On peut utiliser le nom de la méthode seul, uniquement dans la définition de la classe.

Il est important de noter que les méthodes **static** ne peuvent en aucun cas faire référence aux méthodes ou aux variables non **static** de la classe. Elles ne peuvent non plus faire référence à une instance. (La référence **this** ne peut pas être employée dans une méthode **static**.)

Les méthodes **static** ne peuvent pas non plus être redéfinies dans les classes dérivées.

Initialisation des variables *static*

Nous savons déjà que les variables non **static** sont initialisées lorsqu'une instance est créée. Qu'en est-il des variables **static** ? Elles sont tout simplement initialisées lorsque c'est nécessaire, la première fois que la classe est chargée. Si nous reprenons l'exemple précédent en enlevant la ligne contenant la création de l'instance **maVoiture**, que se passe-t-il ?

```
public class Automobile {
   public static void main(String[] argv) {

      System.out.println (maVoiture.getCapacité());
   .
   .
   .
```

Le compilateur indique une erreur car l'appel de la méthode **getCapacité()** est réalisé à travers une instance qui n'existe plus. En revanche, il est toujours possible d'y faire référence au moyen du nom de la classe :

```
      System.out.println (Voiture.getCapacité());
```

Dans ce cas, la méthode **getCapacité()** retourne 0. En effet, la variable **static capacité** est initialisée dès la première référence à la classe. La définition de la classe ne comporte pas d'initialisation de la variable. Java l'initialise automatiquement à 0 car il s'agit d'une variable membre de la classe.

Comment faire si nous voulons initialiser nous-mêmes la variable ? Pour l'initialiser à l'aide d'une valeur littérale, il suffit de le faire au moment de la déclaration :

```
static int capacité = 80;
```

En revanche, si l'initialisation est plus complexe, nous pouvons la mener à bien dans le constructeur, mais cela présente deux inconvénients :

- L'initialisation n'aura pas lieu avant qu'une instance soit créée.
- Elle aura lieu de nouveau chaque fois qu'une instance sera créée.

Pour pallier ces inconvénients, nous devons nous souvenir des *initialiseurs d'instances* que nous avons décrits au Chapitre 5. Les initialiseurs d'instances sont des blocs de codes qui sont exécutés chaque fois qu'une instance est créée. Leur utilité ne paraissait pas évidente alors, étant limitée aux classes dépourvues de constructeurs (les classes anonymes, que nous étudierons dans un prochain chapitre) et à certains cas particuliers. Tout comme il existe des initialiseurs d'instances, nous disposons également d'*initialiseurs statiques*.

Les initialiseurs statiques

Un initialiseur statique est semblable à un initialiseur d'instance, mais il est précédé du mot clé **static**. Par exemple :

```
public class Automobile {
   public static void main(String[] argv) {
      System.out.println (Voiture.getCapacité());
      Voiture maVoiture = new Voiture();
      System.out.println (Voiture.getCapacité());
   }
}

class Voiture {
   static int capacité;
   static {
```

```
            capacité = 80;
         System.out.println(
            "La variable statique capacité vient d'être initialisée");
      }
      int carburant = 0;

      Voiture () {
      }

      static int getCapacité() {
         return capacité;
      }
   }
```

Si vous compilez et exécutez ce programme, vous obtiendrez l'affichage suivant :

```
La variable statique capacité vient d'être initialisée
80
80
```

En revanche, si vous supprimez la ligne imprimée en italique, le programme affiche :

```
La variable statique capacité vient d'être initialisée
80
```

L'initialiseur statique est exécuté au premier chargement de la classe, que ce soit pour utiliser un membre statique ou pour l'instancier.

Note 1 : Les membres statiques (ici la variable **capacité**) doivent être déclarés avant l'initialiseur et non dedans car, sinon, leur portée serait limitée à l'initialiseur lui-même.

Note 2 : Vous pouvez placer autant d'initialiseurs statiques que vous le souhaitez, à n'importe quels endroits de la définition de la classe. Ils seront

tous exécutés au premier chargement de celle-ci, dans l'ordre dans lequel ils apparaissent.

final

Nous venons d'étudier les restrictions qui pouvaient être apportées à l'utilisation d'un élément en fonction de son appartenance à une instance ou à une classe. D'autres caractéristiques peuvent influencer les conditions d'accès. De nombreux langages de programmation, par exemple, font la différence entre les données dont la valeur peut être modifiée (les variables) et celles dont la valeur est fixe (les constantes). Java ne dispose pas de constantes. Ce n'est pas une limitation, car Java dispose d'un mécanisme beaucoup plus puissant qui permet non seulement d'utiliser une forme de constantes, mais également d'appliquer ce concept à d'autres éléments comme les méthodes ou les classes.

Les variables *final*

Une variable déclarée **final** ne peut plus voir sa valeur modifiée. Elle remplit alors le rôle d'une constante dans d'autres langages. Une variable **final** est le plus souvent utilisée pour encoder des valeurs constantes. Par exemple, si vous voulez utiliser dans votre programme la valeur de Pi, il est peu probable que vous soyez amené à modifier cette valeur. Vous pouvez donc l'attribuer à une variable déclarée **final** :

```
final float pi = 3.14;
```

(Ce n'est qu'un exemple. Java met à votre disposition une valeur de Pi beaucoup plus précise que cela !)

Déclarer une variable **final** offre deux avantages. Le premier concerne la sécurité. En effet, le compilateur refusera toute affectation ultérieure d'une valeur à la variable. De cette façon, toute tentative de modification se traduira par un message d'erreur :

```
Can't assign a value to a final variable.
```

De cette façon, si votre programme tente de modifier la valeur de Pi, l'erreur sera détectée à la compilation.

Le deuxième avantage concerne l'optimisation du programme. Sachant que la valeur en question ne sera jamais modifiée, le compilateur est à même de produire un code plus efficace. En particulier, il peut effectuer lors de la compilation certains calculs qui, avec des variables non statiques, devraient être effectués au moment de l'exécution du programme. Ainsi, lorsque le compilateur rencontre la ligne :

```
int x = a + 2;
```

où **a** est une variable normale, il ne peut effectuer le calcul et doit donc remplacer cette ligne de code source par le code objet équivalent. L'opération sera donc effectuée au moment de l'exécution du programme. En revanche, si **a** est une variable **final**, le compilateur sait que sa valeur ne sera jamais modifiée. Il peut donc effectuer immédiatement le calcul et affecter la valeur correcte à **x**.

Les variables *final* non initialisées

Il existe en fait deux types de constantes dans un programme : celles dont la valeur est connue avant l'exécution du programme, et celles dont la valeur n'est connue qu'au moment de l'exécution. Ces deux types se prêtent à des optimisations différentes. En effet, le calcul au moment de la compilation n'est possible que si la valeur des constantes est connue. Un autre type d'optimisation consiste, par exemple, à stocker la valeur dans une zone de la mémoire où elle pourra être lue plus rapidement, mais où, en revanche, sa modification prendrait plus de temps. Ce type d'optimisation est tout à fait possible avec des constantes dont la valeur n'est connue qu'au moment de l'exécution. Java permet de simuler ce type de constantes en utilisant des variables **final** non initialisées. Ce type de variable peut être initialisé ultérieurement. Une fois l'initialisation effectuée, la valeur ne peut plus être modifiée. Une variable **final** non initialisée est simplement déclarée de la façon suivante :

```
final int a;
```

Évidemment, cette variable ne pourra pas être utilisée avant d'être initialisée. Le cas le plus fréquent d'utilisation de ce type de variable est celui des paramètres de méthodes. Ceux-ci peuvent être déclarés **final** afin que le compilateur soit en mesure d'optimiser le code. Pour autant, leur valeur ne peut pas être connue au moment de la compilation. Vous pouvez cependant parfaitement écrire :

```
int calcul (final int i, final int j) {
   corps de la méthode...
}
```

Les variables **i** et **j** ne seront initialisées que lors de l'appel de la méthode. Leur valeur ne pourra pas être modifiée à l'intérieur de celle-ci.

Les méthodes *final*

Les méthodes peuvent également être déclarées **final**, ce qui restreint leur accès d'une tout autre façon. En effet, les méthodes **final** ne peuvent pas être redéfinies dans les classes dérivées. Vous devez donc utiliser ce mot clé lorsque vous voulez être sûr qu'une méthode d'instance aura bien le fonctionnement déterminé dans la classe parente. (S'il s'agit d'une méthode **static**, il n'est pas nécessaire de la déclarer **final** car les méthodes **static** ne peuvent jamais être redéfinies.)

Les méthodes **final** permettent également au compilateur d'effectuer certaines optimisations qui accélèrent l'exécution du code. Pour déclarer une méthode **final**, il suffit de placer ce mot clé dans sa déclaration, de la façon suivante :

```
final int calcul (int i, in j) {
```

Notez que le fait que la méthode soit déclarée **final** n'a rien à voir avec le fait que ses arguments le soient ou non.

L'intérêt des méthodes **final** est lié à une caractéristique des objets Java que nous n'avons pas encore étudiée : le polymorphisme. Les objets Java peu-

vent être manipulés en tant que ce qu'ils sont (une instance d'une classe) ou comme s'ils étaient une instance d'une classe parente. Ainsi, dans l'exemple suivant :

```
public class Animaux3 {
   public static void main(String[] argv) {
   Animal milou = new Chien();
   Animal titi = new Canari();
   milou.crie();
   titi.crie();
   }
}
class Animal {
   void crie() {
   }
}
class Canari extends Animal {
   void crie() {
      System.out.println("Cui-cui !");
   }
}
class Chien extends Animal {
   void crie() {
      System.out.println("Ouah-Ouah !");
   }
}
```

les deux objets **milou** et **titi** sont créés en tant qu'objets de type **Animal**. Lorsque nous utilisons la syntaxe **milou.crie()**, Java est obligé d'effectuer une recherche dynamique (c'est-à-dire au moment de l'exécution du programme) pour savoir quelle version de la méthode doit être utilisée. Si la version du type correspondant au type de l'objet était utilisée, le programme n'afficherait rien, puisque cette version ne fait rien. Au contraire, Java trouve que l'objet de type **Animal** référencé par le handle **milou** est en réalité une instance de **Chien**, et la version correspondante de la méthode est exécutée. Cette recherche dynamique est effectuée systématiquement pour tous les appels de méthode. L'utilisation d'une méthode **final** indique à Java qu'il

n'est pas nécessaire d'effectuer une recherche dynamique, puisque la méthode ne peut pas avoir été redéfinie. L'exécution du programme est donc optimisée. (Nous reviendrons dans un prochain chapitre sur les différents aspects du polymorphisme.)

Les classes *final*

Une classe peut également être déclarée **final**, dans un but de sécurité ou d'optimisation. Une classe **final** ne peut pas être étendue pour créer des sous-classes. Par conséquent, ses méthodes ne peuvent pas être redéfinies et leur accès peut donc se faire directement, sans recherche dynamique.

Une classe **final** ne peut être clonée. Le clonage est une technique de création d'objets qui sera étudiée dans un prochain chapitre.

synchronized

Le mot clé **synchronized** modifie également les conditions d'accès aux classes et aux objets. Il concerne uniquement les programmes utilisant plusieurs *threads*, c'est-à-dire plusieurs processus se déroulant simultanément. Java permet de réaliser facilement ce genre de programme. Par exemple, vous pouvez concevoir un programme dans lequel des objets d'un tableau sont modifiés fréquemment. De temps en temps, le tableau doit être trié pour améliorer les conditions d'accès. Cependant, les objets ne doivent pas être modifiés pendant que le tri s'exécute. Il faut donc s'assurer que la méthode permettant de modifier un élément du tableau ne pourra pas accéder à celui-ci lorsque le tri est actif. Avec un langage traditionnel, nous serions obligés d'interdire tout accès au tableau pendant le tri. Java permet d'être plus sélectif. Une des façons de procéder consiste à utiliser pour la modification des éléments du tableau une méthode déclarée **synchronized**. De cette façon, la méthode ne pourra être exécutée que si elle a pu s'assurer l'exclusivité de l'utilisation de l'objet. L'objet sera donc verrouillé pendant la modification d'un élément. Cette approche est plus efficace pour l'utilisateur que le verrouillage pendant le tri car le tri peut être une opération longue. De cette façon, les éléments peuvent toujours être consultés pendant cette opération. Pour déclarer une méthode **synchronized**, il suffit de faire précéder sa déclaration du mot clé :

```
synchronized void trier () {
```

Une classe peut également être déclarée **synchronized**. Dans ce cas, toutes ses méthodes le sont. Les méthodes d'instances ne peuvent s'exécuter que si l'objet auquel elles appartiennent a pu être verrouillé. Les méthodes statiques ne sont exécutables qu'après verrouillage de la classe.

Il est également possible de verrouiller un objet de l'"extérieur", à l'aide de la syntaxe suivante :

```
synchronized (objet) {bloc d'instructions}
```

objet est un handle d'objet ou toute expression dont le résultat de l'évaluation est un objet. Le bloc d'instructions n'est exécuté que si l'objet a pu être verrouillé. Pendant toute la durée de l'exécution du bloc d'instructions, l'objet est inaccessible pour les autres *threads* du programme.

Note : Si le bloc est réduit à une seule instruction, les accolades ne sont pas nécessaires.

(Les *threads* seront étudiés en détail dans un prochain chapitre.)

native

Une méthode peut également être déclarée **native**, ce qui a des conséquences importantes sur la façon de l'utiliser. En effet, une méthode **native** n'est pas écrite en Java, mais dans un autre langage. Les méthodes **native** ne sont donc pas portables d'un environnement à un autre. Les méthodes **native** n'ont pas de définition. Leur déclaration doit être suivie d'un point-virgule. Nous n'étudierons pas les méthodes **native** dans ce livre.

transient

Le mot clé **transient** s'applique aux variables d'instances (primitives et objets). Il indique que la variable correspondante est *transitoire* et que sa valeur ne doit pas être conservée lors des opérations de *sérialisation*. La

sérialisation sera étudiée dans un prochain chapitre. Sachez simplement pour l'instant que cette opération permet d'enregistrer sur disque un objet Java afin de le conserver pour une session ultérieure, ou de l'envoyer à travers un réseau. Lors de la sérialisation, tous les champs de l'objet sont sauvegardés à l'exception de ceux déclarés **transient**.

volatile

Le mot clé **volatile** s'applique aux variables pour indiquer qu'elles ne doivent pas être l'objet d'optimisation. En effet, le compilateur effectue certaines manipulations pour accélérer le traitement. Par exemple, certaines variables peuvent être recopiées dans une zone de mémoire dont l'accès est plus rapide. Si, pendant ce temps, un autre processus modifie la valeur de la variable, la version utilisée risque de ne pas être à jour. Il est donc possible d'empêcher cette optimisation pas mesure de sécurité. Evidemment, cela ne concerne que les programmes utilisant plusieurs processus simultanés. Nous y reviendrons dans un prochain chapitre.

abstract

Le mot clé **abstract** peut être employé pour qualifier une classe ou une méthode. Une méthode déclarée **abstract** ne peut pas être exécutée. En fait, elle n'a pas d'existence réelle. Sa déclaration indique simplement que les classes dérivées doivent la redéfinir. Ainsi, dans l'exemple **Animaux3** des pages précédentes, la méthode **crie()** de la classe **Animal** aurait pu être déclarée **abstract** :

```
abstract class Animal {
   abstract void crie();
}
```

ce qui signifie que "tout animal doit être capable de crier", mais que "le cri d'un animal est une notion abstraite". Qui, en effet, peut dire quel est le cri d'un animal ? La question n'a pas de sens. Nous savons simplement qu'un animal a un cri. C'est exactement la signification du mot clé **abstract**. La

méthode ainsi définie indique qu'une sous-classe devra définir la méthode de façon concrète.

Les méthodes **abstract** présentent les particularités suivantes :

- Une classe qui contient une méthode **abstract** doit être déclarée **abstract** elle-même.
- Une classe **abstract** ne peut pas être instanciée.
- Une classe peut être déclarée **abstract**, même si elle ne comporte pas de méthodes **abstract**.
- Pour pouvoir être instanciée, une sous-classe d'une classe **abstract** doit redéfinir toutes les méthodes **abstract** de la classe parente.
- Si une des méthodes n'est pas redéfinie de façon concrète, la sous-classe est elle-même **abstract** et doit être déclarée explicitement comme telle.
- Les méthodes **abstract** n'ont pas de définition. Leur déclaration doit être suivie d'un point-virgule.

Notre programme précédent réécrit en utilisant une classe **abstract** devient donc :

```
public class Animaux3 {
   public static void main(String[] argv) {
   Animal milou = new Chien();
   Animal titi = new Canari();
   milou.crie();
   titi.crie();
   }
}

abstract class Animal {
   abstract void crie();
}
```

```
class Canari extends Animal {
   void crie() {
      System.out.println("Cui-cui !");
   }
}

class Chien extends Animal {
   void crie() {
      System.out.println("Ouah-Ouah !");
   }
}
```

De cette façon, il n'est plus possible de créer un animal en instanciant la classe **Animal**. En revanche, grâce à la définition de la méthode **abstract crie()** dans la classe **Animal**, il est possible de faire crier un animal sans savoir s'il s'agit d'un **Chien** ou d'un **Canari**.

Note : Vous vous demandez peut-être quelle est l'utilité de tout cela. En effet, il aurait été plus simple de définir **milou** de type **Chien** et **titi** de type **Canari**. Vous avez en partie raison, mais nous avons voulu simuler ici certains traitements dépendant de l'utilisation de structures que nous n'avons pas encore étudiées. Vous verrez dans un prochain chapitre que Java traite parfois les objets comme des instances des classes parentes.

Les interfaces

Une classe peut contenir des méthodes **abstract** et des méthodes non **abstract**. Cependant, il existe une catégorie particulière de classes qui ne contient que des méthodes **abstract**. Il s'agit des interfaces. Les interfaces sont toujours **abstract**, sans qu'il soit nécessaire de l'indiquer explicitement. De la même façon, il n'est pas nécessaire de déclarer leurs méthodes **abstract**.

Les interfaces obéissent par ailleurs à certaines règles supplémentaires. Elles ne peuvent contenir que des variables **static** et **final**. Elles peuvent être étendues comme les autres classes, avec une différence majeure : une interface peut dériver de plusieurs autres interfaces. En revanche, une classe ne

peut pas dériver uniquement d'une ou de plusieurs interfaces. Une classe dérive toujours d'une autre classe et peut dériver, en plus, d'une ou de plusieurs interfaces. Les interfaces seront étudiées en détail au chapitre suivant.

Qui peut le faire (Qui)

Maintenant que nous avons étudié tous les éléments qui permettent de contrôler l'emplacement (*Où*) des éléments et ce qui peut être fait avec (*Quoi*), nous allons nous intéresser aux autorisations d'accès (*Qui*).

En Java, il existe quatre catégories d'autorisations d'accès, spécifiées par les modificateurs suivants :

- **private**
- **protected**
- **public**

La quatrième catégorie correspond à l'absence de modificateur. C'est donc l'autorisation par défaut, que nous appellerons ***package*** (en italique pour indiquer qu'il ne s'agit pas d'un vrai mot clé).

Note : Dans certains livres, vous trouverez cette catégorie sous le nom de ***friendly***, par référence au langage C++.

public

Les classes, les interfaces, les variables (primitives ou objets) et les méthodes peuvent être déclarées **public**. Les éléments **public** peuvent être utilisés par n'importe qui sans restriction. (Du moins sans restriction d'autorisation, car les restrictions d'usage étudiées dans les sections précédentes s'appliquent.)

Certains éléments doivent impérativement être **public**. Ainsi, la classe contenant la méthode **main()** (votre programme principal) doit être **public**.

Les méthodes d'une interface doivent, lorsqu'elles sont redéfinies de manière concrète, être déclarées **public**.

Note : Un fichier contenant un programme Java ne peut contenir qu'une seule définition de classe déclarée **public** De plus, le fichier doit porter le même nom que la classe, avec l'extension **.java**.

Les accesseurs et les mutateurs sont le plus souvent déclarés **public** alors que l'accès direct aux variables est plus restreint. Cela permet de contrôler la façon dont les variables sont modifiées.

protected

Les membres d'une classe peuvent être déclarés **protected**. Dans ce cas, l'accès en est réservé aux méthodes des classes appartenant au même package, aux classes dérivées de ces classes, ainsi qu'aux classes appartenant aux mêmes packages que les classes dérivées. Cette autorisation s'applique uniquement aux membres de classes, c'est-à-dire aux variables (primitives et objets), aux méthodes et aux classes internes. (Les classes internes seront étudiées dans un prochain chapitre.) Les classes qui ne sont pas membres d'une autre classe ne peuvent pas être déclarées **protected**.

package

L'autorisation par défaut, que nous appelons ***package***, s'applique aux classes, interfaces, variables et méthodes. Les éléments qui disposent de cette autorisation sont accessibles à toutes les méthodes des classes du même package. Les classes dérivées ne peuvent donc y accéder que si elles ont été explicitement déclarées dans le même package.

Note : Rappelons que les classes n'appartenant pas explicitement à un package appartiennent automatiquement au package par défaut. Toute classe sans indication de package dispose donc de l'autorisation d'accès à toutes les classes se trouvant dans le même cas.

private

L'autorisation **private** est la plus restrictive. Elle s'applique aux membres d'une classe (variables, méthodes et classes internes). Les éléments déclarés **private** ne sont accessibles que depuis la classe qui les contient. Ce type d'autorisation est souvent employé pour les variables qui ne doivent être modifiées ou lues qu'à l'aide d'un *mutateur* ou d'un *accesseur*. Les mutateurs et les accesseurs, de leur côté, sont déclarés **public** afin que tout le monde puisse utiliser la classe. Ainsi, l'exemple décrit au début de ce chapitre peut s'écrire de la façon suivante :

```
public class Automobile {
   public static void main(String[] argv) {
      Voiture maVoiture = new Voiture();
      maVoiture.setCarburant(30);
      Voiture taVoiture = new Voiture();
      taVoiture.setCarburant(90);
   }
}

class Voiture {
   private static int capacité = 80;
   private int carburant = 0;

   Voiture () {
   }

   public static int getCapacité() {
      return capacité;
   }
   public void setCarburant(final int c) {
      final int maxi = capacité - carburant;
      if (c <= maxi) {
         carburant += c;
         System.out.println(
                 "Le remplissage a été effectué sans problème.");
      }
```

```
      else {
         carburant = capacité;
         System.out.println(
                 (c - maxi) + " litre(s) de carburant ont débordé.");
      }
   }
   public int getCarburant() {
      return carburant;
   }
}
```

De cette façon, tout le monde peut faire le plein ou consulter la jauge d'essence, mais vous êtes assuré que le réservoir ne contiendra jamais plus que sa capacité. (Pour être complet, il faudrait traiter les autres cas possibles d'erreur, comme l'utilisation de valeurs négatives.)

Autorisations d'accès aux constructeurs

Les constructeurs peuvent également être affectés d'une autorisation d'accès. Un usage fréquent de cette possibilité consiste, comme pour les variables, à contrôler leur utilisation. Supposons que vous écriviez un programme de simulation d'un élevage de lapins. Il pourrait s'articuler autour de la classe suivante :

```
public class Elevage {
   public static void main(String[] argv) {
   Lapin lapin1 = Lapin.créerLapin();
   }
}

class Lapin {
   private static int nombre = 0;
   private static int maxi = 50;
   private boolean vivant = true;

   private Lapin() {
      System.out.println("Un lapin vient de naître.");
```

```
        System.out.println("Vous possédez " + (++nombre) + " lapin(s).");
    }

    public void passeALaCasserole() {
        if (vivant) {
            vivant = false;
            System.out.println("Un lapin vient de passer à la casserole.");
            System.out.println("Vous possédez " + (--nombre) + " lapin(s).");
        }
        else {
            System.out.println("Ce lapin a déjà été mangé !");
        }
    }

    public static Lapin créerLapin() {
        if (nombre < maxi) {
            return new Lapin();
        }
        else {
            System.out.println("Elevage surpeuplé. Naissance impossible.");
            return null;
        }
    }
}
```

Le constructeur étant déclaré **private**, il est impossible de créer un nouveau lapin, sauf depuis la classe **Lapin** elle-même. C'est ce que fait la méthode **public créerLapin()**, après avoir vérifié que le nombre de lapins permet une nouvelle naissance. La méthode retourne alors un objet instance de la classe **Lapin** créé à l'aide de l'opérateur **new**. Dans le cas contraire, la méthode retourne **null**.

Attention : Ce programme n'est pas une illustration de la façon dont le problème devrait être traité. Il sert juste à montrer comment on peut tirer parti d'un constructeur privé pour effectuer certains traitements *avant* la création d'un objet afin, par exemple, de déterminer si cet objet doit être créé ou non. En effet, il est toujours possible d'effectuer des traitements

avant que le constructeur soit appelé, au moyen d'un initialiseur. Cependant, lorsque ces traitements sont effectués, la création de l'objet est déjà commencée. Ici, du fait de l'utilisation d'une méthode **static**, les traitements nécessaires peuvent être effectués avant que la création ne soit entamée. Il est donc possible de décider si l'objet doit être créé ou non.

Cette technique est également employée pour créer des banques d'objets. Une banque d'objet (*object pool* en anglais) permet d'optimiser certains traitements. Cette technique consiste à conserver les objets qui ne sont plus utilisés pour les affecter de nouveau lorsque cela est demandé, au lieu de créer de nouvelles instances. Le fonctionnement s'établit de la façon suivante :

- Un élément demande une instance d'un objet en faisant appel à une méthode du même type que la méthode **créerLapin()** de l'exemple ci-dessus.
- Le programme cherche si un objet est disponible.
- Si un objet est trouvé, il est réinitialisé, et un handle vers cet objet est retourné.
- Si aucun objet n'est disponible, un nouvel objet est créé et un handle vers cet objet est retourné.

Idéalement, la réinitialisation peut être effectuée lorsque les objets sont libérés, afin qu'ils soient plus rapidement disponibles. De plus, un processus peut contrôler en permanence le nombre d'objets libres, pour en supprimer s'il y en a trop, ou en créer quelques-uns d'avance si le stock est trop bas.

Une version simplifiée de cette technique est employée pour s'assurer qu'un objet n'existe qu'à un seul exemplaire.

```
public class DemoUnique {
   public static void main(String[] args) {
      Unique s1 = Unique.créer(5);
      System.out.println(s1.getValeur());
```

```
      Unique s2 = Unique.créer(10);
      System.out.println(s1.getValeur());
   }
}

final class Unique {
   private int i;
   static Unique s0 = new  Unique();
   private Unique() {
   }
   public static Unique créer(final int x) {
      s0.setValeur(x);
      return s0;
   }
   public int getValeur() {
      return i;
   }
   public void setValeur(final int x) {
      i = x;
   }
}
```

Dans ce programme, un objet de type **Unique** est créé la première fois que la classe est utilisée. Lors des utilisations suivantes, le même objet est réinitialisé et renvoyé comme s'il s'agissait d'un objet neuf. Le constructeur ne fait rien, mais il est indispensable. En effet, s'il n'existait pas, il ne pourrait pas être déclaré **private**. Une instance pourrait alors être créée à l'aide de l'opérateur **new**. Si vous exécutez ce programme, vous constaterez que les handles **s1** et **s2** pointent en fait vers le même objet.

Pour évaluer le gain en performances, vous pouvez compiler et exécuter les deux versions suivantes :

```
import java.util.*;
public class DemoNonUnique{
   public static void main(String[] args) {
      long t = System.currentTimeMillis();
```

```
      for (int i = 0; i < 10000000; i++) {
         NonUnique s1 = new NonUnique(i);
      }
      t = System.currentTimeMillis() - t;
      System.out.println(t);
      Runtime rt = Runtime.getRuntime();
      System.out.println(rt.totalMemory());
      System.out.println(rt.freeMemory());   }
}

final class NonUnique {
   private int i;
   NonUnique(final int x) {
      i = x;
   }
   public int getValeur() {
      return i;
   }
   public void setValeur(final int x) {
      i = x;
   }
}
```

et

```
import java.util.*;
public class DemoUnique{
   public static void main(String[] args) {
      long t = System.currentTimeMillis();
      for (int i = 0; i < 10000000; i++) {
          Unique s1 =  Unique.créer(i);
      }
      t = System.currentTimeMillis() - t;
      System.out.println(t);
      Runtime rt = Runtime.getRuntime();
      System.out.println(rt.totalMemory());
      System.out.println(rt.freeMemory());   }
}
```

```
final class Unique {
   private int i;
   static Unique s0 = new Unique();
   private Unique() {
   }
   public static Unique créer(final int x) {
      s0.setValeur(x);
      return s0;
   }
   public int getValeur() {
      return i;
   }
   public void setValeur(final int x) {
      i = x;
   }
}
```

La première fonctionne "normalement" et crée dix millions d'objets de type **Non Unique**. La deuxième version ne crée qu'un seul objet et l'initialise dix millions de fois. Le premier programme s'exécute en 13,680 secondes et le deuxième en 7,850 secondes, soit un gain de 42 %. En ce qui concerne la mémoire nécessaire, la deuxième version fait apparaître un gain de 56 %.

Note : Ces deux programmes utilisent une instance de la classe **java.util.Runtime** pour obtenir des informations concernant la mémoire disponible. Pour plus de renseignements sur les méthodes disponibles dans cette classe, reportez-vous à la documentation en ligne de Java.

Résumé

Vous connaissez maintenant les principaux modificateurs qui peuvent être utilisés avec les classes, les méthodes et les variables Java. Nous allons pouvoir aborder la partie intéressante de notre étude. Le prochain chapitre sera consacré au *polymorphisme,* qui est un des aspects les plus fascinants de Java.

Chapitre 9

Le polymorphisme

Nous avons étudié jusqu'ici des aspects de Java assez conventionnels. Bien sûr, la faculté de créer de nouvelles classes en étendant les classes existantes est un mécanisme puissant, mais en fin de compte assez trivial. Il ne fait que reproduire la façon dont nous structurons notre environnement. Cependant, si nous y réfléchissons un peu, nous nous apercevrons rapidement qu'il n'existe pas une seule façon de structurer l'environnement. Chaque sujet structure son environnement en fonction du problème qu'il a à résoudre. Pour un enfant de 8 ans, un vélo est un objet du même type qu'un ours en peluche ou un ballon de football. Exprimé en Java, nous dirions que c'est une instance de la classe **Jouet**. Pour un adulte, un vélo pourra être un objet du même type qu'une voiture ou une moto, c'est-à-dire une instance de la classe **Véhicule**. Pour un industriel fabriquant également des brouettes, des échelles ou des étagères métalliques, un vélo sera simplement une instance de la classe **Produit**.

Cependant, une structure différente n'implique pas forcément un sujet différent. Le programmeur pourra être amené à traiter des problèmes de produits, de véhicules ou de jouets, et devra donc utiliser une structure différente pour chaque problème. Il s'apercevra rapidement qu'il aura intérêt à s'affranchir de sa propre façon de structurer son environnement pour s'attacher à celle correspondant le mieux au problème à traiter. Pour autant, il constatera bientôt qu'au sein d'un même problème, il peut exister simultanément plusieurs structures. Il ne s'agit pas alors de choisir la structure la plus adaptée, mais de concevoir le programme de telle sorte que ces structures coexistent de manière efficace.

Nous avons certainement tous appris à l'école qu'il était impossible d'additionner des pommes et des oranges. Cette affirmation dénote une étroitesse d'esprit qui choque généralement ceux à qui elle est assénée, jusqu'à ce qu'ils finissent par se faire une raison et acceptent l'idée qu'un problème ne peut être traité qu'à travers une structure unique rigide. Pourtant, un enfant de cinq ans est à même de comprendre que :

```
3 pommes + 5 oranges = 8 fruits
```

Bien sûr, cette façon d'écrire est un peu cavalière. Nous préciserons bientôt dans quelles conditions et par quel mécanisme ce type de manipulation peut être mis en œuvre en Java.

Un autre problème peut surgir lorsqu'un objet paraît pouvoir appartenir à une structure ou une autre suivant l'angle sous lequel on aborde le problème. Par exemple, il paraît naturel de définir une classe **Cheval** comme dérivant de la classe **Animaux**. Cependant, il peut exister également une classe **MoyenDeTransport**, qui serait éventuellement également candidate. Un objet peut-il être à la fois un animal et un moyen de transport ? Le bon sens nous indique que oui, tout comme nous savons qu'un objet peut être à la fois une pomme et un fruit. Pour autant, la relation n'est pas la même dans les deux cas. Dans ce chapitre, nous étudierons les différents aspects de cette possibilité qu'ont les objets d'appartenir à plusieurs catégories à la fois, et que l'on nomme *polymorphisme*.

Le sur-casting des objets

Une façon de décrire l'exemple consistant à additionner des pommes et des oranges serait d'imaginer que nous disons pommes et oranges mais que nous manipulons en fait des fruits. Nous pourrions écrire alors la formule correcte :

```
  3 fruits (qui se trouvent être des pommes)
+ 5 fruits (qui se trouvent être des oranges)
-------------------------------------------
= 8 fruits
```

Cette façon de voir les choses implique que les pommes et les oranges soient "transformés" en fruits préalablement à l'établissement du problème. Cette transformation est appelée *sur-casting*, bien que cela n'ait rien à voir avec le sur-casting des primitives. Avec la syntaxe de Java, nous écririons quelque chose comme :

```
  3 (fruits)pommes
+ 5 (fruits)oranges
-------------------------------------------
= 8 fruits
```

Une autre façon de voir les choses consiste à poser le problème de la façon suivante :

```
  3 pommes
+ 5 oranges
-------------------------------------------
= ?
```

puis à laisser l'interpréteur se poser la question : "Peut-on effectuer l'opération ainsi ?" La réponse étant non, celui-ci devrait effectuer automatiquement le sur-casting nécessaire pour produire le résultat.

Ici, les deux approches semblent fonctionner de manière identique. Cependant, dans un programme, on ne sait pas toujours à l'avance quel type d'objets on va rencontrer. La première approche oblige à effectuer le sur-casting *avant* que l'opération soit posée. Avec la deuxième approche, le sur-casting peut être effectué à la demande. C'est exactement ce qui se passe en Java. Un objet peut être considéré comme appartenant à sa classe ou à une classe parente selon le besoin, et cela de façon dynamique. En d'autres termes, le lien entre une instance et une classe n'est pas unique et statique. Au contraire, il est établi de façon dynamique, au moment où l'objet est utilisé. C'est là la première manifestation du polymorphisme.

Retour sur l'initialisation

Pour qu'un objet puisse être considéré comme une instance de la classe **Pomme** ou une instance de la classe **Fruit**, une condition est nécessaire : il doit être réellement une instance de chacune de ces classes. Il faut donc que, lors de l'initialisation, un objet de chaque classe soit créé. Nous pouvons le vérifier au moyen du programme suivant :

```
public class Polymorphisme {
   public static void main(String[] argv) {
      Pomme pomme = new Pomme(85);
      Orange orange = new Orange(107);
      pèse(orange);
   }
   static void pèse(Fruit f) {
      int p = f.poids;
      System.out.println("Ce fruit pese " + p + " grammes.");
   }
}
abstract class Fruit {
   int poids;
   Fruit() {
      System.out.println("Creation d'un Fruit.");
   }
}
```

```
class Pomme extends Fruit {
   Pomme(int p) {
      poids = p;
      System.out.println("Creation d'une Pomme.");
   }
}

class Orange extends Fruit {
   Orange(int p) {
      poids = p;
      System.out.println("Creation d'une Orange.");
   }
}
```

Ce programme définit une classe **Fruit** qui contient un constructeur sans argument et un champ **poids** de type **int**. La classe **Fruit** est **abstract** car on ne souhaite pas qu'il soit possible de créer un fruit sans indiquer de quel type de fruit il s'agit. Le programme définit également deux classes **Pomme** et **Orange** qui étendent la classe **Fruit** et possèdent un constructeur prenant un argument de type **int** dont la valeur sert à initialiser le champ **poids**. Il est intéressant de noter que les classes **Pomme** et **Orange** ne possèdent pas de champs **poids**.

La classe **Polymorphisme**, qui est la classe principale du programme, contient une méthode **main** et une méthode **pèse(Fruit f)**. Celle-ci prend pour argument un objet de type **Fruit** et affiche son poids. Dans la méthode **main**, nous créons une instance de **Pomme**, puis une instance d'**Orange**. Enfin, nous appelons la méthode **pèse** avec pour argument l'objet **orange**.

Si nous exécutons ce programme, nous obtenons le résultat suivant :

```
Création d'un Fruit.
Création d'une Pomme.
Création d'un Fruit.
Création d'une Orange.
Ce fruit pese 107 grammes
```

Que se passe-t-il ? Nous nous apercevons qu'avant de créer une **Pomme**, le programme crée un **Fruit**, comme le montre l'exécution du constructeur de cette classe. La même chose se passe lorsque nous créons une **Orange**. Lorsque le constructeur de la classe **Pomme** est exécuté, le champ **poids** prend la valeur du paramètre passé. Ce champ correspond à la variable d'instance **poids** de la classe **Fruit**.

La partie la plus intéressante du programme se trouve à la dernière ligne de la méthode **main**. Celle-ci appelle la méthode **pèse** avec un argument apparemment de type **Orange** :

```
pèse(orange);
```

Or, il n'existe qu'une seule version de la méthode **pèse** et elle prend un argument de type **Fruit** :

```
static void pèse(Fruit f) {
```

D'après ce que nous savons de la façon dont Java gère les méthodes, cela devrait provoquer un message d'erreur du type :

```
Incompatible type for method. Can't convert Orange to Fruit.
```

Pourtant, tout se passe sans problème. En effet, l'objet passé à la méthode est certes de type **Orange**, mais également de type **Fruit**. L'objet pointé par le handle **orange** peut donc être affecté sans problème au paramètre **f**, qui est déclaré de type **Fruit**. Le fait d'établir le lien entre l'objet et la classe parente (**Fruit**) est un sur-casting. Le sur-casting est effectué automatiquement par Java lorsque cela est nécessaire.

Le sur-casting

L'objet pointé par le handle **orange** n'est en rien modifié par le sur-casting. A l'appel de la méthode **pèse**, l'objet pointé par le handle **orange** est passé à la méthode. Le handle **f** est alors pointé sur cet objet. **f** étant déclaré de

type **Fruit**, Java vérifie si l'objet correspond bien à ce type, ce qui est le cas. Le handle **orange** continue de pointer vers l'objet **orange**, qui est toujours de type **Orange** et **Fruit**. (Incidemment, cet objet est également de type **Object**, comme tous les objets Java.)

L'avantage de cette façon de procéder est que nous n'avons besoin que d'une seule méthode **pèse**. Sans le sur-casting, il nous faudrait écrire une méthode pour peser les oranges et une autre pour peser les pommes. En somme, tout cela est parfaitement naturel et correspond une fois de plus à la volonté de créer un langage permettant d'exprimer la solution avec les termes du problème. Lorsque vous pesez des pommes ou des oranges, vous n'utilisez pas un pèse-pommes ou un pèse-oranges ; éventuellement un pèse-fruits, mais plus probablement un pèse-petits-objets. Ce faisant, vous effectuez un sur-casting de pomme et orange en petit-objet.

Le sur-casting permet ici d'effectuer l'inverse de ce que permet la surcharge de méthodes. Il aurait été tout à fait possible d'écrire le programme de la façon suivante :

```
public class Polymorphisme2 {
   public static void main(String[] argv) {
      Pomme pomme = new Pomme(85);
      Orange orange = new Orange(107);
      pèse(orange);
   }

   static void pèse(Pomme f) {
      int p = f.poids;
      System.out.println("Ce fruit pese " + p + " grammes.");
   }
   static void pèse(Orange f) {
      int p = f.poids;
      System.out.println("Ce fruit pese " + p + " grammes.");
   }
}
```

Cette façon de faire nous aurait simplement obligés à créer et maintenir deux méthodes utilisant le même code, ce qui est contraire aux règles de la programmation efficace. En revanche, elle aurait un intérêt incontestable si nous souhaitions effectuer un traitement différent en fonction du type de **Fruit**, par exemple afficher le type. Nous pourrions alors modifier le programme de la façon suivante :

```
public class Polymorphisme3 {
   public static void main(String[] argv) {
      Pomme pomme = new Pomme(85);
      Orange orange = new Orange(107);
      pèse(orange);
   }

   static void pèse(Pomme f) {
      int p = f.poids;
      System.out.println("Cette pomme pese " + p + " grammes.");
   }
   static void pèse(Orange f) {
      int p = f.poids;
      System.out.println("Cette orange pese " + p + " grammes.");
   }
}
```

Le résultat correspond incontestablement à ce que nous souhaitons, mais les règles de la programmation efficaces ne sont toutefois pas complètement respectées. En effet, une partie du traitement (la première ligne) est encore commune aux deux méthodes. Bien entendu, dans un cas aussi simple, nous ne nous poserions pas toutes ces questions et nous nous satisferions sans problème d'une si petite entorse aux règles. Il est cependant utile, d'un point de vue pédagogique, de considérer la manière "élégante" de traiter le problème. Celle-ci consiste à mettre la partie commune du traitement dans une troisième méthode et à appeler cette méthode dans chacune des méthodes spécifiques, ce qui pourrait être réalisé de la façon suivante :

```
public class Polymorphisme4 {
   public static void main(String[] argv) {
```

```
      Pomme pomme = new Pomme(85);
      Orange orange = new Orange(107);
      pèse(orange);
   }

   static void pèse(Pomme f) {
      String p = getPoids(f);
      System.out.println("Cette pomme pese " + p);
   }
   static void pèse(Orange f) {
      String p = getPoids(f);
      System.out.println("Cette orange pese " + p);
   }

   static String getPoids(Fruit f) {
      return f.poids + " grammes";
   }
}
```

Nous avons ici déplacé dans la méthode **getPoids** non seulement la lecture du poids, mais également la transformation de celui-ci en une chaîne de caractères composée de la valeur du poids suivie de l'unité. A priori, le programme semble plus complexe. Cependant, si vous décidez d'exprimer le poids en kilogrammes, vous n'avez qu'une seule ligne à modifier. Le listing suivant montre le programme complet affichant le poids en kilogrammes :

```
public class Polymorphisme5 {
   public static void main(String[] argv) {
      Pomme pomme = new Pomme(85);
      Orange orange = new Orange(107);
      pèse(orange);
   }
   static void pèse(Pomme f) {
      String p = getPoids(f);
      System.out.println("Cette pomme pese " + p);
   }
```

```
   static void pèse(Orange f) {
      String p = getPoids(f);
      System.out.println("Cette orange pese " + p);
   }

   static String getPoids(Fruit f) {
      return (float)f.poids / 1000 + " kilogramme(s)";
   }
}

abstract class Fruit {
   int poids;
   Fruit() {
      System.out.println("Creation d'un Fruit.");
   }
}

class Pomme extends Fruit {
   Pomme(int p) {
      poids = p;
      System.out.println("Creation d'une Pomme.");
   }
}

class Orange extends Fruit {
   Orange(int p) {
      poids = p;
      System.out.println("Creation d'une Orange.");
   }
}
```

Le sur-casting explicite

Comme dans le cas des primitives (et bien qu'il s'agisse d'une opération tout à fait différente), il existe des sur-castings implicites et explicites. Cependant, la distinction n'est pas toujours aussi évidente. En effet, il ne faut pas

perdre de vue que les handles servent à manipuler des objets mais ne sont pas des objets. Le sur-casting le plus explicite est celui utilisant l'opérateur de sur-casting, constitué du type vers lequel on souhaite effectuer le sur-casting, placé entre parenthèses avant la référence à l'objet. Dans notre programme précédent, nous pourrions utiliser ce type de sur-casting de la façon suivante :

```
public class Polymorphisme6 {
   public static void main(String[] argv) {
      Pomme pomme = new Pomme(85);
      Orange orange = new Orange(107);
      pèse(orange);
   }

   static void pèse(Pomme f) {
      String p = getPoids((Fruit)f);
      System.out.println("Cette pomme pese " + p);
   }
   static void pèse(Orange f) {
      String p = getPoids((Fruit)f);
      System.out.println("Cette orange pese " + p);
   }

   static String getPoids(Fruit f) {
      return (float)f.poids / 1000 + " kilogramme(s)";
   }
}
```

Ici, **f** est sur-casté explicitement en **Fruit** avant d'être passé à la méthode **getPoids**. Cette opération est tout à fait inutile.

Un sur-casting un tout petit peu moins explicite est celui obtenu en affectant un objet à un handle de type différent. Par exemple :

```
Fruit pomme = new Pomme(85);
```

ou, en version longue :

```
Fruit pomme;
pomme = new Pomme(85);
```

Ici, le handle **pomme** est déclaré de type **Fruit** ; à la ligne suivante, un objet de type **Pomme** est créé et affecté au handle **pomme**. Répétons encore une fois que ni le handle ni l'objet ne sont modifiés. Les handles ne peuvent jamais être redéfinis dans le courant de leur existence. (En revanche, lorsqu'un handle est hors de portée, on peut définir un handle de même nom et d'un type différent.) Les objets, de leur côté, ne sont pas modifiés par le sur-casting. Seule la nature du lien qui les lie aux handles change en fonction de la nature des handles.

Le sur-casting lié au passage des paramètres, comme dans notre exemple, est exactement de ce type. Il n'est pas moins explicite. Il est simplement un peu moins lisible.

Le sur-casting implicite

Le sur-casting est implicite lorsque aucun handle n'est là pour indiquer qu'il a lieu. Ainsi, par exemple, nous verrons dans un prochain chapitre que les tableaux, en Java, ne peuvent contenir que des références d'objets. L'affectation d'un handle d'objet à un tableau provoque donc implicitement un sur-casting vers le type **Object**. Nous verrons ce que cela implique pour l'utilisation des tableaux.

Le sous-casting

Dans notre exemple précédent, nous avons vu que, dans la méthode **getPoids**, le handle d'objet manipulé est de type **Fruit**. L'objet, lui, est toujours de type **Pomme** ou **Orange**. Existe-t-il un moyen pour retrouver le type spécifique de l'objet ? En d'autres termes, est-il possible d'effectuer d'abord le

traitement général, et ensuite seulement le traitement particulier ? Reprenons la première version de notre programme :

```
public class Polymorphisme7 {
   public static void main(String[] argv) {
      Pomme pomme = new Pomme(85);
      Orange orange = new Orange(107);
      pèse(orange);
   }
   static void pèse(Fruit f) {
      int p = f.poids;
      System.out.println("Ce fruit pese " + p + " grammes.");
   }
}

abstract class Fruit {
   int poids;
   Fruit() {
      System.out.println("Creation d'un Fruit.");
   }
}

class Pomme extends Fruit {
   Pomme(int p) {
      poids = p;
      System.out.println("Creation d'une Pomme.");
   }
}

class Orange extends Fruit {
   Orange(int p) {
      poids = p;
      System.out.println("Creation d'une Orange.");
   }
}
```

Nous souhaiterions que la méthode **pèse** affiche, après le poids, un message indiquant s'il s'agit d'une **Pomme** ou d'une **Orange**. Nous pourrions modifier le programme de la façon suivante :

```
public class Polymorphisme8 {
   public static void main(String[] argv) {
      Pomme pomme = new Pomme(85);
      Orange orange = new Orange(107);
      pèse(orange);
   }

   static void pèse(Fruit f) {
      int p = f.poids;
      System.out.println("Ce fruit pese " + p + " grammes.");
      quoi(f);
   }

   static void quoi(Pomme p) {
      System.out.println("C'est une Pomme.");
   }

   static void quoi(Orange o) {
      System.out.println("C'est une Orange.");
   }
}
```

Malheureusement, ce programme provoque une erreur de compilation :

```
Incompatible type for method. Explicit cast needed to
convert Fruit to Orange.
```

En effet, Java n'a aucun moyen de savoir comment effectuer le sous-casting. Celui-ci doit donc être effectué explicitement, ce qui n'est pas possible ici directement.

Le "late binding"

Notre problème peut tout de même être résolu grâce au fait que Java utilise une technique particulière pour déterminer quelle méthode doit être appelée. Dans la plupart des langages, lorsque le compilateur rencontre un appel de méthode (ou de *fonction* ou *procédure*, selon la terminologie employée), il doit être à même de savoir exactement de quelle méthode il s'agit. Le lien entre l'appel et la méthode est alors établi au moment de la compilation. Cette technique est appelée *early binding*, que l'on pourrait traduire par *liaison précoce*. Java utilise cette technique pour les appels de méthodes déclarées **final**. Elle a l'avantage de permettre certaines optimisations. En revanche, pour les méthodes qui ne sont pas **final**, Java utilise la technique du *late binding* (liaison tardive). Dans ce cas, le compilateur n'établit le lien entre l'appel et la méthode qu'au moment de l'exécution du programme. Ce lien est établi avec la version la plus spécifique de la méthode. Dans notre cas, il nous suffit d'utiliser une méthode pour chaque classe (**Pomme** et **Orange**) ainsi que pour la classe parente **Fruit**, au lieu d'une méthode statique dans la classe principale (**Polymorphisme**). Ainsi, la méthode la plus spécifique (celle de la classe **Pomme** ou **Orange**, selon le cas) sera utilisée chaque fois que la méthode sera appelée avec un argument de type **Fruit**. Il nous faut donc modifier le programme de la façon suivante :

```
public class Polymorphisme9 {
   public static void main(String[] argv) {
      Pomme pomme = new Pomme(85);
      Orange orange = new Orange(107);
      pèse(pomme);
   }

   static void pèse(Fruit f) {
      int p = f.poids;
      System.out.println("Ce fruit pese " + p + " grammes.");
      f.quoi();
   }
}
```

```
class Fruit {
   int poids;

   Fruit() {
      System.out.println("Creation d'un Fruit.");
   }

   void quoi() {}
}

class Pomme extends Fruit {
   Pomme(int p) {
      poids = p;
      System.out.println("Creation d'une Pomme.");
   }

   void quoi() {
      System.out.println("C'est une Pomme.");
   }

}

class Orange extends Fruit {
   Orange(int p) {
      poids = p;
      System.out.println("Creation d'une Orange.");
   }

   void quoi() {
      System.out.println("C'est une Orange.");
   }
}
```

A la fin de la méthode **pèse**, nous appelons la méthode **quoi** de l'objet **f**. Or, l'objet **f**, tout en étant une instance de **Fruit**, est également une instance de **Pomme**. Java lie donc l'appel de la méthode avec la méthode correspondant au type le plus spécifique de l'objet. La méthode existant dans la classe

Fruit ainsi que dans la classe **Pomme**, c'est cette dernière version qui est utilisée, ce qui nous permet d'atteindre le but recherché.

Note : Nous avons supprimé la déclaration **abstract** de la classe **Fruit** uniquement pour montrer que le *late binding* est bien la cause de ce fonctionnement. De la même façon, nous n'avons pas déclaré **abstract** la méthode **quoi** de la classe **Fruit** pour qu'aucune interférence ne puisse être suspectée. Dans un cas pratique, la classe **Fruit** et la méthode **quoi** de cette classe seraient déclarées **abstract** (ou tout du moins la méthode **quoi**, ce qui rend automatiquement la classe **Fruit abstract** également) :

```
abstract class Fruit {
   int poids;
   Fruit() {
      System.out.println("Creation d'un Fruit.");
   }
   abstract void quoi(Fruit f);
}
```

Certains programmes tirent parti de la technique du *late binding* sans que cela apparaissent au premier coup d'œil. Examinez, par exemple, le programme suivant :

```
public class Dressage1 {
   public static void main(String[] argv) {
      Chien milou = new Chien();
      Cheval tornado = new Cheval();
      Dinosaure denver = new Dinosaure();
      avance(milou);
      avance(tornado);
      avance(denver);
      arrête(milou);
      arrête(tornado);
      arrête(denver);
   }
   static void avance(Animal a) {
      if (a.avance)
```

```
            System.out.println("Ce " + a + " est deja en marche.");
         else {
            a.crie();
            a.avance = true;
            System.out.println("Ce " + a + " est maintenant en marche.");
         }
      }
      static void arrête(Animal a) {
         if (a.avance) {
            a.avance = false;
            System.out.println("Ce " + a + " est maintenant arrete.");
         }
         else
            System.out.println("Ce " + a + " est deja arrete.");
      }
}

abstract class Animal {
      boolean avance;
      void crie() {
         System.out.println("Le cri de cet animal est inconnu.");
      }
}

class Chien extends Animal {
      public void crie() {
         System.out.println("Ouah-Ouah !");
      }
      public String toString() {
         return "chien";
      }
}

class Cheval extends Animal {
      public void crie() {
         System.out.println("hiiiii !");
      }
      public String toString() {
```

```
      return "cheval";
   }
}
class Dinosaure extends Animal {
   public String toString() {
      return "dinosaure";
   }
}
```

Dans ce programme, nous définissons une classe **Animal** contenant une variable d'instance **avance** de type **boolean** qui vaut **true** si l'animal est en marche et **false** s'il est arrêté, une méthode **crie**, et deux méthodes **static**, **avance** et **arrête**.

La méthode **avance** prend pour argument un objet de type **Animal** et teste la variable **avance** de cet objet pour savoir si l'animal est déjà en mouvement. Si c'est le cas, un message est affiché pour l'indiquer. Dans le cas contraire, l'animal pousse un cri et se met en mouvement.

La méthode **crie** est appelée ici sans argument. Ce n'est donc pas la nature de l'argument qui détermine quelle méthode va être appelée. L'appel est de la forme :

```
a.crie();
```

C'est donc la méthode **crie** de l'objet **a** qui est appelée. Comme dans l'exemple précédent, le compilateur ne peut pas savoir quel est le type spécifique de l'objet **a**. Tout ce qu'il sait est qu'il s'agit d'un **Animal**. Cette fois encore, le lien est établi au moment de l'exécution. Lorsque la méthode a été redéfinie dans la classe spécifique de l'objet, comme dans le cas de **Chien** ou **Cheval**, c'est cette version qui est appelée. Dans le cas contraire (celui de la classe **Dinosaure**), la version générique de la classe **Animal** est appelée.

Dans ce programme, le *late binding* est mis à profit d'une autre façon. Examinez la ligne suivante, extraite de la méthode **avance**.

```
System.out.println("Ce " + a + " est maintenant en marche.");
```

a est l'argument passé à la méthode **avance()**. Il s'agit d'un objet de type **Animal**. Que se passe-t-il lorsque l'on essaie d'afficher un objet comme s'il s'agissait d'une chaîne de caractères ? La méthode **toString()** de cet objet est appelée. La méthode **toString()** est définie dans la classe **Object**, dont dérivent toutes les classes Java. La ligne ci-dessus est donc équivalente à :

```
System.out.println("Ce " + (Object)a.toString()
                    + " est maintenant en marche.");
```

Nous sommes alors exactement ramenés à la situation précédente. La méthode de la classe la plus spécifique est liée à l'appel de méthode au moment de l'exécution du programme. **a** est :

- un **Object**,
- un **Animal**,
- un **Chien**, un **Cheval** ou un **Dinosaure**.

C'est donc dans la classe la plus spécifique (**Chien**, **Cheval** ou **Dinosaure**) que Java cherche d'abord la méthode **toString()**. Comme nous avons redéfini cette méthode dans chacune de ces classes, c'est la version redéfinie qui est liée à l'appel de méthode.

Dans cet exemple, nous voyons qu'il n'est pas nécessaire que la méthode existe dans la classe de l'objet. Il suffit qu'elle existe dans une classe parente pour que la version d'une classe dérivée soit appelée.

Note : Dans cet exemple, nous avons utilisé une technique particulière en ce qui concerne la déclaration de la classe **Animal**. En effet, la méthode **crie** de cette classe ne peut être déclarée **abstract** puisqu'elle possède une définition. Cette définition doit être utilisée pour toutes les classes dérivées qui ne redéfinissent pas la méthode **crie**. Pour empêcher l'instanciation de la classe, il est donc nécessaire dans ce cas de déclarer la classe elle-même **abstract**. Il est d'ailleurs prudent de toujours déclarer explicitement une classe **abstract** plutôt que de se reposer sur le fait qu'elle contient des méthodes **abstract**. En effet, si la définition de la classe est longue, la pré-

sence de ces méthodes peut ne pas être évidente, ce qui complique la maintenance en réduisant la lisibilité. Par ailleurs, vous pouvez être amené à supprimer une méthode **abstract** d'une classe. S'il s'agissait de la seule méthode de ce type, votre classe devient de ce fait non **abstract,** éventuellement sans que vous y pensiez, ce qui peut être une source d'erreurs futures.

Les interfaces

Une autre forme de polymorphisme consiste à dériver une classe de plusieurs autres classes. Certains langages permettent cela sans restriction. Java ne l'autorise pas, mais permet d'aboutir au même résultat en utilisant les *interfaces*.

Une interface est une sorte de classe qui ne contient que des méthodes **abstract** et des variables **static** et **final**.

Une interface peut être créée de la même façon qu'une classe, en remplaçant, dans sa déclaration, le mot **class** par le mot **interface**. Comme une classe, elle peut être déclarée **public** (à condition d'être enregistrée dans un fichier portant le même nom). Elle peut être attribuée à un package. Si elle ne l'est pas, elle fera partie du package par défaut.

Une interface peut être étendue au moyen du mot clé **extends**. Dans ce cas, vous obtiendrez automatiquement une nouvelle interface. Pour dériver une classe d'une interface, vous devez utiliser le mot clé **implements**. Celui-ci doit être suivi de la liste des interfaces dont la classe dérive, séparées par des virgules. En effet, une classe peut dériver de plusieurs interfaces au moyen du mot clé **implements**.

Les méthodes d'une interface sont toujours **public**. Il est important de noter que l'implémentation des méthodes d'une interface dans la classe dérivée sera toujours **public**. En effet, Java ne permet pas de restreindre l'accès d'une méthode héritée.

Utiliser les interfaces pour gérer des constantes

Supposons que vous écriviez un programme produisant de la musique à l'aide d'une interface MIDI. Chaque note (correspondant à une fréquence) est représentée par une valeur numérique. Cette valeur est la même pour tous les instruments MIDI. Cependant, le cas de la batterie est particulier. Une batterie dispose de plusieurs instruments (caisse claire, grosse caisse, tom aigu, tom médium, tom basse, charley, crash, ride, etc.) Chacun de ces instruments n'émet qu'une seule note. (Cette affirmation ferait hurler les batteurs !) On utilise donc une valeur de note pour chaque instrument. Cependant, les boîtes à rythmes produisant les sons de batterie n'utilisent pas toutes la même correspondance entre une note et un instrument. Pour une marque de boîte à rythmes, la caisse claire correspondra à la note 63 alors que pour une autre, ce sera la note 84. Il vous faut donc gérer un ensemble de constantes correspondant à chaque marque de boîte à rythmes. Vous pouvez définir ces constantes dans vos classes, de la façon suivante :

```
class MaClasse extends XXX {
   final static int
              CAISSE_CLAIRE   = 48,
              GROSSE_CAISSE   = 57,
              CHARLEY_PIED    = 87,
              CHARLEY_MAIN    = 89,
              TOM_BASSE       = 66,
                 .
                 .
                 .
      etc.
}
```

Si plusieurs classes de votre programme utilisent ces valeurs, vous devrez les coder dans chacune de ces classes. Qu'arrivera-t-il si vous voulez adapter votre programme à une nouvelle marque de boîte à rythmes ? Vous devrez modifier toutes les classes qui utilisent ces données.

Une autre solution consiste à placer ces données dans une interface :

```
interface Yamaha {
     final static int
             CAISSE_CLAIRE   = 48,
             GROSSE_CAISSE   = 57,
             CHARLEY_PIED    = 87,
             CHARLEY_MAIN    = 89,
             TOM_BASSE       = 66,
     etc.
}
class maClasse extends XXX implements Yamaha {
     etc.
}
```

Tout ce que vous avez à faire maintenant pour adapter votre programme à une nouvelle boîte à rythmes est de créer une nouvelle interface pour celle-ci et de modifier la déclaration des classes qui l'utilisent. Si plusieurs classes sont dans ce cas, vous pouvez vous simplifier la vie encore un peu plus en utilisant une structure comme celle-ci :

```
interface Yamaha {
   final static int
           CAISSE_CLAIRE   = 48,
           GROSSE_CAISSE   = 57,
           CHARLEY_PIED    = 87,
           CHARLEY_MAIN    = 89,
           TOM_BASSE       = 66,
     etc.
}
interface Roland {
   final static int
           CAISSE_CLAIRE   = 68,
           GROSSE_CAISSE   = 69,
           CHARLEY_PIED    = 72,
           CHARLEY_MAIN    = 74,
           TOM_BASSE       = 80,
     etc.
}
```

```
interface BAR extends Yamaha {}
class maClasse extends XXX implements BAR {
     etc.
```

Ici, nous avons créé une interface intermédiaire. De cette façon, toutes les classes qui utilisent ces données implémentent l'interface **BAR**. Pour passer d'une boîte à rythmes à une autre, il suffit d'apporter une seule modification à la classe **BAR**.

Les classes Java étant chargées dynamiquement en fonction des besoins, il suffit de remplacer le fichier contenant la classe **BAR** pour que le programme fonctionne avec le nouveau matériel. On ne peut rêver d'une maintenance plus facile !

Note : Nous avons ici déclaré les constantes explicitement **final** et **static**, ce qui n'est pas nécessaire car les variables d'une interface sont toujours **final** et **static**. Par ailleurs, nous avons respecté une convention qui consiste à écrire les noms des variables **final** et **static** en majuscules, en séparant les mots à l'aide du caractère _. Dans d'autres langages, on écrit ainsi les constantes globales, c'est-à-dire celles qui sont disponibles dans la totalité du programme. En Java, les variables **final** et **static** sont ce qui se rapproche le plus des constantes globales.

Un embryon d'héritage multiple

Grâce aux interfaces, une classe Java peut hériter des constantes de plusieurs autres classes. Si une classe ne peut étendre qu'une seule classe parente, elle peut en revanche implémenter un nombre quelconque d'interfaces. Pour continuer notre programme de musique MIDI, nous pourrions ajouter sur le même principe des interfaces permettant de gérer différents synthétiseurs. Les synthétiseurs utilisent des numéros de *programmes* pour sélectionner les différentes sonorités. Ici encore, chaque marque a son propre arrangement.

```
interface T1 {
   final static int
           PIANO          = 1,
              .
```

```
            .
            .
          GUITARE         = 22,
     etc.
}

interface EX7 {
   final static int
          PIANO           = 1,
            .
            .
          ACCOUSTIC_BASS  = 22,
     etc.
}

interface Synthétiseur extends T1 {}
class maClasse extends XXX implements BAR, Synthétiseur {
     etc.
```

Le polymorphisme et les interfaces

De la même façon qu'un objet est à la fois une instance de sa classe et de toutes ses classes parentes, un objet sera valablement reconnu comme une instance d'une quelconque de ses interfaces. Le programme suivant en montre un exemple :

```
public class Midi2 {
   public static void main(String[] argv) {
      Sequence maSequence = new Sequence();
      System.out.println(maSequence instanceof Sequence);
      System.out.println(maSequence instanceof Object);
      System.out.println(maSequence instanceof Synthetiseur);
      System.out.println(maSequence instanceof BAR);
      System.out.println(maSequence instanceof Roland);
      System.out.println(maSequence instanceof T1);
   }
}
```

```
interface Yamaha {
   final static int
              CAISSE_CLAIRE   = 48,
              GROSSE_CAISSE   = 57,
              CHARLEY_PIED    = 87,
              CHARLEY_MAIN    = 89,
              TOM_BASSE       = 66;
}

interface Roland {
   final static int
              CAISSE_CLAIRE   = 68,
              GROSSE_CAISSE   = 69,
              CHARLEY_PIED    = 72,
              CHARLEY_MAIN    = 74,
              TOM_BASSE       = 80;
}

interface T1 {
   final static int
            PIANO     = 1,
            GUITARE      = 22;
//   etc.
}

interface EX7 {
   final static int
            PIANO     = 1,
            ACCOUSTIC_BASS  = 22;
//   etc.
}

interface BAR extends Roland {}

interface Synthetiseur extends T1 {}

class Sequence implements Synthetiseur, BAR {
}
```

Ce programme affiche :

```
true
true
true
true
true
true
```

montrant que l'objet **maSequence** est une instance de **Sequence**, d'**Object** (la classe **Sequence** étend automatiquement la classe **Object**), de **BAR**, de **Synthetiseur**, de **Roland** et de **T1**.

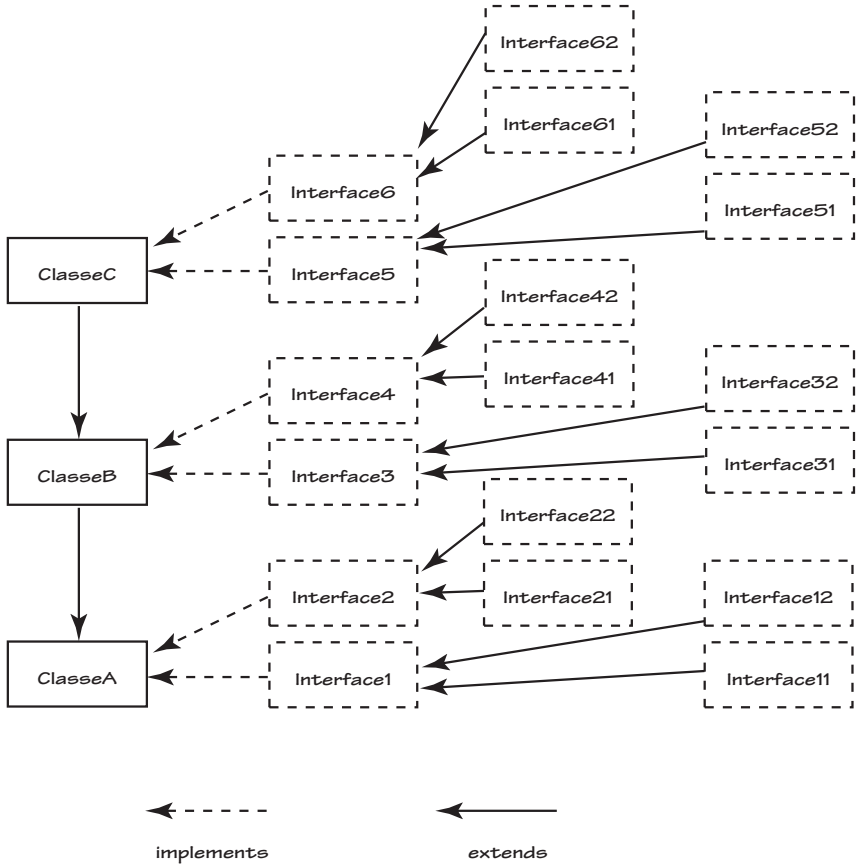
Nous aurions pu écrire notre programme différemment, en remplaçant les trois dernières lignes par :

```
interface Materiel extends Roland, T1 {}
class Sequence implements Materiel {
}
```

car une interface peut étendre un nombre quelconque d'interfaces. Pour résumer, un objet est une instance de :

- sa classe,
- toutes les classes parentes de sa classe,
- toutes les interfaces qu'il implémente,
- toutes les interfaces parentes des interfaces qu'il implémente,
- toutes les interfaces qu'implémentent les classes parentes de sa classe,
- toutes les interfaces parentes des précédentes.

Le schéma suivant résume la situation :



Dans cette structure, tout objet instance de la classe **ClasseA** est également instance (au sens de l'opérateur **instanceof**) de toutes les classes et interfaces représentées.

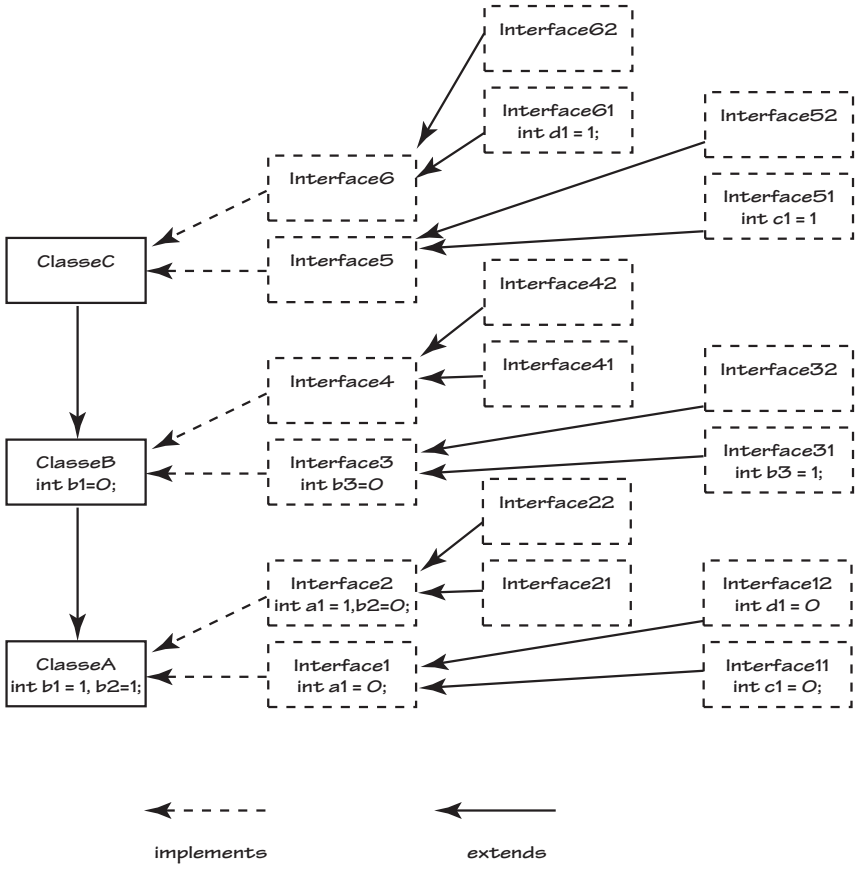
Cette particularité peut être employée, comme dans notre exemple précédent, pour permettre à une classe d'hériter des constantes de plusieurs autres classes, mais également pour servir simplement de marqueur, à l'aide d'instructions conditionnelles du type :

```
if (objet instanceof ClasseOuInterface1)
   traitement1;
else if (objet instanceof ClasseOuInterface2)
   traitement 2;
```

```
else if (objet instanceof ClasseOuInterface3)
   traitement 3;
etc
```

Nous verrons même dans un prochain chapitre qu'il est possible d'obtenir directement la classe d'un objet pendant l'exécution d'un programme.

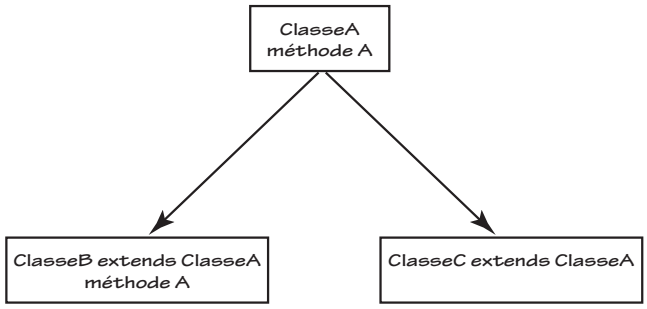
Attention : Ce genre de structure pose un problème potentiel : vous devez vous assurer que deux classes ou interfaces situées dans des branches différentes de la hiérarchie ne possèdent pas deux constantes de même nom. Dans le cas contraire, la référence sera ambiguë et le compilateur affichera un message d'erreur. Par exemple, le schéma ci-dessous :



comporte trois ambiguïtés : les variables **a1**, **c1** et **d1** sont en effet définies plusieurs fois dans des branches différentes. Dans une même branche, la variable la plus basse dans la hiérarchie masque les variables de même nom situées plus haut. En revanche, le compilateur n'est pas à même de résoudre les cas où les variables sont définies dans des branches différentes.

Pas de vrai héritage multiple

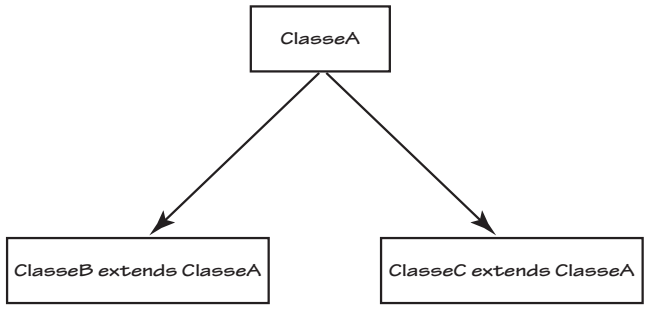
Malheureusement, exécuter un traitement ou un autre selon le type d'un objet en l'interrogeant pour savoir de quoi il est l'instance n'est pas une façon très élégante de programmer. Il est indiscutablement préférable, chaque fois que cela est possible, de laisser agir le polymorphisme. Ainsi dans l'exemple suivant :



si la méthode A est appelée pour un objet instance de la classe B, la méthode A de cette classe sera exécutée. En revanche, si la méthode A est appelée pour un objet de la classe C, c'est la méthode A de la classe A qui sera exécutée.

Cette structure est tout à fait impossible avec les interfaces. En effet, les interfaces ne peuvent pas comporter de définition de méthodes, mais seulement des déclarations. Par conséquent, une classe n'hérite pas des méthodes des interfaces qu'elle implémente, mais uniquement de l'obligation de les définir elle-même.

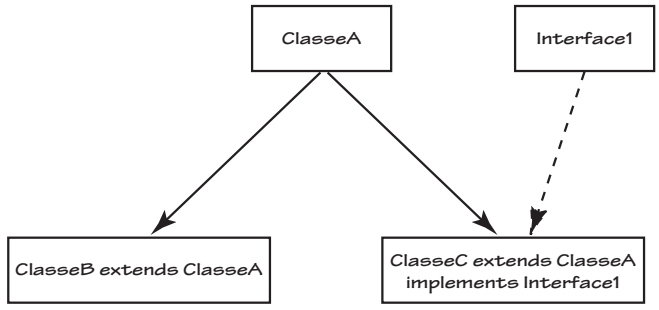
Le polymorphisme est mis en œuvre d'une autre manière dans la structure suivante :




```
méthodeA(ClasseA a){}
méthodeA(ClasseB b){}
```

Dans cet exemple, si la méthode A (qui peut être définie n'importe où) reçoit un objet instance de la classe A, la première version est exécutée. Si elle reçoit un objet instance de la classe B, la deuxième version est exécutée. Enfin, si elle reçoit un objet instance de la classe C, alors qu'il n'existe pas de version spécifique de la méthode pour ce type d'objet, la première version est exécutée car un objet de la classe C est aussi instance de la classe A.

Cette exploitation du polymorphisme est tout à fait possible avec les interfaces, à condition qu'il n'y ait pas d'ambiguïté. Par exemple, la structure suivante est impossible :




```
méthodeA(ClasseA a){}
méthodeA(ClasseB b){}
méthodeA(Interface1 i){}
```

En effet, la première et la troisième version de la méthode **A** sont ambiguës car leurs signatures ne se résolvent pas de façons distinctes si la méthode est appelée avec pour paramètre un objet de type **ClasseC**.

Quand utiliser les interfaces ?

La plupart des livres sur Java mettent en avant la possibilité de déclarer des méthodes abstraites comme un intérêt majeur des interfaces. On peut se demander, dans ce cas, quels avantages ont les interfaces sur les classes abstraites ? Le fait de pouvoir implémenter plusieurs interfaces n'est pas pertinent dans ce cas, puisque cela ne fait qu'augmenter le nombre de méthodes que doivent définir les classes dérivées, sans leur permettre d'hériter de quoi que ce soit d'autre, en dehors des constantes. Le fait d'obliger les classes qui les implémentent à définir les méthodes déclarées dans les interfaces est présenté comme une contrainte bénéfique pour l'organisation des programmes. Cet argument est fallacieux, d'autant qu'il arrive souvent d'implémenter une interface sans avoir besoin de définir toutes ses méthodes. Il faudra pourtant le faire, car cela est obligatoire pour que la classe ne soit pas abstraite. Certains éléments de la bibliothèque de Java sont fournis sous forme d'interfaces. La gestion des fenêtres, par exemple, nécessite l'implémentation de l'interface **WindowListener**, ce qui oblige à définir 7 méthodes, même si vous n'avez besoin que d'une seule d'entre elles. Pour échapper à cette contrainte, Java fournit par ailleurs une classe appelée **WindowAdapter** qui implémente l'interface **WindowListener** en définissant les 7 méthodes avec une définition vide. Cela permet au programmeur d'étendre la classe **WindowAdapter** au lieu d'implémenter l'interface **WindowListener**, et de n'avoir ainsi à redéfinir que les méthodes qui l'intéressent.

En fait, pour comprendre la nécessité des interfaces, il faut savoir comment circule l'information entre les classes Java. Supposons que vous souhaitiez écrire une application comportant une fenêtre avec un bouton. Lorsque l'utilisateur clique sur ce bouton, un texte est affiché dans une zone prévue à cet effet.

En Java, un clic sur un bouton est un *événement*. Cet événement déclenche l'envoi d'un message, qui est reçu par les objets qui ont la capacité de recevoir ces messages. Le problème est que tous les objets sont susceptibles

d'avoir besoin un jour ou l'autre de recevoir tous les messages. Le nombre de messages différents étant très important, il faudrait que tous les objets soient des récepteurs pour tous les messages potentiels, ce qui serait très lourd. Java est donc organisé d'une autre façon. Pour qu'un message du type "clic sur un bouton" soit reçu par un objet, il faut que celui-ci soit du type "receveur_de_clic". En fait, le véritable type est "écouteur d'action", ou **ActionListener**. Si **ActionListener** était une classe, tous les objets susceptibles de réagir au clic d'un bouton devraient être dérivés de cette classe, ce qui ne serait pas du tout pratique. Voilà pourquoi les concepteurs de Java ont préféré réaliser **ActionListener** sous la forme d'une interface, de façon que cette fonctionnalité puisse être ajoutée à n'importe quel objet, quelle que soit sa classe. Bien entendu, il serait totalement inutile que **ActionListener** définisse la méthode exécutée en cas de clic, car il peut s'agir de n'importe quoi, au gré des besoins du programmeur. Il a paru cependant logique de rendre obligatoire la présence de cette définition dans les classes dérivées, de façon à s'assurer qu'un message envoyé à un objet implémentant **ActionListener** ne finira pas dans le vide.

Le résultat est que l'on voit fleurir ce type de construction dans les programmes Java. Tout objet peut être un **ActionListener** (ou un *listener* de n'importe quoi d'autre), et les programmeurs ne s'en privent pas, alors qu'il existe d'autres façons beaucoup plus logiques de traiter le problème, comme nous le verrons dans un prochain chapitre. Disons qu'il vaut mieux utiliser des objets dont la seule fonction soit de traiter les messages d'événements, plutôt que d'intégrer cette fonctionnalité dans des objets dont ce n'est pas la fonction. Il suffit que ces objets spécialisés se trouve à l'intérieur des objets qui ont besoin de cette fonction. Par exemple, le programme suivant applique la première méthode :

```
class MaClasse extends AutreClasse implements ActionListener {
   définition de MaClasse
   void actionPerformed(...) {
      définition du traitement en cas de clic
   }
}
```

La méthode **actionPerformed** est déclarée dans l'interface **ActionListener** et doit donc être définie dans notre classe. Sa définition est évidemment tout à fait spécifique à cette classe.

L'exemple suivant applique la deuxième méthode :

```
class MaClasse extends AutreClasse {
        .
        .
   définition de MaClasse
        .
        .
}

class MonListener extends ActionListener
   void actionPerformed(...) {
      définition du traitement en cas de clic
   }
}
```

Bien entendu, la cible du message envoyé en cas de clic doit être définie comme étant une instance de **MaClasse** dans le premier cas, et une instance de **MonListener** dans le second. Le seul problème lié à la seconde approche est la nécessité de faire communiquer la classe **MontListener** avec la classe **MaClasse**, car il est évident que le traitement effectué dans la première concerne exclusivement la seconde. Nous verrons dans un prochain chapitre que ce problème se résout très simplement en plaçant la classe **MonListener** à l'intérieur de la classe **MaClasse**.

Héritage et composition

Les sections précédentes peuvent laisser à penser que Java est un langage plus pauvre que ceux qui autorisent l'héritage multiple. On peut débattre longtemps de cette question. En fait, les deux points de vue opposés peuvent être défendus :

- Les interfaces ne sont qu'un pâle succédané de l'héritage multiple.
- Les interfaces ne sont pas indispensables. Tout ce qui peut être réalisé avec elles peut l'être sans elles.

Il est vrai que les langages autorisant l'héritage multiple permettent des constructions plus sophistiquées. Le prix à payer est la diminution de la lisibilité et de la maintenabilité des programmes. L'héritage multiple est une fonctionnalité un peu plus puissante mais beaucoup plus complexe à maîtriser. Les concepteurs de Java ont pensé que le jeu n'en valait pas la chandelle.

D'un autre côté, on peut très facilement se passer de l'héritage multiple, tout comme on peut facilement se passer des interfaces. Il serait évidemment stupide de s'en passer lorsqu'elles apportent un avantage évident. Il faut cependant se méfier de la tendance consistant à faire reposer toute la structure d'une application sur l'héritage. L'héritage est certes une fonctionnalité première des langages objets, mais au même titre que la composition. Chaque fois que vous devez résoudre un problème qui semblerait mériter l'utilisation de l'héritage multiple, vous devez vous demander si la composition n'est pas une solution plus appropriée. En d'autres termes, si vous pensez qu'un objet devrait être à la fois un X et un Y, demandez-vous si vous ne pouvez pas résoudre le problème avec un objet X possédant un Y (ou un Y possédant un X). C'est l'approche que nous avons décrite dans la section précédente. Plutôt que de considérer que notre classe devait être un **ActionListener**, nous avons décidé qu'elle posséderait un **ActionListener**.

Dans le cas des constantes, la question est encore plus simple. En effet, s'agissant de variables **final** et **static**, la relation impliquée par l'interface (de type *est un*) ne concerne que l'accès aux variables, et non les variables elles-mêmes.

Si nous reprenons l'exemple des boîtes à rythmes, nous pouvons obtenir exactement la même fonctionnalité sans utiliser les interfaces. Pour le montrer, nous avons inclus les deux versions de la classe dans un programme permettant de mettre en évidence la différence d'utilisation.

Version avec interfaces :

```
public class BAR1 {
   public static void main(String[] argv) {
      MaClasse maClasse = new MaClasse();
      maClasse.joueCaisseClaire();
   }
}

interface Yamaha {
   final static int
           CAISSE_CLAIRE   = 48,
           GROSSE_CAISSE   = 57,
           CHARLEY_PIED    = 87,
           CHARLEY_MAIN    = 89,
           TOM_BASSE       = 66;
}

interface Roland {
   final static int
           CAISSE_CLAIRE   = 68,
           GROSSE_CAISSE   = 69,
           CHARLEY_PIED    = 72,
           CHARLEY_MAIN    = 74,
           TOM_BASSE       = 80;
}

interface BAR extends Yamaha {}

class MaClasse implements BAR {

   void joueCaisseClaire(){
      System.out.println(CAISSE_CLAIRE);
   }
}
```

Version sans interface :

```
public class BAR2 {
   public static void main(String[] argv) {
      MaClasse maClasse = new MaClasse();
      maClasse.joueCaisseClaire();
   }
}

class Yamaha {
   final static int
           CAISSE_CLAIRE   = 48,
           GROSSE_CAISSE   = 57,
           CHARLEY_PIED    = 87,
           CHARLEY_MAIN    = 89,
           TOM_BASSE       = 66;
}

class Roland {
   final static int
           CAISSE_CLAIRE   = 68,
           GROSSE_CAISSE   = 69,
           CHARLEY_PIED    = 72,
           CHARLEY_MAIN    = 74,
           TOM_BASSE       = 80;
}

class BAR extends Yamaha {}

class MaClasse{

   void joueCaisseClaire(){
      System.out.println(BAR.CAISSE_CLAIRE);
   }
}
```

La seule différence importante est dans la dernière ligne du programme et concerne la façon d'accéder aux variables. Il est en effet indispensable d'indiquer dans quelle classe elles se trouvent, puisqu'elles ne se trouvent plus dans la classe qui les utilise. En revanche, la deuxième version permettrait de définir la classe **BAR** à l'intérieur de la classe **MaClasse** (comme nous apprendrons à le faire dans un prochain chapitre), ce que ne permet pas la version utilisant les interfaces.

Un autre avantage de cette approche est qu'elle offre une plus grande souplesse. En effet, nous avons choisi ici d'utiliser une classe intermédiaire, de façon que le changement d'implémentation soit transparent pour l'utilisateur de la classe **MaClasse**. C'est la définition de la classe **BAR** qui détermine le matériel utilisé. Cependant, il est également possible d'utiliser directement les classes correspondant à chaque matériel, et ainsi de choisir l'une ou l'autre de façon dynamique, au moment de l'exécution du programme :

```
void joueCaisseClaire(){
   if(yamaha)
      System.out.println(Yamaha.CAISSE_CLAIRE);
   else
      System.out.println(Roland.CAISSE_CLAIRE);
}
```

Résumé

Dans ce chapitre, nous avons présenté les différents aspects d'un élément essentiel des langages orientés objets, et de Java en particulier : le polymorphisme, qui permet à un objet d'appartenir à plusieurs types à la fois. Nous avons vu qu'il existait deux formes différentes de polymorphisme : un polymorphisme "hiérarchique", qui fait qu'un objet instance d'une classe est automatiquement instance des classes parentes, et un polymorphisme "non hiérarchique", qui permet qu'un objet soit considéré comme une instance d'un nombre quelconque d'interfaces.

Dans le prochain chapitre, nous nous changerons les idées en étudiant les structures qui permettent de stocker des collections de données. Nous verrons alors comment le polymorphisme est mis en œuvre par certaines de ces structures.

Chapitre 10

Les tableaux et les collections

Dans les chapitres précédents, nous avons appris à créer des objets et des primitives pour stocker des données. Chacun de ces éléments était considéré individuellement. Cependant, il arrive fréquemment que des éléments doivent être traités collectivement. Dans la vie courante, les exemples ne manquent pas. Le personnel d'une entreprise est un ensemble d'employés. Un mot est un ensemble de caractères. Une phrase est un ensemble de mots. Un polygone peut être décrit comme un ensemble de sommets. Les résultats d'un examen peuvent être un ensemble de couples *élève, note*.

Tous ces ensembles ne sont pas de même nature. Certains sont ordonnés : l'ensemble des mots d'une phrase ne constitue cette phrase que s'ils sont parcourus dans un ordre précis. D'autres ne le sont pas : à l'ensemble des

articles en stock ne correspond aucun ordre fondamental. Certains sont ordonnés de façon particulière : les sommets d'un polygone décrivent ce polygone s'ils sont ordonnés, mais cet ordre est double. (Selon certains points de vue, les polygones définis par le parcours d'un ensemble de sommets dans un sens ou dans l'autre sont équivalents.) Certains ensembles contiennent des objets de nature identique (les sommets d'un polygone) alors que d'autres contiennent des objets de différentes natures (les articles d'un stock). Certains ensembles ont un nombre d'éléments fixe alors que d'autres peuvent croître indéfiniment. Java dispose de plusieurs structures pour traiter ces différents cas.

Les tableaux

Les tableaux Java sont des structures pouvant contenir un nombre fixe d'éléments de même nature. Il peut s'agir d'objets ou de primitives. Chaque élément est accessible grâce à un indice correspondant à sa position dans le tableau. Les tableaux Java sont des objets, même lorsqu'ils contiennent des primitives.

Déclaration

Les tableaux doivent être déclarés comme tous les objets. Java dispose de deux syntaxes équivalentes pour la déclaration des tableaux, ce qui peut troubler les néophytes :

```
int[] x;
```

ou :

```
int x[];
```

L'utilisation de l'une ou l'autre de ces syntaxes est une affaire de goût personnel. Nous avons déjà affirmé notre préférence pour la première, qui exprime clairement que le handle de droite **x** sera de type **int[]**, c'est-à-dire

un tableau contenant des entiers, alors que le second signifie que le handle **x** est un tableau, et qu'il contiendra des éléments du type indiqué à gauche (**int**).

Cependant, pour un utilisateur non averti, ces deux formes peuvent sembler signifier :

x *est de type* int[] *; le type* x *est un tableau contenant des* int

x *est un tableau ; ce tableau est de type* int

Or, la deuxième affirmation est manifestement fausse. Il est donc plus lisible d'utiliser la première.

Souvenez-vous qu'une déclaration ne concerne qu'un handle et non l'objet correspondant. L'opération consiste simplement à créer le handle et à indiquer qu'il pourra être utilisé pour pointer vers un objet de type *tableau de int*. Aucun tableau n'est créé à ce stade.

Initialisation

L'initialisation est l'opération dont le résultat est la création d'un objet tableau. Les tableaux Java sont de taille fixe. L'initialisation devra donc indiquer la taille du tableau. La syntaxe à employer est la suivante :

```
x = new int[dimension];
```

Dimension est le nombre d'éléments que le tableau pourra contenir. L'initialisation crée un tableau contenant le nombre d'entrées indiqué, par exemple :

```
int[] x;
x = new int[5];
```

La première ligne crée un handle **x** qui pourra pointer vers un objet de type tableau de **int**. La deuxième ligne crée un tableau de 5 entrées et fait pointer **x** vers celui-ci. Les entrées du tableau seront numérotées de 0 à 4.

A ce stade, le tableau ne contient rien. En effet, les opérations ci-dessus ont créé des références à cinq **int**, **x[0]** à **x[4]**, un peu comme si nous avions écrit :

```
int x[0], x[1], x[2], x[3], x[4];
```

Comme les autres objets, les tableaux peuvent être déclarés et initialisés sur la même ligne :

```
int[] x = new int[5];
```

Initialisation automatique

Si nous écrivons le programme suivant, nous devrions, d'après ce que nous avons appris jusqu'ici, obtenir une erreur de compilation :

```
class TestArray {
   public static void main(String[] args) {
      int[] x;
      x = new int[5];
      System.out.println(x[4]);
   }
}
```

car **x[4]** n'a pas été explicitement initialisée. Pourtant, ce programme est compilé sans erreur et produit le résultat suivant :

```
0
```

De la même façon, une version de ce programme utilisant un "tableau d'objets" :

```
class TestArray2 {
   public static void main(String[] args) {
      Integer[] x;
      x = new Integer[5];
      System.out.println(x[4]);
   }
}
```

est compilée sans erreur et son exécution affiche :

```
null
```

Cela nous montre que Java initialise automatiquement les éléments des tableaux, avec la valeur **0** pour les primitives entières (**byte**, **short**, **int** et **long**), **0.0** pour les primitives flottantes (**float** et **double**), **false** pour les **boolean** et **null** pour les objets.

Les tableaux littéraux

De la même façon qu'il existe des valeurs littérales correspondant à chaque type de primitives, il existe également des tableaux littéraux. L'expression suivante est un tableau de cinq éléments de type **int** :

```
{1, 2, 3, 4, 5}
```

Cependant, leur usage est très limité. Ils ne peuvent en effet être employés que pour initialiser un tableau au moment de sa déclaration, en utilisant la syntaxe :

```
int[] x = {1, 2, 3, 4, 5};
```

Dans ce cas, le handle de tableau **x** est créé et affecté au type *tableau de* **int**, puis un nouveau tableau est créé comportant 5 éléments, et enfin, les valeurs 1, 2, 3, 4 et 5 sont affectées aux éléments **x[0]**, **x[1]**, **x[2]**, **x[3]** et **x[4]**.

Les tableaux littéraux peuvent contenir des variables, comme dans l'exemple suivant :

```
int a = 1, b = 2, c = 3, d = 4, e = 5;
int[] y = {a, b, c, d, e};
```

Ils peuvent même contenir des variables de types différents :

```
byte a = 1;
short b = 2;
char c = 3;
int d = 4;
long e = 5;
long[] y = {a, b, c, d, e};
```

Cela n'est cependant possible que grâce à la particularité suivante : comme avec les opérateurs, tous les éléments d'un tableau littéral sont sur-castés vers le type supérieur. Ici, bien que chaque variable soit d'un type différent, tous les éléments du tableau littéral **{a, b, c, d, e}** sont des **long**, ayant subi un sur-casting implicite vers le type de l'élément le plus long.

Les tableaux littéraux peuvent contenir des objets anonymes, comme le montre l'exemple suivant :

```
class TestArray3 {
   public static void main(String[] args) {
      A[] a = {new A(), new A(), new A(),};
   }
}

class A {}
```

Les objets du tableau littéral sont créés de façon anonyme. Ils ne sont pas perdus pour autant puisqu'ils sont immédiatement affectés aux éléments du tableau **a**.

Les tableaux de primitives sont des objets

Il est important de réaliser que la différence fondamentale existant entre les primitives et les objets existe également entre les primitives et les tableaux de primitives. Les tableaux sont des objets. Lorsque vous manipulez un tableau, vous ne manipulez réellement qu'un handle vers ce tableau. Le programme suivant met cela en évidence :

```
class TestArray4 {
   public static void main(String[] args) {
      int[] x = {1, 2, 3, 4, 5};
      int[] y = {10, 11, 12};
      x = y;
      x[2] = 100;
      System.out.println(y[2]);
   }
}
```

A la cinquième ligne de ce programme, nous affectons la valeur de **y** à **x**. Cela paraît choquant si l'on considère que **x** et **y** sont des tableaux de longueurs différentes. En fait, cela ne l'est pas du tout, car il s'agit simplement de détacher le handle **x** du tableau vers lequel il pointe pour le faire pointer vers le même tableau que le handle **y**. La seule chose nécessaire pour que cela soit possible est que les deux handles soient de même type.

A la ligne 6 du programme, nous modifions la valeur de l'élément d'indice 2 du tableau **x**. Comme nous le voyons à la ligne suivante, cette modification concerne également le tableau pointé par **y**, puisqu'il s'agit du même.

Le sur-casting implicite des tableaux

Tout comme les autres objets et les primitives, les tableaux peuvent être sur-castés implicitement. Le programme suivant en fait la démonstration :

```
class TestArray5 {
   public static void main(String[] args) {
   A[] a = {new A(), new A(), new A()};
```

```
      B[] b = new B[] {new B(), new B()};
      a = b;
      }
   }
   class A {}
   class B extends A {}
```

Ici, **a**, qui et un handle de type **A[]**, peut pointer vers le même objet que **b**, qui est un handle de type **B[]**, en raison du polymorphisme. Nous sommes en présence d'un sur-casting implicite. Nous constatons donc que la relation entre les types **A[]** et **B[]** est la même qu'entre les types **A** et **B**. Il n'aurait pas été possible d'écrire l'affectation inverse **b = a**.

En revanche, ce type de sur-casting est impossible avec les primitives. En effet, la relation entre les types **int** et **short** n'a rien à voir avec le polymorphisme. Un **short** peut être converti en un **int**, mais un **short** n'est pas un **int**. Le programme suivant :

```
class TestArray6 {
   public static void main(String[] args) {
      int[] x = {1, 2, 3, 4, 5};
      short[] y = {10, 11, 12};
      x = y;
      x[2] = 100;
      System.out.println(y[2]);
   }
}
```

ne peut donc être compilé sans produire l'erreur :

```
Incompatible type for =. Can't convert short[] to int[].
```

Note : Nous avons utilisé dans le programme **TestArray5** deux syntaxes différentes pour l'initialisation des tableaux **a** et **b**. La deuxième n'apporte rien de plus sous cette forme. Cependant, nous aurions pu écrire :

```
A[] b = new B[] {new B(), new B()};
```

ce qui est une nouvelle application du polymorphisme. En effet, cela consiste tout simplement à prendre un tableau initialisé comme instance d'une classe et à l'affecter à un handle de la classe parente. En revanche, la ligne :

```
B[] b = new A[] {new A(), new A()};
```

produit un message d'erreur, car il s'agit d'un sous-casting, qui doit donc être effectué explicitement. Le sur-casting décrit plus haut est légèrement différent du cas suivant :

```
A[] b = new A[] {new B(), new B()};
```

Dans ce cas, le sur-casting a lieu au niveau des éléments du tableau et non au niveau du tableau lui-même. Cette application du polymorphisme est souvent mise en œuvre pour créer des tableaux contenant des objets de types différents. Un tableau ne peut en effet contenir que des objets de même type, mais il peut s'agir du type d'une classe parente. Le programme suivant en montre un exemple :

```
class MesAnimaux {
   public static void main(String[] args) {

   Animal[] menagerie = new Animal[3];
   menagerie[0] = new Chien();
   menagerie[1] = new Chat();
   menagerie[2] = new Canari();
   menagerie[0].afficheType();
   menagerie[1].afficheType();
   menagerie[2].afficheType();
   }
}

abstract class Animal {
```

```
   public void afficheType() {
      System.out.println("Animal");
   }
}

class Chien extends Animal {
   public void afficheType() {
      System.out.println("Chien");
   }
}

class Chat extends Animal {
   public void afficheType() {
      System.out.println("Chat");
   }
}

class Canari extends Animal {
   public void afficheType() {
      System.out.println("Canari");
   }
}
```

Ce programme stocke trois objets de type **Chien**, **Chat** et **Canari** dans un tableau déclaré de classe **Animal**. La suite du programme utilise les méthodes **afficheType** (définies dans chaque classe) pour afficher une chaîne de caractères spécifique à chaque type.

Note : Les méthodes **afficheType** ne peuvent pas être statiques. En effet, une fois les objets sur-castés en **Animal** pour être placés dans le tableau, un appel à une méthode statique utilisant leur nom d'instance entraîne l'exécution de la méthode statique de la classe vers laquelle ils ont été sur-castés. Pour la même raison, l'utilisation du polymorphisme est impossible (pour l'instant) pour afficher le type. En effet, le programme suivant :

```
class MesAnimaux2 {
   public static void main(String[] args) {
```

```
      Animal[] menagerie = new Animal[3];
      menagerie[0] = new Chien();
      menagerie[1] = new Chat();
      menagerie[2] = new Canari();
      afficheType(menagerie[0]);
      afficheType(menagerie[1]);
      afficheType(menagerie[2]);
      }
      void afficheType(Chien c) {
         System.out.println("Chien");
      }
      void afficheType(Chat c) {
         System.out.println("Chat");
      }
      void afficheType(Canari c) {
         System.out.println("Canari");
      }
   }
   abstract class Animal {
   }
   class Chien extends Animal {
   }
   class Chat extends Animal {
   }
   class Canari extends Animal {
   }
```

produit trois erreurs de compilation :

```
Incompatible type for method. Explicit cast needed to
convert Animal to Canari.
```

Il est intéressant de noter qu'il s'agit trois fois de la même erreur, ce qui montre que Java est apparemment incapable de retrouver le type d'origine des objets sur-castés. Nous verrons dans un prochain chapitre comment contourner cette difficulté.

Les tableaux d'objets sont des tableaux de handles

Tout comme les handles d'objets ne sont pas des objets, les tableaux d'objets ne contiennent pas des objets, mais des handles d'objets. Nous pouvons mettre cela en (relative) évidence à l'aide du programme suivant :

```
class TestArray7 {
   public static void main(String[] args) {
      A a1 = new A(1);
      A a2 = new A(2);
      A a3 = new A(3);
      A a4 = new A(4);
      A a5 = new A(5);
      A[] x = {a1, a2, a3};
      A[] y = {a4, a5};
      y[0] = x[0];
      a1.setValeur(10);
      System.out.println(x[0].getValeur());
      System.out.println(y[0].getValeur());
   }
}
class A {
   int valeur;
   A(int v){
      valeur = v;
   }
   void setValeur(int v) {
      valeur = v;
   }
   int getValeur() {
      return valeur;
   }
}
```

Ce programme crée deux tableaux **x** et **y**, tous deux de type **A[]**. L'élément 0 du tableau **x** contient le handle **a1**, pointant vers un objet de type **A** dont

le champ **valeur** vaut 1. L'élément 0 du tableau **y** contient le handle **a4**, pointant vers un objet de type **A** dont le champ **valeur** vaut 4. La ligne :

```
y[0] = x[0];
```

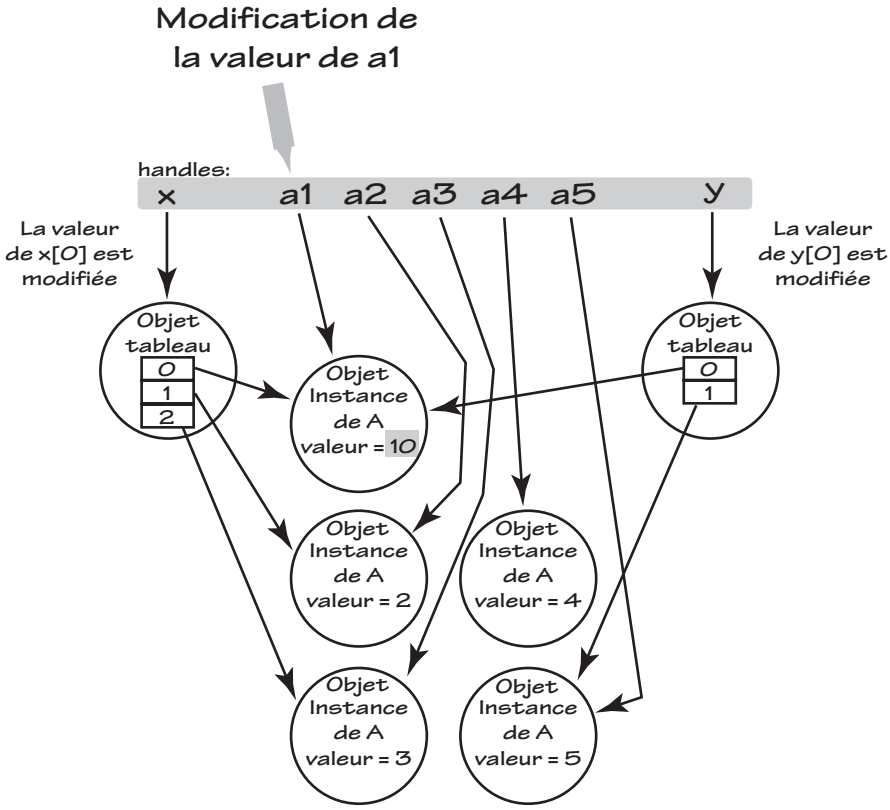
fait pointer l'élément 0 de **y** vers le même objet que l'élément 0 de **x**, c'est-à-dire l'objet pointé par **a1**. Lorsque, à la ligne suivante, nous modifions le champ **valeur** de **a1**, **x[0]** et **y[0]** sont également modifiés.

L'opération inverse donne un résultat équivalent :

```
class TestArray8 {
   public static void main(String[] args) {
      A a1 = new A(1);
      A a2 = new A(2);
      A a3 = new A(3);
      A[] x = {a1, a2, a3};
      A[0].setValeur(10);
      System.out.println(a1.getValeur());
   }
}
class A {
   int valeur;
   A(int v){
      valeur = v;
   }
   void setValeur(int v) {
      valeur = v;
   }
   int getValeur() {
      return valeur;
   }
}
```

Ici, lorsque nous modifions la valeur de **A[0]**, **a1** est également modifié. Encore une fois, répétons que les tableaux permettent de stocker et de ma-

nipuler des handles et des primitives, mais en aucun cas des objets. La situation de l'exemple **TestArray7** peut être représentée graphiquement de la façon suivante :



La taille des tableaux

La taille des tableaux est fixe. Une fois initialisée, elle ne peut plus être modifiée. Cependant, il n'est pas nécessaire qu'elle soit connue au moment de la compilation. Un tableau peut parfaitement être initialisé dynamiquement, pendant l'exécution du programme. Dans l'exemple suivant, nous créons un tableau dont le nombre d'éléments est déterminé par la variable **nbe**, dont la valeur est calculée de manière aléatoire :

```
import java.util.*;
class TestArray9 {
   public static void main(String[] args) {
      int x = Math.abs((new Random()).nextInt()) % 10;
      int[] y;
      y = new int[x];
      System.out.println(x)
   }
}
```

La quatrième ligne du programme est un peu obscure, mais finalement assez simple à comprendre. Une instance de la classe **java.util.Random** est créée. Il s'agit d'un générateur de nombres aléatoires. La méthode **nextInt** de cet objet est appelée. Elle renvoie un **int** quelconque. Cette valeur est fournie comme argument de la méthode **abs** de la classe **java.lang.Math** pour obtenir une valeur toujours positive. Le résultat est modulé par 10, ce qui nous donne un entier compris entre 0 (inclus) et 10 (exclu). Une fois le tableau créé, la valeur de **x** est affichée afin de vérifier qu'elle est différente à chaque exécution. Il est donc évident que la taille du tableau ne peut pas être connue lors de la compilation.

Ici, nous connaissons la taille du tableau, puisqu'il s'agit de la valeur de **x**. Nous aurions pu cependant écrire le programme sans utiliser la variable **x**, de la façon suivante :

```
import java.util.*;
class TestArray10 {
   public static void main(String[] args) {
      int[] y;
      y = new int[Math.abs((new Random()).nextInt()) % 10];
   }
}
```

Comment connaître, alors, la taille du tableau ? Tous les tableaux possèdent un champ **length** qui peut être interrogé pour connaître leur nombre d'éléments. Il nous suffit d'ajouter une ligne à notre programme pour afficher la taille du tableau :

```
   System.out.println(y.length);
```

Note : Remarquez au passage que cet exemple met en évidence le fait qu'il est parfaitement possible de créer un tableau de taille nulle. Le plus difficile est de trouver quelque chose à faire avec !

Le champ **length** des tableaux est souvent utilisé pour initialiser ses éléments à l'aide d'une boucle **for**. En effet, Java initialise les éléments des tableaux contenant des objets à la valeur **null**, correspondant à un handle pointant vers rien du tout. Ce type de handle peut parfaitement être manipulé, mais si vous tentez d'accéder à l'objet correspondant, vous provoquerez une erreur d'exécution.

Par exemple, le programme suivant est compilé sans erreur :

```
class TestArray11 {
   public static void main(String[] args) {
      A[] x = new A[10];
      x[0].setValeur(10);
   }
}
class A {
   int valeur;
   A(int v){
      valeur = v;
   }
   void setValeur(int v) {
      valeur = v;
   }
   int getValeur() {
      return valeur;
   }
}
```

Cependant, si vous tentez de l'exécuter, vous obtiendrez le message :

```
java.lang.NullPointerException
        at TestArray10.main(TestArray10.java:4)
```

car **x[0]** contient **null** et non un handle pointant vers une instance de la classe **A**.

Pour initialiser les éléments du tableau, nous pouvons procéder de la manière suivante :

```
class TestArray12 {
   public static void main(String[] args) {
      A[] x = new A[10];
      for (int i = 0; i < x.length; i++)
         x[i] = new A(0);
      x[0].setValeur(10);
   }
}
```

Attention : Deux erreurs se produisent fréquemment dans ce type de programme. La première consiste à écrire :

```
      for (int i = 0; i <= x.length; i++)
```

Dans ce cas, le programme sera compilé correctement mais produira une erreur à l'exécution. En effet, si la taille du tableau est 10, les indices vont de 0 à 9. La condition testée doit donc être **i < x.length** et non **i <= x.length**. Une autre erreur plus difficile à détecter produit le message suivant lors de la compilation :

```
TestArray.java:4: Attempt to reference field lenght in a A[].
          for (int i = 0; i < x.lenght; i++)
                                ^
```

Il est arrivé à certains de chercher longtemps la cause de cette erreur avant de s'apercevoir qu'elle venait d'une faute de frappe (**lenght** au lieu de **length**).

Les tableaux multidimensionnels

Les tableaux Java peuvent comporter plusieurs dimensions. Il est ainsi possible de créer des tableaux rectangulaires, cubiques, ou à un nombre quelconque de dimensions. Pour déclarer un tableau multidimensionnel, vous devez utiliser la syntaxe suivante :

```
int [][] x;
```

L'initialisation du tableau peut être effectuée à l'aide de tableaux littéraux :

```
int [][] x = {{1, 2, 3, 4},{5, 6, 7, 8}};
```

ou à l'aide de l'opérateur **new** :

```
int [][] x = new int[2][4];
```

Pour accéder à tous les éléments du tableau, vous devez utiliser des boucles imbriquées :

```
int [][] x = new int[2][4];
for (int i = 0; i < x.length; i++) {
   for (int j = 0; j < x[i].length; j++) {
      x[i][j] = 10;
   }
}
```

Ces exemples mettent en évidence la véritable nature des tableaux multidimensionnels. Il s'agit en fait de tableaux de tableaux. Par conséquent, il est possible de créer des tableaux de forme quelconque, par exemple des tableaux triangulaires :

```
int [][] x = {{1}, {2, 3}, {4, 5, 6}};
```

ou en losange :

```
int [][] x = {{1}, {2, 3}, {4, 5, 6}}, {7, 8}, {9}};
```

En fait, la syntaxe :

```
int [][] x = new int[4][5];
```

crée un tableau de quatre tableaux. Ces quatre tableaux ont tous cinq éléments. Si vous voulez utiliser cette syntaxe pour créer des tableaux non rectangulaires, il faut initialiser séparément chaque sous-tableau. Pour créer le tableau en triangle de l'exemple précédent, vous pouvez procéder de la façon suivante :

```
int [][] x = new int[3][];
for (int i = 0; i < 3; i++) {
   x[i] = new int[i + 1];
}
```

Vous pouvez, de la même façon, construire un tableau pyramidal de hauteur 10 :

```
int [][][] x = new int[10][][];
for (int i = 0; i < 10; i++) {
   x[i] = new int[i + 1][i + 1];
}
```

ou encore un tableau tétraédrique :

```
int [][][] x = new int[10][][];
for (int i = 0; i < 10; i++) {
   x[i] = new int[i + 1][];
   for (int j = 0; j < 10; j++) {
      x[i][j] = new int [j + 1];
   }
}
```

Les tableaux et le passage d'arguments

Les tableaux peuvent être passés comme arguments lors des appels de méthodes, ou utilisés par celles-ci comme valeurs de retour, au même titre que n'importe quels autres objets : en effet, les arguments passés ou les valeurs retournées ne sont que des handles. Vous devez évidemment en tenir compte

lors de la modification des tableaux (comme pour tous les autres objets d'ailleurs). Si, à l'intérieur d'une méthode, vous modifiez un tableau passé en argument, cette modification affecte également le tableau à l'extérieur de la méthode, puisqu'il s'agit du même objet. Le programme suivant en fait la démonstration :

```
class MesAnimaux3 {
   public static void main(String[] args) {
   Animal[] menagerie = new Animal[3];
   menagerie[0] = new Chien();
   menagerie[1] = new Chat();
   menagerie[2] = new Canari();

   menagerie[0].afficheType();
   menagerie[1].afficheType();
   menagerie[2].afficheType();
   System.out.println();

   Animal[] menagerie2 = modifierEléments(menagerie);
   menagerie2[0].afficheType();
   menagerie2[1].afficheType();
   menagerie2[2].afficheType();
   System.out.println();

   menagerie[0].afficheType();
   menagerie[1].afficheType();
   menagerie[2].afficheType();

   }

   static Animal[] modifierEléments(Animal[] a) {
      for (int i = 0; i < a.length; i++) {
         a[i] = new Canari();
      }
      return a;
   }
}
```

```
abstract class Animal {
   public  void afficheType() {
      System.out.println("Animal");
   }
}

class Chien extends Animal {
   public  void afficheType() {
      System.out.println("Chien");
   }
}

class Chat extends Animal {
   public void afficheType() {
      System.out.println("Chat");
   }
}

class Canari extends Animal {
   public  void afficheType() {
      System.out.println("Canari");
   }
}
```

Ce programme affiche :

```
Chien
Chat
Canari

Canari
Canari
Canari

Canari
Canari
Canari
```

ce qui met en évidence le fait que le tableau original est modifié par la méthode **modifierEléments**, pour la raison bien simple que, pendant toute l'exécution de ce programme, il n'existe qu'un seul tableau (mais trois handles).

Copie de tableaux

Il peut toutefois être nécessaire de travailler sur une copie d'un tableau en laissant l'original intact. Nous verrons dans un prochain chapitre qu'il est possible d'obtenir une copie d'un tableau, comme d'autres objets, au moyen de techniques conçues à cet effet. Le cas des tableaux est toutefois un peu particulier. En effet, les tableaux ne possèdent d'autres caractéristiques spécifiques que leurs nombres d'éléments et leurs éléments. Il est donc possible de faire une copie d'un tableau en créant un nouveau tableau de même longueur, et en recopiant un par un ses éléments, comme dans l'exemple suivant :

```
class MesAnimaux4 {
   public static void main(String[] args) {

      Animal[] menagerie = new Animal[3];

      menagerie[0] = new Chien();
      menagerie[1] = new Chat();
      menagerie[2] = new Canari();

      System.out.println("menagerie:");
      menagerie[0].afficheType();
      menagerie[1].afficheType();
      menagerie[2].afficheType();
      System.out.println();

      Animal[] menagerie2 = copierTableau(menagerie);

      System.out.println("menagerie2:");
      menagerie2[0].afficheType();
      menagerie2[1].afficheType();
```

```
        menagerie2[2].afficheType();
        System.out.println();

        Animal[] menagerie3 = modifierEléments(menagerie);


        System.out.println("menagerie3:");
        menagerie3[0].afficheType();
        menagerie3[1].afficheType();
        menagerie3[2].afficheType();
        System.out.println();

        System.out.println("menagerie:");
        menagerie[0].afficheType();
        menagerie[1].afficheType();
        menagerie[2].afficheType();
        System.out.println();

        System.out.println("menagerie2:");
        menagerie2[0].afficheType();
        menagerie2[1].afficheType();
        menagerie2[2].afficheType();
    }
    static Animal[] modifierEléments(Animal[] a) {
        for (int i = 0; i < a.length; i++) {
            a[i] = new Canari();
        }
        return a;
    }
    static Animal[] copierTableau(Animal[] a) {
        Animal[] b = new Animal[a.length];
        for (int i = 0; i < a.length; i++) {
            b[i] = a[i];
        }
        return b;
    }
}
```

```
abstract class Animal {
   public  void afficheType() {
      System.out.println("Animal");
   }
}

class Chien extends Animal {
   public  void afficheType() {
      System.out.println("Chien");
   }
}

class Chat extends Animal {
   public void afficheType() {
      System.out.println("Chat");
   }
}

class Canari extends Animal {
   public  void afficheType() {
      System.out.println("Canari");
   }
}
```

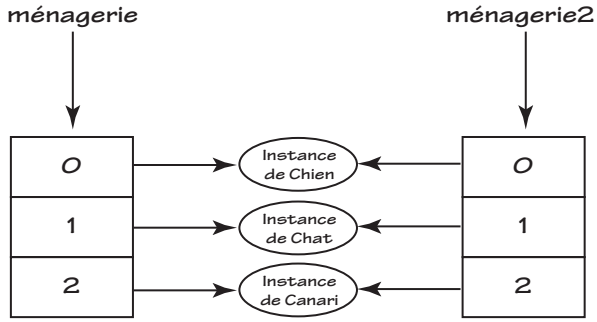
Ce programme affiche :

```
menagerie:
Chien
Chat
Canari

menagerie2:
Chien
Chat
Canari

menagerie3:
```

```
Canari
Canari
Canari

menagerie:
Canari
Canari
Canari

menagerie2:
Chien
Chat
Canari
```

Nous voyons ici que le tableau **menagerie2[]** est bien une copie du tableau original et non un handle vers le même tableau, et n'est donc pas modifié lorsque la méthode **modifierEléments** est appelée.

Notez toutefois qu'il ne s'agit que d'une copie sur un seul niveau. En effet, les handles du tableau **menagerie2[]** sont bien des copies des handles du tableau original, mais ils pointent vers les mêmes éléments. Le schéma ci-après met en évidence la situation obtenue après la copie :



Cette situation serait également obtenue lors de la copie d'un tableau à deux dimensions ou plus si vous ne copiez que le premier niveau. En effet, comme nous l'avons dit précédemment, les tableaux à deux dimensions sont des

tableaux de tableaux, les tableaux à trois dimensions sont des tableaux de tableaux de tableaux, etc.

En fait, la copie de tableaux par ce procédé est une véritable copie uniquement dans le cas des tableaux de primitives. Dans le cas des tableaux d'objets, seule la structure du tableau est copiée, mais pas les objets qu'il contient.

Les vecteurs

Les tableaux permettent de gérer de façon simple et efficace des collections d'objets de même type. Leur principal inconvénient est que leur taille est fixe. Ce choix est fait pour des raisons d'efficacité. La manipulation des tableaux est beaucoup plus rapide lorsque leur taille est fixe.

Cependant, il est très facile de créer des tableaux à taille variable. Il suffit de créer un tableau de taille fixe et de le recopier dans un tableau plus grand chaque fois que cela est nécessaire. Il faudra éventuellement aussi, gérer la diminution de la taille du tableau. De plus, il est clair qu'il n'est pas très efficace d'augmenter la taille de une unité chaque fois qu'un élément doit être ajouté. Il serait préférable de créer un tableau assez grand, puis d'augmenter sa taille de plusieurs unités lorsque l'on approche du remplissage. Java dispose d'un type d'objet qui prend en charge ces opérations sans que vous ayez à vous en soucier. Il s'agit du type **Vector** (en français : vecteur).

Outre le fait que leur manipulation est beaucoup plus lente que celle des tableaux, les vecteurs ont une autre particularité : ils n'existent que pour le type **Object**. Cela ne pose pas de problème lors de l'affectation d'un objet à un vecteur. Java effectue alors un sur-casting exactement comme si vous affectiez un objet quelconque à un tableau déclaré de type **Object[]**.

Une des conséquences de cette particularité est que les vecteurs ne peuvent pas contenir de primitives. Pour contourner cette limitation, il faut utiliser les *enveloppeurs*, qui sont des classes spécialement conçues à cet effet : **Integer**, **Boolean**, **Float**, etc.

L'autre particularité est que vous devez effectuer un sous-casting explicite pour utiliser les objets contenus dans un vecteur. Cependant, contrairement au cas étudié précédemment avec les tableaux, ce sous-casting ne pose pas de problème si votre vecteur ne contient que des objets d'un type connu, ce qui est le cas le plus fréquent. De plus, Java surveille les sous-castings et ne vous laissera pas effectuer une telle opération vers un type ne correspondant pas au type de l'objet concerné.

Un vecteur peut être créé de plusieurs façons. En effet, la classe **Vector** dispose de quatre constructeurs :

```
Vector()
Vector(int initialCapacity)
Vector(int initialCapacity, int capacityIncrement)
Vector(Collection c)
```

La première version crée un vecteur de taille 10 et d'incrément 0. L'incrément d'un vecteur est la quantité dont sa taille sera augmentée lorsqu'il sera plein. Si l'incrément est 0, la taille est doublée.

La deuxième version permet de créer un vecteur en indiquant sa capacité initiale, alors que la troisième permet de spécifier l'incrément.

La quatrième version crée un vecteur de la taille nécessaire pour contenir les éléments de l'objet de type **Collection** qui lui est passé en argument. Nous parlerons des collections dans une prochaine section.

Vous pouvez connaître la longueur d'un vecteur en utilisant l'accesseur **size()** (qui, incidemment, aurait dû s'appeler **getSize**). (Souvenez-vous que les variables ne doivent pas être accessibles directement de l'extérieur, mais seulement à travers une méthode spéciale appelée *accesseur*. Dans le cas des tableaux, la longueur est directement accessible car il s'agit d'une constante.)

Vous pouvez également modifier la taille d'un vecteur à tous moments en utilisant la méthode **setSize()**. La classe **Vector** dispose de nombreuses autres méthodes permettant de gérer la taille et de manipuler les éléments. Pour

plus de détails, vous pouvez vous reporter à la documentation en ligne fournie avec le JDK.

Note : La classe **Vector** appartient au package **java.util**.

L'ajout d'un élément s'effectue au moyen de la méthode **addElement()**. Pour obtenir un élément correspondant à une position, il suffit d'utiliser la méthode **elementAt()** en fournissant, en paramètre, l'indice de l'élément recherché. Pour modifier un élément, vous devez employer la méthode **setElementAt()**.

L'exemple suivant montre le programme **MesAnimaux2** réécrit pour utiliser des vecteurs :

```
import java.util.*;
class MesAnimauxV2 {
   public static void main(String[] args) {
      Vector menagerie = new Vector(3);

      menagerie.addElement(new Chien());
      menagerie.addElement(new Chat());
      menagerie.addElement(new Canari());
      ((Animal)menagerie.elementAt(0)).afficheType();
      ((Animal)menagerie.elementAt(1)).afficheType();
      ((Animal)menagerie.elementAt(2)).afficheType();
      System.out.println();

      Vector menagerie2 = modifierEléments(menagerie);
      ((Animal)menagerie2.elementAt(0)).afficheType();
      ((Animal)menagerie2.elementAt(1)).afficheType();
      ((Animal)menagerie2.elementAt(2)).afficheType();
      System.out.println();

      ((Animal)menagerie.elementAt(0)).afficheType();
      ((Animal)menagerie.elementAt(1)).afficheType();
      ((Animal)menagerie.elementAt(2)).afficheType();
   }
```

```
   static Vector modifierEléments(Vector a) {
      for (int i = 0; i < a.size(); i++) {
         a.setElementAt(new Canari(), i);
      }
      return a;
   }
}

abstract class Animal {
   public  void afficheType() {
      System.out.println("Animal");
   }
}

class Chien extends Animal {
   public  void afficheType() {
      System.out.println("Chien");
   }
}

class Chat extends Animal {
   public void afficheType() {
      System.out.println("Chat");
   }
}

class Canari extends Animal {
   public  void afficheType() {
      System.out.println("Canari");
   }
}
```

Notez le problème posé par le fait que les vecteurs ne contiennent que des objets :

```
((Animal)menagerie.elementAt(0)).afficheType();
```

Nous devons ici effectuer un sous-casting pour pouvoir utiliser la méthode de l'objet contenu par le vecteur. Remarquez que nous ne savons pas de quel type exact il s'agit. Nous effectuons donc un sous-casting vers la classe parente **Animal**. Faites bien attention aux parenthèses. L'opérateur du sous-casting est **(Animal)**. Il est préfixé à l'objet à sous-caster qui, ici, est **menagerie.elementAt(0)**. Il n'est pas nécessaire de placer la référence d'objet entre parenthèses, car le point (.) a priorité sur l'opérateur de sous-casting. Cette même priorité oblige à placer l'ensemble **((Animal)menagerie.elementAt(0))** entre parenthèses. Dans le cas contraire, le point suivant serait pris en compte avant le sous-casting et Java essaierait d'invoquer la méthode **afficheType()** d'un objet de type **Object**, ce qui provoquerait une erreur de compilation.

Le type *Stack*

Le type **Stack** (*pile* en français) correspond à un vecteur de type particulier. S'agissant d'une sous-classe de **Vector**, le type **Stack** possède les mêmes caractéristiques que celui-ci, plus un certain nombre de fonctionnalités spécifiques. Le principe d'une pile est d'offrir un accès uniquement au dernier élément ajouté, un peu comme dans un distributeur de bonbons, dans lequel les bonbons sont placés dans un tube au fond duquel se trouve un ressort. Le dernier bonbon introduit est le premier disponible. (C'est même le seul, comme avec la plupart des piles, alors qu'en Java, tous les éléments restent disponibles grâce à leur indice.)

Les méthodes spécifiques de la classe **Stack** sont les suivantes :

- **boolean empty()**, qui renvoie **true** si la pile est vide.
- **Object peek()**, qui renvoie le premier objet disponible sur la pile (le dernier à y avoir été placé), sans le retirer de celle-ci.
- **Object pop()**, qui renvoie le premier objet disponible sur la pile en le retirant de celle-ci.
- **Object push(Object o)**, qui place l'objet **o** sur la pile, où il devient le premier objet disponible, et renvoie celui-ci.

- **int search(Object o)**, qui renvoie la première position de l'objet argument dans la pile.

Note 1 : N'oubliez jamais que les objets renvoyés ou placés sur la pile sont en fait des handles.

Note 2 : Le type **Stack** étant implémenté à l'aide d'un vecteur, le premier objet disponible est en fait le dernier du vecteur.

Les **Stack** ont un seul constructeur sans argument.

Le type *BitSet*

Le type **BitSet** est une structure semblable à celle d'un vecteur dans lequel chaque élément serait représenté par un bit. (Pour autant, la classe **BitSet** ne dérive pas de la classe **Vector**, mais directement de la classe **Object**.) Les **BitSet** sont employés pour contenir de façon efficace des indicateurs logiques. Les méthodes qui permettent de consulter les bits renvoient des valeurs de type **boolean**. La création d'un **BitSet** se fait de deux façons :

```
BitSet()
BitSet(int i)
```

La première version crée un **BitSet** de 64 bits. La seconde crée un **BitSet** du nombre de bits indiqué par l'argument. Par défaut, tous les bits valent **false**.

La classe **BitSet** possède (entre autres) les méthodes suivantes :

- **boolean get(int index)** permet de lire la valeur d'un bit.
- **void set(int index)** donne au bit d'index **i** la valeur **true**.
- **void clear(int index)** donne au bit d'index **i** la valeur **false**.

- **void and(BitSet b)** effectue une opération logique ET entre ce **BitSet** et l'argument **b**.

- **void or(BitSet b)** effectue une opération logique OU entre ce **BitSet** et l'argument **b**.

- **void xor(BitSet b)** effectue une opération logique OU exclusif entre ce **BitSet** et l'argument **b**.

- **int hashCode()** retourne un hashcode pour ce **BitSet**. Un hashcode est un code numérique unique correspondant à une combinaison de bits. Deux **BitSet** différents ont obligatoirement un hashcode différent. Toute modification d'un **BitSet** modifie son hashcode.

- **int size()** renvoie la taille du **BitSet**.

Les tables (*Map*)

Les tables sont des structures dans lesquelles les objets, au lieu d'être accessibles à l'aide d'un index correspondant à leur position, le sont au moyen d'une clé. Lorsque vous cherchez une définition dans le dictionnaire, peu vous importe qu'elle soit la première, la dernière ou la 354e. Tout ce qui compte est qu'elle soit celle qui corresponde au mot dont vous voulez connaître le sens. Une structure de type **Map** permet ainsi de stocker des couples clé/valeur. La notion d'ordre n'y est pas pertinente.

En Java, le type **Map** est une interface qui doit être implémentée par des classes et n'offre donc qu'une structure de méthodes sans définitions. Les méthodes déclarées dans l'interface **Map** sont les suivantes :

void clear() Retire toutes les paires clé/valeur de la structure. Cette méthode peut, au choix, être définie pour exécuter l'opération correspondante ou pour renvoyer une condition d'erreur **UnsupportedOperationException** (opération non supportée).

boolean containsKey(Object clé) Renvoie **true** si la clé est trouvée.

boolean containsValue(Object valeur) Renvoie **true** si la valeur est trouvée (associée à une ou plusieurs clés).

Set entries() Renvoie le contenu de la table sous la forme d'un ensemble (**Set**). Le lien est dynamique. Toute modification apportée à l'ensemble est reflétée dans la table. (Les ensembles sont présentés un peu plus loin.)

boolean equals(Object o) Renvoie **true** si l'objet est égal à la structure **Map**.

Object get(Object key) Renvoie la valeur correspondant à la clé.

int hashCode() Renvoie le hashcode de la structure.

boolean isEmpty() Renvoie **true** si la structure est vide.

Set keySet() Renvoie l'ensemble des clés sous la forme d'un ensemble. Le lien est dynamique. Toute modification apportée à l'ensemble est reflétée dans la table.

Object put(Object clé, Object valeur) Ajoute une entrée associant la clé et la valeur. Cette méthode peut, au choix, être définie pour exécuter l'opération correspondante ou pour renvoyer une condition d'erreur **UnsupportedOperationException** (opération non supportée).

void putAll(Map m) Copie toutes les entrées de l'argument. Cette méthode peut, au choix, être définie pour exécuter l'opération correspondante ou pour renvoyer une condition d'erreur **UnsupportedOperationException** (opération non supportée).

Object remove(Object key) Supprime l'entrée correspondant à la clé. Cette méthode peut, au choix, être définie pour exécuter l'opération correspondante ou pour renvoyer une condition d'erreur **UnsupportedOperationException** (opération non supportée).

int size() Renvoie le nombre d'entrées.

Collection values() Renvoie le contenu de la structure sous forme de collection.

*Les tables ordonnées (*SortedMap*)*

Les tables ordonnées sont des tables dont les éléments sont triés dans un ordre croissant. Le tri des éléments est basé sur un objet d'un type spécial (**Comparator**) fourni comme argument lors de la création de la table ou, dans le cas où cet objet n'est pas fourni, sur la méthode **compareTo()** des éléments (qui doivent, dans ce cas, implémenter l'interface **Comparable**). L'interface **SortedMap** déclare les méthodes suivantes :

Comparator comparator() Renvoie le comparateur utilisé pour ordonner la table. Si aucun comparateur n'est utilisé (les éléments sont ordonnés en fonction de leur méthode **compareTo()**), la méthode renvoie **null**.

Object firstKey() Renvoie la première clé de la table.

SortedMap headMap(Object o) Renvoie une table ordonnée contenant les références aux objets strictement inférieurs à l'objet argument.

Object lastKey() Renvoie la dernière clé de la table.

SortedMap subMap(Object de, Object à) Renvoie une table ordonnée contenant les références aux objets compris entre l'objet **de** (inclus) et l'objet **à** (exclu).

SortedMap tailMap(Object o) Renvoie une table ordonnée contenant les références aux objets supérieurs ou égaux à l'objet argument.

Les classes implémentant l'interface **Map** sont : **AbstractMap**, **HashMap**, **HashTable** et **Attributes**.

L'interface **SortedMap** est implémentée par la classe **TreeMap**.

Le type *Hashtable*

Le type **Hashtable** est une implémentation de l'interface **Map**. Dans cette structure, le hashcode des objets est utilisé pour accélérer l'accès aux entrées. Seuls les objets définissant la méthode **hashcode()** et la méthode

equals() peuvent figurer dans une **Hashtable**. Un handle dont la référence est **null** ne peut donc être placé dans cette structure.

Une **Hashtable** peut être créée de quatre façons :

```
public Hashtable(int capacitéInitiale, float facteurDeRehashing)
public Hashtable(int initialCapacity)
public Hashtable()
public Hashtable(Map t)
```

La première version crée une **Hashtable** avec la capacité initiale et le facteur de rehashing spécifiés. Le facteur de rehashing est une valeur indiquant à partir de quel taux de remplissage la table est agrandie. Cette valeur doit être comprise entre 0 et 1. Pour une valeur de 0.7, par exemple, la table sera agrandie dès qu'elle sera remplie à 70 %. L'agrandissement de la table nécessite un recalcul des hashcodes.

La deuxième version crée une **Hashtable** avec la capacité initiale indiquée et un facteur de rehashing de 0.75.

La troisième version crée une table de la capacité par défaut et un facteur de rehashing de 0.75.

La quatrième version crée une table contenant les entrées de la structure **Map** fournie en argument.

Les collections

Le terme de collection désigne de façon générique un ensemble d'objets. En Java, il est courant de traiter un objet comme un représentant d'une classe ou d'une interface parente. L'interface **Collection** permet de traiter ainsi des ensembles génériques. Les ensembles d'objets peuvent être soumis à des contraintes. Si les objets sont ordonnés, il s'agit de **List**. Si les entrées sont uniques, il s'agit de **Set**. Si les deux contraintes sont réunies, il

s'agit de **SortedSet**. Enfin, une collection sans contraintes est appelée *Bag* (sac). L'interface **Collection** déclare les méthodes suivantes :

boolean add(Object o) Ajoute l'objet **o** à la collection. Retourne **true** si la collection est modifiée par l'opération, **false** dans le cas contraire. La définition de cette méthode dans les classes implémentant l'interface peut limiter les possibilités d'ajout d'un élément, par exemple en interdisant la duplication (cas des **Set**) ou certains types d'objets. Cette méthode peut, au choix, être définie pour exécuter l'opération correspondante ou pour renvoyer une condition d'erreur **UnsupportedOperationException** (opération non supportée).

boolean addAll(Collection c) Cette méthode est identique à la précédente, mais prend pour argument une collection dont tous les objets sont ajoutés.

void clear() Retire tous les éléments de la collection. Cette méthode peut, au choix, être définie pour exécuter l'opération correspondante ou pour renvoyer une condition d'erreur **UnsupportedOperationException** (opération non supportée).

boolean contains(Object o) Renvoie **true** si l'objet est contenu dans la collection.

boolean containsAll(Collection c) Méthode identique à la précédente, mais prenant pour argument une collection. Renvoie **true** si tous les objets de la collection argument sont présents dans la collection contenant la méthode.

boolean equals(Object o) Renvoie **true** si l'objet est égal à la collection.

int hashCode() Calcule un hashcode pour cette collection.

boolean isEmpty() Renvoie **true** si la collection est vide.

Iterator iterator() Renvoie un **Iterator** sur les éléments de la collection. Les itérateurs sont présentés dans une prochaine section.

boolean remove(Object o) Supprime la première instance de l'objet dans la collection. Renvoie **true** si la collection est modifiée par l'opération. Cette méthode peut, au choix, être définie pour exécuter l'opération correspon-

dante ou pour renvoyer une condition d'erreur **UnsupportedOperationException** (opération non supportée).

boolean removeAll(Collection c) Supprime de la collection contenant la méthode toutes les instances de tous les éléments présents dans la collection argument. Renvoie **true** si la collection est modifiée par l'opération. Le résultat de cette méthode est que les deux collections ne contiennent plus aucun élément commun. Cette méthode peut, au choix, être définie pour exécuter l'opération correspondante ou pour renvoyer une condition d'erreur **UnsupportedOperationException** (opération non supportée).

boolean retainAll(Collection c) Supprime de la collection contenant la méthode toutes les instances de tous les éléments absents de la collection argument. Renvoie **true** si la collection est modifiée par l'opération. Le résultat de cette méthode est que la collection appelante ne contient plus qu'un sous-ensemble des éléments de la collection argument. Cette méthode peut, au choix, être définie pour exécuter l'opération correspondante ou pour renvoyer une condition d'erreur **UnsupportedOperationException** (opération non supportée).

int size() Renvoie le nombre d'éléments dans la collection.

Object[] toArray() Renvoie un tableau contenant tous les éléments de la collection.

Object[] toArray(Object[] a) Renvoie un tableau contenant tous les éléments de la collection dont le type correspond à l'argument. Le résultat est placé dans le tableau argument si celui-ci est assez grand. Dans le cas contraire, un tableau de même type est créé.

*Les listes (*List*)*

Une liste est semblable à un vecteur ordonné, c'est-à-dire qu'un élément peut être ajouté à n'importe quelle position, et non seulement à la fin (comme dans un vecteur) ou au début (comme dans une pile). La structure **List** est réalisée, en Java, à l'aide d'une interface étendant l'interface **Collection** et ajoutant quatre méthodes nouvelles ainsi que des versions supplémentaires de certaines méthodes existantes :

void add(int index, Object o) Insère l'objet à la position indiquée. Cette méthode peut, au choix, être définie pour exécuter l'opération correspondante ou pour renvoyer une condition d'erreur **UnsupportedOperationException** (opération non supportée).

boolean addAll(int index, Collection c) Insère tous les éléments de la collection à partir de la position indiquée. Cette méthode peut, au choix, être définie pour exécuter l'opération correspondante ou pour renvoyer une condition d'erreur **UnsupportedOperationException** (opération non supportée).

Object get(int index) Renvoie l'objet se trouvant à la position indiquée.

int indexOf(Object o) Renvoie l'index de la première occurrence de l'objet argument, ou -1 si l'objet est absent.

int indexOf(Object o, int index) Renvoie l'index de la première occurrence de l'objet argument à partir de l'index indiqué, ou -1 si aucune occurrence de l'objet n'est trouvée à partir de cette position.

int lastIndexOf(Object o) Renvoie l'index de la dernière occurrence de l'objet argument, ou -1 si l'objet est absent.

int lastIndexOf(Object o, int index) Renvoie l'index de la dernière occurrence de l'objet argument avant la position indiquée, ou - 1 si l'objet est absent.

ListIterator listIterator() Renvoie un itérateur de type **ListIterator** sur les éléments de la liste. Les itérateurs sont présentés dans une prochaine section.

ListIterator listIterator(int index) Renvoie un itérateur de type **ListIterator** sur les éléments de la liste à partir de l'élément indiqué.

Object remove(int index) Renvoie l'objet se trouvant à la position indiquée en le supprimant de la liste.

void removeRange(int de, int à) Retire de la liste tous les éléments dont la position est comprise entre **de** (inclus) et **à** (exclu).

L'interface **List** est implémentée par les classes **AbstractList**, **LinkedList**, **Vector** et **ArrayList**.

*Les ensembles (*Set*)*

Un ensemble est une collection non ordonnée qui ne peut contenir qu'un seul exemplaire de chacun de ses éléments. L'unicité des objets est basée sur la valeur renvoyée par la méthode **equals()**. Deux objets **o1** et **o2** ne peuvent se trouver dans le même ensemble si **o1.equals(o2)** renvoie **True**. De plus, un ensemble ne peut contenir qu'un seul handle s'évaluant à **null**.

La notion d'"un seul exemplaire" est quelque peu trompeuse. En effet, la méthode **equals()** de la classe **Object** renvoie **true** si et seulement si les deux handles comparés font référence au même objet. En revanche, cette méthode peut être redéfinie dans d'autres classes. La méthode **equals()** peut être redéfinie de façon quelconque à condition de respecter les règles suivantes :

- Elle doit être réflexive : pour tout **x**, **x.equals(x)** doit renvoyer **True**.
- Elle doit être symétrique : quels que soient **x** et **y**, si **x.equals(x)** renvoie **True**, alors **y.equals(x)** doit renvoyer **True**.
- Elle doit être transitive : quels que soient **x**, **y** et **z**, si **x.equals(y)** renvoie **True** et si **y.equals(z)** renvoie **True**, alors **x.equals(z)** doit renvoyer **True**.
- Elle doit être cohérente : quels que soient **x** et **y**, **x.equals(y)** doit toujours renvoyer la même valeur.
- Quel que soit **x**, **x.equals(null)** doit renvoyer **False**.

Les objets dont la valeur peut changer (relativement à leur méthode **equals()**) posent un problème particulier. Si la valeur d'un tel objet est modifiée pendant que l'objet fait partie de l'ensemble, l'état de l'ensemble devient indéfini.

L'interface **Set** est implémentée par les classes **AbstractSet** et **HashSet**.

*Les ensembles ordonnés (*SortedSet*)*

Les **SortedSet** sont des ensembles dont les éléments sont ordonnés. Le tri des éléments est basé sur un objet d'un type spécial (**Comparator**) fourni comme argument lors de la création de l'ensemble, ou, dans le cas où cet objet n'est pas fourni, sur la méthode **compareTo()** des éléments (qui doivent, dans ce cas, implémenter l'interface **Comparable**). L'interface **SortedSet** déclare les méthodes suivantes :

Comparator comparator() Renvoie le comparateur utilisé pour ordonner l'ensemble. Si aucun comparateur n'est utilisé (les éléments sont ordonnés en fonction de leur méthode **compareTo()**), la méthode renvoie **null**.

Object first() Renvoie le premier élément de l'ensemble.

SortedSet headSet(Object o) Renvoie un **SortedSet** contenant les références aux objets strictement inférieurs à l'objet argument.

Object last() Renvoie le dernier objet de l'ensemble.

SortedSet subSet(Object de, Object à) Renvoie un sous-ensemble contenant les références aux objets compris entre l'objet **de** (inclus) et l'objet **à** (exclu).

SortedSet tailSet(Object o) Renvoie un **SortedSet** contenant les références aux objets supérieurs ou égaux à l'objet argument.

L'interface **SortedSet** est implémentée par la classe **TreeSet**.

Les itérateurs

Il est parfois utile de traiter une **Collection** de façon générique, sans se préoccuper de savoir s'il s'agit d'une liste, d'une pile, d'un vecteur ou d'un ensemble. On utilise pour cela un *Itérateur*, représenté par l'interface **Iterator**.

L'interface **Iterator** déclare trois méthodes :

boolean hasNext() Renvoie **true** si l'itérateur contient encore des éléments.

Object next() Renvoie l'élément suivant.

void remove() Supprime de la collection le dernier élément renvoyé par l'itérateur.

Un itérateur n'est pas une copie d'une collection, mais seulement une vue différente. Toute modification apportée à un objet de l'itérateur est reflétée dans la collection correspondante.

Le programme suivant montre un exemple d'utilisation d'un itérateur :

```
import java.util.*;
class Iterateur {
   public static void main(String[] args) {

      Vector menagerie = new Vector(5);

      menagerie.addElement(new Chien());
      menagerie.addElement(new Chat());
      menagerie.addElement(new Canari());
      menagerie.addElement(new Chat());
      menagerie.addElement(new Chien());

      Iterator it = menagerie.iterator();
      while (it.hasNext()) {
         ((Animal)it.next()).afficheType();
      }
   }
}

abstract class Animal {
   public  void afficheType() {
      System.out.println("Animal");
   }
}
```

```
class Chien extends Animal {
   public  void afficheType() {
      System.out.println("Chien");
   }
}

class Chat extends Animal {
   public void afficheType() {
      System.out.println("Chat");
   }
}

class Canari extends Animal {
   public  void afficheType() {
      System.out.println("Canari");
   }
}
```

L'avantage des itérateurs est qu'ils permettent d'accéder à tous les éléments d'une collection sans se préoccuper du type réel de celle-ci. De cette façon, il est possible de modifier le type de structure utilisé pour stocker les données dans un programme en limitant au minimum les modifications à apporter à celui-ci, et en localisant ces modifications à un seul endroit : celui où la structure est manipulée. En revanche, la manipulation des éléments de la structure n'est pas remise en question, ce qui facilite beaucoup la maintenance des programmes.

Les itérateurs de listes

Le type **List** dispose d'un itérateur particulier de type **ListIterator**. Cette interface définit des méthodes supplémentaires permettant (entre autres) de parcourir la liste en ordre descendant :

void add(Object o) Insère l'objet argument dans la liste immédiatement avant l'élément renvoyé par **next()** et après l'élément renvoyé par **previous()**. L'indice n'est pas modifié, ce qui fait qu'après l'insertion, **getNext()** renvoie l'élément qui vient d'être inséré.

boolean hasNext() Comme dans **Iterator**.

boolean hasPrevious() Renvoie **true** s'il existe un élément précédent.

Object next() Comme dans **Iterator**.

int nextIndex() Renvoie l'index de l'élément qui serait renvoyé par le prochain appel de **next()**, ou -1 s'il n'existe aucun élément suivant.

Object previous() Renvoie l'élément précédent.

int previousIndex() Renvoie l'index de l'élément qui serait renvoyé par le prochain appel de **previous()**, ou -1 s'il n'existe aucun élément précédent.

void remove() Supprime de la liste l'élément renvoyé par le dernier appel de **next()** ou **previous()**.

void set(Object o) Remplace l'élément renvoyé par le dernier appel de **next()** ou **previous()** par l'objet argument.

Les comparateurs

Les structures ordonnées classent leurs éléments de deux façons : selon leur "ordre naturel", c'est-à-dire en utilisant le résultat de leur méthode **compareTo()**, ou à l'aide d'un *comparateur*, c'est-à-dire d'un objet implémentant l'interface **Comparator**.

Note : Pour que des objets puissent être classés selon l'"ordre naturel", il est nécessaire qu'ils implémentent l'interface **Comparable**, qui déclare la méthode **compareTo()**. Ils doivent évidemment aussi définir cette méthode de façon à fournir soit un *ordre total* (il ne peut y avoir d'égalité) soit un *ordre partiel*.

L'utilité des comparateurs vient de ce que l'ordre naturel ne convient pas toujours à certaines structures. Par exemple, certains objets peuvent disposer d'une méthode **compareTo()** donnant un ordre partiel alors qu'une structure peut nécessiter un ordre total.

La méthode :

```
int compareTo(Object o)
```

renvoie une valeur négative si l'argument est plus petit, nulle s'il est égal et positive s'il est plus grand. Il faut noter que si :

```
x.equals(y) == True
```

alors :

```
x.compareTo(y) == 0
```

En revanche :

```
x.compareTo(y) == 0
```

n'implique pas :

```
x.equals(y) == True
```

Si on peut avoir **x.compareTo(y) == 0** et **x.equals(y) == False**, l'ordre n'est pas total. Dans ce cas, si l'on veut un ordre total (ou si l'on veut simplement utiliser un critère de tri différent), il faut utiliser un comparateur.

L'interface **Comparator** déclare simplement la méthode :

```
int compare(Object o1, Object o2)
```

Cette méthode doit renvoyer une valeur négative pour indiquer que le premier argument est plus petit que le second, une valeur nulle si les deux arguments sont égaux et une valeur positive si le second est plus petit que le premier. De plus, elle doit satisfaire aux conditions suivantes :

- Quels que soient **x** et **y** :

```
sgn(compare(x, y)) == -sgn(compare(y, x))
```

- Quels que soient **x**, **y** et **z** :

```
((compare(x, y)>0) && (compare(y, z)>0))
```

implique :

```
compare(x, z) > 0
```

- Quels que soient **x** et **y** :

```
x.equals(y)||(x==null && y==null)
```

implique :

```
compare(x, y) == 0
```

- Quels que soient **x**, **y** et **z** :

```
compare(x, y) == 0
```

implique :

```
sgn(compare(x, z)) == sgn(compare(y, z))
```

Le programme suivant montre un exemple de **TreeSet**, structure implémentant l'interface **SortedSet**, créé en appelant le constructeur prenant un **Comparator** comme argument. L'ensemble (**TreeSet**) reçoit des caractères qui doivent être triés en ordre alphabétique sans tenir compte des ma-

juscules ou des signes diacritiques (cédilles et accents). L'interface **Comparator** est implémentée par la classe **Comparaison**. La méthode **compare()** utilise une version modifiée de la classe **Conversion** (développée dans un chapitre précédent) pour convertir les caractères en majuscules non accentuées. Notez que cette méthode est incomplète car elle ne prend pas en compte le cas où les arguments (qui sont des objets quelconques) ne sont pas des instances de la classe **Lettre**. Le traitement d'erreur dans ce cas fait appel à des techniques que nous n'avons pas encore abordées.

La classe **Lettre** est un enveloppeur pour le type **char** car les ensembles, comme les autres structures de données à l'exception des tableaux, ne peuvent contenir que des objets. Nous aurions pu tout aussi bien utiliser la classe **Character** qui fait partie du package **java.lang**. La classe **Lettre** contient une méthode supplémentaire permettant de convertir les caractères Unicode utilisés par Java en caractères ASCII étendus permettant l'affichage dans une fenêtre DOS.

```
import java.util.*;
class Comparateur {
   public static void main(String[] args) {
      TreeSet lettres = new TreeSet(new Comparaison());
      lettres.add(new Lettre('C'));
      lettres.add(new Lettre('ç'));
      lettres.add(new Lettre('S'));
      lettres.add(new Lettre('u'));
      lettres.add(new Lettre('c'));
      lettres.add(new Lettre('é'));
      lettres.add(new Lettre('e'));
      lettres.add(new Lettre('R'));
      lettres.add(new Lettre('ù'));

      Iterator it = lettres.iterator();
      while (it.hasNext()) {
         ((Lettre)it.next()).affiche();
      }
   }
}
```

```
class Lettre {
   char valeur;
   Lettre (char v) {
      valeur = v;
   }

   public void affiche() {
      System.out.println(win2DOS(valeur));
   }

   public static char win2DOS(char c) {
      switch (c) {
         case 'à':
            return (char)133;
         case 'â':
            return (char)131;
         case 'ä':
            return (char)132;
         case 'é':
            return (char)130;
         case 'è':
            return (char)138;
         case 'ë':
            return (char)137;
         case 'ê':
            return (char)136;
         case 'ï':
            return (char)139;
         case 'î':
            return (char)140;
         case 'ô':
            return (char)147;
         case 'ö':
            return (char)148;
         case 'ü':
            return (char)129;
         case 'û':
            return (char)150;
```

```
            case 'ù':
               return (char)151;
            case 'ç':
               return (char)135;
         }
            return c;
         }
}

class Comparaison implements Comparator {
   int retval = 0;
   public int compare(Object o1, Object o2) {
      if ((o1 instanceof Lettre) && (o2 instanceof Lettre)) {
         if (Conversion.conv(((Lettre)o1).valeur) <
                  Conversion.conv(((Lettre)o2).valeur))
            retval = -1;
         else if (Conversion.conv(((Lettre)o1).valeur) >
                  Conversion.conv(((Lettre)o2).valeur))
            retval = 1;
      }
      else {
         // traitement d'erreur
      }
      return retval;
   }
}

class Conversion {
   static int conv (char c) {
      switch (c) {
         case 'à':
            return 'A' * 10 + 2;
         case 'â':
            return 'A' * 10 + 3;
         case 'ä':
            return 'A' * 10 + 4;
         case 'é':
            return 'E' * 10 + 2;
```

```
            case 'è':
               return 'E' * 10 + 3;
            case 'ë':
               return 'E' * 10 + 4;
            case 'ê':
               return 'E' * 10 + 5;
            case 'ï':
               return 'I' * 10 + 2;
            case 'î':
               return 'I' * 10 + 3;
            case 'ô':
               return 'O' * 10 + 2;
            case 'ö':
               return 'O' * 10 + 3;
            case 'ü':
               return 'U' * 10 + 2;
            case 'û':
               return 'U' * 10 + 3;
            case 'ù':
               return 'U' * 10 + 4;
            case 'ç':
               return 'C' * 10 + 2;
            default:
               if (c > 96)
                  return (c - 32) * 10 + 1;
               else
                  return c * 10;
         }
      }
   }
```

Note : Ce programme n'est pas d'une grande utilité. En effet, la classe **TreeSet** n'est supposée fonctionner correctement que si l'ordre naturel ou celui fourni par le comparateur est total (deux éléments ne peuvent être égaux). Il n'est donc pas possible d'utiliser un comparateur renvoyant une égalité (0) pour les caractères avec et sans accent, ou majuscules et minuscules. Si vous utilisez un comparateur ne fournissant pas un ordre total, les

résultats seront incohérents. Certains objets seront ajoutés alors qu'un objet égal est déjà présent, d'autres non. Nous verrons plus loin comment effectuer des tris avec un ordre partiel.

La version suivante du programme donne le même résultat en utilisant l'*ordre naturel*, c'est-à-dire celui fourni par la méthode **compareTo()** des objets ajoutés à la structure :

```
import java.util.*;
class OrdreNaturel {
   public static void main(String[] args) {

      TreeSet lettres = new TreeSet();

      lettres.add(new Lettre('r'));
      lettres.add(new Lettre('ç'));
      lettres.add(new Lettre('S'));
      lettres.add(new Lettre('u'));
      lettres.add(new Lettre('c'));
      lettres.add(new Lettre('é'));
      lettres.add(new Lettre('e'));
      lettres.add(new Lettre('R'));
      lettres.add(new Lettre('ù'));

      Iterator it = lettres.iterator();
      while (it.hasNext()) {
         ((Lettre)it.next()).affiche();
      }
   }
}

class Lettre implements Comparable {
   char valeur;
   Lettre (char v) {
      valeur = v;
   }
```

```
   public void affiche() {
      System.out.println(win2DOS(valeur));
   }

   public int compareTo(Object o) {
      int retval = 0;
      if (o instanceof Lettre) {
         if (Conversion.conv(((Lettre)o).valeur) <
                                 Conversion.conv(valeur))
            retval = 1;
         else if (Conversion.conv(((Lettre)o).valeur) >
                                 Conversion.conv(valeur))
            retval = -1;
      }
      else {
         // traitement d'erreur
      }
      return retval;
   }

   public static char win2DOS(char c) {
      // identique à la version précédente
   }
}


class Conversion {
   static char conv (char c) {
      // identique à la version précédente
   }
}
```

Les méthodes de la classe *Collections*

La classe **java.util.Collections** (à ne pas confondre avec l'interface **java.util.Collection**, avec laquelle il n'existe aucun lien hiérarchique) contient dix-neuf méthodes statiques qui manipulent ou renvoient des **Collec-**

tion. Les méthodes qui renvoient des collections sont appelées *vues* car elles opèrent sur des collections dont elles renvoient une vue déterminée en fonction d'une caractéristique particulière. Il n'y a en effet jamais de duplication des éléments des collections manipulées.

static int binarySearch(List liste, Object clé) Recherche la clé dans la liste en utilisant l'algorithme de recherche dichotomique. Toutes les contraintes de cet algorithme s'appliquent ici. En particulier, la liste doit être triée. L'utilisation d'une liste non triée entraîne un résultat imprévisible et peut conduire à une boucle infinie. Par ailleurs, si la liste contient des objets "égaux" (selon le critère employé par leur méthode **compareTo()**), l'objet trouvé est celui dont le chemin de recherche produit par l'algorithme de recherche dichotomique est le plus court, ce qui ne garantit en aucun cas qu'il s'agisse du premier. Pour une liste indexée de *n* éléments (un vecteur, par exemple), cette méthode offre un temps d'accès proportionnel à log(*n*) (quasi constant pour les valeurs élevées de *n*).

static int binarySearch(List liste, Object clé, Comparator c) Effectue la même recherche que la méthode précédente, mais en utilisant un Comparator au lieu de la méthode **compareTo()** des objets de la liste.

static Enumeration enumeration(Collection c) Produit une énumération à partir de la collection argument. Une énumération est une instance de l'interface **Enumeration**, qui possède des fonctionnalités réduites de l'interface **Iterator**. L'interface **Iterator** a remplacé **Enumeration** à partir de la version 2 de Java. Cette méthode n'est donc utile que pour les besoins de la compatibilité avec les versions précédentes.

static Object max(Collection c) Renvoie l'objet maximal contenu dans la collection en utilisant l'ordre naturel, c'est-à-dire celui fourni par la méthode **compareTo()** des objets de la collection.

static Object max(Collection c, Comparator comp) Renvoie l'objet maximal contenu dans la collection en utilisant l'ordre impliqué par la méthode **compare()** du comparateur.

static Object min(Collection c) Renvoie l'objet minimal contenu dans la collection en utilisant l'ordre naturel, c'est-à-dire celui fourni par la méthode **compareTo()** des objets de la collection.

static Object min(Collection c, Comparator comp) Renvoie l'objet minimal contenu dans la collection en utilisant l'ordre impliqué par la méthode **compare()** du comparateur.

static List nCopies(int n, Object o) Renvoie une liste non modifiable composée de **n** occurrences de l'objet **o**.

static void sort(List liste) Trie les éléments de la liste en fonction de leur ordre naturel. Il s'agit d'un tri stable, c'est-à-dire que l'ordre des éléments égaux n'est pas modifié.

static void sort(List liste, Comparator c) Trie les éléments de la liste en fonction de l'ordre impliqué par la méthode **compare()** du comparateur. Il s'agit d'un tri stable, c'est-à-dire que l'ordre des éléments égaux n'est pas modifié.

static List subList(List list, int de, int à) Renvoie une liste composée des éléments d'indice **de** (inclus) à **à** (exclu).

static Collection synchronizedCollection(Collection c) Renvoie une collection synchronisée contenant les mêmes éléments que la collection argument. Une collection synchronisée est protégée par le fait qu'elle ne peut être utilisée que par un seul processus à la fois, ce qui garantit qu'elle ne sera pas modifiée pendant qu'une opération est en cours. Les processus seront étudiés dans un prochain chapitre.

static List synchronizedList(List liste) Renvoie une liste synchronisée contenant les mêmes éléments que la liste argument.

static Map synchronizedMap(Map m) Renvoie une structure **Map** synchronisée contenant les mêmes éléments que la structure **Map** argument.

static Set synchronizedSet(Set s) Renvoie un ensemble synchronisé contenant les mêmes éléments que l'ensemble argument.

static SortedMap synchronizedSortedMap(SortedMap m) Renvoie une structure **SortedMap** synchronisée contenant les mêmes éléments que la structure **SortedMap** argument.

static SortedSet synchronizedSortedSet(SortedSet s) Renvoie un ensemble ordonné synchronisé contenant les mêmes éléments que l'ensemble ordonné argument.

static Collection unmodifiableCollection(Collection c) Renvoie une version non modifiable de l'argument.

static List unmodifiableList(List list) Renvoie une version non modifiable de l'argument.

static Map unmodifiableMap(Map m) Renvoie une version non modifiable de l'argument.

static Set unmodifiableSet(Set s) Renvoie une version non modifiable de l'argument.

static SortedMap unmodifiableSortedMap(SortedMap m) Renvoie une version non modifiable de l'argument.

static SortedSet unmodifiableSortedSet(SortedSet s) Renvoie une version non modifiable de l'argument.

Note : La classe **Collections** possède également le champ **static REVERSE_ORDER**, de type **Comparator** permettant de trier une collection d'objets implémentant l'interface **Comparable** en ordre inverse de l'ordre naturel.

Exemple : utilisation de la méthode sort()

L'exemple précédent de tri des caractères peut maintenant être réalisé de façon complète en utilisant un comparateur et la méthode **sort()** de la classe **Collections** :

```
import java.util.*;
class Tri {
   public static void main(String[] args) {
      Vector lettres = new Vector();
      lettres.add(new Lettre('C'));
      lettres.add(new Lettre('ç'));
      lettres.add(new Lettre('S'));
      lettres.add(new Lettre('u'));
      lettres.add(new Lettre('c'));
      lettres.add(new Lettre('é'));
      lettres.add(new Lettre('e'));
      lettres.add(new Lettre('R'));
      lettres.add(new Lettre('ù'));
      Collections.sort(lettres, new Comparaison());
      Iterator it = lettres.iterator();
      while (it.hasNext()) {
         ((Lettre)it.next()).affiche();
      }
   }
}
class Lettre {
   char valeur;
   Lettre (char v) {
      valeur = v;
   }
   public void affiche() {
      System.out.println(win2DOS(valeur));
   }
   public static char win2DOS(char c) {
      switch (c) {
         case 'à':
            return (char)133;
         case 'â':
            return (char)131;
         case 'ä':
            return (char)132;
         case 'é':
            return (char)130;
```

```
            case 'è':
               return (char)138;
            case 'ë':
               return (char)137;
            case 'ê':
               return (char)136;
            case 'ï':
               return (char)139;
            case 'î':
               return (char)140;
            case 'ô':
               return (char)147;
            case 'ö':
               return (char)148;
            case 'ü':
               return (char)129;
            case 'û':
               return (char)150;
            case 'ù':
               return (char)151;
            case 'ç':
               return (char)135;
         }
         return c;
      }
   }

   class Comparaison implements Comparator {
      int retval = 0;
      public int compare(Object o1, Object o2) {
         if ((o1 instanceof Lettre) && (o2 instanceof Lettre)) {
            if (Conversion.conv(((Lettre)o1).valeur) <
                        Conversion.conv(((Lettre)o2).valeur))
               retval = -1;
         else if (Conversion.conv(((Lettre)o1).valeur) >
                        Conversion.conv(((Lettre)o2).valeur))
            retval = 1;
         }
```

```
      else {
         // traitement d'erreur
      }
      return retval;
   }
}
class Conversion {
   static char conv (char c) {
      switch (c) {
         case 'à':
         case 'â':
         case 'ä':
            return 'A';
         case 'é':
         case 'è':
         case 'ë':
         case 'ê':
            return 'E';
         case 'ï':
         case 'î':
            return 'I';
         case 'ô':
         case 'ö':
            return 'O';
         case 'ü':
         case 'û':
         case 'ù':
            return 'U';
         case 'ç':
            return 'C';
         default:
            if (c > 96)
               return (char)(c - 32);
            else
               return c;
      }
   }
}
```

Résumé

Dans ce chapitre, nous n'avons fait qu'effleurer le sujet des structures de données, qui mériterait un livre à lui tout seul. Si vous souhaitez plus d'informations sur les classes et les interfaces disponibles pour la manipulation de structures complexes, nous vous conseillons de vous reporter à la documentation en ligne fournie avec le JDK.

Chapitre 11

Les objets meurent aussi

Au Chapitre 5, nous avons étudié tout ce qui concerne le commencement de la vie des objets. Nous allons maintenant nous intéresser à ce qui se passe à l'autre extrémité de la chaîne, lorsque les objets cessent leur existence.

Certains objets deviennent inaccessibles

Comme nous l'avons déjà dit, les objets sont créés dans une partie de la mémoire appelée *tas* (*heap*). Au moment de leur création, les objets sont créés en même temps qu'une référence permettant de les manipuler. Cette référence peut être un *handle*, comme dans le cas suivant :

```
Object monObjet;
monObjet = new Object();
```

L'objet peut également être référencé par un élément d'une structure comme dans le cas suivant :

```
Vector monVecteur;
monVecteur.addElement(new Object());
```

Ici, aucun handle n'est créé. La référence à l'objet est tout simplement un des éléments du vecteur (le dernier au moment de la création).

Un objet peut être créé sans qu'une référence explicite soit employée. La référence peut alors être définie dans un autre objet :

```
TreeSet lettres = new TreeSet(new Comparaison());
```

Ici, un objet de type **TreeSet** est créé et référencé par le handle **lettres**. En revanche, un objet de type **Comparaison** est créé sans référence. Cet objet est passé en argument au constructeur de la classe **TreeSet**. A ce moment, l'objet en question se trouve référencé par le handle déclaré comme argument du constructeur. Nous savons qu'il existe une version du constructeur de **TreeSet** ayant pour argument un handle pour un objet de type **Comparaison** ou d'un type parent (en l'occurrence **Comparator**).

L'objet ainsi créé peut également être passé à une méthode. Dans ce cas, il se trouve référencé par le handle correspondant déclaré dans les arguments de la méthode :

```
Collections.sort(lettres, new Comparaison());
```

La question qui peut se poser ici est de savoir ce que deviennent ensuite ces objets.

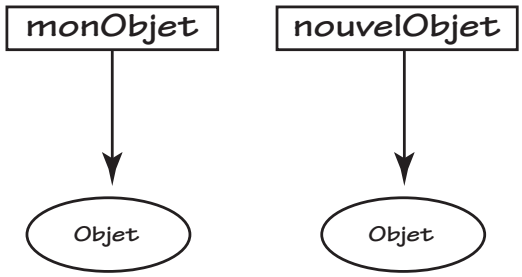
Dans le premier cas, l'objet existe de façon évidente tant que le handle correspondant lui est affecté. En revanche, considérez l'exemple suivant :

```
Object monObjet;
Object nouvelObjet;
```

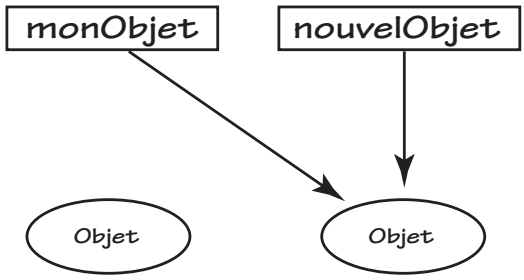
```
monObjet = new Object();
nouvelObjet = new Object();
monObjet = nouvelObjet();
```

Au moment où le handle **monObjet** est affecté à l'objet pointé par le handle **nouvelObjet**, le premier objet devient inaccessible. Le schéma suivant résume la situation :

```
Object monObjet;
Object nouvelObjet;
monObjet = new Object();
nouvelObjet = new Object();
```




```
monObjet = nouvelObjet();
```



A partir de ce moment, le premier objet créé, qui n'est plus référencé par aucun handle, n'est plus accessible. Pour autant, il n'est pas supprimé de la mémoire. Il reste simplement à sa place, en attendant qu'on s'occupe de lui.

Dans le deuxième exemple :

```
TreeSet lettres = new TreeSet(new Comparaison());
```

on ne peut pas à priori savoir ce que devient l'objet créé. Pour le savoir, il faut consulter les sources de la classe **TreeSet** (livrées avec le JDK), puis celles de la classe **TreeMap**, ce qui nous amène au code suivant :

```
public class TreeMap extends AbstractMap implements SortedMap,
                             Cloneable, java.io.Serializable {
   private Comparator comparator = null;
      .
      .
   public TreeMap(Comparator c) {
      this.comparator = c;
      .
      .
```

Nous savons maintenant que l'objet de type **Comparaison** créé est affecté à un membre de l'objet **TreeSet** référencé par le handle **lettres**.

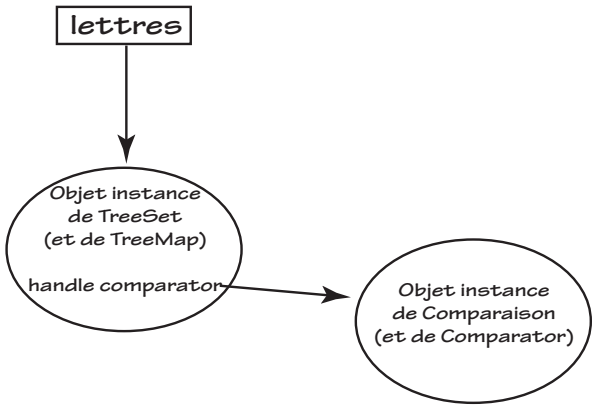
Que se passe-t-il si le lien entre le handle **lettres** et l'objet correspondant est coupé, par exemple dans le cas suivant :

```
lettres = null;
```

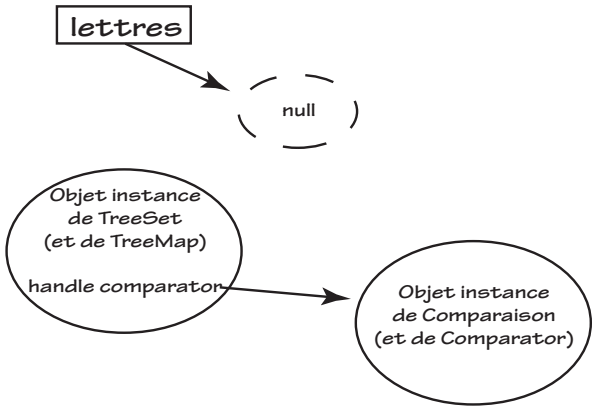
Le schéma de la page ci-contre représente la situation correspondante.

L'objet de type **TreeSet** n'est plus référencé et n'est donc plus accessible. L'objet de type **Comparaison** est toujours référencé. Il n'est cependant plus utilisable puisque l'objet qui le référence ne l'est pas lui-même.

```
TreeSet lettres = new TreeSet(new Comparaison());
```




```
lettres = null;
```

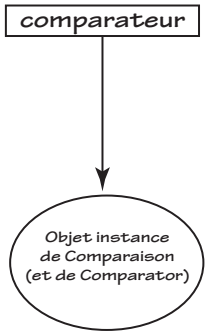


Notez qu'une situation différente peut se produire si nous écrivons le code de la façon suivante :

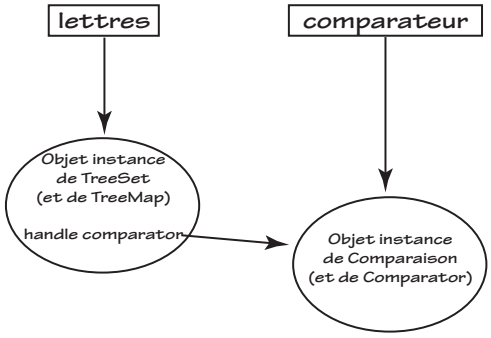
```
Comparaison comparateur = new Comparaison();
TreeSet lettres = new TreeSet(comparateur);
lettres = null;
```

Dans ce cas, la situation est la suivante :

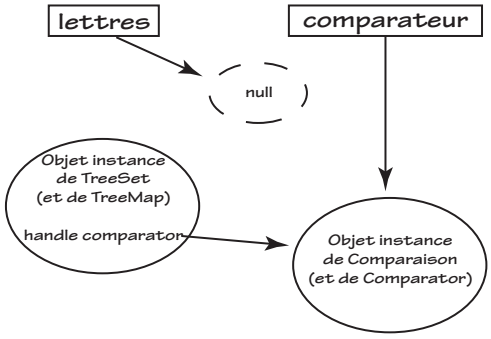
```
Comparaison comparateur = new Comparaison();
```




```
TreeSet lettres = new TreeSet(new Comparaison());
```




```
lettres = null;
```



Dans ce cas, l'objet de type **TreeSet** n'est plus accessible. En revanche, l'objet de type **Comparaison** l'est toujours.

Dans le troisième cas :

```
Collections.sort(lettres, new Comparaison());
```

l'objet de type **Comparaison** est passé à la méthode statique **sort** de la classe **Collections**. Si nous examinions le code source de cette méthode, nous verrions que l'objet est affecté à un handle (**c**) puis passé en argument à la méthode statique **sort** de la classe **Arrays** (après que **lettres** a été converti en tableau). L'objet est ensuite passé à la méthode **MergeSort** qui, elle, ne le passe à aucune autre méthode. Aucune de ces méthodes ne retourne l'objet. Lorsque **Collections.sort** retourne, l'objet n'est donc plus accessible.

Que deviennent les objets inaccessibles ?

La question est importante. En effet, ces objets occupent de la place en mémoire. Si l'on n'y prend garde, la mémoire risque de devenir entièrement pleine, ce qui entraînerait un blocage de l'ordinateur.

On entend souvent dire que Java s'occupe automatiquement de ce problème contrairement à d'autres langages, qui obligent le programmeur à s'en préoccuper. Pour cette raison, certains pensent qu'il n'est pas nécessaire, en Java, de s'inquiéter de ce que deviennent les objets dont on n'a plus besoin. La réalité est tout autre.

En fait, Java ne dispose, d'après ses spécifications, d'aucun moyen de résoudre le problème. Il se trouve que toutes les versions existantes de Java disposent d'un mécanisme prévu à cet effet. Cependant, il faut savoir que cela n'est pas obligatoire. Vous pouvez très bien rencontrer un jour une implémentation de Java n'en disposant pas. Toutes les versions actuelles de Java en étant munies, nous ferons comme si cela était systématique. Les

différentes JVM peuvent toutefois disposer de mécanismes totalement différents. Nous décrirons ici celui de la JVM de Sun Microsystems.

Le garbage collector

Le mécanisme qui permet à Java de récupérer la mémoire occupée par les objets devenus inutiles est appelé *garbage collector*, ce qui signifie "ramasseur de déchets". Ce mécanisme fonctionne grâce à un processus plus ou moins indépendant de votre programme et selon deux modes : synchrone et asynchrone.

Principe du garbage collector

Le rôle du garbage collector consiste non seulement à récupérer la mémoire occupée par les objets devenus inaccessibles, mais également à compacter la mémoire disponible. En effet, si la mémoire était simplement libérée, cela laisserait des "trous" inoccupés entre les espaces occupés par les objets valides. Lors de la création de nouveaux objets, la JVM devrait rechercher un trou de taille suffisante pour y placer ceux-ci. Certaines JVM peuvent fonctionner de cette façon. Leurs performances en matière de gestion de la mémoire en sont fortement diminuées.

En matière de récupération de la mémoire, il existe de nombreux algorithmes. Les spécifications de Java ne préconisent pas une approche plutôt qu'une autre. Pour simplifier, nous dirons que la JVM de Sun utilise une approche consistant à parcourir tous les objets se trouvant en mémoire à partir des références disponibles au niveau "racine" et à marquer ceux qui sont accessibles. Lors d'une session ultérieure, les objets non marqués peuvent être éliminés. Cette approche permet d'éliminer les objets disposant de références circulaires (l'objet A possède un membre b qui fait référence à l'objet B ; l'objet B possède un membre a qui fait référence à l'objet A).

Une partie du travail du garbage collector (la recherche et le marquage des objets) est effectuée de manière asynchrone, sans qu'il soit possible de le contrôler. Les seuls contrôles possibles consistent à :

- Empêcher ce fonctionnement asynchrone ;
- Déclencher le fonctionnement synchrone, ce qui a bien entendu un effet sur le processus qui vient d'être décrit.

L'autre partie du travail du garbage collector (libération de la mémoire et compactage) se déroule de façon synchrone. Il est déclenché par deux conditions :

- La diminution des ressources mémoire au-delà d'une certaine limite ;
- L'appel explicite du garbage collector par le programmeur.

Lorsque le garbage collector démarre son processus synchrone, deux cas peuvent se présenter :

- Le fonctionnement asynchrone était autorisé et a eu lieu. Les objets sont donc déjà marqués ;
- Le fonctionnement asynchrone n'était pas autorisé. Les objets ne sont donc pas encore marqués.

Dans le second cas, le garbage collector recherche tous les objets devenus inaccessibles (sans référence) et les marque comme candidats à la destruction. Dans le premier cas, cette étape est inutile. Puis, la méthode **finalize()** de chacun de ces objets est exécutée et les objets sont marqués pour indiquer qu'ils ont été "finalisés". (Nous reviendrons dans une prochaine section sur la méthode **finalize()**.) Et c'est tout ! Ce n'est qu'au prochain démarrage du garbage collector que les objets finalisés sont enfin détruits et que la mémoire est libérée et compactée. Par la même occasion, les objets qui seraient devenus inaccessibles lors de l'étape précédente sont traités à leur tour, c'est-à-dire que leur méthode **finalize()** est exécutée, et ainsi de suite. Le garbage collector s'arrête lorsqu'une quantité suffisante de mémoire a été libérée.

De ce principe de fonctionnement, on peut déduire les conclusions suivantes :

- Le programmeur n'a aucun moyen d'empêcher le garbage collector de démarrer. En revanche, l'utilisateur peut empêcher le garbage collector de fonctionner de manière asynchrone, grâce à un paramètre de la ligne de commande. De cette façon, un programme peut contrôler un processus en temps réel sans risquer d'être ralenti. Il faut seulement s'assurer que le processus synchrone ne démarrera pas suite à un encombrement de la mémoire.

- Il est possible qu'un programme se termine sans que le garbage collector ait jamais démarré. Dans ce cas, les méthodes **finalize()** des objets devenus inaccessibles ne seront jamais exécutées.

- Il n'y a aucun moyen de déterminer l'ordre dans lequel les objets dont les références sont de même nature sont finalisés (sauf s'il s'agit d'objets contenus dans d'autres objets, les contenants étant alors finalisés avant les contenus).

- Il existe de nombreuses façons de faciliter et d'optimiser le travail du garbage collector et la gestion de la mémoire. L'utilisation d'un garbage collector garantit qu'il n'y aura pas de *fuites de mémoire* dues à la persistance d'objets inaccessibles. Elle ne garantit pas que tous les objets inutiles seront rendus inaccessibles, ni que les objets seront créés de la manière la plus efficace.

Optimiser le travail du garbage collector

Une façon d'optimiser la gestion de la mémoire est de s'assurer que les objets devenus inutiles sont immédiatement reconnaissables comme tels par le garbage collector. Considérons l'exemple suivant dans lequel un objet est créé sans handle :

```
g.setColor(new Color(223, 223, 223));
g.drawLine(545, 0, 545, 338);
```

Dans cet exemple, un objet de type **Color** est créé à la première ligne afin de servir de paramètre à la méthode **setColor** qui détermine la couleur qui

sera utilisée pour tracer une ligne. Cet objet devient inaccessible dès la fin de la ligne suivante. En revanche, dans l'exemple suivant :

```
couleur = new Color(223, 223, 223);
g.setColor(couleur);
g.drawLine(545, 0, 545, 338);
```

l'objet créé reste accessible grâce au handle **couleur** et continue donc d'occuper de l'espace en mémoire.

Cette façon de faire peut être plus efficace, comme nous le verrons plus loin. Cependant, si vous n'y prenez pas garde, l'objet en question risque de traîner en mémoire beaucoup plus longtemps que nécessaire. Une façon de s'assurer que l'objet devient éligible pour le prochain passage du garbage collector consiste à supprimer explicitement le lien entre le handle et l'objet, au moyen de l'expression :

```
couleur = null;
```

Cette méthode est la plus efficace si vous devez réutiliser l'objet en question à plusieurs reprises. Le handle permet de le conserver en mémoire et de le réutiliser autant de fois que nécessaire. Lorsqu'il est devenu inutile, vous pouvez vous en débarrasser au moyen de l'instruction ci-dessus. Souvenez-vous toutefois que l'objet ainsi traité peut rester en mémoire un certain temps, puisqu'il faut au moins deux passages du garbage collector pour l'éliminer. Ainsi, si votre programme comporte plusieurs tracés de couleurs différentes, comme dans l'exemple suivant :

```
couleur = new Color(223, 223, 223);
g.setColor(couleur);
couleur = null;
g.drawLine(545, 0, 545, 338);
couleur = new Color(123, 12, 255);
g.setColor(couleur);
couleur = null;
g.drawLine(245, 0, 245, 338);
```

les deux objets de type **Color** qui ont été créés risquent de séjourner en mémoire un long moment avant d'être éliminés. L'utilisation d'objets anonymes, si elle simplifie l'écriture, ne résout pas le problème :

```
g.setColor(new Color(223, 223, 223));
g.drawLine(545, 0, 545, 338);
g.setColor(new Color(123, 12, 255));
g.drawLine(245, 0, 245, 338);
```

Après l'exécution de ces lignes, nous nous retrouvons dans la même situation, avec deux objets inaccessibles mais occupant de l'espace mémoire jusqu'à leur élimination. De plus, la création d'objets est une opération longue et pénalisante en ce qui concerne les performances.

Chaque fois que cela est possible, il est préférable de recycler un objet existant en modifiant ses caractéristiques plutôt que d'en créer un nouveau. C'est malheureusement impossible pour les instances de la classe **Color**. En revanche, cela peut être le cas pour les classes que vous créez vous-même, comme nous l'avons vu au Chapitre 8.

Les finaliseurs

Tout comme il existe des initialiseurs qui permettent d'effectuer certains traitements lors de la création des objets, Java met à notre disposition des *finaliseurs*, qui sont exécutés avant la destruction des objets par le garbage collector. Il s'agit simplement d'une méthode dont le nom est **finalize()**. Le programme suivant met en évidence l'exécution d'une telle méthode :

```
public class Elevage2 {
   public static void main(String[] argv) {
      while (!Lapin.gc) {
         Lapin.créerLapin();
      }
   }
}
```

```
class Lapin {
   private static int nombre = 0;
   private int numero;
   static boolean gc = false;
   private Lapin() {
      numero = ++nombre;
      System.out.println("Creation du lapin no " + numero);
   }
   public void finalize() {
      System.out.println("Finalisation du lapin no " + numero);
      gc=true;
   }
   public static Lapin créerLapin() {
      return new Lapin();
   }
}
```

Ce programme crée des instances de **Lapin** sans référence tant que la variable statique **gc** vaut **false**. Chaque instance créée est immédiatement éligible pour le garbage collector. Lorsque la mémoire commence à être remplie, le garbage collector se met en route et trouve le premier **Lapin** sans référence. Cet objet disposant d'une méthode appelée **finalize()**, celle-ci est exécutée avant que l'objet soit supprimé. Cette méthode affiche un message et donne à la variable statique **gc** la valeur **true**, ce qui a pour effet d'arrêter la création de lapins.

L'exécution de ce programme affiche le résultat suivant :

```
      .
      .
      .
   Creation du lapin no 2330
   Creation du lapin no 2331
   Finalisation du lapin no 1
   Finalisation du lapin no 2
   Creation du lapin no 2332
   Finalisation du lapin no 3
```

(Les valeurs peuvent changer en fonction de la mémoire disponible.) On voit que, dans cet exemple, le premier objet est supprimé après que 2 331 ont été créés. Par ailleurs, on peut remarquer que le garbage collector a le temps de supprimer 3 objets avant que le programme soit arrêté. Si nous modifions la méthode **finalize()** pour la ralentir, par exemple en incluant une boucle vide avant que la variable **gc** soit modifiée :

```
public class Elevage2 {
   public static void main(String[] argv) {
      while (!Lapin.gc) {
         Lapin.créerLapin();
      }
   }
}

class Lapin {
   private static int nombre = 0;
   private int numero;
   static boolean gc = false;

   private Lapin() {
      numero = ++nombre;
      System.out.println("Creation du lapin no " + numero);
   }

   public void finalize() {
      System.out.println("Finalisation du lapin no " + numero);
      for (int i = 0;i < 10000000;i++) {}
      gc=true;
   }

   public static Lapin créerLapin() {
      return new Lapin();
   }
}
```

nous obtenons le résultat suivant :

```
.
.
.
Creation du lapin no 2330
Creation du lapin no 2331
Finalisation du lapin no 1
Creation du lapin no 2332
Creation du lapin no 2333
Creation du lapin no 2334
Creation du lapin no 2335
Creation du lapin no 2336
Finalisation du lapin no 2
```

qui montre que la création de quatre objets supplémentaires a pu avoir lieu pendant l'exécution de la méthode **finalize()**. En revanche, si la modification de la variable est effectuée au début de la méthode :

```
public void finalize() {
   gc=true;
   System.out.println("Finalisation du lapin no " + numero);
   for (int i = 0;i < 10000000;i++) {}
}
```

deux objets seulement sont finalisés avant que le programme soit arrêté :

```
.
.
.
Creation du lapin no 2330
Creation du lapin no 2331
Finalisation du lapin no 1
Creation du lapin no 2332
Finalisation du lapin no 2
```

On voit que, s'il est possible d'interférer avec le garbage collector par ces moyens, ce n'est toutefois pas d'une façon exploitable. On pourrait être tenté de pousser le contrôle un peu plus loin, par exemple de la façon suivante :

```
public class Elevage3 {
   public static void main(String[] argv) {
      while (true) {
         if (!Lapin.gc)
            Lapin.créerLapin();
      }
   }
}

class Lapin {
   private static int nombre = 0;
   private int numero;
   static boolean gc = false;

   private Lapin() {
      numero = ++nombre;
      System.out.println("Creation du lapin no " + numero);
   }

   public void finalize() {
      --nombre;
      if (nombre > 2000)
         gc = true;
      else
         gc = false;
      System.out.println("Finalisation du lapin no " + numero);
   }

   public static Lapin créerLapin() {
      return new Lapin();
   }
}
```

Ce programme tente de contrôler la création des lapins en maintenant leur nombre un peu au-dessous du maximum (ici entre 2 000 et le maximum, mais cette valeur doit être adaptée à chaque cas.) En fait, le fonctionnement obtenu est tout à fait aléatoire du fait de l'asynchronisme des processus. (Pour arrêter le programme, vous devrez taper les touches *Ctrl + C*.)

Il est intéressant également de remarquer que lorsque le programme s'arrête, de très nombreux objets n'ont pas vu leur méthode **finalize()** exécutée. Par ailleurs, les objets sont ici finalisés dans l'ordre dans lequel ils ont été créés, mais cela n'est absolument pas garanti par Java.

Contrôler le travail du garbage collector

Au-delà des techniques décrites dans les pages précédentes, il existe d'autres possibilités de contrôler la façon dont le garbage collector dispose des objets devenus "plus ou moins inutiles". En effet, si certains objets deviennent totalement inutiles, d'autres restent cependant "moyennement utiles", en ce sens que, si la mémoire est suffisante, on préférerait les conserver, sachant qu'ils sont susceptibles d'être réutilisés, alors qu'en cas de pénurie, on les sacrifierait toutefois volontiers.

Prenons, par exemple, le cas d'une application multimédia de type "livre électronique". Cette application peut comporter une page servant de table des matières. Chaque fois que l'utilisateur quitte une page pour en consulter une autre, les éléments de la page précédente peuvent être conservés en mémoire pour le cas où cette page serait de nouveau consultée. Cependant, si la mémoire manque, il est nécessaire de supprimer les pages les plus anciennes de la mémoire. Toutefois, on pourrait préférer conserver, si possible, la table des matières, sachant que les chances de retourner à celle-ci sont plus élevées que pour les autres pages. De la même façon, on peut souhaiter conserver en mémoire avec une priorité plus ou moins élevée certaines pages sélectionnées par l'utilisateur.

D'après le processus décrit précédemment, le garbage collector est susceptible de détruire tout objet pour lequel il n'existe plus de référence. En revanche, s'il existe une seule référence à un objet, celui-ci ne sera jamais détruit.

Java permet toutefois d'établir un type de référence spéciale, sorte de "référence limitée", au moyen d'un objet particulier de la classe **Reference**. Cette classe est une classe abstraite et est étendue par diverses autres classes permettant d'établir des références plus ou moins fortes.

L'exemple suivant montre la création d'une référence de type **SoftReference** pour un objet de la classe **Lapin**. Le premier objet créé est ainsi référencé :

```
import java.lang.ref.*;
public class Elevage3 {
   public static void main(String[] argv) {
      while (!Lapin.gc) {
         Lapin.créerLapin();
      }
   }
}

class Lapin {
   private static int nombre = 0;
   private int numero;
   static boolean gc = false;
   static SoftReference sr;

   private Lapin() {
      numero = ++nombre;
      if (numero == 1)
         sr = new SoftReference(this);
      System.out.println("Creation du lapin no " + numero);
   }
   public void finalize() {
      gc=true;
      System.out.println("Finalisation du lapin no " + numero);
   }
   public static Lapin créerLapin() {
      return new Lapin();
   }
}
```

L'exécution du programme montre que, lorsque le garbage collector démarre, il ignore cet objet :

```
   .
   .
   .
Creation du lapin no 2330
Creation du lapin no 2331
Finalisation du lapin no 2
Finalisation du lapin no 3
Creation du lapin no 2332
Finalisation du lapin no 4
```

(Vous remarquerez par ailleurs que cela a également une incidence sur les performances des deux processus mis en œuvre.)

Vous objecterez peut-être que ce comportement est tout à fait normal et qu'une **SoftReference** produit exactement le même résultat qu'une référence ordinaire. Le programme suivant montre que ce n'est pas le cas :

```
import java.lang.ref.*;

public class Elevage4 {
   public static void main(String[] argv) {
      while (!Lapin.gc) {
         Lapin.créerLapin();
      }
   }
}

class Lapin {
   private static int nombre = 0;
   private int numero;
   static boolean gc = false;
   static Reference[] r = new Reference[300000];
```

```
   private Lapin() {
      numero = ++nombre;
      r[numero] = new SoftReference(this);
      System.out.println("Creation du lapin no " + numero);
   }

   public void finalize() {
      gc = true;
      System.out.println("Finalisation du lapin no " + numero);
   }

   public static Lapin créerLapin() {
      return new Lapin();
   }
}
```

Cette fois, tous les objets créés reçoivent une référence de type **SoftReference**. L'exécution de ce programme donne le résultat suivant :

```
      .
      .
      .
   Creation du lapin no 59278
   Creation du lapin no 59279
   Finalisation du lapin no 1
   Finalisation du lapin no 2
   Finalisation du lapin no 3
   Finalisation du lapin no 4
      .
      .
      .
   Finalisation du lapin no 31
   Finalisation du lapin no 32
   Creation du lapin no 59280
   Finalisation du lapin no 33
```

Cette fois, le garbage collector a attendu beaucoup plus longtemps avant d'éliminer les objets.

Références et accessibilité

(Notez que l'accessibilité dont il est question ici n'a rien à voir avec celle dont nous avons parlé au Chapitre 8.)

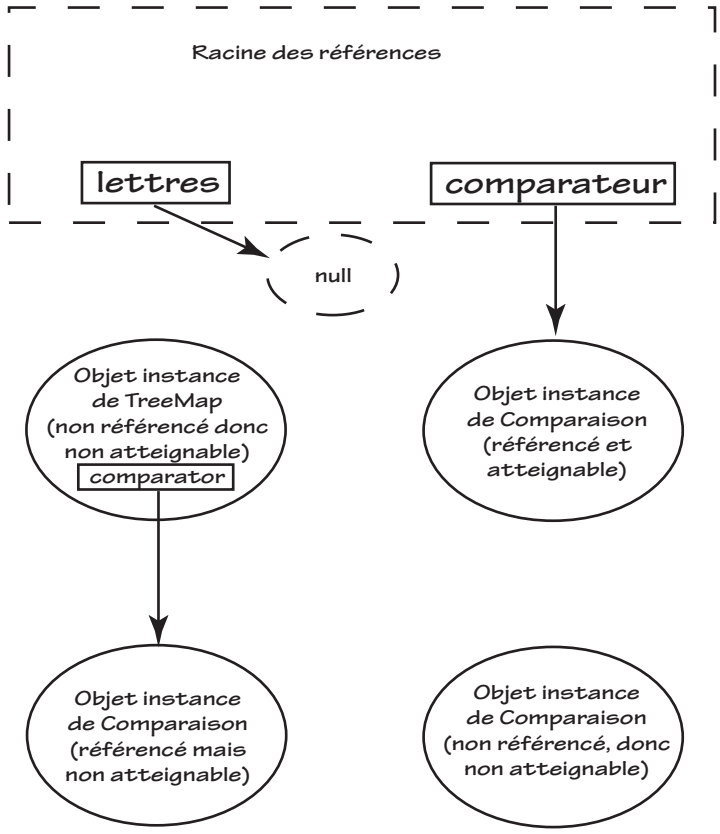
Pour comprendre les références limitées, il est nécessaire de revenir sur la façon dont les objets Java sont référencés et sur ce que cela implique sur la possibilité ou non de les atteindre.

Nous avons vu au début de ce chapitre qu'un objet pouvait être accessible s'il existait une référence à cet objet dans un espace particulier, que l'on appelle *racine des références*. Le programme :

```
TreeMap lettres = new TreeMap(new Comparaison());
Collections.sort(lettres, new Comparaison());
Comparaison comparateur = new Comparaison();
Collections.sort(lettres, comparateur);
lettres = null;
```

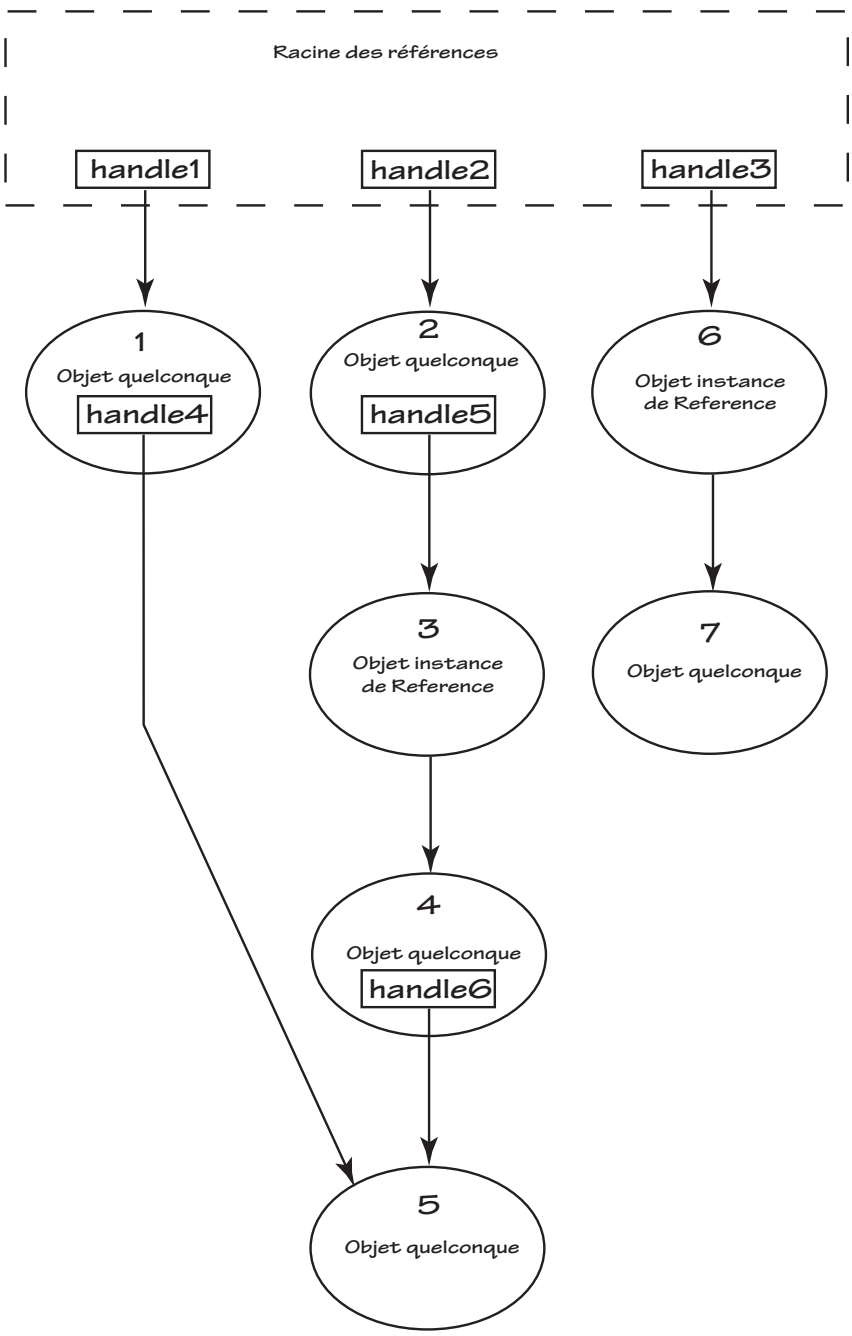
aboutit à la situation décrite par la figure de la page suivante.

Dans ce schéma, on voit qu'il existe des références pour deux des objets, alors que les deux autres n'en ont pas. Les objets sans référence sont inaccessibles, et donc éligibles pour le garbage collector. Une des instances de **Comparaison** est référencée par le handle **comparator**, mais ce handle n'appartient pas à la racine des références. Bien que référencé, cet objet est également inaccessible et sera supprimé par le garbage collector (dans un délai plus long que les objets sans référence).



Les références faibles

Les objets peuvent également être référencés à l'aide de références faibles, comme nous l'avons vu dans l'exemple des lapins. Une référence faible est obligatoirement une chaîne de références dont la première est forte. En effet, une référence faible est établie par l'intermédiaire d'un objet de type **Reference**. Cet objet doit lui-même avoir une référence. Il peut s'agir soit d'une référence forte, soit d'une référence faible, ce qui nous ramène au cas précédent. Sur le schéma de la page ci-contre, on peut voir les différents cas possibles.


Racine des références
handle1
handle2
handle3
1
Objet quelconque
handle4
2
Objet quelconque
handle5
6
Objet instance
de Reference
3
Objet instance
de Reference
7
Objet quelconque
4
Objet quelconque
handle6
5
Objet quelconque

Dans cet exemple, tous les objets sont accessibles.

- L'objet 1 a une référence forte car son handle appartient à la racine des références.
- L'objet 2 a une référence forte pour la même raison.
- L'objet 3 a une référence forte car la chaîne de référence qui le rend accessible (elle commence dans la racine des références) ne comporte que des références fortes.
- L'objet 4 a une référence faible car la chaîne de référence qui le rend accessible comporte une référence faible.
- L'objet 5 a une référence forte car une des chaînes qui le rendent accessible ne comporte que des références fortes.
- L'objet 6 a une référence forte car son handle appartient à la racine des références.
- L'objet 7 a une référence faible.

Pour résumer :

- Un objet a une référence forte si une des chaînes de références qui le rendent accessible ne comporte que des références fortes.
- Un objet a une référence faible si toutes les chaînes de références qui le rendent accessible comportent au moins une référence faible.

Tous les objets qui ont une référence faible sont éligibles pour le garbage collector. Cela ne signifie pas pour autant qu'ils seront supprimés automatiquement. Le garbage collector traite ces objets différemment selon le type exact de référence. Il existe en effet plusieurs sortes de références faibles, correspondant à différentes classes dérivées de la classe **Reference**. Par ordre de force décroissante, on trouve les références suivantes :

SoftReference

Les objets référencés par cette classe peuvent être supprimés lorsque la quantité de mémoire devient faible. Le garbage collector fait de son mieux pour effectuer la suppression en commençant par l'objet le moins récemment utilisé, et pour supprimer tous les objets de ce type avant d'indiquer une erreur de type **OutOfMemoryError**. Ce faisant, il essaie tout de même d'en conserver le plus grand nombre.

La classe **SoftReference** comporte deux méthodes :

- **public Object get()** retourne l'objet référencé, ou **null** si la référence a été supprimée par le garbage collector ou au moyen de la méthode **clear()**.

- **public void clear()** permet de supprimer la référence et de rendre ainsi l'objet référencé inaccessible (et donc plus immédiatement supprimable par le garbage collector). Toutefois, cette méthode ne supprime pas les autres références à l'objet (fortes ou faibles).

WeakReference

Les objets référencés par cette classe sont supprimés de la même façon que les précédents. La seule différence est que le garbage collector n'essaie pas d'en conserver un nombre maximal en mémoire. Une fois qu'il a démarré, tous les objets de ce type seront supprimés. L'intérêt de cette classe de références est qu'elle permet à un processus d'être informé lorsqu'un objet créé par un autre processus n'a plus de référence forte. Pour cela, il est nécessaire d'utiliser une *queue*. Les *queues* seront décrites dans la prochaine section.

PhantomReference

Les objets référencés par cette classe sont placés dans la queue correspondante par le garbage collector après qu'ils ont été finalisés (c'est-à-dire après l'exécution de leur méthode **finalize()**) mais ne sont pas supprimés.

La méthode **get()** de cette classe ne permet pas de récupérer les objets référencés et retourne toujours la valeur **null**.

Ce type de référence permet d'effectuer divers traitements de désallocation de ressources (par exemple fichiers ou sockets ouverts) avant la suppression des objets. Il est de la responsabilité du programmeur d'appeler ensuite la méthode **clear()** pour rendre l'objet supprimable par le garbage collector.

Les queues

Chaque référence faible peut être associée à une *queue*. Une queue est un objet instance de la classe **ReferenceQueue**. Il s'agit d'une structure dans laquelle seront placés les objets traités par le garbage collector. (L'utilisation d'une queue est même obligatoire avec les références de type **PhantomReference**.)

Un processus peut tester une queue au moyen de la méthode **poll()** pour savoir si elle contient une référence. Cette méthode retourne la première référence disponible, s'il y en a une, **null** dans le cas contraire.

La classe **ReferenceQueue** contient également les méthodes **remove()**, qui attend jusqu'à ce qu'un objet soit disponible dans la queue pour le retourner, et **remove(long timeout)**, qui attend le nombre de millisecondes indiqué par **timeout**. Si une référence est disponible avant l'expiration du délai, elle est immédiatement retournée par la méthode. Dans le cas contraire, elle retourne la valeur **null**. Si **timeout** vaut 0, la méthode se comporte comme dans le cas où il n'y a pas d'argument.

Les queues sont particulièrement utiles avec les tables. En effet, une table stocke des couples clés/valeurs. Les clés comme les valeurs sont des objets. L'utilisation de références faibles associées à une queue pour les valeurs permet d'informer un processus de la suppression d'une valeur de façon qu'il soit possible de supprimer la clé correspondante.

Exemple d'utilisation de références faibles

Les références faibles sont particulièrement intéressantes pour stocker les images en mémoire cache. Il est ainsi possible de conserver ces images en

mémoire à condition que la quantité de mémoire soit suffisante. L'utilisation des différents types de références permet d'établir une priorité. Le programme suivant crée une applet qui affiche deux images mises en mémoire cache :

```
import java.applet.*;
import java.awt.*;
import java.lang.ref.*;

public class Cache extends Applet {

   Reference[] rf = new Reference[10];

   public void init() {
      // Ici, le code de l'applet
   }

   public void paint (Graphics g) {
      Image image;
      // affichage de l'image 1
      if (rf[1] != null)
         image = (Image)(rf[1].get());
      else {
         image = getImage(getCodeBase(), "image1.gif");
         rf[1] = new SoftReference(image);
      }
      g.drawImage(image, 20, 40, this);
      // affichage de l'image 2
      if (rf[2] != null)
         image = (Image)(rf[2].get());
      else {
         image = getImage(getCodeBase(), "image2.gif");
         rf[1] = new SoftReference(image);
      }
      g.drawImage(image, 120, 230, this);
      image = null;
   }
}
```

Ce programme fait appel à des notions que nous n'avons pas encore étudiées. Vous devriez néanmoins être à même de comprendre son fonctionnement. La ligne :

```
Reference[] rf = new Reference[10];
```

crée un tableau de références. La classe **Reference** est utilisée afin de pouvoir y stocker plusieurs types de références (grâce au polymorphisme). Le tableau comporte 10 entrées, mais il devrait en fait en contenir autant qu'il y a d'images à mettre en cache. (L'utilisation d'un tableau est beaucoup plus pratique que celle de références individuelles, mais un tout petit peu moins efficace en termes de gestion de la mémoire.)

La méthode **init()** :

```
public void init() {
   // Ici, le code de l'applet
}
```

contient le code de l'applet.

La méthode **paint()** est celle qui est appelée chaque fois que l'applet est affichée. C'est donc elle qui contient le code affichant les images.

Le handle **image** est déclaré de type **Image** :

```
Image image;
```

Avant d'afficher la première image, nous testons le tableau des références pour savoir si le premier élément est non **null**. Si c'est le cas, cela signifie que l'image a déjà été affichée et se trouve donc en mémoire. Elle est alors récupérée à l'aide de la méthode **get()**. Notez que cette méthode retourne un **Object**, qui doit donc être sous-casté explicitement en **Image**.

```
if (rf[1] != null)
   image = (Image)(rf[1].get());
```

Dans le cas contraire, cela signifie que l'image n'a jamais été affichée ou qu'elle a été supprimée de la mémoire. Il faut donc la charger :

```
else {
   image = getImage(getCodeBase(), "image1.gif");
```

puis créer une référence faible. Celle-ci est créée de type **SoftReference** :

```
   rf[1] = new SoftReference(image);
}
```

L'image est ensuite affichée :

```
g.drawImage(image, 20, 40, this);
```

Il faudrait normalement ensuite supprimer la référence forte à l'image, mais cela n'est pas nécessaire puisque nous allons immédiatement réattribuer le handle **image** à un autre objet :

```
// affichage de l'image 2 (non prioritaire)
if (rf[2] != null)
   image = (Image)(rf[2].get());
else {
   image = getImage(getCodeBase(), "image2.gif");
   rf[1] = new SoftReference(image);
}
g.drawImage(image, 120, 230, this);
```

Dans ces lignes, la deuxième image est traitée de la même façon. Une fois l'affichage terminé, la référence forte de la dernière image affichée est supprimée grâce à l'instruction :

```
image = null;
```

Note : Si vous souhaitez tester ce programme, vous devez disposer d'un fichier HTML contenant les instructions suivantes :

```
<html>
   <head>
      <title>Titre facultatif</title>
   </head>
   <body>
      <applet code="Cache" width="400" height="300">
      </applet>
   </body>
</html>
```

Compilez le programme après l'avoir enregistré dans le fichier **Cache.java** puis ouvrez le fichier HTML (qui peut avoir n'importe quel nom) avec l'AppletViewer, en tapant :

```
AppletViewer nom_du_fichier_html
```

ou tout simplement :

```
va
```

suivi de la touche F3 si vous avez créé les fichiers batch indiqués au Chapitre 1. Pour que le programme fonctionne, vous devez en outre disposer de deux fichiers images appelés respectivement **image1.gif** et **image2.gif** et placés dans le même dossier que le programme.

Autres formes de contrôle du garbage collector

Java dispose d'autres moyens pour contrôler le garbage collector. Parmi ceux-ci, le plus direct est la méthode **gc()**. Il s'agit d'une méthode statique de la classe **java.lang.System**, qui a pour effet de lancer le garbage collector.

L'exemple suivant en fait la démonstration. Tout d'abord, considérez le programme ci-dessous :

```
public class Elevage5 {
   public static void main(String[] argv) {
      for (int i = 0; i < 10; i++) {
         Lapin.créerLapin();
      }
   }
}

class Lapin {
   private static int nombre = 0;
   private int numero;

   private Lapin() {
      numero = ++nombre;
      System.out.println("Creation du lapin no " + numero);
   }

   public void finalize() {
      System.out.println("Finalisation du lapin no " + numero);
   }

   public static Lapin créerLapin() {
      return new Lapin();
   }
}
```

Ce programme créé dix instances de la classe **Lapin** sans références et se termine. Il affiche le résultat suivant :

```
Creation du lapin no 1
Creation du lapin no 2
Creation du lapin no 3
Creation du lapin no 4
Creation du lapin no 5
```

```
Creation du lapin no 6
Creation du lapin no 7
Creation du lapin no 8
Creation du lapin no 9
Creation du lapin no 10
```

Voyons maintenant ce que produit ce programme légèrement modifié :

```
public class Elevage5 {
   public static void main(String[] argv) {
      for (int i = 0; i < 10; i++) {
         Lapin.créerLapin();
      }
      System.gc();
   }
}
class Lapin {
   .
   .
   .
```

La ligne ajoutée lance le garbage collector. Le programme affiche maintenant :

```
Creation du lapin no 1
Creation du lapin no 2
Creation du lapin no 3
Creation du lapin no 4
Creation du lapin no 5
Creation du lapin no 6
Creation du lapin no 7
Creation du lapin no 8
Creation du lapin no 9
Creation du lapin no 10
Finalisation du lapin no 1
Finalisation du lapin no 2
Finalisation du lapin no 3
Finalisation du lapin no 4
```

Le résultat peut être légèrement différent selon la vitesse de votre ordinateur. Que se passe-t-il ?

Après la création des 10 lapins, le garbage collector est lancé et le programme se termine. Le garbage collector a ici le temps de finaliser quatre lapins avant d'être interrompu. Dans ces conditions, la méthode **gc()** n'est pas très utile. Elle ne permet pas, en effet, de s'assurer que les méthodes **finalize()** de tous les objets se trouvant en mémoire seront exécutées. Une solution pourrait être d'ajouter un temps d'attente après l'appel du garbage collector. Cela relève un peu du bricolage dans la mesure où il n'y a aucun moyen de connaître le temps nécessaire.

Une meilleure solution consiste à employer la méthode **runFinalization()** :

```
public class Elevage5 {
   public static void main(String[] argv) {
      for (int i = 0; i < 10; i++) {
         Lapin.créerLapin();
      }
      System.gc();
      System.runFinalization();
   }
}
class Lapin {
   .
   .
   .
```

Le programme affiche maintenant :

```
Creation du lapin no 1
Creation du lapin no 2
Creation du lapin no 3
Creation du lapin no 4
Creation du lapin no 5
Creation du lapin no 6
Creation du lapin no 7
Creation du lapin no 8
```

```
Creation du lapin no 9
Creation du lapin no 10
Finalisation du lapin no 1
Finalisation du lapin no 2
Finalisation du lapin no 3
Finalisation du lapin no 4
Finalisation du lapin no 5
Finalisation du lapin no 6
Finalisation du lapin no 7
Finalisation du lapin no 8
Finalisation du lapin no 9
Finalisation du lapin no 10
```

A priori, le résultat souhaité a été obtenu. Pourtant, vous ne pouvez pas être absolument sûr du résultat. En effet, la documentation précise que Java fait "de son mieux" pour que cette méthode entraîne la finalisation de tous les objets restant en mémoire, mais cela n'est pas garanti. Ainsi, le programme suivant :

```
public class Elevage6 {
   public static void main(String[] argv) {
      for (int i = 0; i < 10000; i++) {
         Lapin.créerLapin();
      }
      System.gc();
      System.runFinalization();
   }
}

class Lapin {
   private static int nombre = 0;
   private static int nombre2 = 0;
   private Lapin() {
      ++nombre;
      if (nombre == 10000)
         System.out.println(nombre + " lapins crees");
   }
```

```
    public void finalize() {
       ++nombre2;
       if (nombre2 == 10000)
          System.out.println(nombre2 + " lapins finalises");
    }

    public static Lapin créerLapin() {
       return new Lapin();
    }
}
```

affiche :

```
10000 lapins crees
10000 lapins finalises
```

alors que si vous remplacez les valeurs 10000 par 100000, le résultat devient :

```
100000 lapins crees
```

montrant que certains objets n'ont pas été finalisés.

Attention : La méthode **runFinalization()** ne concerne que les objets qui ont été collectés par le garbage collector. Si vous inversez l'ordre des lignes :

```
        System.gc();
        System.runFinalization();
```

le résultat sera tout à fait différent !

Une autre possibilité consiste à utiliser la méthode **runFinalizersOnExit(true)**, comme dans l'exemple suivant :

```
public class Elevage7 {
   public static void main(String[] argv) {
      for (int i = 0; i < 10; i++) {
         Lapin.créerLapin();
      }
      Runtime.getRuntime().runFinalizersOnExit(true);
   }
}

class Lapin {
   private static int nombre = 0;
   private int numero;

   private Lapin() {
      numero = ++nombre;
      System.out.println("Creation du lapin no " + numero);
   }
   public void finalize() {
      System.out.println("Finalisation du lapin no " + numero);
   }
   public static Lapin créerLapin() {
      return new Lapin();
   }
}
```

qui produit le résultat suivant :

```
Creation du lapin no 1
Creation du lapin no 2
Creation du lapin no 3
Creation du lapin no 4
Creation du lapin no 5
Creation du lapin no 6
Creation du lapin no 7
Creation du lapin no 8
Creation du lapin no 9
Creation du lapin no 10
```

```
Finalisation du lapin no 10
Finalisation du lapin no 9
Finalisation du lapin no 8
Finalisation du lapin no 7
Finalisation du lapin no 6
Finalisation du lapin no 5
Finalisation du lapin no 4
Finalisation du lapin no 3
Finalisation du lapin no 2
Finalisation du lapin no 1
```

Notez qu'il n'a pas été nécessaire de lancer le garbage collector. Par ailleurs, vous pouvez remarquer que les objets sont finalisés dans l'ordre inverse de leur création, mais cela n'est absolument pas garanti ! Par ailleurs, cela ne concerne que les objets non traités par le garbage collector.

En revanche, il semble que vous puissiez compter sur le fait que tous les objets seront finalisés. (Nous l'avons testé avec plusieurs millions d'objets en obtenant un résultat positif.) De plus, cette façon de faire entraîne la finalisation de tous les objets, qu'ils aient été ou non collectés par le garbage collector.

Note : La méthode **runFinalizersOnExit()** est appelée ici à partir d'une instance de la classe **Runtime** retournée par la méthode statique **getRuntime()**. C'est exactement ce que fait la méthode **runFinalizersOnExit()** de la classe **System**, comme en témoignent les sources de Java :

```
public static void runFinalizersOnExit(boolean value) {
   Runtime.getRuntime().runFinalizersOnExit(value);
}
```

Ce programme avait été écrit avec la version bêta 4 de Java 2. Dans cette version, la méthode **runFinalizersOnExit()** était déclarée "deprecated", c'est-à-dire obsolète parce que "*Cette méthode est intrinsèquement dangereuse. Elle peut entraîner l'appel de finaliseurs sur des objets valides pendant que d'autres processus sont en train de les manipuler et produire ainsi un comportement erratique ou un blocage*".

En revanche, la version de la classe **Runtime** n'était pas déclarée deprecated. Ce problème a été corrigé dans la version finale. En compilant la classe **Elevage7**, vous obtiendrez donc un message d'avertissement. Cela n'empêche toutefois pas le programme de fonctionner.

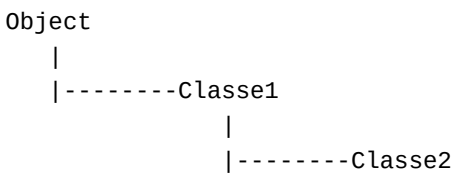
La finalisation et l'héritage

Il est une chose importante à ne pas oublier lorsque vous dérivez une classe comportant une méthode **finalize()**. Si la classe dérivée ne nécessite pas de traitement particulier en ce qui concerne la finalisation, il n'y a pas de problème. En revanche, dans le cas contraire, vous devrez redéfinir cette méthode. Vous ne devez pas oublier alors d'appeler explicitement la méthode **finalize()** de la classe parente, en terminant votre définition par :

```
super.finalize();
```

Dans le cas contraire, la finalisation ne serait pas exécutée correctement. En effet, contrairement aux constructeurs, les finaliseurs des classes parentes ne sont pas appelés automatiquement par Java. Pourquoi finaliser la classe parente après la classe dérivée ? Tout simplement parce qu'il est possible que la finalisation de la classe dérivée repose sur certains membres de la classe parente, alors que le contraire est évidemment impossible.

Bien sûr, vous pensez peut-être que cela est inutile si la classe parente ne comporte pas de méthode **finalise()**. Tout d'abord, notez que, toutes les classes dérivant de la classe **Object**, elles disposent automatiquement d'une telle méthode qu'elles héritent de cette classe. Par ailleurs, même si votre classe dérive d'une classe qui ne redéfinit pas cette méthode, rien ne peut vous assurer qu'il en sera toujours ainsi. Si votre programme comporte la hiérarchie suivante :



où **Classe2** comporte une méthode **finalize()** mais pas **Classe1**, il est tout de même préférable de terminer le code de cette méthode par l'instruction :

```
super.finalize()
```

qui aura pour effet d'appeler la méthode de la classe parente **Object**, qui ne fait rien comme on peut le vérifier dans les sources de cette classe :

```
protected void finalize() throws Throwable { }
```

De cette façon, si vous modifiez un jour la classe **Classe1** et que vous vous apercevez qu'il devient nécessaire de lui ajouter une redéfinition de cette méthode, vous n'aurez pas à modifier la classe **Classe2**. Il s'agit là encore du respect d'une des règles essentielles de la programmation efficace. Imaginez maintenant que votre application comporte une centaine de classes dérivées de **Classe1**, et c'est une centaine de modifications que vous seriez obligé d'effectuer si vous n'aviez pas respecté cette règle. Qui plus est, imaginez le résultat si vos classes sont destinées à une diffusion publique !

La finalisation et le traitement d'erreur

Nous verrons dans un prochain chapitre quel mécanisme permet de traiter les différentes erreurs d'exécution qui peuvent se produire au cours du déroulement d'un programme. Sachez simplement pour l'instant que lorsqu'une erreur survient, elle peut être traitée dans un bloc de code particulier. Il est fréquent que ce traitement entraîne l'arrêt du programme. Si cette situation se produit dans une méthode **finalize()**, vous devrez prendre les mesures nécessaires pour que les opérations de nettoyage indispensables aient tout de même lieu. Nous y reviendrons.

Contrôler le garbage collector à l'aide des options de l'interpréteur

Une autre façon de contrôler le fonctionnement du garbage collector consiste à employer certains paramètres sur la ligne de commande de l'interpréteur. En effet, nous avons vu que le garbage collector offre deux modes de fonctionnement : synchrone et asynchrone. Le paramètre :

```
-noasyncgc
```

permet d'empêcher le fonctionnement asynchrone du garbage collector. De cette façon, aucun ralentissement n'aura lieu jusqu'à ce que le garbage collector commence son fonctionnement synchrone à cause d'un manque de mémoire ou à la suite de l'appel de la méthode **gc()**.

Par ailleurs, le paramètre :

```
-noclassgc
```

permet d'indiquer au garbage collector que les classes qui ont été chargées et ne sont plus utilisées ne doivent pas être supprimées. L'utilisation de ce paramètre peut, lorsque la mémoire disponible est importante, accélérer certains programmes.

Enfin, les deux paramètres suivants :

```
-ms initmem[k|m]
-mx maxmem[k|m]
```

permettent d'indiquer, pour le premier, la quantité de mémoire initialement attribuée au *tas* (la structure dans laquelle les objets sont créés) et, pour le second, la quantité maximale de mémoire qui pourra lui être allouée. Les valeurs par défaut sont respectivement de 1 Mo et 16 Mo. L'indication de valeurs supérieures peut améliorer le fonctionnement des programmes fai-

sant une utilisation intensive de la mémoire. Les lettres **k** et **m** indiquent si les valeurs sont en kilo-octets ou en mégaoctets. La quantité de mémoire allouée a une influence déterminante sur la fréquence de démarrage du garbage collector ainsi que sur le temps mis par celui-ci à effectuer sa tâche.

Résumé

Dans ce chapitre, nous avons étudié le mécanisme de gestion de la mémoire de la JVM. Nous avons vu comment chacun des aspects de ce mécanisme pouvait être plus ou moins contrôlé, directement ou indirectement. Il s'agit là d'un problème crucial pour les performances et la sécurité des programmes. Nous avons également étudié les références faibles, qui sont une des nouveautés de la version 2 de Java.

Chapitre 12

Les classes internes

Dans les chapitres précédents, nous avons déjà eu plusieurs fois l'occasion d'indiquer que les classes Java peuvent contenir, outre des primitives, des objets (du moins leurs références) et des définitions de méthodes, des définitions de classe. Nous allons maintenant nous intéresser de plus près à cette possibilité.

Les classes imbriquées

Il arrive fréquemment que certaines classes ne soient utilisées que par une seule autre classe. Considérez, par exemple, le programme suivant :

```
import java.util.*;
class Tri {
   public static void main(String[] args) {
      Vector lettres = new Vector();
      lettres.add(new Lettre('C'));
      lettres.add(new Lettre('ç'));
      lettres.add(new Lettre('S'));
      lettres.add(new Lettre('u'));
      lettres.add(new Lettre('c'));
      lettres.add(new Lettre('é'));
      lettres.add(new Lettre('e'));
      lettres.add(new Lettre('R'));
      lettres.add(new Lettre('ù'));
      Collections.sort(lettres, new Comparaison());
      Iterator it = lettres.iterator();
      while (it.hasNext()) {
         ((Lettre)it.next()).affiche();
      }
   }
}
class Lettre {
   char valeur;
   Lettre (char v) {
      valeur = v;
   }
   public void affiche() {
      System.out.println(win2DOS(valeur));
   }
   public static char win2DOS(char c) {
      switch (c) {
         case 'à':
            return (char)133;
         case 'â':
            return (char)131;
         case 'ä':
            return (char)132;
         case 'é':
            return (char)130;
```

```
            case 'è':
               return (char)138;
            case 'ë':
               return (char)137;
            case 'ê':
               return (char)136;
            case 'ï':
               return (char)139;
            case 'î':
               return (char)140;
            case 'ô':
               return (char)147;
            case 'ö':
               return (char)148;
            case 'ü':
               return (char)129;
            case 'û':
               return (char)150;
            case 'ù':
               return (char)151;
            case 'ç':
               return (char)135;
         }
         return c;
      }
   }

   class Comparaison implements Comparator {
      int retval = 0;
      public int compare(Object o1, Object o2) {
         if ((o1 instanceof Lettre) && (o2 instanceof Lettre)) {
            if (Conversion.conv(((Lettre)o1).valeur) <
                           Conversion.conv(((Lettre)o2).valeur))
               retval = -1;
            else if (Conversion.conv(((Lettre)o1).valeur) >
                           Conversion.conv(((Lettre)o2).valeur))
               retval = 1;
         }
```

```
      else {
         // traitement d'erreur
      }
      return retval;
   }
}
class Conversion {
   static char conv (char c) {
      switch (c) {
         case 'à':
         case 'â':
         case 'ä':
              return 'A';
         case 'é':
         case 'è':
         case 'ë':
         case 'ê':
              return 'E';
         case 'ï':
         case 'î':
              return 'I';
         case 'ô':
         case 'ö':
              return 'O';
         case 'ü':
         case 'û':
         case 'ù':
              return 'U';
         case 'ç':
              return 'C';
         default:
              if (c > 96)
                 return (char)(c - 32);
              else
                 return c;
      }
   }
}
```

Ce programme définit quatre classes :

- La classe **Tri** constitue le programme principal (celui qui contient la méthode **main()**).

- La classe **Conversion** sert à convertir les minuscules accentuées en majuscules. Elle ne contient qu'une méthode statique.

- La classe **Lettre** est un enveloppeur pour la primitive **char** et contient en outre une méthode permettant de convertir les caractères ANSI (sous-ensemble des caractères UNICODE utilisés par Java) en caractères ASCII (permettant l'affichage dans une fenêtre DOS).

- La classe **Comparaison** définit un comparateur pour les objets instances de la classe **Lettre**.

Toutes ces classes sont définies dans le même fichier, ce qui convient dans le cadre de la démonstration, mais certainement pas pour la pratique courante de la programmation efficace.

Chacune de ces classes pourrait être définie séparément dans un fichier et affectée à un package. Rappelons que, telles qu'elles sont définies ici, toutes ces classes sont affectées au package par défaut.

Cependant, si l'on examine de plus près ce que fait chaque classe, on s'aperçoit qu'elles n'ont pas du tout le même statut. Ainsi, la classe **Conversion** ne contient qu'une méthode statique. Elle n'est pas destinée à être instanciée. Elle pourrait donc avantageusement être rangée dans un package d'utilitaires.

La classe **Lettre** est un enveloppeur créé spécialement pour l'application et pourrait donc être placée dans un package contenant toutes les classes spécifiques de celle-ci.

En revanche, il est évident que la classe **Comparaison** ne concerne que la classe **Lettre**. On voit donc qu'il existe de fait une sorte de lien hiérarchique entre les classes **Lettre** et **Comparaison** puisque **Comparaison** est un outil de **Lettre**. Dès lors, il serait plus simple de placer la définition de

Comparaison à l'intérieur de celle de **Lettre**. En Java, cela est tout à fait possible, comme le montre l'exemple suivant :

```
   import java.util.*;
   class Imbrique {
      public static void main(String[] args) {
         Vector lettres = new Vector();
         lettres.add(new Lettre('C'));
         lettres.add(new Lettre('ç'));
         lettres.add(new Lettre('S'));
         lettres.add(new Lettre('u'));
         lettres.add(new Lettre('c'));
         lettres.add(new Lettre('é'));
         lettres.add(new Lettre('e'));
         lettres.add(new Lettre('R'));
         lettres.add(new Lettre('ù'));
         Collections.sort(lettres, new Lettre.Comparaison());
         Iterator it = lettres.iterator();
         while (it.hasNext()) {
            ((Lettre)it.next()).affiche();
         }
      }
   }
   class Lettre {
      char valeur;
      Lettre (char v) {
         valeur = v;
      }
      public void affiche() {
         System.out.println(win2DOS(valeur));
      }
      public static char win2DOS(char c) {
         switch (c) {
            case 'à':
               return (char)133;
            case 'â':
               return (char)131;
            case 'ä':
   return (char)132;
case 'é':
   return (char)130;
```

```
            case 'è':
               return (char)138;
            case 'ë':
               return (char)137;
            case 'ê':
               return (char)136;
            case 'ï':
               return (char)139;
            case 'î':
               return (char)140;
            case 'ô':
               return (char)147;
            case 'ö':
               return (char)148;
            case 'ü':
               return (char)129;
            case 'û':
               return (char)150;
            case 'ù':
               return (char)151;
            case 'ç':
               return (char)135;
         }
         return c;
      }
      static class Comparaison implements Comparator {
         int retval = 0;
         public int compare(Object o1, Object o2) {
            if ((o1 instanceof Lettre) && (o2 instanceof Lettre)) {
               if (Conversion.conv(((Lettre)o1).valeur) <
                                 Conversion.conv(((Lettre)o2).valeur))
                  retval = -1;
               else if (Conversion.conv(((Lettre)o1).valeur) >
                                 Conversion.conv(((Lettre)o2).valeur))
                  retval = 1;
            }
            else {
               // traitement d'erreur
            }
            return retval;
         }
      }
   }
```

Les parties modifiées sont indiquées en gras. Notez que la définition de la classe **Conversion** n'a pas été incluse. En effet, cette classe a déjà été compilée et doit donc figurer dans le package par défaut. Si vous n'avez pas compilé le programme **Tri**, ajoutez simplement la définition de cette classe à la fin du fichier ou compilez-la séparément.

La définition de la classe **Comparaison** est maintenant imbriquée dans la définition de la classe **Lettre**. Notez que la référence à la classe **Comparaison** devient maintenant **Lettre.Comparaison**. Lors de la compilation du programme, le compilateur produit trois fichiers :

- **Imbrique.class** (le programme principal),
- **Lettre.class**,
- **Lettre$Comparaison.class** (la classe imbriquée). Le caractère $ est employé par Java pour pallier le fait que certains systèmes d'exploitation interdisent l'utilisation du point dans les noms de fichiers. Cela est cependant transparent pour le programmeur grâce à un mécanisme de conversion automatique. Toutes les références à une classe imbriquée en Java se font en utilisant le point comme séparateur.

La seule différence, si vous utilisez cette façon d'organiser vos classes, réside dans l'emplacement où elles sont stockées. Notez au passage que cela contredit ce que nous avions affirmé au chapitre consacré aux packages : si vous respectez la convention consistant à faire commencer les noms de classes par une capitale, nous avions indiqué que les chemins d'accès ne pouvaient comporter une capitale que dans la partie la plus à droite. Ainsi :

```
mesclasses.util.Lettre
```

désigne la classe **Lettre** dans le package **mesclasses.util**.

En revanche :

```
mesclasses.util.Lettre.Comparaison
```

désigne la classe **Comparaison** imbriquée dans la classe **Lettre** se trouvant elle-même dans le package **mesclasses.util**. Il est possible d'utiliser la directive **import** pour importer les classes imbriquées explicitement :

```
import mesclasses.util.Lettre.Comparaison;
```

ou en bloc :

```
import mesclasses.util.Lettre.*;
```

Notez également que la classe **Comparaison** est déclarée **static**, ce qui est obligatoire. En revanche, les interfaces imbriquées sont automatiquement statiques et il n'est pas nécessaire de les déclarer explicitement comme telles. (Vous êtes toutefois libre de le faire.)

Enfin, les classes imbriquées peuvent elles-mêmes contenir d'autres classes imbriquées, sans limitation de profondeur, du moins du point de vue de Java. (En ce qui concerne le système d'exploitation, une imbrication à un niveau trop profond pourrait conduire à dépasser la limite de longueur des noms de fichiers.)

Les classes membres

La particularité des classes imbriquées est qu'elles sont déclarées statiques. Il est cependant également possible de déclarer une classe non statique à l'intérieur d'une autre classe. Il s'agit alors d'une classe membre, ce qui est fondamentalement différent. En effet, lorsqu'une instance d'une classe est créée, une instance de chacune de ses classes membres est également créée. (Ce qui, au passage, explique pourquoi les interfaces imbriquées n'ont pas besoin d'une déclaration explicite **static**, puisqu'elles ne peuvent pas, de toute façon, être instanciées.)

Les exemples suivants montrent l'utilisation de classes membres et mettent en évidence la manière de faire référence à leurs champs :

```
public class Mandat1 {
   class Montant {
      private int v;

      Montant(int m) {
         v = m;
      }

      public int valeur() {
         return v;
      }
   }

   class Destinataire {
      private String nom;

      Destinataire(String qui) {
         nom = qui;
      }

      String aQui() {
         return nom;
      }
   }
      public void envoyer(int mont, String dest) {
      Montant montant1 = new Montant(mont);
      Destinataire destinataire1 = new Destinataire(dest);
      System.out.println("Un mandat de " + montant1.v
           + " francs a ete expedie a " + destinataire1.nom);
      System.out.println("Un mandat de " + montant1.valeur()
           + " francs a ete expedie a " + destinataire1.aQui());
   }

   public static void main (String[] args) {
      Mandat1 mandat1 = new Mandat1();
      mandat1.envoyer(1500, "Albert");
   }
}
```

Dans cet exemple, les classes **Montant** et **Destinataire** sont des membres de la classe **Mandat1**. Une instance de la classe **Mandat1** est créée dans la méthode **main()**, puis la méthode **envoyer()** est appelée avec les paramètres qui serviront à créer une instance de **Montant** et de **Destinataire**. Nous pouvons remarquer que, bien que les champs **valeur** et **nom** soient déclarés **private**, ils sont tout à fait accessibles depuis la classe externe **Mandat1**. En fait, les champs **private** des classes internes et externes sont accessibles librement à toutes les classes internes et externes. Ici, le champ **nom** de la classe **Destinataire** est donc accessible librement depuis la classe externe **Mandat1** mais également depuis la classe **Montant**, tout comme il le serait dans les classes internes aux classes **Montant et Destinataire**. L'exemple suivant en fait la démonstration :

```
public class Interne {
   private int n = 0;
   private Niveau1 o1;
   private Niveau2 o2;
   Interne() {
      o1 = new Niveau1();
      o2 = new Niveau2();
   }

   class Niveau1 {
      private int n1 = 1;
      private Niveau11 o11;
      private Niveau12 o12;

      Niveau1() {
         o11 = new Niveau11();
         o12 = new Niveau12();
      }

      class Niveau11 {
         private int n11 = 11;

         public void affiche() {
            System.out.println("Acces depuis niveau 2 :");
            System.out.println(n);
```

```
            System.out.println(o1.n1);
            System.out.println(o1.o11.n11);
            System.out.println(o1.o12.n12);
            System.out.println(o2.n2);
            System.out.println(o2.o21.n21);
            System.out.println(o2.o22.n22);
         }
      }

      class Niveau12 {
         private int n12 = 12;
      }

      public void affiche() {
         System.out.println("Acces depuis niveau 1 :");
         System.out.println(n);
         System.out.println(o1.n1);
         System.out.println(o1.o11.n11);
         System.out.println(o1.o12.n12);
         System.out.println(o2.n2);
         System.out.println(o2.o21.n21);
         System.out.println(o2.o22.n22);
      }
   }

   class Niveau2 {
      private int n2 = 2;
      private Niveau21 o21;
      private Niveau22 o22;

      Niveau2() {
         o21 = new Niveau21();
         o22 = new Niveau22();
      }

      class Niveau21 {
         private int n21 = 21;
      }
```

```
      class Niveau22 {
         private int n22 = 22;
      }
   }

   public void affiche() {
      System.out.println("Acces depuis niveau 0 :");
      System.out.println(n);
      System.out.println(o1.n1);
      System.out.println(o1.o11.n11);
      System.out.println(o1.o12.n12);
      System.out.println(o2.n2);
      System.out.println(o2.o21.n21);
      System.out.println(o2.o22.n22);
   }

   public static void main (String[] args) {
      Interne i = new Interne();
      i.affiche();
      i.o1.affiche();
      i.o1.o11.affiche();
   }
}
```

Ce programme affiche le résultat suivant :

```
Acces depuis niveau 0 :
0
1
11
12
2
21
22
Acces depuis niveau 1 :
0
1
```

```
11
12
2
21
22
Acces depuis niveau 2 :
0
1
11
12
2
21
22
```

On voit que, bien que tous les membres soient déclarés **private**, ils sont accessibles depuis toutes les classes exactement comme s'ils leur appartenaient. La syntaxe employée pour y accéder reprend la hiérarchie d'imbrication des classes. Ici, les trois versions de la méthode **Affiche()** utilisent des références explicites. Elles peuvent cependant être simplifiées. Dans une référence, les éléments désignant la classe contenant cette référence ou des classes d'un niveau hiérarchique supérieur peuvent être omis. Par exemple, la méthode **i.o1.o11.affiche()** peut être remplacée par :

```
public void affiche() {
   System.out.println("Acces depuis niveau 2 :");
   System.out.println(n);
   System.out.println(n1);
   System.out.println(n11);
   System.out.println(o12.n12);
   System.out.println(o2.n2);
   System.out.println(o2.o21.n21);
   System.out.println(o2.o22.n22);
}
```

Pour comprendre le fonctionnement exact de ces références, il faut savoir que nous avons parlé de *références explicites* par abus de langage. En fait, la version utilisant réellement des références explicites est la suivante :

```
public void affiche() {
   System.out.println("Acces depuis niveau 2 :");
   System.out.println(Interne.this.n);
   System.out.println(Interne.this.o1.n1);
   System.out.println(Interne.this.o1.o11.n11);
   System.out.println(Interne.this.o1.o12.n12);
   System.out.println(Interne.this.o2.n2);
   System.out.println(Interne.this.o2.o21.n21);
   System.out.println(Interne.this.o2.o22.n22);
}
```

Interne.this est la façon de faire référence à l'instance de la classe extérieure **Interne** et correspond donc à **i** dans la référence à la méthode :

```
i.o1.o11.affiche()
```

(**i** est en l'instance de la classe **Interne** référencée par **Interne.this**).

Nous voyons qu'il est possible d'omettre dans les références aux objets tous les éléments qui figurent dans la référence source (ici la référence à la méthode **affiche()**). Si nous remplaçons **i** par **Interne.this**, nous obtenons la référence :

```
Interne.this.o1.o11.
```

Nous pouvons donc supprimer les éléments qui figurent ci-après en gras :

```
public void affiche() {
   System.out.println("Acces depuis niveau 2 :");
   System.out.println(Interne.this.n);
   System.out.println(Interne.this.o1.n1);
   System.out.println(Interne.this.o1.o11.n11);
   System.out.println(Interne.this.o1.o12.n12);
   System.out.println(Interne.this.o2.n2);
   System.out.println(Interne.this.o2.o21.n21);
   System.out.println(Interne.this.o2.o22.n22);
}
```

Bien entendu, les autres versions de la méthode **affiche()** peuvent également être simplifiées, en respectant le même principe.

Le programme **Mandat1** peut aussi être écrit d'une autre façon :

```
public class Mandat2 {
   class Montant {
      private int v;

      Montant(int m) {
         v = m;
      }

      public int valeur() {
         return v;
      }
   }

   class Destinataire {
      private String nom;

      Destinataire(String qui) {
         nom = qui;
      }

      String aQui() {
         return nom;
      }
   }

   public Montant créerMontant(int m) {
      return new Montant(m);
   }

   public Destinataire créerDestinataire(String s) {
      return new Destinataire(s);
   }
```

```
   public static void main (String[] args) {
      Mandat2 mandat2 = new Mandat2();
      Montant montant2 = mandat2.créerMontant(1500);
      Destinataire destinataire2 =
                              mandat2.créerDestinataire("Alfred");
      System.out.println("Un mandat de " + montant2.v
                              + " francs a ete expedie a "
                              + destinataire2.nom);
      System.out.println("Un mandat de " + montant2.valeur()
                              + " francs a ete expedie a "
                              + destinataire2.aQui());
   }
}
```

Remarquez la syntaxe utilisée à l'intérieur des méthodes créant les instances :

```
public Montant créerMontant(int m) {
   return new Montant(m);
}
```

Rien n'indique ici dans quelle instance de la classe externe doit être créée l'instance de la classe interne **Montant**. En fait, cette indication est passée automatiquement par le compilateur en fonction de l'appel de la méthode :

```
Montant montant2 = mandat2.créerMontant(1500);
```

Nous aurions ainsi pu écrire la méthode sous la forme :

```
public Montant créerMontant(int m) {
   return this.new Montant(m);
}
```

Le mot clé **this** peut être remplacé par toute expression faisant référence à une instance de la classe externe. La référence ainsi évaluée est passée au constructeur de la classe interne.

Instances externes anonymes

Nous pouvons créer une instance d'une classe interne dans une instance anonyme d'une classe externe. Il est alors tout à fait possible de récupérer une référence à la classe externe, comme dans l'exemple suivant :

```
public class Mandat3 {
   static Mandat3 mandat3;

   class Montant {
      private int v;

      Montant(int m) {
         v = m;
      }

      public int valeur() {
         return v;
      }
   }

   class Destinataire {
      private String nom;

      Destinataire(String qui) {
         nom = qui;
      }

      String aQui() {
         return nom;
      }
   }

   public Montant créerMontant(int m) {
      mandat3 = Mandat3.this;
      return this.new Montant(m);
   }
```

```
   public Destinataire créerDestinataire(String s) {
      return this.new Destinataire(s);
   }

   public static void main (String[] args) {
      Montant montant3 = new Mandat3().créerMontant(1500);
      Destinataire destinataire3 =
                     mandat3.créerDestinataire("Alfred");
      System.out.println("Un mandat de " + montant3.v
                              + " francs a ete expedie a "
                              + destinataire3.nom);
   }
}
```

Il aurait été parfaitement possible de créer une instance de la classe interne **Montant** à l'aide de la syntaxe suivante :

```
public static void main (String[] args) {
   Montant montant3 = (new Mandat3()).new Montant(1500);
```

et de récupérer la référence à la classe externe dans le constructeur de la classe interne :

```
Montant(int m) {
   mandat3 = Mandat3.this;
   v = m;
}
```

Classes membres et héritage

Il ne faut surtout pas confondre la hiérarchie qui résulte de l'extension d'une classe par une autre classe et la hiérarchie impliquée par le fait qu'une classe contient d'autres classes, et cela d'autant plus qu'il est parfaitement possible qu'une classe externe étende une de ses classes membres. (Le fait que cela soit possible ne doit pas vous encourager à utiliser cette particularité.)

Nous avons vu qu'il était possible, dans une classe interne, de faire référence implicitement à un champ d'une classe externe. Il est donc possible également que cette référence implicite conduise à une ambiguïté si la classe interne dérive d'une classe parente possédant un champ de même nom. (En revanche, si la classe interne contient elle-même un champ de même nom, il n'y a pas d'ambiguïté car celui-ci masque le champ de la classe externe.) Voici un exemple :

```
public class Interne3 {
   private int n = 0;
   private Niveau1 o1;
   Interne3() {
      o1 = new Niveau1();
   }
   class Niveau1 {
      private int n1 = 1;
      private Niveau11 o11;
      private Niveau12 o12;

      Niveau1() {
         o11 = new Niveau11();
         o12 = new Niveau12();
      }

      class Niveau11 extends NiveauX {
         private int n11 = 11;

         public void affiche() {
            System.out.println("Acces depuis niveau 2 :");
            System.out.println(Interne3.this.n);
            System.out.println(this.n);
         }
      }

      class Niveau12 {
         private int n12 = 12;
      }
   }
```

```
   public static void main (String[] args) {
      Interne3 i  = new Interne3();
      i.o1.o11.affiche();
   }
}

class NiveauX {
   int n = 99;
}
```

Ici, la classe **Niveau11** étend la classe **NiveauX** qui contient un champ **n**. L'instruction :

```
System.out.println(n);
```

serait donc ambiguë. Une telle instruction provoque donc l'affichage d'un message d'erreur par le compilateur. Il est donc nécessaire de préciser la référence sous la forme :

```
this.n
```

pour désigner le champ de la classe parente, et :

```
Interne3.this.n
```

pour désigner celui de la classe externe. Pour récapituler, l'exemple suivant reprend toutes les possibilités :

```
public class Interne4 {
   private String s = "Classe Interne4";
   private Niveau1 o1;
   Interne4() {
      o1 = new Niveau1();
   }
```

```
   class Niveau1 {
      private int n1 = 1;
      private String s = "Classe Niveau1";
      private Niveau11 o11;

      Niveau1() {
         o11 = new Niveau11();
      }

      class Niveau11 extends NiveauX {
         private int n11 = 11;
         private String s = "Classe Niveau11";

         public void affiche() {
            System.out.println("Acces depuis niveau 2 :");
            System.out.println(Interne4.this.s);
            System.out.println(this.s);
            System.out.println(super.s);
            System.out.println(Interne4.this.o1.s);
         }
      }
   }

   public static void main (String[] args) {
      Interne4 i  = new Interne4();
      i.o1.o11.affiche();
   }
}

class NiveauX {
   String s = "Classe NiveauX";
}
```

Ce programme affiche :

```
Acces depuis niveau 2 :
Classe Interne4
```

```
Classe Niveau11
Classe NiveauX
Classe Niveau1
```

Remarque concernant les classes membres

Comme les classes imbriquées et les autres classes internes, les classes membres ne peuvent pas contenir de membres statiques. Il ne s'agit pas d'une limitation intrinsèque mais simplement d'un choix des concepteurs de Java afin de limiter au maximum les risques d'erreurs en obligeant les programmeurs à déclarer tous les membres statiques au niveau supérieur de la hiérarchie, ce qui est de toute évidence la façon logique de procéder.

Par ailleurs, les classes membres ne peuvent pas porter le même nom qu'une classe ou un package les contenant.

Les classes membres sont accessibles à d'autres classes en utilisant une syntaxe particulière, comme le montre l'exemple suivant :

```
public class Access {
   private Interne5 o;
   Interne5.Niveau1 n1;
   Access() {
      o = new Interne5();
      n1 = o.new Niveau1();
   }

   void affiche() {
      n1.affiche();
   }

   public static void main (String[] args) {
      Access i  = new Access();
      i.o.affiche();
      i.affiche();
   }
}
```

```
class Interne5 {
   Niveau1 o1;
   Interne5() {
      o1 = new Niveau1();
   }

   void affiche() {
      System.out.println(this);
      o1.affiche();
   }

   class Niveau1 {
      int n1 = 1;
      Niveau11 o11;
      Niveau1() {
         o11 = new Niveau11();
      }

      void affiche() {
         System.out.println(this);
         o11.affiche();
      }

      class Niveau11 {
         int n11 = 11;
         public void affiche() {
            System.out.println(this);
            System.out.println();
         }
      }
   }
}
```

Dans cet exemple, deux classes sont définies : **Access** et **Interne5**. La classe **Interne5** définit en outre deux classes membres, **Niveau1** et **Niveau11**. Ce programme affiche le résultat suivant :

```
Interne5@b0e1e784
Interne5$Niveau1@b7b9e784
Interne5$Niveau1$Niveau11@b615e784

Interne5$Niveau1@b611e784
Interne5$Niveau1$Niveau11@b61de784
```

ce qui montre bien que la première instance de **Niveau1** est créée depuis une instance de **Interne5** à l'adresse **b7b9e784** alors que la seconde est créée directement à l'adresse **b611e784**.

La partie intéressante se trouve dans le constructeur de la classe **Access :**

```
Access() {
   o = new Interne5();
   n1 = o.new Niveau1();
}
```

Ici, une instance de la classe **Interne** est créée, ce qui entraîne automatiquement la création d'une instance de chacune des classes membres. A la ligne suivante, une instance de la classe membre **Niveau1** est créée directement et affectée au handle n1, déclaré de type **Interne5.Niveau1**. Cela est possible grâce à l'utilisation d'une syntaxe particulière et grâce au fait que la classe membre **Niveau1** dispose de l'accessibilité par défaut (***package***).

Les classes membres peuvent également être déclarées **public**, **protected** ou **private**. Toutefois, il n'est pas très logique de les déclarer **public**. Si nous modifions le programme de la façon suivante :

```
private class Niveau1 {
   int n1 = 1;
   Niveau11 o11;
```

le compilateur affiche alors le message d'erreur suivant :

```
Inner type Niveau1 in classe Interne5 not accessible from
class Access
```

indiquant que la classe **Niveau1** n'est plus accessible depuis la classe **Access**.

Attention : Ce message est parfaitement normal. Cependant, il est fréquent de compiler les classes à partir de fichiers séparés, ce qui, en principe, devrait fournir le même résultat. Dans ce cas, il n'en est rien. Si vous compilez séparément les classes **Interne5** puis **Access**, le message affiché par le compilateur est alors :

```
Class Interne5 .Niveau1 not found
```

Les classes locales

Les classes locales présentent la même différence par rapport aux classes membres que celle qui existe entre les variables locales et les variables membres : elles sont définies à l'intérieur d'un bloc qui en limite la portée. Elles ne sont donc visibles que dans ce bloc (et dans les blocs qui sont inclus dans celui-ci). Elles peuvent utiliser toutes les variables et tous les paramètres qui sont visibles dans le bloc qui les contient. Cependant, les variables et paramètres définis dans le même bloc ne leur sont accessibles que s'ils sont déclarés **final**. Le programme suivant montre un exemple d'utilisation de classes locales. (Cet exemple, comme les précédents, est complètement stupide et n'a aucun autre intérêt que de montrer les particularités des classes locales. Des exemples concrets seront abordés au Chapitre 18, consacré aux interfaces utilisateurs.)

```
public class Access2 {
   private Interne6 o;

   Access2() {
      o = new Interne6();
   }
```

```
   public static void main (String[] args) {
      Access2 i  = new Access2();
      i.o.afficheCarré(5);
      i.o.afficheCube(5);
   }
}

class Interne6 {

   void afficheCarré(final int x) {
      class Local {
         Local() {
            System.out.println(this);
         }
         long carré() {
            return x * x;
         }
      }
      System.out.println(new Local().carré());
   }

   void afficheCube(final int x) {
      class Local {
         Local() {
            System.out.println(this);
         }
         long cube() {
            return x * x * x;
         }
      }
      System.out.println(new Local().cube());
   }
}
```

Ce programme définit les classes de premier niveau **Access2** et **Interne6**. La classe **Interne6** contient deux définitions de méthodes qui constituent donc deux blocs logiques. Chacun de ces blocs contient une définition dif-

férente de la classe **Local**. Chacune de ces classes contient une méthode qui retourne le carré ou le cube du paramètre passé à la méthode qui contient la classe. Pour que ce paramètre soit accessible dans la classe locale, il est déclaré **final**. Chaque classe interne contient en outre un constructeur qui affiche la classe et l'adresse mémoire de chaque instance créée. Les deux méthodes **afficheCarré** et **afficheCube** affichent leurs résultats grâce aux instructions :

```
System.out.println(new Local().carré());
```

et :

```
System.out.println(new Local().cube());
```

dans lesquelles une instance de la classe locale **Local** est créée.

Ce programme affiche le résultat suivant :

```
Interne6$1$Local@b758c0ff
25
Interne6$2$Local@b6ccc0ff
125
```

Nous voyons ici que le compilateur Java a créé deux classes distinctes appelées **Interne6$1$Local** et **Interne6$2$Local**, ce que nous pouvons vérifier en constatant la présence des deux fichiers **Interne6$1$Local.class** et **Interne6$2$Local.class**.

Notes : Tout comme les classes membres, les classes locales ne peuvent pas contenir de membres statiques (et donc pas de définitions d'interfaces, puisque celles-ci seraient implicitement statiques). Elles ne peuvent pas non plus être déclarées **static**, **public**, **protected** ou **private** car ces modificateurs ne concernent que les membres.

Les handles des instances de classes locales

Le programme de l'exemple précédent créait des instances des classes locales sans toutefois leur affecter des handles. Nous pouvons nous demander s'il est possible de créer des handles vers ces objets, par exemple :

```
class Interne6 {

   void afficheCarré(final int x) {
      Local y;

      class Local {
         Local() {
            System.out.println(this);
         }
         long carré() {
            return x * x;
         }
      }
      y = new Local();
      System.out.println(y.carré());
   }
```

Curieusement, cette façon de faire produit un message d'erreur. Le compilateur Java ne permet pas ici la référence *en avant*, c'est-à-dire qu'il ne nous autorise pas à faire référence à la classe **Local** avant de l'avoir définie. En revanche, il est possible d'utiliser la forme suivante :

```
class Interne6 {

   void afficheCarré(final int x) {

      class Local {
         Local() {
            System.out.println(this);
         }
```

```
      long result() {
        return x * x;
      }
    }
    Local y = new Local();
    System.out.println(y.carré());
  }
```

Et si nous voulions utiliser l'objet en dehors du bloc dans lequel il est créé ? Est-il possible de l'affecter à un handle utilisable en dehors du bloc ? Pour cela, il faudrait tout d'abord pouvoir faire référence à la classe locale d'une façon explicite. En fait, nous savons que les deux classes sont nommées **Interne6$1$Local** et **Interne6$2$Local**. Il nous suffit donc d'utiliser ces noms de la façon suivante :

```
class Interne6 {
  Interne6$1$Local y;
  void afficheCarré(final int x) {
    class Local {
      Local() {
        System.out.println(this);
      }
      long carré() {
        return x * x;
      }
    }
    y = new Local();
  }
```

Attention : Ce type de référence aux classes internes est possible à l'extérieur de celles-ci, mais pas à l'intérieur.

Supposons maintenant que nous modifiions notre programme de la façon suivante :

```
public class Access3 {
   private Interne6 o;
   Access3() {
      o = new Interne6();
   }
   public static void main (String[] args) {
      Access3 i  = new Access3();
      i.o.afficheCarré(5);
      i.o.afficheCube(5);
   }
}
class Interne6 {
   void afficheCarré(final int x) {
      class Local {
         long result() {
            return x * x;
         }
      }
      print(new Local());
   }

   void afficheCube(final int x) {
      class Local {
         long result() {
            return x * x * x;
         }
      }
      print(new Local());
   }

   void print(Interne6$1$Local o) {
      System.out.println(o.result());
   }

   void print(Interne6$2$Local o) {
      System.out.println(o.result());
   }
}
```

Ici, les instances des classes locales sont utilisées à l'extérieur des blocs où elles sont créées. Cependant, la présence des deux méthodes **print** n'est pas satisfaisante car il semble que nous pourrions nous contenter d'une seule. Pour parvenir à ce résultat, deux solutions sont possibles. La première consiste à sur-caster les instances des classes locales en **Object** puis à utiliser une fonction très puissante appelée RTTI (pour *RunTime Type Identification*) que nous étudierons dans un prochain chapitre.

La seconde solution consiste à créer une classe parente disposant de la méthode que l'on souhaite utiliser. En créant les classes locales par extension de cette classe, il devient possible d'y faire référence à l'extérieur du bloc où elles sont créées et d'invoquer leurs méthodes. Pour une telle utilisation, les interfaces sont tout à fait indiquées :

```
public class Access3 {
   private Interne6 o;
   Access3() {
      o = new Interne6();
   }
   public static void main (String[] args) {
      Access3 i  = new Access3();
      i.o.afficheCarré(5);
      i.o.afficheCube(5);
   }
}
class Interne6 {
   interface MonInterface {
      long result();
   }
   void afficheCarré(final int x) {
      class Local implements MonInterface {
         public long result() {
            return x * x;
         }
      }
      print(new Local());
   }
```

```
    void afficheCube(final int x) {
       class Local implements MonInterface {
          public long result() {
             return x * x * x;
          }
       }
       print(new Local());
    }

    void print(MonInterface o) {
       System.out.println(o.result());
    }
 }
```

Un point intéressant à noter dans cet exemple est le fait que les méthodes **result()** des classes locales doivent être déclarées **public**. En l'absence de cette déclaration, la redéfinition de la méthode de l'interface parente entraînerait une restriction de l'accessibilité de celle-ci, ce qui est interdit. Pour plus de détails, reportez-vous au chapitre consacré à l'accessibilité.

Note : Une version de ce programme utilisant les fonctions RTTI sera présentée au Chapitre 17.

Les classes anonymes

Les classes anonymes sont un cas particulier des classes locales. Comme vous l'avez certainement deviné, il s'agit simplement de classes définies sans noms, en utilisant une syntaxe particulière :

```
new nom_de_classe([liste_d'arguments]) {définition}
```

ou :

```
new nom_d'interface () {définition)
```

Dans les expressions ci-dessus, les éléments entre crochets carrés [] sont optionnels. (Les crochets ne doivent pas être tapés.) Les opérateurs **new** ne font pas partie de la déclaration de classe. Cependant, une classe anonyme est généralement utilisée pour la création immédiate d'une instance. Sa définition est donc presque toujours précédée de cet opérateur. Le programme suivant correspond à l'exemple précédent réécrit en utilisant des classes anonymes :

```
public class Access4 {
   private Interne7 o;
   Access4() {
      o = new Interne7();
   }

   public static void main (String[] args) {
      Access4 i  = new Access4();
      i.o.afficheCarré(5);
      i.o.afficheCube(5);
   }
}

class Interne7 {
   void afficheCarré(final int x) {
      System.out.println(new Object() {
         {
            System.out.println(this);
         }
         long result() {
            return x * x;
         }
      }.result());
   }
   void afficheCube(final int x) {
      System.out.println(new Object() {
         {
            System.out.println(this);
         }
```

```
            long result() {
               return x * x * x;
            }
         }.result());
      }
   }
```

La principale particularité des classes anonymes est que, n'ayant pas de noms, elles ne peuvent avoir de constructeurs. Si des tâches d'initialisation doivent être effectuées, ce que nous avons représenté ici par :

```
System.out.println(this);
```

elles doivent l'être au moyen d'*initialiseurs*. Rappelons que les initialiseurs sont des blocs de code qui sont exécutés immédiatement après le constructeur de la classe parente et avant le constructeur de la classe qui les contient, lorsqu'il y en a un. Si la classe contient plusieurs initialiseurs, ils sont exécutés dans l'ordre dans lequel ils se présentent.

Une autre particularité est qu'une classe anonyme doit dériver explicitement d'une autre classe ou interface. En revanche, elle ne peut à la fois étendre une classe et implémenter une interface, ni implémenter plusieurs interfaces. En d'autres termes, l'héritage multiple est impossible.

Les classes anonymes, comme les classes locales, ne peuvent contenir aucun membre **static**. Elles ne peuvent elles-mêmes recevoir aucun modificateur d'accessibilité. Il n'est pas non plus possible de définir des interfaces anonymes.

Dans la plupart des cas, la définition des classes anonymes est incluse dans une expression qui fait elle-même partie d'une instruction. Cette instruction doit donc se terminer par un point-virgule. Celui-ci se trouvant généralement plusieurs lignes après le début de l'instruction, il est facile de l'oublier.

Les classes anonymes sont principalement employées pour la définition de *listeners*, qui sont des classes d'objets destinés à recevoir des messages. Par exemple, si vous créez une fenêtre pour une interface, vous aurez besoin de

recevoir le message envoyé par le système lorsque la fenêtre est fermée. Si la fenêtre est de type **JInternalFrame**, l'objet qui doit recevoir ce message doit être un **InternalFrameListener**. Le programme suivant montre un exemple d'une telle classe :

```
this.addInternalFrameListener(new InternalFrameAdapter() {
   public void internalFrameClosed(InternalFrameEvent e) {
        System.exit(0);
   }
});
```

L'instruction **this.addInternalFrameListener()** ajoute un **InternalFrameListener** à l'objet désigné par **this**. La classe anonyme utilisée n'implémente pas l'interface **InternalFrameListener**, comme on pourrait s'y attendre, mais étend la classe **InternalFrameAdapter**. Cette classe est elle-même une implémentation de l'interface précitée. Cette interface contient six méthodes qui devraient toutes être redéfinies. L'*adapter* correspondant est une classe qui redéfinit ces six méthodes comme ne faisant rien. De cette façon, l'utilisation de l'*adapter* permet de ne redéfinir que les méthodes qui nous intéressent. Ce type d'exemple sera étudié en détail au chapitre consacré aux interfaces utilisateurs et à la création de fenêtres, boutons et autres menus déroulants.

Comment sont nommées les classes anonymes

Toutes anonymes qu'elles soient, ces classes ont tout de même un nom, ne serait-ce que pour que le compilateur puisse les placer dans un fichier. Si nous compilons l'exemple précédent, nous constatons que le compilateur crée quatre fichiers :

- **Access4.class**
- **Interne7.class**
- **Interne7$1.class**
- **Interne7$2.class**

Nous retrouvons ici les mêmes conventions de "nommage" que pour les classes locales. Il est ainsi possible de faire référence explicitement à des classes anonymes en dehors du bloc dans lequel elles sont créées :

```
public class Access4 {
   private Interne7 o;
   Access4() {
      o = new Interne7();
   }
   public static void main (String[] args) {
      Access4 i  = new Access4();
      i.o.afficheCarré(5);
      i.o.afficheCube(5);
   }
}
class Interne7 {
   void afficheCarré(final int x) {
      print(new Object() {
         long result() {
            return x * x;
         }
      });
   }
   void afficheCube(final int x) {
      print(new Object() {
         long result() {
            return x * x * x;
         }
      });
   }
   void print(Interne7$1 o) {
      System.out.println(o.result());
   }
   void print(Interne7$2 o) {
      System.out.println(o.result());
   }
}
```

Résumé

Dans ce chapitre, nous avons étudié toutes les formes de classes internes. Nous reviendrons en détail sur les classes locales et les classes anonymes au chapitre consacré à la construction d'interfaces utilisateurs (fenêtres, boutons, etc.). En attendant, le prochain chapitre sera consacré à un sujet peu passionnant, mais cependant très important : le traitement des erreurs.

Chapitre 13
Les exceptions

Dans tout programme, le traitement des erreurs représente une tâche importante, souvent négligée par les programmeurs. Java dispose d'un mécanisme très efficace pour obliger, autant que faire se peut, les programmeurs à prendre en compte cet aspect de leur travail, tout en leur facilitant la tâche au maximum.

La philosophie de Java, en ce qui concerne les erreurs, peut se résumer à deux principes fondamentaux :

- La plus grande partie des erreurs qui pourraient survenir doit être détectée par le compilateur, de façon à limiter autant que possible les occasions de les voir se produire pendant l'utilisation des programmes.

- Le traitement des erreurs doit être séparé du reste du code, de façon que celui-ci reste lisible. L'examen d'un programme doit faire apparaître, à

première vue, ce que celui-ci est censé faire lors de son fonctionnement normal, et non lorsque des erreurs se produisent.

Stratégies de traitement des erreurs

Lorsqu'une erreur se produit pendant l'exécution d'un programme, deux cas peuvent se présenter :

- L'erreur a été prévue par le programmeur. Dans ce cas, il ne s'agit pas réellement d'une erreur, mais d'un cas exceptionnel. Cela peut se produire, par exemple, si l'utilisateur entre une valeur négative ou une valeur trop grande, lorsqu'on lui demande son âge, ou encore lorsqu'un fichier ne peut être ouvert (par exemple parce qu'il est utilisé par quelqu'un d'autre). Dans ce cas, le programme doit contenir des lignes de codes spécialement prévues pour traiter l'erreur.

- Il s'agit d'une erreur imprévue. Nous sommes alors de nouveau en présence d'une alternative :

    - L'erreur était prévisible. Ce type de situation ne doit pas se produire en Java. En effet, comme nous l'avons dit plus haut, le compilateur Java détecte le risque d'erreurs de ce type et oblige le programmeur à les prendre en compte. Le cas de l'impossibilité d'ouvrir un fichier fait partie de cette catégorie. Si le programmeur écrit une instruction ouvrant un fichier sans prendre en compte le cas où l'ouverture est impossible, le compilateur refuse de compiler le programme.

    - L'erreur était imprévisible. Dans ce cas, il s'agit d'une erreur de conception du programme qui dépend des conditions d'exécution. L'interpréteur produit un message d'erreur et arrête l'exécution. Nous sommes alors en présence d'un "bug".

Nous savons donc maintenant que toutes les erreurs prévisibles doivent être traitées par le programmeur. Cependant, devant la possibilité qu'une erreur survienne, le programmeur peut adopter trois positions.

Signaler et stopper

La première attitude consiste à signaler l'erreur et à arrêter le traitement. Par exemple, si l'ouverture d'un fichier est impossible, le programme affiche un message d'erreur indiquant à l'utilisateur que l'ouverture du fichier n'a pas pu être effectuée, puis le traitement est interrompu.

Corriger et réessayer

En présence d'une erreur, le programme peut tenter de la corriger puis essayer de reprendre le traitement. Dans certains cas, le traitement consiste simplement à attendre. Par exemple, si un fichier ne peut pas être ouvert parce qu'il est verrouillé par un autre utilisateur, il peut suffire d'attendre que celui-ci le déverrouille. Le programme peut ainsi attendre quelques secondes et tenter d'accéder de nouveau au fichier. Il va de soi que si après un certain nombre d'essais l'accès est toujours impossible, il faudra envisager une autre voie. Dans le cas contraire, on risquerait de se trouver devant une boucle infinie.

Dans d'autres cas, par exemple si l'erreur provient d'une valeur hors des limites acceptables, le programme peut corriger lui-même cette valeur au moyen d'algorithmes divers (par exemple prendre une valeur par défaut).

Signaler et réessayer

En présence d'une erreur, le programme peut aussi signaler celle-ci et demander à l'utilisateur de la corriger. Cette approche est évidemment adaptée aux cas où l'erreur provient d'une saisie incorrecte.

La stratégie de Java

Java ne se préoccupe pas des deux dernières options, qui sont laissées à la discrétion du programmeur. En revanche, la première option est imposée au programmeur (ce qui ne l'empêche pas de mettre en œuvre les deux autres).

Il peut paraître tout à fait insuffisant de simplement signaler une erreur et arrêter le traitement. C'est effectivement le cas avec les programmes qui doivent fonctionner de façon autonome. Cependant, avec ce type de programme, le compilateur ne peut pas faire grand-chose. C'est au programmeur qu'il revient d'imaginer tous les cas d'erreurs possibles et les traitements à mettre en œuvre pour que l'exécution puisse continuer. Si une erreur non prévue se produit, il n'y a alors pas d'autre solution que d'arrêter le traitement.

En revanche, avec un programme interactif, la solution *signaler et stopper* est tout à fait appropriée. En effet, *stopper* n'implique pas l'arrêt du programme, mais simplement l'arrêt du traitement en cours.

Supposons, par exemple, qu'un programme permette à un utilisateur d'ouvrir un fichier. Le programme affiche une *interface utilisateur* (rien à voir avec les interfaces de Java) comportant, par exemple, un bouton *Ouvrir*. Tant que l'utilisateur ne fait rien, le programme ne fait rien non plus. S'il clique sur le bouton, le programme affiche une boîte de dialogue comportant la liste des fichiers présents sur le disque et un bouton *OK*, puis il s'arrête de nouveau, attendant que l'utilisateur sélectionne un fichier et clique sur *OK*. Une fois cela fait, le programme tente d'ouvrir le fichier. S'il y parvient, il en affiche le contenu (par exemple). Dans le cas contraire, il affiche un message d'erreur et s'arrête. Cela ne signifie pas que le programme soit stoppé et effacé de la mémoire. Il s'arrête simplement et attend la suite. L'utilisateur peut alors corriger l'erreur, et sélectionner de nouveau le même fichier, ou choisir un autre fichier, ou quitter le programme. Ce qui est important ici est que ces trois options ne font pas partie du traitement de l'erreur. Du point de vue du programme, le traitement de l'erreur consiste simplement à la signaler.

Les deux types d'erreurs de Java

En Java, on peut classer les erreurs en deux catégories :

- Les erreurs surveillées,
- Les erreurs non surveillées.

Java oblige le programmeur à traiter les erreurs surveillées. Les erreurs non surveillées sont celles qui sont considérées trop graves pour que leur traitement soit prévu à priori. Par exemple, lorsque vous écrivez :

```
int x, y, z;
   .
   .
   .
z = x / y;
```

une erreur se produira si **y** vaut 0. Il s'agit là cependant d'une erreur non surveillée. Les concepteurs de Java ont considéré qu'il n'était pas possible d'obliger les programmeurs à traiter à priori ce type d'erreur. Parfois, il existe un risque qu'elle se produise. Par exemple, si nous écrivons un programme permettant de calculer la vitesse d'un véhicule, connaissant le temps qu'il met à parcourir un kilomètre, il se produira immanquablement une erreur si l'utilisateur entre 0 pour le temps. Ce type d'erreur est cependant parfaitement prévisible et il revient au programmeur d'en tenir compte. Si vous compilez le programme suivant :

```
public class Erreur {
   public static void main( String[] args ) {
      int x = 10, y = 0, z = 0;
      z = x / y;
   }
}
```

il est évident qu'il produira une erreur. Pourtant, le compilateur ne s'en préoccupe pas. Cette erreur appartient à la catégorie des erreurs non surveillées. L'exécution de ce programme produit le résultat suivant :

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
  at Erreur.main(Erreur.java:5)
```

Ce message nous apprend plusieurs choses. Tout d'abord, le fait qu'il soit affiché indique que l'erreur a été traitée. Si ce n'était pas le cas, l'interpré-

teur ne saurait pas quoi faire. Ici, il sait parfaitement. Un message est affiché et l'exécution est interrompue. Il ne s'agit donc pas d'une erreur mais d'un "traitement exceptionnel".

Les exceptions

Lorsqu'une erreur de ce type est rencontrée (ici dans la méthode **main** de la classe **Erreur**), l'interpréteur crée immédiatement un objet instance d'une classe particulière, elle-même sous-classe de la classe **Exception**. Cet objet est créé normalement à l'aide de l'opérateur **new**. Puis l'interpréteur part à la recherche d'une portion de code capable de recevoir cet objet et d'effectuer le traitement approprié. S'il s'agit d'une erreur surveillée par le compilateur, celui-ci a obligé le programmeur à fournir ce code. Dans le cas contraire, le traitement est fourni par l'interpréteur lui-même. Cette opération est appelée "lancement (en anglais *throw*) d'une exception", par analogie avec le lancement d'un objet qui doit être attrapé par le code prévu pour le traiter.

Pour trouver le code capable de traiter l'objet, l'interpréteur se base sur le type de l'objet, c'est-à-dire sur la classe dont il est une instance. S'il existe un tel code, l'objet lui est transmis, et l'exécution se poursuit à cet endroit. Il faut noter que le premier bloc de code capable de traiter l'objet reçoit celui-ci. A cause du polymorphisme, il peut s'agir d'un bloc de code traitant une classe parente de celle de l'objet en question.

Dans le cas de notre programme précédent, une instance de la classe **ArithmeticException** est donc lancée. Notre programme ne comportant aucun bloc de code capable de traiter cet objet, celui-ci est attrapé par l'interpréteur lui-même. Le traitement effectué consiste à afficher la chaîne de caractères "**Exception in thread**" suivie du nom de la méthode d'où l'objet a été lancé, du nom de la classe de l'objet précédé du nom du package auquel elle appartient, d'un message décrivant l'erreur ("**/ by zero**") et d'une indication sur l'origine de l'erreur "**at Erreur.main(Erreur.java:5)**". Nous allons revenir sur chacun de ces éléments.

- **Exception int thread "main"**

Cet élément est fourni par l'interpréteur sur la base des indications extraites des éléments suivants.

- **java.lang.ArithmeticException**

Il s'agit là simplement du nom de la classe de l'objet reçu. Cette indication peut être déterminée au moyen d'un procédé que nous n'avons pas encore étudié. Java permet en effet, en présence d'un objet, de déterminer la classe dont il est une instance.

- **/ by zero**

La classe **ArithmeticException** comporte deux constructeurs. Le premier ne prend aucun paramètre. Le deuxième prend pour paramètre une chaîne de caractères. Lorsque l'instance d'**ArithmeticException** a été créée, elle l'a été avec pour paramètre la chaîne de caractères "**/ by zero**". C'est cette chaîne qui est restituée ici.

- **at Erreur.main(Erreur.java:5)**

Lors de l'exécution d'un programme, l'interpréteur Java tient à jour la pile des appels de méthode. Chaque fois qu'une méthode est appelée, une entrée est ajoutée sur la pile avec l'emplacement de l'appel et le nom de la méthode appelante. Le lancement d'une exception est assimilable à un appel de méthode. Au moment du lancement de l'exception, l'entrée **Erreur.main(Erreur.java:5)** a été ajoutée sur la pile. **Erreur.main** est le nom de la méthode appelante. **Erreur.java:5** est l'emplacement de l'appel, constitué du nom du fichier et du numéro de ligne.

Attraper les exceptions

Nous avons vu que Java n'oblige pas le programmeur à attraper tous les types d'exceptions. Seuls ceux correspondant à des erreurs *surveillées* doivent obligatoirement être attrapés. En fait, les exceptions qui peuvent ne pas être attrapées sont les instances de la classe **RuntimeException** ou d'une classe dérivée de celle-ci.

Cependant, rien n'interdit d'attraper ces exceptions. Nous avons dit que lorsqu'une exception est lancée, l'interpréteur part à la recherche d'un bloc de code susceptible de la traiter. Il fait cela en commençant par parcourir le code. S'il s'agit d'une exception surveillée, il trouve forcément un tel bloc. Dans le cas d'une exception non surveillée, deux cas se présentent :

- Un bloc de code est trouvé. Le contrôle est passé à ce bloc avec, pour paramètre, un handle vers l'objet exception.

- Aucun bloc n'est trouvé. Le contrôle est passé au code de traitement des exceptions de type **RuntimeException** de l'interpréteur.

Pour pouvoir attraper une exception, il faut entourer le code susceptible de la lancer dans un bloc précédé de l'instruction **try**. L'exception lancée dans le bloc peut être attrapée par une sorte de méthode d'un type particulier désignée par le mot clé **catch** et prenant pour paramètre un objet du type de l'exception lancée (ou, grâce au polymorphisme du type d'une classe parente). Nous pouvons réécrire le programme précédent pour attraper l'exception lancée :

```
public class Erreur1 {
     public static void main( String[] args ) {
     int x = 10, y = 0, z = 0;
     try {
        z = x / y;
     }
     catch (ArithmeticException e) {
     }
   }
}
```

Ici, la ligne susceptible de lancer une exception est placée dans un bloc **try**. Si une exception est lancée dans ce bloc, l'interpréteur cherche un bloc **catch** susceptible de la traiter. Le premier trouvé fait l'affaire. L'exception lui est passée en paramètre. Ici, le bloc **catch** ne fait tout simplement rien. Si vous exécutez ce programme, vous constaterez qu'il n'affiche aucun message d'erreur.

Aucune ligne de code ne doit se trouver entre le bloc **try** et le bloc **catch**. Cependant, le bloc **try** peut inclure plusieurs lignes susceptibles de lancer une exception.

Dans quelle direction sont lancées les exceptions ?

Lorsqu'une exception est lancée, nous avons dit que l'interpréteur cherchait un bloc **catch** capable de la traiter, et que, s'il n'en trouvait pas, il la traitait lui-même. Mais dans quelle direction effectue-t-il sa recherche ? La recherche est effectuée à partir de l'endroit où l'exception est créée, et vers le bas, c'est-à-dire dans la suite du code. Cependant, la recherche ne continue pas jusqu'à la fin du fichier, mais seulement jusqu'à la fin du bloc logique contenant le bloc **try**. Si aucun bloc **catch** (également appelé *handler d'exception* – attention à ne pas confondre *handler* et *handle*) n'est trouvé, l'exception remonte vers le bloc de niveau supérieur. Nous pouvons en faire la démonstration à l'aide du programme suivant :

```
public class Erreur2 {
   public static void main( String[] args ) {
      int x = 10, y = 0, z = 0;
      try {
         z = calcul(x, y);
      }
      catch (ArithmeticException e) {
      }
   }
   static int calcul(int a, int b) {
      return a / b;
   }
}
```

Ici, l'exception est lancée dans la méthode **calcul**. Celle-ci ne comporte pas de handler (bloc **catch**) pour cette exception. L'exception remonte alors dans le bloc de niveau supérieur, c'est-à-dire celui où a eu lieu l'appel de la méthode. Là, un handler est trouvé et l'exception est traitée.

Manipuler les exceptions

Le programme ci-dessus ne fait rien de bien intéressant en matière de traitement d'exception. Nous allons le modifier afin qu'il affiche les mêmes informations que l'interpréteur :

```
public class Erreur3 {
   public static void main( String[] args ) {
      int x = 10, y = 0, z = 0;
      try {
         z = calcul(x, y);
      }
      catch (ArithmeticException e) {
         System.out.print("Exception in thread \"main\" ");
         e.printStackTrace();
      }
   }
   static int calcul(int a, int b) {
      return a / b;
   }
}
```

Ce programme affiche exactement le même message que la première version.

Note : Vous vous demandez peut-être s'il est possible d'afficher de façon dynamique le nom de la méthode dans laquelle est lancée l'exception. Cela est tout à fait possible mais nécessite des connaissances que nous n'avons pas encore abordées.

Créer un handler d'erreur pour afficher le même message que l'interpréteur n'a évidemment pas beaucoup l'intérêt. En fait, il peut exister deux raisons pour créer des handlers pour les exceptions de type **RuntimeException** :

- Empêcher l'exécution du handler de l'interpréteur ;
- Compléter le handler de l'interpréteur.

Nous supposerons donc que nous souhaitons compléter l'affichage du handler de l'interpréteur en ajoutant un message plus explicite (du moins pour des utilisateurs francophones). Pour cela, nous créerons un handler qui affichera le message voulu puis appellera le handler de l'interpréteur. Comme nous l'avons vu, notre handler attrape l'exception lancée. Pour exécuter le handler de l'interpréteur après avoir affiché un message, il suffit de relancer cette exception, de la façon suivante :

```
public class Erreur4 {
   public static void main( String[] args ) {
      int x = 10, y = 0, z = 0;
      try {
         z = calcul(x, y);
      }
      catch (ArithmeticException e) {
         System.out.println("Erreur : division par zero.");
         throw e;
      }
   }
   static int calcul(int a, int b) {
      return a / b;
   }
}
```

Ce programme affiche :

```
Erreur : division par zero.
Exception in thread "main" java.lang.ArithmeticException: / by zero
   at Erreur4.main(Erreur4.java:5)
```

Il faut noter ici un point important : le message d'erreur affiché par le handler de l'interpréteur indique que l'exception s'est produite à la ligne 5, alors qu'elle a été lancée à la ligne 9. La raison en est très simple. Le handler ne se préoccupe de vérifier ni la provenance ni le contenu de l'exception. Comme l'exception qui est lancée à la ligne 9 est celle créée à la ligne 5, elle contient les mêmes informations. Pour obtenir un résultat différent, Java pro-

pose deux options. La première consiste à modifier l'exception avant de la relancer. Une exception possède trois caractéristiques importantes (qui sont, en fait, héritées de la classe parente **Throwable**) :

- Son type ;
- Le message qu'elle contient ;
- Son origine.

Le message qu'elle contient ne peut pas être modifié car il s'agit d'un champ privé auquel ne correspond aucun mutateur. En revanche, nous pouvons agir sur son type et sur son origine.

Modification de l'origine d'une exception

Si nous voulons que l'exception soit renvoyée avec pour origine l'endroit où le renvoi est effectué, il nous suffit d'utiliser la méthode **fillInStackTrace()** de la façon suivante :

```
public class Erreur5 {
   public static void main( String[] args ) {
      int x = 10, y = 0, z = 0;
      try {
         z = calcul(x, y);
      }
      catch (ArithmeticException e) {
         System.out.println("Erreur : division par zero.");
         e.fillInStackTrace();
         throw e;
      }
   }
   static int calcul(int a, int b) {
      return a / b;
   }
}
```

Le programme affiche maintenant :

```
Erreur : division par zero.
Exception in thread "main" java.lang.ArithmeticException: / by zero
   at Erreur5.main(Erreur5.java:9)
```

Une autre façon de modifier l'exception est de changer son type. Cela n'est évidemment pas possible n'importe comment et ne peut être fait qu'au moyen d'un sur-casting, par exemple de la façon suivante :

```
public class Erreur6 {
   public static void main( String[] args ) {
      int x = 10, y = 0, z = 0;
      try {
         z = calcul(x, y);
      }
      catch (ArithmeticException e) {
         System.out.println("Erreur : division par zero.");
         throw (RuntimeException)e;
      }
   }
   static int calcul(int a, int b) {
      return a / b;
   }
}
```

Ici, la modification n'a pas beaucoup d'influence sur le résultat, car le handler d'exception de l'interpréteur traite toutes les exceptions de la même façon, en affichant leur origine et le message qu'elles contiennent puis en arrêtant l'exécution. En revanche, nous verrons dans une prochaine section que cette technique peut permettre d'effectuer un traitement spécifique à une exception particulière, puis de renvoyer celle-ci pour qu'elle soit interceptée de nouveau par un handler traitant les cas plus généraux.

Ici, nous avons renvoyé une exception de la classe parente **RuntimeException**, ce qui ne change pas grand-chose car il s'agit également d'un type non surveillé. En revanche, si nous voulons sur-caster l'exception au niveau supérieur, c'est-à-dire **Exception,** le compilateur affiche le message d'erreur suivant :

```
Erreur6.java:9 Exception java.lang.Exception must be caught,
or it must be declared in the throws clause of this method.
```

Cela est dû au fait que les exceptions de cette classe sont surveillées et doivent donc recevoir, de la part du programmeur, un traitement spécial. Ce traitement peut prendre deux formes :

- Attraper l'exception.
- Signaler que l'exception n'est pas attrapée.

Java n'est donc vraiment pas contraignant dans ce domaine, d'autant que "attraper l'exception" ne vous oblige pas à la traiter !

Ici, attraper l'exception n'aurait aucun sens. On ne lance pas une exception pour l'attraper au même endroit ! C'est pourtant possible, comme le montre l'exemple suivant :

```
public class Erreur7 {
   public static void main( String[] args ){
      int x = 10, y = 0, z = 0;
      try {
         z = calcul(x, y);
      }
      catch (ArithmeticException e) {
         System.out.println("ArithmeticException attrapee ligne 8.");
         try {
            throw (Exception)e;
         }
         catch (Exception e2) {
            System.out.println("Exception attrapée ligne 13.");
         }
      }
   }
   static int calcul(int a, int b) {
      return a / b;
   }
}
```

Cet exemple n'a évidemment aucun sens.

L'autre solution consiste à indiquer que la méthode est susceptible de lancer une exception. Ici, la méthode concernée est **main**. Sa déclaration doit être modifiée de la façon suivante :

```
public class Erreur8 {
   public static void main( String[] args ) throws Exception {
      int x = 10, y = 0, z = 0;
      try {
         z = calcul(x, y);
      }
      catch (ArithmeticException e) {
         System.out.println("ArithmeticException attrapee ligne 8.");
         try {
            throw (Exception)e;
         }
   }
   static int calcul(int a, int b) {
      return a / b;
   }
}
```

Un tel exemple n'a pas beaucoup d'intérêt non plus ! Dans la pratique, le fait d'indiquer qu'une méthode est susceptible de lancer une exception oblige l'utilisateur de cette méthode à l'attraper ou à la transmettre. Par exemple, la méthode **readLine()** de la classe **BufferedReader** permet de lire une ligne saisie par l'utilisateur. Si nous consultons la documentation de Java, nous constatons que sa signature est :

```
public String readLine() throws IOException
```

Si nous voulons créer une méthode **demander**() qui crée un objet de type **BufferedReader**, lit une ligne et renvoie le résultat, nous pouvons le faire de la façon suivante :

```
public static int demander(String s) {
   System.out.print(s);
   BufferedReader br =
             new BufferedReader(new InputStreamReader(System.in));
   try {
      return (new Integer(br.readLine())).intValue();
   }
   catch (IOException e) {
      System.out.println("Erreur de lecture.");
   }
}
```

Cependant, une telle méthode ne pourra pas être compilée. En effet, le compilateur constate que, si une erreur se produit dans le bloc **try**, la méthode ne retourne pas. Une solution consiste à ajouter une instruction **return** dans le bloc **catch** :

```
   catch (IOException e) {
      System.out.println("Erreur de lecture.");
      return 0;
   }
```

ou après celui-ci :

```
      catch (IOException e) {
         System.out.println("Erreur de lecture.");
      }
      return 0;
```

La deuxième solution n'est vraiment pas à conseiller. Une autre façon, plus élégante, consiste à sortir l'instruction **return** du bloc **try**. Elle présente l'inconvénient d'empêcher l'anonymat de la valeur de retour :

```
   public static int demander(String s) {
      int f = 0;
```

```
   System.out.print(s);
   BufferedReader br =
             new BufferedReader(new InputStreamReader(System.in));
   try {
      f = (new Integer(br.readLine())).intValue();
   }
   catch (IOException e) {
      System.out.println("Erreur de lecture.");
   }
   return f;
}
```

Le programme complet utilisant la méthode **demander()** pourrait être le suivant :

```
import java.io.*;

public class Erreur9 {
   public static void main( String[] args ) {
      int x = demander("Temps en secondes pour 1 km : ");
      int z = calcul(x);
      System.out.println("La vitesse est de " + z + " km/h.");
   }
   static int calcul(int a) {
      int x = 0;
      try {
         x = 3600 / a;
      }
      catch (ArithmeticException e) {
         System.out.println("Erreur : division par zero");
      }
      return x;
   }

   public static int demander(String s) {
      int f = 0;
      System.out.print(s);
```

```
      BufferedReader br =
           new BufferedReader(new InputStreamReader(System.in));
      try {
         f = (new Integer(br.readLine())).intValue();
      }
      catch (IOException e) {
         System.out.println("Erreur de lecture.");
      }
      return f;
   }
}
```

Si maintenant nous décidons de ne pas attraper l'exception **IOException** dans la méthode **demander()**, nous devons le signaler afin que les utilisateurs de cette méthode soient avertis que cette exception risque de remonter vers la méthode appelante. Nous le faisons en ajoutant une indication dans la déclaration de la méthode (nous avons retiré le traitement de la division par zéro pour plus de clarté) :

```
   public static int demander(String s) throws IOException {
      System.out.print(s);
      BufferedReader br =
           new BufferedReader(new InputStreamReader(System.in));
      return (new Integer(br.readLine())).intValue();
   }
```

Le problème est ainsi évacué de la méthode. L'utilisateur de cette méthode sait maintenant qu'il doit attraper les exceptions de type **IOException** ou bien indiquer qu'elles doivent être relancées. Le programme complet pourra être le suivant :

```
import java.io.*;

public class Erreur10 {
   public static void main( String[] args ) {
      int x = 0;
```

```
      try {
         x = demander("Temps en secondes pour 1 km : ");
      }
      catch (IOException e) {
         System.out.println("Erreur de lecture.");
      }
      int z = calcul(x);
      System.out.println("La vitesse est de " + z + " km/h.");
   }

   static int calcul(int a) {
      return 3600 / a;
   }

   public static int demander(String s) throws IOException {
      System.out.print(s);
      BufferedReader br =
           new BufferedReader(new InputStreamReader(System.in));
      return (new Integer(br.readLine())).intValue();
   }
}
```

Notez que d'autres petites modifications ont dû être effectuées pour que le programme fonctionne. En effet, placer une ligne dans un bloc **try** limite automatiquement à ce bloc la portée des variables qui y sont déclarées. Pour que **x** puisse être utilisée pour le calcul et l'affichage, elle doit être déclarée en dehors du bloc.

Une autre solution consiste à ne pas traiter l'erreur et la relancer implicitement :

```
import java.io.*;

public class Erreur11 {
   public static void main( String[] args ) throws IOException {
      int x = demander("Temps en secondes pour 1 km : ");
      int z = calcul(x);
```

```
      System.out.println("La vitesse est de " + z + " km/h.");
   }

   static int calcul(int a) {
      return 3600 / a;
   }

   public static int demander(String s) throws IOException {
      System.out.print(s);
      BufferedReader br =
           new BufferedReader(new InputStreamReader(System.in));
      return (new Integer(br.readLine())).intValue();
   }
}
```

Dans ce cas, l'exception remonte jusqu'à l'interpréteur.

Créer ses propres exceptions

Les exceptions sont des objets comme les autres. En particulier, les classes d'exceptions peuvent être étendues. Ainsi, dans notre exemple, rien n'empêche de rentrer une valeur négative. Cependant, si nous voulons l'empêcher, nous pouvons créer une classe spéciale d'exceptions. Le programme suivant en montre un exemple et permet d'illustrer d'autres particularités du traitement des exceptions :

```
import java.io.*;

public class Erreur12 {
   public static void main( String[] args ) {
      int x = 0;
      int z = 0;
      try {
         x = demander("Temps en secondes pour 1 km : ");
      }
```

```
      catch (IOException e) {
         System.out.println("Erreur de lecture.");
      }
      try {
         z = calcul(x);
      }
      catch (NegativeValueException e) {
         System.out.println("Erreur : " + e.getMessage());
      }
      catch (ArithmeticException e) {
         System.out.println("Erreur : division par zero.");
      }
      System.out.println("La vitesse est de " + z + " km/h.");
   }
   static int calcul(int a) {
      if (a < 0)
         throw new NegativeValueException("Valeur negative");
      return 3600 / a;
   }
      public static int demander(String s) throws IOException {
      System.out.print(s);
      BufferedReader br =
           new BufferedReader(new InputStreamReader(System.in));
      return (new Integer(br.readLine())).intValue();
   }
}

class NegativeValueException extends ArithmeticException {
   NegativeValueException(String s) {
      super(s);
   }
}
```

La classe **NegativeValueException** est créée dans le même fichier que le programme pour les besoins de l'exemple. Normalement, elle devrait probablement être créée de façon indépendante. Elle étend simplement la classe

ArithmeticException et déclare un constructeur qui fait simplement appel à celui de la classe parente en lui passant son paramètre.

La méthode **calcul** lance une exception de type **NegativeValueException** si la valeur qui lui est passée en paramètre est négative. Cette exception est créée normalement, comme n'importe quel autre objet, et est utilisée comme argument de l'instruction **throw**. Notez que la déclaration de la méthode n'indique pas que celle-ci est susceptible de lancer ce type d'exception. Cette déclaration n'est pas ici obligatoire. De la même façon, le bloc **catch** correspondant n'est pas obligatoire. En effet, la classe **NegativeValueException** dérive de **ArithmeticException**, et ce type d'exception n'est pas surveillé. L'utilisateur de la méthode **demander** est donc libre d'attraper ou non cette exception.

Que se passe-t-il s'il ne le fait pas ? L'exception lancée remonte jusqu'à l'interpréteur. Celui-ci ne dispose d'aucun handler pour le type **NegativeValueException**. Cependant, il dispose d'un handler pour la classe parente. Grâce au polymorphisme, c'est ce handler qui est exécuté. Evidemment, le résultat n'est pas celui que nous souhaitions car le message indique qu'il s'agit d'une division par zéro.

Notez qu'il existe un autre danger dans le fait de dériver notre classe d'exception d'une classe explicitement utilisée. En effet, si nous inversons l'ordre des blocs **catch** :

```
    try {
       z = calcul(x);
    }
    catch (ArithmeticException e) {
       System.out.println("Erreur : division par zero.");
    }
    catch (NegativeValueException e) {
       System.out.println("Erreur : " + e.getMessage());
    }
    System.out.println("La vitesse est de " + z + " km/h.");
  }
```

Le programme ne peut plus être compilé. Le compilateur affiche alors le message d'erreur :

```
Erreur12.java:19: catch not reached
                catch (NegativeValueException e) {
                ^
1 error
```

pour indiquer que le bloc **catch** ne sera jamais exécuté. En effet, nous avons dit que Java cherche le premier bloc susceptible de correspondre au type d'exception lancée. En raison du polymorphisme, le premier bloc convient parfaitement. Le second ne sera donc jamais atteint.

Pour remédier à ce problème, il nous faut dériver notre classe d'une classe d'exception non utilisée par ailleurs. D'autre part, il est préférable d'utiliser un type d'exception surveillé si nous voulons que les utilisateurs de notre méthode soient obligés de la traiter. Nous pouvons le faire en modifiant simplement sa déclaration :

```
class NegativeValueException extends Exception {

   NegativeValueException(String s) {
      super(s);
   }
}
```

Cette fois, si nous conservons la déclaration et la définition de la méthode **calcul** inchangées :

```
static int calcul(int a) {
   if (a < 0)
      throw new NegativeValueException("Valeur negative");
   return 3600 / a;
}
```

le compilateur produit deux messages d'erreur :

```
Erreur13.java:19: Exception NegativeValueException is never thrown
in teh body of the correspondind try statement.
          catch (NegativeValueException e) {
          ^
Erreur13.java:27: Exception NegativeValueException must be caught or
it must be declared in teh throw clause of this method.
          throw new NegativeValueException ("Valeur negative") {
          ^
2 errors
```

Nous connaissons déjà le deuxième. Le premier message indique que nous essayons d'attraper une exception qui n'est jamais lancée. L'exemple suivant montre le programme corrigé :

```
import java.io.*;
public class Erreur13 {
   public static void main( String[] args ) {
      int x = 0;
      int z = 0;
      try {
         x = demander("Temps en secondes pour 1 km : ");
      }
      catch (IOException e) {
         System.out.println("Erreur de lecture.");
      }
      try {
         z = calcul(x);
      }
      catch (ArithmeticException e) {
         System.out.println("Erreur : division par zero.");
      }
      catch (NegativeValueException e) {
         System.out.println("Erreur : " + e.getMessage());
      }
```

```
      System.out.println("La vitesse est de " + z + " km/h.");
   }

   static int calcul(int a) throws NegativeValueException {
      if (a < 0)
         throw new NegativeValueException("Valeur negative");
      return 3600 / a;
   }

   public static int demander(String s) throws IOException {
      System.out.print(s);
      BufferedReader br =
           new BufferedReader(new InputStreamReader(System.in));
      return (new Integer(br.readLine())).intValue();
   }
}

class NegativeValueException extends Exception {

   NegativeValueException(String s) {
      super(s);
   }
}
```

La clause *finally*

Ce programme n'est pas encore satisfaisant. En effet, si nous l'exécutons, voici le résultat obtenu :

```
Temps en seconde pour 1 km : -5
Erreur : Valeur Negative
La vitesse est de 0 km/h.
```

Ici, malgré l'erreur, le programme affiche un résultat. Ce que nous souhaiterions, par exemple, serait que le programme soit interrompu. La même chose

doit se produire s'il s'agit d'une division par 0. Nous pourrions bien sûr ajouter à chacun des blocs **catch** une instruction terminant le programme, comme :

```
System.exit(0);
```

Cependant, cela est contraire aux principes de la programmation efficace, qui prescrivent que le même traitement ne doit pas être effectué à plusieurs endroits différents. La meilleure solution serait d'indiquer qu'un certain traitement doit être effectué quel que soit le type d'erreur rencontrée. Une solution serait d'appeler une méthode qui arrête le programme :

```
static void arrêteProgramme() {
System.exit(0);
}
```

Bien sûr, l'appel de la méthode serait dupliqué dans chaque bloc, mais cela ne contredit pas nos principes :

```
   try {
      z = calcul(x);
   }
   catch (ArithmeticException e) {
      System.out.println("Erreur : division par zero.");
      arrêteProgramme();
   }
   catch (NegativeValueException e) {
      System.out.println("Erreur : " + e.getMessage());
      arrêteProgramme();
   }
   System.out.println("La vitesse est de " + z + " km/h.");
}

static void arrêteProgramme() {
   System.exit(0);
}
```

Une autre solution serait de relancer une exception de type **RuntimeException**, qui remonterait jusqu'à l'interpréteur et provoquerait l'arrêt du programme. Ce n'est ni élégant, ni propre, en raison de l'affichage provoqué par le handler d'exception de l'interpréteur.

```
try {
   z = calcul(x);
}
catch (ArithmeticException e) {
   System.out.println("Erreur : division par zero.");
   throw new RuntimeException();
}
catch (NegativeValueException e) {
   System.out.println("Erreur : " + e.getMessage());
   throw new RuntimeException();
}
System.out.println("La vitesse est de " + z + " km/h.");
```

Java dispose d'une instruction spéciale pour traiter ce cas. Il s'agit simplement d'un bloc ajouté après les blocs **catch** et indiqué par le mot clé **finally**. Ce bloc est exécuté après que n'importe quel bloc **catch** de la structure a été exécuté. Malheureusement, il est également exécuté si aucun de ces blocs ne l'est. Cette structure est particulièrement utile si un nettoyage doit être effectué, par exemple dans le cas de l'ouverture d'un fichier, qui devra être refermé même si aucune erreur ne s'est produite.

Dans notre cas, nous pouvons utiliser cette structure à condition de détecter si une erreur s'est produite ou non. Le programme peut donc être réécrit de la façon suivante :

```
import java.io.*;

public class Erreur14 {
   public static void main( String[] args ) {
      int x = 0;
      int z = 0;
      try {
```

```
            x = demander("Temps en secondes pour 1 km : ");
         }
         catch (IOException e) {
            System.out.println("Erreur de lecture.");
         }
         try {
            z = calcul(x);
         }
         catch (ArithmeticException e) {
            System.out.println("Erreur : division par zero.");
         }
         catch (NegativeValueException e) {
            System.out.println("Erreur : " + e.getMessage());
         }
         finally {
            if (z == 0)
               System.exit(0);
         }
         System.out.println("La vitesse est de " + z + " km/h.");
      }
      static int calcul(int a) throws NegativeValueException {
         if (a < 0)
            throw new NegativeValueException("Valeur negative");
         return 3600 / a;
      }
      public static int demander(String s) throws IOException {
         System.out.print(s);
         BufferedReader br =
               new BufferedReader(new InputStreamReader(System.in));
         return (new Integer(br.readLine())).intValue();
      }
   }
   class NegativeValueException extends Exception {
      NegativeValueException(String s) {
         super(s);
      }
   }
```

Organisation des handlers d'exceptions

L'organisation des handlers d'exceptions doit être envisagée soigneusement. En effet, il n'est pas toujours évident de savoir comment les regrouper. Ainsi, notre programme pourrait également être écrit de la façon suivante :

```
import java.io.*;
public class Erreur15 {
   public static void main( String[] args ) {
      int x = 0;
      int z = 0;
      try {
         x = demander("Temps en secondes pour 1 km : ");
         z = calcul(x);
      }
      catch (IOException e) {
         System.out.println("Erreur de lecture.");
      }
      catch (ArithmeticException e) {
         System.out.println("Erreur : division par zero.");
      }
      catch (NegativeValueException e) {
         System.out.println("Erreur : " + e.getMessage());
      }
      finally {
         System.exit(0);
      }
      System.out.println("La vitesse est de " + z + " km/h.");
   }

   static int calcul(int a) throws NegativeValueException {
      if (a < 0)
         throw new NegativeValueException("Valeur negative");
      return 3600 / a;
   }
```

```
   public static int demander(String s) throws IOException {
      System.out.print(s);
      BufferedReader br =
           new BufferedReader(new InputStreamReader(System.in));
      return (new Integer(br.readLine())).intValue();
   }
}

class NegativeValueException extends Exception {

   NegativeValueException(String s) {
      super(s);
   }
}
```

Ici, il n'existe plus qu'une seule structure, dans laquelle toutes les erreurs possibles sont traitées. Il revient au programmeur de décider quel type d'organisation est le mieux adapté au problème à traiter.

Pour un véritable traitement des erreurs

Les procédures que nous avons décrites jusqu'ici consistent à signaler l'erreur et à arrêter le traitement. Elles sont adaptées à certains types de programmes. Cependant, il est souvent préférable d'essayer de corriger les erreurs et de reprendre le traitement. Java ne propose pas de moyens particuliers pour cela. Il revient donc au programmeur de s'en préoccuper. Par exemple, dans le programme précédent, deux types d'erreurs au moins peuvent être traités. En fait, deux types d'erreurs se ramènent même à un seul traitement. Ainsi, le fait qu'une valeur soit négative ou nulle peut parfaitement se traiter de la même façon, en essayant d'obtenir une autre valeur. Le programme suivant montre une façon de traiter le problème :

```
import java.io.*;

public class Erreur17 {
```

```
   public static void main( String[] args ) {
      int x = 0;
      int z = 0;
      x = demanderVP("Entrez le temps en secondes pour 1 km");
      z = calcul(x);
      System.out.println("La vitesse est de " + z + " km/h.");
   }

   static int calcul(int a) {
      return 3600 / a;
   }

   public static int demanderVP(String s) {
      int v = 0;
      int i = 0;
      BufferedReader br =
           new BufferedReader(new InputStreamReader(System.in));
      System.out.println(s);
      while (v <= 0) {
         System.out.print("La valeur doit etre positive : ");
         try {
            v = (new Integer(br.readLine())).intValue();
         }
         catch (NumberFormatException e) {
         }
         catch (IOException e) {
            System.out.println("Erreur de lecture.");
            System.exit(0);
         }
         i++;
         if (i > 3) {
            System.out.println("Maintenant, y'en a marre !");
            System.exit(0);
         }
      }
      return v;
   }
}
```

La méthode **demander** a été renommée **demanderVP** pour indiquer qu'elle doit retourner une valeur positive. En situation réelle, nous aurions probablement besoin d'une méthode avec plusieurs signatures, par exemple sans argument pour une valeur quelconque, et avec un ou des arguments si l'on veut que la valeur retournée soit limitée d'une façon ou d'une autre.

D'autres objets jetables

Si vous êtes curieux, vous avez peut-être déjà jeté un coup d'œil à la généalogie des exceptions. Dans ce cas, vous aurez remarqué que les exceptions sont une extension d'une classe plus générale appelée **Throwable** (qui signifie *jetable* en anglais).

La classe **Throwable** possède deux sous-classes : **Error** et **Exception**. Les **Exception** sont les objets susceptibles d'être jetés et interceptés (ou attrapés). Les **Errors** sont les objets jetés lorsque les conditions sont telles qu'il n'y a rien à faire pour remédier au problème. Elles ne devraient donc pas être interceptées.

Les exceptions dans les constructeurs

Les constructeurs peuvent lancer des exceptions au même titre que les méthodes. Cependant, il faut noter une particularité importante. Si une exception est lancée par un constructeur, il est possible que la clause **finally** soit exécutée alors que le processus de création de l'objet n'est pas terminé, laissant celui-ci dans un état d'initialisation incomplète. Nous reviendrons sur ce problème dans le chapitre consacré aux entrées/sorties, avec un exemple concernant l'ouverture d'un fichier.

Exceptions et héritage

Vous savez maintenant depuis longtemps qu'une classe peut redéfinir les méthodes des classes parentes. Lorsque celles-ci déclarent des exceptions,

les possibilités de redéfinition supportent une restriction. Les méthodes redéfinies ne peuvent pas lancer d'autres exceptions que celles déclarées par les méthodes de la classe parente, ou celles dérivées de celles-ci.

Résumé

Dans ce chapitre, nous avons étudié le mécanisme qui permet de traiter les "conditions exceptionnelles". Cette étude était indispensable en raison du fait que Java oblige le programmeur à tenir compte des exceptions déclarées par les méthodes. De fait, plus de la moitié des méthodes des classes standard de Java déclarent des exceptions et seraient donc inutilisables sans la connaissance de ces techniques.

Le prochain chapitre sera consacré aux entrées/sorties, un domaine dans lequel il est largement fait appel aux exceptions.

Chapitre 14

Les entrées/sorties

Aucun programme ne peut se passer de communiquer avec le monde extérieur. Cette communication consiste à recevoir des données à traiter, et à renvoyer des données traitées. La première opération constitue une entrée, la seconde une sortie. Les programmes que nous avons réalisés jusqu'ici ne comportaient le plus souvent aucune entrée, car ils traitaient les données qui avaient été incluses dans le programme lui-même. Par ailleurs, la seule sortie employée était l'écran de la console. Dans ce chapitre, nous allons étudier certaines possibilités d'entrée/sortie des programmes Java. Nous laisserons de côté le cas des programmes utilisant une interface fenêtrée, qui sera traité dans un chapitre spécifique, ainsi que celui des entrées/sorties réseau. Nous nous intéresserons plus particulièrement à la console, c'est-à-dire l'ensemble clavier/écran pour l'entrée et la sortie en mode caractères, ainsi qu'aux fichiers sur disques.

Principe des entrées/sorties

Pour effectuer une entrée ou une sortie de données en Java, le principe est simple et se résume aux opérations suivantes :

- Ouverture d'un moyen de communication.
- Écriture ou lecture des données.
- Fermeture du moyen de communication.

En Java, les moyens de communication sont représentés par des objets particuliers appelés (en anglais) *stream*. Ce mot, qui signifie *courant*, ou *flot*, a une importance particulière. En effet, dans de nombreux langages, on ouvre un *canal de communication*. La différence sémantique entre les deux termes est flagrante : le canal est l'endroit où coule le flot. Java s'intéresse donc davantage à ce qui est transmis qu'au moyen physique utilisé pour la transmission. Dans la suite de cette discussion, afin qu'il n'y ait aucun risque de confusion, nous utiliserons le mot *stream* (prononcez "strime").

Il existe de nombreuses sortes de streams, qui peuvent être classés selon plusieurs critères :

- Les streams d'entrées et les streams de sortie.
- Les streams de caractères (texte) et les streams de données binaires.
- Les streams de traitement des données et les streams de communication de données.
- Les streams à accès séquentiel et les streams à accès direct (parfois appelé improprement "accès aléatoire").
- Les streams avec et sans tampon de données.

Les streams relient le programme et un élément particulier appelé *sink*, ce qui signifie en anglais *évier* et doit être pris au sens de "récipient". Cepen-

dant, plusieurs streams peuvent être chaînés, par exemple pour cumuler divers traitements sur les données transférées.

Les streams de données binaires

Les streams de données binaires dérivent de deux classes du package **java.io** : **InputStream**, pour les entrées de données, et **OutputStream**, pour les sorties.

Les streams d'entrée

Les streams d'entrée sont des sous-classes de la classe **java.io. InputStream** :

Streams de communication

- **FileInputStream** : permet la lecture séquentielle de données dans un fichier.
- **PipedInputStream** : permet d'établir une connexion entre un stream d'entrée et un stream de sortie (de type **PipedOutputStream**).
- **ByteArrayInputStream** : permet la lecture de données binaires au moyen d'un tampon indexé.

Streams de traitement

- **FilterInputStream** : cette classe sert de classe parente à diverses classes de streams effectuant des traitements sur les données lues. Les principales classes dérivées sont les suivantes :
  - **BufferedInputStream** : lecture des données à l'aide d'un tampon.
  - **CheckedInputStream** : lecture des données avec vérification (contrôle de *checksum*). Cette classe se trouve dans le package **java.util.zip**.

- **DataInputStream** : lecture de données au format Java (**byte**, **short**, **int**, **long**, **char**, **float**, **double**, **boolean**, etc.).

- **DigestInputStream** : lecture de données avec vérification de l'intégrité. Cette classe se trouve dans le package **java.security**.

- **InflaterInputStream** : lecture de données compressées. Trois sous-classes sont disponibles pour trois algorithmes de décompression : **GZIPInputStream**, **ZipInputStream** et **JarInputStream**.

- **ProgressMonitorInputStream** : lecture de données avec affichage d'une barre de progression. L'utilisateur peut interrompre la lecture en cliquant sur un bouton. Ce stream concerne uniquement les applications fenêtrées et se trouve dans le package **javax.swing**.

- **PushBackInputStream** : lecture de données avec la possibilité de renvoyer la dernière donnée lue. De cette façon, la dernière donnée lue sera également la première lue lors de la prochaine lecture.

- **SequenceInputStream** : enchaînement d'**InputStream**. Un tel stream permet, par exemple, d'effectuer une lecture tamponnée avec décompression et vérification des données.

- **ObjectInputStream** : ce stream permet de lire des données représentant directement des objets Java (qui ont préalablement été écrits à l'aide d'un **ObjectOutputStream**). Cette opération est appelée *désérialisation*.

Les streams de sortie

Il existe un stream de sortie correspondant à chaque stream d'entrée, à condition qu'un tel objet ait un sens :

Streams de communication

- **FileOutputStream** : écriture séquentielle de données dans un fichier.

- **PipedOutputStream** : permet d'établir une connexion entre un stream d'entrée (de type **PipedInputStream**) et un stream de sortie.

- **ByteArrayOutputStream** : écriture de données binaires dans un tampon indexé.

Streams de traitement

- **FilterOutputStream** : cette classe sert de classe parente à diverses classes de streams effectuant des traitements sur les données écrites. Les principales classes dérivées sont les suivantes :

    - **BufferedOutputStream** : écriture de données à l'aide d'un tampon.

    - **CheckedOutputStream** : écriture des données avec vérification (contrôle de *checksum*). Cette classe se trouve dans le package **java.util.zip**.

    - **DataOutputStream** : écriture de données au format Java (**byte**, **short**, **int**, **long**, **char**, **float**, **double**, **boolean**, etc.). Ces données sont ainsi portables d'une application à une autre, indépendamment du système hôte.

    - **DigestOutputStream** : écriture de données avec création d'un hashcode permettant d'en vérifier l'intégrité lors de la relecture au moyen d'un **DigestInputStream**. Cette classe se trouve dans le package **java.security**.

    - **DeflaterInputStream** : écriture de données avec compression. Trois sous-classes sont disponibles pour différents algorithmes de compression : **GZIPOutputStream**, **ZipOutputStream** et **JarOutputStream**.

    - **PrintStream** : écriture de données avec conversion en octets en fonction du système hôte. Ce type de stream est dédié à l'affichage. Il offre deux particularités : d’une part, il ne lance pas d'exception de type **IOException** en cas d'erreur, mais initialise

un indicateur qui peut être consulté à l'aide de la méthode **checkError()** ; d'autre part, il est tamponné et le tampon est vidé automatiquement lorsqu'un tableau de caractères a été entièrement écrit, ou lors de l'écriture du caractère **\n**.

Note : Vous pouvez penser que cette classe n'a rien à faire ici et devrait figurer avec les streams de caractères. C'est également ce que se sont dit les concepteurs de Java lors du passage à la version 1.1. Cette classe a alors été remplacée par **PrintWriter**. La classe **PrintStream** s'est alors trouvée "deprecated", c'est-à-dire conservée uniquement pour assurer la compatibilité. Avec la version 2, elle est réhabilitée, avec l'explication suivante : "L'expérience montre que le remplacement de **PrintStream** par **PrintWriter** n'est pas toujours pratique", ce qui, en langue de bois, signifie "nous avons fait une bêtise et les utilisateurs ont hurlé leur mécontentement".

- **ObjectOutputStream** : ce stream permet d'écrire des données représentant directement des objets Java (qui pourront être relus à l'aide d'un **ObjectInputStream**). Cette opération est appelée *sérialisation.*

Les streams de caractères

Les streams de caractères sont conçus pour la lecture et l'écriture de texte. Les caractères pourraient parfaitement être considérés comme des données binaires. Cependant, un certain nombre de particularités justifient l'utilisation de classes de streams spécialisées. En effet, les caractères Java sont des données de 16 bits, alors que les streams de données binaires traitent des octets.

Les streams de caractères sont dérivés de deux classes abstraites : **Reader** et **Writer**.

Les streams d'entrée

Les streams d'entrée sont des sous-classes de la classe **java.io.Reader**. Deux des méthodes de cette classe sont abstraites :

- **read(char[] cbuf, int off, int len)**, qui permet de lire **len** caractères et de les placer dans le tableau **cbuf**, à partir de l'indice **off**.

- **close()**, qui ferme le stream.

Toutes les classes dérivées de **Reader** redéfinissent donc obligatoirement ces deux méthodes.

Streams de communication

- **PipedReader** : permet d'établir une connexion entre un stream d'entrée et un stream de sortie (de type **PipedWriter**).

- **CharArrayReader** : permet la lecture de caractères au moyen d'un tampon indexé.

- **StringReader** : permet la lecture de caractères à partir d'une chaîne. La chaîne est ainsi traitée comme la source d'un stream.

- **FileReader** : sous-classe particulière de **InputStreamReader** utilisant l'encodage et la taille de tampon par défaut. Cette classe convient dans la plupart des cas de lecture d'un fichier de caractères. Pour utiliser un autre encodage ou une taille de tampon personnalisée, il est nécessaire de sous-classer **InputStreamReader**.

Streams de traitement

- **InputStreamReader** : permet la conversion d'un stream de données binaires en stream de caractères.

- **FilterReader** : cette classe sert de classe parente aux classes de streams effectuant des traitements sur les caractères lus. **java.io** contient une sous-classe de **FilterReader**, **PushBackReader**, qui permet de lire des caractères avec la possibilité de renvoyer le dernier caractère lu. De cette façon, il est possible de lire des caractères comportant un indicateur de début de champ. Lorsque cet indicateur (un caractère particulier) est rencontré, il peut être renvoyé de façon à être le premier caractère lu lors de la prochaine lecture.

- **BufferedReader** : lecture de caractères à l'aide d'un tampon. Les caractères peuvent ainsi être lus en bloc. **java.io** contient une sous-classe de **BufferedReader**, **LineNumberReader**, qui permet de lire des caractères en comptant les lignes. (Les lignes sont délimitées par les caractères CR (\r), LF (\n) ou CRLF (\r\n).)

Les streams de sortie

Les streams de sortie correspondent aux streams d'entrée, à l'exception de ceux qui n'auraient pas de sens. (Aucun stream de sortie ne correspond à **PushBackReader** ni à **LineNumberReader**.)

Les streams de sortie sont des sous-classes de la classe **java.io.Writer**, qui contient trois méthodes abstraites :

- **write(char[] cbuf, int off, int len)**, qui permet d'écrire **len** caractères à partir du tableau **cbuf**, en commençant à l'indice **off**.

- **flush()**, qui vide le tampon éventuel du stream, provoquant l'écriture effective des caractères qui s'y trouvent. Si le stream est chaîné avec un autre stream, la méthode **flush()** de celui-ci est automatiquement appelée.

- **close()**, qui ferme le stream.

Toutes les classes dérivées de **Writer** redéfinissent donc obligatoirement ces trois méthodes.

Streams de communication

- **PipedWriter** : permet d'établir une connexion entre un stream d'entrée (de type **PipedReader**) et un stream de sortie.

- **CharArrayWriter** : permet la lecture de caractères au moyen d'un tampon indexé.

- **StringWriter** : permet l'écriture de caractères dans un **StringBuffer**, qui peut ensuite être utilisé pour créer une chaîne de caractères.

- **FileWriter** : sous-classe particulière de **OutputStreamWriter** utilisant l'encodage et la taille de tampon par défaut. Cette classe convient dans la plupart des cas d'écriture d'un fichier de caractères. Pour utiliser un autre encodage ou une taille de tampon personnalisée, il est nécessaire de sous-classer **OutputStreamWriter**.

Streams de traitement

- **OutputStreamWriter** : permet la conversion d'un stream de données binaires en stream de caractères.

- **FilterWriter** : cette classe sert de classe parente aux classes de streams effectuant des traitements sur les caractères écrits. **java.io** ne contient aucune sous-classe de **FilterWriter**.

- **BufferedWriter** : écriture de caractères à l'aide d'un tampon. Les caractères peuvent ainsi être écrits en bloc. L'utilisation la plus courante consiste à écrire des lignes de texte en les terminant par un appel à la méthode **newLine()**, de façon à obtenir automatiquement le caractère de fin de ligne correspondant au système – CR (\r), LF (\n) ou CRLF (\r\n).

- **PrintWriter** : cette classe permet d'écrire des caractères formatés. Elle est particulièrement utilisée pour l'affichage en mode texte.

Les streams de communication

Il est parfois plus utile de classer les streams selon des critères différents. Nous les avons précédemment classés selon la nature de ce qu'ils manipulent, c'est-à-dire en streams de caractères et streams de données binaires. Il est parfois plus pertinent de les classer selon ce qu'ils font des données qu'ils manipulent, c'est-à-dire en streams de communication et streams de traitement.

Les streams de communication peuvent à leur tour être classés en fonction de la destination des données. Un stream de communication établit une liaison entre le programme et une destination, qui peut être :

- La mémoire :

  Données binaires :
  ByteArrayInputStream
  ByteArrayOutputStream
  StringBufferInputStream

  Caractères :
  CharArrayReader
  CharArrayWriter
  StringReader
  StringWriter

- Un fichier :

  Données binaires :
  FileInputStream
  FileOutputStream

  Caractères :
  FileReader
  FileWriter

- Un chaînage de streams (*pipe*) :

  Données binaires :
  PipedInputStream
  PipedOutputStream

  Caractères :
  PipedReader
  PipedWriter

Lecture et écriture d'un fichier

A titre d'exemple, nous allons écrire un programme capable de lire le contenu d'un fichier et de l'écrire dans un autre fichier. Nous commencerons par traiter un fichier de texte. Pour faire fonctionner ce programme, vous

devrez disposer d'un fichier texte contenant n'importe quoi et nommé **original.txt**. La copie sera nommée **copie.txt**. Nous aurons besoin d'un stream de type **FileReader** et d'un stream de type **FileWriter**. Si nous consultons la documentation de Java concernant ces deux classes, nous constatons qu'il existe trois versions de leurs constructeurs, prenant respectivement pour argument :

- Un objet de type **File**,
- Un objet de type **FileDescriptor**,
- Une chaîne de caractères.

La version utilisant une chaîne de caractères est la plus simple à utiliser :

```
import java.io.*;
public class CopieTXT {
   public static void main (String[] args) throws IOException {
      int c;
      FileReader entrée = new FileReader("original.txt");
      FileWriter sortie = new FileWriter("copie.txt");
      while ((c = entrée.read()) != -1)
         sortie.write(c);
      entrée.close();
      sortie.close();
   }
}
```

Ce programme fonctionne parfaitement, mais il est un peu simpliste. En effet, il ne peut copier un fichier que s'il est nommé **original.txt**. Si ce fichier n'existe pas, le programme produit une erreur. En revanche, il écrase sans avertissement le fichier de sortie s'il existait déjà. Nous le perfectionnerons plus tard. En attendant, il est utile d'analyser son fonctionnement.

La méthode **main** du programme déclare qu'elle lance des exceptions de type **IOException**. De cette façon, elle n'a pas à les traiter et laisse ce soin à l'interpréteur.

Deux streams de type **FileReader** et **FileWriter** sont créés en appelant les constructeurs de ces classes avec deux chaînes de caractères représentant les noms des fichiers.

La partie la plus intéressante se trouve dans la boucle **while**. La condition de cette boucle appelle la méthode **read()** du stream **entrée**. Dans sa version sans argument, cette méthode lit un caractère du stream et le retourne sous la forme d'un **int**. (On retrouve là la vielle habitude de Java de surcaster toutes les valeurs entières en **int**.) Si la fin du stream est atteinte, cette fonction retourne -1. (Ce qui explique qu'il n'est pas possible d'utiliser le type **char**.)

Une fois la boucle terminée, les deux streams sont fermés.

Pour que ce programme soit plus efficace, il faudrait demander à l'utilisateur le nom du fichier à copier, tester son existence et afficher un message d'erreur s'il n'existe pas, puis tester l'existence du fichier de sortie et, s'il existe, demander à l'utilisateur s'il doit être écrasé. Pour cela, nous avons besoin d'un stream de traitement.

Les streams de traitement

Les streams de traitement peuvent être classés en fonction du type de traitement que subissent les données :

- Tamponnage :

  Données binaires :
  BufferedInputStream
  BufferedOutputStream

  Caractères :
  BufferedReader
  BufferedWriter

  Le tamponnage consiste à regrouper les données dans un tampon de façon à accélérer certains traitements. Par exemple, il est plus rapide

de lire un fichier ligne par ligne et de placer chaque ligne dans un tampon (une zone de mémoire réservée à cet effet) d'où chaque caractère pourra être extrait individuellement, plutôt que de lire le fichier caractère par caractère. En effet, la lecture dans un fichier est beaucoup plus lente que la lecture du tampon. Ainsi, si une opération de lecture dans un fichier prend 10 millisecondes et une opération de lecture dans le tampon 0,1 milliseconde, la lecture de 10 000 caractères prendra 100 secondes sans tampon (10 000 x 10 millisecondes) et 2 secondes avec un tampon ((100 x 10) + (10 000 x 0,1) millisecondes).

- Filtrage :

  Données binaires :
  FilterInputStream
  FilterOutputStream

  Caractères :
  FilterReader
  FilterWriter

- Concaténation :

  Données binaires :
  SequenceInputStream

- Conversion de données :

  Données binaires :
  DataInputStream
  DataOutputStream

  Caractères :
  InputStreamReader
  OutputStreamWriter

- Comptage :

  Caractères :
  LineNumberReader

- Sérialisation :

  Données binaires :
  ObjectInputStream
  ObjectOutputStream

- Lecture anticipée :

  Données binaires :
  PushBackInputStream

  Caractères :
  PushBackReader

- Affichage et impression :

  Données binaires :
  PrintStream

  Caractères :
  PrintReader

Nous utiliserons pour l'instant un stream de type **BufferedReader** pour lire les données entrées par l'utilisateur. (Nous avons déjà utilisé cette technique au chapitre précédent sans la décrire.)

Un **BufferedReader** peut être créé en utilisant, comme paramètre de son constructeur, un objet de type **InputStreamReader**. La création de celui-ci nécessite pour sa part le passage d'un paramètre de type **InputStream**. Nous utiliserons ici le champ statique **in** de la classe **System**, qui correspond au clavier de la console. L'instruction complète créant le chaînage de ces streams se présente sous la forme :

```
BufferedReader br =
            new BufferedReader(new InputStreamReader(System.in));
```

La classe **BufferedReader** contient une méthode **readLine()** qui lit une ligne de texte, c'est-à-dire, dans le cas présent, tous les caractères fournis

par l'**InputStreamReader** jusqu'au caractère "fin de ligne". Le résultat de cette méthode est affecté à une chaîne de caractères :

```
String s = br.readLine();
```

Il ne reste plus alors qu'à utiliser cette chaîne pour créer un **FileReader**. Bien sûr, le traitement des erreurs représente la plus grosse partie du travail. Le programme suivant montre un exemple de ce type de traitement :

```
import java.io.*;
public class CopieTXT3 {
   public static void main (String[] args) throws IOException {
      FileReader original = demanderOriginal
              ("Entrez le nom du fichier texte a copier : ");
      FileWriter copie = demanderCopie
              ("Entrez le nom a donner a la copie du fichier : ");
      int c;
      while ((c = original.read()) != -1)
         copie.write(c);

      original.close();
      copie.close();
   }
}

class Util {
   public static FileReader demanderOriginal(String s) {
      FileReader fr = null;
      String s1 = null;
      int i = 0;
      BufferedReader br =
           new BufferedReader(new InputStreamReader(System.in));
      System.out.print(s);
      while (fr == null) {
         try {
            s1 = br.readLine();
         }
```

```
      catch (IOException e) {
         System.out.println("Erreur de lecture de la console.");
         System.exit(0);
      }
      try {
         fr = new FileReader(new File(s1));
      }
      catch (NullPointerException e) {
      }
      catch (FileNotFoundException e) {
         if (i < 2)
            System.out.print
            ("Ce fichier n'existe pas. Entrez un autre nom : ");
         else if (i == 2)
            System.out.print
            ("Vous avez encore droit a un essai : ");
      }
      i++;
      if (i > 4) {
         System.out.println("Nombre d'essais depasse.");
         System.exit(0);
      }
}
   return fr;
}

public static FileWriter demanderCopie(String s) {
   FileWriter fr = null;
   File f = null;
   String s1 = null;
   int i = 0;
   BufferedReader br =
        new BufferedReader(new InputStreamReader(System.in));
   System.out.print(s);
   while (fr == null) {
      try {
         s1 = br.readLine();
      }
```

```
catch (IOException e) {
   System.out.println("Erreur de lecture de la console.");
   System.exit(0);
}
try {
   f = new File(s1);
   if (f.exists()) {
      System.out.print
   ("Le fichier existe. Voulez-vous l'ecraser (o/n) : ");
      try {
         s1 = br.readLine();
      }
      catch (IOException e) {
         System.out.println
            ("Erreur de lecture de la console.");
         System.exit(0);
      }
      if (!s1.equals("O") && !s1.equals("o")) {
         fr = null;
         i = 0;
         System.out.print
                ("Entrez un autre nom de fichier : ");
      }
      else
         fr = new FileWriter(f);
   }
   else
      fr = new FileWriter(f);
}
catch (NullPointerException e) {
}
catch (IOException e) {
   if (i < 2)
      System.out.print
      ("Ce fichier n'existe pas. Entrez un autre nom : ");
   else if (i == 2)
      System.out.print
         ("Vous avez encore droit a un essai : ");
```

```
            }
            i++;
            if (i > 4) {
               System.out.println("Nombre d'essais depasse.");
               System.exit(0);
            }
      }
         return fr;
      }
   }
```

Le programme principal est extrêmement simple. Il fait appel aux méthodes **demanderOriginal()** et **demanderCopie()** de la classe **Util**. Ces méthodes sont beaucoup plus complexes car elles doivent prendre en compte de nombreux paramètres. Les exemples présentés ici sont de type *Q&D* (*Quick and Dirty*), ce qui signifie qu'elles sont écrites rapidement sans souci d'optimisation. Ce qui importe est que leurs interfaces soient clairement définies. Il sera toujours possible de réécrire ces méthodes par la suite. Dans les exemples de la suite de ce chapitre, nous ferons appel à ces méthodes sans les reproduire.

Exemple de traitement : utilisation d'un tampon

Jusqu'ici, la façon de traiter les entrées/sorties en Java ne semblait pas particulièrement excitante. Cependant, nous allons voir maintenant qu'elle offre de nombreux avantages. Il est en effet possible de combiner à loisir les différents streams, par exemple pour effectuer des traitements pendant la transmission des données. Un traitement simple consiste à utiliser un tampon pour accélérer la copie. Il suffit pour cela d'"envelopper" les streams **FileReader** et **FileWriter** dans des streams **BufferedReader** et **BufferedWriter**, ce qui ne présente aucune difficulté :

```
import java.io.*;

public class Tampon {
```

```
   public static void main (String[] args) throws IOException {
      FileReader o = Util.demanderOriginal
              ("Entrez le nom du fichier texte a copier : ");
      FileWriter d = Util.demanderCopie
              ("Entrez le nom a donner a la copie du fichier : ");
      BufferedReader original = new BufferedReader(o);
      BufferedWriter copie = new BufferedWriter(d);
      int c;

      while ((c = original.read()) != -1)
         copie.write(c);
      original.close();
      copie.close();
   }
}
```

Les modifications nécessaires apparaissent en gras. La copie est maintenant beaucoup plus rapide.

Exemple de traitement : conversion des fins de lignes

Le programme précédent copiait simplement le contenu d'un fichier. Le prochain exemple ajoutera un traitement entre la lecture et l'écriture. Ce traitement consistera à convertir les fins de lignes MAC (CR), UNIX (LF) ou MS-DOS (CRLF) en fins de lignes adaptées au système. Ce programme permettra donc de convertir un fichier texte venant de n'importe quel système vers le système utilisé.

Attention : Il ne s'agit ici que de la conversion des fins de lignes et non des caractères accentués.

```
import java.io.*;

public class CRLF {
   public static void main (String[] args) throws IOException {
```

```
        FileReader o = Util.demanderOriginal
                ("Entrez le nom du fichier texte a copier : ");
        FileWriter d = Util.demanderCopie
                ("Entrez le nom a donner a la copie du fichier : ");
        BufferedReader original = new BufferedReader(o);
        BufferedWriter copie = new BufferedWriter(d);
        String c;

        while ((c = original.readLine()) != null) {
           copie.write(c);
           copie.newLine();
        }
        original.close();
        copie.close();
    }
}
```

Ce type de traitement ne nécessite pas la mise en œuvre d'un nouveau stream, mais simplement l'utilisation de la méthode **readLine()** au lieu de la méthode **read()**. Cette méthode lit une ligne entière en ignorant le caractère de fin de ligne. Java reconnaît les fins de lignes de type CR, LF ou CRLF. Lors de la copie, la méthode **newLine()** est appelée pour écrire une fin de ligne correspondant au système utilisé. Nous n'avons donc pas à nous préoccuper de sa nature !

Compression

La compression des données est un type de traitement fréquemment employé. Il peut être réalisé facilement en Java en utilisant un stream de type **DeflaterOutputStream**, qui est une sous-classe de **FilterOutputStream**. Java propose trois classes de **DeflaterOutputStream :**

- **ZipOutputStream**, pour la création de fichiers de type Zip, ce qui est le standard de compression sous MS-DOS et Windows.

- **GZIPOutputStream**, pour la création de fichiers de type GZIP, couramment employés sous UNIX.

- **JarOutputStream**, pour la création de fichiers de type JAR, standard Java utilisant le même algorithme que les fichiers Zip mais ajoutant certaines informations. Les fichiers JAR permettent de compresser en un seul fichier toutes les classes nécessaires à une application et, surtout, à une applet, ce qui permet d'accélérer considérablement son chargement à travers un réseau tel qu'Internet.

Nous utiliserons le type Zip. Avant de créer le programme proprement dit, il nous faut ajouter à notre classe **Util** deux méthodes retournant des **FileInputStream** et **FileOutputStream** au lieu des **FileReader** et **FileWriter**. En effet, les fichiers compressés contiennent des données binaires et non du texte. Il nous suffit simplement de recopier les deux méthodes en effectuant quelques petites modifications :

```
import java.io.*;
public class Util {
   public static FileReader demanderOriginal(String s){
         .
         .
         .
      return fr;
   }

   public static FileWriter demanderCopie(String s) {
         .
         .
         .
      return fr;
   }

   public static FileInputStream binOriginal(String s) {
      FileInputStream fr = null;
      String s1 = null;
      int i = 0;
      BufferedReader br =
            new BufferedReader(new InputStreamReader(System.in));
      System.out.print(s);
```

```
   while (fr == null) {
      try {
         s1 = br.readLine();
      }
      catch (IOException e) {
         System.out.println("Erreur de lecture de la console.");
         System.exit(0);
      }
      try {
         fr = new FileInputStream (new File(s1));
      }
      catch (NullPointerException e) {
      }
      catch (FileNotFoundException e) {
         if (i < 2)
            System.out.print
            ("Ce fichier n'existe pas. Entrez un autre nom : ");
         else if (i == 2)
            System.out.print
            ("Vous avez encore droit a un essai : ");
      }
      i++;
      if (i > 4) {
         System.out.println("Nombre d'essais depasse.");
         System.exit(0);
      }
   }
   return fr;
}

public static FileOutputStream binCopie(String s) {
   FileOutputStream fr = null;
   File f = null;
   String s1 = null;
   int i = 0;
   BufferedReader br =
        new BufferedReader(new InputStreamReader(System.in));
   System.out.print(s);
```

```
while (fr == null) {
   try {
      s1 = br.readLine();
   }
   catch (IOException e) {
      System.out.println("Erreur de lecture de la console.");
      System.exit(0);
   }
   try {
      f = new File(s1);
      if (f.exists()) {
         System.out.print
      ("Le fichier existe. Voulez-vous l'ecraser (o/n) : ");
         try {
            s1 = br.readLine();
         }
         catch (IOException e) {
            System.out.println
               ("Erreur de lecture de la console.");
            System.exit(0);
         }
         if (!s1.equals("O") && !s1.equals("o")) {
            fr = null;
            i = 0;
            System.out.print
                    ("Entrez un autre nom de fichier : ");
         }
         else
            fr = new FileOutputStream (f);
      }
      else
         fr = new FileOutputStream (f);
   }
   catch (NullPointerException e) {
   }
   catch (IOException e) {
      if (i < 2)
         System.out.print
```

```
            ("Ce fichier n'existe pas. Entrez un autre nom : ");
         else if (i == 2)
            System.out.print
               ("Vous avez encore droit a un essai : ");
      }
      i++;
      if (i > 4) {
         System.out.println("Nombre d'essais depasse.");
         System.exit(0);
      }
   }
      return fr;
   }
}
```

Le programme lui-même est très simple. Nous réaliserons tout d'abord une version copiant des fichiers binaires sans compression :

```
import java.io.*;

public class Zip {
   public static void main (String[] args) throws IOException {
      FileInputStream o = Util.binOriginal
              ("Entrez le nom du fichier texte a copier : ");
      FileOutputStream d = Util.binCopie
              ("Entrez le nom a donner a la copie du fichier : ");
      BufferedInputStream original = new BufferedInputStream(o);
      BufferedOutputStream copie = new BufferedOutputStream(d);
      int c;

      while ((c = original.read()) != -1)
         copie.write(c);

      original.close();
      copie.close();
   }
}
```

Pour ajouter la compression, les modifications à apporter sont minimes :

```
import java.io.*;
import java.util.zip.*;

public class Zip {
   public static void main (String[] args) throws IOException {
      FileInputStream o = Util.binOriginal
             ("Entrez le nom du fichier texte a copier : ");
      FileOutputStream d = Util.binCopie
             ("Entrez le nom a donner a la copie du fichier : ");
      BufferedInputStream original = new BufferedInputStream(o);
      BufferedOutputStream cB = new BufferedOutputStream(d);
      ZipOutputStream copie = new ZipOutputStream(cB);
      int c;
      copie.setMethod(ZipOutputStream.DEFLATED);
      copie.putNextEntry(new ZipEntry("fichier1.txt"));

      while ((c = original.read()) != -1)
         copie.write(c);

      original.close();
      copie.close();
   }
}
```

Le **BufferedOutputStream** est simplement enveloppé dans un **ZipOutputStream**. Deux appels de méthodes spécifiques sont ensuite ajoutés :

- **setMethod(ZipOutputStream.DEFLATED)** détermine le type d'opération qui va être effectuée. **DEFLATED** signifie que les données seront compressées. **STORED** signifie que les données seront simplement stockées sans compression. Il peut paraître bizarre qu'un stream de compression soit employé pour écrire des données non compressées, mais cela s'explique par le fait que ce stream permet de stocker plusieurs fichiers dans un même fichier zip.

- **putNextEntry(new ZipEntry("fichier1.txt"))** crée une nouvelle entrée dans le fichier zip. Si plusieurs fichiers sont compressés dans le même fichier zip, une **ZipEntry** doit être créée pour chaque fichier. Ici, le nom donné à l'entrée est arbitrairement **fichier.txt**. Normalement, on utilise le nom du fichier qui a été compressé.

Note : Pour compresser plusieurs fichiers, il faut créer la première entrée, copier le fichier, créer la deuxième entrée, copier le deuxième fichier, et ainsi de suite, et non créer toutes les entrées à la suite les unes des autres.

Décompression

Si vous voulez tester le fichier compressé, vous pouvez employer le programme suivant :

```
import java.io.*;
import java.util.zip.*;

public class Unzip {
   public static void main (String[] args) throws IOException {
      FileInputStream o = new FileInputStream(args[0]);
      ZipInputStream z =
            new ZipInputStream(new BufferedInputStream(o));
      ZipEntry ze;
      ze = z.getNextEntry();
      int c;
      while ((c = z.read()) != -1)
         System.out.write(c);
      z.close();
   }
}
```

Ce programme décompresse un fichier dont le nom lui est passé sur la ligne de commande, sous la forme :

```
java Unzip x.zip
```

et affiche le contenu du fichier décompressé à l'écran. (Ici, on suppose qu'il s'agit d'un fichier texte ayant été compressé sous le nom **x.zip**. L'adaptation de ce programme pour qu'il écrive les données décompressées dans un fichier ne devrait maintenant vous poser aucun problème.

La sérialisation

Nous avons vu que certains streams permettent d'enregistrer sur disque des objets Java. Cette opération est appelée *sérialisation*. Elle permet de conserver l'état des objets entre deux exécutions d'un programme, ou d'échanger des objets entre programmes. Le programme suivant montre un exemple de sérialisation :

```
import java.io.*;

public class Serialisation {
   public static void main (String[] args) throws IOException {
      Voiture voiture = new Voiture("V6", "Cabriolet");
      voiture.setCarburant(50);
      FileOutputStream f = new FileOutputStream("garage");
      ObjectOutputStream o = new ObjectOutputStream(f);

      o.writeObject(voiture);
      o.close();
   }
}

class Voiture implements Serializable {
   Moteur moteur;
   Carrosserie carrosserie;
   transient int essence;

   Voiture (String m, String c) {
      moteur = new Moteur(m);
      carrosserie = new Carrosserie(c);
   }
```

```
   String getMoteur() {
      return moteur.getValeur();
   }
   String getCarrosserie() {
      return carrosserie.getValeur();
   }
   void setCarburant(int e) {
      essence += e;
   }
   int getCarburant() {
      return essence;
   }
}

class Carrosserie implements Serializable {
   String valeur;

   Carrosserie (String s) {
      valeur = s;
   }

   String getValeur() {
      return valeur;
   }
}

class Moteur implements Serializable {
   String valeur;

   Moteur (String s) {
      valeur = s;
   }

   String getValeur() {
      return valeur;
   }
}
```

Ce programme contient la classe **Voiture**, qui possède deux membres qui sont des instances des classes **Carrosserie** et **Moteur**, et un membre de type **essence** de type **int** déclaré **transient**. Ces trois classes implémentent l'interface **Serializable**. Cette interface ne comporte aucune méthode. Elle constitue simplement un marqueur qui indique que les instances pourront être *sérialisées*, c'est-à-dire, par exemple, enregistrées sur disque. Lors de la sérialisation, tous les membres sont sérialisés, à condition que les classes dont ils sont des instances soient elles-mêmes sérialisables. La sérialisation est effectuée par Java de manière récursive avec autant de niveaux que nécessaire. En revanche, les champs qui sont déclarés de type **transient** ne sont pas sérialisés. Par exemple, si les éléments d'une transaction effectuée par un utilisateur à l'aide d'un mot de passe sont sérialisés, il est probable que le mot de passe sera déclaré de type **transient** afin qu'il ne soit pas enregistré avec le reste des données.

Le programme crée une instance de **Voiture**, modifie la valeur du champ **transient** grâce à un appel à la méthode **setCarburant**, puis sérialise l'objet à l'aide d'un stream **ObjectOutputStream** enveloppé dans un **FileOutputStream**.

L'objet sérialisé est enregistré sur disque dans un fichier nommé **garage**.

Le programme suivant permet de relire l'objet sérialisé et de constater que les objets membres ont été automatiquement sérialisés. En revanche, le champ **transient** a perdu sa valeur.

```
import java.io.*;

public class Deserialisation {
   public static void main (String[] args) throws IOException,
                                        ClassNotFoundException {
      FileInputStream f = new FileInputStream("garage");
      ObjectInputStream o = new ObjectInputStream(f);

      Voiture voiture = (Voiture)o.readObject();
      o.close();
```

```
     System.out.println("Carrosserie : " +
voiture.getCarrosserie());
     System.out.println("Moteur : " + voiture.getMoteur());
     System.out.println("Carburant : " + voiture.getCarburant());
   }
}
```

Ce programme affiche le résultat suivant :

```
Carrosserie : Cabriolet
Moteur : V6
Carburant : 0
```

Les fichiers à accès direct

Tous les exemples que nous avons vus jusqu'ici impliquaient un accès séquentiel aux données. Il s'agit là du principe même des streams, dont les données doivent être traitées de la première à la dernière. La recherche d'un élément d'information particulier dans une telle structure peut être très longue. En moyenne, il faudra lire la moitié des données pour accéder à une donnée quelconque, comme si, pour rechercher une information dans un livre, on était contraint de lire celui-ci depuis le début.

Heureusement, les livres disposent de tables des matières et de la possibilité d'accéder directement à une page donnée. La constitution d'une table des matières revient au créateur des données. En revanche, le langage doit fournir un moyen d'accéder directement à des données dont l'adresse a été trouvée dans la table.

Dans de nombreuses structures de données conçues pour un accès direct, la "table des matières" est placée à la fin des données. Du point de vue de la programmation, cela ne fait aucune différence, sinon que l'adresse de cette table doit être connue ou inscrite au début des données.

Les fichiers à accès direct étant fondamentalement différents des streams, ils ne dérivent pas des mêmes classes. Java dispose de la classe **RandomAccessFile**, qui dérive directement de la classe **Object**.

La création d'un fichier à accès direct s'effectue très simplement à l'aide de deux paramètres. Le premier peut être une chaîne de caractères indiquant le nom du fichier, ou un objet de type **File**, qui offre l'avantage de pouvoir être manipulé de diverses façons pour obtenir des informations concernant le système hôte (chemin d'accès réel du fichier, existence, attributs, etc.).

Le deuxième paramètre est une chaîne de caractères indiquant si le fichier est ouvert en lecture seule (**r**) ou en lecture et écriture (**rw**). L'utilisation de deux modes différents permet de ne pas verrouiller les fichiers dont l'accès est fait en lecture seule.

Un nouveau fichier pourra donc être créé de la façon suivante :

```
RandomAccesFile raf;
raf = new RandomAccessFile("fichier.txt", "rw");
```

Le principe d'un fichier à accès direct repose sur l'utilisation d'un pointeur indiquant la position dans le fichier. Chaque opération de lecture ou d'écriture augmente la valeur du pointeur. Il est donc tout à fait possible d'utiliser ce type de fichier comme un stream. L'exemple suivant effectue une copie de fichiers à l'aide d'un **RandomAccessFile** :

```
import java.io.*;

public class AccesDirect {
   public static void main (String[] args) throws IOException {
      RandomAccessFile original, copie;
      original = new RandomAccessFile
         (ROriginal("Entrez le nom du fichier source : "), "r");
      copie = new RandomAccessFile
         (Rcopie("Entrez le nom du fichier destination : "), "rw");
      int c;
```

```
   while ((c = original.read()) != -1)
      copie.write(c);

   original.close();
   copie.close();
}

public static File ROriginal(String s) {
   File f = null;
   String s1 = null;
   int i = 0;
   BufferedReader br =
        new BufferedReader(new InputStreamReader(System.in));
   System.out.print(s);
   while (f == null) {
      try {
         s1 = br.readLine();
      }
      catch (IOException e) {
         System.out.println("Erreur de lecture de la console.");
         System.exit(0);
      }
      try {
         f = new File(s1);
      }
      catch (NullPointerException e) {
      }
      if (!f.exists()) {
         if (i < 2)
            System.out.print
            ("Ce fichier n'existe pas. Entrez un autre nom : ");
         else if (i == 2)
            System.out.print
                      ("Vous avez encore droit a un essai : ");
         f = null;
      }
      i++;
      if (i > 3) {
```

```
            System.out.println("Nombre d'essais depasse.");
            System.exit(0);
         }
      }
      return f;
   }

   public static File Rcopie(String s) {
      File f = null;
      String s1 = null;
      int i = 0;
      BufferedReader br =
            new BufferedReader(new InputStreamReader(System.in));
      System.out.print(s);
      while (f == null) {
         try {
            s1 = br.readLine();
         }
         catch (IOException e) {
            System.out.println("Erreur de lecture de la console.");
            System.exit(0);
         }
         try {
            if (s1.length() == 0)
               throw new IOException();
            f = new File(s1);
            if (f.exists()) {
               System.out.print
            ("Le fichier existe. Voulez-vous l'ecraser (o/n) : ");
               try {
                  s1 = br.readLine();
               }
               catch (IOException e) {
                  System.out.println
                          ("Erreur de lecture de la console.");
                  System.exit(0);
               }
               if (!s1.equals("O") && !s1.equals("o")) {
```

```
                f = null;
                i = 0;
                System.out.print
                        ("Entrez un autre nom de fichier : ");
             }
          }
       }
       catch (NullPointerException e) {
       }
       catch (IOException e) {
          if (i < 2)
             System.out.print
  ("Nom de fichier incorrect. Entrez un autre nom de fichier : ");
          else if (i == 2)
             System.out.print
                        ("Vous avez encore droit a un essai : ");
       }
       i++;
       if (i > 3) {
          System.out.println("Nombre d'essais depasse.");
          System.exit(0);
       }
    }
    return f;
  }
}
```

Note : La gestion des erreurs dans l'entrée des noms de fichiers est très sommaire. En particulier, seule l'entrée d'un nom de longueur nulle est considérée comme incorrecte. Dans la réalité, il faudrait évidemment tester la validité du nom de fichier de façon plus approfondie.

Il faut noter que les méthodes **read()** et **write()**, bien qu'utilisées avec des données de type **int**, lisent et écrivent des **byte**. La classe **RandomAccessFile** dispose de méthodes permettant de lire et écrire tous les types de données : **readChar()**, **readShort()**, **readInt()**, **readFloat()**, **readBoolean()**, etc. Mais les méthodes les plus intéressantes sont celles qui permettent de lire et de modifier le pointeur indiquant la position courante dans le fichier :

- **public long getFilePointer()** Cette méthode permet de connaître la valeur du pointeur, c'est-à-dire la position à laquelle aura lieu la prochaine lecture ou écriture.

- **public void seek(long pos)** Cette méthode permet de modifier la valeur du pointeur afin de lire ou d'écrire à une position donnée.

- **public long length()** Renvoie la longueur du fichier, c'est-à-dire la valeur maximale que peut prendre le pointeur plus 1. (Pour un fichier de longueur *l*, le pointeur peut prendre une valeur comprise entre 0 et *l* - 1.)

- **public void setLength(long newLength)** Modifie la longueur du fichier. Si la nouvelle longueur est plus petite que la longueur courante, le fichier est tronqué. Dans ce cas, si le pointeur avait une valeur supérieure à la nouvelle longueur, il prend la valeur de la nouvelle longueur. (Cette valeur est supérieure de 1 à la position du dernier octet du fichier. Une opération de lecture renvoie donc alors la valeur -1 pour indiquer la fin du fichier.)

Résumé

Dans ce chapitre, nous avons étudié un certain nombre des fonctions d'entrées/sorties de Java. Il existe d'autres possibilités d'entrées/sorties. Par exemple, certains programmes utilisent une interface basée sur l'affichage de fenêtres. Des sorties peuvent alors être effectuées sur l'écran grâce à des objets spéciaux que nous étudierons au chapitre consacré aux interfaces utilisateur. De même, les entrées/sorties d'un programme peuvent avoir lieu par l'intermédiaire d'un réseau. Nous traiterons ce cas particulier dans un prochain chapitre.

Chapitre 15

Le passage des paramètres

Dans ce chapitre, nous allons revenir rapidement sur un sujet que nous avons déjà abordé implicitement : la façon dont les paramètres sont passés aux méthodes et l'influence que cela peut avoir sur la vie des objets. Traditionnellement, on considère qu'il existe deux façons de passer les paramètres : par valeur et par référence.

Passage des paramètres par valeur

La première façon de procéder consiste à passer à la méthode la valeur du paramètre dont elle a besoin. Considérons l'exemple d'une méthode qui prend

pour paramètre une valeur entière et affiche son carré. Nous pourrions écrire ce programme de la façon suivante :

```
public class Param1 {
   public static void main (String[] args) {
      int nombre = 12;
      System.out.println(nombre);
      afficheCarré(nombre);
      System.out.println(nombre);
   }

   public static void afficheCarré(int n) {
      n = n * n;
      System.out.println(n);
   }
}
```

Ce programme affiche :

```
12
144
12
```

Lors de l'appel de la méthode **afficheCarré**, la valeur de **nombre** est passée à la méthode qui utilise cette valeur pour initialiser la variable **n**. Cette variable n'a rien à voir avec la variable nombre, si ce n'est qu'elle a temporairement la même valeur. A la deuxième ligne de la méthode, la valeur de **n** est modifiée. Cela ne modifie en rien la valeur de **nombre**. Lorsque la méthode retourne, nous constatons (en l'affichant) que cette variable n'a pas été modifiée. Il s'agit ici d'un paramètre passé "par valeur", car seule la valeur de la variable est passée, et non la variable elle-même.

Passage des paramètres par référence

Une autre façon de passer les paramètres consiste à transmettre un pointeur vers une variable. Considérons l'exemple suivant :

```
public class Param2 {
   public static void main (String[] args) {
      Entier nombre = new Entier(12);
      System.out.println(nombre);
      afficheCarré(nombre);
      System.out.println(nombre);
   }

   public static void afficheCarré(Entier n) {
      n.setValeur(n.getValeur() * n.getValeur());
      System.out.println(n);
   }
}

class Entier {
   private int valeur;

   Entier(int v) {
      valeur = v;
   }

   public int getValeur() {
      return valeur;
   }

   public void setValeur(int v) {
      valeur = v;
   }

   public String toString() {
      return ("" + valeur);
   }
}
```

Ce programme fait à peu près la même chose que le précédent, à la différence qu'il manipule un objet (instance de la classe **Entier**) qui est un enveloppeur pour le type **int**. La classe **Entier** comporte un champ **valeur**

de type **int**, un constructeur initialisant la valeur de ce champ, et deux méthodes, **getValeur()** et **setValeur()**, qui permettent respectivement de lire et de modifier la valeur du champ **valeur**. De plus, la méthode **toString()** est redéfinie afin d'afficher la valeur de ce champ.

Si nous exécutons le programme, nous obtenons le résultat suivant :

```
12
144
144
```

La différence est ici fondamentale : la méthode **afficheCarré** a modifié l'objet qui lui a été passé et non une copie de celui-ci.

Certains langages permettent de choisir si une méthode (ou une procédure, ou une fonction, selon la terminologie employée par chacun) utilise le passage de paramètres par valeur ou par référence. Avec Java, ce n'est pas le cas. Il peut paraître curieux qu'une seule possibilité soit offerte pour les primitives et une autre, différente, pour les objets. En fait, il n'en est rien. Le même mécanisme est mis en œuvre dans les deux cas.

En effet, nous devons nous souvenir de ce que nous avons dit sur les handles. Une primitive **est** ce qu'elle représente. (Cela est vrai, du moins, du point de vue du programmeur en Java. Pour celui qui développe une machine virtuelle Java, c'est une autre histoire !) En revanche, un handle pointe vers un objet mais il n'est pas l'objet vers lequel il pointe.

Dans le deuxième programme, le handle **nombre** est passé à la méthode **afficheCarré** par valeur. Cependant, cette méthode n'en fait qu'un seul usage : elle crée un nouveau handle pointant vers le même objet. A l'intérieur de la méthode, nous n'avons aucun accès au handle passé comme paramètre.

Note : Bien sûr, si le handle avait été déclaré membre de la classe **Param2**, comme dans l'exemple suivant :

```
public class Param2 {
  static Entier nombre;
  public static void main (String[] args) {
     nombre = new Entier(12);
        .
        .
```

il serait accessible dans la méthode **afficheCarré**, mais cela ne serait en aucun cas dû au passage de paramètre.

Passer les objets par valeur

Il arrive souvent que l'on souhaite passer un objet comme paramètre à une méthode pour que celle-ci y apporte des modifications, sans souhaiter pour autant que ces modifications soit reflétées dans l'objet original. La solution paraît simple. Il suffit d'effectuer une copie de l'objet et d'apporter les modifications à cet objet. Dans le cas du programme précédent, ce n'est pas très compliqué :

```
public class Param3 {
   static Entier nombre;
   public static void main (String[] args) {
      nombre = new Entier(12);
      System.out.println(nombre);
      afficheCarré(nombre);
      System.out.println(nombre);
   }
   public static void afficheCarré(Entier n2) {
      Entier n = new Entier(n2.getValeur());
      n.setValeur(n.getValeur() * n.getValeur());
      System.out.println(n);
   }
}
```

Ce programme affiche maintenant le résultat correct :

```
12
144
12
```

Ici, il a été possible de créer une copie de l'objet passé comme paramètre parce que nous connaissions la nature de cet objet. De plus, cette création était un processus très simple. Cependant, ce n'est pas toujours le cas. Il est parfaitement possible qu'une méthode reçoive en paramètre un objet sur-casté vers une classe parente. Comment, alors, en effectuer une copie ? Java dispose d'un mécanisme spécialement prévu pour cela.

Le clonage des objets

Effectuer une copie d'un objet est très simple. Il suffit de faire appel à sa méthode **clone()**. Pour l'instant, cela ne nous avance pas beaucoup. En effet, notre classe **Entier** ne dispose pas d'une telle méthode. Nous pourrions l'écrire de la façon suivante :

```
public class Param4 {
   static Entier nombre;
   public static void main (String[] args) {
      nombre = new Entier(12);
      System.out.println(nombre);
      afficheCarré(nombre);
      System.out.println(nombre);
   }
   public static void afficheCarré(Entier n2) {
      Entier n = n2.clone();
      n.setValeur(n.getValeur() * n.getValeur());
      System.out.println(n);
   }
}
class Entier {
   private int valeur;
   Entier(int v) {
```

```
      valeur = v;
   }
   public int getValeur() {
      return valeur;
   }
   public void setValeur(int v) {
      valeur = v;
   }
   public String toString() {
      return ("" + valeur);
   }
   public Entier clone() {
      return new Entier(valeur);
   }
}
```

Malheureusement (ou heureusement), ce programme ne se compile pas. En effet, la classe **Object** contient déjà une méthode **clone()** dont la valeur de retour est de type **Object**. Il est donc impossible de redéfinir cette méthode avec un autre type. En revanche, il est possible de modifier le programme de la façon suivante :

```
public static void afficheCarré(Entier n2) {
   Entier n = (Entier)n2.clone();
```

et :

```
public Object clone() {
   return new Entier(valeur);
}
```

Mais il y a mieux ! Nous pouvons utiliser la méthode **clone()** de la classe **Object**. Si nous examinons la documentation de cette classe, nous constatons que sa méthode **clone()** est **protected**. Il nous est donc impossible de l'appeler directement depuis la classe **Entier** sans l'avoir redéfinie dans notre

classe. Cela est conçu de la sorte pour empêcher que tous les objets soient automatiquement clonables.

Cependant, il est parfaitement possible de redéfinir la méthode **clone()** de la façon suivante :

```
public Object clone() {
   return super.clone();
}
```

Si cela est si simple, on ne voit pas bien pourquoi la méthode **clone()** de la classe **Object** est **protected**. En fait, si vous essayez de compiler le programme ainsi, vous obtiendrez un message indiquant que l'exception **CloneNotSupportedException** doit être traitée. Il peut paraître simple de la déclarer lancée par notre méthode **clone()** :

```
public Object clone() throws CloneNotSupportedException {
   return super.clone();
}
```

Nous devrons dans ce cas faire de même pour les méthodes **afficheCarré()** et **main()**. Il est peut-être plus facile d'intercepter l'exception dans la méthode **clone()** :

```
public Object clone() {
   Object o = null;
   try {
      o = super.clone();
   }
   catch (CloneNotSupportedException e) {
      System.out.println(e);
   }
   return o;
}
```

Malheureusement, ainsi modifié, le programme ne fonctionne pas. En effet, une exception **CloneNotSupportedException** est systématiquement lancée lors de l'appel de la méthode **clone()**. En fait, nous avons là la réponse à notre question précédente. Bien qu'accessible grâce à la redéfinition **public** de la méthode, celle-ci ne fonctionne pas. En effet, avant d'accomplir son travail, cette méthode vérifie si elle agit sur un objet instance de l'interface **Cloneable**. De cette façon, seuls les objets explicitement déclarés comme implémentant cette interface pourront être clonés. Il nous suffit donc, pour que notre programme fonctionne, de modifier la déclaration de la méthode :

```
public Object clone() implements Cloneable {
   Object o = null;
   try {
      o = super.clone();
   }
   catch (CloneNotSupportedException e) {
      System.out.println(e);
   }
   return o;
}
```

Que contient l'interface **Cloneable** ? Rien du tout. Elle ne sert en fait que de marqueur, pour indiquer qu'un objet peut être cloné. Nous avons ici un exemple d'un type d'utilisation des interfaces dont nous avions parlé au Chapitre 9.

Clonage de surface et clonage en profondeur

La méthode **clone()** renvoie une copie de l'objet. Cependant, qu'en est-il des objets référencés par l'objet cloné ? Le programme suivant permet de comprendre ce qui se passe lors du clonage d'un objet contenant des liens vers d'autres objets clonables :

```
public class Clone1 {

   public static void main (String[] args) {
      Voiture voiture = new Voiture();
      voiture.setCarburant(30);
      System.out.println(voiture.getCarburant());
      pleinClone(voiture);
      System.out.println(voiture.getCarburant());
   }

   public static void pleinClone(Voiture v) {
      Voiture v2 = (Voiture)v.clone();
      v2.setCarburant(100);
      System.out.println(v2.getCarburant());
   }
}

class Voiture implements Cloneable {
   Reservoir reservoir;

   Voiture () {
      reservoir = new Reservoir();
   }

   void setCarburant(int e) {
      reservoir.setContenu(e);
   }

   int getCarburant() {
      return reservoir.getContenu();
   }

   public Object clone() {
      Object o = null;
      try {
         o = super.clone();
      }
      catch (CloneNotSupportedException e) {
```

```
            System.out.println(e);
         }
         return o;
      }
   }

   class Reservoir implements Cloneable {
      private int contenu;

      public int getContenu() {
         return contenu;
      }

      public void setContenu(int e) {
         contenu = e;
      }

      public Object clone() {
         Object o = null;
         try {
            o = super.clone();
         }
         catch (CloneNotSupportedException e) {
            System.out.println(e);
         }
         return o;
      }
   }
```

Ce programme crée une instance de la classe **Voiture**. Le constructeur de cette classe initialise la variable **reservoir** en créant une instance de la classe **Reservoir**. Cette classe comporte un membre privé de type **int** appelé **contenu** qui peut être lu et mis à jour au moyen des méthodes **setContenu** et **getContenu**. Ces méthodes sont appelées par les méthodes **setCarburant et getCarburant** de la classe **Voiture**.

La méthode **main()** de notre programme crée tout d'abord une instance de **Voiture**, puis appelle la méthode **setCarburant** qui affecte la valeur utilisée comme paramètre au champ **contenu** de l'instance de **Reservoir**. La valeur de ce champ est ensuite affichée.

La méthode **pleinClone** est ensuite appelée pour créer un clone de l'objet **voiture** et appeler la méthode **setCarburant** de ce nouvel objet avec un paramètre différent. La méthode **getCarburant** est alors appelée pour afficher la valeur modifiée.

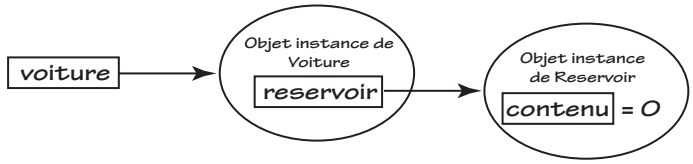
Au retour de la méthode, **setCarburant** est de nouveau appelée sur l'objet **voiture**. Nous constatons alors que cet objet a également été modifié. La figure de la page ci-contre illustre ce qui s'est passé.

La méthode **clone** a bien créé une copie de l'objet **voiture**. Cette copie contient une copie de chacun des membres de l'original. Les membres qui sont des primitives sont donc également copiés. En revanche, lorsque les membres sont des objets, les handles correspondants sont copiés mais ils pointent toujours vers les mêmes objets. Il s'agit là d'une copie "en surface". Pour remédier à ce problème, il faudrait réaliser une copie "en profondeur", c'est-à-dire en copiant de façon récursive tous les objets référencés à l'intérieur de l'objet copié, puis de nouveau tous les objets référencés par ces objets, et ainsi de suite.

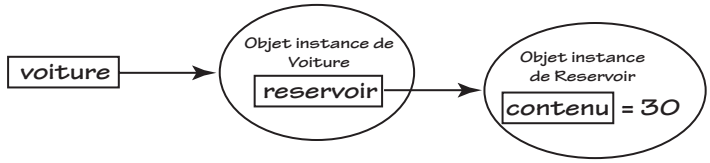
Pour exécuter un clonage en profondeur, vous devez modifier la méthode **clone()** de la classe **Voiture** afin qu'elle effectue un clonage explicite de tous les membres qui sont des objets. Pour obtenir ce résultat, nous pouvons modifier la méthode de la façon suivante :

```
public Object clone() {
   Object o = null;
   try {
      o = super.clone();
      ((Voiture)o).reservoir = (Reservoir)this.reservoir.clone();
   }
   catch (CloneNotSupportedException e) {
      System.out.println(e);
   }
   return o;
}
```

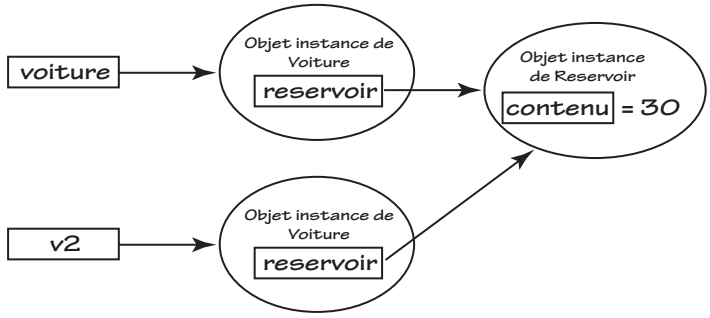
```
Voiture voiture = new Voiture();
```



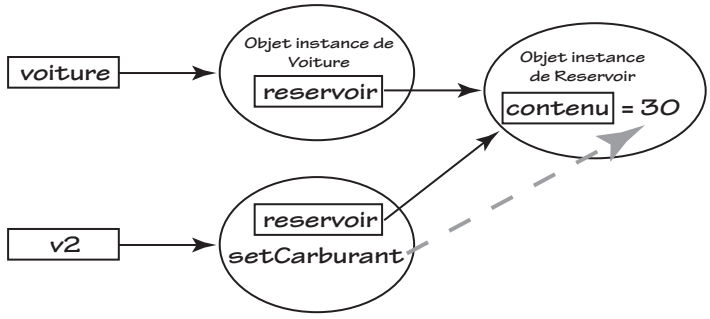

```
voiture.setCarburant(30);
```




```
lePlein(voiture);
    Voiture v2 = (Voiture)v.clone();
```




```
lePlein(voiture);
    v2.setCarburant(100);
```



Notez la syntaxe particulière utilisée ici pour le sous-casting de la valeur de retour de **super.clone()** (de type **Object**) en **Voiture**, sous-casting indispensable pour pouvoir accéder au champ **reservoir** :

```
((Voiture)o).reservoir =
```

Si vous préférez, vous pouvez modifier la méthode de la façon suivante :

```
public Object clone() {
   Voiture o = null;
   try {
      o = (Voiture)super.clone();
      o.reservoir = (Reservoir)this.reservoir.clone();
   }
   catch (CloneNotSupportedException e) {
      System.out.println(e);
   }
   return o;
}
```

Ici, c'est le résultat de **super.clone()** qui est explicitement sous-casté en **Voiture**. Le sur-casting inverse se produit implicitement dans l'instruction :

```
   return o;
```

o est de type **Voiture** mais est implicitement sur-casté en **Object** car la valeur de retour de la méthode est déclarée de ce type.

Bien entendu, pour que le clonage récursif fonctionne, il faut que chaque objet référencé soit lui-même clonable en profondeur. Pour les objets qui sont des instances de vos propres classes, il n'y a pas de problème. Il suffit que vous ayez défini correctement la méthode **clone()** pour chacune d'elles. La situation est évidemment différente lorsque vous ne contrôlez pas la définition des classes. Le cas le plus fréquent est celui des collections d'**Object**. Nous avons vu au Chapitre 10 que les collections référencent des

objets sur-castés en **Object**. Dans le cas du clonage d'un objet contenant ce type de collection, il est parfois impossible de savoir *a priori* si les objets référencés sont clonables en profondeur ou non. Il est facile de savoir s'ils sont clonables en utilisant l'expression logique **(o instanceof Cloneable)**, mais cela n'indique pas s'il s'agit d'un clonage de surface ou en profondeur.

Clonage en profondeur d'un objet de type inconnu

Si l'on souhaite cloner en profondeur un objet dont le type exact n'est pas connu, il est nécessaire d'effectuer les opérations suivantes :

- Interroger l'objet pour connaître la liste de ses champs.
- Tester chaque champ pour savoir s'il s'agit d'une primitive ou d'un handle d'objet.
- Pour chaque champ objet, vérifier s'il est clonable et, le cas échéant, cloner l'objet correspondant.
- Répéter l'opération de façon récursive.

Il s'agit là d'une opération très longue et complexe, faisant appel à des techniques que nous n'avons pas encore étudiées (*RTTI* et *réflexion*) qui permettent d'obtenir des informations sur la structure de la classe d'un objet (et en particulier, le nom de sa classe, la liste de ses champs, de ses méthodes, etc.).

Clonabilité et héritage

Bien que toutes les classes dérivent de la classe **Object**, elles ne sont clonables que si elles réunissent deux conditions :

- Elles redéfinissent la méthode **clone** ;
- Elles implémentent l'interface **Clonable**.

Si l'on considère maintenant l'héritage, on peut remarquer plusieurs points intéressants :

- Une classe dérivée d'une classe clonable est clonable.
- Une classe dérivée d'une classe ne réunissant aucune des deux conditions ci-dessus peut être clonable si elle réunit elle-même ces conditions. Dans l'exemple ci-après :

```
class Vehicule {
}
class Voiture extends Vehicule implements Cloneable {
   Reservoir reservoir;
   Voiture () {
      reservoir = new Reservoir();
   }
   void setCarburant(int e) {
      reservoir.setContenu(e);
   }
   int getCarburant() {
      return reservoir.getContenu();
   }
   public Object clone() {
      Voiture o = null;
      try {
         o = (Voiture)super.clone();
         o.reservoir = (Reservoir)this.reservoir.clone();
      }
      catch (CloneNotSupportedException e) {
         System.out.println(e);
      }
      return o;
   }
}
```

la classe **Voiture** est clonable, bien qu'elle dérive d'une classe qui ne l'est pas.

- Les deux conditions peuvent être remplies à des échelons différents de la hiérarchie. Dans les deux cas suivants, la classe **Voiture** est clonable. Dans le premier cas, la classe parente **Véhicule** implémente l'interface **Cloneable** et la classe dérivée **Voiture** redéfinit la méthode **Clone()** :

```
class Vehicule implements Cloneable {
}

class Voiture extends Vehicule {
   Reservoir reservoir;

   Voiture () {
      reservoir = new Reservoir();
   }


   void setCarburant(int e) {
      reservoir.setContenu(e);
   }

   int getCarburant() {
      return reservoir.getContenu();
   }

   public Object clone() {
      Voiture o = null;
      try {
         o = (Voiture)super.clone();
         o.reservoir =
                 (Reservoir)this.reservoir.clone();
      }
      catch (CloneNotSupportedException e) {
         System.out.println(e);
      }
      return o;
   }
}
```

Dans le second cas, c'est la classe parente qui redéfinit la méthode **clone()** et la classe dérivée qui implémente l'interface **Cloneable** :

```
class Vehicule {
   public Object clone() {
      Voiture o = null;
      try {
         o = (Voiture)super.clone();
         o.reservoir = (Reservoir)((Voiture)this).reservoir.clone();
      }
      catch (CloneNotSupportedException e) {
         System.out.println(e);
      }
      return o;
   }
}

class Voiture extends Vehicule implements Cloneable {
   Reservoir reservoir;

   Voiture () {
      reservoir = new Reservoir();
   }


   void setCarburant(int e) {
      reservoir.setContenu(e);
   }

   int getCarburant() {
      return reservoir.getContenu();
   }
}
```

Notez la syntaxe quelque peu alambiquée due à la nécessité d'effectuer un sous-casting explicite.

Interdire le clonage d'une classe dérivée

Il peut parfois être nécessaire d'interdire qu'une classe dérivée soit clonable. Deux cas se présentent :

- La classe parente n'est pas clonable.
- La classe parente est clonable.

Dans le premier cas, tout est simple. Il suffit de définir dans la classe parente une méthode **clone()** lançant l'exception **CloneNotSupportedException** :

```
public Object clone() throws CloneNotSupportedException {
   throw new CloneNotSupportedException();
}
```

Cette méthode ne retourne pas, mais cela n'a aucune importance car elle lance systématiquement une exception. Rien n'empêchera un utilisateur de la classe de la dériver en implémentant l'interface **Cloneable** et en définissant une méthode **clone()**. Cependant, si cette méthode fait appel à **super.clone()**, une exception **CloneNotSupportedException** sera systématiquement lancée.

Dans le second cas, il suffit de déclarer **final** la méthode **clone()** de la classe parente.

Note : Il est également possible de déclarer cette méthode **final** dans le premier cas, ce qui produira une erreur de compilation si l'utilisateur tente de redéfinir la méthode. C'est à vous, en tant que développeur de classes, de déterminer votre stratégie.

Une alternative au clonage : la sérialisation

Une autre approche est possible pour copier des objets en profondeur. Elle consiste à utiliser la *sérialisation*, technique que nous avons étudiée au

chapitre précédent. Telle que nous l'avions présentée, la sérialisation permettait d'enregistrer des objets sur disque afin de pouvoir les récupérer lors d'une exécution ultérieure du programme. Nous avons vu par ailleurs que la sérialisation était automatiquement exécutée en profondeur.

Pour réaliser le même programme en utilisant la sérialisation, il nous faut :

- Modifier la classe de l'objet à copier afin qu'elle implémente l'interface **Serializable**.
- Modifier les classes des objets membres de cette classe afin qu'elles implémentent également l'interface **Serializable**.
- Créer un **ObjectOutputStream** afin d'y écrire l'objet à copier.
- Créer un **ObjectInputStream** afin de lire l'objet copié.
- Relier ces deux streams afin que la sortie du premier corresponde à l'entrée du second.

Les deux premières conditions sont très faciles à réaliser. Voici le listing des classes modifiées :

```
class Voiture implements Serializable {
   Reservoir reservoir;
   Voiture () {
      reservoir = new Reservoir();
   }
   void setCarburant(int e) {
      reservoir.setContenu(e);
   }
   int getCarburant() {
      return reservoir.getContenu();
   }
}

class Reservoir implements Serializable {
   private int contenu;
```

```
    public int getContenu() {
       return contenu;
    }

    public void setContenu(int e) {
       contenu = e;
    }
 }
```

Vous pouvez voir que ces classes sont beaucoup plus simples car elles n'ont pas besoin de définir l'équivalent de la méthode **clone()**, pas plus en ce qui concerne la copie "en surface" (cas de la classe **Reservoir**) que la copie en profondeur (classe **Voiture**). Avec la sérialisation, tout est pris en charge automatiquement.

La troisième et la quatrième condition sont également faciles à réaliser en fonction de ce que nous avons appris au chapitre précédent. Il suffit de créer les deux streams de la façon suivante :

```
ObjectOutputStream objOutput = new ObjectOutputStream(...);
ObjectInputStream objInput = new ObjectInputStream(...);
```

Pour connecter ces deux streams, il nous faut créer deux ***PipedStream*** reliés l'un à l'autre. Nous pouvons le faire de deux façons équivalentes :

```
PipedOutputStream pipedOut = new PipedOutputStream();
PipedInputStream pipedIn = new PipedInputStream(pipedOut);
```

ou :

```
PipedInputStream pipedIn = new PipedInputStream();
PipedOutputStream pipedOut = new PipedOutputStream(pipedIn);
```

Les deux streams ainsi créés sont utilisés comme paramètres des constructeurs des ***ObjectStream*** précédents. Nous pouvons maintenant écrire les lignes complètes :

```
ObjectOutputStream objOutput = new ObjectOutputStream(pipedOut);
ObjectInputStream objInput = new ObjectInputStream(pipedIn);
```

Le listing complet du programme apparaît ci-après :

```
import java.io.*;

public class Clone2 {
   public static void main (String[] args) throws IOException {

      Voiture voiture = new Voiture();
      voiture.setCarburant(30);
      System.out.println(voiture.getCarburant());
      pleinSerial(voiture);
      System.out.println(voiture.getCarburant());
   }

   public static void pleinSerial(Voiture v) throws IOException {
      Voiture v2 = null;
      PipedInputStream pipedIn = new PipedInputStream();
      PipedOutputStream pipedOut = new PipedOutputStream(pipedIn);
      ObjectOutputStream objOutput =
                                    new ObjectOutputStream(pipedOut);
      ObjectInputStream objInput = new ObjectInputStream(pipedIn);

      objOutput.writeObject(v);
      try {
         v2 = (Voiture)objInput.readObject();
      }
      catch (ClassNotFoundException e) {
         e.printStackTrace();
      }
      objInput.close();
```

```
      objOutput.close();
      v2.setCarburant(100);
      System.out.println(v2.getCarburant());
   }
}

class Voiture implements Serializable {
   Reservoir reservoir;
   Voiture () {
      reservoir = new Reservoir();
   }

   void setCarburant(int e) {
      reservoir.setContenu(e);
   }

   int getCarburant() {
      return reservoir.getContenu();
   }
}

class Reservoir implements Serializable {
   private int contenu;
   public int getContenu() {
      return contenu;
   }

   public void setContenu(int e) {
      contenu = e;
   }
}
```

Vous pouvez noter quelques modifications supplémentaires :

- Nous importons le package **java.io** ;
- La méthode **main** et la méthode **pleinSerial** lancent une exception de type **IOException** ;

- La variable **v2** recevant sa valeur dans un bloc **try**, sa déclaration et son initialisation doivent être effectuées séparément, avant le bloc. Dans le cas contraire, le compilateur affiche un message d'erreur indiquant que la variable pourrait ne pas avoir été initialisée.

Ce programme donne le même résultat que le précédent, à une différence (de taille !) près. Il met, pour s'exécuter, 490 millisecondes, alors que le premier ne met que 160 millisecondes. Ces valeurs dépendent bien entendu de la machine sur laquelle le programme fonctionne. Si vous voulez les mesurer, il suffit de modifier la méthode **main** de la façon suivante :

```
long t1 = System.currentTimeMillis();
Voiture voiture = new Voiture();
voiture.setCarburant(30);
System.out.println(voiture.getCarburant());
lePlein(voiture);
System.out.println(voiture.getCarburant());
System.out.println(System.currentTimeMillis() - t1);
```

Ce qui est important ici est le rapport entre les temps d'exécution. Le programme utilisant la sérialisation met trois fois plus de temps à s'exécuter.

Une autre alternative au clonage

Une autre façon de copier un objet consiste à fournir une méthode ou un constructeur accomplissant le travail. De cette façon, vous pouvez contrôler totalement la façon dont les objets sont copiés. Voici un exemple utilisant une méthode :

```
class Voiture {
   Reservoir reservoir;

   Voiture () {
      reservoir = new Reservoir();
   }
```

```
   Voiture copier() {
      Voiture v = new Voiture();
      v.setCarburant(this.getCarburant());
      return v;
   }

   void setCarburant(int e) {
      reservoir.setContenu(e);
   }

   int getCarburant() {
      return reservoir.getContenu();
   }
}
```

Cette classe peut être utilisée de la façon suivante :

```
   public static void lePlein(Voiture v) {
      Voiture v2 = v.copier();
      System.out.println(v2.getCarburant());
```

Vous pouvez également employer un constructeur spécial :

```
class Voiture {
   Reservoir reservoir;

   Voiture () {
      reservoir = new Reservoir();
   }

   Voiture (Voiture v) {
      reservoir = new Reservoir();
      this.setCarburant(v.getCarburant());
   }
```

```
    void setCarburant(int e) {
       reservoir.setContenu(e);
    }
    int getCarburant() {
       return reservoir.getContenu();
    }
}
```

Dans ce cas, la copie sera effectuée de la façon suivante :

```
public static void lePlein(Voiture v) {
   Voiture v2 = new Voiture(v);
   System.out.println(v2.getCarburant());
```

Bien sûr, ces techniques sont beaucoup moins sophistiquées et ne permettent pas toujours de répondre à tous les problèmes, particulièrement dans le domaine de l'héritage.

Résumé

Dans ce chapitre, nous avons étudié la façon dont les arguments sont passés aux méthodes, ce qui nous a amenés à découvrir le clonage des objets. La façon dont le clonage des objets est implémenté en Java est source de discussions interminables. La question qui revient le plus souvent est : pourquoi cela a-t-il été fait ainsi ? Pourquoi implémenter la clonabilité dans la classe **Object**, mère de toutes les autres, et ne pas permettre automatiquement d'utiliser cette fonction dans les sous-classes. Chacun a son explication. Il n'y en a pas d'officielle. Une explication parfois avancée est un changement d'orientation dans les lignes directrices imposées aux concepteurs du langage du fait des problèmes de sécurité survenus en raison de son adaptation à Internet. C'est peut-être le cas. Pourtant, il n'y a rien de choquant dans la façon dont les choses sont organisées. La clonabilité est implémentée pour tous les objets, mais il y a une sécurité. Avancer l'explication précédente revient à se poser cette question : pourquoi les fabricants de pistolets automatiques ont-ils prévu un cran de sûreté ? Est-ce que cela

signifie qu'en fin de compte ils n'étaient plus très sûrs d'avoir envie de voir leurs produits utilisés ? Cette supposition est absurde. Telle qu'elle est implémentée, la clonabilité est une fonction très sûre et très facile à utiliser. Que demander de plus ? La clonabilité récursive automatique !

Chapitre 16

Exécuter plusieurs processus simultanément

Jusqu'ici, nous ne nous sommes pas vraiment occupés de savoir comment nos programmes fonctionnaient. Chaque programme était composé d'une classe principale et, éventuellement, d'autres classes annexes. La commande utilisée pour lancer le programme consistait en le nom de l'interpréteur suivi du nom de la classe. Nous allons voir maintenant ce qui se passe lorsque nous exécutons un programme de cette façon.

Lorsque nous lançons l'interpréteur à l'aide d'une ligne de commande en lui passant en paramètre le nom d'une classe, l'interpréteur est chargé en mémoire, puis il charge la classe indiquée et la parcourt à la recherche d'une méthode nommée **main**. S'il trouve une telle méthode, il démarre un *processus* et appelle cette méthode.

Qu'est-ce qu'un processus ?

Jusqu'ici, nous n'avons pas eu à répondre à cette question. L'interpréteur Java se chargeait pour nous de toutes les tâches nécessaires à l'exécution de nos programmes. Ce faisant, nous utilisions Java comme un langage mono-processus, c'est-à-dire ne permettant d'exécuter qu'une seule séquence d'instructions à la fois. Cependant, Java est un langage multi-processus, c'est-à-dire capable d'exécuter plusieurs séquences d'instructions. Chaque séquence est appelée *thread*, ce qui signifie *fil* en anglais. En Java, les threads sont tout simplement des objets instances de la classe **java.lang.Thread**, qui dérive directement de **java.lang.Object**.

Une instance de la classe **Thread** permet de créer un thread, mais sans grand intérêt, car il s'agit d'un thread ne faisant rien !

Avertissement

Dans ce chapitre, nous créerons et exécuterons des programmes qui lancent plusieurs processus (*threads*) s'exécutant en mémoire. Sous Windows, dans certaines circonstances, un programme peut se terminer alors qu'un processus continue de fonctionner. C'est le cas, en particulier, si vous interrompez à l'aide des touches *Ctrl+C* un programme qui est entré dans une boucle infinie. Si ce programme fonctionne dans une fenêtre DOS démarrée normalement, les threads seront automatiquement interrompus. En revanche, si vous exécutez ce type de programme depuis un environnement de développement (un simple éditeur de texte permettant de compiler et d'exécuter les programmes, par exemple), il est possible que certains processus ne soient pas interrompus et continuent d'occuper le processeur. Si vous constatez un ralentissement du système, tapez les touches *Ctrl+Alt+Del* (une seule fois !) pour afficher une fenêtre indiquant tous les processus en cours d'exécution. Si vous y trouvez le couple Java/Winoldap, vous êtes en présence du problème précité. Interrompez alors explicitement le processus en sélectionnant *Java* et en cliquant sur *Fin de tâche*.

Créer explicitement un thread

En Java, il existe deux façons de créer un thread. La première consiste à dériver la classe **Thread** et à instancier la classe dérivée. La seconde consiste à créer une classe implémentant une interface spéciale (**Runnable**) et à l'instancier. Cette deuxième façon de faire est utilisée lorsque la classe créée doit étendre une classe particulière et ne peut donc étendre la classe **Thread** (car l'héritage multiple n'est pas possible). C'est souvent le cas, en particulier, des applets, que nous étudierons dans un prochain chapitre.

Pour commencer, nous créerons un thread très simple qui affichera les nombres de 1 à 1000. La première chose à faire consiste à déclarer une classe étendant la classe **Thread** :

```
class monThread extends Thread {
}
```

Il nous faut maintenant définir ce que fera notre thread. Pour cela, nous devons redéfinir la méthode **run()** (en la déclarant **public** car nous ne pouvons en restreindre l'accès par rapport à la méthode de la classe parente) :

```
class monThread extends Thread {
   public void run() {
      for (int i = 1; i <= 1000; i++)
         System.out.println(i);
   }
}
```

Et voilà ! C'est aussi simple que cela. La question qui se pose maintenant est : comment exécuter notre thread ? Ce n'est pas plus compliqué. Il suffit d'instancier cette classe et d'invoquer la méthode **start()** de cette instance. (La méthode **start()** est évidemment définie dans la classe parente **Thread**.)

```
Thread t = new monThread();
t.start();
```

Ces instructions peuvent être placées dans la méthode **main** d'une autre classe :

```
public class Exemple {
   public static void main (String[] args) {
      Thread t = new monThread();
      t.start();
   }
}

class monThread extends Thread {
   public void run() {
      for (int i = 1; i <= 1000; i++)
         System.out.println(i);
   }
}
```

Nous pouvons également inclure une méthode **main** dans la classe **monThread** afin de rendre celle-ci exécutable :

```
public class monThread extends Thread {
   public void run() {
      for (int i = 1; i <= 1000; i++)
         System.out.println(i);
   }

   public static void main (String[] args) {
      Thread t = new monThread();
      t.start();
   }
}
```

Bien entendu, nous n'avions pas besoin de créer un thread pour afficher les 1000 premiers entiers. L'interpréteur Java aurait pu s'en charger pour nous. Il nous suffisait d'écrire le programme de la façon suivante :

```
public class Exemple {
   public static void main (String[] args) {
      for (int i = 1; i <= 1000; i++)
         System.out.println(i);
   }
}
```

Exécuter plusieurs threads simultanément

Le principal intérêt des threads est qu'ils permettent de définir plusieurs séquences d'instructions complètement indépendantes. Dans certains cas, ces séquences peuvent même s'exécuter de façon quasi simultanée. Le programme suivant en fait la démonstration :

```
public class monThread extends Thread {
   public void run() {
      for (int i = 1; i <= 1000; i++)
         System.out.println(i);
   }

   public static void main (String[] args) {
      new monThread().start();
      new monThread().start();
   }
}
```

Ici, deux instances de **monThread** sont créées anonymement et exécutées. Si vous lancez ce programme, vous constaterez que l'exécution des deux threads est simultanée, les lignes d'affichage étant parfaitement entrelacées :

```
1
1
2
2
```

```
3
3
4
4
etc.
```

Il n'y a cependant aucun moyen de savoir quel thread a produit chaque ligne, même si on peut supposer que la première de chaque groupe de deux est due au premier thread lancé. Pour avoir plus d'informations à ce sujet, nous pouvons modifier le programme de la façon suivante :

```
public class monThread extends Thread {
   public void run() {
      for (int i = 1; i <= 1000; i++)
         System.out.println(i + "  " + getName());
   }
   public static void main (String[] args) {
      new monThread().start();
      new monThread().start();
   }
}
```

Si vous exécutez plusieurs fois ce programme, vous risquez d'obtenir des résultats surprenants. Nous constatons qu'un thread peut parfois afficher plusieurs lignes pendant que l'autre semble ne rien faire ! Il s'agit là d'un point essentiel : à priori, les deux threads se déroulent de façon simultanée sans aucune synchronisation. Cet aspect ne doit jamais être négligé car il est source de nombreux problèmes. Examinez, par exemple, le programme suivant :

```
public class monThread2 extends Thread {
   static int j = 1;
   public void run() {
      for (int i = 1; i <= 20; i++)
         System.out.println(j++);
   }
```

```
    public static void main (String[] args) {
       new monThread2().start();
       new monThread2().start();
    }
}
```

Ce programme affiche les nombres de 1 à 40, ce qui nous paraît tout à fait naturel. En effet, chaque thread vient à son tour augmenter d'une unité la valeur de la variable static **j**. Mais ce n'est qu'une apparence. Rien ne nous permet d'affirmer avec certitude que le premier thread affiche les valeurs impaires et le second les valeurs paires. En modifiant cette fois encore le programme de la façon suivante :

```
System.out.println(j++ + "   " + getName());
```

on obtient un résultat variable, par exemple :

```
1   Thread-0
2   Thread-1
3   Thread-0
4   Thread-1
5   Thread-0
6   Thread-1
7   Thread-0
8   Thread-1
9   Thread-0
10   Thread-1
11   Thread-0
12   Thread-1
13   Thread-0
14   Thread-1
15   Thread-0
16   Thread-1
17   Thread-0
18   Thread-1
19   Thread-0
20   Thread-1
```

```
21   Thread-0
22   Thread-1
23   Thread-0
24   Thread-1
25   Thread-0
26   Thread-1
27   Thread-0
28   Thread-1
30   Thread-1
31   Thread-1
32   Thread-1
29   Thread-0
33   Thread-1
34   Thread-0
35   Thread-1
36   Thread-0
37   Thread-1
38   Thread-0
39   Thread-0
40   Thread-0
```

Attention : Le phénomène constaté ici n'est pas inhérent à Java, mais au système d'exploitation sur lequel fonctionne l'interpréteur. Ce point prendra une importance fondamentale lorsque nous écrirons des programmes exploitant cette particularité.

Si nous essayons une nouvelle version du programme, nous constatons un autre phénomène surprenant :

```
public class monThread2 extends Thread {
   static int j = 1;
   public void run() {
      for (int i = 1; i <= 20; i++) {
         System.out.println(j);
         j++;
      }
   }
```

```
    public static void main (String[] args) {
      new monThread2().start();
      new monThread2().start();
    }
}
```

Ce programme affiche (par exemple) le résultat suivant :

```
1
1
2
3
4
5
6
7
8
9
10
etc.
```

Il est tout à fait possible que vous obteniez un résultat différent avec, par exemple, des nombres manquants dans la série. Comment expliquer un tel résultat ? Il s'agit là encore du même problème, l'absence de synchronisation. Si l'on essaie d'analyser le fonctionnement du programme, on s'aperçoit que l'on peut avoir plusieurs cas de figure, par exemple :

```
Thread-0 : System.out.println(j);
Thread-0 : j++;
Thread-1 : System.out.println(j);
Thread-1 : j++;
```

Dans ce cas, chaque thread affiche la valeur incrémentée par le précédent. Voici un autre cas possible :

```
Thread-0 : System.out.println(j);
Thread-1 : System.out.println(j);
Thread-0 : j++;
Thread-1 : j++;
```

Dans ce cas, on obtient un affichage du type :

```
1
1
3
3
etc.
```

Les deux possibilités peuvent être mélangées à loisir. Dans le cas précédent, on peut supposer que l'ordre d'exécution était :

```
Thread-0 : System.out.println(j);
Thread-1 : System.out.println(j);
Thread-0 : j++;
Thread-0 : System.out.println(j);
Thread-1 : j++;
Thread-1 : System.out.println(j);
Thread-0 : j++;
etc.
```

Pour résoudre ce type de problème, il faut utiliser la synchronisation, que nous étudierons un peu plus loin.

Caractéristiques des threads

Comme vous avez pu le deviner à la lecture des exemples précédents, les threads ont un nom. Si aucun nom ne leur est attribué lors de leur création, ils reçoivent automatiquement le nom **Thread-** suivi du numéro d'ordre de leur création.

Attention : Le nom d'un thread n'a rien à voir avec le handle qui permet de le référencer. La documentation de Java parle de "thread anonyme" pour désigner les threads créés sans noms. Il ne s'agit pas pour autant d'objets anonymes. La ligne suivante crée un thread anonyme (au sens de la documentation de Java) qui n'est pas un objet anonyme :

```
Thread t = new MonThread2();
```

alors que, dans l'exemple suivant, on crée un thread nommé qui est un objet anonyme :

```
new MonThread2("Alfred").start();
```

Un thread peut également avoir une *cible*. Une cible est un objet instance d'une classe implémentant l'interface **Runnable**. Cette interface ne comporte qu'une seule méthode : **run()**. Si un thread est créé avec une cible, c'est la méthode **run()** de la cible qui est appelée lorsque la méthode **start()** est invoquée. Dans le cas contraire, la méthode **run()** du thread est invoquée.

Un thread peut également être créé avec l'indication d'un groupe auquel il appartiendra. Un groupe est une instance de la classe **java.lang.ThreadGroup**. L'appartenance à un groupe permet de traiter plusieurs threads en bloc. Il est ainsi possible, par exemple, de contrôler les autorisations de plusieurs threads en contrôlant leur appartenance à un groupe.

L'ordre des paramètres pour la création d'un thread est :

groupe, cible, nom

Chacun de ces paramètres peut être omis, mais *groupe* ne peut être utilisé seul.

Contrôler les threads

Plusieurs méthodes peuvent être invoquées pour contrôler le déroulement d'un thread :

- **yield()** interrompt le déroulement de façon quasi imperceptible afin de laisser du temps pour l'exécution des autres threads d'un même niveau de priorité.

- **sleep()** interrompt le déroulement pendant une durée qui peut être spécifiée en millisecondes ou en millisecondes et nanosecondes.

- **interrupt()** interrompt le déroulement.

- **join()** attend la fin de l'exécution (lance une interruption si le thread est interrompu).

Note : La méthode **yield()** a une importance considérable souvent sous-estimée. En effet, Java ne gère pas le fonctionnement concurrent de plusieurs threads, mais se repose entièrement sur le système d'exploitation. Avec un système comme Windows ou UNIX, le système alloue des "tranches" de temps machine à chaque thread. En revanche, avec d'autres systèmes, le premier thread démarré s'exécute jusqu'à ce qu'une des conditions suivantes soit remplie :

- Le thread se termine.

- Un autre thread de priorité supérieure démarre.

- Le thread cède explicitement le contrôle à un autre thread de priorité égale au moyen des méthodes **yield()** ou **sleep()**.

Si votre programme repose sur le fait que deux ou plusieurs threads doivent fonctionner de façon concurrente, et si ce programme doit être utilisable dans tous les environnements supportant Java, chacun de vos threads doit céder explicitement du temps aux autres (sans toutefois pouvoir choisir lesquels). Faites attention à la réciprocité : si un thread cède du temps à un

autre thread, il ne récupérera le contrôle que si celui-ci cède du temps à son tour. Dans les environnements gérant le fonctionnement concurrent des threads, l'utilisation de la méthode **yield()** peut entraîner un léger ralentissement global du programme en augmentant le nombre de transitions d'un thread à l'autre. (Deux threads s'exécutant l'un après l'autre sont terminés plus rapidement que deux threads s'exécutant de façon concurrente.)

Il existe également les méthodes **stop()**, **suspend()** et **resume()**, mais elles sont *deprecated* et ne doivent pas être utilisées. (Elles peuvent avoir disparu dans la prochaine version du langage.)

Ce sont des méthodes agressives qui risquent de laisser des objets dans un état indéfini. Pour arrêter un thread, par exemple, il est préférable d'implémenter dans ce thread une fonction d'arrêt pouvant être appelée de l'extérieur. Une façon de faire consiste à modifier une variable qui sera testée par le thread. En voici un exemple :

```
class Th1 extends Thread {
   boolean arret = false;

   public void run() {
      for (int i = 1; i <= 10000; i++) {
         if (arret)
            break;
         System.out.println(i);
      }
   }
}
```

Ce thread comporte une boucle affichant les nombres jusqu'à 10 000. Avant chaque affichage, la variable **arret** est testée. Si elle vaut **true**, le thread est interrompu.

Note : Ici, seule la boucle est interrompue. Cependant, comme aucune instruction ne se trouve après la boucle, cela revient à interrompre le thread.

Une autre façon de procéder très souvent utilisée consiste à employer une boucle **while** :

```
class Th1 extends Thread {
   boolean arret = false;

   public void run() {
      int i = 0;
      while (!arret)
         System.out.println(i++);
   }
}
```

Un tel thread pourra être arrêté à l'aide de l'instruction suivante :

```
non_du_thread.arret = true;
```

Il est cependant préférable de restreindre l'accès au champ **arret** et d'utiliser un accesseur, comme dans l'exemple suivant :

```
class Th1 extends Thread {
   private boolean arret = false;
   Th1(ThreadGroup tg) {
      super(tg, "Th1");
   }

   public void run() {
      int i = 0;
      while (!arret)
         System.out.println(i++);
   }

   public void arrete() {
      arret = true;
   }
}
```

Utilisation des groupes pour référencer les threads

Pour contrôler un thread de cette façon, il faut avoir accès à un handle pointant vers celui-ci. Si un thread A crée deux threads B et C, il sera facile de référencer B et C depuis A. En revanche, il est moins évident de référencer B depuis C ou C depuis B. Une des façons de procéder consiste à faire appartenir les deux threads à un même groupe. Chaque thread pourra ainsi interroger son groupe afin de connaître les autres membres du groupe. Le programme suivant montre un exemple de cette façon de procéder :

```
import java.io.*;

public class MonThread3 extends Thread {
   public static void main (String[] args) {
      ThreadGroup tg = new ThreadGroup("Groupe");
      new Th1(tg).start();
      new Th2(tg).start();
   }
}

class Th1 extends Thread {
   private boolean arret = false;
   Th1(ThreadGroup tg) {
      super(tg, "Th1");
   }

   public void run() {
      int i = 0;
      while (!arret){
           System.out.println(i++);
      }
   }

   public void arrete() {
      arret = true;
   }
}
```

```
class Th2 extends Thread {
   Thread[] ta = new Thread[2];
   Th2(ThreadGroup tg) {
      super(tg, "Th2");
   }

   public void run() {
      enumerate(ta);
      String s1 = "";
      BufferedReader r =
                     new BufferedReader(
                     new InputStreamReader(System.in));
      while (!s1.equals("s")) {
         try {
            while (s1 == "") {
                 s1 = r.readLine();
            }
            if (!s1.equals("s")) {
               s1 = "";
            }
         }
         catch (IOException e) {
         }
      }
      ((Th1)ta[0]).arrete();
   }
}
```

Le programme principal crée tout d'abord un groupe de threads appelé **tg**, puis deux threads affectés à ce groupe, l'un instance de la classe **Th1** et l'autre de la classe **Th2**. Ces deux threads sont volontairement créés en tant qu'objets anonymes et immédiatement démarrés. Il n'existe donc aucun handle permettant de les manipuler :

```
ThreadGroup tg = new ThreadGroup("Groupe");
new Th1(tg).start();
new Th2(tg).start();
```

Les classes **Th1** et **Th2** sont pourvues d'un constructeur, qui appelle le constructeur de la classe parente en lui passant les arguments. (Souvenez-vous qu'en l'absence de constructeur dans une classe, c'est le constructeur sans arguments de la classe parente qui est appelé automatiquement, ce qui ne conviendrait pas ici.)

La classe **Th1** définit un thread qui affiche la valeur de la variable **i** en l'incrémentant chaque fois, et ce tant que la variable **boolean arret** vaut **false**.

La variable **arret** est **private**, mais la classe dispose d'un *mutateur* pour modifier sa valeur. Il s'agit de la méthode **public arrete()**.

Ce thread affiche donc les nombres successifs à partir de 0 jusqu'à ce qu'une des conditions suivantes soit remplie :

- La variable **arret** prend la valeur **true**,
- Le thread est interrompu pour une cause extérieure (une panne d'électricité, par exemple !).

La classe **Th2** est un peu plus complexe. Elle commence par définir un tableau de threads. Rappelons qu'un tableau de threads contient des handles de threads, et non des threads. Ce tableau comporte deux éléments :

```
Thread[] ta = new Thread[2];
```

La première chose que fait la méthode **run()** est de remplir ce tableau au moyen de l'instruction suivante :

```
enumerate(ta);
```

La méthode **enumerate**, définie dans la classe **Thread**, remplit le tableau de threads qui lui est passé en argument avec les threads appartenant au même groupe que celui depuis lequel la méthode est invoquée.

La suite de la méthode **run** lit simplement le clavier jusqu'à ce que le contenu de la chaîne **s1** soit égal à "s". Lorsque c'est le cas, le premier thread du groupe est arrêté grâce à l'instruction :

```
((Th1)ta[0]).arrete();
```

Le premier élément de **ta** (correspondant à l'indice 0) est ici sous-casté en **Th1** (car il s'agit d'un handle de **Thread**) et sa méthode **arrete()** est invoquée, causant l'arrêt du thread.

Compilez et exécutez ce programme. Les chiffres défilent sur l'affichage jusqu'à ce que vous tapiez *s* puis la touche *Entrée*.

Ce programme n'était conçu ainsi que pour montrer comment il est possible de référencer un thread anonyme. Normalement, il est plus simple d'utiliser des handles, comme dans l'exemple suivant :

```
import java.io.*;

public class MonThread4 extends Thread {
   static Th1 th1;
   static Th2 th2;
   public static void main (String[] args) {
      th1 = new Th1();
      th2 = new Th2();
      th1.start();
      th2.start();
   }
}

class Th1 extends Thread {
   private boolean arret = false;
   private boolean interruption = false;

   public void run() {
      int i = 0;
      while (!arret){
```

```
            if (!interruption)
               System.out.println(i++);
         }
      }
      public void arrete() {
         arret = true;
      }
      public void interromp() {
         interruption = !interruption;
      }
   }

   class Th2 extends Thread {
      public void run() {
         String s1 = "";
         BufferedReader r =
                     new BufferedReader(
                     new InputStreamReader(System.in));
         try {
            while (s1 == "") {
               s1 = r.readLine();
               if (s1.equals("i"))
                  MonThread3.th1.interromp();
               if (s1.equals("s")) {
                  MonThread3.th1.arrete();
                  break;
               }
               s1 = "";
            }
         }
         catch (IOException e) {
         }
      }
   }
```

Ce programme est plus perfectionné : il permet en effet d'interrompre ou de reprendre l'affichage en tapant *i* et la touche *Entrée*, ou de l'arrêter définiti-

vement en tapant *s + Entrée*. Pourtant, il est plus facile à lire car nous avons utilisé l'instruction **break** pour arrêter le thread **Th2**.

Gérer la répartition du temps entre les threads

Lorsque plusieurs threads fonctionnent en même temps, il est parfois nécessaire de contrôler la façon dont le temps du processeur est réparti entre chacun d'eux. Considérez l'exemple suivant, qui affiche les nombres pendant dix secondes :

```
public class MonThread5 extends Thread {
   static Th1 th1;
   static Th2 th2;
   public static void main (String[] args) {
      th1 = new Th1();
      th2 = new Th2();
      th1.start();
      th2.start();
   }
}

class Th1 extends Thread {
   private boolean arret = false;
   private boolean interruption = false;

   public void run() {
      int i = 0;
      while (!arret){
         System.out.println(i++);
      }
   }

   public void arrete() {
      arret = true;
   }
}
```

```
class Th2 extends Thread {
   public void run() {
      long duree = 10000;
      long t1 = System.currentTimeMillis();
      long t = 0;
      while (t < duree)
         t = System.currentTimeMillis() - t1;
      MonThread5.th1.arrete();
   }
}
```

Le premier thread affiche les nombres comme précédemment. Le second compte le temps qui passe et arrête le premier lorsque ce temps a dépassé dix secondes. Compilez et exécutez ce programme et notez jusqu'à combien les nombres ont été affichés (dans notre cas, environ 1 300). Vous trouvez que ce n'est pas beaucoup ? Pour afficher un plus grand nombre de nombres en dix secondes, il suffit de donner plus de temps au premier thread. Une façon de procéder consiste à modifier le second thread de la manière suivante :

```
class Th2 extends Thread {
   public void run() {
      long duree = 10000;
      long t1 = System.currentTimeMillis();
      long t = 0;
      while (t < duree) {
         t = System.currentTimeMillis() - t1;
         yield();
      }
      MonThread5.th1.arrete();
   }
}
```

L'instruction **yield()** signifie "ne rien faire pendant un certain temps afin de donner du temps aux autres threads". Ainsi modifié, notre programme affiche les nombres jusqu'à 23 000. Notez qu'il est possible de placer plusieurs

instructions **yield()** les unes après les autres. Avec deux instructions, le compte atteint 30 000. Avec quatre, le programme va jusqu'à 34 000.

A titre de comparaison, nous aurions pu effectuer le même traitement avec un seul thread, de la façon suivante :

```
public class Simple {
   public static void main (String[] args) {
      int i = 0;
      long duree = 10000;
      long t1 = System.currentTimeMillis();
      long t = 0;
      while (t < duree) {
         t = System.currentTimeMillis() - t1;
         System.out.println(i++);
      }
   }
}
```

C'est beaucoup plus simple, mais également beaucoup moins performant. Ce programme ne compte que jusqu'à 9 000. Il est possible de l'optimiser :

```
public class MoinsSimple {
   public static void main (String[] args) {
      int i = 0;
      long duree = 10000;
      long t1 = System.currentTimeMillis();
      long t = 0;
      while (t < duree) {
         for (int j = 0; j < 1000; j++)
            System.out.println(i++);
         t = System.currentTimeMillis() - t1;
      }
      System.out.println(t);
   }
}
```

Ce programme peut paraître plus performant, surtout si l'on remplace la valeur 1 000 (en gras) par 10 000. Cependant, nous nous trouvons devant un autre problème : ici, nous vérifions le temps chaque fois que nous avons affiché 1 000 (ou 10 000) lignes. Le résultat est qu'au moment de la vérification, le temps peut être un peu dépassé (environ 200 millisecondes pour une valeur de 1 000), voire beaucoup (2 000 millisecondes pour une valeur de 10 000). En revanche, le programme utilisant les threads ne varie pas de plus de 50 millisecondes.

La priorité

Une autre façon de gérer la répartition du temps consiste à attribuer à chaque thread des niveaux de priorité différents. La priorité d'un thread est une caractéristique héritée du thread qui l'a créé. Il est cependant possible de la modifier, comme dans l'exemple suivant :

```
import java.io.*;

public class MonThread6 extends Thread {
   static Th1 th1;
   static Th2 th2;
   public static void main (String[] args) {
      th1 = new Th1();
      th2 = new Th2();
      th1.start();
      th2.start();
   }
}

class Th1 extends Thread {
   private boolean arret = false;
   private boolean interruption = false;

   public void run() {
      setPriority(MAX_PRIORITY);
      int i = 0;
      while (!arret){
```

```
            System.out.println(i++);
         }
      }

      public void arrete() {
         arret = true;
      }
   }


   class Th2 extends Thread {
      public void run() {
         setPriority(MIN_PRIORITY);
         long duree = 10000;
         long t1 = System.currentTimeMillis();
         long t = 0;
         while (t < duree) {
            t = System.currentTimeMillis() - t1;
         }
         MonThread5.th1.arrete();
         System.out.println(t);
      }
   }
```

MAX_PRIORITY et **MIN_PRIORITY** sont des valeurs de type **int** définies dans la classe **Thread**.

Il faut bien comprendre ce que signifie la priorité. Sur un ordinateur à un seul processeur, un seul thread peut fonctionner à un moment donné. Dans ce cas, le thread qui possède la plus haute priorité est exécuté. S'il se termine, ou s'il donne explicitement du temps en exécutant la méthode **yield**, le thread de priorité inférieure est exécuté.

Cependant, certains systèmes comme Windows ou UNIX simulent le fonctionnement d'un ordinateur multiprocesseur en affectant à tour de rôle à chaque thread un peu de temps machine. De cette façon, les threads sem-

blent s'exécuter simultanément. Le niveau de priorité détermine le rapport entre les temps alloués à chaque thread.

Tant qu'un thread s'exécute normalement, il ne peut être interrompu que par le système, qui "décide" de donner du temps à un autre thread. Comme nous l'avons vu précédemment, un thread peut également "laisser du temps" aux autres threads en invoquant la méthode **yield()**. Tout cela fait que deux threads qui fonctionnent simultanément ne sont pas synchronisés, ce qui explique les résultats que nous avons obtenus au début de ce chapitre.

La synchronisation

Une façon de résoudre ce problème consiste à synchroniser les threads. Lorsque deux threads doivent accéder à un même objet, il est parfois nécessaire qu'ils aient la possibilité de verrouiller cet objet. Ce principe est mis en œuvre par exemple, dans les bases de données. Si deux utilisateurs accèdent simultanément à la même fiche pour la modifier, le premier qui aura terminé sa modification verra son travail effacé lorsque le second enregistrera la sienne. Pour éviter cela, le premier utilisateur doit avoir un moyen de verrouiller la fiche pendant toute la durée de la modification.

Un autre exemple est celui d'une imprimante partagée. Si deux utilisateurs peuvent y accéder en même temps, il est tout de même préférable que le premier arrivé puisse la verrouiller le temps que son impression soit terminée. Dans le cas contraire, l'imprimante risquerait d'imprimer alternativement une page d'un utilisateur, puis une page d'un autre, et ainsi de suite. S'il s'agit d'imprimer deux documents de cent pages chacun, il n'y a plus qu'à trier.

Le programme suivant simule l'impression des deux documents :

```
public class Impression extends Thread {
   int j = 1;
   Impression(String s) {
      super(s);
   }
```

```
   public void run() {
      for (int i = 1; i <= 10; i++) {
         System.out.println("Impression de la page "
                         + j + " du " + getName());
         j++;
      }
   }

   public static void main (String[] args) {
      new Impression("premier document").start();
      new Impression("deuxieme document").start();
   }
}
```

Ce programme affiche le résultat suivant :

```
Impression de la page 1 du premier document
Impression de la page 1 du deuxieme document
Impression de la page 2 du premier document
Impression de la page 2 du deuxieme document
Impression de la page 3 du premier document
Impression de la page 3 du deuxieme document
Impression de la page 4 du premier document
Impression de la page 4 du deuxieme document
Impression de la page 5 du premier document
Impression de la page 5 du deuxieme document
Impression de la page 6 du premier document
Impression de la page 6 du deuxieme document
Impression de la page 7 du premier document
Impression de la page 7 du deuxieme document
Impression de la page 8 du premier document
Impression de la page 8 du deuxieme document
Impression de la page 9 du premier document
Impression de la page 9 du deuxieme document
Impression de la page 10 du premier document
Impression de la page 10 du deuxieme document
```

Pour éviter ce résultat, il faut que le premier thread puisse empêcher le second de commencer son travail jusqu'à ce qu'il ait fini le sien. Pour cela, il doit verrouiller un objet. Évidemment, il ne s'agit pas de n'importe quel objet. Il doit s'agir d'un objet sans le contrôle duquel le thread ne pourra pas fonctionner. L'objet **System.out** est évidemment un bon candidat.

Le programme modifié apparaît ci-dessous :

```
public class Impression extends Thread {
   int j = 1;
   Impression(String s) {
      super(s);
   }
   public void run() {
      synchronized (System.out) {
         for (int i = 1; i <= 10; i++) {
            System.out.println("Impression de la page "
                               + j + " du " + getName());
            j++;
         }
      }
   }
   public static void main (String[] args) {
      new Impression("premier document").start();
      new Impression("deuxieme document").start();
   }
}
```

La boucle correspondant à l'intégralité du traitement est placée dans un bloc contrôlé par **synchronized (System.out)**. De cette façon, l'objet **System.out** est verrouillé jusqu'à ce que le traitement soit terminé. Le programme produit maintenant le résultat suivant :

```
Impression de la page 1 du premier document
Impression de la page 2 du premier document
Impression de la page 3 du premier document
Impression de la page 4 du premier document
```

```
Impression de la page 5 du premier document
Impression de la page 6 du premier document
Impression de la page 7 du premier document
Impression de la page 8 du premier document
Impression de la page 9 du premier document
Impression de la page 10 du premier document
Impression de la page 1 du deuxieme document
Impression de la page 2 du deuxieme document
Impression de la page 3 du deuxieme document
Impression de la page 4 du deuxieme document
Impression de la page 5 du deuxieme document
Impression de la page 6 du deuxieme document
Impression de la page 7 du deuxieme document
Impression de la page 8 du deuxieme document
Impression de la page 9 du deuxieme document
Impression de la page 10 du deuxieme document
```

Note : La synchronisation consiste à obtenir un verrou sur un objet utilisé pour le traitement (ce qui empêche un thread concurrent de s'exécuter). Ici, nous aurions pu également verrouiller l'un ou l'autre thread (à condition de disposer d'un handle adéquat) de la façon suivante :

```
public class Impression extends Thread {
   int j = 1;
   static Impression I1, I2;
   Impression(String s) {
      super(s);
   }
   public void run() {
      synchronized (I1) {
         for (int i = 1; i <= 10; i++) {
            System.out.println("Impression de la page "
                               + j + " du " + getName());
            j++;
         }
      }
   }
```

```
      public static void main (String[] args) {
         I1 = new Impression("premier document");
         I2 = new Impression("deuxieme document");
         I1.start();
         I2.start();
      }
   }
```

Cette méthode est beaucoup moins logique et élégante, mais il est bon de la connaître : cela peut parfois servir.

La partie du code concerné par la synchronisation est dite *critique*. Souvent, cette partie critique est l'ensemble d'une méthode, et l'objet qui doit être verrouillé est celui qui contient la méthode. Dans ce cas, il est possible d'utiliser le modificateur **synchronized** dans la déclaration de la méthode :

```
   public synchronized void Methode() {
```

S'il s'agit d'une méthode d'instance, l'objet instance est verrouillé. S'il s'agit d'une méthode statique, c'est la classe qui est verrouillée.

L'exemple suivant simule l'impression de documents sur une imprimante. Chaque thread construit tout d'abord un document sous la forme d'un tableau de chaînes de caractères représentant les pages, puis la méthode **imprime** de l'imprimante est appelée avec ce tableau pour paramètre. Cette méthode imprime les pages une à une (en les affichant à l'écran). La première version du programme n'est pas synchronisée :

```
public class Impression extends Thread {
   Impression(String s) {
      super(s);
   }
   static Imprimante imprimante;
   String[] pages;
   public void run() {
      pages = new String[10];
```

```
      for (int i = 0; i < pages.length; i++)
         pages[i] = "Page " + (i+1) + " du " + getName();
      imprimante.imprime(pages);
   }

   public static void main (String[] args) {
      imprimante = new Imprimante();
      new Impression("premier document").start();
      new Impression("deuxieme document").start();
   }
}


class Imprimante {
   public void imprime(String[] s) {
      for (int i = 0; i < s.length; i++)
         System.out.println(s[i]);
   }
}
```

Elle imprime les pages mélangées :

```
   Page 1 du premier document
   Page 1 du deuxieme document
   Page 2 du premier document
   Page 2 du deuxieme document
   Page 3 du premier document
   Page 3 du deuxieme document
   Page 4 du premier document
   Page 4 du deuxieme document
   Page 5 du premier document
   Page 5 du deuxieme document
   Page 6 du premier document
   Page 6 du deuxieme document
   Page 7 du premier document
   Page 7 du deuxieme document
   Page 8 du premier document
```

```
Page 8 du deuxieme document
Page 9 du premier document
Page 9 du deuxieme document
Page 10 du premier document
Page 10 du deuxieme document
```

Les deux versions suivantes utilisent les deux types de synchronisation et produisent le résultat désiré :

```
public class Impression2 extends Thread {
   Impression2(String s) {
      super(s);
   }
   static Imprimante imprimante;
   String[] pages;
   public void run() {
      pages = new String[10];
      for (int i = 0; i < pages.length; i++)
         pages[i] = "Page " + (i+1) + " du " + getName();
      synchronized (imprimante) {
         imprimante.imprime(pages);
      }
   }

   public static void main (String[] args) {
      imprimante = new Imprimante();
      new Impression2("premier document").start();
      new Impression2("deuxieme document").start();
   }
}

class Imprimante {
   public void imprime(String[] s) {
      for (int i = 0; i < s.length; i++)
         System.out.println(s[i]);
   }
}
```

```
public class Impression3 extends Thread {
   Impression3(String s) {
      super(s);
   }
   static Imprimante imprimante;
   String[] pages;
   public void run() {
      pages = new String[10];
      for (int i = 0; i < pages.length; i++)
         pages[i] = "Page " + (i+1) + " du " + getName();
      imprimante.imprime(pages);
   }

   public static void main (String[] args) {
      imprimante = new Imprimante();
      new Impression3("premier document").start();
      new Impression3("deuxieme document").start();
   }
}
class Imprimante {
   public synchronized void imprime(String[] s) {
      for (int i = 0; i < s.length; i++)
         System.out.println(s[i]);
   }
}
```

Problèmes de synchronisation

Nous allons maintenant perfectionner ce programme en simulant une imprimante dotée d'un spooler :

```
   public class Impression4 extends Thread {
      Impression4(String s) {
         super(s);
      }
```

```
   static Imprimante imprimante;
   String[] pages;

   public void run() {
      synchronized(imprimante) {
         pages = new String[10];
         for (int i = 0; i < pages.length; i++)
            pages[i] = "Page " + (i+1) + " du " + getName();
         imprimante.spool(pages);
      }
   }

   public static void main (String[] args) {
      imprimante = new Imprimante();
      new Impression4("premier document").start();
      new Impression4("deuxieme document").start();
      imprimante.imprime();
   }
}

class Imprimante {
   String[] spooler = new String[100];
   int index = 0;

   public void imprime() {
      System.out.println("Page de garde : début d'impression");
      int j = 0;
      for (int i = 0; i < spooler.length; i++) {
         if (spooler[i] != null) {
            System.out.println(spooler[i]);
            j++;
            spooler[i] = null;
         }
      }
      System.out.println("Fin d'impression. " + j
                                          + " pages imprimees.");
   }
```

```
    public synchronized void spool(String[] s) {
       for (int i = 0; i < s.length; i++)
          spooler[index++] = s[i];
    }
 }
```

Cette fois, notre thread **Impression4** effectue les opérations suivantes :

- Il verrouille l'imprimante :

```
synchronized(imprimante) {
```

- Il crée un document vide de 10 pages, représenté par un tableau de chaînes de caractères :

```
pages = new String[10];
```

- Il remplit ce tableau avec des chaînes permettant d'identifier les pages et le document auquel elles appartiennent :

```
for (int i = 0; i < pages.length; i++)
   pages[i] = "Page " + (i+1) + " du " + getName();
```

(**getName()** renvoie le nom du thread, c'est-à-dire, dans notre exemple *premier document* ou *deuxième document*.)

- Le thread appelle ensuite la méthode **spool** de l'imprimante et lui transmet le document (c'est-à-dire le tableau).

```
imprimante.spool(pages);
```

Cette méthode copie une à une les pages du document dans la mémoire de l'imprimante, représentée par un autre tableau nommé **spooler** :

```
public synchronized void spool(String[] s) {
   for (int i = 0; i < s.length; i++)
      spooler[index++] = s[i];
}
```

La méthode **main**, de son côté, crée une imprimante puis deux threads correspondant à deux documents :

```
public static void main (String[] args) {
   imprimante = new Imprimante();
   new Impression4("premier document").start();
   new Impression4("deuxieme document").start();
```

Enfin, elle appelle la méthode **imprime** de l'imprimante, dont le rôle est de réaliser l'impression (affichage) de toutes les pages (chaînes de caractères) stockées dans sa mémoire :

```
   imprimante.imprime();
```

Cette méthode est très simple. Elle parcourt le tableau et affiche toutes les chaînes qu'il contient puis les annule. Une fois le traitement terminé, la méthode affiche un message récapitulatif :

```
public void imprime() {
   System.out.println("Page de garde : début d'impression");
   int j = 0;
   for (int i = 0; i < spooler.length; i++) {
      if (spooler[i] != null) {
         System.out.println(spooler[i]);
         j++;
         spooler[i] = null;
      }
   }
   System.out.println("Fin d'impression. " + j
                                      + " pages imprimees.");
}
```

Ce programme fonctionne très bien, mais il pourrait être amélioré. En effet, il n'est pas normal d'être obligé de demander explicitement à l'imprimante d'imprimer les pages stockées en mémoire. Elle devrait être capable de faire cela automatiquement. Nous entendons par là une contrainte bien précise :

- Le programme devra fonctionner de la même façon en supprimant simplement la ligne qui déclenche l'impression :

```
imprimante.imprime();
```

- Aucune autre modification ne devra être apportée à la classe principale. En revanche, nous avons toute liberté pour modifier la classe **Imprimante** (à condition de ne pas changer son interface, ce qui nous obligerait à modifier le programme principal).

Pour que cela soit possible, il faut que l'imprimante soit en mesure de vérifier régulièrement si des pages se trouvent en mémoire. Une solution consiste à faire de notre imprimante un thread. Pour compliquer un peu les choses, nous supposerons que la classe imprimante doive dériver d'une autre classe (que nous appellerons **Materiel** et qui sera vide dans notre exemple).

Puisque notre classe doit dériver de la classe **Materiel**, elle ne peut pas dériver de la classe **Thread**. Nous lui ferons donc implémenter l'interface **Runnable** :

```
class Materiel {
}

class Imprimante extends Materiel implements Runnable {
   String[] spooler = new String[100];
   int index = 0;
```

Nous ajouterons ensuite deux variables. La première est une variable de type **boolean** qui indique si des pages se trouvent en attente d'être imprimées. La deuxième est le thread que nous allons utiliser.

```
boolean pagesEnAttente = false;
Thread imprimanteThread;
```

Nous aurons besoin d'initialiser le thread et de le démarrer, ce que nous ferons dans le constructeur de la classe **Imprimante**.

```
Imprimante() {
   imprimanteThread = new Thread(Imprimante.this);
   imprimanteThread.start();
}
```

Nous voyons là la principale différence entre les deux façons de créer des threads (dériver la classe **Thread** ou implémenter l'interface **Runnable**). Dans le deuxième cas, nous créons un thread en instanciant directement la classe **Thread**, et non une classe dérivée, et en passant à son constructeur une *cible* qui est un objet implémentant l'interface **Runnable** et donc la méthode **Run**. Ici, nous avons choisi la classe imprimante comme cible, d'où l'utilisation de **this**.

La méthode **spool** sera modifiée par l'ajout d'une ligne permettant de donner la valeur **true** à la variable **pagesEnAttente** :

```
pagesEnAttente = true;
```

De la même façon, la méthode **Imprime** redonne la valeur **false** à cet indicateur lorsque toutes les pages sont imprimées.

La méthode **run** est évidemment entièrement nouvelle :

```
public void run() {
   imprimanteThread.setPriority(imprimanteThread.MIN_PRIORITY);
   while (true) {
      if (pagesEnAttente)
         imprime();
   }
}
```

Cette méthode est très simple : elle commence par donner au thread la priorité minimale puis entre dans une boucle sans fin dans laquelle elle teste la variable **pagesEnAttente**. Si cette variable vaut **true**, les pages sont imprimées. Le listing complet du programme apparaît ci-après :

```
public class Impression5 extends Thread {
   Impression5(String s) {
      super(s);
   }
   static Imprimante imprimante;
   String[] pages;

   public void run() {
      synchronized(imprimante) {
         pages = new String[10];
         for (int i = 0; i < pages.length; i++)
            pages[i] = "Page " + (i+1) + " du " + getName();
         imprimante.spool(pages);
      }
   }

   public static void main (String[] args) {
      imprimante = new Imprimante();
      new Impression5("premier document").start();
      new Impression5("deuxieme document").start();
   }
}

class Materiel {
}

class Imprimante extends Materiel implements Runnable {
   String[] spooler = new String[100];
   int index = 0;
   boolean pagesEnAttente = false;
   Thread imprimanteThread;
```

```
    Imprimante() {
       imprimanteThread = new Thread(Imprimante.this);
       imprimanteThread.start();
    }
    public synchronized void imprime() {
       System.out.println("Page de garde : debut d'impression");
       int j = 0;
       for (int i = 0; i < spooler.length; i++) {
          if (spooler[i] != null) {
             System.out.println(spooler[i]);
             j++;
             spooler[i] = null;
          }
       }
       System.out.println("Fin d'impression. " + j
                                              + " pages imprimees.");
       pagesEnAttente = false;
    }
    public synchronized void spool(String[] s) {
       for (int i = 0; i < s.length; i++)
          spooler[index++] = s[i];
       pagesEnAttente = true;
    }
    public void run() {
       imprimanteThread.setPriority(imprimanteThread.MIN_PRIORITY);
       while (true) {
          if (pagesEnAttente)
             imprime();
       }
    }
}
```

Ce programme fonctionne, mais présente un défaut. En effet, lorsque le thread principal (celui de la méthode **main**) et les deux threads créés par cette méthode sont terminés, le thread de l'imprimante est toujours actif. Il ne s'arrêtera jamais de lui-même car il comporte une boucle infinie (**while(true)**). Cela est cependant tout à fait normal. Lorsque nous créons

l'imprimante, elle se met en marche automatiquement. Il nous faut l'éteindre explicitement ou trouver un moyen d'obtenir automatiquement le même résultat. (En attendant, tapez *Ctrl+C* pour arrêter le programme.)

Pour éteindre automatiquement l'imprimante, il suffit de lui ajouter une méthode accomplissant cela, puis d'appeler cette méthode depuis la méthode **main**. Bien sûr, il faudra faire attention à ne pas éteindre l'imprimante avant qu'elle ait fini d'imprimer. La meilleure solution consiste à la doter d'un indicateur que nous appellerons **marche** et qui prendra la valeur **true** au démarrage. La méthode **eteindre** lui donnera la valeur **false**. Il nous suffira de remplacer alors la boucle **while(true)** par **while(marche || pagesEnAttente)**. Voici les parties du programme modifiées :

```
   public static void main (String[] args) {
      imprimante = new Imprimante();
      new Impression6("premier document").start();
      new Impression6("deuxieme document").start();
      imprimante.eteindre();
   }
            .
            .
            .
class Imprimante extends Materiel implements Runnable {
            .
            .
            .
   public void eteindre() {
      marche = false;
   }
   public void run() {
      marche = true;
      imprimanteThread.setPriority(imprimanteThread.MIN_PRIORITY);
      while (marche || pagesEnAttente) {
         if (pagesEnAttente)
            imprime();
      }
   }
}
```

Malheureusement ce programme ne fonctionne pas. En effet, l'instruction :

```
imprimante.eteindre();
```

de la méthode **main** est exécutée avant même que le premier thread ait pu terminer de spooler ses pages et de donner à **pagesEnAttente** la valeur **true**. La solution qui parait évidente consiste à modifier cet indicateur au début de la méthode, et non à la fin :

```
public synchronized void spool(String[] s) {
   pagesEnAttente = true;
   for (int i = 0; i < s.length; i++)
      spooler[index++] = s[i];
}
```

Cependant, un autre problème se pose maintenant : le programme ne s'arrête pas ! Pour comprendre pourquoi, modifions le programme de la façon suivante :

```
public synchronized void spool(String[] s) {
   pagesEnAttente = true;
   System.out.println("Spool : " + imprimanteThread.getName()
                      + " pagesEnAttente : " + pagesEnAttente
                      + ", marche : " + marche);
   for (int i = 0; i < s.length; i++)
      spooler[index++] = s[i];
}

public void eteindre() {
   marche = false;
   System.out.println("Eteindre : " + imprimanteThread.getName()
                      + " pagesEnAttente : " + pagesEnAttente
                      + ", marche : " + marche);
}
```

Si on exécute le programme ainsi modifié, on obtient le résultat suivant :

```
Eteindre : Thread-0 pagesEnAttente : false, marche :
false
Spool : Thread-0 pagesEnAttente : true, marche : true
Spool : Thread-0 pagesEnAttente : true, marche : true
Page de garde : debut d'impression
Page 1 du deuxieme document
Page 2 du deuxieme document
Page 3 du deuxieme document
      .
      .
      .
```

qui nous montre que la méthode **eteindre** est appelée avant que la méthode **spool** ait eu le temps de configurer l'indicateur **pagesEnAttente**.

La solution consiste à initialiser la variable avec la valeur **true** :

```
class Imprimante extends Materiel implements Runnable {
   String[] spooler = new String[100];
   int index = 0;
   boolean pagesEnAttente = false, marche = true;
   Thread imprimanteThread;
```

(Bien sûr, nous le savions déjà et l'erreur était volontaire, pour mettre en évidence les problèmes de synchronisation.)

Notre programme fonctionne ainsi, mais il ne respecte pas les critères que nous nous étions fixés. En effet, nous avons dû modifier la méthode **main()** en lui ajoutant l'invocation de la méthode **eteindre**. Ce que nous souhaitons obtenir est une imprimante qui s'éteigne automatiquement. Il existe plusieurs façons de le réaliser. L'imprimante pourrait tester la variable **pagesEnAttente** pendant un certain temps, puis s'éteindre si la valeur est toujours **false**. Nous allons cependant utiliser une autre méthode.

Mise en œuvre d'un démon

Une autre possibilité est de traiter l'imprimante comme un démon. Un démon (*daemon*) est un thread qui reste en mémoire pour servir d'autres threads. Il ne se termine pas de lui-même. En revanche, il est automatiquement terminé s'il reste seul à s'exécuter. Plusieurs démons peuvent fonctionner en même temps. Tant qu'il reste un thread vivant qui ne soit pas un démon, les démons continuent de fonctionner. S'il ne reste plus que des démons, ils sont tous terminés automatiquement.

Ce type de thread paraît adapté à l'usage que nous voulons en faire. Cependant, il se pose là aussi un problème de synchronisation. Le constructeur de la classe **Imprimante** pourrait être modifié de la façon suivante :

```
Imprimante() {
   imprimanteThread = new Thread(Imprimante.this);
   imprimanteThread.setDaemon(true);
   imprimanteThread.start();
}
```

Cependant, voilà ce qui risque d'arriver lors de l'exécution de la méthode **main** :

```
public static void main (String[] args) {
   imprimante = new Imprimante();
```

Le constructeur de la classe **Imprimante** est exécuté. Le thread est lancé en tant que démon. Comme le thread principal (celui de la méthode **main**) est en cours d'exécution, le démon reste actif.

```
new Impression5("premier document").start();
```

Le thread correspondant au premier document démarre.

```
        new Impression5("deuxieme document").start();
```

Le thread correspondant au deuxième document démarre. Il faut bien comprendre que le démarrage d'un thread prend un certain temps. Lorsque la méthode **main** se termine, ces deux threads sont toujours en cours de démarrage. (On peut dire en quelque sorte qu'ils n'ont pas fini de commencer.) A ce moment, aucun thread n'est en fonctionnement à part le démon. Celui-ci s'arrête donc avant que les deux autres ne soient considérés comme en fonctionnement.

Pour bien comprendre ce qui se passe, il faut savoir qu'un thread peut avoir trois états :

- Vivant
    - Fonctionnel (*runnable*)
    - Non fonctionnel
- Mort

Lorsqu'un thread est initialisé, par exemple :

```
imprimanteThread = new Thread(Imprimante.this);
```

il devient vivant mais non fonctionnel. C'est dans cet état (et dans cet état seulement) qu'il peut être déclaré démon.

Le thread devient fonctionnel lorsque sa méthode **start** est invoquée. Un thread fonctionnel est susceptible d'être en fonctionnement. Il n'y a cependant aucun moyen de s'en assurer. Cela dépend du gestionnaire de tâche, de sa priorité, et d'autres paramètres.

Un thread fonctionnel peut devenir non fonctionnel dans les cas suivants :

- Sa méthode **sleep** est invoquée.

- La méthode **wait** est appelée pour attendre la réalisation d'une condition donnée.

- Le thread est bloqué dans l'attente de l'exécution d'une opération d'entrée/sortie.

Un thread est mort lorsque son exécution est terminée.

A la lumière de ces explications, nous pouvons préciser la définition d'un démon :

> *Un démon est un thread qui reste vivant tant qu'il existe au moins un thread fonctionnel qui n'est pas un démon.*

Dans notre cas, lorsque le démon est démarré, les deux threads correspondant aux deux documents sont peut-être déjà vivants, mais pas encore fonctionnels. Le démon se termine donc immédiatement. Nous pouvons facilement imaginer un moyen de mettre cela en évidence. Il suffit de modifier la méthode **main** pour que le thread dans lequel elle s'exécute continue de vivre suffisamment longtemps pour que les deux autres threads prennent vie à leur tour :

```
public static void main (String[] args) {
   imprimante = new Imprimante();
   new Impression7("premier document").start();
   new Impression7("deuxieme document").start();
   try {
      do {
         sleep(1000);
      }
      while (imprimante.pagesEnAttente);
   }
   catch(InterruptedException e) {
   }
}
```

Ainsi modifié, le programme fonctionne, mais cela ne correspond évidemment pas à ce que nous souhaitions. En revanche, ce fonctionnement pourrait convenir si la méthode **main** devait effectuer d'autres traitements dont nous serions assurés qu'ils dureraient suffisamment longtemps pour que les threads démarrent. Cela peut paraître du bricolage, mais ce n'est pas obligatoirement le cas. Par exemple, si la suite du programme consiste à attendre l'entrée d'un utilisateur, on peut être assuré que cela durera suffisamment pour que cette condition soit remplie. Cependant il existe une solution bien plus élégante et efficace dans tous les cas.

Communication entre les threads : *wait* et *notifyAll*

Si on observe notre programme, on se rend compte que, pour résoudre ce problème, il suffirait de maintenir en vie un thread jusqu'à ce que le démon ait terminé son travail. Cela nécessite deux choses :

- Que le démon puisse envoyer un message pour indiquer qu'il a terminé.
- Qu'un thread reste en vie et attende ce message pour se terminer.

Nous avons immédiatement un volontaire pour attendre (et même deux !) : chaque thread imprimant un document peut parfaitement, avant de se terminer, attendre un signal indiquant que l'impression a été entièrement exécutée. Cette façon de procéder élude d'ailleurs un problème potentiel : que se serait-il passé si le démon avait correctement démarré et si tous les autres threads s'étaient terminés avant que toutes les pages soient imprimées ? L'impression aurait été incomplète !

Pour que le thread envoyant chaque document à l'imprimante attende le message indiquant que l'impression est terminée, il suffit d'ajouter, à la fin de la méthode **spool**, quelques lignes de code :

```
public synchronized void spool(String[] s) {
   for (int i = 0; i < s.length; i++)
      spooler[index++] = s[i];
   pagesEnAttente = true;
```

```
        try {
           wait();
        }
        catch(InterruptedException e) {
        }
    }
```

La méthode **wait()** ne vient pas de la classe **Thread**, mais de la classe **Object**, ce qui explique qu'elle puisse être utilisée ici de cette façon. (La classe imprimante n'étend pas la classe **Thread**.) Cette méthode entraîne l'arrêt du thread courant s'il possède le contrôle de l'objet dont la méthode **wait** est invoquée. Si la méthode n'a pas d'argument, le thread attend jusqu'à ce qu'il reçoive un message de notification. La méthode peut également prendre en argument un **int** indiquant un nombre de millisecondes, ou deux **int** indiquant un temps en millisecondes et nanosecondes. Dans ce cas, le thread attend pendant le temps indiqué sauf s'il reçoit un message de notification avant ce délai. Lorsque l'attente est interrompue par l'une ou l'autre condition, le thread attend de pouvoir reprendre le contrôle de l'objet pour continuer son exécution.

Pour envoyer le message de notification, nous pouvons utiliser les méthodes **notify()** ou **notifyAll()**, qui appartiennent également à la classe **Object**. La méthode **notify()** notifie arbitrairement un des threads en attente sur l'objet en question. Comme nous avons deux threads, nous pourrions utiliser deux fois de suite cette méthode. Nous pourrions également tester le nombre de threads et utiliser une boucle pour invoquer **notify()** autant de fois que nécessaire. Cependant, il est plus simple d'utiliser **notifyAll()**, qui notifie tous les threads en attente sur l'objet en question. Lorsque cette méthode est exécutée, tous les threads ainsi sortis de leur sommeil se battent comme des chiffonniers pour prendre le contrôle de l'objet sur lequel ils étaient en attente (celui dont la méthode **wait()** a été invoquée). Ce que nous voulons dire par là, c'est qu'il n'y a aucun moyen (simple) de déterminer quel thread reprendra son exécution le premier.

Attention : Cela ne signifie pas que l'ordre de reprise de contrôle soit indéterminé. Il est parfaitement déterminé pour la plupart des implémentations de JVM. Simplement, vous ne pouvez pas être sûr que l'ordre obtenu avec

une JVM sera identique à celui obtenu avec une autre. Par ailleurs, avec une même JVM, il n'est pas impossible que des conditions d'exécution modifiées produisent un ordre différent. Cela dit, toutes les expériences que nous avons pu réaliser ont montré que le premier arrêté était le premier à reprendre le contrôle.

Le listing suivant montre le programme complet :

```
public class Impression7 extends Thread {
   Impression7(String s) {
      super(s);
   }
   static Imprimante imprimante;
   String[] pages;
   public void run() {
      synchronized(imprimante) {
         pages = new String[10];
         for (int i = 0; i < pages.length; i++)
            pages[i] = "Page " + (i+1) + " du " + getName();
         imprimante.spool(pages);
      }
   }
   public static void main (String[] args) {
      imprimante = new Imprimante();
      new Impression7("premier document").start();
      new Impression7("deuxieme document").start();
   }
}
class Materiel {
}
class Imprimante extends Materiel implements Runnable {
   String[] spooler = new String[100];
   int index = 0;
   boolean pagesEnAttente = false, runnable = false;
   Thread imprimanteThread;
   Imprimante() {
      imprimanteThread = new Thread(Imprimante.this);
```

```
      imprimanteThread.setDaemon(true);
      imprimanteThread.start();
   }

   public synchronized void imprime() {
      System.out.println("Page de garde : debut d'impression");
      int j = 0;
      for (int i = 0; i < spooler.length; i++) {
         if (spooler[i] != null) {
            System.out.println(spooler[i]);
            j++;
            spooler[i] = null;
         }
      }
      System.out.println("Fin d'impression. " + j
                                          + " pages imprimees.");
      pagesEnAttente = false;
      notifyAll();
   }
   public synchronized void spool(String[] s) {
      for (int i = 0; i < s.length; i++)
         spooler[index++] = s[i];
      pagesEnAttente = true;
      try {
         wait();
      }
      catch(InterruptedException e) {
      }
   }

   public void run() {
      imprimanteThread.setPriority(imprimanteThread.MIN_PRIORITY);
      while (true) {
         if (pagesEnAttente)
            imprime();
      }
   }
}
```

Résumé

Nous avons vu dans ce chapitre l'essentiel de ce qui concerne l'utilisation des threads. Les threads sont d'une utilisation très simple, à condition de bien maîtriser les problèmes de synchronisation. Il est particulièrement important, si vous développez des applications devant faire l'objet d'une distribution publique, de bien faire la part de ce qui est du domaine fonctionnel de Java et de ce qui ressort des idiosyncrasies d'une JVM particulière. Ne supposez jamais qu'un thread s'exécutera avant un autre simplement parce que c'est logique ou parce que vous l'avez observé dans un environnement donné. Vous ne pouvez même pas supposer que deux threads identiques se termineront dans l'ordre où ils ont été lancés. Si plusieurs threads doivent être synchronisés, vous devez toujours le faire explicitement.

Exercice

Si vous avez encore du courage et souhaitez réviser les collections, vous pouvez essayer de modifier notre dernier programme pour que l'impression commence dès que la première page est spoolée. C'est un problème intéressant, car, dans ce cas, vous ne pouvez synchroniser l'imprimante, celle-ci devant être à même de recevoir les pages spoolées tout en imprimant les précédentes. (Pour que le problème soit réaliste, vous devrez ajouter un délai entre l'impression de chaque page afin de simuler le temps d'impression, qui doit être supérieur au temps nécessaire pour spooler. Le temps d'impression peut même être aléatoire pour simuler l'impression de pages plus ou moins complexes. Par ailleurs, toujours dans un souci de réalisme, la méthode **spool** devrait recevoir des tableaux de chaînes (les documents) et les stocker dans une liste liée.

Si ce problème vous semble trop complexe, vous pouvez essayer le suivant, plus simple :

- Écrire une classe simulant une usine produisant des marchandises, représentées par des nombres aléatoires.

- Écrire une classe simulant un entrepôt stockant ces marchandises. Lorsque le stock atteint un niveau donné, l'entrepôt doit notifier l'usine qui arrête sa production. Si le stock descend au-dessous d'une certaine valeur, l'entrepôt notifie l'usine qui reprend sa production.

- Écrire une classe simulant un client qui retire des marchandises de l'entrepôt. Si l'entrepôt est vide, le client doit attendre.

- Instancier une fois l'usine et l'entrepôt et plusieurs fois le client. Établir un mécanisme permettant de faire varier le rythme de la production en fonction du nombre de clients, de façon à maintenir l'usine en fonctionnement permanent.

Chapitre 17

RTTI et réflexion

Dans les chapitres précédents, nous avons souvent été amenés à effectuer des sur-castings implicites, en traitant un objet comme une instance d'une classe parente. Nous allons voir maintenant ce que cela implique du point de vue de l'interpréteur Java.

Le RTTI ou comment Java vérifie les sous-castings explicites

Java effectue automatiquement des sur-castings, par exemple lorsque nous stockons des objets dans un vecteur. En effet, un vecteur ne peut contenir que des objets. Lorsque nous plaçons dans un vecteur une instance d'une classe quelconque, elle est automatiquement sur-castée en instance de la classe **Object**. Considérez, par exemple, le programme suivant :

```
import java.util.*;
public class Animaux5 {
   public static void main(String[] argv) {
      Vector zoo = new Vector();
      int j;
      Random r = new Random();
      for (int i = 0; i < 10;i++) {
         j = Math.abs(r.nextInt()) % 3 + 1;
         switch(j) {
            case 1:
               zoo.add(new Chien());
               break;
            case 2:
               zoo.add(new Chat());
               break;
            case 3:
               zoo.add(new Canari());
         }
      }
      for (Enumeration e = zoo.elements() ; e.hasMoreElements() ;) {
         ((Animal)e.nextElement()).crie();
      }
   }
}
interface Animal {
   void crie();
}

class Canari implements Animal {
   public void crie() {
      System.out.println("Cui-cui !");
   }
}
class Chien implements Animal {
   public void crie() {
      System.out.println("Ouah-Ouah !");
   }
}
```

```
class Chat implements Animal {
   public void crie() {
      System.out.println("Miaou !");
   }
}
```

Ce programme crée au hasard dix instances de **Chien**, **Chat** ou **Canari** et les stocke dans le vecteur **zoo**. Une boucle **for** permet ensuite de les extraire une à une. Une fois extraites du vecteur, nous sommes en présence d'instances d'**Object**. Nous effectuons donc un sous-casting implicite en **Animal** afin de pouvoir invoquer la méthode **crie()** :

```
((Animal)e.nextElement()).crie();
```

Ce faisant, Java s'assure lors de l'exécution du programme que le sous-casting est possible en vérifiant le type de chaque objet. Cette vérification étant effectuée pendant l'exécution (et non pendant la compilation), elle est appelée *Run Time Type Identification* ou RTTI.

Pour vérifier si la vérification a lieu lors de l'exécution, il suffit d'ajouter la ligne suivante :

```
for (int i = 0; i < 10;i++) {
   j = Math.abs(r.nextInt()) % 3 + 1;
   switch(j) {
      case 1:
         zoo.add(new Chien());
         break;
      case 2:
         zoo.add(new Chat());
         break;
      case 3:
         zoo.add(new Canari());
   }
}
zoo.add(new Object());
```

```
   for (Enumeration e = zoo.elements() ; e.hasMoreElements() ;)
      ((Animal)e.nextElement()).crie();
   }
}
```

Le programme est alors compilé sans erreur, mais son exécution produit une exception :

```
java.lang.ClassCastException
```

Connaître la classe d'un objet

Ici, il nous a suffi de sous-caster les **Object** en **Animal** pour obtenir le résultat souhaité. Mais que faire si nous voulons connaître la classe exacte de chaque objet ? Pour comprendre la difficulté de la chose, il faut réaliser que, lorsque nous désignons une classe dans un programme, nous ne manipulons pas réellement une classe, mais une chaîne de caractères qui la représente. Considérez le problème suivant : On souhaite demander à l'utilisateur de taper un nom de classe au clavier. Si ce nom correspond à une classe existante, on l'instancie. En supposant que le nom tapé par l'utilisateur se trouve placé dans une chaîne de caractères appelée **s**, on obtiendrait quelque chose comme :

```
// entrée() est une méthode qui lit le nom entré au
// clavier par l'utilisateur
Object o;
String s = entrée();
      .
      .
o = new s();
```

Vous voyez le problème ? **s** n'est pas un nom de classe, mais un handle vers une chaîne contenant le nom de la classe. Certains langages possèdent une instruction ***evaluate*** qui permet de remplacer une variable par sa valeur. Java utilise un autre mécanisme.

A chaque classe correspond un objet de type **Class**. Cet objet est utilisé pour toutes les manipulations des classes. Un tel objet peut être obtenu de deux façons :

- En utilisant la méthode statique **forName()** de la classe **Class**.
- En utilisant une classe littérale, obtenue en faisant suivre le nom de la classe par **.class**.

La première manière donne, par exemple :

```
Class c = forName("Chien");
```

La deuxième manière est plus concise :

```
Class c = Chien.class
```

Grâce à cette technique, nous pouvons maintenant construire un tableau de classes contenant toutes les classes auxquelles un objet est susceptible d'appartenir et vérifier si une instance particulière appartient à chacune de ces classes. Le programme suivant en donne un exemple :

```
import java.util.*;

public class Animaux6 {
   public static void main(String[] argv) {
      Vector zoo = new Vector();
      int j = 0, chats = 0, chiens = 0, canaris = 0;
      Random r = new Random();
      for (int i = 0; i < 10;i++) {
         j = Math.abs(r.nextInt()) % 3 + 1;
         switch(j) {
            case 1:
               zoo.add(new Chien());
               break;
            case 2:
               zoo.add(new Chat());
               break;
```

```
            case 3:
               zoo.add(new Canari());
         }
      }
      Class[] ct = {Chien.class, Chat.class, Canari.class};
      Animal a;
      for (Enumeration e = zoo.elements() ; e.hasMoreElements() ;) {
         a = (Animal)(e.nextElement());
         for (j = 0; j < ct.length; j++) {
            if (ct[j].isInstance(a))
               break;
         }
         switch(++j) {
            case 1:
               System.out.print("Un chien : ");
               chiens++;
               break;
            case 2:
               System.out.print("Un chat : ");
               chats++;
               break;
            case 3:
               System.out.print("Un canari : ");
               canaris++;
         }
         a.crie();
      }
      System.out.println("Total : " + chiens + " chien(s), "
                + chats + " chat(s) et " + canaris + " canari(s).");
   }
}
interface Animal {
   void crie();
}
class Canari implements Animal {
   public void crie() {
      System.out.println("Cui-cui !");
   }
}
```

```
class Chien implements Animal {
   public void crie() {
      System.out.println("Ouah-Ouah !");
   }
}
class Chat implements Animal {
   public void crie() {
      System.out.println("Miaou !");
   }
}
```

Le tableau de classes est créé à l'aide de classes littérales :

```
Class[] ct = {Chien.class, Chat.class, Canari.class};
```

Deux boucles imbriquées permettent de comparer toutes les instances **a** avec toutes les classes au moyen de l'instruction :

```
if (ct[j].isInstance(a))
```

Notez le nom particulier de cette méthode qui est assez trompeur. Elle s'appelle **isInstance**, qui signifie "est instance" alors que son véritable sens est plutôt "est la classe à laquelle appartient le paramètre".

Notez que nous aurions pu utiliser l'opérateur **instanceof** pour vérifier la classe de chaque objet, mais que cela nous aurait obligés à écrire explicitement le nom de chaque classe, ce qui est beaucoup moins souple. (Attention de ne pas confondre le nom de la classe, par exemple **Chien**, avec la classe littérale **Chien.class**.)

Instancier une classe à l'aide de son objet *Class*

Jusqu'ici, nous avions instancié des classes en utilisant l'opérateur **new**. Nous pouvons maintenant utiliser une autre méthode. En effet, possédant un handle vers un objet de type **Class**, nous pouvons en créer une instance

à l'aide de la méthode **newInstance()**. Cette méthode crée une instance exactement comme si le constructeur sans argument était utilisé. Elle renvoie un **Object**, effectuant donc immédiatement un sur-casting. A l'aide de cette méthode, notre programme peut être réécrit de façon plus concise :

```
import java.util.*;
public class Animaux7 {
   public static void main(String[] argv) {
      Vector zoo = new Vector();
      int j = 0, chats = 0, chiens = 0, canaris = 0;
      Random r = new Random();
      Class[] ct = {Chien.class, Chat.class, Canari.class};
      for (int i = 0; i < 10;i++) {
         j = Math.abs(r.nextInt()) % 3;
         try {
            zoo.add(ct[j].newInstance());
         }
         catch(IllegalAccessException e) {
            e.printStackTrace(System.err);
         }
         catch(InstantiationException e) {
            e.printStackTrace(System.err);
         }
      }
      Animal a;
      for (Enumeration e = zoo.elements() ; e.hasMoreElements() ;) {
         a = (Animal)(e.nextElement());
         for (j = 0; j < ct.length; j++) {
            if (ct[j].isInstance(a))
               break;
         }
```

Connaître la classe exacte d'un objet

La classe **Class** ne comporte pas de méthode permettant de connaître directement la classe d'un objet. En revanche, la classe **Object** possède la méthode **getClass()** qui permet d'obtenir ce résultat. Cette méthode renvoie

bien évidemment un objet de type **Class**. Nous pouvons, dans notre programme, utiliser cette méthode pour remplacer la méthode **isInstance** :

```
for (j = 0; j < ct.length; j++) {
   if (a.getClass() == ct[j])
   break;
}
```

Il manque cependant une méthode importante : celle qui permettrait de sous-caster l'objet vers la classe obtenue. Si, par exemple, nos trois classes **Chien**, **Chat** et **Canari** possèdent chacune une méthode non héritée d'une classe parente, ce que nous avons appris jusqu'ici ne nous permet pas de l'invoquer sans effectuer un sous-casting explicite. Il est évidemment possible de le faire à l'aide d'une série d'instructions conditionnelles, mais cela n'est pas très pratique. La réflexion peut alors apporter une solution.

Utiliser la réflexion pour connaître le contenu d'une classe

Les techniques que nous allons décrire ici n'ont pas été prévues pour être employées par les développeurs d'applications Java, mais plutôt par ceux qui doivent concevoir des "méta-programmes", c'est-à-dire des programmes manipulant des programmes. C'est le cas, par exemple, des compilateurs ou des environnements de développement.

Utilité de la réflexion

Le concepteur d'un environnement de développement est amené à écrire un programme manipulant des composants Java (les JavaBeans). Un développeur d'application peut connaître l'interface d'une classe qu'il souhaite utiliser en se référant à la documentation fournie avec elle. En revanche, si un programme tel qu'un environnement doit utiliser un composant Java, il ne peut utiliser la même documentation. Il aurait été possible de créer un format de documentation lisible par un programme. Les concepteurs de Java ont jugé plus efficace de permettre à un programme d'interro-

ger directement un composant pour en connaître la structure. La réflexion est tout simplement la possibilité d'interroger une classe pour connaître ses membres. Cette technique peut être employée, par exemple, pour écrire un programme de décompilation.

Avertissement : En dehors des cas cités, utiliser la réflexion n'est généralement pas la meilleure solution. Comme nous l'avons déjà dit plusieurs fois, le polymorphisme permet le plus souvent de résoudre les problèmes de façon plus efficace et plus élégante. C'est le cas, en particulier, des exemples présentés ici.

Imaginons que chaque animal de l'exemple précédent soit capable de faire autre chose que crier, par exemple :

```
class Canari implements Animal {
   public void crie() {
      System.out.println("Cui-cui !");
   }
}
class Chien implements Animal {
   public void crie() {
      System.out.println("Ouah-Ouah !");
   }
   public void grogne() {
      System.out.println("Grrrrrrr...");
   }
}
class Chat implements Animal {
   public void crie() {
      System.out.println("Miaou !");
   }
   public void ronronne() {
      System.out.println("Rrrr...Rrrr...");
   }
}
```

Imaginons de plus que vous ne possédiez pas les sources de ces classes, mais seulement les classes compilées, sans documentation. Vous savez que

chaque animal possède la méthode **crie()**. Comment savoir ce que chaque animal peut faire de plus ? En utilisant la réflexion. Ici, il ne nous servirait pas à grand-chose de connaître la classe à laquelle chaque objet appartient. Nous voulons seulement connaître la liste des méthodes qu'il déclare, ou plutôt, sachant qu'il peut déclarer ou non une méthode **public**, nous voulons pouvoir savoir si c'est le cas et, si oui, l'exécuter.

Pour résoudre ce problème, il nous suffit d'interroger l'objet pour connaître la liste de ces méthodes, puis comparer le dernier élément de cette liste avec le dernier élément de la liste des méthodes de la classe instanciée par l'objet. Si ces deux éléments désignent la même méthode, cela signifie que l'objet ne possède pas de méthode supplémentaire. Dans le cas contraire, nous invoquerons la dernière méthode de l'objet.

Note : Il s'agit d'un exemple simplifié. Normalement, nous devrions essayer de savoir quels sont les paramètres de la méthode. Nous supposerons ici que nous savons qu'elle n'en prend pas.

```
import java.util.*;
import java.lang.reflect.*;
public class Animaux9 {
   public static void main(String[] argv) {
      Vector zoo = new Vector();
      int j = 0, chats = 0, chiens = 0, canaris = 0;
      Random r = new Random();
      Class[] ct = {Chien.class, Chat.class, Canari.class};
      for (int i = 0; i < 10;i++) {
         j = Math.abs(r.nextInt()) % 3;
         try {
            zoo.add(ct[j].newInstance());
         }
         catch(IllegalAccessException e) {
            e.printStackTrace(System.err);
         }
         catch(InstantiationException e) {
            e.printStackTrace(System.err);
         }
      }
```

```
        Animal a;
        for (Enumeration e = zoo.elements() ; e.hasMoreElements() ;) {
           a = (Animal)(e.nextElement());
           for (j = 0; j < ct.length; j++) {
              if (a.getClass() == ct[j])
                 break;
           }
           switch(++j) {
              case 1:
                 System.out.print("Un chien : ");
                 chiens++;
                 break;
              case 2:
                 System.out.print("Un chat : ");
                 chats++;
                 break;
              case 3:
                 System.out.print("Un canari : ");
                 canaris++;
           }
           a.crie();
           try {
              Method[] ma = a.getClass().getMethods();
              Method[] mAnimal = Animal.class.getMethods();
              Method m1 = ma[ma.length - 1];
              Method m2 = mAnimal[mAnimal.length - 1];
              if (!m1.getName().equals(m2.getName()))
                 m1.invoke(a, new Object[0]);
           }
           catch(IllegalAccessException se) {
              System.out.println(se);
           }
           catch(InvocationTargetException se) {
              System.out.println(se);
           }
        }
        System.out.println("Total : " + chiens + " chien(s), "
                                      + chats + " chat(s) et "
                                      + canaris + " canari(s).");
     }
  }
```

```
interface Animal {
   void crie();
}

class Canari implements Animal {
   public void crie() {
      System.out.println("Cui-cui !");
   }
}
class Chien implements Animal {
   public void crie() {
      System.out.println("Ouah-Ouah !");
   }
   public void grogne() {
      System.out.println("Grrrrrrr...");
   }
}
class Chat implements Animal {
   public void crie() {
      System.out.println("Miaou !");
   }
   public void ronronne() {
      System.out.println("Rrrr...Rrrr...");
   }
}
```

La partie importante est celle imprimée en gras. Elle construit tout d'abord deux tableaux, **ma** et **mAnimal**, contenant la liste des méthodes publiques de la classe **Animal** (en fait, une interface) et de la classe de l'objet **a** :

```
Method[] ma = a.getClass().getMethods();
Method[] mAnimal = Animal.class.getMethods();
```

Nous créons ensuite deux objets de type **Method**, appelés **m1** et **m2**, pour manipuler facilement le dernier élément de chacun des tableaux. Cette étape n'est là que pour la lisibilité du programme :

```
Method m1 = ma[ma.length - 1];
Method m2 = mAnimal[mAnimal.length - 1];
```

Si les noms de ces deux méthodes sont différents, la dernière méthode de l'objet **a** est invoquée :

```
if (!m1.getName().equals(m2.getName()))
   m1.invoke(a, new Object[0]);
```

La syntaxe utilisée pour invoquer une méthode est :

```
méthode.invoke(objet, tableau_de_paramètres)
```

où méthode est un objet de type **Method**, *objet* l'objet dont la méthode est invoquée et *tableau_de_paramètres* un tableau d'objets correspondant aux paramètres de la méthode. Ici, la méthode ne prend pas de paramètres, mais nous devons tout de même lui passer un tableau de longueur nulle.

Attention : Nous comparons les noms des méthodes, ce que nous aurions pu faire sous la forme :

```
if (!(m1.getName() == m2.getName())
```

En revanche, il n'est pas possible de comparer directement les méthodes de la façon suivante :

```
if (!(m1 == m2)
```

car **m1** et **m2** sont forcément des handles pointant vers des objets différents. En effet, les tableaux **ma** et **mAnimal** ne contiennent pas les méthodes de l'objet **a** et de la classe **Animal**, mais des objets de type méthodes représentant ces méthodes et créés à l'aide de **getMethod()**. Il faut être bien conscient de cette différence. Tout comme un handle n'est pas l'objet qu'il représente, une instance de **Method** représente une méthode mais n'est pas

une méthode. **m1** et **m2** ne sont pas des méthodes et ne représentent jamais la même méthode, ce qui fait deux raisons suffisantes pour que l'égalité :

```
m1 == m2
```

ne soit jamais vérifiée. **m1** représente la dernière méthode de la classe de l'objet **a**, c'est-à-dire par exemple :

```
public void Canari.crie()
```

si l'objet est une instance de **Canari**, alors que **m2** représente la dernière méthode de la classe **Animal**, c'est-à-dire :

```
public abstract void crie()
```

car **Animal** est une interface. (Toutes les méthodes d'une interface sont implicitement **abstract**.)

De plus, considérez les lignes suivantes :

```
Method[] ma = a.getClass().getMethods();
Method[] mAnimal = a.getClass().getMethods();
Method m1 = ma[ma.length - 1];
Method m2 = mAnimal[mAnimal.length - 1];
System.out.println(m1);
System.out.println(m2);
System.out.println(m1 == m2);
```

Si **a** est une instance de **Chien**, le résultat affiché par ces lignes sera :

```
public void Canari.crie()
public void Canari.crie()
false
```

Bien que les objets désignés par **m1** et **m2** représentent ici la même méthode (contrairement au cas précédent), il s'agit tout de même de deux objets différents. L'égalité renvoie donc la valeur **false**.

Utiliser la réflexion pour manipuler une classe interne

Au Chapitre 12, nous avions évoqué un problème qui pouvait être résolu à l'aide de la réflexion. Voici maintenant le listing du programme mettant en œuvre cette solution :

```
import java.lang.reflect.*;
public class Reflexion {
   private Interne7 o;
   Reflexion() {
      o = new Interne7();
   }
   public static void main (String[] args) {
      Reflexion i  = new Reflexion();
      i.o.afficheCarré(5);
      i.o.afficheCube(5);
   }
}

class Interne7 {

   void afficheCarré(final int x) {
      class Local {
         public long result() {
            return x * x;
         }
      }
      print(new Local());
   }

   void afficheCube(final int x) {
      class Local {
         public long result() {
```

```
            return x * x * x;
          }
        }
        print(new Local());
    }

    void print(Object o) {
        try {
            Method m = o.getClass().getMethod("result",
                                                new
Class[0]);
            System.out.println(m.invoke(o, new Object[0]));
        }
        catch(Exception e) {System.out.println(e);}
    }
}
```

Ici, la RTTI ne peut être utilisée car le compilateur refuse l'utilisation des noms de classes internes du type **Interne7$1$Local**. Ces noms peuvent être employés uniquement dans la signature de la méthode, mais pas dans le corps de celle-ci. La réflexion permet de résoudre le problème.

Il va de soi que cet exemple n'a d'autre intérêt que celui de la démonstration. La seule solution logique consiste à exploiter le polymorphisme en faisant dériver les classes locales d'une interface comportant une méthode **print**, comme nous l'avons fait au Chapitre 12.

Utiliser la réflexion pour créer des instances

Nous avons vu que la méthode **newInstance()** permet de créer une instance d'un objet exactement comme avec l'opérateur **new**, en appelant le constructeur sans argument. La réflexion permet d'aller plus loin en invoquant n'importe quel constructeur avec les paramètres convenable. Les paramètres doivent simplement être placés dans un tableau d'**Object**. La méthode **getConstructor()** renvoie un tableau contenant tous les constructeurs de la classe. La méthode **getConstructor(Class[] type)** renvoie le constructeur

ayant pour signature les types correspondant aux éléments du tableau **type**. La classe **Constructor** possède une méthode **newInstance(Object[] initargs)** qui permet d'invoquer un constructeur en lui passant ses paramètres sous forme d'un tableau d'objets. (Ne confondez pas **type**, qui est un tableau de **Class** et **initargs**, qui est un tableau d'objets.)

Conclusion : quand utiliser la réflexion ?

Nous avons dit que, dans la plupart des cas, il valait mieux utiliser le polymorphisme que de mettre en œuvre la réflexion, sauf lorsque l'obtention d'information concernant une classe est l'objet principal du programme, comme dans le cas d'un décompilateur ou lorsque vous utilisez des classes non documentées et dont les sources ne sont pas disponibles. Il est toutefois une autre situation où la réflexion peut être utile. Si vous avez créé un certain nombre de classes que vous avez diffusées publiquement, vous pouvez être amené à ajouter des fonctionnalités sans vouloir modifier les classes existantes pour, par exemple, leur faire implémenter une interface particulière. Dans ce cas, la réflexion n'est peut-être pas la solution la plus élégante, mais c'est parfois la seule qui vous permette d'étendre une application sans imposer une mise à jour des classes existantes à tous vos utilisateurs. Ce type de *patch* ("Rustine") n'est peut-être pas toujours d'une grande élégance, mais il rend tout de même bien des services.

Chapitre 18

Fenêtres et boutons

Déjà plus de six cent pages et toujours pas la moindre fenêtre, pas le plus petit bouton, aucun menu ni aucune liste déroulante ! Vous vous demandez peut-être si seulement Java dispose de ces éléments. Java dispose de tous ces éléments et de bien d'autres qui permettent de construire des interfaces utilisateur très performantes. Il se trouve simplement que leur utilisation nécessite de bien comprendre les mécanismes du langage. Voilà pourquoi leur étude a été mise de côté jusqu'ici, alors qu'il s'agit probablement d'un des sujets qui intéressent le plus les utilisateurs.

Java va beaucoup plus loin que la plupart des langages dans ce domaine, en proposant un choix de plusieurs interfaces utilisateur pouvant être sélectionnées pendant l'exécution d'un programme. Un utilisateur peut ainsi choisir une interface semblable à celle d'un PC sous Windows, d'un Macintosh, d'une station UNIX sous Motif, ou encore une interface spéciale développée exclusivement pour Java. Cela est possible parce que les concepteurs de Java ont dissocié l'aspect fonctionnel de l'interface de son aspect graphi-

que, encore appelé "look and feel". Il est aussi possible de choisir une interface de Macintosh sur un PC ou inversement. Ce choix peut même être fait pendant l'exécution du programme. En tant que programmeur, vous pouvez laisser ou non l'utilisateur choisir son interface, ou faire en sorte que votre programme se conforme automatiquement à l'interface du système sur lequel il est utilisé, ou encore qu'il ait la même interface sur tous les systèmes.

Toutes ces possibilités reposent simplement sur le fait que les éléments de l'interface (les boutons, les fenêtres, les barres de défilement, etc.) sont écrits en Java et non empruntés au système (à quelques exceptions près, comme nous allons le voir).

Les composants lourds et les composants légers

En Java, les éléments de l'interface utilisateur sont des *composants*, c'est-à-dire des objets dérivant de la classe **Component**. Il existe deux sortes de composants : les composants lourds et les composants légers.

Les composants lourds

Les composants lourds sont ceux dont l'implémentation dépend du système. Dans la plupart des langages, l'interface utilisateur ne comporte aucun de ces composants. L'utilisation de fenêtres ou de boutons se fait grâce à des appels aux fonctions du système hôte. Dans ce cas, un programme ne peut fonctionner que sur un seul système, même si le langage existe sur un autre système. Ainsi, un programme écrit en C pour Windows ne pourra pas être compilé pour un Macintosh car les fonctions de ces deux systèmes, bien que très proches au plan fonctionnel, n'ont pas la même syntaxe. Ce programme déjà non compatible sous forme compilée ne l'est donc pas non plus sous forme de source.

Certains langages proposent leurs propres fonctions s'interfaçant avec celles du système. Ainsi, l'utilisation d'un bouton se fera à l'aide d'une syntaxe propre au langage que le compilateur traduira en un appel au système. Le

programme pourra être compatible au niveau source si le programmeur (ou le compilateur) se restreint aux fonctions communes aux deux langages.

En Java, quelques composants sont de ce type. Cela est indispensable car un programme doit bien, d'une façon ou d'une autre, communiquer avec le système. La différence est qu'en Java, le nombre de composants de ce type est extrêmement réduit. La fenêtre est un des composants qui, par nature, doivent dépendre du système. Les composants contenus dans une fenêtre n'ont pas besoin de communiquer avec le système. Il leur suffit de communiquer avec la fenêtre. Cela est également vrai pour les fenêtres secondaires créées à l'intérieur d'une autre fenêtre.

Depuis le début de ce livre, nous avons dit et répété que tout, en Java, était objet. Comment un composant dépendant du système peut-il être un objet standard de Java ? Une solution consiste à doter cet objet de méthode *native*, c'est-à-dire dépendante du système. Dans ce cas, l'objet en question sera différent selon le système, ce qui pose de nombreux problèmes. La solution employée par Java est un peu différente. Le composant est identique pour tous les systèmes, mais il est associé à un élément dépendant du système et appelé *peer*. Grâce à ce procédé, la communication entre le composant et le système est réduite au minimum. Un tel composant est dit *lourd*.

Les composants légers

Les composants *légers* sont des composants entièrement écrits en Java et donc identiques sur tous les systèmes. La plupart des composants d'interface utilisateur sont de ce type. Cela a une implication directe importante. Un bouton, par exemple, est un composant léger. Il n'est donc pas dessiné par le système, mais par Java. Par conséquent, un programme fonctionnant sur un Macintosh peut parfaitement dessiner un bouton Windows. Le programme peut même être doté d'un menu transformant immédiatement l'interface sans avoir à redémarrer le programme. Par ailleurs, si, en tant que programmeur, vous n'êtes pas satisfait des interfaces proposées, vous pouvez parfaitement créer la vôtre. Vous n'avez même pas à vous préoccuper de l'aspect fonctionnel de l'interface, mais seulement de créer un nouvel aspect graphique. Vous pouvez cependant parfaitement créer de nouveaux

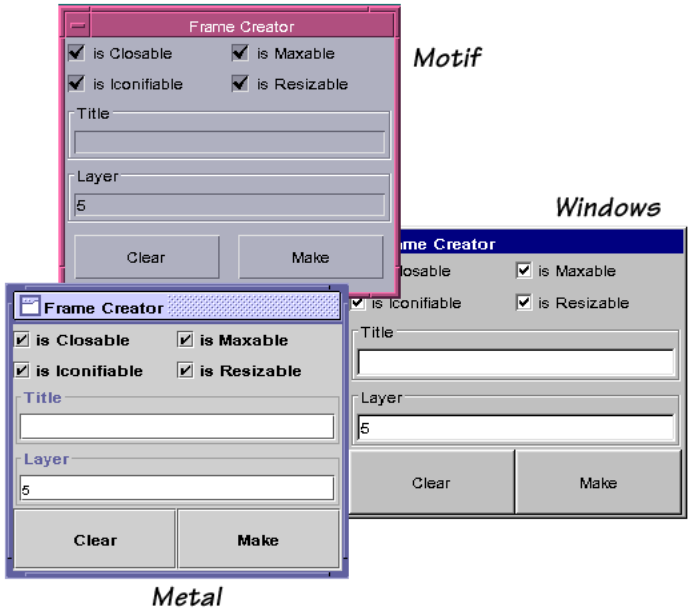
composants si vous avez des idées originales sur les fonctionnalités que devrait offrir une interface utilisateur.

Le look & feel

La modification de l'aspect de l'interface se fait à l'aide des "pluggable look and feel". Java est livré avec quatre "look and feel" différents :

- Windows
- Macintosh
- Motif
- Metal

Le look *Motif* est celui employé sur de très nombreuses stations de travail UNIX. Le look *Metal* a été créé spécialement pour Java. Les illustrations suivantes montrent les différents look & feel :



Note : Java possède aussi tous les éléments de l'interface utilisateur sous forme de composants lourds. Nous ne nous y intéresserons pas dans ce livre.

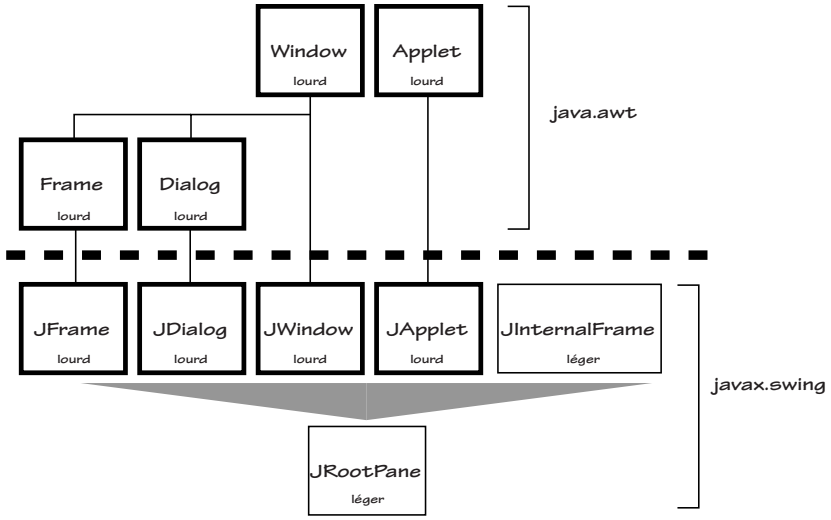
Les fenêtres

L'élément le plus important dans une interface utilisateur à fenêtre est... la fenêtre. Il y a plusieurs sortes de fenêtres : celle créée par le système pour le programme Java, et les autres, créées par Java à l'intérieur de la première. La fenêtre créée par le système est obligatoirement un composant lourd.

Java dispose de deux composants principaux de ce type : **Window** et **Applet**. Le premier est utilisé pour les applications fonctionnant localement. Le second est employé pour les programmes fonctionnant par l'intermédiaire d'un réseau et auxquels de nombreuses restrictions sont imposées pour des raisons de sécurité.

Hiérarchie des fenêtres Java

La figure suivante montre la hiérarchie des fenêtres Java :



Outre les classes **Window** et **Applet** que nous avons déjà mentionnées, il existe deux autres types de fenêtres implémentés sous forme de composants lourds et dérivés de la classe **Window** : **Frame** et **Dialog**. **Dialog** sert à produire les fenêtres de type "boîte de dialogue", qui ont un comportement particulier. **Frame** est une classe servant à créer des fenêtres pourvues d'une barre de titre, de bordures redimensionnables, et des contrôles habituels tels les cases de fermeture et d'agrandissement, ou le menu système. Les instances de la classe **Window** sont simplement des rectangles uniformes sans aucun élément. Tous ces composants appartiennent au package **java.awt**, auquel appartiennent également les versions lourdes des autres composants tels que boutons, menus déroulants, etc.

A chacune des quatre classes du package **java.awt** correspond une classe du package **javax.swing**, portant le même nom précédé d'un *J*. Il s'agit également de composants lourds. Ce sont d'ailleurs les seuls de ce package, qui contient tous les composants nécessaires pour créer une interface. Ce package est généralement désigné par les termes de *Bibliothèque Swing*, ou *Package Swing*. Il existe par ailleurs un cinquième type de fenêtre, **JInternalFrame**, qui est un composant léger, et dont les instances sont des fenêtres appartenant à d'autres fenêtres.

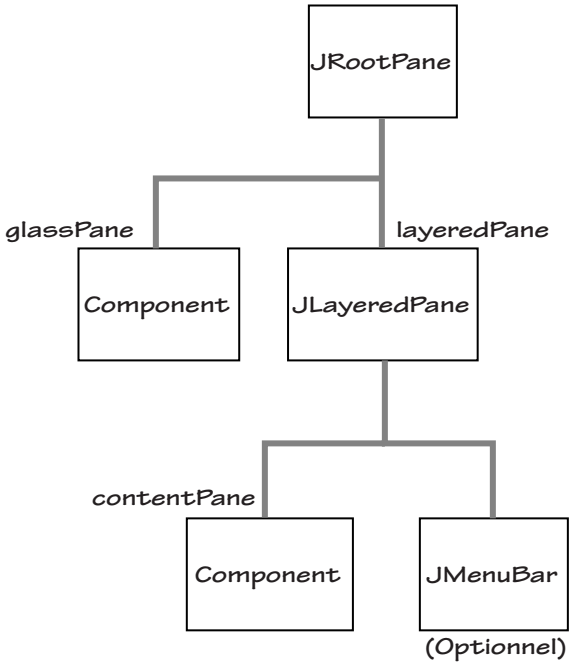
Toutes ces fenêtres ont certaines caractéristiques en commun. En particulier, elles possèdent un composant de type **JRootPane**. C'est ce composant qui recevra tous les éléments constituant le contenu de la fenêtre.

Structure d'une fenêtre

La figure de la page ci-contre montre la structure d'une fenêtre. Notez bien qu'il ne s'agit plus ici d'héritage, comme dans la figure précédente, mais de composition : la classe **JFrame**, par exemple, hérite de la classe **Frame** qui hérite elle-même de la classe **Window**. En revanche, la classe **JFrame** possède un élément de type **JRootPane**. La composition est représentée sur les figures par un filet gras grisé et l'héritage par un filet noir.

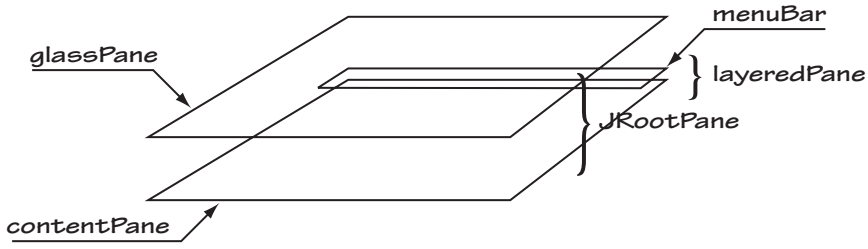
La **JRootPane** contient une *glassPane* et une *layeredPane*. **JRootPane** correspond à une classe de **com.java.swing**. Il s'agit donc d'un type de composant particulier. *layeredPane* et *glassPane* sont des concepts implémen-

tés par la classe **JLayeredPane** pour le premier, et **Component** pour le second. La *glassPane* peut donc être n'importe quel type de composant (instance de la classe **Component** ou d'une de ses classes dérivées).

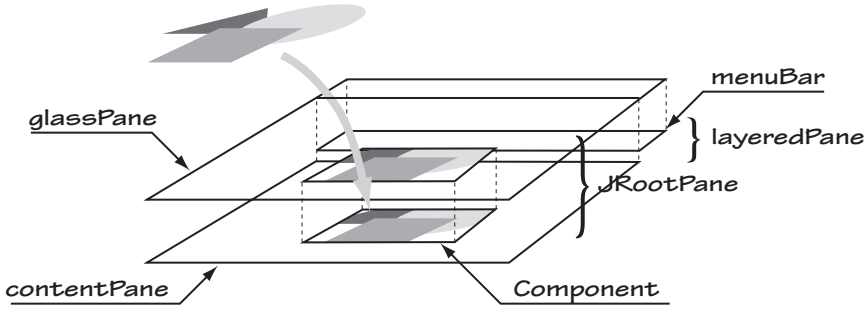


La *layeredPane* contient à son tour une *contentPane*, qui peut être un composant quelconque, et une barre de menus optionnelle, instance de la classe **JMenuBar**.

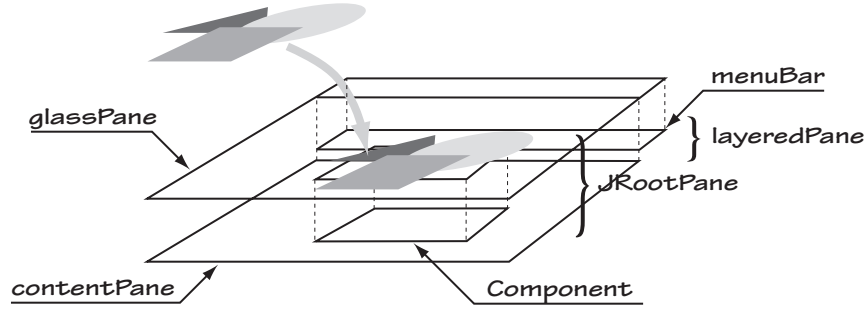
Ces éléments sont disposés de la façon indiquée dans la figure suivante :



La *glassPane* se trouve au-dessus de tous les autres éléments. C'est elle qui reçoit les clic de souris. S'agissant d'un composant quelconque, il est possible de la choisir de façon à pouvoir y afficher des objets divers ou des images, ou encore y dessiner. Elle est particulièrement utile si on souhaite placer un objet ou un dessin au-dessus d'un autre composant sans être limité par les bordures de celui-ci. Dans l'illustration suivante, un dessin est placé dans un composant situé sur la *contentPane*. Il est donc limité par la taille de ce composant :



Au contraire, l'illustration suivante montre le même dessin placé sur la *glassPane,* il peut déborder du cadre du composant placé sur la *contentPane* :



Par défaut, la *glassPane* n'est pas visible. Elle doit être rendue visible explicitement si nécessaire. Ses limites sont identiques à celles de la **JRootPane**.

Les limites de la *layeredPane* sont également identiques à celles de la **JRootPane**. Elle peut contenir un nombre quelconque de composants organisés en couches. Un (et un seul) de ces composants peut être une barre de menus. Il est placé en haut de la *layeredPane*.

La *contentPane* occupe toute la surface de la *layeredPane* moins, éventuellement, la surface occupée par la barre de menus.

Les layout managers

Tous les composants qui peuvent contenir d'autres composants font appel à un *layout manager* pour organiser la disposition des composants. La disposition qui vient d'être décrite est contrôlée par le *layout manager* de la **JRootPane**. Le programmeur qui souhaite organiser la disposition des éléments de l'interface doit éventuellement modifier le *layout manager* de la *contentPane* et non celui de la **JRootPane**. Java propose plusieurs *layout managers* en standard pour la *contentPane*. Vous pouvez en créer d'autres. En revanche, il n'existe qu'un seul *layout manager* pour la **JRootPane**. Si vous ne souhaitez pas l'utiliser, vous devrez obligatoirement en créer un nouveau (si vous souhaitez, par exemple, une barre de menus au bas de l'écran). Nous n'aborderons pas ici la programmation des *layout managers*. Nous montrerons cependant un moyen de s'affranchir de ces outils parfois pesants.

Créer une application fenêtrée

Le programme suivant montre la façon la plus simple de créer une application fenêtrée :

```
import javax.swing.*;
public class Window1 {
   static JFrame maFenêtre;
     public static void main( String[] args ) {
   maFenêtre = new JFrame("Ma fenêtre");
     }
}
```

Si vous compilez et exécutez ce programme, vous ne verrez rien du tout. En effet, vous avez créé une instance de la classe **JFrame**, mais, par défaut, ces instances ne sont pas visibles. Vous remarquerez, en revanche, que si votre programme semble ne rien faire, il ne rend pas la main pour autant. En effet, un thread est en fonctionnement, qui attend les événements qui pourraient se produire pour cette fenêtre. La seule façon de l'interrompre proprement est d'intercepter un de ces événements et de l'associer à une instruction arrêtant le programme. (En attendant, vous pouvez taper *Ctrl+C*.) En premier lieu, il faut que la fenêtre soit visible. Pour cela, il nous faut utiliser la méthode **setVisible**. D'autre part, nous donnerons à la fenêtre une position et une taille plus pratiques, grâce à la méthode **setBounds** qui permet de déterminer les coordonnées de l'angle supérieur gauche de la fenêtre, ainsi que ses dimensions :

```
import javax.swing.*;
public class Window1 {
   static JFrame maFenêtre;
    public static void main( String[] args ) {
   maFenêtre = new JFrame("Ma fenêtre");
   maFenêtre.setBounds(100, 100, 300, 200);
   maFenêtre.setVisible(true);
    }
}
```

Si vous lancez le programme ainsi modifié, vous obtenez une fenêtre de dimensions 300 x 200. Vous pouvez agrandir et manipuler cette fenêtre comme n'importe quelle autre fenêtre d'application. En effet, les événements reçus par la fenêtre sont traités automatiquement en ce qui concerne la fenêtre elle-même. En revanche, ils ne produisent aucun autre effet. En particulier, cliquer dans la case de fermeture (ou, sous Windows, sélectionner *Fermer* dans le menu *Système*) entraîne bien la fermeture de la fenêtre, mais ne termine pas le programme pour autant. Pour obtenir cet effet, il nous faut intercepter l'événement et lui associer une instruction permettant d'obtenir le résultat souhaité.

Quel événement intercepter ?

Il pourrait sembler évident que l'événement à intercepter soit un clic de souris. Il faudrait alors détecter l'endroit où l'utilisateur a cliqué et, s'il s'agit de la case de fermeture, quitter le système. Cette façon de procéder comporte de nombreux inconvénients :

- Tout d'abord, elle est impossible car la case de fermeture n'est pas un objet Java. Elle fait partie des attributs de la fenêtre et nous n'y avons pas accès. Cette raison, à elle seule, est évidemment suffisante, mais il y en a d'autres.

- S'il était possible d'intercepter le clic dans la fenêtre, il faudrait traiter tous les cas possibles, c'est-à-dire ceux où le clic a lieu en dehors de la case de fermeture.

- Cette solution ne fonctionnerait pas lorsque l'utilisateur sélectionne la commande du menu système, ni son équivalent clavier (*Alt + F4* sous Windows).

Pour ces raisons, il est préférable de laisser le système s'occuper de la fermeture de la fenêtre. En revanche, nous pouvons intercepter l'événement *fermeture de la fenêtre*. De cette façon, tous les contrôles standard continueront de fonctionner sans que nous ayons à nous en préoccuper.

Attention : Le comportement décrit ici pour une instance de **JFrame** est spécifique. Comme de nombreux autres composants, **JFrame** a un équivalent dans le package **java.awt**, nommé **Frame**. Si vous êtes amené à utiliser cette classe, vous verrez que le fait de cliquer dans la case de fermeture d'une instance de **Frame** provoque bien l'invocation de **windowClosing** mais ne ferme pas la fenêtre. La raison à cela est que **JFrame** dérive de **Frame** mais définit la méthode **setDefaultCloseOperation(int operation)** qui détermine son comportement lorsque l'on clique dans sa case de fermeture. Les paramètres possibles sont définis dans l'interface **WindowConstants**, implémentée par **JFrame** :

- **DISPOSE_ON_CLOSE** entraîne la fermeture de la fenêtre.

- **HIDE_ON_CLOSE** entraîne le masquage de la fenêtre.

- **DO_NOTHING_ON_CLOSE** ne fait rien, ce qui donne à une **JFrame** le même comportement qu'une **Frame**.

La valeur par défaut est **HIDE_ON_CLOSE**.

Intercepter les événements de fenêtre

Nous avons dit maintes fois qu'en Java, tout était objet. Les événements sont donc des objets. L'événement "fermeture de la fenêtre" est un objet qui est "envoyé", un peu comme l'étaient les exceptions (un peu seulement). Chaque fois qu'une fenêtre est fermée, activée, icônifiée, etc., un événement est créé et envoyé. Chaque fois que la souris est déplacée ou qu'un de ses boutons est cliqué, un événement est créé et envoyé. Chaque fois qu'un article de menu est sélectionné, qu'une case est cochée, qu'une bande de défilement est actionnée, que le texte d'une zone de texte est modifié, un événement est créé et envoyé.

Les événements ainsi envoyés doivent être reçus pour être traités. C'est la fonction des *listeners*, ce qui signifie littéralement "écouteurs". Pour que l'événement correspondant à la fermeture d'une fenêtre soit reçu (afin d'être traité), il faut que cette fenêtre possède un *listener* correspondant à ce type d'événement. Il existe un type de *listener* pour chaque type d'événement. Un *listener* est simplement un objet qui contient du code qui est exécuté lorsque l'objet reçoit un événement.

Les événements et les listeners

Les événements sont des instances de classes Java dont les noms se terminent par **Event**. Ces classes sont réparties dans deux packages :

- **java.awt.event**
- **javax.swing.event**

La raison de cette répartition en deux packages est purement historique. La version 1.1 de Java ne possédait que le package **java.awt.event**. Lorsque

les composants *Swing* ont été ajoutés (soit comme une extension de la version 1.1, soit comme partie intégrante de la version 2), il a fallu créer les événements correspondant aux nouveaux types de contrôles. En revanche, un bouton *Swing* envoie le même événement que les anciens boutons du package **java.awt**, qui sont d'ailleurs toujours présents. (Les anciens boutons sont des composants lourds, alors que les composants *Swing* sont tous des composants légers, à l'exception des quatre que nous avons mentionnés précédemment.)

Si vous consultez la documentation du JDK, vous constaterez que le package **java.awt.event** contient quatorze événements :

- **ActionEvent**
- **AdjustmentEvent**
- **ComponentEvent**
- **ContainerEvent**
- **FocusEvent**
- **InputEvent**
- **InputMethodEvent**
- **InvocationEvent**
- **ItemEvent**
- **KeyEvent**
- **MouseEvent**
- **PaintEvent**
- **TextEvent**
- **WindowEvent**

et treize *listeners* :

- **ActionListener**
- **AdjustmentListener**
- **ComponentListener**
- **ContainerListener**
- **EventQueueListener**
- **FocusListener**
- **InputMethodListener**
- **ItemListener**
- **KeyListener**
- **MouseListener**
- **MouseMotionListener**
- **TextListener**
- **WindowListener**

De son côté, le package **javax.swing.event** contient vingt événements :

- **DocumentEvent**
- **DocumentEvent**.**ElementChange**
- **AncestorEvent**
- **CaretEvent**
- **ChangeEvent**

- **EventListenerList**
- **HyperlinkEvent**
- **InternalFrameEvent**
- **ListDataEvent**
- **ListSelectionEvent**
- **MenuDragMouseEvent**
- **MenuEvent**
- **MenuKeyEvent**
- **PopupMenuEvent**
- **TableColumnModelEvent**
- **TableModelEvent**
- **TreeExpansionEvent**
- **TreeModelEvent**
- **TreeSelectionEvent**
- **UndoableEditEvent**

et vingt et un *listeners* :

- **AncestorListener**
- **CaretListener**
- **CellEditorListener**
- **ChangeListener**

- **DocumentListener**
- **HyperlinkListener**
- **InternalFrameListener**
- **ListDataListener**
- **ListSelectionListener**
- **MenuDragMouseListener**
- **MenuKeyListener**
- **MenuListener**
- **MouseInputListener**
- **PopupMenuListener**
- **TableColumnModelListener**
- **TableModelListener**
- **TreeExpansionListener**
- **TreeModelListener**
- **TreeSelectionListener**
- **TreeWillExpandListener**
- **UndoableEditListener**

Nous souhaitons traiter la fermeture d'une fenêtre. Nous nous intéresserons aux événements de type **WindowEvent** et aux **WindowListener**.

En consultant la documentation, nous constatons que **WindowListener** est une interface. Cela a été conçu ainsi pour permettre à un objet quelconque

d'être un **WindowListener**. Il est ainsi devenu d'usage courant qu'une classe comportant une fenêtre soit également **WindowListener**. La fenêtre peut même être son propre **WindowListener**. Cette pratique n'est pourtant pas très élégante et il semble nettement préférable d'utiliser un objet spécifiquement conçu pour cela. Il y a en tout cas une excellente raison à cela. L'interface **WindowListener** possède en effet sept méthodes. Si une classe implémente cette interface, elle doit redéfinir les sept méthodes (sous peine d'être implicitement **abstract**). En revanche, Java dispose également d'une classe appelée **WindowAdapter** qui implémente l'interface **WindowListener** et redéfinit les six méthodes comme ne faisant rien. En étendant cette classe, nous n'avons à redéfinir que la méthode qui nous intéresse.

A titre informatif, nous montrerons quatre façons de résoudre le problème. Dans le premier cas, la fenêtre sera son propre **WindowListener**. Dans le deuxième, la classe contenant la méthode **main** sera un **WindowListener**. Dans le troisième cas, nous utiliserons une classe que nous créerons spécialement pour cette fonction. Enfin, le quatrième exemple utilisera une classe interne.

```
import javax.swing.*;
import java.awt.event.*;

public class Window1 {
   static MaJFrame maFenêtre;
   public static void main( String[] args ) {
      maFenêtre = new MaJFrame("Ma fenêtre");
      maFenêtre.addWindowListener(maFenêtre);
   maFenêtre.setBounds(100, 100, 300, 200);
      maFenêtre.setVisible(true);
   }
}

class MaJFrame extends JFrame implements WindowListener {
   MaJFrame(String s) {
      super(s);
   }
```

```
    public void windowClosing(WindowEvent e) {
       System.exit(0);
    }
    public void windowActivated(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowOpened(WindowEvent e) {}
}
```

Dans ce programme, les seules modifications apportées à la méthode **main** sont le remplacement de **JFrame** par **MaJFrame** et l'ajout de la ligne :

```
maFenêtre.addWindowListener(maFenêtre);
```

La méthode **addWindowListener** prend pour argument un objet de type **WindowListener** et l'attribue à l'objet sur lequel la méthode est invoquée. Ici, l'objet **WindowListener** se trouve être la fenêtre elle-même, ce qui ne pose pas de problème. Bien entendu, il a fallu remplacer le type **JFrame** par une nouvelle classe étendant **JFrame** et implémentant l'interface **WindowListener**.

La classe **MaJFrame**, pour implémenter l'interface **WindowListener**, doit redéfinir les sept méthodes. Seule la méthode **windowClosing** est utilisée et contient donc une définition.

Lorsque l'on ferme la fenêtre par un moyen quelconque, un objet de type **WindowEvent** est envoyé à la fenêtre. Si celle-ci possède un **WindowListener**, celui-ci reçoit l'objet et exécute la méthode correspondant au type d'événement. Cela présente un avantage certain : bien que l'événement reçu soit simplement de type **WindowEvent**, vous n'avez pas à déterminer de quel événement exact il s'agit ! La méthode **windowClosing(WindowEvent e)** n'a donc rien à faire du paramètre qui lui est passé ! Elle se contente de terminer le programme à l'aide de l'instruction :

```
System.exit(0);
```

Une autre façon d'écrire ce programme est de faire implémenter l'interface **WindowListener** par la classe qui contient la méthode **main**, ce qui évite d'avoir à créer une classe étendant la classe **JFrame** :

```
import javax.swing.*;
import java.awt.event.*;

public class Window2 implements WindowListener {
   static JFrame maFenêtre;
   public static void main( String[] args ) {
      maFenêtre = new JFrame("Ma fenêtre");
      maFenêtre.addWindowListener(new Window2());
   maFenêtre.setBounds(100, 100, 300, 200);
      maFenêtre.setVisible(true);
   }
   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }
   public void windowActivated(WindowEvent e) {}
   public void windowClosed(WindowEvent e) {}
   public void windowDeactivated(WindowEvent e) {}
   public void windowDeiconified(WindowEvent e) {}
   public void windowIconified(WindowEvent e) {}
   public void windowOpened(WindowEvent e) {}
}
```

Ici, c'est la classe **Window2** qui implémente **WindowListener** et qui doit donc redéfinir les sept méthodes. La méthode **addWindowListener** doit prendre pour paramètre une instance de la classe **Window2**. La méthode **main** étant statique, il faut instancier spécialement cette classe. Nous verrons plus loin qu'avec les applets, dans lesquelles la fonction de la méthode **main** est tenue par une méthode non statique appelée **init**, on peut faire référence à l'instance qui contient la fenêtre, sous la forme :

```
maFenêtre.addWindowListener(this);
```

Les deux exemples précédents ont l'inconvénient de nous obliger à redéfinir toutes les méthodes de l'interface **WindowListener**. L'exemple suivant utilise une classe créée spécifiquement pour servir de listener. L'avantage est qu'elle peut étendre la classe **WindowAdapter**, qui contient déjà une redéfinition vide des sept méthodes. Notre code sera ainsi plus concis :

```
import javax.swing.*;
import java.awt.event.*;

public class Window3 {
   static JFrame maFenêtre;
   public static void main( String[] args ) {
      maFenêtre = new JFrame("Ma fenêtre");
      maFenêtre.addWindowListener(new MonAdapter());
   maFenêtre.setBounds(100, 100, 300, 200);
      maFenêtre.setVisible(true);
   }
}

class MonAdapter extends WindowAdapter {
   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }
}
```

L'inconvénient de ce code est qu'il est peu probable que la classe **monAdapter** serve à autre chose. En effet, si l'application comporte d'autres fenêtres, il s'agira de fenêtres filles de la fenêtre principale et il est peu probable que le fait de les fermer doivent entraîner l'arrêt du programme. De la même façon, un listener conçu pour un bouton exécutera une action spécifique à ce bouton et ne sera donc instancié qu'une seule fois. Si une application comporte une trentaine de contrôles différents, il faudra créer une trentaine de classes toutes instanciées une seule fois. De plus, ici, le programme principal est très court. Dans un cas réel, il pourrait être beaucoup plus long. Pour savoir ce que fait un listener, il faudrait alors se reporter plusieurs pages en avant ou en arrière dans le listing. Cela n'améliore pas la lisibilité. De plus, il est fréquent, pour ne pas dire plus, que le listener

doive dialoguer avec l'objet qui contient le programme principal. Ici, le programme étant une méthode statique, ce n'est pas très difficile. En revanche, c'est moins pratique avec les applets. Une solution élégante à ces problèmes consiste à utiliser une classe interne. Cependant, nous avons toujours un problème lié au fait que la méthode **main** est statique. Il n'est pas possible de créer une instance de classe interne sans une référence implicite à une instance de la classe externe qui la contient. La solution pourrait consister à créer une instance de la classe **Window4** spécifiquement pour pouvoir instancier la classe interne, ce qui donne le listing suivant :

```
import javax.swing.*;
import java.awt.event.*;

public class Window4 {
   static JFrame maFenêtre;
   public static void main( String[] args ) {
      maFenêtre = new JFrame("Ma fenêtre");
      maFenêtre.addWindowListener((new Window4()).new MonAdapter());
   maFenêtre.setBounds(100, 100, 300, 200);
      maFenêtre.setVisible(true);
   }

   class MonAdapter extends WindowAdapter {
      public void windowClosing(WindowEvent e) {
         System.exit(0);
      }
   }
}
```

Si vous ne comprenez pas parfaitement la syntaxe de la ligne imprimée en gras, vous devriez peut-être réviser le Chapitre 12 ! Plaisanterie mise à part, ce n'est ni très lisible, ni très élégant. On pourrait bien sûr simplifier en déclarant statique la classe interne :

```
import javax.swing.*;
import java.awt.event.*;
```

```
public class Window5 {
   static JFrame maFenêtre;
   public static void main( String[] args ) {
      maFenêtre = new JFrame("Ma fenêtre");
      maFenêtre.addWindowListener(new MonAdapter());
   maFenêtre.setBounds(100, 100, 300, 200);
      maFenêtre.setVisible(true);
   }

   static class MonAdapter extends WindowAdapter {
      public void windowClosing(WindowEvent e) {
         System.exit(0);
      }
   }
}
```

Une solution beaucoup plus élégante consiste à utiliser une classe anonyme. De cette façon, nous n'avons plus besoin de faire référence à la classe externe et le code du *listener* se trouve vraiment à l'endroit où il est utilisé. Cette façon de procéder est de loin la plus adaptée à la conception du modèle d'événements de Java. C'est celle que nous utiliserons chaque fois que possible dans les exemples de la suite de ce livre :

```
import javax.swing.*;
import java.awt.event.*;
public class Window6 {
   static JFrame maFenêtre;
   public static void main( String[] args ) {
      maFenêtre = new JFrame("Ma fenêtre");
      maFenêtre.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                   System.exit(0);
                }});
   maFenêtre.setBounds(100, 100, 300, 200);
      maFenêtre.setVisible(true);
   }
}
```

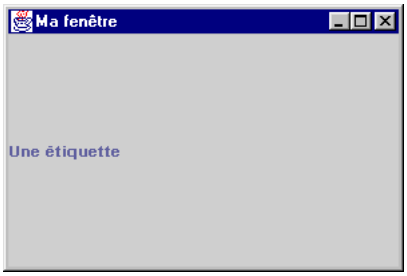
Utilisation des composants

Un des composants les plus simples est l'étiquette. Les étiquettes permettent simplement d'afficher du texte non modifiable par l'utilisateur. Dans l'exemple suivant, vous noterez que l'étiquette n'est pas ajoutée à la fenêtre, mais à sa *contentPane*, obtenue au moyen de la méthode **getContentPane()** :

```
import javax.swing.*;
import java.awt.event.*;

public class Window7 extends JFrame {
   static Window7 maFenêtre;

   Window7(String s) {
      super(s);
      addWindowListener(new WindowAdapter() {
              public void windowClosing(WindowEvent e) {
                 System.exit(0);
              }});
      setBounds(100, 100, 300, 200);
      getContentPane().add(new JLabel("Une étiquette"));
      setVisible(true);
   }

   public static void main( String[] args ) {
      maFenêtre = new Window7("Ma fenêtre");
   }
}
```

La figure de la page suivante montre la fenêtre affichée par ce programme.

Dans ce programme, nous avons utilisé une structure un peu différente. La classe principale étend la classe **JFrame**. La fenêtre est configurée dans le constructeur de la classe principale. L'avantage est que nous n'avons plus besoin d'utiliser le handle de la fenêtre pour y faire référence. Nous pouvons donc écrire :




```
setBounds(100, 100, 300, 200);
```

au lieu de :

```
maFenêtre.setBounds(100, 100, 300, 200);
```

La méthode **main** est ainsi réduite au minimum qui lui appartient, la création d'une instance de **Window7**. (On pourrait évidemment discuter pour savoir si l'instruction **setVisible(true)** appartient plus logiquement à la méthode **main** ou au constructeur de la fenêtre. Il nous semble qu'il est plus fréquent de créer des fenêtres visibles que des fenêtres invisibles et qu'il est donc préférable que la fenêtre se rende visible elle-même.)

Dans la plupart des cas, vous utiliserez une technique à mi-chemin entre les deux derniers exemples. La fenêtre aura une bonne dose de self-control, mais elle ne sera pas implémentée dans la classe principale. L'exemple suivant, qui affiche deux étiquettes, montre cette technique :

```
import javax.swing.*;
import java.awt.event.*;

public class Window8 {
   static FenetrePrincipale maFenêtre;

   public static void main( String[] args ) {
      maFenêtre = new FenetrePrincipale("Ma fenêtre");
   }
```

```
   static class FenetrePrincipale extends JFrame {
      FenetrePrincipale(String s) {
         super(s);
         addWindowListener(new WindowAdapter() {
                 public void windowClosing(WindowEvent e) {
                    System.exit(0);
                 }});
         setBounds(100, 100, 300, 200);
         getContentPane().add(new JLabel("Une étiquette"));
         getContentPane().add(new JLabel("Une autre étiquette"));
         setVisible(true);
      }
   }
}
```

Indépendamment de l'utilisation d'une classe interne statique, ce programme a une particularité. En effet, si vous le compilez et l'exécutez, vous constaterez qu'il n'affiche que la deuxième étiquette. Pour comprendre ce qui se passe, essayez de modifier les dimensions de la fenêtre. Vous constaterez que l'étiquette est toujours placée à gauche, au milieu de la hauteur de la fenêtre.

La raison à ce comportement étrange tient dans l'utilisation systématique par Java des *layout managers*.

Utilisation des layout managers

Avec un langage classique (normal ?), une étiquette ainsi affichée serait probablement positionnée dans l'angle supérieur gauche de la fenêtre. L'équivalent s'était d'ailleurs produit lorsque nous n'avions pas spécifié la position de la fenêtre : elle avait été affichée dans l'angle supérieur gauche de l'écran. A l'intérieur des fenêtres, tout est différent. Un composant ajouté à un conteneur (une fenêtre, par exemple) est passé au *layout manager* de ce conteneur. Le *layout manager* décide de la position de chaque composant.

Philosophie des layout managers

La philosophie qui sous-tend l'utilisation des *layout managers* est tout à fait respectable. Il en va des layout managers comme de certaines utopies philosophiques ou politiques (nous ne nommerons personne pour ne pas nous faire trop d'ennemis, mais chacun reconnaîtra les siens) : une bonne intention qui apporte beaucoup de désillusions. L'idée est que les programmes doivent fonctionner sur tous les systèmes existants (c'est déjà ambitieux) et à venir (là est l'utopie). Il faut donc que la disposition des éléments dans une fenêtre soit la même (fonctionnellement la même, ce qui ne signifie pas forcément identique physiquement), quel que soit l'affichage utilisé. Or, les divers systèmes d'affichage diffèrent par :

- leur taille,
- leur résolution,
- le nombre de couleurs,
- les ressources système disponibles.

Vous vous attendiez peut-être à ne trouver que les trois premiers éléments, suivant en cela les affirmations des bons apôtres de Java qui clament haut et fort que ce langage doit permettre de construire une interface indépendante des ressources du système (du moins à partir du moment où le système dispose d'un affichage graphique). Malheureusement, pour les interfaces utilisateur, Java fait un usage intensif de certaines ressources fournies par le système : les polices de caractères. Pour simplifier le problème au maximum, les concepteurs de Java proposent de n'utiliser que des polices génériques correspondant au minimum vital : police avec et sans empattement, police à espacement fixe et police de symbole. Cependant, même si vous vous limitez à ces polices (par exemple en ne vous préoccupant jamais d'en choisir une), vous n'êtes pas à l'abri de problèmes. En effet, les polices génériques sont un concept et n'ont pas d'existence propre. (Cela viendra peut-être mais, pour l'instant, il n'y a pas de polices Java.) Elles sont donc remplacées lors de l'exécution du programme par les polices les plus proches physiquement présentes sur le système (en général, *Times New Roman*, *Arial* et *Courier New* sous Windows, *Times*, *Helvetica* et *Courier*

sous Motif et MacOs). Les polices de caractères pouvant être de tailles différentes, cela peut poser un problème sur certains systèmes.

Le problème de la taille et de la résolution de l'écran est encore plus crucial. Si deux ordinateurs sont équipés chacun d'un écran de 17 pouces, l'un affichant avec une résolution de 800 x 600 points et l'autre avec une résolution de 1600 x 1200, une fenêtre devra avoir une définition deux fois supérieure pour avoir la même taille. (La définition est la taille en pixels, ou élément d'image. Un pixel correspond à un point de l'affichage.)

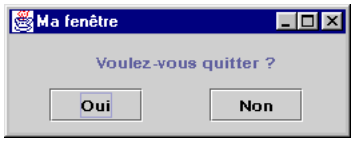
Si l'on souhaite que l'affichage soit identique sur les deux systèmes, il existe deux possibilités. La première consiste à utiliser une disposition fixe des éléments avec un système de coordonnées indépendant du système physique, par exemple des dimensions absolues en centimètres. Cette approche ne résout les problèmes que dans la mesure où les caractéristiques géométriques de l'affichage le permettent. Une boîte de dialogue de 4 centimètres par 3 sera affichée correctement sur les deux écrans de 17 pouces dont nous parlions précédemment. En revanche, une fenêtre de 30 centimètres par 20 ne pourra être affichée sur un écran de 13 pouces. Cette approche est la seule qui permettrait d'atteindre le but recherché. Le programmeur devrait choisir une taille pour chaque boîte de dialogue et une taille pour la police de caractères. Deux centimètres feraient toujours 2 centimètres et une police de 12 points aurait la même taille sur tous les écrans. (Il s'agit ici du point typographique, qui sert à mesurer la taille des polices de caractères.)

La seconde approche possible consiste à ne pas indiquer la position précise des éléments, mais simplement le type de disposition souhaitée. A chaque type de disposition correspond un *layout manager*. Si la disposition dont vous avez besoin est particulière, vous pouvez créer votre propre *layout manager*.

Ce beau principe est cependant en partie inapplicable. Tout d'abord, la volonté d'avoir un affichage indépendant des caractéristiques du système est souvent injustifiable. Si vous concevez une application multimédia consacrée à la peinture, vous n'aurez probablement rien à faire de savoir que votre application ne fonctionne pas sur un écran de 640 x 480 points affichant 16 couleurs. Vous indiquerez simplement aux utilisateurs que la configuration minimale est, par exemple, 800 x 600 en 65 000 couleurs.

Un autre cas où l'utilisation des *layout managers* devrait être un avantage est celui des boîtes de dialogue redimensionnables. Il ne s'agit pas ici de la faculté pour l'utilisateur de modifier la taille de la boîte de dialogue, mais, pour le programmeur, d'afficher par exemple une boîte de dialogue avec un élément centré. C'est exactement ce que fait le **BorderLayout**, qui est le *layout manager* par défaut des **JFrame**. (Curieusement, le *layout manager* par défaut des composants *Swing* est différent de celui de leurs équivalents non-*Swing*. Sans doute les concepteurs de Java se sont-ils rendu compte du peu d'efficacité de la solution choisie et essaient par là de corriger le tir.)

Voyons comment un programmeur traditionnel résout le problème de l'affichage d'une boîte de dialogue comportant un message (une question, par exemple) et deux boutons (oui/non). Il peut créer une fenêtre de taille fixe, calculer, en fonction de la longueur du message, la position de l'étiquette qui le contient, puis placer les deux boutons au-dessous du message à égale distance de chaque bord, par exemple :

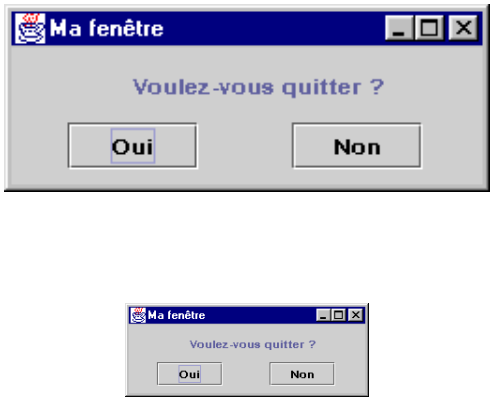


L'obtention d'un tel résultat passe par un certain nombre de tâtonnements. Il est obtenu en utilisant les coordonnées suivantes :

- Fenêtre : **setBounds(100, 100, 260, 100)**
- Étiquette : **setBounds(65, 10, 200, 20)**
- Bouton 1 : **setBounds(30, 40, 70, 25)**
- Bouton 2 : **setBounds(150, 40, 70, 25)**

Un problème est que cette boîte de dialogue est totalement impossible à réutiliser avec un autre message En effet, si le message est plus court, il n'est plus centré. S'il est plus long, il dépasse de la fenêtre. Si on agrandit la fenêtre, les boutons ne sont plus centrés.

Un autre problème est que les valeurs indiquées ne sont valables qu'en fonction de la taille des caractères utilisés. Ici, nous avons utilisé la taille standard, c'est-à-dire que nous n'en avons spécifié aucune. La figure suivante montre le résultat obtenu sur un écran de 17 pouces en 800 x 600 (en haut) et en 1600 x 1200 (en bas).



On est loin de l'objectif initial stipulant qu'un programme doit tourner sur tous les systèmes en donnant le même résultat. La seule solution consiste à gérer ce problème nous-mêmes. Malheureusement, les layout managers livrés avec Java ne sont pas d'un grand secours !

La plupart des programmeurs se font leur propre bibliothèque de sous-programmes réutilisables. Pour une boîte de dialogue telle que celle-ci, voici à peu près ce que devrait faire le sous-programme :

- Déterminer la taille de police à utiliser.
- Calculer la longueur physique du message.
- Calculer la largeur de la boîte de dialogue en ajoutant une valeur fixe ou un pourcentage à la longueur du message. Si la valeur obtenue est inférieure à une valeur minimale, prendre la valeur minimale. (On considère que le programmeur choisit des messages ne dépassant jamais la valeur maximale.)

- Calculer la taille de l'étiquette (longueur du message + marge, hauteur fixe).

- Calculer la position de l'étiquette ((largeur de la fenêtre - longueur du message) / 2).

- Calculer la taille et la position des boutons (en fonction de la taille des caractères et de la taille de la boîte de dialogue, elle-même déterminée par la longueur du message).

Il est parfaitement possible d'écrire un sous-programme qui fonctionne dans tous les cas usuels, mais un certain nombre de problèmes peuvent survenir :

- La police utilisée pour afficher le message est plus haute que prévu. Le message est tronqué en hauteur. Ce problème est facile à prévoir mais cela complique encore le sous-programme.

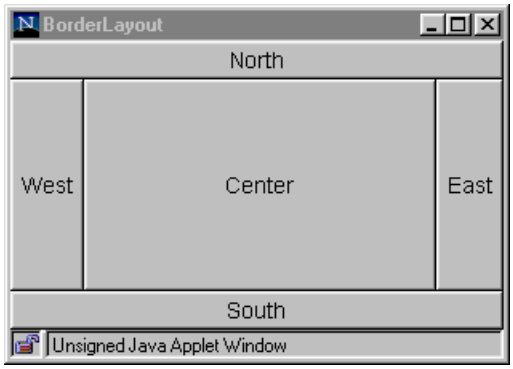
- L'écran de l'utilisateur est d'excellente qualité et utilise une résolution très élevée. Résultat, la boîte de dialogue est minuscule.

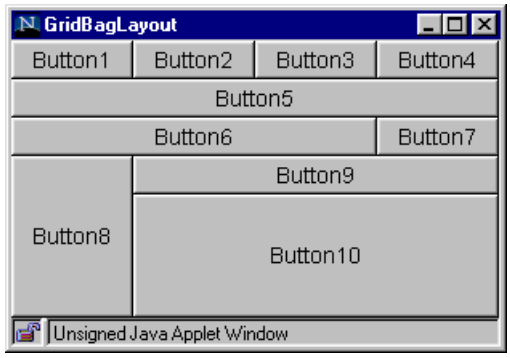
- Le programme doit être traduit dans une langue étrangère. Le message est beaucoup plus long et dépasse la valeur maximale possible pour la largeur de la fenêtre.

- Le programmeur a besoin d'une boîte de dialogue à trois boutons. Il doit écrire un autre sous-programme.

On voit facilement que l'efficacité d'un tel sous-programme dépend de l'intelligence que l'on y met. Il est possible d'écrire un tel sous-programme capable de traiter tous les cas possibles. C'est exactement ce qu'est supposé faire un layout manager. Pour juger de la capacité des layout managers standard, il suffit de regarder les exemples qui sont proposés dans les livres "officiels" sur Java. Vous trouverez ci-contre un exemple de **BorderLayout** et un exemple de **GridBagLayout**.

Si vous pensez vraiment avoir un jour besoin de l'une ou l'autre de ces boîtes de dialogue, n'hésitez pas. Il est probable que leurs auteurs ne vous demanderont pas de royalties pour vous permettre d'utiliser leur code source.





Bien utiliser les layout managers

Peut-on utiliser quand même les layout managers ? Un layout manager est un objet doté d'une certaine intelligence lui permettant de disposer au mieux des composants dans un conteneur. La tâche qui revient au programmeur (ou plutôt au concepteur de l'interface du programme, qui, sur les petits projets, est souvent la même personne) est de modéliser le contenu du conteneur en utilisant les termes d'un layout manager existant.

Une question très importante est la suivante : le même layout manager doit-il disposer *tous* les composants d'un conteneur ? La réponse est oui, car cela est imposé par Java. Un même conteneur ne peut posséder qu'un seul layout manager. Ayant constaté cela, certains programmeurs tentent désespérément de créer un modèle de disposition exprimable, dans le meilleur des

cas à l'aide du layout manager qu'ils estiment le plus adapté et, dans le pire des cas, à l'aide de celui qu'ils connaissent le mieux.

Pour bien utiliser les layout managers, il faut comprendre qu'ils offrent chacun la traduction d'un concept essentiel en matière de disposition des composants :

- Le **FlowLayout** implémente le concept d'une séquence horizontale dont les paramètres essentiels sont :
    - L'alignement (à droite, à gauche, centré ou justifié).
    - L'espacement horizontal séparant les composants.
    - L'espacement vertical séparant les différentes lignes de composants.
- Le **GridLayout** implémente le concept d'une disposition en tableau dont les paramètres essentiels sont :
    - Le nombre de colonnes.
    - Le nombre de lignes.
    - L'espace entre les colonnes.
    - L'espace entre les lignes.
- Le **ViewportLayout** implémente le concept d'une disposition d'un seul composant, aligné en bas de la surface disponible tant que celle-ci est inférieure à la taille du composant et en haut lorsqu'elle est supérieure.
- Le **ScrollPaneLayout** implémente le concept d'un composant pouvant défiler horizontalement et verticalement dans la surface d'affichage.
- Le **BorderLayout** divise la zone d'affichage en cinq zones : Nord, Sud, Est, Ouest et Centre. Ces principaux paramètres sont :
    - L'espacement horizontal entre les composants.

    - L'espacement vertical entre les composants.

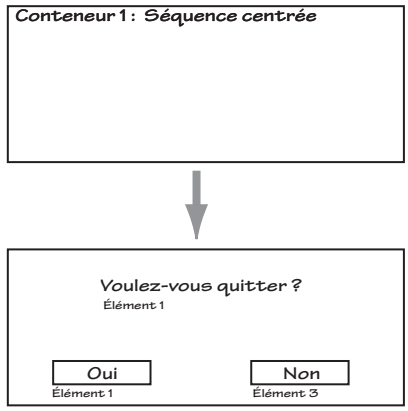
- Le **BoxLayout** implémente un concept proche de celui du **GridLayout** mais avec une seule dimension au lieu de deux. Il utilise un seul paramètre :

    - L'alignement (vertical ou horizontal).

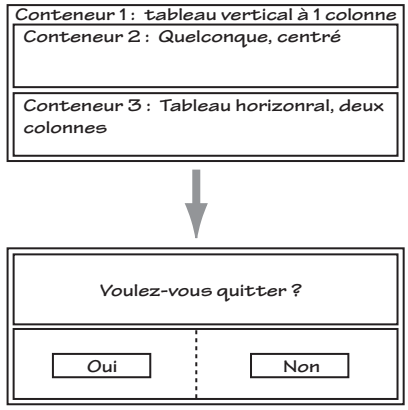
- Le **OverlayLayout** correspond à l'empilement des composants les uns au-dessus des autres.

Les autres layout managers ne sont pas fondamentaux (au sens propre du terme) en ce qu'ils expriment un concept complexe et non un concept de base. Ainsi, le **CardLayout** exprime le concept d'un arrangement en pages (sorte d'empilement fonctionnel). Le **GridBagLayout** tente d'exprimer un concept universel. Quant au **JRootPane.RootLayout**, il est l'exemple d'une implémentation complexe parfaitement fonctionnelle dont la tâche est de gérer la disposition de la *layeredPane*, de la *glassPane* et de la barre de menus d'une **JRootPane**.

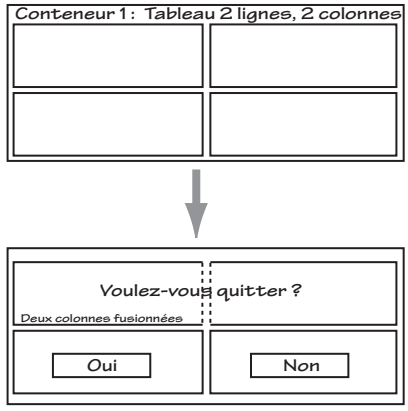
En fait, il nous faut exprimer le résultat souhaité en termes de trois fondamentaux : séquence, tableau, bordures. Pour chaque cas, l'analyse peut amener à plusieurs expressions équivalentes. Ainsi, notre boîte de dialogue précédente peut être exprimée à l'aide d'une seule séquence :



En assemblant les composants en plusieurs sous-groupes, on pourra parvenir à la structure suivante :

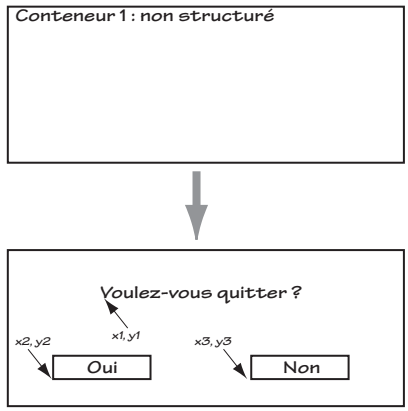


Il est également possible d'arriver au même résultat avec une structure à un seul niveau, mais plus complexe :



Enfin, il est possible de positionner précisément chaque élément en indiquant ses coordonnées *x*,*y*. Ces coordonnées peuvent avoir été calculées par

le programmeur et intégrées dans le programme (*hardcoded*, comme on dit en américain). On utilise alors aucun layout manager. Elles peuvent aussi être calculées au moment voulu par un layout manager personnalisé :



Nous allons donner un exemple de réalisation de chacun de ces cas.

Utilisation du FlowLayout

Le premier exemple utilisera le layout manager **FlowLayout**. Le résultat paraît efficace pour un exemple simple. Pourtant, les limites sont évidentes. Compilez et exécutez ce programme pour voir le résultat obtenu :

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class Flow1 {
   public static void main( String[] args ) {
      FenetrePrincipale maFenêtre =
                     new FenetrePrincipale("Ma fenêtre");
   }
}
```

```
class FenetrePrincipale extends JFrame {
   FenetrePrincipale(String s) {
      super(s);
      addWindowListener(new WindowAdapter() {
              public void windowClosing(WindowEvent e) {
                 System.exit(0);
              }});
      getContentPane().setLayout(new FlowLayout(FlowLayout.CENTER,
                                                          50, 20));
      JLabel L1 = new JLabel("Voulez-vous quitter ?");
      getContentPane().add(L1);
      JButton B1 = new JButton("Oui");
      getContentPane().add(B1);
      JButton B2 = new JButton("Non");
      getContentPane().add(B2);
      setBounds(calculBounds(L1));
      setVisible(true);
      setResizable(false);
   }

   Rectangle calculBounds(JLabel jl) {
      Dimension tailleEcran =
                     Toolkit.getDefaultToolkit().getScreenSize();
      int largeurEcran = tailleEcran.width;
      int hauteurEcran = tailleEcran.height;
      int largeur = (int)((jl.getPreferredSize()).width + 100);
      int hauteur = (int)((jl.getPreferredSize()).height * 7);
      int xPos = (largeurEcran - largeur) / 2;
      int yPos = (hauteurEcran - hauteur) / 2;
      return new Rectangle(xPos, yPos, largeur, hauteur);
   }
}
```

La méthode **calculBounds** est intéressante, car elle montre ce que ne fait pas le layout manager. En effet, celui-ci se contente de disposer les éléments les uns à la suite des autres, comme le ferait un traitement de texte en disposant les mots sur une ligne. Si le mot suivant ne tient pas sur la ligne, il est reporté à la ligne suivante. La ligne :

```
getContentPane().setLayout(new FlowLayout(FlowLayout.CENTER, 50, 20));
```

attribue à la *contentPane* un **FlowLayout** manager utilisant l'alignement centré et laissant un espace de 50 pixels entre les éléments et un espace de 20 pixels entre les lignes.

Pour effectuer sa "mise en page", le layout manager doit connaître les dimensions de la fenêtre. Lors de sa création, celle-ci a des dimensions nulles. Le layout manager agit sur le contenu de la fenêtre, mais il est incapable de modifier la fenêtre elle-même pour l'adapter à son contenu. Pour qu'un tel layout manager soit utilisable, il faudrait lui fournir les contraintes suivantes :

- Passer à la ligne après le premier élément.
- Si la taille des deux boutons diffère de moins d'un certain seuil (20 %, par exemple), donner aux deux boutons la taille du plus grand. Il est en effet inesthétique de juxtaposer deux boutons de tailles proches mais inégales.
- Comparer la longueur des deux lignes et choisir pour taille de la boîte de dialogue la longueur de la plus grande ligne augmentée d'une marge.
- Calculer la hauteur de la boîte de dialogue en fonction de la hauteur des lignes.
- Calculer la hauteur et la largeur de l'écran. Renvoyer une exception si la taille de la boîte de dialogue est supérieure à celle de l'écran.
- Calculer la position de la boîte de dialogue afin qu'elle soit centrée à l'écran.

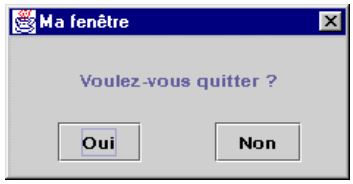
Notez que nous n'avons pas inclus le calcul de la taille de la police de caractères à utiliser. Nous considérerons en effet que la taille standard a été configurée par l'utilisateur du système et convient donc à ses goûts et ses besoins. (Il est toujours mal perçu par l'utilisateur de se voir imposer des choix esthétiques différents de ceux qu'il a effectués pour la configuration de son système.)

La méthode **calculBounds** effectue une partie du travail décrit ici. Pour des raisons de simplicité, nous n'avons pas inclus la vérification de la taille des boutons ni de la longueur respective des lignes. Le calcul de la taille de l'écran se fait à l'aide de l'instruction :

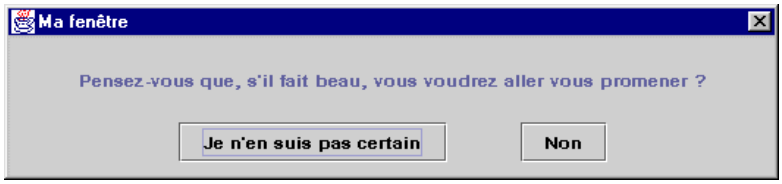
```
Dimension tailleEcran = Toolkit.getDefaultToolkit().getScreenSize();
```

Bien entendu, dans le cas d'une boîte de dialogue d'une application, on pourra prendre une autre base de référence que la taille de l'écran, par exemple la taille de la fenêtre principale.

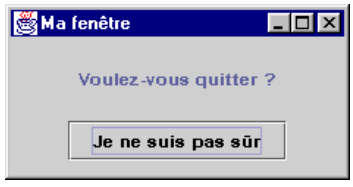
Ce programme affiche le résultat suivant :



Si vous modifiez le programme en remplaçant simplement les trois chaînes de caractères, vous pourrez obtenir automatiquement un résultat semblable au suivant :



En revanche, si le texte des boutons est plus long que celui de l'étiquette, il risque de se poser un problème :



Ici, le deuxième bouton a été repoussé sur une troisième ligne. Il est facile de traiter ce cas, mais on finit par ne plus bien voir ce qu'apporte le layout manager.

Notez que la longueur du texte a été obtenue en appelant la méthode **getPreferredSize()** de l'étiquette :

```
int largeur = (int)((jl.getPreferredSize()).width + 100);
```

(Nous rajoutons 100 pour les marges.) Chaque composant dispose des trois méthodes **getPreferredSize()**, **getMinimuSize()** et **getMaximumSize()** qui renvoient les dimensions idéales, minimales et maximales du composant. (Pour une **JLabel**, ces trois méthodes renvoient les mêmes valeurs.) Le layout manager utilise ces méthodes pour faire sa mise en page. Il interroge ainsi les composants pour savoir jusqu'à quel point il peut les réduire ou les agrandir. Il peut également se dimensionner par lui-même en fonction de son contenu, comme nous le verrons plus loin. Les composants disposent aussi de la méthode **getSize()**, mais celle-ci ne peut être employée ici. En effet, elle renvoie la taille réelle du composant à un moment donné. Avant que le layout manager du conteneur dans lequel se trouve le composant ait effectué son travail, la méthode ne renvoie pas la bonne taille, mais la taille avant la mise en page. Ici, comme il s'agit de la première mise en page, **getSize()** renvoie une taille nulle. Pour que cette méthode renvoie la valeur correcte, il faudrait l'utiliser après que la méthode **setVisible()** du conteneur aura été invoquée. Le résultat serait que la fenêtre s'afficherait brièvement dans l'angle supérieur gauche de l'écran avec des dimensions nulles avant d'être redimensionnée et repositionnée. Sur un système peu rapide, cette opération est visible à l'écran, ce qui ne fait pas très professionnel.

Une autre approche possible consiste à interroger le layout manager pour connaître la dimension idéale de la fenêtre :

```
getLayout().preferredLayoutSize(this)
```

Cette instruction obtient le layout manager au moyen de la méthode **getLayout()**, puis appelle la méthode **preferredLayoutSize()** de celui-ci en lui passant comme argument le conteneur dont on cherche la taille idéale.

Ici, il s'agit de celui depuis lequel la méthode est invoquée et on utilise l'argument **this**. (Cette méthode peut servir pour interroger un layout manager pour savoir quelle taille il donnerait à un autre conteneur si on l'y appliquait.)

Cependant, cette façon de faire ne peut pas être employée ici, car le layout manager indiquerait la taille idéale pour que les trois composants soient sur une seule ligne, soit 451 x 67. Pour que ce procédé donne le résultat voulu, il faudrait qu'il existe un "saut de ligne", ce qui n'est pas le cas avec le **FlowLayout**. Un "espace insécable" serait également utile.

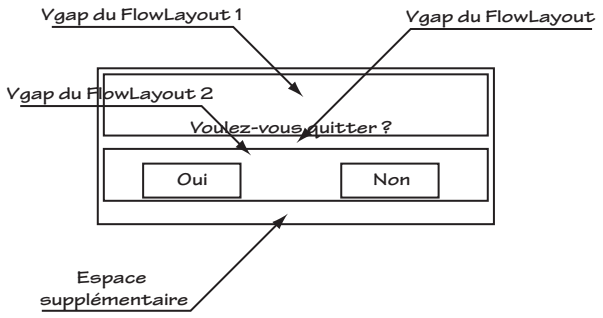
De la même façon, la méthode **pack()** permet de donner automatiquement à la fenêtre la taille "idéale". Pour la même raison, elle n'est d'aucun secours ici.

Utilisation d'un layout à plusieurs niveaux

Les deux fonctions qui nous manquent (saut de ligne et espace insécable) peuvent être implémentées facilement en utilisant une structure à plusieurs niveaux. Pour le saut de ligne, il suffit d'utiliser un tableau à une seule colonne et de placer chaque ligne dans une cellule du tableau. L'espace insécable peut être obtenu en regroupant les deux composants qui ne doivent pas être séparés dans un même sous-conteneur. Ici, le fait d'appliquer la première technique implique automatiquement l'utilisation de la seconde.

Pour mettre en œuvre ces techniques, il nous faut un conteneur spécial. Java dispose d'un tel élément : le *panel*. Un panel est tout simplement un conteneur invisible qui n'a d'autre fonction que de grouper les composants. Bien entendu, les composants peuvent être eux-mêmes des panels contenant des panels, etc. Chaque panel peut utiliser un layout manager différent. Nous pourrions être tentés d'utiliser le **GridLayout**, pour créer le tableau à deux lignes et placer dans chaque cellule un panel muni d'un **FlowLayout**. Cette solution ne convient cependant pas car le contenu des cellules d'un **GridLayout** est aligné en haut et à gauche. De plus, toutes les cellules ont la même taille. Par conséquent, le **FlowLayout** étant aligné en

haut, il nous serait impossible d'obtenir un contenu centré. La figure suivante explique pourquoi :



On voit sur cette figure qu'il existe quatre espaces verticaux :

- Celui ajouté par le premier **FlowLayout** avant l'étiquette :

```
class PanelHaut extends JPanel {
  PanelHaut(String s) {
    setLayout(new FlowLayout(FlowLayout.CENTER, 0, 20));
```

- Celui ajouté entre les cellules par le **GridLayout** :

```
class FenetrePrincipale extends JFrame {
  FenetrePrincipale(String s) {
    super(s);
    getContentPane().setLayout(new GridLayout(2, 1, 0, 5));
```

- Celui ajouté par le second **FlowLayout** avant les boutons :

```
class PanelBas extends JPanel {
  PanelBas(String s1, String s2) {
    setLayout(new FlowLayout(FlowLayout.CENTER, 50, 10));
```

- Celui ajouté par le **GridLayout** après le second panel afin que les deux cellules aient la même taille. (Toutes les cellules ont la même taille et leur contenu est aligné en haut et à gauche.)

On constate donc qu'il n'est pas possible d'aligner les éléments correctement lorsque la première ligne est moins haute que les suivantes. Il n'est pas très logique que la hauteur des cellules soit déterminée par la première ligne. Il serait plus utile que la ligne la plus haute soit prise en compte.

Une solution peut consister à rétablir la situation en imposant au premier panel la hauteur du plus haut. Ici, nous savons que le plus haut sera toujours le second. Le programme suivant utilise cette technique :

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class Grid1 {
   public static void main( String[] args ) {
      FenetrePrincipale maFenêtre =
                           new FenetrePrincipale("Ma fenêtre");
   }
}

class FenetrePrincipale extends JFrame {
   FenetrePrincipale(String s) {
      super(s);
      addWindowListener(new WindowAdapter() {
              public void windowClosing(WindowEvent e) {
                 System.exit(0);
              }});
      getContentPane().setLayout(new GridLayout(2, 1, 0, 15));
      PanelHaut P1 = new PanelHaut("Voulez-vous quitter ?");
      getContentPane().add(P1);
      PanelBas P2 = new PanelBas("Oui", "Non");
      getContentPane().add(P2);
      P1.setSize(P2.getSize());
      setSize(getLayout().preferredLayoutSize(this));
      center();
      setVisible(true);
      setResizable(false);
   }
```

```
   void center() {
      Dimension tailleEcran =
                     Toolkit.getDefaultToolkit().getScreenSize();
      int largeurEcran = tailleEcran.width;
      int hauteurEcran = tailleEcran.height;
      int largeur = getSize().width;
      int hauteur = getSize().height;
      int xPos = (largeurEcran - largeur) / 2;
      int yPos = (hauteurEcran - hauteur) / 2;
      setLocation(xPos, yPos);
   }
}

class PanelHaut extends JPanel {
   PanelHaut(String s) {
      setLayout(new FlowLayout(FlowLayout.CENTER, 0, 15));
      JLabel L1 = new JLabel(s);
      add(L1);
      setSize(getLayout().preferredLayoutSize(this));
   }
}

class PanelBas extends JPanel {
   PanelBas(String s1, String s2) {
      setLayout(new FlowLayout(FlowLayout.CENTER, 50, 0));
      JButton B1 = new JButton(s1);
      add(B1);
      JButton B2 = new JButton(s2);
      add(B2);
      setSize(getLayout().preferredLayoutSize(this));
   }
}
```

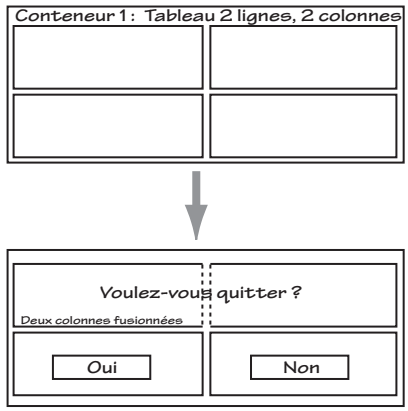
Utilisation du GridBagLayout

Plutôt que d'utiliser un layout complexe (composé de plusieurs panels utilisant des layout managers différents, il est possible d'utiliser un layout ma-

nager conçu spécialement pour les situations "complexes" (pas trop quand même !). Il s'agit du **GridBagLayout**. A propos du **GridBagLayout**, on rencontre principalement deux attitudes : ceux qui n'osent pas dire qu'ils n'arrivent pas à l'utiliser, et qui n'en fournissent que des exemples aussi intéressants que celui que nous avons déjà signalé (voir page 698), et ceux qui avouent franchement qu'il est impossible d'en obtenir quelque chose d'utile. La réalité est un peu différente.

Le **GridBagLayout** fonctionne comme un tableau dont les cellules peuvent être fusionnées. Imaginez une feuille de papier millimétré. Vous pouvez dessiner dessus n'importe quelle interface avec tous les composants que vous voulez, mais chaque composant doit toujours occuper un rectangle fait d'un nombre entier de cellules. Vous pouvez indiquer une marge autour du tableau et entre les composants. Vous pouvez également déterminer comment les composants sont alignés dans leur rectangle, et comment ils sont traités lorsque la taille de celui-ci n'est pas égale à celle du composant. (Les composants peuvent garder une taille fixe, ou être modifiés horizontalement, verticalement, ou dans les deux directions.) Il semble qu'il y ait là de quoi satisfaire tous les besoins.

La structure à mettre en œuvre est la suivante :



Le **GridBagLayout** est créé de la même façon que les autres layouts à la différence qu'il ne demande aucuns paramètres. Ceux-ci sont fournis par un

objet de type particulier appelé **GridBagConstraints**. Le processus peut donc être représenté de la façon suivante :

- Création du **GridBagLayoout**.
- Création du **GridBagConstraints** correspondant au premier composant.
- Initialisation des paramètres du **GridBagConstraints**.
- Ajout du composant.
- Création du **GridBagConstraints** correspondant au deuxième composant.
- Initialisation des paramètres du **GridBagConstraints**.
- Ajout du composant, etc.

Dans notre cas, nous utiliserons trois composants. Le premier sera une instance de **JLabel** et les deux autres des instances de **JButton**. Les paramètres d'un **GridBagConstraints** sont les suivants :

- **gridx, gridy** : ces deux paramètres indiquent le numéro de colonne et le numéro de ligne correspondant à l'angle supérieur gauche du composant. Notre **JLabel** utilisera les valeurs **0,0** alors que les boutons correspondront respectivement à **0,1** et **1,1**.
- **gridwidth, gridheight** : ces paramètres désignent la largeur et la hauteur du composant en nombre de cellules. Nous aurons donc **2,1** pour l'étiquette et **1,1** pour chacun des boutons.
- **fill** indique quel doit être le comportement du composant lorsque sa taille ne correspond pas à celle de sa zone d'affichage. Les valeurs possibles sont :
    - **GridBagConstraints.NONE** : par de modification de la taille.

- **GridBagConstraints.HORIZONTAL** : le composant est ajusté horizontalement.
- **GridBagConstraints.VERTICAL** : le composant est ajusté verticalement.
- **GridBagConstraints.BOTH** : le composant est ajusté dans les deux directions.

- **anchor** indique la façon dont le composant est aligné dans sa zone d'affichage. Les valeurs possibles sont :
  - **GridBagConstraints.CENTER** : alignement centré verticalement et horizontalement.
  - **GridBagConstraints.NORTH** : aligné en haut et centré horizontalement.
  - **GridBagConstraints.NORTHEAST** : aligné en haut et à droite.
  - **GridBagConstraints.EAST** : aligné à droite et centré verticalement.
  - **GridBagConstraints.SOUTHEAST** : aligné à droite et en bas.
  - **GridBagConstraints.SOUTH** : aligné en bas et centré horizontalement.
  - **GridBagConstraints.SOUTHWEST** : aligné en bas et à gauche.
  - **GridBagConstraints.WEST** : aligné à gauche et centré verticalement.
  - **GridBagConstraints.NORTHWEST** : aligné en haut et à gauche.
- **weightx, weighty** déterminent, pour chaque direction, la façon dont la zone d'affichage sera agrandie (ou rétrécie) lorsque la taille totale du

tableau sera modifiée. Si la valeur est 0, la zone d'affichage n'est pas modifiée.

- **ipadx, ipady** sont des valeurs qui seront ajoutées systématiquement à la taille minimale du composant. (Ces valeurs sont ajoutées 2 fois **à l'intérieur** du composant : au-dessus et au-dessous (**ipady**) ou à gauche et à droite (**ipadx**).)

- **insets** correspond aux marges des composants dans leur zone d'affichage.

Au lieu de spécifier la taille absolue des zones d'affichage (**gridx, gridy**), il est possible d'utiliser des coordonnées relatives en donnant à ces paramètres la valeur **GridBagConstraints.RELATIVE**. Dans ce cas, les composants sont ajoutés sur la même ligne. Pour indiquer qu'un composant est le dernier d'une ligne, il suffit de donner à la largeur de sa zone d'affichage (**gridwidth**) la valeur **GridBagConstraints.REMAINDER**. Pour la dernière ligne, il suffit de donner la même valeur à la hauteur (**gridheight**).

Le programme suivant permet d'obtenir la même boîte de dialogue que les précédents à l'aide du **GridBagLayout** :

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class GridBag1 {

   public static void main( String[] args ) {
      FenetrePrincipale maFenêtre =
                new FenetrePrincipale("Ma fenêtre");
   }

}

class FenetrePrincipale extends JFrame {
   FenetrePrincipale(String s) {
      super(s);
```

```
    addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
               System.exit(0);
            }});
    getContentPane().setLayout(new GridBagLayout());
    GridBagConstraints c = new GridBagConstraints();
    JLabel L1 = new JLabel("Voulez-vous quitter ?");
    c.weightx = 1u ;
    c.ipadx = 10;
    c.gridx = 0;
    c.gridy = 0;
    c.gridwidth = 2;
    c.gridheight = 1;
    c.insets = new Insets(30, 50, 10, 50);
    getContentPane().add(L1, c);
    JButton B1 = new JButton("Oui");
    c.gridx = 0;
    c.gridy = 1;
    c.gridwidth = 1;
    c.gridheight = 1;
    c.insets = new Insets(10, 0, 30, 0);
    getContentPane().add(B1, c);
    JButton B2 = new JButton("Non");
    c.gridx = 1;
    c.gridy = 1;
    c.gridwidth = 1;
    c.gridheight = 1;
    c.insets = new Insets(10, 0, 30, 0);
    getContentPane().add(B2, c);
    setSize(getLayout().preferredLayoutSize(this));
    center();
    setVisible(true);
    setResizable(false);
}

void center() {
    Dimension tailleEcran =
         Toolkit.getDefaultToolkit().getScreenSize();
```

```
        int largeurEcran = tailleEcran.width;
        int hauteurEcran = tailleEcran.height;
        int largeur = getSize().width;
        int hauteur = getSize().height;
        int xPos = (largeurEcran - largeur) / 2;
        int yPos = (hauteurEcran - hauteur) / 2;
        setLocation(xPos, yPos);
    }
}
```

Si vous préférez utiliser la spécification relative des colonnes, le code sera un tout petit peu plus compact. Voici le constructeur de la fenêtre principale ainsi modifié :

```
FenetrePrincipale(String s) {
    super(s);
    addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
               System.exit(0);
            }});
    getContentPane().setLayout(new GridBagLayout());
    GridBagConstraints c = new GridBagConstraints();
    JLabel L1 = new JLabel("Voulez-vous quitter ?");
    c.weightx = 1;
    c.ipadx = 10;
    c.gridwidth = GridBagConstraints.REMAINDER;
    c.gridheight = 1;
    c.insets = new Insets(30, 50, 10, 50);
    getContentPane().add(L1, c);
    JButton B1 = new JButton("Oui");
    c.gridwidth = 1;
    c.gridheight = GridBagConstraints.REMAINDER;
    c.insets = new Insets(10, 0, 30, 0);
    getContentPane().add(B1, c);
    JButton B2 = new JButton("Non");
    c.insets = new Insets(10, 0, 30, 0);
    getContentPane().add(B2, c);
```

```
        setSize(getLayout().preferredLayoutSize(this));
        center();
        setVisible(true);
        setResizable(false);
    }
```

Il est légèrement plus court car les valeurs de **gridx** et **gridy** n'ont plus besoin d'être spécifiées car **GridBagConstraints.RELATIVE** est la valeur par défaut et ne doit être spécifiée qu'une seule fois (au début de la deuxième ligne).

Note : Il est nécessaire de donner à **weightx** une valeur quelconque différente de 0. Dans le cas contraire, chaque colonne prend pour largeur celle du premier composant qui l'occupe. Dans le cas présent, la largeur des colonnes n'est fixée qu'à la deuxième ligne car, dans la première, les colonnes sont fusionnées. Nous obtenons donc une première colonne de la largeur du bouton et la deuxième beaucoup plus grande (largeur de l'étiquette moins celle de la première colonne). En donnant une valeur à **weightx** pour chaque colonne, le **GridBagLayout** agrandit ou rétrécit les colonnes dans les proportions de ces valeurs. Nous donnons donc la valeur 1 pour la première cellule. Les cellules suivantes utilisent automatiquement la même valeur.

Attention : Le **GridBagLayout** a, dans certains domaines, un comportement un peu bizarre. (Irons-nous jusqu'à dire qu'il est "buggé" ?) En effet, sa méthode **preferredSizeLayout()** renvoie une valeur nettement inférieure à la somme des **preferredSize()** de ses composants (ici, 160 au lieu de 168). Voilà pourquoi nous avons été obligés d'ajouter l'instruction **ipadx = 10**, qui ajoute 10 pixels à l'intérieur des composants (de chaque côté, soit 20 pixels en tout).

Une autre solution (préférable car elle évite des surprises dans les cas extrêmes) serait de faire nous-mêmes la somme des tailles des composants. C'est ce que nous ferons dans l'exemple suivant.

Rendre l'interface réellement indépendante du système

Les exemples précédents étaient intéressants, mais ils n'atteignent pas le but recherché qui est de rendre le programme indépendant du système. En

effet, certaines valeurs sont indiquées en pixels. Le résultat obtenu sera donc différent en fonction de la résolution de l'écran. Une solution consiste à exprimer ces valeurs en rapport avec la taille de la police de caractères.

Nous pourrions choisir nous-mêmes une taille de police en points, mais cela ne serait pas très efficace. En effet, nous retomberions dans le travers décrit précédemment : en fonction de la résolution, la taille physique de la police pourrait varier du simple au double. Nous pourrions interroger le système pour connaître la résolution, mais cela ne donnerait pas le résultat escompté. En revanche, nous pouvons utiliser les dimensions des composants pour dimensionner la fenêtre. C'est évidemment ce que Java devrait faire automatiquement mais, comme nous l'avons vu, il y a un léger défaut.

La solution que nous mettrons en œuvre consistera à faire la somme des dimensions des composants et à dimensionner la fenêtre à l'aide de ces valeurs, auxquelles nous appliquerons une fonction du type $ax + b$. Il ne sera ainsi plus du tout nécessaire d'utiliser les **insets**. Nous devrons en revanche spécifier **weightx** et **weighty**. Leur valeur sera quelconque mais identique pour les deux. Dans le programme suivant, nous avons ajouté une ligne pour spécifier une police de caractères afin que vous puissiez comparer les résultats obtenus avec différentes tailles, simulant par là les conditions rencontrées sur des systèmes variés :

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class GridBag3 {
   public static void main( String[] args ) {
      FenetrePrincipale maFenêtre =
                  new FenetrePrincipale("Ma fenêtre");
   }
}

class FenetrePrincipale extends JFrame {
   JLabel L1;
   JButton B1, B2;
```

```
FenetrePrincipale(String s) {
   super(s);
   addWindowListener(new WindowAdapter() {
           public void windowClosing(WindowEvent e) {
              System.exit(0);
           }});
   Font f = new Font("Dialog", Font.BOLD, 12);
   getContentPane().setLayout(new GridBagLayout());
   GridBagConstraints c = new GridBagConstraints();
   L1 = new JLabel("Voulez-vous quitter ?");
   L1.setFont(f);
   c.weightx = 1;
   c.weighty = 1;
   c.gridwidth = GridBagConstraints.REMAINDER;
   c.gridheight = 1;
   getContentPane().add(L1, c);
   B1 = new JButton("Oui");
   B1.setFont(f);
   c.gridwidth = GridBagConstraints.RELATIVE;
   c.gridheight = GridBagConstraints.REMAINDER;
   getContentPane().add(B1, c);
   B2 = new JButton("Non");
   B2.setFont(f);
   getContentPane().add(B2, c);
   setup();
   setVisible(true);
   setResizable(false);
}

void setup() {
   int xB = Math.max(B1.getPreferredSize().width,
                     B2.getPreferredSize().width);
   int yB = Math.max(B1.getPreferredSize().height,
                     B2.getPreferredSize().height);
   B1.setPreferredSize(new Dimension(xB, yB));
   B1.setMaximumSize(new Dimension(xB, yB));
   B1.setMinimumSize(new Dimension(xB, yB));
   B2.setPreferredSize(new Dimension(xB, yB));
```

```
        B2.setMaximumSize(new Dimension(xB, yB));
        B2.setMinimumSize(new Dimension(xB, yB));
        int x = Math.max(L1.getPreferredSize().width,
                                              2 * xB);
        int y = L1.getPreferredSize().height + yB;
        setSize((int)((x + 50) * 1.2),
                             (int)((y + 20) * 1.8));

        Dimension tailleEcran =
             Toolkit.getDefaultToolkit().getScreenSize();
        int largeurEcran = tailleEcran.width;
        int hauteurEcran = tailleEcran.height;
        int largeur = getSize().width;
        int hauteur = getSize().height;
        int xPos = (largeurEcran - largeur) / 2;
        int yPos = (hauteurEcran - hauteur) / 2;
        setLocation(xPos, yPos);
    }
}
```

Nous avons regroupé le dimensionnement et le positionnement de la boîte de dialogue dans la méthode **setup()**. Pour essayer différentes tailles de caractères, modifiez simplement la ligne :

```
Font f = new Font("Dialog", Font.BOLD, 12);
```

La taille 12, sélectionnée ici, correspond à la taille par défaut.

Affichage sans layout manager

Il est parfaitement possible de disposer les composants sans l'aide d'un layout manager. Ainsi, le programme précédent peut être réécrit de la façon suivante :

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class NullLayout {
   static FenetrePrincipale maFenêtre;

   public static void main( String[] args ) {
      maFenêtre = new FenetrePrincipale("Ma fenêtre");
   }

   static class FenetrePrincipale extends JFrame {
      FenetrePrincipale(String s) {
         super(s);
         addWindowListener(new WindowAdapter() {
               public void windowClosing(WindowEvent e) {
                  System.exit(0);
               }});
         getContentPane().setLayout(null);
         JLabel L1 = new JLabel("Voulez-vous quitter ?");
         L1.setBounds(39, 11, 135, 17);
         getContentPane().add(L1);
         JButton B1 = new JButton("Oui");
         B1.setBounds(23, 50, 59, 27);
         getContentPane().add(B1);
         JButton B2 = new JButton("Non");
         B2.setBounds(130, 50, 59, 27);
         getContentPane().add(B2);

         Dimension tailleEcran =
            Toolkit.getDefaultToolkit().getScreenSize();
         int largeurEcran = tailleEcran.width;
         int hauteurEcran = tailleEcran.height;
         int largeur = 222;
         int hauteur = 115;
         int xPos = (largeurEcran - largeur) / 2;
         int yPos = (hauteurEcran - hauteur) / 2;
```

```
            setBounds(xPos, yPos, largeur, hauteur);
            setVisible(true);
        }
    }
}
```

Avec une police de caractères de taille 12, le résultat obtenu est exactement identique (les valeurs utilisées ici ont été obtenues à l'aide du programme précédent !).

Si votre application est destinée à une seule plate-forme, vous pouvez utiliser cette approche. Cependant, vous aurez à subir les inconvénients suivant :

- Vous devrez calculer les valeurs correctes pour chacune de vos boîtes de dialogue.
- Si votre programme doit fonctionner un jour sous un autre système, ou si la prochaine version de votre système utilise des caractères différents, vous devrez modifier chacune de vos boîtes de dialogue.
- Si un utilisateur modifie son système pour obtenir un affichage plus gros ou plus petit, les résultats risquent d'être imprévisibles.

Créer votre propre layout manager

La plupart des programmeurs se créent leur bibliothèque personnelle de sous-programmes. Dans le cas qui nous préoccupe, cela consisterait simplement à calculer la position et la taille de chaque composant. Cela ne pose pas de problème particulier. Vous pouvez parfaitement implémenter cela sous la forme d'une classe quelconque étendant la classe **JFrame** ou un autre conteneur plus approprié. Vous pouvez également créer un layout manager. Les deux approches peuvent être mélangées. La création d'un layout manager ne doit être envisagée que pour traiter des cas qui ne peuvent l'être avec les layout managers existants. S'il s'agit uniquement de simplifier l'interface, il suffit de créer une classe implémentant un layout manager existant. L'exemple suivant montre une telle classe dont le constructeur prend

pour argument une chaîne de caractères représentant une question et affiche une boîte de dialogue avec trois boutons correspondant aux réponses *Oui*, *Non* et *Je ne sais pas*. Un autre constructeur permet de spécifier la taille des caractères à utiliser. Ce programme n'est pas très différent du précédent :

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Question extends JFrame {
   JLabel L1;
   JButton B1, B2, B3;
   Question(String s) {
      this(s, 12);
   }

   Question(String s, int t) {
      addWindowListener(new WindowAdapter() {
              public void windowClosing(WindowEvent e) {
                 System.exit(0);
              }});
      Font f = new Font("Dialog", Font.BOLD, t);
      getContentPane().setLayout(new GridBagLayout());
      GridBagConstraints c = new GridBagConstraints();
      L1 = new JLabel(s);
      L1.setFont(f);
      c.weightx = 1;
      c.weighty = 1;
      c.gridwidth = GridBagConstraints.REMAINDER;
      c.gridheight = 1;
      getContentPane().add(L1, c);
      B1 = new JButton("Oui");
      B1.setFont(f);
      c.gridwidth = 1;
      getContentPane().add(B1, c);
      B2 = new JButton("Je ne sais pas");
```

```
        B2.setFont(f);
        c.gridwidth = 2;
        getContentPane().add(B2, c);
        B3 = new JButton("Non");
        B3.setFont(f);
        c.gridheight = GridBagConstraints.REMAINDER;
        getContentPane().add(B3, c);
        setup();
        setVisible(true);
        setResizable(false);
    }
    void setup() {
        int xB = Math.max(B1.getPreferredSize().width,
                          B3.getPreferredSize().width);
        int yB = B1.getPreferredSize().height;
        int xB2 = B2.getPreferredSize().width;
        B1.setPreferredSize(new Dimension(xB, yB));
        B1.setMaximumSize(new Dimension(xB, yB));
        B1.setMinimumSize(new Dimension(xB, yB));
        B3.setPreferredSize(new Dimension(xB, yB));
        B3.setMaximumSize(new Dimension(xB, yB));
        B3.setMinimumSize(new Dimension(xB, yB));
        int x = Math.max(L1.getPreferredSize().width,
                                          2 * xB + xB2);
        int y = L1.getPreferredSize().height + yB;
        setSize((int)((x + 50) * 1.2),
                            (int)((y + 20) * 1.8));
        Dimension tailleEcran =
             Toolkit.getDefaultToolkit().getScreenSize();
        int largeurEcran = tailleEcran.width;
        int hauteurEcran = tailleEcran.height;
        int largeur = getSize().width;
        int hauteur = getSize().height;
        int xPos = (largeurEcran - largeur) / 2;
        int yPos = (hauteurEcran - hauteur) / 2;
        setLocation(xPos, yPos);
    }
}
```

Le listing suivant montre un exemple d'utilisation de cette classe. (En principe, cette classe devrait être affectée à un package.)

```
public class TestQuestion {
   public static void main( String[] args ) {
      Question uneQuestion = new Question("Aimez-vous Java ?");
   }
}
```

Les éléments de l'interface utilisateur

Nous ne décrirons pas tous les éléments de l'interface utilisateur. Tout d'abord, ils sont très nombreux et beaucoup vous seront totalement inutiles. En effet, Java 2 est livré avec tout ce qui est nécessaire pour assurer la compatibilité avec les versions 1.0 et 1.1. La version 1.1 utilisait un modèle d'événement totalement différent et très peu "orienté objet", dans lequel il était nécessaire de consulter les événements pour connaître leur nature et leur émetteur (le plus souvent au moyen d'une série d'instructions **if else**). Java 1.1 utilisait le même modèle d'événements que la version 2, mais avec un ensemble de composants lourds. Nous ne nous intéresserons ici qu'aux composants *Swing*, apparus comme une addition à la version 1.1 et complètement intégrés à la version 2.

Note : Ayant été tout d'abord distribuée comme une bibliothèque additionnelle à la version 1.1 de Java, la bibliothèque Swing respectait la convention de "nommage" des packages utilisant l'adresse Internet inversée du créateur. Les packages *Swing* étaient donc nommés **com.sun.java.swing...** Lors de l'intégration de la bibliothèque *Swing* à la version bêta de Java 2, ces packages ont tout à fait normalement été renommés **java.awt.swing...** ce qui a immédiatement entraîné des protestations de la part des utilisateurs de la version 1.1 qui se voyaient obligés de modifier tous leurs programmes (ce qui n'est pas vraiment conforme au slogan *write once, run everywhere*). C'est pourquoi dans la version 2 bêta 4, on est revenu aux noms de la version 1.1. Cependant, la version actuelle adopte un nouveau système de nommage dans lequel les packages d'abord distribués comme des exten-

sions sont ensuite intégrés avec un nom commençant par **javax**. Les packages *Swing* sont donc nommés **javax.swing...**

Nous ne traiterons pas non plus tous les éléments de la bibliothèque *Swing*. Ils sont extrêmement nombreux. En effet, contrairement à l'approche de la version 1.1, utilisant les ressources du système hôte, ce qui restreignait l'interface de Java au sous-ensemble commun aux différents systèmes utilisés (en gros *Windows*, *Motif* et *Macintosh*), le fait d'avoir choisi des éléments légers pour la bibliothèque *Swing* a retiré toute limitation. Il existe donc d'innombrables composants, dont certains mériteraient un livre à eux seuls. Il existe par exemple un composant **JTable** offrant pratiquement les fonctionnalités d'un petit tableur. On trouve également des composants qui sont de véritables éditeurs de texte, au format texte seul, HTML ou RTF. La bibliothèque *Swing* permet également la gestion complète du presse-papiers, du glisser déposer, de l'annulation répétition multiple, de l'affichage d'arbres (avec extension sélective des différents niveaux), etc. Dans la suite de ce chapitre, nous ne donnerons que quelques exemples des composants les plus courants.

Les boutons

Nous avons déjà employé des boutons, mais ceux-ci ne faisaient rien. Nous allons maintenant voir ce qui nous manque pour créer des boutons fonctionnels.

Comme nous l'avons déjà dit, un clic sur un bouton est un événement. Cet événement est un objet envoyé à un *listener*. Pour recevoir les clics sur nos boutons, nous devons donc créer des listeners, comme nous l'avions déjà fait pour la fermeture de la fenêtre. Nous allons modifier le programme précédent afin d'afficher une boîte de dialogue contenant deux boutons portant les inscriptions *Rouge* et *Bleu*. Lorsque nous cliquerons sur un bouton, la couleur correspondante sera affectée au texte de l'étiquette.

Nous avions indiqué précédemment qu'il était plus simple de réaliser un listener en étendant un adapter plutôt qu'en implémentant directement l'interface correspondante, ce qui nous évitait d'avoir à redéfinir toutes les méthodes dont nous n'avons pas besoin. Pour les boutons, l'interface à implé-

menter ne comporte qu'une seule méthode. Il n'existe donc pas d'adapter correspondant à cette interface. Nous utiliserons pour chacun des boutons une technique différente. Dans les deux cas, nous créerons une classe pour le listener de chaque bouton. Nous utiliserons une classe interne pour le premier et une classe anonyme pour le second. Rappelons qu'il est, à notre avis, tout à fait "déconseillé" (c'est un euphémisme) d'utiliser pour cela le bouton lui-même, le conteneur dans lequel il se trouve ou la classe principale. Autant que possible, un listener doit être une classe spécialement prévue à cet effet.

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Boutons {
   public static void main( String[] args ) {
      ChoixCouleur choix =
           new ChoixCouleur("Choisissez une couleur");
   }
}

class ChoixCouleur extends JFrame {
   JLabel L1;
   JButton B1, B2;

   ChoixCouleur(String s) {
      this(s, 12);
   }

   ChoixCouleur(String s, int t) {
      addWindowListener(new WindowAdapter() {
           public void windowClosing(WindowEvent e) {
              System.exit(0);
           }});
      Font f = new Font("Dialog", Font.BOLD, t);
      getContentPane().setLayout(new GridBagLayout());
      GridBagConstraints c = new GridBagConstraints();
```

```
      L1 = new JLabel(s);
      L1.setFont(f);
      c.weightx = 1;
      c.weighty = 1;
      c.gridwidth = GridBagConstraints.REMAINDER;
      c.gridheight = 1;
      getContentPane().add(L1, c);
      B1 = new JButton("Rouge");
      class AL1 implements ActionListener {
         public void actionPerformed(ActionEvent e) {
            L1.setForeground(new Color(255, 0, 0));
            L1.repaint();
         }
      }
      B1.addActionListener(new AL1());
      B1.setFont(f);
      c.gridwidth = 1;
      getContentPane().add(B1, c);
      B2 = new JButton("Bleu");
      B2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
               L1.setForeground(new Color(0, 0, 255));
               L1.repaint();
            }});
      B2.setFont(f);
      c.gridheight = GridBagConstraints.REMAINDER;
      getContentPane().add(B2, c);
      setup();
      setResizable(false);
      setVisible(true);
   }

   void setup() {
      int xB = Math.max(B1.getPreferredSize().width,
                        B2.getPreferredSize().width);
      int yB = B1.getPreferredSize().height;
      B1.setPreferredSize(new Dimension(xB, yB));
      B1.setMaximumSize(new Dimension(xB, yB));
```

```
        B1.setMinimumSize(new Dimension(xB, yB));
        B2.setPreferredSize(new Dimension(xB, yB));
        B2.setMaximumSize(new Dimension(xB, yB));
        B2.setMinimumSize(new Dimension(xB, yB));
        int x = Math.max(L1.getPreferredSize().width,
                                              2 * xB);
        int y = L1.getPreferredSize().height + yB;
        setSize((int)((x + 50) * 1.2),
                              (int)((y + 20) * 1.8));
        Dimension tailleEcran =
             Toolkit.getDefaultToolkit().getScreenSize();
        int largeurEcran = tailleEcran.width;
        int hauteurEcran = tailleEcran.height;
        int largeur = getSize().width;
        int hauteur = getSize().height;
        int xPos = (largeurEcran - largeur) / 2;
        int yPos = (hauteurEcran - hauteur) / 2;
        setLocation(xPos, yPos);
    }
}
```

En dehors de quelques modifications mineures, les parties importantes sont celles qui ont été imprimées en gras. Dans le cas du premier bouton, nous créons une classe locale (**BL1**) en implémentant l'interface **ActionListener** :

```
        class AL1 implements ActionListener {
           public void actionPerformed(ActionEvent e) {
              L1.setForeground(new Color(255, 0, 0));
              L1.repaint();
           }
        }
```

Dans cette classe, nous redéfinissons la méthode **actionPerformed** afin qu'elle modifie la couleur de texte de l'étiquette **L1** en lui attribuant une nouvelle couleur dont les composantes RVB (rouge, vert, bleu) sont 255 (le maximum), 0, 0. Nous appelons ensuite la méthode **repaint()** afin que la modification soit répercutée sur l'affichage.

Attention : Nous invoquons ici la méthode **repaint()** sur l'étiquette, ce qui ne pose pas de problème parce que nous savons qu'un changement de couleur n'a aucune répercussion sur la mise en page (le layout). Si ce n'était pas le cas, comme nous le verrons plus loin, il faudrait invoquer la méthode **repaint()** du conteneur principal. Par ailleurs, notez que cette méthode appelle simplement la méthode **paint()**, qui redessine le composant à l'écran. La méthode **paint()** est également appelée automatiquement chaque fois que le composant doit être redessiné, et en particulier la première fois qu'il est affiché et chaque fois que la fenêtre est découverte après avoir été masquée. Si vous omettez d'appeler **repaint()**, ne soyez pas surpris si le texte change de couleur chaque fois qu'il est masqué à l'écran puis redécouvert !

Cette classe est ensuite instanciée et l'instance est affectée au bouton au moyen de l'instruction :

```
B1.addActionListener(new AL1());
```

Note : Il n'y a aucune raison de définir la classe à cet endroit précis. Nous aurions pu la définir à la fin du constructeur. Nous aurions aussi pu utiliser une classe membre, ou même une classe externe. Rappelons que les classes internes ne sont internes que dans le programme source. Elles sont compilées dans des fichiers séparés. Cependant, nous faisons référence dans cette classe à un membre (**L1**) de la classe **ChoixCouleur**. Il est donc beaucoup plus facile de le faire à partir d'une classe interne à la classe **ChoixCouleur** qu'à partir d'une classe externe. De plus, **L1** pourrait très bien ne pas être membre de la classe **ChoixCouleur**. Ici, il l'est pour pouvoir être utilisé facilement dans la méthode **Setup**. Sans cette méthode, **L1** pourrait très bien être une variable locale au constructeur de la classe. Une classe locale permet de référencer facilement une variable locale, alors que cela ne serait pas possible avec une classe membre. (Il existe bien sûr des moyens de parvenir au même résultat, mais ils sont plus complexes, moins élégants et moins performants.)

Il faut remarquer que la classe **AL1** n'est instanciée qu'une seule fois, immédiatement après avoir été définie. Ce type d'architecture se prête parfaitement à l'utilisation d'une classe anonyme, ce que nous avons fait pour le second bouton :

```
B2.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        L1.setForeground(new Color(0, 0, 255));
        L1.repaint();
     }});
```

Le code est ainsi plus compact et plus élégant. Il apporte l'avantage d'une définition locale de la classe (pas besoin de tourner trois pages de listing pour trouver ce que fait le bouton) tout en limitant au minimum les termes inutiles. C'est l'architecture que nous préférerons chaque fois que possible.

Note : Un **ActionListener** reçoit les événements de type **ActionEvent**. Ces événements n'ont rien à voir avec la souris, même si, ici, ils peuvent être générés par un clic. Ils correspondent au fait qu'un bouton est actionné. Cela peut se faire en cliquant, mais également en sélectionnant le bouton à l'aide de la touche tabulation et en l'actionnant à l'aide de la barre d'espacement. Il est également possible d'envoyer explicitement par programme un tel événement.

La gestion des événements en Java doit maintenant être bien claire :

- Un objet réagit à des événements parce qu'il est doté d'un listener pour ce type d'événement. Cliquer sur un bouton ne fait rien si le bouton n'est pas doté d'un listener adéquat.

- Un listener est réalisé en implémentant l'interface **xxxListener** où **xxx** désigne le type d'événement écouté. Les types d'événements sont nommés **xxxEvent**.

- Un listener écoute un type d'événement et non un événement particulier. Le type **WindowEvent** regroupe sept événements. Le type **ActionEvent** n'en comporte qu'un seul.

- Un listener n'a pas à déterminer quel événement précis l'a réveillé. Java s'en charge en invoquant automatiquement la méthode correspondant à l'événement. (L'événement est toutefois passé en paramètre à la méthode.)

- Un listener n'a pas à déterminer l'origine de l'événement. Cela est d'une évidence absolue, mais c'est tellement contraire à ce qui se passe dans d'autres langages, et en particulier avec la version 1.0 de Java, qu'il est utile de le rappeler. (Si vous ne comprenez pas l'importance de cela, de deux choses l'une : Java 2 est votre première expérience de la programmation, et tout va bien, ou, si ce n'est pas le cas, il est urgent de relire tout ce qui précède.)

La dernière affirmation, pour évidente qu'elle soit, nécessite quelques commentaires supplémentaires. Avec un langage plus ancien (par exemple la première version de Java), la fonction du listener aurait été implémentée globalement, par exemple dans le programme principal ou dans une seule classe dédiée à cet usage. Lorsqu'une instance de cette classe aurait reçu un événement, il lui aurait fallu en déterminer la nature et la cible. Ici, la cible est tout simplement l'objet auquel le listener a été ajouté au moyen de la méthode **addActionListener**. Tous les objets qui définissent cette méthode ou qui dérivent d'une classe qui la définit peuvent donc recevoir un **ActionListener**.

Par ailleurs, cette structure implique que le lien entre un objet et son ou ses **xxxListener** (un objet peut recevoir plusieurs listeners pour chaque type d'événement) soit établi pendant l'exécution du programme. Il est donc parfaitement possible de modifier le comportement du programme pendant son exécution. C'est ce qui s'appelle le *dynamic binding* ou lien dynamique. Cela a des implications considérables. Vous pouvez parfaitement prévoir un bouton qui ne fasse rien dans une version d'un programme et qui devienne fonctionnel dans la version suivante sans que le programme contenant le bouton soit modifié. Il suffit que, dans la première version, vous ayez inclus un **ActionListener** ne faisant rien. Il suffira alors, pour mettre à jour le programme, de distribuer le nouvel **ActionListener** pour que le programme soit modifié. S'il s'agit d'un programme de quelques dizaines de mégaoctets, on voit tout de suite l'avantage de pouvoir faire une mise à jour en modifiant uniquement une classe de quelques octets ! On peut trouver d'autres applications au *dynamic binding*. On peut, par exemple, distribuer un programme dont le comportement des boutons sera déterminé par un **ActionListener** se trouvant à l'autre bout de la planète, sur un serveur. (En réalité ce type d'application est impossible car une planète n'a pas de bouts.)

Évidemment, pour cela, il est nécessaire d'implémenter les listeners sous forme de classes externes.

Les événements de souris

Il est tout à fait possible de modifier le programme précédent pour obtenir le même résultat en écoutant un autre type d'événement. Nous pouvons écouter les événements souris (**MouseEvent**) en remplaçant les **ActionListener** par des **MouseListener**, de la façon suivante :

```
B1 = new JButton("Rouge");
class AL1 extends MouseAdapter {
   public void mouseClicked(MouseEvent e) {
      L1.setForeground(new Color(255, 0, 0));
      L1.repaint();
   }
}
B1.addMouseListener(new AL1());
```

ou encore :

```
B2.addMouseListener(new MouseAdapter() {
      public void mouseClicked(MouseEvent e) {
         L1.setForeground(new Color(0, 0, 255));
         L1.repaint();
      }});
```

Dans ce cas, le fonctionnement obtenu avec la souris est exactement identique. En revanche, le clavier n'est plus opérationnel car il n'y a plus de listener pour les **ActionEvent**. Vous vous demandez peut-être ce que deviennent alors ces événements. Contrairement à ce qui se passe avec d'autres langages, ils sont perdus. Pour mettre cela en évidence, il nous faut modifier le programme pour que le premier bouton et la fenêtre écoutent les événements de souris, et que le second bouton ne fasse rien :

```
ChoixCouleur(String s, int t) {
   addWindowListener(new WindowAdapter() {
           public void windowClosing(WindowEvent e) {
              System.exit(0);
           }});
   Font f = new Font("Dialog", Font.BOLD, t);
   getContentPane().setLayout(new GridBagLayout());
   GridBagConstraints c = new GridBagConstraints();
   L1 = new JLabel(s);
   L1.setFont(f);
   c.weightx = 1;
   c.weighty = 1;
   c.gridwidth = GridBagConstraints.REMAINDER;
   c.gridheight = 1;
   getContentPane().add(L1, c);
   B1 = new JButton("Rouge");
   class AL1 extends MouseAdapter {
      public void mouseClicked(MouseEvent e) {
         L1.setForeground(new Color(255, 0, 0));
         L1.repaint();
      }
   }
   B1.addMouseListener(new AL1());
   B1.setFont(f);
   c.gridwidth = 1;
   getContentPane().add(B1, c);
   B2 = new JButton("Bleu");
   B2.setFont(f);
   c.gridheight = GridBagConstraints.REMAINDER;
   getContentPane().add(B2, c);
   addMouseListener(new MouseAdapter() {
        public void mouseClicked(MouseEvent e) {
           L1.setForeground(new Color(0, 255, 0));
           L1.repaint();
        }});
   setup();
   setResizable(false);
   setVisible(true);
}
```

Nous n'avons représenté ici que le deuxième constructeur de la classe **ChoixCouleur**. Avec ce programme, si nous cliquons sur le premier bouton, le texte de l'étiquette est affiché en rouge. Si nous cliquons sur la fenêtre, le texte est affiché en vert. En revanche, le deuxième bouton ne modifie pas le texte. Il faut noter deux choses importantes :

- Le bouton réagit toujours au clic de souris ou au clavier, en s'enfonçant.

- Aucun événement souris n'est transmis à qui que ce soit. En particulier, contrairement à ce qui se passe avec d'autres langages, l'élément qui contient le bouton ne reçoit pas d'événement. Vous pouvez noter qu'il en est de même si vous cliquez sur l'étiquette. (C'est du moins le cas si vous cliquez sur le texte. Si vous cliquez entre les caractères, la fenêtre reçoit un événement souris. Le fond des étiquettes est donc "transparent" aux événements souris.)

Le fait que les événements soient locaux et ne soient pas transmis d'un objet à son conteneur est tout à fait cohérent avec le modèle des événements Java. Dans les langages qui font remonter aux événements la hiérarchie des conteneurs jusqu'à en trouver un qui l'écoute, celui-ci est obligé d'interroger l'événement pour connaître son origine, ce que les concepteurs de Java ont cherché à éviter.

Si vous estimez nécessaire de transmettre un événement d'un objet à son conteneur, il faut le faire explicitement, en écoutant l'événement pour l'attraper et le renvoyer. Notez qu'il est ainsi possible de transmettre un événement à n'importe quel listener et pas seulement à celui du conteneur. Dans le programme suivant, le second bouton écoute les événements souris et les renvoie à la fenêtre :

```
ChoixCouleur(String s, int t) {
   addWindowListener(new WindowAdapter() {
           public void windowClosing(WindowEvent e) {
              System.exit(0);
           }});
   addMouseListener(new MouseAdapter() {
        public void mouseClicked(MouseEvent e) {
           L1.setForeground(new Color(0, 255, 0));
```

```
            L1.repaint();
         }});
      Font f = new Font("Dialog", Font.BOLD, t);
      getContentPane().setLayout(new GridBagLayout());
      GridBagConstraints c = new GridBagConstraints();
      L1 = new JLabel(s);
      L1.setFont(f);
      c.weightx = 1;
      c.weighty = 1;
      c.gridwidth = GridBagConstraints.REMAINDER;
      c.gridheight = 1;
      getContentPane().add(L1, c);
      B1 = new JButton("Rouge");
      class AL1 extends MouseAdapter {
         public void mouseClicked(MouseEvent e) {
            L1.setForeground(new Color(255, 0, 0));
            L1.repaint();
         }
      }
      B1.addMouseListener(new AL1());
      B1.setFont(f);
      c.gridwidth = 1;
      getContentPane().add(B1, c);
      B2 = new JButton("Bleu");
      B2.addMouseListener(new MouseAdapter() {
         public void mouseClicked(MouseEvent e) {
            processMouseEvent(e);
         }});
      B2.setFont(f);
      c.gridheight = GridBagConstraints.REMAINDER;
      getContentPane().add(B2, c);
      setup();
      setResizable(false);
      setVisible(true);
   }
   protected void processMouseEvent(MouseEvent e) {
      super.processMouseEvent(e);
   }
```

Dans ce programme (nous n'avons représenté ici que le second constructeur de la classe **ChoixCouleur** et une nouvelle méthode), la fenêtre reçoit un **MouseListener** qui affiche le texte de l'étiquette en vert. Le second bouton reçoit lui aussi un **MouseListener**. (Il ne pourrait pas retransmettre l'événement s'il n'était pas en mesure de l'écouter.) Celui-ci appelle simplement la méthode **processMouseEvent** de la classe **ChoixCouleur** en lui passant l'événement.

La méthode **processMouseEvent** mérite quelques commentaires. Celle-ci appartient en effet à la classe **Component**, qui est un lointain ancêtre de **ChoixCouleur** par l'intermédiaire (entre autres) de **JFrame**. Cette méthode ne peut être appelée que si le composant a reçu un **MouseListener**. De plus, cette méthode est protégée, ce qui nous empêche de l'appeler directement depuis le listener du bouton. Il nous faut donc la redéfinir dans la classe **ChoixCouleur**. Le listener qui l'appelle étant une classe anonyme, il fait évidemment partie du même package que celle-ci. Nous pouvons donc redéfinir la méthode en la déclarant **protected**. Si le listener était une classe externe appartenant à un autre package, il nous faudrait la déclarer **public**.

Une utilisation de ce principe pourrait être d'effectuer un traitement spécifique pour certains boutons et un traitement général pour d'autres. Ainsi, on peut imaginer qu'un bouton donne au texte la couleur rouge, un autre la couleur bleue et qu'un troisième mette à jour l'affichage. Dans l'exemple suivant, chaque bouton change la couleur du texte et du fond. Cliquer dans la fenêtre ne change que le style du fond :

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;

public class Boutons2 {
   public static void main( String[] args ) {
      ChoixCouleur choix =
            new ChoixCouleur("Choisissez une couleur");
   }
}
```

```
class ChoixCouleur extends JFrame {
   JLabel L1;
   JButton B1, B2;
   Random random = new Random();
   Color c;

   ChoixCouleur(String s) {
      this(s, 12);
   }

   ChoixCouleur(String s, int t) {
      addWindowListener(new WindowAdapter() {
              public void windowClosing(WindowEvent e) {
                 System.exit(0);
              }});
      addMouseListener(new MouseAdapter() {
           public void mouseClicked(MouseEvent e) {
              c = new Color(Math.abs(random.nextInt()) % 256,
                      Math.abs(random.nextInt()) % 256,
                      Math.abs(random.nextInt()) % 256);
              getContentPane().setBackground(c);
              B1.setBackground(c);
              B2.setBackground(c);
              repaint();
           }});
      Font f = new Font("Dialog", Font.BOLD, t);
      getContentPane().setLayout(new GridBagLayout());
      GridBagConstraints c = new GridBagConstraints();
      L1 = new JLabel(s);
      L1.setFont(f);
      c.weightx = 1;
      c.weighty = 1;
      c.gridwidth = GridBagConstraints.REMAINDER;
      c.gridheight = 1;
      getContentPane().add(L1, c);
      B1 = new JButton("Rouge");
      B1.addMouseListener(new MouseAdapter() {
           public void mouseClicked(MouseEvent e) {
```

```
            L1.setForeground(new Color(255, 0, 0));
            processMouseEvent(e);
         }});
   B1.setFont(f);
   c.gridwidth = 1;
   getContentPane().add(B1, c);
   B2 = new JButton("Bleu");
   B2.addMouseListener(new MouseAdapter() {
         public void mouseClicked(MouseEvent e) {
            L1.setForeground(new Color(0, 0, 255));
            processMouseEvent(e);
         }});
   B2.setFont(f);
   c.gridheight = GridBagConstraints.REMAINDER;
   getContentPane().add(B2, c);
   setup();
   setResizable(false);
   setVisible(true);
}

protected void processMouseEvent(MouseEvent e) {
   super.processMouseEvent(e);
}
void setup() {
   int xB = Math.max(B1.getPreferredSize().width,
                     B2.getPreferredSize().width);
   int yB = B1.getPreferredSize().height;
   B1.setPreferredSize(new Dimension(xB, yB));
   B1.setMaximumSize(new Dimension(xB, yB));
   B1.setMinimumSize(new Dimension(xB, yB));
   B2.setPreferredSize(new Dimension(xB, yB));
   B2.setMaximumSize(new Dimension(xB, yB));
   B2.setMinimumSize(new Dimension(xB, yB));
   int x = Math.max(L1.getPreferredSize().width,
                                          2 * xB);
   int y = L1.getPreferredSize().height + yB;
   setSize((int)((x + 50) * 1.2),
                           (int)((y + 20) * 1.8));
```

```
        Dimension tailleEcran =
             Toolkit.getDefaultToolkit().getScreenSize();
        int largeurEcran = tailleEcran.width;
        int hauteurEcran = tailleEcran.height;
        int largeur = getSize().width;
        int hauteur = getSize().height;
        int xPos = (largeurEcran - largeur) / 2;
        int yPos = (hauteurEcran - hauteur) / 2;
        setLocation(xPos, yPos);
    }
}
```

Notez que pour changer la couleur du fond de la fenêtre, nous devons appeler la méthode **setBackground** de sa *contentPane*. Si nous modifions la couleur du fond de la fenêtre elle-même, nous n'obtiendrons pas le résultat souhaité car le fond de la fenêtre est masqué par la *contentPane*.

La classe **Component** comporte une méthode **processXXXEvent** pour chaque type d'événement.

Les menus

Java permet de créer très facilement des menus, qu'ils soient liés à une barre de menus ou flottants. Le programme suivant crée une fenêtre avec une barre de menus :

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;

public class Menus extends JFrame {
   public static void main( String[] args ) {
      MaFenetre fenetre = new MaFenetre("Exemple de menus");
   }
}
```

```
class MaFenetre extends JFrame {
   MaFenetre(String s) {
      super(s);
      addWindowListener(new WindowAdapter() {
              public void windowClosing(WindowEvent e) {
                 System.exit(0);
              }});
      setup();

      JMenu menu = new JMenu("Commandes", true);
      JMenuItem item1 = new JMenuItem("Ouvrir");
      item1.addActionListener(new ActionListener() {
           public void actionPerformed(ActionEvent e) {
              new BD("Affichée par Commandes/Ouvrir");
           }});
      menu.add(item1 );
      JMenuItem item2 = new JMenuItem("Ouvrir aussi");
      item2.addActionListener(new ActionListener() {
           public void actionPerformed(ActionEvent e) {
              new BD("Affichée par Commandes/Ouvrir aussi");
           }});
      menu.add(item2 );
      menu.add(new JSeparator());
      JMenuItem item3 = new JMenuItem("Quitter");
      item3.addActionListener(new ActionListener() {
           public void actionPerformed(ActionEvent e) {
              System.exit(0);
           }});
      menu.add(item3 );
      JMenuBar mb = new JMenuBar();
      mb.add(menu);
      setJMenuBar(mb);
      setVisible(true);
   }

   void setup() {
      int largeur = 400, hauteur = 300;
      setSize(largeur, hauteur);
```

```
      Dimension tailleEcran =
           Toolkit.getDefaultToolkit().getScreenSize();
      int largeurEcran = tailleEcran.width;
      int hauteurEcran = tailleEcran.height;
      int xPos = (largeurEcran - largeur) / 2;
      int yPos = (hauteurEcran - hauteur) / 2;
      setLocation(xPos, yPos);
   }
}

class BD extends JFrame {
   JLabel L1;
   JButton B1, B2;

   BD(String s) {
      addWindowListener(new WindowAdapter() {
              public void windowClosing(WindowEvent e) {
                 dispose();
              }});
      Font f = new Font("Dialog", Font.BOLD, 12);
      getContentPane().setLayout(new GridBagLayout());
      GridBagConstraints c = new GridBagConstraints();
      L1 = new JLabel(s);
      L1.setFont(f);
      c.weightx = 1;
      c.weighty = 1;
      c.gridwidth = GridBagConstraints.REMAINDER;
      c.gridheight = 1;
      getContentPane().add(L1, c);
      B1 = new JButton("Rouge");
      B1.addActionListener(new ActionListener() {
           public void actionPerformed(ActionEvent e) {
              L1.setForeground(new Color(255, 0, 0));
              repaint();
           }});
      B1.setFont(f);
      c.gridwidth = 1;
      getContentPane().add(B1, c);
```

```
        B2 = new JButton("Bleu");
        B2.addActionListener(new ActionListener() {
             public void actionPerformed(ActionEvent e) {
               L1.setForeground(new Color(0, 0, 255));
               repaint();
             }});
        B2.setFont(f);
        c.gridheight = GridBagConstraints.REMAINDER;
        getContentPane().add(B2, c);
        setup();
        setResizable(false);
        setVisible(true);
      }
      void setup() {
        int xB = Math.max(B1.getPreferredSize().width,
                          B2.getPreferredSize().width);
        int yB = B1.getPreferredSize().height;
        B1.setPreferredSize(new Dimension(xB, yB));
        B1.setMaximumSize(new Dimension(xB, yB));
        B1.setMinimumSize(new Dimension(xB, yB));
        B2.setPreferredSize(new Dimension(xB, yB));
        B2.setMaximumSize(new Dimension(xB, yB));
        B2.setMinimumSize(new Dimension(xB, yB));
        int x = Math.max(L1.getPreferredSize().width, 2 * xB);
        int y = L1.getPreferredSize().height + yB;
        setSize((int)((x + 50) * 1.2),
                               (int)((y + 20) * 1.8));
        Dimension tailleEcran =
             Toolkit.getDefaultToolkit().getScreenSize();
        int largeurEcran = tailleEcran.width;
        int hauteurEcran = tailleEcran.height;
        int largeur = getSize().width;
        int hauteur = getSize().height;
        int xPos = (largeurEcran - largeur) / 2;
        int yPos = (hauteurEcran - hauteur) / 2;
        setLocation(xPos, yPos);
      }
    }
```

Nous avons redonné ici le listing de la classe **BD**, qui est très proche de celui du programme précédent, en imprimant en gras le principal changement, qui consiste à remplacer **System.exit(0)** par **dispose()** afin que la fermeture de la boîte de dialogue n'entraîne pas la fin du programme.

Pour créer une barre de menus, nous commençons par créer une nouvelle instance de la classe **JMenu**, avec pour paramètres le nom du menu et une valeur de type **boolean** indiquant si le menu peut-être détaché ou non.

```
JMenu menu = new JMenu("Commandes", true);
```

Nous créons ensuite un article de menu en instanciant la classe **JMenuItem** avec le nom de l'article pour paramètre. Nous lui ajoutons ensuite un **ActionListener** :

```
JMenuItem item1 = new JMenuItem("Ouvrir");
item1.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        new BD("Affichée par Commandes/Ouvrir");
     }});
```

L'article est ajouté au menu à l'aide de l'instruction :

```
menu.add(item1 );
```

Nous créons de la même façon le deuxième article :

```
JMenuItem item2 = new JMenuItem("Ouvrir aussi");
item2.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        new BD("Affichée par Commandes/Ouvrir aussi");
     }});
menu.add(item2 );
```

puis le troisième après avoir inséré un séparateur :

```
menu.add(new JSeparator());
JMenuItem item3 = new JMenuItem("Quitter");
item3.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        System.exit(0);
     }});
menu.add(item3 );
```

Enfin, nous créons une barre de menus en instanciant la classe **JMenuBar**, puis nous lui ajoutons le menu :

```
JMenuBar mb = new JMenuBar();
mb.add(menu);
```

Il ne reste plus qu'à affecter cette barre de menus à notre fenêtre au moyen de la méthode **setJMenuBar()**. (Nous ne pouvons pas utiliser la méthode **add** car la barre de menus est portée par la *RootPane*.)

```
setJMenuBar(mb);
```

Les fenêtres internes

La fenêtre que nous utilisions jusqu'ici était une **JFrame**, c'est-à-dire un composant lourd. Java possède également des fenêtres implémentées sous forme de composants légers **JInternalFrame**. Ces fenêtres ne peuvent être créées qu'à l'intérieur d'une autre fenêtre. Nous pouvons modifier notre programme pour que la boîte de dialogue soit une **JInternalFrame** et non une **JFrame** :

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;
```

```
public class Menus2 extends JFrame {
   public static void main( String[] args ) {
      MaFenetre fenetre = new MaFenetre("Exemple de menus");
   }
}

class MaFenetre extends JFrame {
   JDesktopPane desktop;
   MaFenetre(String s) {
      super(s);
      addWindowListener(new WindowAdapter() {
              public void windowClosing(WindowEvent e) {
                 System.exit(0);
              }});
      setup();
      JMenu menu = new JMenu("Commandes", true);
      JMenuItem item1 = new JMenuItem("Ouvrir");
      item1.addActionListener(new ActionListener() {
           public void actionPerformed(ActionEvent e) {
              BD bd = new BD("Affichée par Commandes/Ouvrir");
              desktop.add(bd);
              try {
                 bd.setSelected(true);
              } catch (java.beans.PropertyVetoException ve) {}
           }});
      menu.add(item1 );
      JMenuItem item2 = new JMenuItem("Ouvrir aussi");
      item2.addActionListener(new ActionListener() {
           public void actionPerformed(ActionEvent e) {
              BD bd = new BD("Affichée par Commandes/Ouvrir aussi");
              desktop.add(bd);
              try {
                 bd.setSelected(true);
              } catch (java.beans.PropertyVetoException ve) {}
           }});
      menu.add(item2 );
      menu.add(new JSeparator());
      JMenuItem item3 = new JMenuItem("Quitter");
```

```
        item3.addActionListener(new ActionListener() {
             public void actionPerformed(ActionEvent e) {
                System.exit(0);
             }});
        menu.add(item3 );
        JMenuBar mb = new JMenuBar();
        mb.add(menu);
        setJMenuBar(mb);

        desktop = new JDesktopPane();
        getContentPane().add(desktop);

        setVisible(true);
     }
     void setup() {
        int largeur = 400, hauteur = 300;
        setSize(largeur, hauteur);

        Dimension tailleEcran =
                Toolkit.getDefaultToolkit().getScreenSize();
        int largeurEcran = tailleEcran.width;
        int hauteurEcran = tailleEcran.height;
        int xPos = (largeurEcran - largeur) / 2;
        int yPos = (hauteurEcran - hauteur) / 2;
        setLocation(xPos, yPos);
     }
}
class BD extends JInternalFrame {
     static int compte = 0;
     JLabel L1;
     JButton B1, B2;

     BD(String s) {
        super(s, false, true, false, false);
        addInternalFrameListener(new InternalFrameAdapter() {
             public void internalFrameClosing(InternalFrameEvent e) {
                dispose();
             }});
```

```
   Font f = new Font("Dialog", Font.BOLD, 12);
   getContentPane().setLayout(new GridBagLayout());
   GridBagConstraints c = new GridBagConstraints();
   L1 = new JLabel(s);
   L1.setFont(f);
   c.weightx = 1;
   c.weighty = 1;
   c.gridwidth = GridBagConstraints.REMAINDER;
   c.gridheight = 1;
   getContentPane().add(L1, c);
   B1 = new JButton("Rouge");
   B1.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
           L1.setForeground(new Color(255, 0, 0));
           repaint();
        }});
   B1.setFont(f);
   c.gridwidth = 1;
   getContentPane().add(B1, c);
   B2 = new JButton("Bleu");
   B2.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
           L1.setForeground(new Color(0, 0, 255));
           repaint();
        }});
   B2.setFont(f);
   c.gridheight = GridBagConstraints.REMAINDER;
   getContentPane().add(B2, c);
   setup();
   setResizable(false);
   setVisible(true);
}

void setup() {
   int xB = Math.max(B1.getPreferredSize().width,
                     B2.getPreferredSize().width);
   int yB = B1.getPreferredSize().height;
   B1.setPreferredSize(new Dimension(xB, yB));
```

```
        B1.setMaximumSize(new Dimension(xB, yB));
        B1.setMinimumSize(new Dimension(xB, yB));
        B2.setPreferredSize(new Dimension(xB, yB));
        B2.setMaximumSize(new Dimension(xB, yB));
        B2.setMinimumSize(new Dimension(xB, yB));
        int x = Math.max(L1.getPreferredSize().width, 2 * xB);
        int y = L1.getPreferredSize().height + yB;
        setSize((int)((x + 50) * 1.2), (int)((y + 20) * 1.8));
        int w = (10 + 10 * (compte++ % 10));
        setLocation(w, w);
      }
   }
```

Différentes modifications ont été apportées au programme afin d'utiliser des **JInternalFrame**. Tout d'abord, nous avons importé un nouveau package contenant les événements spécifiques correspondant aux composants Swing :

```
import javax.swing.event.*;
```

La classe **MaFenetre** déclare une variable de type **JDesktopPane**. Cette classe est un conteneur qui permet de disposer les **JInternalFrame** en plusieurs couches.

```
class MaFenetre extends JFrame {
   JDesktopPane desktop;
```

Une instance de **JDesktopPane** est créée et ajoutée à la *contentPane* de la fenêtre.

```
desktop = new JDesktopPane();
getContentPane().add(desktop);
```

Les deux premiers articles du menu créent une instance de la classe **BD** (qui étend **JInternalFrame**). Cette instance est ajoutée à la **JDesktopPane**

(appelée **desktop**) et non à la fenêtre principale. Puis la fenêtre créée est sélectionnée en invoquant sa méthode **setSelected()** avec le paramètre **true**. L'invocation de cette méthode doit être placée dans un bloc **try** de façon à intercepter l'exception **java.beans.PropertyVetoException**.

```
public void actionPerformed(ActionEvent e) {
   BD bd = new BD("Affichée par Commandes/Ouvrir");
   desktop.add(bd);
   try {
      bd.setSelected(true);
   } catch (java.beans.PropertyVetoException ve) {}
}});
```

La méthode **setup** de la fenêtre principale détermine arbitrairement une taille de 400 x 300 pixels.

La classe **BD** reprend également l'essentiel du programme précédent, à l'exception des points liés au fait qu'elle dérive de **JInternalFrame** et non de **JFrame** :

```
class BD extends JInternalFrame {
```

Son constructeur appelle le constructeur de la classe parente avec pour arguments une chaîne de caractères (le titre de la fenêtre) et quatre **boolean** indiquant si la fenêtre peut être redimensionnée, fermée, maximisée et icônifiée. Un **InternalFrameListener** remplace le **WindowListener**. Sa fonction est de supprimer la fenêtre (**dispose()**) et non plus de terminer le programme :

```
BD(String s) {
   super(s, true, true, true, true);
   addInternalFrameListener(new InternalFrameAdapter() {
        public void internalFrameClosing(InternalFrameEvent e) {
          dispose();
        }});
```

Enfin, sa méthode **setup** utilise la variable statique **compte** pour disposer les fenêtres créées en diagonale, espacées de 10 pixels. Chaque fois que dix fenêtres ont été affichées, la suivante est affichée de nouveau dans l'angle supérieur gauche :

```
int w = (10 + 10 * (compte++ % 10));
setLocation(w, w);
```

Si vous compilez et exécutez ce programme, vous obtiendrez le résultat suivant :

Le plaf (Pluggable Look And Feel)

Vous avez pu constater que l'aspect de ces fenêtres est très différent de celui de la fenêtre principale. Il s'agit là du "look and feel" *Metal*, conçu spécialement pour les applications Java. Grâce à cette conception, les applications peuvent avoir le même aspect sur tous les systèmes. Java propose également les "look & feel" Windows, Motif et Macintosh. De plus, le terme "Pluggable" indique qu'il s'agit d'un élément qui peut être ajouté à un programme, et ce même de façon dynamique, au moment de son exécution. Le programme suivant reprend le précédent en ajoutant un menu permettant à l'utilisateur de choisir son *plaf*. (Au moment où ce programme a été écrit, le plaf Macintosh n'était pas encore disponible.)

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;

public class Menus3 extends JFrame {
   public static void main( String[] args ) {
      MaFenetre fenetre = new MaFenetre("Exemple de menus");
   }
}
class MaFenetre extends JFrame {
   JDesktopPane desktop;
   MaFenetre(String s) {
      super(s);
      addWindowListener(new WindowAdapter() {
              public void windowClosing(WindowEvent e) {
                 System.exit(0);
              }});
      setup();

      JMenu menu = new JMenu("Commandes", true);
      JMenuItem item1 = new JMenuItem("Ouvrir");
      item1.addActionListener(new ActionListener() {
           public void actionPerformed(ActionEvent e) {
              BD bd = new BD("Affichée par Commandes/Ouvrir");
              desktop.add(bd);
              try {
                 bd.setSelected(true);
              } catch (java.beans.PropertyVetoException ve) {}
           }});
      menu.add(item1);
      JMenuItem item2 = new JMenuItem("Ouvrir aussi");
      item2.addActionListener(new ActionListener() {
           public void actionPerformed(ActionEvent e) {
              BD bd =
                 new BD("Affichée par Commandes/Ouvrir aussi");
              desktop.add(bd);
```

```
            try {
               bd.setSelected(true);
            } catch (java.beans.PropertyVetoException ve) {}
         }});
menu.add(item2);
menu.add(new JSeparator());
JMenuItem item3 = new JMenuItem("Quitter");
item3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         System.exit(0);
      }});
menu.add(item3);
JMenuBar mb = new JMenuBar();
mb.add(menu);
JMenu menu2 = new JMenu("Look & Feel", true);
JMenuItem item11 = new JMenuItem("Metal");
item11.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         changePlaf(1);
      }});
menu2.add(item11);
JMenuItem item12 = new JMenuItem("Motif");
item12.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         changePlaf(2);
      }});
menu2.add(item12);
JMenuItem item13 = new JMenuItem("Windows");
item13.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         changePlaf(3);
      }});
menu2.add(item13);
JMenuItem item14 = new JMenuItem("Macintosh");
item14.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         changePlaf(4);
      }});
```

```
      menu2.add(item14);
      mb.add(menu2);

      setJMenuBar(mb);

      desktop = new JDesktopPane();
      getContentPane().add(desktop);
      setVisible(true);
   }

   void setup() {
      int largeur = 400, hauteur = 300;
      setSize(largeur, hauteur);

      Dimension tailleEcran =
                     Toolkit.getDefaultToolkit().getScreenSize();
      int largeurEcran = tailleEcran.width;
      int hauteurEcran = tailleEcran.height;
      int xPos = (largeurEcran - largeur) / 2;
      int yPos = (hauteurEcran - hauteur) / 2;
      setLocation(xPos, yPos);
   }

   void changePlaf(int s) {
       try {
        switch (s) {
           case 1:
              UIManager.setLookAndFeel
              ("javax.swing.plaf.metal.MetalLookAndFeel");
              break;
           case 2:
              UIManager.setLookAndFeel
              ("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
              break;
           case 3:
              UIManager.setLookAndFeel
        ("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
              break;
```

```
            case 4:
               //UIManager.setLookAndFeel
   //("javax.swing.plaf.mac.MacLookAndFeel");
               break;
            }
            SwingUtilities.updateComponentTreeUI(this);
         }
         catch (UnsupportedLookAndFeelException e) {
             System.out.println
         ("Ce Look & Feel n'est pas disponible sur votre système.");
         }
         catch (IllegalAccessException e) {
             System.out.println
         ("Ce Look & Feel n'est pas accessible sur votre système.");
         }
         catch (ClassNotFoundException e) {
             System.out.println
                        ("Ce Look & Feel n'a pas été trouvé.");
         }
         catch (InstantiationException e) {
             System.out.println
                        ("Ce Look & Feel ne peut être instancié.");
         }
         catch (Exception e) {
             System.out.println("Erreur d'exécution.");
         }
      }
   }

   class BD extends JInternalFrame {
      static int compte = 0;
      JLabel L1;
      JButton B1, B2;

      BD(String s) {
         super(s, false, true, false, false);
         addInternalFrameListener(new InternalFrameAdapter() {
              public void internalFrameClosing(InternalFrameEvent e) {
```

```
            dispose();
         }});
      Font f = new Font("Dialog", Font.BOLD, 12);
      getContentPane().setLayout(new GridBagLayout());
      GridBagConstraints c = new GridBagConstraints();
      L1 = new JLabel(s);
      L1.setFont(f);
      c.weightx = 1;
      c.weighty = 1;
      c.gridwidth = GridBagConstraints.REMAINDER;
      c.gridheight = 1;
      getContentPane().add(L1, c);
      B1 = new JButton("Rouge");
      B1.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            L1.setForeground(new Color(255, 0, 0));
            repaint();
         }});
      B1.setFont(f);
      c.gridwidth = 1;
      getContentPane().add(B1, c);
      B2 = new JButton("Bleu");
      B2.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            L1.setForeground(new Color(0, 0, 255));
            repaint();
         }});
      B2.setFont(f);
      c.gridheight = GridBagConstraints.REMAINDER;
      getContentPane().add(B2, c);
      setup();
      setResizable(false);
      setVisible(true);
   }

   void setup() {
      int xB = Math.max(B1.getPreferredSize().width,
                        B2.getPreferredSize().width);
```

```
        int yB = B1.getPreferredSize().height;
        B1.setPreferredSize(new Dimension(xB, yB));
        B1.setMaximumSize(new Dimension(xB, yB));
        B1.setMinimumSize(new Dimension(xB, yB));
        B2.setPreferredSize(new Dimension(xB, yB));
        B2.setMaximumSize(new Dimension(xB, yB));
        B2.setMinimumSize(new Dimension(xB, yB));
        int x = Math.max(L1.getPreferredSize().width, 2 * xB);
        int y = L1.getPreferredSize().height + yB;
        setSize((int)((x + 50) * 1.2), (int)((y + 20) * 1.8));
        int w = (10 + 10 * (compte++ % 10));
        setLocation(w, w);
    }
}
```

La principale modification apportée au programme consiste en l'ajout d'un menu et d'une méthode permettant de changer le look & feel. Le menu est très simple :

```
JMenuItem item11 = new JMenuItem("Metal");
item11.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        changePlaf(1);
     }});
menu2.add(item11);
JMenuItem item12 = new JMenuItem("Motif");
item12.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        changePlaf(2);
     }});
menu2.add(item12);
JMenuItem item13 = new JMenuItem("Windows");
item13.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        changePlaf(3);
     }});
menu2.add(item13);
```

```
JMenuItem item14 = new JMenuItem("Macintosh");
item14.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        changePlaf(4);
     }});
menu2.add(item14);
mb.add(menu2);
```

Chaque article appelle la méthode **changePlaf** avec un argument numérique compris entre 1 et 4. La méthode **changePlaf** est plus intéressante. Elle comporte un sélecteur qui, en fonction de l'argument, active un look & feel. Pour sélectionner un look & feel, nous utilisons l'instruction :

```
UIManager.setLookAndFeel
          ("javax.swing.plaf.motif.MotifLookAndFeel");
```

(L'instruction est sur deux lignes uniquement pour une question d'encombrement.) Ici, le plaf Motif est sélectionné. Une fois la sélection effectuée, il faut l'appliquer au moyen de l'instruction :

```
SwingUtilities.updateComponentTreeUI(this);
```

La méthode **updateComponentTreeUI** est une méthode statique de la classe **SwingUtilities**. Elle prend pour argument un container de type **Window** ou **Applet** et modifie le look & feel de celui-ci (si c'est possible) et de tous les composants contenus. Ici, nous utilisons l'argument **this** car nous sommes au niveau de la fenêtre. Il est cependant tout à fait possible d'obtenir le même résultat à partir de n'importe quel composant en utilisant la méthode **SwingUtilities.getRoot()** qui renvoie le composant de type **Window** ou **Applet** le plus haut dans la hiérarchie contenant le composant appelant. Il est ainsi très facile d'implémenter dans une classe anonyme un changement de look & feel, par exemple :

```
JMenuItem item11 = new JMenuItem("Metal");
item11.addActionListener(new ActionListener() {
```

```
public void actionPerformed(ActionEvent e) {
   UIManager.setLookAndFeel
        ("javax.swing.plaf.motif.MotifLookAndFeel");
   SwingUtilities.updateComponentTreeUI
                        (SwingUtilities.getRoot());
}});
```

Cependant, il faut tenir compte des exceptions lancées par ces instructions.

Note : La version actuelle de Java 2 place le look Metal dans le package **javax.swing.plaf.metal** et les looks Motif et Windows dans les packages **com.sun.java.swing.plaf.motif** et **com.sun.java.swing.plaf.windows**. Il n'est pas certain que cette disposition soit conservée.

Exemples de composants

Il existe de nombreux autres éléments permettant de construire des interfaces utilisateurs. Les boutons de sélection offrent la particularité de pouvoir avoir deux types de comportement. En effet, ils peuvent être associés en groupes à l'intérieur desquels un seul bouton peut être sélectionné. Les boutons de sélection sont des instances de la classe **JCheckBox**. Ils peuvent être créés avec pour paramètres une icône, une chaîne de caractères et une valeur initiale. Ces boutons permettent une sélection exclusive ou non exclusive selon qu'ils sont utilisés de façon indépendante ou associés à un conteneur spécial, instance de **ButtonGroup**.

Nous ne pouvons pas détailler ici tous les composants d'interfaces utilisateurs de Java. Il faudrait pour cela y consacrer un livre entier. Ce livre (consacré essentiellement aux composants Swing) est aujourd'hui en préparation et devrait sortir d'ici quelques mois.) A titre d'exemple, nous donnons ici l'exemple d'un programme permettant de sélectionner une couleur pour le fond des fenêtres :

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.beans.*;
```

```
import javax.swing.*;
import javax.swing.event.*;

public class ExempleSwing {
   static Background frame;
   public static void main( String[] args ) {
      frame = new Background();
      frame.setVisible(true);
   }
}

class Background extends JWindow {
   JDesktopPane desktop;
   AppInternalFrame mainFrame;
   static final Dimension screenSize =
             Toolkit.getDefaultToolkit().getScreenSize();
   static final int largeurEcran = screenSize.width;
   static final int hauteurEcran = screenSize.height;

   public Background() {
      super();
      setBounds (0, 0, largeurEcran, hauteurEcran);
      desktop = new JDesktopPane();
      getContentPane().add(desktop);
      mainFrame = new AppInternalFrame
                ("Exemple de composants Swing", 640, 480);
      desktop.add(mainFrame);
      try {
          mainFrame.setSelected(true);
      }
      catch (PropertyVetoException e2) {}
      setBackground(new Color(192,192,192));
    }
}

class AppInternalFrame extends JInternalFrame {
   int xPos;
   int yPos;
```

```
JDesktopPane desktop;
Color BCouleur, FCouleur;
SelecteurCouleur sc = null;
AppInternalFrame moi;

AppInternalFrame(String s, int l, int h) {
   super(s, false, true, false, false);
   desktop = new JDesktopPane();
   getContentPane().add(desktop);
   xPos = (Background.largeurEcran - l) / 2;
   yPos = (Background.hauteurEcran - h) / 2;
   this.addInternalFrameListener(new InternalFrameAdapter() {
      public void internalFrameClosed(InternalFrameEvent e) {
         System.exit(0);
      }
   });
   BCouleur = new Color(255, 255, 255);
   FCouleur = new Color(192, 192, 192);
   desktop.setBackground(BCouleur);
   setBounds(xPos, yPos, l, h);
   moi = this;

   JMenu menu = new JMenu("Commandes", true);
   JMenuItem item1 = new JMenuItem("Couleurs");
   item1.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
           sc = SelecteurCouleur.create(moi);
           if (!(sc == null)) {
              desktop.add(sc);
              try {
                 sc.setSelected(true);
              } catch (PropertyVetoException ve) {}
           }
        }});
   menu.add(item1);
   JMenuItem item2 = new JMenuItem("Ouvrir");
   item2.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
```

```
            BD bd = new BD("Boîte de dialogue", moi);
            desktop.add(bd);
            try {
               bd.setSelected(true);
            } catch (PropertyVetoException ve) {}
         }});
menu.add(item2);
menu.add(new JSeparator());
JMenuItem item3 = new JMenuItem("Quitter");
item3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         System.exit(0);
      }});
menu.add(item3);
JMenuBar mb = new JMenuBar();
mb.add(menu);

JMenu menu2 = new JMenu("Look & Feel", true);
JMenuItem item11 = new JMenuItem("Metal");
item11.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         changePlaf(1);
      }});
menu2.add(item11);
JMenuItem item12 = new JMenuItem("Motif");
item12.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         changePlaf(2);
      }});
menu2.add(item12);
JMenuItem item13 = new JMenuItem("Windows");
item13.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         changePlaf(3);
      }});
menu2.add(item13);
JMenuItem item14 = new JMenuItem("Macintosh");
item14.addActionListener(new ActionListener() {
```

```
        public void actionPerformed(ActionEvent e) {
           changePlaf(4);
        }});
   menu2.add(item14);
   mb.add(menu2);

   setJMenuBar(mb);
}

void changePlaf(int s) {
    try {
     switch (s) {
        case 1:
           UIManager.setLookAndFeel
           ("javax.swing.plaf.metal.MetalLookAndFeel");
           break;
        case 2:
           UIManager.setLookAndFeel
           ("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
           break;
        case 3:
           UIManager.setLookAndFeel
        ("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
           break;
        case 4:
           //UIManager.setLookAndFeel
   ("javax.swing.plaf.mac.MacLookAndFeel");
           break;
     }
     SwingUtilities.updateComponentTreeUI(this);
   }
   catch (UnsupportedLookAndFeelException e) {
       System.out.println
   ("Ce Look & Feel n'est pas disponible sur votre système.");
   }
   catch (IllegalAccessException e) {
       System.out.println
   ("Ce Look & Feel n'est pas accessible sur votre système.");
```

```
        }
        catch (ClassNotFoundException e) {
            System.out.println
                          ("Ce Look & Feel n'a pas été trouvé.");
        }
        catch (InstantiationException e) {
            System.out.println
                       ("Ce Look & Feel ne peut être instancié.");
        }
        catch (Exception e) {
            System.out.println ("Erreur d'exécution.");
        }
    }
}

class BD extends JInternalFrame {
    static int compte = 0;
    JLabel L1;
    JButton B1, B2;
    AppInternalFrame parent;
    Color couleur;

    BD(String s, AppInternalFrame p) {
        super(s, false, true, false, false);
        parent = p;
        couleur = p.FCouleur;
        getContentPane().setBackground(couleur);
        addInternalFrameListener(new InternalFrameAdapter() {
             public void internalFrameClosing(InternalFrameEvent e) {
                dispose();
             }});
        Font f = new Font("Dialog", Font.BOLD, 12);
        getContentPane().setLayout(new GridBagLayout());
        GridBagConstraints c = new GridBagConstraints();
        L1 = new JLabel(s);
        L1.setFont(f);
        c.weightx = 1;
        c.weighty = 1;
```

```
        c.gridwidth = GridBagConstraints.REMAINDER;
        c.gridheight = 1;
        getContentPane().add(L1, c);
        B1 = new JButton("Rouge");
        B1.setBackground(couleur);
        B1.addActionListener(new ActionListener() {
             public void actionPerformed(ActionEvent e) {
                L1.setForeground(new Color(255, 0, 0));
                repaint();
             }});
        B1.setFont(f);
        c.gridwidth = 1;
        getContentPane().add(B1, c);
        B2 = new JButton("Bleu");
        B2.setBackground(couleur);
        B2.addActionListener(new ActionListener() {
             public void actionPerformed(ActionEvent e) {
                L1.setForeground(new Color(0, 0, 255));
                repaint();
             }});
        B2.setFont(f);
        c.gridheight = GridBagConstraints.REMAINDER;
        getContentPane().add(B2, c);
        setup();
        setResizable(false);
        setVisible(true);
    }

    void setup() {
        int xB = Math.max(B1.getPreferredSize().width,
                          B2.getPreferredSize().width);
        int yB = B1.getPreferredSize().height;
        B1.setPreferredSize(new Dimension(xB, yB));
        B1.setMaximumSize(new Dimension(xB, yB));
        B1.setMinimumSize(new Dimension(xB, yB));
        B2.setPreferredSize(new Dimension(xB, yB));
        B2.setMaximumSize(new Dimension(xB, yB));
        B2.setMinimumSize(new Dimension(xB, yB));
```

```
      int x = Math.max(L1.getPreferredSize().width, 2 * xB);
      int y = L1.getPreferredSize().height + yB;
      setSize((int)((x + 50) * 1.2), (int)((y + 20) * 1.8));
      int w = (10 + 10 * (compte++ % 10));
      setLocation(w, w);
   }
}

class SelecteurCouleur extends JInternalFrame {
   static int ninstances = 0;
   JLabel C1, C2, C3;
   JButton B1, B2;
   JSlider S1, S2, S3;
   JCheckBox CB1, CB2;
   ButtonGroup bg;
   JPanel panel, panel2;
   int cr, cv, cb, cr0, cv0, cb0;
   Color precBCouleur, precFCouleur;
   AppInternalFrame parent;
   boolean fond = false;
   boolean contenu = true;

   public static SelecteurCouleur create(AppInternalFrame p) {
      if (ninstances == 0) {
         ninstances++;
         return new SelecteurCouleur(p);
      }
      else {
         return null;
      }
   }
   private SelecteurCouleur(AppInternalFrame p) {
      super("Selecteur de couleur", false, true, false, false);
      addInternalFrameListener(new InternalFrameAdapter() {
           public void internalFrameClosed(InternalFrameEvent e) {
              ninstances = 0;
              dispose();
           }});
```

```
parent = p;
precBCouleur = parent.BCouleur;
precFCouleur = parent.FCouleur;
cr = precFCouleur.getRed();
cv = precFCouleur.getGreen();
cb = precFCouleur.getBlue();
cr0 = cr;
cv0 = cv;
cb0 = cb;
C1 = new JLabel("Rouge");
C2 = new JLabel("Vert");
C3 = new JLabel("Bleu");
bg = new ButtonGroup();
CB1 = new JCheckBox("Arrière plan", false);
CB1.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        cr = cr0;
        cv = cv0;
        cb = cb0;
        changeCouleur();
        fond = true;
        contenu = false;
        cr = precBCouleur.getRed();
        cv = precBCouleur.getGreen();
        cb = precBCouleur.getBlue();
        S1.setValue(cr);
        S2.setValue(cv);
        S3.setValue(cb);
        cr0 = cr;
        cv0 = cv;
        cb0 = cb;
     }});
CB2 = new JCheckBox("fenêtres", true);
CB2.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        cr = cr0;
        cv = cv0;
        cb = cb0;
```

```
            changeCouleur();
            fond = false;
            contenu = true;
            cr = precFCouleur.getRed();
            cv = precFCouleur.getGreen();
            cb = precFCouleur.getBlue();
            S1.setValue(cr);
            S2.setValue(cv);
            S3.setValue(cb);
            cr0 = cr;
            cv0 = cv;
            cb0 = cb;
         }});
      bg.add(CB1);
      bg.add(CB2);
      S1 = new JSlider(SwingConstants.HORIZONTAL, 0, 255, cr);
      S1.addChangeListener(new ChangeListener() {
         public void stateChanged(ChangeEvent e) {
            cr = S1.getValue();
            changeCouleur();
         }});
      S2 = new JSlider(SwingConstants.HORIZONTAL, 0, 255, cv);
      S2.addChangeListener(new ChangeListener() {
         public void stateChanged(ChangeEvent e) {
            cv = S2.getValue();
            changeCouleur();
         }});
      S3 = new JSlider(SwingConstants.HORIZONTAL, 0, 255, cb);
      S3.addChangeListener(new ChangeListener() {
         public void stateChanged(ChangeEvent e) {
            cb = S3.getValue();
            changeCouleur();
         }});
      B1 = new JButton("Appliquer");
      B1.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            cr0 = cr;
            cv0 = cv;
```

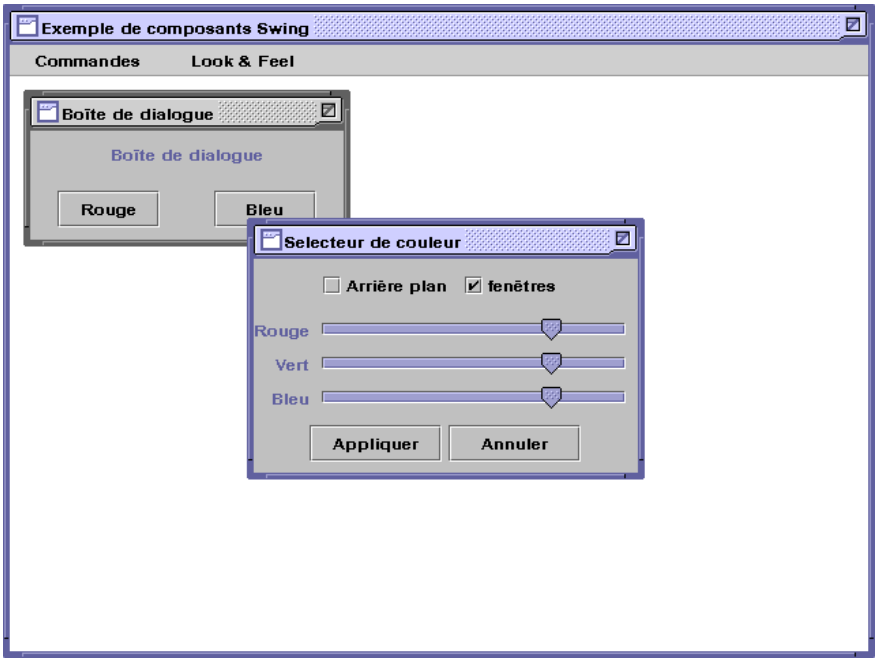
```
            cb0 = cb;
            if (contenu) {
               precFCouleur = new Color(cr, cv, cb);
               parent.FCouleur = precFCouleur;
               changeCouleur();
            }
         }});
      B2 = new JButton("Annuler");
      B2.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            cr = cr0;
            cv = cv0;
            cb = cb0;
            S1.setValue(cr);
            S2.setValue(cv);
            S3.setValue(cb);
         }});
      panel = new JPanel();
      panel.add(B1);
      panel.add(B2);
      panel2 = new JPanel();
      panel2.add(CB1);
      panel2.add(CB2);

      getContentPane().setLayout(new GridBagLayout());
      GridBagConstraints c = new GridBagConstraints();
      c.weighty = 1;
      c.weightx = 1;
      c.gridwidth = GridBagConstraints.REMAINDER;
      getContentPane().add(panel2, c);
      c.weightx = 0;
      c.anchor = GridBagConstraints.EAST;
      c.gridwidth = GridBagConstraints.RELATIVE;
      c.gridheight = 1;
      getContentPane().add(C1, c);
      c.weightx = 1;
      c.gridwidth = GridBagConstraints.REMAINDER;
      c.fill = GridBagConstraints.HORIZONTAL;
```

```
        getContentPane().add(S1, c);
        c.weightx = 0;
        c.fill = GridBagConstraints.NONE;
        c.gridwidth = GridBagConstraints.RELATIVE;
        c.gridheight = 1;
        getContentPane().add(C2, c);
        c.weightx = 1;
        c.gridwidth = GridBagConstraints.REMAINDER;
        c.fill = GridBagConstraints.HORIZONTAL;
        getContentPane().add(S2, c);
        c.weightx = 0;
        c.fill = GridBagConstraints.NONE;
        c.gridwidth = GridBagConstraints.RELATIVE;
        c.gridheight = 1;
        getContentPane().add(C3, c);
        c.weightx = 1;
        c.gridwidth = GridBagConstraints.REMAINDER;
        c.fill = GridBagConstraints.HORIZONTAL;
        getContentPane().add(S3, c);
        c.fill = GridBagConstraints.NONE;
        c.anchor = GridBagConstraints.CENTER;
        c.gridheight = GridBagConstraints.REMAINDER;
        getContentPane().add(panel, c);
        setup();
        setResizable(false);
        changeCouleur();
        setVisible(true);
    }

    void setup() {
        int xB = Math.max(B1.getPreferredSize().width,
                          B2.getPreferredSize().width);
        int yB = B1.getPreferredSize().height;
        B1.setPreferredSize(new Dimension(xB, yB));
        B1.setMaximumSize(new Dimension(xB, yB));
        B1.setMinimumSize(new Dimension(xB, yB));
        B2.setPreferredSize(new Dimension(xB, yB));
        B2.setMaximumSize(new Dimension(xB, yB));
```

```
            B2.setMinimumSize(new Dimension(xB, yB));
            int l = Math.max(C1.getPreferredSize().width, 2 * xB);
            int h = C1.getPreferredSize().height + yB;
            l = (int)((l + 50) * 1.2);
            h = (int)((h + 20) * 3);
            int x = (int)((parent.getSize().width - l) / 2);
            int y = (int)((parent.getSize().height - l) / 1.8);
            setBounds(x, y, l, h);
         }

         void changeCouleur() {
            if (fond) {
               precBCouleur = new Color(cr, cv, cb);
               parent.BCouleur = precBCouleur;
               parent.desktop.setBackground(parent.BCouleur);
            }
            else {
               precFCouleur = new Color(cr, cv, cb);
               setBackground(precFCouleur);
               panel.setBackground(precFCouleur);
               panel2.setBackground(precFCouleur);
               S1.setBackground(precFCouleur);
               S2.setBackground(precFCouleur);
               S3.setBackground(precFCouleur);
               B1.setBackground(precFCouleur);
               B2.setBackground(precFCouleur);
               CB1.setBackground(precFCouleur);
               CB2.setBackground(precFCouleur);
            }
            parent.repaint();
         }
      }
```

Ce programme utilise de nombreux éléments des programmes précédents, avec quelques améliorations. Tout d'abord, il crée une fenêtres de type **JWindow** recouvrant la totalité de l'écran. Les fenêtres **JWindow** ne comportent aucun élément graphique (barre de titre, bordure, etc.). Le but est

ici de masquer le reste de l'écran et d'afficher l'application dans une fenêtre respectant le look & feel choisi. En effet, le look & feel s'applique à tous les composants légers, mais pas aux composants lourds (ce qui est fort dommage !). Nous utilisons donc une fenêtre de type **JInternalFrame** centrée dans une fenêtre **JWindow**. Le résultat est la fenêtre suivante affichée sur un fond mauve recouvrant la totalité de l'écran :



La boîte de dialogue affiche un groupe de cases à cocher mutuellement exclusives. Les boutons sont créés tout à fait normalement en instanciant la classe **JCheckBox** :

```
CB1 = new JCheckBox("Arrière plan", false);
```

La valeur **false** indique ici que le bouton n'est pas coché.

Note : Au moment de sa création, un groupe peut ne comporter que des boutons non cochés. Une fois qu'un bouton a été coché, il n'est plus possible de le décocher sans en cocher un autre. Si vous créez une liste d'options et si vous voulez qu'il soit possible de n'en sélectionner aucune, vous devez prévoir un bouton pour l'option "Aucune". Si la désélection doit être le résultat d'une autre action, par exemple dans un formulaire comportant un bouton de remise à zéro, vous pouvez ne pas afficher le bouton "Aucune" et l'actionner au moyen de sa méthode **doClick()**.

Le bouton est ajouté à un groupe créé spécialement en instanciant la classe **ButtonGroup**. Ce groupe n'a aucune autre fonction que de rendre la sélection exclusive. En particulier, il ne peut pas être utilisé pour afficher en bloc le groupe. Il est cependant tout à fait possible d'affecter les boutons à un panel afin de les manipuler ensemble. Ils doivent alors être ajoutés individuellement au groupe *et* au panel, comme nous l'avons fait dans cet exemple :

```
bg = new ButtonGroup();
CB1 = new JCheckBox("Arrière plan", false);
      .
      .
      .
CB2 = new JCheckBox("fenêtres", true);
      .
      .
      .
bg.add(CB1);
bg.add(CB2);

      .
      .
      .
panel2.add(CB1);
panel2.add(CB2);
```

Les cases à cocher sont des boutons comme les autres et reçoivent à ce titre un **ActionListener**, implémenté ici sous la forme d'une classe anonyme :

```
B1.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        cr0 = cr;
        cv0 = cv;
        cb0 = cb;
        if (contenu) {
           precFCouleur = new Color(cr, cv, cb);
           parent.FCouleur = precFCouleur;
           changeCouleur();
        }
     }});
```

Les glissières permettant de faire varier les valeurs de rouge, vert et bleu de la couleur sont des composants d'utilisation très simple :

```
S1 = new JSlider(SwingConstants.HORIZONTAL, 0, 255, cr);
S1.addChangeListener(new ChangeListener() {
     public void stateChanged(ChangeEvent e) {
        cr = S1.getValue();
        changeCouleur();
     }});
```

Ils prennent pour paramètre quatre **int** : une orientation (verticale ou horizontale), leur limite basse et leur limite haute, ainsi que leur valeur initiale. Ils reçoivent un **ChangeListener**. Celui-ci a la charge d'interroger le composant auquel il appartient afin de connaître la valeur résultant de la modification. On utilise pour cela la méthode **getValue()**.

Attention : Ce programme a été testé avec la version bêta 4 de Java 2, qui comportait quelques bugs. En particulier, les glissières ne s'affichaient pas correctement lorsqu'une autre fenêtre était affichée.

Résumé

Il reste de très nombreux composants à étudier. Heureusement, leur fonctionnement est très intuitif. Une fois que vous avez compris le principe, la documentation fournie avec Java suffit pour les mettre en œuvre. Au moment où ces lignes sont écrites, il n'existe pas de livre sur la bibliothèque Swing. Si vous peinez un peu pour trouver les informations recherchées, dites-vous que plus un problème se pose avec acuité, plus la solution est profitable. Et si vous êtes vraiment bloqué, attendez la sortie de notre prochain livre sur le sujet !

Chapitre 19

Le graphisme

Dans le chapitre précédent, nous avons étudié un certain nombre de composants de l'interface utilisateur de Java. Ces composants permettent, en les assemblant, de construire l'interface graphique d'un programme. Cependant, il est des cas où le programmeur doit ajouter des éléments ne pouvant pas être obtenus à l'aide des composants de l'interface. Il dispose alors de trois moyens pour parvenir au résultat recherché : les primitives graphiques, le texte et les images.

Les primitives graphiques

Les primitives graphiques sont des méthodes qui permettent de tracer des éléments graphiques simples tels que lignes droites, rectangles, ellipses,

polygones ou arcs de cercle. Le programmeur utilise ces primitives pour composer un dessin complexe. Ces primitives étant des méthodes, vous vous doutez qu'elles appartiennent à une classe ou à une instance de classe.

Les objets graphiques

Toutes les primitives permettant d'obtenir des formes graphiques quelconques doivent être invoquées sur une instance de la classe **Graphics** ou de la classe **Graphics2D**. Cette instance est appelée *contexte graphique*. Cependant, ces classes ne peuvent pas être instanciées car il s'agit de classes abstraites. Il existe en fait deux moyens principaux pour obtenir un contexte graphique. L'un consiste à invoquer la méthode **getGraphics** sur un objet possédant un contexte graphique. L'autre consiste à utiliser le paramètre passé à la méthode **paint**. La classe **Graphics** fournit les primitives disponibles jusqu'à la version 1.1 de Java. La classe **Graphics2D** fournit les nouvelles primitives de la version 2.

Un composant pour les graphismes

De nombreux composants de l'interface utilisateur ont un contexte graphique. Il existe toutefois un composant prévu spécialement pour fournir son contexte graphique. Il s'agit de la classe **Canvas**. La première chose à faire pour utiliser des fonctions graphiques est donc de créer une instance de **Canvas**. Le programme suivant utilise des éléments des programmes des chapitres précédents pour créer un **Canvas** :

```
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;

public class Graphisme {
   static Tableau frame;
   public static void main( String[] args ) {
   frame = new Tableau();
   frame.setVisible(true);
   }
}
```

```
class Tableau extends JFrame {

   static final Dimension screenSize =
                  Toolkit.getDefaultToolkit().getScreenSize();
   static final int largeurEcran = screenSize.width;
   static final int hauteurEcran = screenSize.height + 2;
   int l = 640;
   int h = 480;

   public Tableau() {
      super();
      this.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }});
      setBounds ((largeurEcran - l) / 2,
                                 (hauteurEcran - h) / 2, l, h);
      getContentPane().setBackground(new Color(255, 255, 255));
   }
}
```

Ce programme affiche simplement une fenêtre vide sur un fond uni. Pour utiliser les méthodes graphiques, il nous faut un support. Nous pouvons utiliser pour cela un *canvas*. Un canvas est un composant offrant simplement une surface rectangulaire sur laquelle nous allons pouvoir dessiner.

Pour pouvoir dessiner sur un canvas, il faut :

- Obtenir un handle pour son contexte graphique.
- Être notifié du moment auquel le canvas est affiché afin d'ajouter au dessin du canvas le tracé de nos primitives.

Comme tout composant, un canvas est affiché en invoquant sa méthode **paint**. Cette méthode est appelée automatiquement dans les cas suivants :

- Le composant doit être affiché pour la première fois.

- Le composant doit être réaffiché parce qu'il a été masqué par un autre élément.

- Le composant doit être réaffiché parce qu'il a été masqué par programme (**setVisible(false)** puis **setVisible(true)**).

- Une demande de mise à jour a été exécutée en invoquant sa méthode **repaint()**.

La méthode **paint()** d'un canvas est très simple :

```
public void paint(Graphics g) {
   Graphics2D g2 = (Graphics2D)g;
   g2.setColor(getBackground());
   g2.fillRect(0, 0, width, height);
}
```

Cette méthode utilise la primitive graphique **fillRect** pour dessiner un rectangle plein de la couleur déterminé par **getBackground** et de dimensions égales à celles du canvas lui-même. Nous n'aurons donc la possibilité de dessiner autre chose qu'un rectangle monochrome qu'en redéfinissant la méthode **paint**.

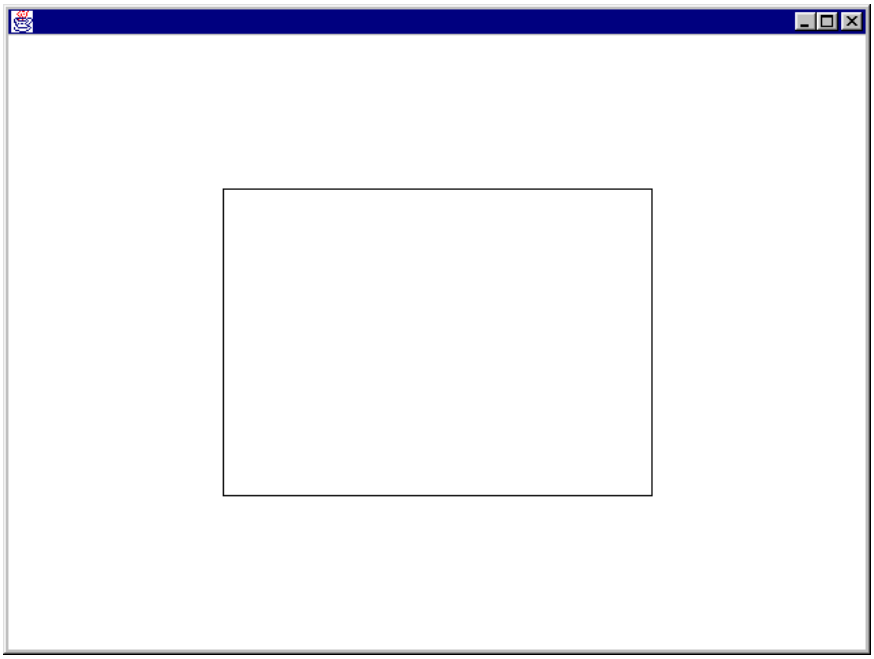
Note : Le sous-casting de **Graphics** en **Graphics2D** n'est pas ici obligatoire car nous n'utilisons que des méthodes définies dans **Graphics**. Cependant, l'effectuer systématiquement nous permet d'utiliser indifféremment (grâce au polymorphisme) les méthodes de **Graphics** et de **Graphics2D**.

Le programme suivant crée une nouvelle classe étendant **Canvas** et traçant un rectangle noir :

```
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;

public class Graphisme2 {
   static Tableau frame;
```

```
   public static void main( String[] args ) {
   frame = new Tableau();
   frame.setVisible(true);
   }
}

class Tableau extends JFrame {

   static final Dimension screenSize =
                  Toolkit.getDefaultToolkit().getScreenSize();
   static final int largeurEcran = screenSize.width;
   static final int hauteurEcran = screenSize.height + 2;
   int l = 640;
   int h = 480;

   public Tableau() {
      super();
      this.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }});
      setBounds ((largeurEcran - l) / 2,
                                 (hauteurEcran - h) / 2, l, h);
      getContentPane().setBackground(new Color(255, 255, 255));
      getContentPane().add(new Canvas() {
         public void paint(Graphics g) {
            Graphics2D g2 = (Graphics2D)g;
            int l2 = getSize().width;
            int h2 = getSize().height;
            g2.setColor(new Color(0, 0, 0));
            g2.drawRect((int)(l2 / 4), (int)(h2 / 4),
                                 (int)(l2 / 2), (int)(h2 / 2));
         }});
   }
}
```

Ce programme affiche le résultat suivant :



Nous aurions pu tout aussi bien redéfinir la méthode **paint** de notre **JFrame**, de la façon suivante :

```
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;

public class Graphisme3 {
   static Tableau frame;
   public static void main( String[] args ) {
   frame = new Tableau();
   frame.setVisible(true);
   }
}

class Tableau extends JFrame {
   static final Dimension screenSize =
              Toolkit.getDefaultToolkit().getScreenSize();
```

```
   static final int largeurEcran = screenSize.width;
   static final int hauteurEcran = screenSize.height + 2;
   int l = 640;
   int h = 480;

   public Tableau() {
      super();
      this.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }});
      setBounds ((largeurEcran - l) / 2,
                                 (hauteurEcran - h) / 2, l, h);
      getContentPane().setBackground(new Color(255, 255, 255));
   }

   public void paint(Graphics g) {
      Graphics2D g2 = (Graphics2D)g;
      int l2 = getSize().width;
      int h2 = getSize().height;
      g2.setColor(new Color(0, 0, 0));
      g2.drawRect((int)(l2 / 4), (int)(h2 / 4),
                                    (int)(l2 / 2), (int)(h2 / 2));
   }
}
```

Cependant, nous avons là un petit problème. En effet, le fond de l'écran est gris au lieu d'être blanc comme spécifié par la ligne :

```
getContentPane().setBackground(new Color(255, 255, 255));
```

En effet, nous avons redéfini la méthode **paint** de la fenêtre. Or, c'est dans cette méthode que la couleur de fond indiquée par **setBackground** est appliquée. Si nous voulons dessiner directement sur la fenêtre, nous devons, au choix, prendre à notre charge l'affichage de celle-ci, ou nous en passer. C'est pourquoi il est préférable d'utiliser un support spécial dont la méthode **paint** ne fait normalement pratiquement rien. La classe **Canvas** est idéale

pour cela. Il faut cependant noter qu'un canvas n'est pas transparent et masque ce qui se trouve derrière.

Remarquez que la redéfinition de la méthode **paint** résout le problème de l'obtention d'un contexte graphique. En effet, chaque fois que cette méthode est invoquée, le contexte graphique de l'objet à afficher lui est passé en paramètre.

Les différentes primitives

La classe **Graphics** dispose des primitives graphiques suivantes :

- **void drawRect(int x, int y, int l, int h)**

  Trace un rectangle vide de largeur *l* et de hauteur *h* en utilisant la couleur courante. L'angle supérieur gauche est situé à la position *x*, *y*.

- **void fillRect(int x, int y, int l, int h)**

  Remplit un rectangle de largeur *l* et de hauteur *h* en utilisant la couleur courante. L'angle supérieur gauche est situé à la position *x*, *y*.

- **void drawRoundRect(int x, int y, int l, int h, int la, int ha)**

  Trace un rectangle de largeur *l* et de hauteur *h* en utilisant la couleur courante. L'angle supérieur gauche est situé à la position *x*, *y*. La largeur des arcs de cercle est *la* et leur hauteur est *ha*.

- **void fillRoundRect(int x, int y, int l, int h, int la, int ha)**

  Remplit un rectangle de largeur *l* et de hauteur *h* en utilisant la couleur courante. L'angle supérieur gauche est situé à la position *x*, *y*. La largeur des arcs de cercle est *la* et leur hauteur est *ha*.

- **void draw3DRect(int x, int y, int l, int h, boolean relevé)**

  Trace un rectangle en relief de largeur *l* et de hauteur *h* en utilisant la couleur courante. L'angle supérieur gauche est situé à la position *x*, *y*. **relevé** indique si le rectangle est saillant (**true**) ou en creux (**false**).

- **void fill3DRect(int x, int y, int l, int h, boolean relevé)**

  Trace un rectangle en relief de largeur *l* et de hauteur *h* en utilisant la couleur courante. L'angle supérieur gauche est situé à la position *x, y*. **relevé** indique si le rectangle est saillant (**true**) ou en creux (**false**).

- **void drawOval(int x, int y, int l, int h)**

  Trace un ovale inscrit dans un rectangle de largeur *l* et de hauteur *h* en utilisant la couleur courante. L'angle supérieur gauche du rectangle est situé à la position *x, y*. (Le rectangle sert de référence et n'est pas tracé.)

- **void fillOval(int x, int y, int l, int h)**

  Remplit un ovale inscrit dans un rectangle de largeur *l* et de hauteur *h* en utilisant la couleur courante. L'angle supérieur gauche du rectangle est situé à la position *x, y*. (Le rectangle sert de référence et n'est pas tracé.)

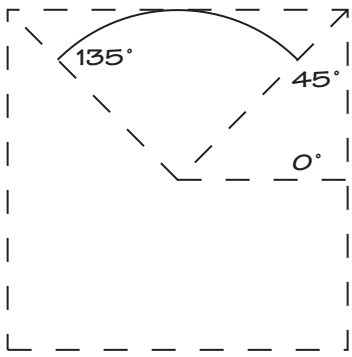
- **void drawArc(int x, int y, int l, int h, int angleInitial, int angle)**

  Trace un arc d'ellipse dans un rectangle de largeur *l* et de hauteur *h* en utilisant la couleur courante. L'angle supérieur gauche du rectangle est situé à la position *x, y*. (Le rectangle sert de référence et n'est pas tracé.) L'arc est tracé à partir de l'angle **angleInitial**, mesuré en degrés à partir de la position trigonométrique 0 (3 heures en langage militaire). Les angles sont mesurés pour un arc inscrit dans un carré qui serait ensuite déformé en un rectangle. Cela a l'air très compliqué dit de cette façon, mais c'est très simple sur un croquis, comme on peut le voir sur la figure de la page suivante.

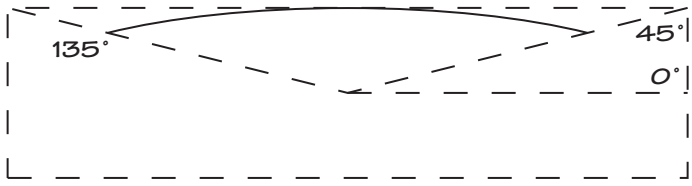
- **void fillArc(int x, int y, int l, int h, int angleInitial, int angle)**

  Remplit un arc d'ellipse dans un rectangle de largeur *l* et de hauteur *h* en utilisant la couleur courante. L'angle supérieur gauche du rectangle est situé à la position *x, y*. (Le rectangle sert de référence et n'est pas tracé.) L'arc est tracé à partir de l'angle **angleInitial**, mesuré en degrés à partir de la position trigonométrique 0 (3 heures en langage militaire).

drawArc(x, y, 100, 100, 45, 135)



drawArc(x, y, 200, 50, 45, 135)



- **void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)**

  Trace un polygone fermé défini par les tableaux de coordonnées **xPoints** et **yPoints** en utilisant la couleur courante. **nPoints** indique le nombre de points à prendre en compte.

- **void drawPolygon(Polygon p)**

  Trace un polygone fermé défini par l'objet **p** en utilisant la couleur courante. Un objet de type **Polygon** est créé avec les paramètres **int[] xPoints**, **int[] yPoints** et **int nPoints**.

- **void fillPolygon(int[] xPoints, int[] yPoints, int nPoints)**

  Remplit un polygone fermé défini par les tableaux de coordonnées **xPoints** et **yPoints** en utilisant la couleur courante. **nPoints** indique le nombre de points à prendre en compte.

- **void fillPolygon(Polygon p)**

  Remplit un polygone fermé défini par l'objet **p** en utilisant la couleur courante.

- **void drawLine(int x1, int y1, int x2, int y2)**

  Trace une ligne reliant les points de coordonnées **x1, x2** et **y1, y2** en utilisant la couleur courante.

- **void clearRect(int x, int y, int width, int height)**

  Remplit un rectangle vide de largeur *l* et de hauteur *h* en utilisant la couleur du fond. L'angle supérieur gauche est situé à la position *x, y*.

La classe **graphics2D** dispose (entre autres) des primitives suivantes :

- **void draw(Shape s)**

  Trace le contour d'une forme **s** en utilisant les attributs graphiques courants.

- **void fill(Shape s)**

  Trace le contour d'une forme **s** en utilisant les attributs graphiques courants.

- **Color getBackground()**
- **void setBackground(Color color)**

  Déterminent la couleur du fond.

- **Color getStroke()**
- **void setStroke(Stroke s)**

  Déterminent le type de ligne utilisé pour le tracé.

- **Color getPaint()**
- **void setPaint(Paint p)**

  Déterminent le type de remplissage.

- **AffineTransform getTransform()**
- **void setTransform(AffineTransform Tx)**

  Déterminent le type de transformation appliqué.

Il existe de nombreuses autres primitives dont vous trouverez la description dans la documentation en ligne de l'API de Java.

Les formes (**Shape**) susceptibles d'être tracées à l'aide des primitives de **Graphics2D** sont les suivantes :

- **Arc2D** (arc de cercle)
- **Area** (union, intersection ou soustraction de formes simples)
- **CubicCurve2D** (courbe cubique ou *courbe de Béziers*)
- **Dimension2D**
- **Ellipse2D**
- **GeneralPath** (combinaison de lignes droites et courbes)
- **Line2D**
- **Point2D**
- **QuaCurve2D** (courbe quadratique)
- **Rectangle2D**
- **RectangularShape**
- **RoundRectangle2D**

A titre d'exemple, nous allons écrire un programme permettant de tracer un rectangle au contour pointillé à l'aide de la souris.

```
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
import java.beans.*;
import com.sun.java.swing.event.*;
import java.awt.geom.*;

public class Graphisme4 {
   static Tableau frame;
      public static void main( String[] args ) {
      frame = new Tableau();
      frame.setVisible(true);
   }
}

class Tableau extends JFrame {
   static final Dimension screenSize =
                     Toolkit.getDefaultToolkit().getScreenSize();
   static final int largeurEcran = screenSize.width;
   static final int hauteurEcran = screenSize.height + 2;
   int l = 640;
   int h = 480;
   JDialog sc = null;
   Color BCouleur = new Color(255, 255, 255);
   Color FCouleur = new Color(127, 127, 127);
   Tableau moi;
   Canvas dessin;
   Container pane;
   JButton couleur;
   JCheckBox plein, vide, relief, plat;
   ButtonGroup bg3D, bgContour;
   boolean is3D, isContour;
   JPanel panelHaut;
   int x1, y1, x2, y2;

   public Tableau() {
      super();
      pane = getContentPane();
```

```
this.addWindowListener(new WindowAdapter() {
   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }});
setBackground(BCouleur);
moi = this;
setBounds ((largeurEcran - l) / 2,
                     (hauteurEcran - h) / 2, l, h);
pane.setBackground(BCouleur);
bgContour = new ButtonGroup();
bg3D = new ButtonGroup();
couleur = new JButton("Couleur");
couleur.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        sc = SelecteurCouleur.create(moi);
     }});
vide = new JCheckBox("Contour", false);
vide.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        isContour = true;
     }});
plein = new JCheckBox("Fond", true);
plein.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        isContour = false;
     }});
plein.setPreferredSize(vide.getPreferredSize());
plat = new JCheckBox("2D", true);
plat.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        is3D = false;
     }});
plat.setPreferredSize(vide.getPreferredSize());
relief = new JCheckBox("3D", false);
relief.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        is3D = true;
     }});
```

```
relief.setPreferredSize(vide.getPreferredSize());
bgContour.add(vide);
bgContour.add(plein);
bg3D.add(plat);
bg3D.add(relief);
panelHaut = new JPanel();
panelHaut.add(vide);
panelHaut.add(plein);
panelHaut.add(plat);
panelHaut.add(relief);
panelHaut.add(couleur);
pane.add(panelHaut, BorderLayout.NORTH);

dessin = new Canvas() {
   public void paint(Graphics g) {
   Graphics2D g2 = (Graphics2D)g;
   float[] motif = {5.0f};
   BasicStroke pointillé = new BasicStroke(2.0f,
                              BasicStroke.CAP_BUTT,
                              BasicStroke.JOIN_MITER,
                              5.0f, motif, 0.0f);
      if (x2 == 0)
         return;
      int x = Math.min(x1, x2);
      int y = Math.min(y1, y2);
      int l = Math.max(x1, x2) - x;
      int h = Math.max(y1, y2) - y;
      g.setColor(FCouleur);
      if (is3D) {
         if (isContour) {
            g.draw3DRect(x, y, l, h, true);
         }
         else {
            g.fill3DRect(x, y, l, h, true);
         }
      }
      else {
         if (isContour) {
```

```
                    g2.setStroke(pointillé);
                    g2.draw(new Rectangle2D.Float (x, y, l, h));
                 }
                 else {
                    g.fillRect(x, y, l, h);
                 }
              }
           }};
        dessin.addMouseListener(new MouseAdapter() {
           public void mousePressed(MouseEvent e) {
              x1 = e.getX();
              y1 = e.getY();
              x2 = 0;
              y2 = 0;
           }});
        dessin.addMouseMotionListener(new MouseMotionAdapter() {
           public void mouseDragged(MouseEvent e) {
              x2 = e.getX();
              y2 = e.getY();
              dessin.repaint();
           }});

        pane.add(dessin, BorderLayout.CENTER);
     }
  }
  class SelecteurCouleur extends JDialog {
     static int nInstances = 0;
     JLabel C1, C2, C3;
     JButton B1, B2;
     JSlider S1, S2, S3;
     JPanel panel, panel2;
     int cr, cv, cb, cr0, cv0, cb0;
     Color precFCouleur;
     Tableau parent;
     Sample sample;
     public static SelecteurCouleur create(Tableau t) {
        if (nInstances == 0) {
           nInstances++;
```

```
      return new SelecteurCouleur(t);
   }
   else {
      return null;
   }
}

private SelecteurCouleur(Tableau t) {
   super(t, "Selecteur de couleur", true);
   parent = t;
   setDefaultCloseOperation(DISPOSE_ON_CLOSE);
   addWindowListener(new WindowAdapter() {
           public void windowClosed(WindowEvent e) {
              nInstances = 0;
           }});
   precFCouleur = parent.FCouleur;
   cr = precFCouleur.getRed();
   cv = precFCouleur.getGreen();
   cb = precFCouleur.getBlue();
   cr0 = cr;
   cv0 = cv;
   cb0 = cb;
   C1 = new JLabel("Rouge");
   C2 = new JLabel("Vert");
   C3 = new JLabel("Bleu");
   S1 = new JSlider(SwingConstants.HORIZONTAL, 0, 255, cr);
   S1.addChangeListener(new ChangeListener() {
        public void stateChanged(ChangeEvent e) {
           cr = S1.getValue();
           changeCouleur();
        }});
   S2 = new JSlider(SwingConstants.HORIZONTAL, 0, 255, cv);
   S2.addChangeListener(new ChangeListener() {
        public void stateChanged(ChangeEvent e) {
           cv = S2.getValue();
           changeCouleur();
        }});
   S3 = new JSlider(SwingConstants.HORIZONTAL, 0, 255, cb);
```

```
S3.addChangeListener(new ChangeListener() {
     public void stateChanged(ChangeEvent e) {
        cb = S3.getValue();
        changeCouleur();
     }});
B1 = new JButton("Appliquer");
B1.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        cr0 = cr;
        cv0 = cv;
        cb0 = cb;
        precFCouleur = new Color(cr, cv, cb);
        parent.FCouleur = precFCouleur;
        changeCouleur();
        parent.dessin.repaint();
        dispose();
     }});
B2 = new JButton("Annuler");
B2.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
        cr = cr0;
        cv = cv0;
        cb = cb0;
        S1.setValue(cr);
        S2.setValue(cv);
        S3.setValue(cb);
        dispose();
     }});
panel = new JPanel();
panel.add(B1);
panel.add(B2);
panel2 = new JPanel();
sample = new Sample();
panel2.add(sample);
getContentPane().setLayout(new GridBagLayout());
GridBagConstraints c = new GridBagConstraints();
c.weighty = 1;
c.weightx = 1;
```

```
        c.gridwidth = GridBagConstraints.REMAINDER;
        getContentPane().add(panel2, c);
        c.weightx = 0;
        c.anchor = GridBagConstraints.EAST;
        c.gridwidth = GridBagConstraints.RELATIVE;
        c.gridheight = 1;
        getContentPane().add(C1, c);
        c.weightx = 1;
        c.gridwidth = GridBagConstraints.REMAINDER;
        c.fill = GridBagConstraints.HORIZONTAL;
        getContentPane().add(S1, c);
        c.weightx = 0;
        c.fill = GridBagConstraints.NONE;
        c.gridwidth = GridBagConstraints.RELATIVE;
        c.gridheight = 1;
        getContentPane().add(C2, c);
        c.weightx = 1;
        c.gridwidth = GridBagConstraints.REMAINDER;
        c.fill = GridBagConstraints.HORIZONTAL;
        getContentPane().add(S2, c);
        c.weightx = 0;
        c.fill = GridBagConstraints.NONE;
        c.gridwidth = GridBagConstraints.RELATIVE;
        c.gridheight = 1;
        getContentPane().add(C3, c);
        c.weightx = 1;
        c.gridwidth = GridBagConstraints.REMAINDER;
        c.fill = GridBagConstraints.HORIZONTAL;
        getContentPane().add(S3, c);
        c.fill = GridBagConstraints.NONE;
        c.anchor = GridBagConstraints.CENTER;
        c.gridheight = GridBagConstraints.REMAINDER;
        getContentPane().add(panel, c);
        setup();
        setResizable(false);
        changeCouleur();
        setVisible(true);
    }
```

```
   void setup() {
      int xB = Math.max(B1.getPreferredSize().width,
                        B2.getPreferredSize().width);
      int yB = B1.getPreferredSize().height;
      B1.setPreferredSize(new Dimension(xB, yB));
      B1.setMaximumSize(new Dimension(xB, yB));
      B1.setMinimumSize(new Dimension(xB, yB));
      B2.setPreferredSize(new Dimension(xB, yB));
      B2.setMaximumSize(new Dimension(xB, yB));
      B2.setMinimumSize(new Dimension(xB, yB));
      int l = Math.max(C1.getPreferredSize().width, 2 * xB);
      int h = C1.getPreferredSize().height + yB;
      l = (int)((l + 50) * 1.2);
      h = (int)((h + 20) * 3);
      int x = (int)((parent.getSize().width - l) / 2) +
                                    parent.getLocation().x;
      int y = (int)((parent.getSize().height - l) / 1.1) +
                                    parent.getLocation().y;
      setBounds(x, y, l, h);
   }

   void changeCouleur() {
      precFCouleur = new Color(cr, cv, cb);
      sample.setForeground(precFCouleur);
      sample.repaint();
   }
}

class Sample extends Canvas {
   Sample() {
      setSize(100,50);
   }
   public void paint(Graphics g) {
      g.fillRect(10, 10, getSize().width - 20,
                                    getSize().height - 20);
   }
}
```

La partie intéressante de ce programme est celle qui trace les rectangles. Au départ, les coordonnées des angles opposés du rectangle valent 0. Le listener de souris initialise les coordonnées du premier angle **x1, y1** lorsque le bouton de la souris est pressé :

```
dessin.addMouseListener(new MouseAdapter() {
   public void mousePressed(MouseEvent e) {
      x1 = e.getX();
      y1 = e.getY();
      x2 = 0;
      y2 = 0;
   }});
```

En même temps, les coordonnées de l'angle opposé sont réinitialisées. Lorsque la souris est déplacée avec le bouton enfoncé, le listener de mouvement de souris initialise les coordonnées de l'angle opposé et redessine le canvas :

```
dessin.addMouseMotionListener(new MouseMotionAdapter() {
   public void mouseDragged(MouseEvent e) {
      x2 = e.getX();
      y2 = e.getY();
      dessin.repaint();
   }});
```

La méthode **paint** du canvas n'effectue un tracé que si le deuxième angle du rectangle a été initialisé. Elle détermine alors les coordonnées de l'angle supérieur gauche et de l'angle inférieur droit en comparant les valeurs **x1** et **x2**, d'une part, **y1** et **y2** d'autre part. Il est ainsi possible de tracer le rectangle en commençant par un angle quelconque :

```
if (x2 == 0)
   return;
int x = Math.min(x1, x2);
int y = Math.min(y1, y2);
int l = Math.max(x1, x2) - x;
int h = Math.max(y1, y2) - y;
```

La couleur est alors appliquée et le rectangle tracé en fonction des valeurs sélectionnées au moyen des boutons :

```
g.setColor(FCouleur);
if (is3D) {
   if (isContour) {
      g.draw3DRect(x, y, l, h, true);
   }
   else {
      g.fill3DRect(x, y, l, h, true);
   }
}
else {
   if (isContour) {
      g2.setStroke(pointillé);
      g2.draw(new Rectangle2D.Float (x, y, l, h));
   }
   else {
      g.fillRect(x, y, l, h);
   }
}
```

Les rectangles sont tracés au moyen des primitives de la classe **Graphics**, sauf dans le cas du contour d'un rectangle en deux dimensions. On utilise alors la méthode **draw** de la classe **Graphics2** pour tracer un rectangle pointillé. La méthode **setStroke** est préalablement appelée pour définir le type de ligne. Elle prend pour paramètre l'objet **pointillé** qui est une instance de **BasicStroke** créée de la façon suivante :

```
float[] motif = {5.0f};
BasicStroke pointillé = new BasicStroke(2.0f,
                                        BasicStroke.CAP_BUTT,
                                        BasicStroke.JOIN_MITER,
                                        5.0f, motif, 0.0f);
```

Toutes les valeurs doivent être des **float**, ce qui explique l'utilisation de la syntaxe **5.0f**. Le premier paramètre du constructeur de **BasicStroke** dési-

gne l'épaisseur de la ligne. Le deuxième et le troisième indiquent respectivement le type d'extrémité et le type d'angle. Le quatrième paramètre indique la projection maximale des angles. Le cinquième paramètre est un tableau indiquant le motif utilisé pour les pointillés. Ici, nous n'utilisons qu'une seule valeur pour indiquer que la ligne est composée de tirets et d'espaces d'égales valeurs. Si vous voulez un schéma plus complexe, essayez, par exemple :

```
float[] motif = {10.0f, 3.0f, 1.0f, 3.0f}
```

Le dernier paramètre indique le décalage à appliquer pour le départ des pointillés. Notez que nous avons utilisé l'instruction :

```
import java.awt.geom.*;
```

pour importer la classe **Rectangle2D**. Cette classe est une classe abstraite qui ne peut pas être instanciée directement. Nous avons choisi d'instancier **Rectangle2D.Float** qui est une classe interne qui étend **Rectangle2D**. Il existe aussi une classe **Rectangle2D.Double**. La seule différence entre ces deux classes est que le constructeur de la première prend quatre **Float** pour paramètres (coordonnée x et y, largeur et hauteur) alors que la seconde prend des **Double**.

Chaque nouveau tracé efface le précédent car les objets tracés ne sont pas mémorisés. Si nous souhaitions ajouter les tracés les uns aux autres, il nous suffirait de stocker les éléments dans une structure et de retracer chaque élément dans la méthode **paint**. Nous pourrions utiliser pour cela un vecteur dans la mesure où l'ordre de tracé est immuable. Si nous souhaitons pouvoir modifier facilement l'ordre de tracé des éléments, nous pouvons opter pour une liste liée.

L'affichage du texte

Jusqu'ici, nous n'avons affiché du texte que sur la sortie standard, ou au moyen de composants prenant une chaîne de caractères comme paramètre.

Nous pouvons également afficher du texte à l'aide de primitives. La classe **Graphics** dispose des primitives suivantes pour l'affichage du texte :

- **void drawString(String str, int x, int y)**

  Affiche la chaîne de caractères **str** à la position **x, y**. Cette position correspond à la *ligne de base* du premier caractère affiché, comme indiqué sur l'illustration suivante :

- **void drawBytes(byte[] données, int d, int l, int x, int y)**
- **void drawChars(char[] données, int d, int l, int x, int y)**

  Affiche **l** éléments du tableau **données** en commençant à la position **d**. L'affichage est effectué aux coordonnées **x**, **y**.

- **void setFont(Font font)**

  Détermine la police de caractères qui sera utilisée pour les prochains affichages.

- **Font getFont(Font font)**

  Renvoie la police courante.

Les polices de caractères

Pour afficher du texte, il faut disposer d'une police de caractères. Une police de caractères peut être créée en instanciant la classe **Font** au moyen de son constructeur :

```
Font(String nom, int style, int taille)
```

Nom est le nom de la police de caractères. Les polices de caractères disponibles diffèrent d'un environnement à un autre. Afin d'assurer une portabilité maximale, il est conseillé de se limiter aux polices suivantes :

- Serif
- SansSerif
- Monospaced
- Dialog

Il s'agit là de polices génériques qui sont remplacées au moment de l'exécution du programme par une police disponible sur le système, par exemple :

	Windows	*Macintosh et Unix*
Serif	Times New Roman	Times
SansSerif	Arial	Helvetica
Monospaced	Courier New	Courier

Les autres paramètres du constructeur de la classe **Font** peuvent prendre les valeurs suivantes :

style :

- **Font.PLAIN** (normal)
- **Font.BOLD** (gras)
- **Font.ITALIC** (italique)
- **Font.BOLD | Font.ITALIC** (gras et italique)

taille : une valeur quelconque indiquant la taille des caractères en points.

Note : Il s'agit là du point typographique qui n'a rien à voir avec les pixels de l'affichage. La taille en points donne une idée approximative de l'encombrement vertical des caractères, ce qui est différent de leur hauteur. Ainsi, deux polices de même taille peuvent présenter des hauteurs de caractères différentes. C'est le cas, en particulier, des polices *Times* et *Helvetica*. Les mots *Times* et *Helvetica* sont ici affichés avec la même taille (11 points).

Une classe particulièrement importante pour l'utilisation des polices de caractères est la classe **FontMetrics**, qui contient de nombreuses méthodes permettant d'obtenir des informations sur une police de caractères.

Une instance de cette classe peut être créée en passant pour paramètre à son constructeur une instance de **Font**. La classe **FontMetrics** comporte, entre autres, les méthodes suivantes :

- **int charWidth(char car)**
- **int charWidth(int car)**

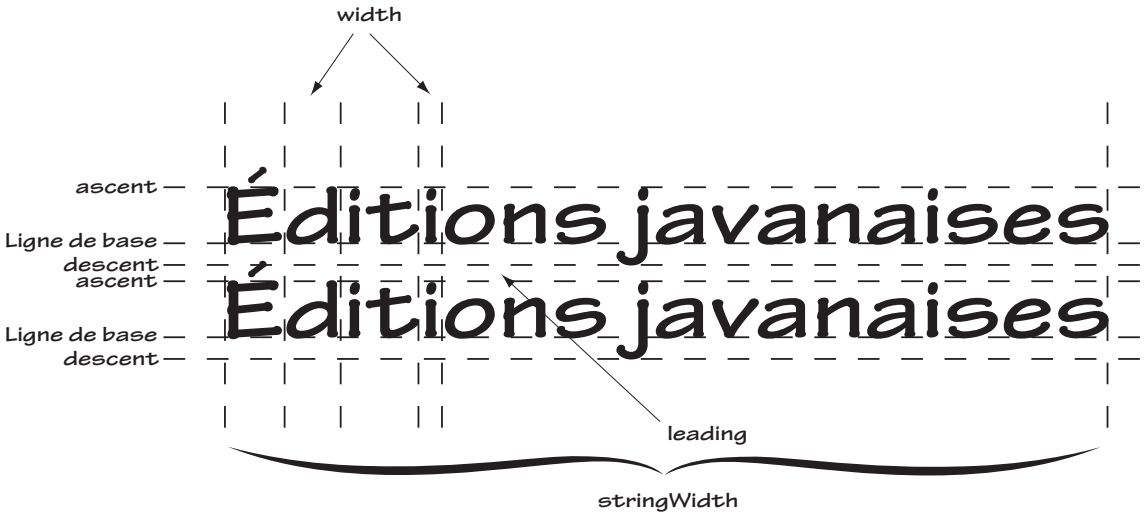
  Renvoie l'encombrement horizontal du caractère.

- **int bytesWidth(byte[] données, int départ, int longueur)**
- **int charsWidth(char[] data, int off, int len)**

  Renvoie l'encombrement horizontal total des caractères.

- **int getAscent()**

  Renvoie la hauteur des caractères "hauts", comme indiqué sur la figure suivante :



Les caractères "haut" sont les capitales (non accentuées) et les lettres telles que *b*, *d*, *f*, *h*, etc.

- **int getMaxAscent()**

  Renvoie la hauteur maximale des caractères, c'est-à-dire, par exemple, celle des capitales accentuées.

- **int getHeight()**

  Renvoie la hauteur normale d'une ligne de texte.

- **int getDescent()**

  Renvoie la hauteur des jambages bas, comme indiqué sur la figure précédente.

- **int getMaxAscent()**

  Renvoie la hauteur maximale des jambages bas.

- **int Leading()**

  Renvoie la hauteur qui sépare les jambages bas d'une ligne (*descent*) des jambages haut de la ligne suivante (*ascent*). Il s'agit donc là de l'interligne au sens typographique du terme (l'espace ajouté entre les lignes et non la distance séparant une ligne de la suivante).

- **int stringWidth(String str)**

  Renvoie l'encombrement horizontal d'une chaîne de caractères dans la police utilisée pour créer l'objet **FontMetrics**.

  La classe **FontMetrics** contient plusieurs autres méthodes dont **getStringBounds** qui permet d'obtenir le rectangle délimitant l'encombrement d'une chaîne de caractères.

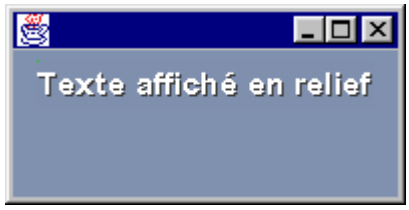
A titre d'exemple, nous allons réaliser un composant permettant d'afficher du texte en relief, pour simuler l'effet obtenu avec une pince "Dymo". (Pour ceux qui ne connaissent pas la marque, il s'agit de ces petites bandes autocollantes de plastique coloré sur lesquelles un texte peut être embossé à l'aide d'une pince. L'embossage étire le plastique qui reste marqué du texte en relief blanc.)

```
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;

public class TexteRelief {
   static Tableau frame;
      public static void main( String[] args ) {
      frame = new Tableau();
      frame.setVisible(true);
   }
}

class Tableau extends JFrame {
   static final Dimension screenSize =
           Toolkit.getDefaultToolkit().getScreenSize();
```

```
   static final int largeurEcran = screenSize.width;
   static final int hauteurEcran = screenSize.height + 2;
   int l = 200;
   int h = 100;
   Color BCouleur = new Color(128, 144, 175);
   Container pane;
   public Tableau() {
      super();
      pane = getContentPane();
      pane.setLayout(new FlowLayout());
      this.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }});
      setBackground(BCouleur);
      setBounds ((largeurEcran - l) / 2,
                        (hauteurEcran - h) / 2, l, h);
      pane.setBackground(BCouleur);


      Label3D label3D =
           new Label3D("Texte affiché en relief", 16);

      pane.add(label3D);
   }
}

class Label3D extends Canvas {
   int
      l = 0,
      h = 0,
      xt = 0,
      yt = 0,
      lt = 0;
   Font police;
    FontMetrics métriques;
   Color
      fCouleurC = new Color(255, 255, 255),
```

```
      fCouleurF = new Color(63, 63, 63),
   String texte;

   Label3D(String s, int taille) {
      police = new Font("SansSerif",
                                  Font.BOLD, taille);
      métriques = getFontMetrics(police);
      texte = s;
      lt = métriques.stringWidth(texte);
      h = métriques.getHeight()
                          + métriques.getLeading();
      yt = métriques.getAscent();
      l = lt + 2;
      setSize(l, h);
   }

   public void paint(Graphics g) {
      g.setFont(police);
      g.setColor(fCouleurF);
      g.drawString(texte, (xt + 1), (yt + 1));
      g.setColor(fCouleurC);
      g.drawString(texte, xt, yt);
   }
}
```

Ce programme affiche le résultat suivant :



La partie intéressante du programme est évidemment la classe **Labe3D**. Cette classe étend la classe **Canvas**. Son constructeur prend deux paramètres : une chaîne de caractères représentant le texte à afficher et une valeur

entière indiquant la taille de caractères à employer. Une police de caractères est tout d'abord créée :

```
police = new Font("SansSerif", Font.BOLD, taille);
```

puis une instance de **FontMetrics** est obtenue au moyen de la méthode **getFontMetrics(police)**. Cette méthode est disponible dans de nombreuses classes, et en particulier dans la classe **Component** dont dérive notre classe par l'intermédiaire de **Canvas** :

```
métriques = getFontMetrics(police);
```

Les différentes dimensions nécessaires sont obtenues à partir de cette instance :

```
lt = métriques.stringWidth(texte);
h = métriques.getHeight() + métriques.getLeading();
yt = métriques.getAscent();
l = lt + 2;
```

lt représente la longueur du texte dans la police choisie, ce qui n'appelle pas de commentaires particuliers. En revanche, le calcul de **h**, représentant la hauteur du composant, et de **yt**, qui représente la coordonnée verticale du texte est un peu plus complexe. **h** est obtenu en additionnant la hauteur du texte et l'interligne, de façon à obtenir un espace suffisant. **yt** est en fait égal à la hauteur des jambages hauts, car il s'agit de la position de la ligne de base du texte comptée depuis le haut du composant qui le contient. Si cela n'est pas clair, reportez-vous aux figures précédentes.

La dernière ligne ajoute deux pixels à la largeur du texte pour obtenir la largeur du composant. Il est nécessaire que le composant soit un peu plus large que le texte en raison du décalage créé pour simuler un effet d'ombre.

La méthode **paint** est très simple :

```
public void paint(Graphics g) {
   g.setFont(police);
   g.setColor(fCouleurF);
   g.drawString(texte, (xt + 1), (yt + 1));
   g.setColor(fCouleurC);
   g.drawString(texte, xt, yt);
}
```

La première ligne sélectionne la police choisie. La deuxième ligne sélectionne la couleur foncée utilisée pour l'ombre du texte. La troisième ligne affiche le texte avec un décalage de 1 point vers la droite et vers le bas. Les deux lignes suivantes affichent le même texte en blanc et sans décalage. L'ensemble donne un effet de relief en simulant un éclairage provenant de l'angle supérieur gauche.

A l'Annexe A, nous verrons un exemple plus complexe permettant d'obtenir le même résultat avec un texte de plusieurs lignes.

Les images

Un autre aspect du graphisme consiste à afficher des images. En Java, l'affichage d'images s'effectue très simplement à l'aide de la primitive **drawImage**. Cette primitive possède plusieurs formes, dont la plus simple est :

```
boolean drawImage(Image img, int x, int y, ImageObserver observer)
```

img est l'image à afficher. **x** et **y** sont les coordonnées de l'angle supérieur gauche de l'image. **observer** est une instance d'**ImageObserver**. Nous verrons plus loin à quoi servent les **ImageObserver**. Pour l'instant, sachez qu'il s'agit d'une interface implémentée, entre autres, par la classe **Component**. Nous pourrons donc utiliser le composant dans lequel nous affichons l'image comme instance d'**ImageObserver**.

Cette méthode est une méthode asynchrone, c'est-à-dire qu'elle retourne immédiatement sans attendre que l'image soit affichée. Dans le cas d'une image locale, cela peut ne pas poser de problème. En revanche, dans le cas d'une image devant être obtenue d'un serveur sur un réseau, le chargement de l'image peut demander un certain temps. Un **ImageObserver** permet de contrôler le chargement des images.

Obtenir une image

Le paramètre le plus important est évidemment l'image à afficher. Une image peut être obtenue de plusieurs façons. Le plus souvent, elle le sera au moyen de la méthode **getImage**. Cette méthode est implémentée de façon différente dans la classe **java.awt.ToolKit** (accessible aux applications) et dans la classe **java.Applet** (la seule pouvant être utilisée par les applets). En effet, les applets sont conçues pour fonctionner dans une page HTML à partir d'un serveur. De ce fait, elles supportent de nombreuses limitations pour des raisons de sécurité. Nous en apprendrons plus sur les applets dans le prochain chapitre.

Pour obtenir une image dans une application, nous pouvons utiliser les méthodes :

- **getToolkit.getImage(String fichier)**
- **getToolkit.getImage(URL url)**

La méthode **getToolkit()** est implémentée (entre autres) dans la classe **Component**.

Dans une applet, nous utiliserons les méthodes :

- **getImage(URL url)**
- **getImage(URL url, String nom)**

Ces méthodes ne créent qu'un handle d'image mais n'effectuent aucun chargement. Le chargement n'est effectué que lorsque la méthode **drawImage** est invoquée. Le programme suivant affiche une image :

```
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;

public class AfficheImage {
   static Tableau frame;
      public static void main( String[] args ) {
      frame = new Tableau();
      frame.setVisible(true);
   }
}

class Tableau extends JFrame {

   static final Dimension screenSize =
           Toolkit.getDefaultToolkit().getScreenSize();
   static final int largeurEcran = screenSize.width;
   static final int hauteurEcran = screenSize.height + 2;
   int l = 400;
   int h = 300;
   Container pane;

   public Tableau() {
      super();
      pane = getContentPane();
      pane.setLayout(new FlowLayout());
      this.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }});
      setBounds ((largeurEcran - l) / 2,
                        (hauteurEcran - h) / 2, l, h);

      PhotoCanvas photo =
                        new PhotoCanvas("voiture.jpg");
      pane.add(photo);
   }
}
```

```
class PhotoCanvas extends Canvas {
   Image image;

   public PhotoCanvas(String s) {
      setSize(400,300);
      image = getToolkit().getImage(s);
   }

   public void paint(Graphics g) {
      g.drawImage(image, 0, 0, this);
   }
}
```

Le nom du fichier est passé en paramètre au constructeur de la classe **PhotoCanvas**. Le fichier doit se trouver dans le même répertoire que l'application. Dans le cas contraire, il est possible de passer au constructeur un chemin d'accès complet. Ce programme affiche le résultat suivant (le fichier *voiture.jpg* se trouve sur le CD-ROM accompagnant ce livre) :

Surveiller le chargement d'une image

Il est souvent nécessaire d'attendre que l'image soit chargée pour effectuer un traitement. Cela est d'autant plus fréquent lorsque l'image se trouve sur un serveur accessible par l'intermédiaire d'un réseau, mais cela peut aussi être le cas pour une image locale. Par exemple, si nous voulons connaître la taille de l'image afin de dimensionner le canvas en conséquence, il faut attendre que l'image soit chargée. En effet, si nous tentons d'utiliser les méthodes disponibles dans la classe **Image** pour connaître la hauteur et la largeur de celle-ci, nous n'obtiendrons pas le résultat escompté. Par exemple, si nous modifions la classe **PhotoCanvas** de la façon suivante :

```
class PhotoCanvas extends Canvas {
   Image image;

   public PhotoCanvas(String s) {
      image = getToolkit().getImage(s);
      setSize(image.getWidth(this),
                           image.getHeight(this));
   }

   public void paint(Graphics g) {
      g.drawImage(image, 0, 0, this);
   }
}
```

le programme n'affiche rien du tout. Pour comprendre ce qui se passe, nous pouvons ajouter les lignes :

```
public PhotoCanvas(String s) {
   image = getToolkit().getImage(s);
   System.out.println(image.getWidth(this));
   System.out.println(image.getHeight(this));
   setSize(image.getWidth(this),
                        image.getHeight(this));
}
```

Le programme affiche alors sur la sortie standard :

```
-1
-1
```

La documentation nous apprend que les méthodes **getHeight()** et **getWidth()** renvoient -1 si l'image n'est pas encore disponible.

Une solution consiste à utiliser la classe **MediaTracker**. Le programme ci-après montre les modifications apportées pour que la fenêtre soit redimensionnée à la taille de l'image :

```
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;

public class AfficheImage2 {
   static Tableau frame;
      public static void main( String[] args ) {
      frame = new Tableau();
      frame.setVisible(true);
   }
}

class Tableau extends JFrame {

   static final Dimension screenSize =
          Toolkit.getDefaultToolkit().getScreenSize();
   static final int largeurEcran = screenSize.width;
   static final int hauteurEcran = screenSize.height + 2;
   Container pane;
   int l, h;

   public Tableau() {
      super();
      pane = getContentPane();
      pane.setLayout(new FlowLayout());
```

```
      this.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }});

      PhotoCanvas photo =
                        new PhotoCanvas("voiture.jpg");
      l = photo.getWidth();
      h = photo.getHeight();
      setBounds((largeurEcran - l) / 2,
                        (hauteurEcran - h) / 2, l, h);
      pane.add(photo);
   }
}

class PhotoCanvas extends Canvas {
   Image image;

   public PhotoCanvas(String s) {
      image = getToolkit().getImage(s);
      MediaTracker tracker = new MediaTracker(this);
      tracker.addImage(image, 0);
      try {
         tracker.waitForAll();
      }
      catch(InterruptedException e) {
         System.out.println(e);
      }
      setSize(image.getWidth(this),
                                    image.getHeight(this));
   }

   public void paint(Graphics g) {
      g.drawImage(image, 0, 0, this);
   }
}
```

Nous instancions la classe **MediaTracker** en passant pour paramètre à son constructeur le composant pour le compte duquel les images doivent être surveillées. Nous ajoutons ensuite au tracker l'image à surveiller en lui attribuant un numéro (ici, 0). Les images sont chargées par le tracker dans l'ordre ascendant des numéros. Nous utilisons ensuite la méthode **waitForAll()** qui, sans argument, attend que toutes les images soient chargées. Nous aurions pu utiliser **waitForID**, qui prend pour argument un numéro d'ordre et attend que toutes les images portant ce numéro soient chargées. Ces deux méthodes peuvent également prendre un second argument indiquant le temps limite que doit durer l'attente.

De cette façon, les méthodes **getWidth** et **getHeight** ne sont invoquées que lorsque l'image est chargée. Elles donnent donc le résultat correct. La classe **Tableau** est également modifiée pour que la taille de la fenêtre soit adaptée à celle du canvas.

Cette méthode présente un inconvénient. Avec une image locale, il n'y a pas de problème. En revanche, avec une image obtenue d'un serveur sur un réseau, le délai de chargement peut être beaucoup plus long. Pendant ce temps, le programme est bloqué. Une autre façon de procéder consiste à demander au programme de nous avertir lorsque les informations sont disponibles. De cette façon, le programme peut continuer son travail en attendant que le chargement s'effectue.

Une autre amélioration consisterait à ne pas attendre le chargement total de l'image. En effet, il nous suffit de connaître ses dimensions. La classe **ImageObserver** permet d'obtenir le résultat souhaité.

Comme nous l'avons vu précédemment, la classe **Component** implémente l'interface **ImageObserver**. Cette interface déclare une seule méthode :

```
boolean imageUpdate(Image img, int info, int x, int y, int l, int h)
```

Lorsqu'un **ImageObserver** est associé à une image, sa méthode **imageUpdate** est appelée chaque fois que de nouvelles informations concernant l'image sont disponibles. Ces informations sont représentées par

des valeurs entières (**int**) qui doivent être utilisées comme des indicateurs binaires. Elles appartiennent aux catégories suivantes :

- **ABORT** : Le chargement a été interrompu.
- **ALLBITS** : Toutes les données ont été chargées.
- **ERROR** : Une erreur s'est produite pendant le chargement.
- **FRAMEBITS** : Une image complète supplémentaire a été chargée. Cet indicateur concerne les images composées de plusieurs images (ou *frames*) telles que les images GIF animées.
- **HEIGHT** : La hauteur de l'image est disponible.
- **PROPERTIES** : Les propriétés de l'image sont disponibles.
- **SOMEBITS** : Des données supplémentaires sont disponibles.
- **WIDTH** : La largeur de l'image est disponible.

Ces informations peuvent être exploitées en effectuant un ET logique avec le deuxième paramètre (**info**) passé à la méthode **imageUpdate**. Par exemple, si nous voulons savoir si l'image est entièrement chargée, nous pouvons tester l'expression logique suivante :

```
if ((info & ALLBITS) != 0)
```

Chacune des constantes (**ABORT**, **ALLBITS**, **ERROR**, etc.) est une valeur de type **int** comportant un seul bit valant 1, tous les autres valant 0. (Exprimé d'une autre manière, chaque constante est une puissance de 2.) L'expression **info & ALLBITS** vaut **true** si le bit qui vaut 1 dans **ALLBITS** vaut également 1 dans **info**.

Pour tester si la hauteur et la largeur sont disponibles, nous pouvons utiliser l'expression :

```
if ((info & (WIDTH | HEIGHT)) != 0)
```

Si cette formulation ne vous semble pas évidente, souvenez-vous que **info**, **WIDTH** et **HEIGHT** ne sont pas des **boolean** mais des **int**.

Nous pouvons donc modifier notre programme de la façon suivante :

```
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
import java.awt.image.*;
public class AfficheImage3 {
   static Tableau frame;
      public static void main( String[] args ) {
      frame = new Tableau();
      frame.setVisible(true);
   }
}
class Tableau extends JFrame {
   Container pane;
   public Tableau() {
      super();
      pane = getContentPane();
      pane.setLayout(new FlowLayout());
      this.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }});
      PhotoCanvas photo = new PhotoCanvas("voiture.jpg");
      pane.add(photo);
   }

}
class PhotoCanvas extends Canvas {
   static final Dimension screenSize =
                  Toolkit.getDefaultToolkit().getScreenSize();
   static final int largeurEcran = screenSize.width;
   static final int hauteurEcran = screenSize.height + 2;
   Image image;
```

```
   public PhotoCanvas(String s) {
      image = getToolkit().getImage(s);
      prepareImage(image, this);
   }
   public void paint(Graphics g) {
      g.drawImage(image, 0, 0, this);
   }
   public boolean imageUpdate(Image image, int info, int x,
                                          int y, int l, int h) {
      if ((info & (WIDTH | HEIGHT)) != 0) {
         setSize(l, h);
         getParent().getParent().getParent().getParent().
                              setBounds((largeurEcran - l) / 2,
                              (hauteurEcran - h) / 2, l, h);
      }
      if ((info & (ERROR | FRAMEBITS | ALLBITS)) != 0) {
         repaint();
         return false;
      }
      else {
         return true;
      }
   }
}
```

La lecture des dimensions de l'écran a été déplacée dans la classe **PhotoCanvas**. L'instruction :

```
prepareImage(image, this);
```

attribue l'**ImageObserver** à l'image et lance son chargement. La méthode **imageUpdate** teste les indicateurs pour savoir si la hauteur et la largeur de l'image sont disponibles. Si c'est le cas, la taille du canvas est ajustée au moyen de la méthode **setSize**. La taille et la position de la fenêtre sont ensuite ajustées. Notez l'utilisation de la syntaxe :

```
getParent().getParent().getParent().getParent().setBounds(
```

En effet, le conteneur parent du canvas est la contentPane de la fenêtre. Le parent de celle-ci est la layeredPane, dont le parent est la JRootPane. Le parent de la JRootPane est l'instance de **Tableau** que notre programme a créée.

La méthode teste ensuite les indicateurs **ERROR**, **FRAMEBITS** et **ALLBITS** pour savoir si le chargement est terminé. Si c'est le cas, la méthode **repaint()** est appelée et la méthode renvoie **false**, ce qui indique à l'appelant que l'opération n'est pas terminée et que la méthode doit de nouveau être appelée lorsque de nouvelles données seront disponibles.

Si l'opération est terminée, la méthode renvoie **true** pour indiquer que la méthode **imageUpdate** ne doit plus être appelée.

Conclusion

Il y aurait encore beaucoup à dire en ce qui concerne le graphisme, et en particulier sur l'animation, la création d'images et le filtrage. Java dispose de nombreuses classes permettant d'effectuer des traitements d'images complexes tels que renforcement, atténuation ou rotation. Par ailleurs, Sun Microsystems propose également une API permettant de traiter les images 3D. La place nous manque malheureusement pour traiter ici ces sujets. Ils le seront dans un prochain ouvrage.

Chapitre 20

Applets et réseaux

Jusqu'ici, tous les programmes que nous avons réalisés étaient des applications. Certaines produisaient un affichage sur la sortie standard, d'autres utilisaient une interface fenêtrée. Toutes étaient conçues pour fonctionner localement sur votre ordinateur. Cependant, un des points forts de Java est son aptitude à produire des programmes pouvant être chargés et exécutés depuis un serveur, à travers un réseau.

Les applets

Les applets sont des programmes Java présentant un certain nombre de particularités :

- Ils utilisent une interface fenêtrée affichée dans une zone rectangulaire d'une page HTML.
- Ils disposent d'un contexte d'exécution fourni par l'application qui affiche la page HTML (en général, un navigateur Web).
- Par mesure de sécurité, ils sont soumis à des restrictions d'accès aux ressources.
- Ils n'exécutent pas automatiquement la méthode **main** mais la méthode **init**, qui n'est pas statique.

Création d'une applet

Une applet est une classe étendant la classe **java.awt.Applet** ou une classe dérivée, comme **javax.swing.JApplet**. Au Chapitre 2, nous avons créé une première applet qui étendait la classe **Applet**. Voici la même applet mise à jour pour utiliser les composants Swing :

```
import javax.swing.*;
public class PremiereApplet extends JApplet {
   public void init() {
      getContentPane().add(new JLabel("Ça marche !"));
   }
}
```

Et voici le fichier HTML permettant d'exécuter l'applet :

```
<html>
   <body>
      <applet code="PremiereApplet" width="200" height="150">
      </applet>
   </body>
</html>
```

Vous pouvez ajouter entre les balises **<applet>** et **</applet>** un texte qui sera ignoré par les navigateurs compatibles Java et affiché par les navigateurs non compatibles. Par exemple :

```
<html>
   <body>
      <applet code="PremiereApplet" width="200" height="150">
      Votre navigateur doit être compatible Java pour voir cette page.
      </applet>
   </body>
</html>
```

De cette façon, les utilisateurs dont le navigateur n'est pas compatible ou qui ont désactivé Java ne seront pas déconcertés.

Problèmes de compatibilité entre les versions

Autant les problèmes de compatibilité entre les différentes versions de Java sont relativement simples à gérer en ce qui concerne les applications, autant c'est un véritable casse-tête américain (les chinois n'y sont pour rien) quand il s'agit des applets. Ici, le slogan *Write once, run everywhere* relève de la plus pure utopie. En effet, l'immense majorité des navigateurs compatibles Java est à ce jour compatible avec la version 1.0. Or, la version 1.0 n'utilise pas du tout le même modèle pour la gestion des événements. Le modèle actuel n'est disponible que depuis la version 1.1. Microsoft Explorer est compatible avec Java 1.1 depuis la version 4. En revanche, la version 4 de Netscape Navigator n'est compatible avec Java 1.1 qu'avec une mise à jour.

Vous pensez peut-être que vos soucis s'arrêtent là ? Pas du tout ! La bibliothèque Swing ne fait pas partie intégrante de Java 1.1. Elle doit être installée séparément. Par conséquent, elle n'est généralement pas prise en charge par les navigateurs compatibles Java 1.1, sauf si leurs propriétaires ont pris soin de l'installer séparément.

Vous comptez peut-être alors demander aux Internautes auxquels vous destinez vos applets d'installer la bibliothèque Swing ? Bon courage ! Tout d'abord, il est très difficile de demander aux utilisateurs d'installer quoi que

ce soit. De plus, même avec la bibliothèque Swing installée, la version 4 des deux navigateurs précités produit immanquablement une erreur lorsqu'une applet tente d'accéder à la *queue* des événements AWT (la *queue* est une structure dans laquelle sont stockés les événements à traiter). Or, les applets Swing accèdent systématiquement à cette queue pour vérifier si l'accès est autorisé. La documentation de Java présente ce problème en l'attribuant à une mauvaise conception des navigateurs, comme s'il était normal, pour vérifier qu'une opération est autorisée, de tenter de l'exécuter. Cette méthode est pour le moins cavalière. En attendant que les navigateurs s'adaptent, il ne reste qu'à modifier le comportement des applets Swing, ce que vous pouvez effectuer en ajoutant à celles-ci la ligne suivante :

```
getRootPane().putClientProperty("defeatSystemEventQueueCheck",
                                                    Boolean.TRUE);
```

Une fois ce problème réglé, obtenir des utilisateurs qu'ils effectuent la mise à jour de leurs navigateurs n'est pas une mince affaire. Une lueur d'espoir cependant : la version 2 de Java est livrée avec un programme permettant de mettre à jour les navigateurs à l'aide d'un plug-in qui permet à ceux-ci d'utiliser le JRE installé par ailleurs, plutôt que leur propre JVM, ce qui est beaucoup plus simple que d'installer une nouvelle version. Malheureusement, cela ne concerne que les versions Windows des navigateurs.

Si vous développez des applets pour une diffusion sur Internet, nous vous conseillons de vous limiter pour l'instant à la bibliothèque AWT et d'ignorer les composants Swing. C'est un juste milieu provisoire qui vous empêche toutefois d'atteindre les utilisateurs de navigateurs compatibles Java 1.0. Si vous voulez une compatibilité maximale, vous devrez apprendre à utiliser l'ancien modèle des événements, que nous n'avons pas abordé dans ce livre. Vous trouverez en librairie des dizaines de titres traitant de ce sujet. Vous aurez ainsi l'occasion de vérifier à quel point le nouveau modèle constitue un progrès pour le programmeur.

Si vous développez des applets pour le plaisir (on ne sait jamais !) et si vous vous contentez de les visualiser à l'aide de l'AppletViewer, vous pouvez utiliser les composants Swing sans problème. Il en est de même si vous utilisez une version compatible Java 2 du navigateur HotJava ou un navigateur équipé du plug-in Java 2

Avertissement

La mise au point d'applets avec un navigateur est une opération pénible. En effet, la plupart des navigateurs ne rechargent pas les classes Java lorsqu'elles ont été modifiées. Il vous arrivera donc probablement plusieurs fois de vous arracher les cheveux en voyant une applet que vous venez de modifier continuer de produire toujours le même résultat. En effet, contrairement aux textes et aux images contenus dans les pages Web, le fait d'actualiser une page ne recharge pas les classes Java utilisées par les applets. La seule façon sûre de procéder consiste à quitter le navigateur et à le relancer.

Deuxième avertissement

Ne croyez pas que vous puissiez utiliser la classe **Applet** provisoirement en attendant de la remplacer pas **JApplet** lorsque la majorité des utilisateurs seront équipés d'une version compatible, sans vous soucier aujourd'hui des conséquences. Par exemple, si vous utilisez dans un composant la méthode **getParent()**, celle-ci donnera un résultat complètement différent avec une **Applet** et avec une **JApplet**. En effet, le parent d'un composant dans une **Applet** sera, par exemple, l'applet elle-même. Avec une **JApplet**, le parent sera la *contentPane* de l'applet. Pour remonter du composant à l'applet, il faudra utiliser la forme :

```
getParent().getParent().getParent().getParent()
```

pour remonter à la *contentPane*, puis à la *layeredPane*, puis à la **JRootPane** et enfin à l'applet. Bien sûr, il vous faudra remplacer tous les **add()** par **getContentPane().add**, mais cela est un moindre mal.

Un autre problème est que les **Applet** et les **JApplet** n'ont pas le même layout manager par défaut. **Applet** utilise le **FlowLayout** alors que **JApplet** préfère le **BorderLayout**. De même, la police par défaut des composants, les couleurs de fond et de premier plan, la taille des caractères, sont différentes. Si vous voulez vous assurer une future migration sans trop de problèmes, ne laissez rien au hasard et spécifiez tous les éléments sans jamais vous reposer sur les valeurs par défaut.

Fonctionnement d'une applet

Le fonctionnement d'une applet est un peu plus complexe que celui d'une application. Cependant, son modèle est conforme à ce que nous connaissons de Java. Une applet contient des méthodes qui sont appelées automatiquement lorsque certains événements se produisent. Jusqu'ici, nous n'avons vu que la méthode **init()**, qui est appelée lorsque l'applet est chargée. En fait, l'invocation de la méthode **init()** n'est pas la seule chose qui se produise au chargement de l'applet. En effet, cette méthode, contrairement à la méthode **main()** d'une application, n'est pas statique. Une instance de la classe chargée doit donc être créée. La liste ci-dessous décrit les différents événements qui se produisent lorsqu'une applet est chargée :

- La classe est chargée.
- La classe est instanciée.
- La méthode **init()** de l'instance créée est invoquée.
- La méthode **start()** est invoquée.

Si la page contenant l'applet est quittée :

- La méthode **stop()** est invoquée.

Si la page contenant l'applet est de nouveau affichée :

- La méthode **start()** est invoquée de nouveau.

Lorsque le navigateur est quitté :

- La méthode **destroy()** est invoquée.

Il n'est pas obligatoire de redéfinir chacune de ces méthodes. Cependant il est des cas où il est important de le faire. Si une applet lance un thread, celui-ci continue de s'exécuter lorsque la page n'est plus affichée. Cela n'est pas toujours nécessaire et occupe inutilement le temps du processeur. La

méthode **stop()** peut donc être employée pour stopper le thread alors que la méthode **start()** servira à le relancer lorsque la page sera affichée de nouveau.

Il n'est généralement pas nécessaire de redéfinir la méthode **destroy()** car le navigateur se charge du nettoyage nécessaire.

Passer des paramètres à une applet

Le fichier HTML que nous avons utilisé était réduit à sa plus simple expression. Les seuls paramètres que nous avons fournis à l'applet étaient le nom de la classe (**code**), la hauteur (**height**) et la largeur (**width**) de la zone qu'elle devait occuper. Ces deux paramètres sont obligatoires. En revanche, il est possible d'en préciser d'autres :

```
align = left | right | top | texttop | middle | absmiddle
| baseline| bottom | absbottom
```

Ce paramètre précise l'alignement de l'applet dans la zone qui lui est réservée.

```
alt = texte
```

Un texte qui sera affiché pendant le chargement de l'applet, avant que celle-ci soit disponible. Utiliser ce paramètre permet d'indiquer aux utilisateurs ce pour quoi ils sont en train de patienter. Ce texte sera également affiché à la place de l'applet si le navigateur reconnaît le tag **<applet>** mais que Java a été désactivé.

```
codebase = url du répertoire contenant le fichier .class
```

Ce paramètre précise l'URL du répertoire contenant le fichier **.class**, dans le cas où celui-ci ne se trouverait pas à la même adresse que le fichier HTML.

```
archive = "fichier1, fichier2, ...."
```

spécifie les fichiers d'archives utilisés par l'applet (voir plus loin).

```
hspace = espace vide à gauche de l'applet
```

Ce paramètre précise la valeur de la marge gauche dans la zone réservée à l'applet.

```
vspace = espace vide au-dessus de l'applet
```

Ce paramètre précise la valeur de la marge haute dans la zone réservée à l'applet.

Tous ces paramètres doivent être placés à l'intérieur du tag **<applet>**, comme dans l'exemple :

```
<applet code="PremiereApplet" width="200" height="150">
```

D'autres paramètres définissables par le programmeur peuvent être placés entre les tags **<applet>** et **</applet>** sous la forme :

```
<param nom = paramètre1 value = valeur1>
<param nom = paramètre2 value = valeur2>
```

Les valeurs ainsi fournies sont des chaînes de caractères. Par exemple, l'applet suivante utilise un paramètre appelé **texte** pour obtenir le texte qu'elle doit afficher :

```
import java.applet.*;
import java.awt.*;
public class DeuxiemeApplet extends Applet {
   public void init() {
      String texte = getParameter("texte");
      add(new Label(texte));
   }
}
```

Le fichier HTML qui permet d'afficher l'applet doit fournir le paramètre :

```
<html>
   <body>
      <applet code="DeuxiemeApplet" width="200" height="150">
         <param name = "texte" value = "Le texte à afficher">
      </applet>
   </body>
</html>
```

Le résultat obtenu avec l'AppletViewer est le suivant :

L'utilisation des paramètres permet d'employer plusieurs fois une même applet en ne la chargeant qu'une seule fois. L'exemple suivant affiche un texte dans un rectangle qui change de couleur lorsque le pointeur y entre. Il peut être utilisé comme base pour un bouton animé :

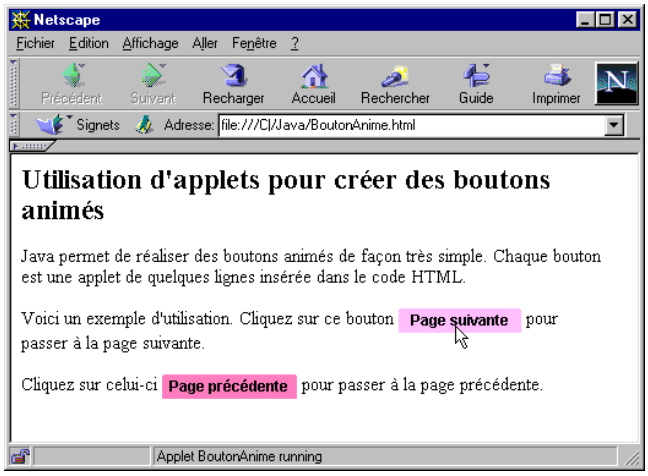
```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class BoutonAnime extends Applet {
   String texte;
   final Color couleur1 = new Color(255, 127, 192);
   final Color couleur2 = new Color(255, 192, 255);
   final Color couleurTexte = new Color(0, 0, 0);
   Color couleur = couleur1;
   int largeur, hauteur, taille, x, y;
   Font police;
   FontMetrics métriques;

   public void init() {
      texte = getParameter("texte");
      largeur = getSize().width;
      hauteur = getSize().height;
      taille = Integer.valueOf(getParameter("taille")).intValue();
      police = new Font("Dialog", Font.BOLD, taille);
      métriques = getFontMetrics(police);
      x = (int)((largeur - métriques.stringWidth(texte)) / 2);
      y = (int)(((hauteur - métriques.getHeight()) / 2)
                                        + métriques.getAscent());
      addMouseListener(new MouseAdapter() {
           public void mouseEntered(MouseEvent e) {
              couleur = couleur2;
              repaint();
           }
           public void mouseExited(MouseEvent e) {
              couleur = couleur1;
              repaint();
           }
        });
   }
   public void paint(Graphics g) {
      g.setFont(police);
      g.setColor(couleur);
```

```
      g.fillRoundRect(0, 0, largeur, hauteur, 5, 5);
      g.setColor(couleurTexte);
      g.drawString(texte, x, y);
   }
}
```

Ce programme est incomplet car l'applet ne fait rien d'autre que changer de couleur lorsque le pointeur y entre ou en sort. Vous pouvez utiliser cette applet dans un document HTML de la façon suivante :

```
<html>
<body>
<h2>Utilisation d'applets pour créer des boutons animés</h2>
<p>Java permet de réaliser des boutons animés de façon
très simple. Chaque bouton est une applet de quelques lignes
insérée dans le code HTML.</p>

<p> Voici un exemple d'utilisation. Cliquez sur ce bouton

<applet code="BoutonAnime" width="100" height="20" align = "top">
   <param name = "texte" value = "Page suivante">
   <param name = "adresse" value = "Page2.html">
   <param name = "taille" value = "12">
</applet>

pour passer à la page suivante.</p>
<p>Cliquez sur celui-ci

<applet code="BoutonAnime" width="110" height="20" align = "top">
   <param name = "texte" value = "Page précédente">
   <param name = "adresse" value = "Page1.html">
   <param name = "taille" value = "12">
</applet>

pour passer à la page précédente.</p>
</body>
</html>
```

Voici le résultat obtenu avec Netscape Navigator :



Agir sur le navigateur

Une applet est en mesure d'agir sur le navigateur qui l'affiche de deux façons. La première consiste à obtenir du navigateur des informations. C'est ce que nous avons fait jusqu'ici en utilisant les paramètres fournis dans la page HTML. L'autre type d'interaction consiste à indiquer au navigateur quelle page il doit afficher. C'est ce qu'est supposé faire notre applet.

Afficher un nouveau document

Pour cela, nous devons obtenir du navigateur le contrôle du *contexte d'exécution* de l'applet. Pour que nos boutons soient fonctionnels, nous devons implémenter dans notre programme les étapes suivantes :

- Lire le paramètre indiquant la destination.
- Créer un *URL* à partir de ce paramètre. (Je sais bien que tout le monde dit "une URL", mais il n'y a aucune raison valable à cela. *URL* signifie *Uniform Resource Locator.* Le mot *locator* n'a aucune raison d'être féminin.)

- Obtenir le contexte d'exécution.
- Exécuter la commande permettant de charger le document recherché.

La lecture du paramètre ne pose aucun problème. Nous savons déjà faire cela :

```
String adresse;
adresse = getParameter("adresse");
```

La transformation de la chaîne de caractères obtenue en URL valide ne pose pas de problème si vous disposez de la documentation de Java. Celle-ci nous indique que le constructeur de la classe **URL** peut être invoqué avec pour paramètre un URL et une chaîne de caractères contenant le nom du document. Si le document à afficher se trouve au même endroit que le document contenant le bouton, il suffit d'utiliser la syntaxe suivante :

```
url = new URL(getDocumentBase(), adresse);
```

Le contexte d'exécution peut être obtenu à l'aide de la méthode **getAppletContext()**, qui renvoie un objet du type **AppletContext**. Cette classe contient la méthode **showDocument(URL)** qui peut être invoquée pour afficher le document demandé.

Cependant, cette façon de procéder nous oblige à créer explicitement un URL, et donc à intercepter l'exception **MalformedURLException**. Aussi, il est plus simple d'employer la forme **showDocument(URL, String)**. C'est ce que fait le programme suivant :

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
public class BoutonAnime2 extends Applet {
   String texte;
   final Color couleur1 = new Color(255, 127, 192);
```

```
final Color couleur2 = new Color(255, 192, 255);
final Color couleurTexte = new Color(0, 0, 0);
Color couleur = couleur1;
int largeur, hauteur, taille, x, y;
Font police;
FontMetrics métriques;
String adresse;
URL url;
public void init() {
   texte = getParameter("texte");
   adresse = getParameter("adresse");
   try {
      url = new URL(getDocumentBase(), adresse);
   }
   catch(MalformedURLException e) {
   }
   largeur = getSize().width;
   hauteur = getSize().height;
   taille = Integer.valueOf(getParameter("taille")).intValue();
   police = new Font("Dialog", Font.BOLD, taille);
   métriques = getFontMetrics(police);
   x = (int)((largeur - métriques.stringWidth(texte)) / 2);
   y = (int)(((hauteur - métriques.getHeight()) / 2)
                                    + métriques.getAscent());
   addMouseListener(new MouseAdapter() {
        public void mouseEntered(MouseEvent e) {
           couleur = couleur2;
           repaint();
        }
        public void mouseExited(MouseEvent e) {
           couleur = couleur1;
           repaint();
        }
        public void mouseClicked(MouseEvent e) {
           getAppletContext().showDocument(url);
        }
     });
   }
```

```
    public void paint(Graphics g) {
       g.setFont(police);
       g.setColor(couleur);
       g.fillRoundRect(0, 0, largeur, hauteur, 5, 5);
       g.setColor(couleurTexte);
       g.drawString(texte, x, y);
    }
}
```

Attention : Si vous essayez ce programme tel quel, n'oubliez pas de modifier le nom de la classe dans le fichier HTML (**BoutonAnime2.class**). Par ailleurs, ce programme ne traite pas les conditions d'erreur telles que texte trop long pour les boutons ou documents inexistants.

Note : Il existe également une méthode **showDocument(URL url, String cible)** qui prend comme deuxième paramètre une chaîne de caractères indiquant dans quelle fenêtre le document doit être affiché. Les valeurs possibles sont :

- *nom_de_fenêtre* : le document est affiché dans une nouvelle fenêtre portant le nom indiqué.
- **_blank** : le document est affiché dans une nouvelle fenêtre sans nom.
- **_self** : le document est affiché dans la même fenêtre ou dans le même cadre.
- **_parent** : le document est affiché dans la fenêtre ou le cadre parent. S'il n'y a pas de fenêtre parente, il est affiché dans la même fenêtre ou le même cadre.
- **_top** : le document est affiché dans la fenêtre de niveau supérieur, ou dans le même cadre/fenêtre si on se trouve déjà au niveau supérieur.

Afficher un message dans la barre d'état

La plupart des navigateurs affichent l'adresse de destination d'un lien dans la barre d'état chaque fois que le pointeur est placé sur le lien. Nous pou-

vons très facilement obtenir le même résultat en modifiant notre programme de la façon suivante :

```
public void mouseEntered(MouseEvent e) {
   couleur = couleur2;
   showStatus(url.toString());
   repaint();
}
public void mouseExited(MouseEvent e) {
   couleur = couleur1;
   showStatus("");
   repaint();
}
```

Afficher des images

Une applet est soumise à de nombreuses restrictions d'accès. En particulier, elle ne peut accéder à des fichiers se trouvant ailleurs qu'à l'endroit d'où elle a été chargée. De plus, elle ne peut obtenir librement toutes les informations concernant la machine sur laquelle elle fonctionne. En conséquence, la procédure à suivre pour charger une image est un peu différente de celle mise en œuvre pour une application. En effet, l'instruction :

```
image = getToolkit().getImage(String nom);
```

doit être remplacée par :

```
image = getImage(URL url , String nom)
```

Si les images sont stockées dans le même répertoire que l'applet, vous n'aurez pas besoin de construire un URL et pourrez employer la syntaxe suivante :

```
image = getImage(getCodeBase(), "image.jpg")
```

Vous pouvez également placer les images dans un sous-répertoire et utiliser la syntaxe :

```
image = getImage(getCodeBase(), "images/image.jpg")
```

Note : Vous ne pouvez pas utiliser directement **getDocumentBase()** car cette méthode renvoie l'URL du document alors que **getCodeBase()** renvoie l'URL du répertoire dans lequel se trouve l'applet.

Les sons

Les possibilités sonores de Java sont extrêmement limitées. Elles se bornent à la lecture des fichiers aux formats *au, wav, aiff, midi* et *rmf*. La lecture d'un fichier s'effectue en créant tout d'abord un *clip*. Un clip est une instance de la classe **AudioClip**. Vous pouvez obtenir un clip à l'aide de la méthode **getAudioClip** de la classe **Applet** :

```
AudioClip monClip = new AudioClip(url)
```

ou :

```
AudioClip monClip = new AudioClip(url, chaîne)
```

La syntaxe est identique à celle employée pour les images. Par exemple, pour créer un clip avec le fichier **musique.au** se trouvant dans le sous-répertoire **sons** du répertoire contenant l'applet, vous utiliserez la syntaxe :

```
AudioClip monClip = new AudioClip(getCodeBase(), "sons/musique.au")
```

Cette méthode ne peut être invoquée que depuis une instance d'**Applet**. Aussi, il existe également la méthode statique **newAudioClip** qui permet de créer un clip depuis une application :

```
AudioClip monClip = Applet.newAudioClip(URL url)
```

Le clip obtenu peut être joué de la façon suivante :

- **monClip.play()** joue le clip une fois.
- **monClip.loop()** joue le clip continuellement.
- **monClip.stop()** arrête le clip.

Ces méthodes proviennent de la classe **AudioClip**. Il existe également dans la classe **Applet** les méthodes :

```
play(URL url)
play(URL url, String nom)
```

qui permettent de jouer un fichier son sans créer un clip.

Bien que les fonctions sonores de Java soient extrêmement limitées, elles offrent cependant la possibilité de mixer automatiquement plusieurs clips.

Remarque : Si vous décidez de jouer un clip en continu lorsque votre applet est affiché, n'oubliez pas que son exécution continuera jusqu'à ce que l'utilisateur quitte le navigateur. Par conséquent, il est fortement conseillé, dans ce cas, d'implémenter la méthode **stop()** de l'applet pour arrêter le clip lorsque l'utilisateur change de page.

Optimiser le chargement des applets à l'aide des fichiers d'archives

Une applet peut faire appel à de nombreux éléments. En effet, chaque classe est placée dans un fichier séparé, ainsi que chaque image, chaque son, et en règle générale chaque ressource. Si votre applet fonctionne à partir d'un serveur, il faudra une transaction séparée pour chaque élément. Cela présente deux inconvénients. Le premier est un ralentissement global du chargement. Le second est plus subtil. En effet, les éléments ne sont chargés qu'au moment où ils sont nécessaires. Cette façon de procéder n'est pas forcément optimale. En effet, si votre applet exécute des traitements en

temps réel, il serait préférable que tous les éléments nécessaires soient chargés avant de commencer. Pour parvenir à ce résultat, il est possible de regrouper tous les éléments dans un fichier de type **.jar**. Vous pouvez alors utiliser la syntaxe suivante dans votre fichier HTML :

```
<applet code = "monApplet.class" archive = "archive.jar" ...
```

Ainsi, Java cherchera les éléments nécessaires dans le fichier **archive.jar**. Ce fichier sera chargé une fois lors du premier accès. Si certains éléments ne sont nécessaires que dans un cas précis, vous pouvez utiliser plusieurs fichiers **.jar** :

```
<applet code = "monApplet.class" archive = "archive1.jar,
archive2.jar" ...
```

Pour créer un fichier d'archives, vous devez employer un utilitaire livré avec le JDK. Par exemple, pour créer une archive contenant toutes les classes, toutes les images JPEG et tous les fichiers sons d'un répertoire, vous devez employer la syntaxe :

```
jar cvf archive.jar *.class *.jpeg *.au
```

Les applets et la sécurité

Les applets sont potentiellement dangereuses si elles sont diffusées sur un réseau largement ouvert au public comme Internet. Par sécurité, Java impose des limitations draconiennes à ce que peut faire une applet. Cela est extrêmement gênant dans le cas d'un Intranet ou d'un Extranet. Dans ces cas, en effet, l'origine des applets est parfaitement connue et il n'y a donc pas à craindre les malversations de programmeurs malveillants.

De la même façon, il est déplorable qu'une applet diffusée sur Internet à partir d'un site connu et auquel vous pouvez faire confiance soit soumise aux mêmes limitations que les applets d'origine inconnue. Java 2 permet de

résoudre ce problème en configurant le *security manager* afin qu'il puisse autoriser les applets d'origine connue à afficher des fenêtres ou à accéder au disque dur local.

Le security manager est installé par le navigateur qui affiche l'applet. Normalement, les applications ne sont donc pas soumises à un tel contrôle. Il est cependant possible d'installer un security manager pour les applications.

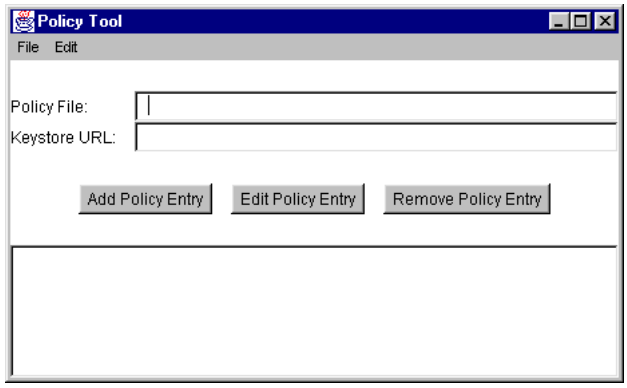
Mettre en place les autorisations

Pour modifier les autorisations concernant l'activité des applets, il est nécessaire de créer un fichier d'autorisations. Il s'agit d'un fichier texte qui peut être créé à l'aide de n'importe quel éditeur de texte. Cependant, Java est fourni avec un utilitaire (écrit en Java) permettant de simplifier cette tâche.

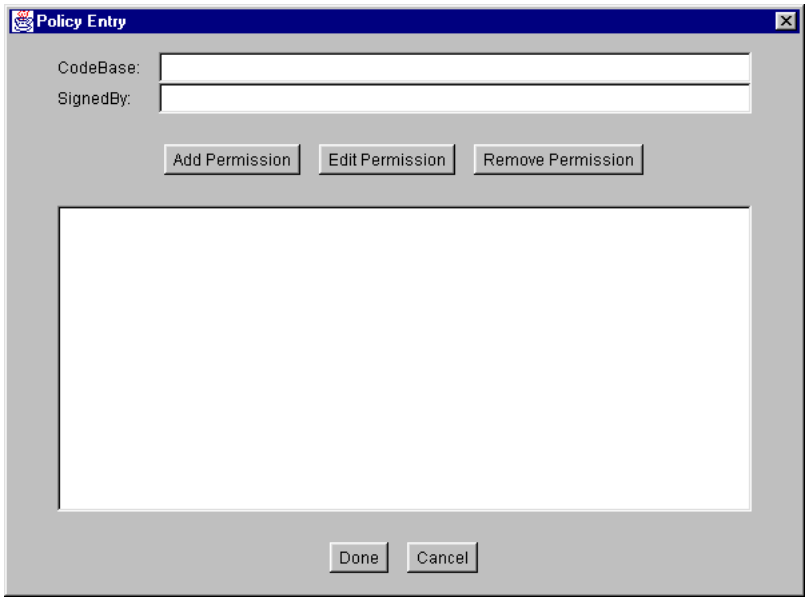
Pour lancer cet utilitaire, il suffit de taper la commande :

```
policytool
```

Ce programme affiche tout d'abord un message indiquant qu'aucun fichier d'autorisation n'a été trouvé, sauf si un tel fichier a déjà été créé. Sous Windows, ce fichier se trouve dans le répertoire **c:\windows\ profiles*votre_nom*** et est nommé **.java.policy**. La fenêtre suivante est alors affichée :



Cette fenêtre permet d'ouvrir ou de créer un fichier d'autorisations. Pour créer une autorisation, cliquez sur le bouton *Add Policy Entry*. Une nouvelle boîte de dialogue est affichée :



- La zone *CodeBase* permet d'indiquer la source des applets auxquelles vous souhaitez accorder l'autorisation.

- La zone *SignedBy* sert à attribuer l'autorisation aux applets portant une certaine signature. La signature est contrôlée à l'aide d'une clé publique enregistrée dans un *certificat*. Pour contrôler l'origine de l'applet, Java utilise la clé publique enregistrée dans le certificat et vérifie qu'elle correspond bien à la clé privée fournie par l'applet. Ce type d'autorisation nécessite évidemment que l'utilisateur ait préalablement obtenu une copie du certificat.

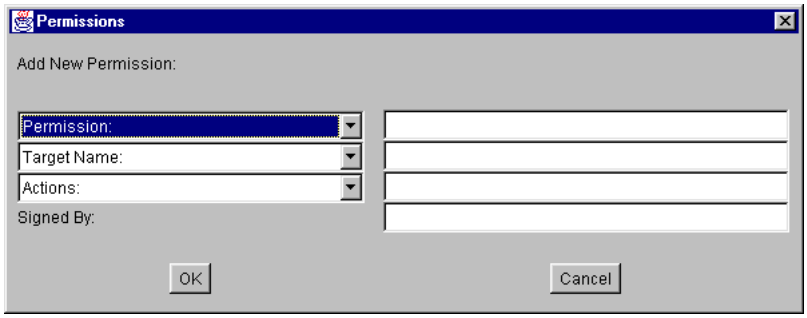
Il est parfaitement possible de remplir les deux zones pour n'autoriser que les applets provenant d'une certaine origine et portant une signature.

Tapez l'URL de l'origine des applets que vous souhaitez autoriser dans la zone *CodeBase*, par exemple :

```
http://www.volga.fr/-
```

Note : Le tiret après la barre oblique indique que l'autorisation concerne toutes les applets provenant de l'URL indiqué *et de ses sous-répertoires*.

Cliquez sur *Add Permission* pour afficher la boîte de dialogue suivante :

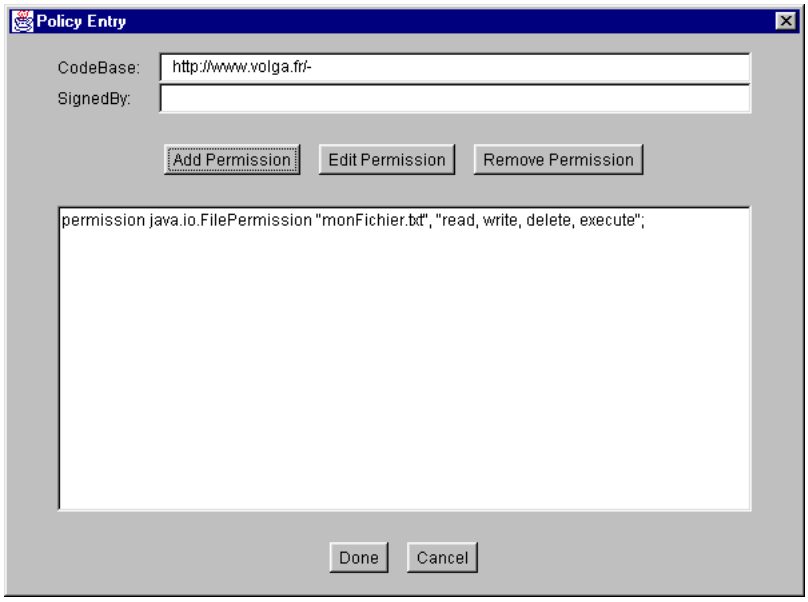


- Dans la zone *Permission*, sélectionnez le type de permission choisi. Par exemple, pour qu'une applet puisse écrire sur votre disque dur, sélectionnez *File permission*.
- Dans la zone de texte à droite de la zone *Target Name*, tapez le nom du fichier qui sera l'objet des autorisations.
- Dans la zone *Actions*, sélectionnez le type d'actions à autoriser. Dans le cas des autorisations concernant les fichiers, vous pouvez sélectionner :
  - *read* (lecture)
  - *write* (écriture)
  - *delete* (effacement)

- *execute* (exécution)
- *read, write, delete, execute* (toutes les autorisations)

Si vous souhaitez accorder deux ou trois autorisations, vous devrez le faire en plusieurs fois.

Cliquez sur *OK*. Vous obtenez l'affichage suivant :



Cliquez maintenant sur *Done*. L'autorisation que vous venez de définir est affichée dans la boîte de dialogue initiale.

Déroulez le menu *File* et sélectionnez *Save as*. Naviguez jusqu'à votre répertoire "*user_home*" (sous Windows, il s'agit de **c:\windows\profile*votre_nom***) et donnez au fichier le nom :

```
.java.policy
```

Le nom et le chemin d'accès choisis apparaissent maintenant dans la zone *Policy File*. Votre fichier d'autorisations est maintenant créé.

Note : S'il existait déjà un fichier **.java.policy**, n'enregistrez pas votre fichier sous le même nom, sous peine de l'écraser. Au contraire, reprenez la procédure depuis le début et modifiez le fichier existant au lieu d'en créer un nouveau.

Comment Java utilise les fichiers d'autorisations

Java utilise par défaut le fichier d'autorisations indiqué dans le fichier :

```
répertoire_de_java\lib\security\java.security
```

sous la forme :

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

Vous pouvez mettre en œuvre un nouveau fichier d'autorisations de plusieurs façons :

- Créer un fichier **.java.policy** comme nous l'avons fait ou modifier le fichier existant. C'est la façon la plus simple et la plus sûre de procéder.
- Ajouter une ligne au fichier **java.security** comme indiqué ci-dessous pour désigner un fichier d'autorisations supplémentaire :

```
policy.url.3=file:c:/java/autorisations
```

Cette façon de procéder peut être plus souple si vous devez modifier souvent les autorisations. Il vous suffit de commenter une ligne pour annuler temporairement les autorisations du fichier correspondant.

- Passer une référence au fichier d'autorisations dans la ligne de commande du programme affichant les pages contenant l'applet. Dans le cas de l'AppletViewer, la syntaxe à utiliser (sur une seule ligne) est :

```
appletviewer -J-Djava.security.policy=c:/java/autorisations
nom_du_fichier.html
```

Cette méthode n'est citée que pour mémoire. En effet, elle est inapplicable avec les autres navigateurs.

Utiliser le security manager avec les applications

Si vous le souhaitez, vous pouvez restreindre les possibilités allouées aux applications en demandant à Java d'utiliser un security manager. Pour cela, il vous suffit d'utiliser la syntaxe suivante pour lancer votre programme :

```
java -Djava.security.manager nom_du_programme
```

Votre programme est alors exécuté avec les mêmes restrictions que s'il s'agissait d'une applet, en utilisant les autorisations indiquées par le fichier **java.security**. Bien entendu, vous voudrez probablement définir un fichier d'autorisations complètement différent pour ce type d'utilisation.

Accéder à un URL à partir d'une application

L'utilisation des URL n'est pas réservée aux applets. Une application peut parfaitement accéder par ce moyen à des ressources se trouvant sur un réseau. Le programme suivant en fait la démonstration. Il accède à l'URL *http://www.volga.fr/* et affiche les données ainsi obtenues sur la sortie standard :

```
import java.net.*;
import java.io.*;
```

```
public class Reseau1 {

   static URL url;
   static InputStreamReader reader;
   static BufferedReader entree;
   static String ligne;

   public static void main( String[] args )  throws Exception {
      url = new URL("http://www.volga.fr/");
      reader = new InputStreamReader(url.openStream());
      entree = new BufferedReader(reader);
      while ((ligne = entree.readLine()) != null)
         System.out.println(ligne);
      entree.close();
   }
}
```

Si vous exécutez ce programme, vous verrez s'afficher à l'écran le contenu du fichier **index.html** se trouvant sur le site de la société *Volga Multimédia* (vous pouvez remplacer l'URL par ce que vous voulez).

Attention : Si ce programme ne fonctionne pas, il peut y avoir plusieurs raisons :

- Vous n'avez pas de connexion Internet. Dans ce cas, vous pouvez utiliser un URL local (c'est beaucoup moins spectaculaire).

- Vous avez une connexion Internet par le réseau commuté (téléphone) et l'établissement de la connexion n'est pas automatique. Dans ce cas, établissez la connexion manuellement avant de lancer le programme.

- Vous accédez à Internet à travers un proxy qui n'est pas configuré. Consultez alors votre administrateur réseau.

Notez que, avec une version de Windows configurée normalement, la connexion est établie automatiquement pour peu que votre fournisseur d'accès le permette. En revanche, le programme ne ferme pas la communication

téléphonique. N'oubliez donc pas de le faire manuellement si vous ne voulez pas avoir de mauvaises surprises avec votre note de téléphone.

Ce programme affiche le code HTML trouvé à l'URL indiqué. Vous pouvez afficher le document correspondant au code en utilisant la classe **JEditorPane**. Ce composant est capable :

- D'afficher du texte balisé en interprétant son format.
- D'exécuter des commandes d'édition sur le texte affiché.
- De détecter le format du texte, du moins pour les deux formats implémentés : HTML et RTF.

Il nous suffit donc d'afficher le texte lu à l'URL choisi dans une instance de **JEditorPane** pour obtenir un embryon de navigateur HTML. Voici un exemple de programme lisant un URL et affichant le contenu interprété. Nous avons inséré l'instance de **JEditorPane** dans une instance de **JScrollPane** qui gère automatiquement l'affichage des barres de défilement lorsque cela est nécessaire :

```
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
import java.net.*;
import com.sun.java.swing.text.html.*;
import com.sun.java.swing.event.*;

class Navigateur extends JFrame {

   static final Dimension screenSize =
                     Toolkit.getDefaultToolkit().getScreenSize();
   static final int largeurEcran = screenSize.width;
   static final int hauteurEcran = screenSize.height + 2;
   static URL url;
   Container pane;
   JScrollPane scrollPane;
   JEditorPane navPane;
```

```
   public Navigateur() {
      super();
      pane = getContentPane();
      this.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }});
      this.addComponentListener(new ComponentAdapter() {
         public void componentResized(ComponentEvent e) {
            scrollPane.setSize(getWidth() - 8, getHeight() - 28);
         }});

      try {
         url = new URL("http://www.volga.fr/");
         navPane = new JEditorPane();
         navPane.setPage(url);
         scrollPane = new JScrollPane(navPane,
                        JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
                        JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
         scrollPane.setPreferredSize(new Dimension(600,450));
         pane.add(scrollPane);
      }
      catch(Exception e) {System.out.println(e);}
   }

   public static void main( String[] args ) {
      Navigateur frame = new Navigateur();
      frame.pack();
      int l = frame.getWidth();
      int h = frame.getHeight();
      frame.setBounds((largeurEcran - l) / 2,
                                  (hauteurEcran - h) / 2, l, h);
      frame.setVisible(true);
   }
}
```

Voici le résultat affiché par ce programme :

Nous voyons que Java fournit les bases permettant de réaliser assez facilement un navigateur Web. Beaucoup plus facilement, en tout cas, qu'avec tout autre langage. La gestion des liens hypertextes est en effet assurée au moyen d'un **HyperlinkListener** dont la création et l'exploitation sont exactement semblables à celles des *listeners* que nous avons étudiés jusqu'ici.

Conclusion

Nous terminons ici notre approche des applets et des réseaux. Java offre de nombreuses autres possibilités que nous ne pouvons pas aborder ici. Il existe de nombreux ouvrages spécialisés dans la programmation d'application réseaux en Java. Si vous êtes intéressé par ce type d'ouvrages (en anglais), nous vous conseillons de consulter le site **www.amazon.com** où vous pourrez éventuellement vous les procurer par correspondance.

Annexe A

Les classes String et StringBuffer

Au Chapitre 4, nous avons étudié les chaînes de caractères en tant qu'objets. Nous avons vu qu'il n'existe pas de primitives pour représenter les chaînes de caractères. Par conséquent, il n'existe pas d'opérateurs pour le traitement du texte, à l'exception, toutefois, d'un cas particulier.

L'opérateur +

L'opérateur + peut être en quelque sorte "surchargé" pour opérer sur les chaînes de caractères. Appliqué à des chaînes de caractères, l'opérateur + représente la concaténation, c'est-à-dire la mise bout à bout des opérandes. Ainsi :

```
System.out.println("Le programme " + "est terminé");
```

est équivalent à :

```
System.out.println("Le programme est terminé");
```

Notez la présence d'un espace après le mot programme. La concaténation n'ajoute aucun espace entre les éléments. Vous devez donc les prendre en compte vous-même.

L'opérateur + est un raccourci qui permet d'éviter l'utilisation de la méthode **concat()**. Celle-ci permet d'obtenir le même résultat de la façon suivante :

```
String chaîne1 = "Le programme "
String chaîne2 = "est terminé.";
System.out.println(chaîne1.concat(chaîne2));
```

Le principal avantage de l'opérateur +, outre la simplicité d'écriture, est qu'il peut s'appliquer à des chaînes littérales.

Les constructeurs de la classe *String*

Une chaîne de caractères peut être créée de multiples façons. La plus simple consiste à affecter une chaîne littérale à un handle déclaré de type **String** :

```
String chaîne1 = "Une chaîne";
```

On peut également initialiser une chaîne à l'aide d'un autre handle de chaîne déjà initialisée :

```
String chaîne1 = "Une chaîne";
String chaîne2 = chaîne1;
```

Attention, cependant, au fait que les objets de type **String** sont statiques. S'agissant d'objet, on peut imaginer que, dans le cas précédent, **chaîne1** et **chaîne2** pointent vers un seul et même objet. C'est vrai. Cependant, si nous modifions **chaîne1**, la valeur de **chaîne2** ne sera pas modifiée, comme le montre l'exemple suivant :

```
public class Chaines {
     public static void main( String[] args ) {
     String chaîne1 = "FR2";
     String chaîne2 = chaîne1;
     chaîne1 = "TF1";
     System.out.println(chaîne1);
     System.out.println(chaîne2);
   }
}
```

qui affiche :

```
TF1
FR2
```

En effet, l'instruction **chaîne1 = "TF1"** ne modifie pas la valeur de **chaîne1** mais crée un nouvel objet de type **String** et lui affecte la valeur littérale indiquée. Cela peut paraître évident, mais ça ne l'est pas du tout ! **String** est la seule classe qui permet de créer ainsi des objets sans s'en rendre compte. Ici, la valeur précédente de **chaîne1** continue d'exister car un autre handle pointe vers cet objet. En revanche, lorsqu'un seul handle pointe vers une chaîne, chaque fois que la valeur de la chaîne est modifiée, l'ancien objet est abandonné et un nouvel objet est créé. Du point de vue des performances, ce processus peut paraître peu optimisé. Ce n'est pas le cas car, le plus souvent, les chaînes ne sont pas modifiées. Il est ainsi possible de leur attribuer une longueur fixe et non une taille dynamique, ce qui rend leur manipulation beaucoup plus rapide (à l'exception, bien sûr, du changement de leur valeur).

Toutes les méthodes qui renvoient un objet de type chaîne reproduisent ce type de comportement. Il n'existe pas de méthodes modifiant le contenu d'une chaîne de caractères. Ainsi, l'instruction suivante :

```
chaîne1.replace('a', 'A');
```

renvoie une chaîne de caractères identique à **chaîne1** mais dans laquelle les *a* sont remplacés par des *A*. Elle ne modifie pas la chaîne **chaîne1**. Dans l'instruction :

```
chaîne1 = chaîne1.replace('a', 'A');
```

l'objet vers lequel pointait **chaîne1** est remplacé par la valeur retournée par la méthode. L'ancien objet pointé par **chaîne1** est abandonné si aucun autre handle ne pointe vers lui. Démonstration :

```
public class Chaines2 {
     public static void main( String[] args ) {
     String chaîne1 = "FR3";
     String chaîne2 = chaîne1;
     chaîne1 = chaîne1.replace('3', '2');;
     System.out.println(chaîne1);
     System.out.println(chaîne2);
   }
}
```

Ce programme produit le résultat suivant :

```
FR2
FR3
```

Une chaîne de caractères peut également être créée de la façon normale, en utilisant l'opérateur **new**. Le constructeur de la classe **String** peut prendre pour argument :

- Une chaîne de caractères, sous la forme d'une valeur littérale, d'un handle, d'un objet retourné par une méthode, ou d'une expression quelconque s'évaluant comme une chaîne :

```
String chaîne1 = new String("une chaîne");
String chaîne2 = new String(chaîne1);
String chaîne3 = new String(objet.toString());
String chaîne4 = new String("une chaîne" + chaîne2 +
                                        objet.toString());
```

- Un tableau de **byte** ou de **char**, éventuellement accompagné d'un indice de début et d'un nombre de caractères, ainsi que de l'indication d'un type d'encodage :

```
String(byte[] bytes)
String(byte[] bytes, String enc)
String(byte[] bytes, int début, int longueur)
String(byte[] bytes, int début, int longueur,
                                        String encodage)
String(char[] cars)
String(char[] cars, int début, int longueur)
```

- Une instance de **StringBuffer** (voir plus loin) :

```
String(StringBuffer buffer)
```

Les méthodes de la classe *String*

La classe **String** contient de nombreuses méthodes permettant de manipuler les chaînes de caractères. Nous ne les décrirons pas ici en détail. Nous vous conseillons de vous reporter à la documentation en ligne de Java. Nous donnerons simplement un exemple commenté de l'utilisation de quelques-unes de ces méthodes. Il s'agit de la réalisation d'un composant semblable à

celui que nous avons réalisé au Chapitre 18 pour afficher du texte en relief. La différence est qu'ici le composant est capable de prendre un texte de n'importe quelle longueur et de le découper en lignes en fonction de sa largeur :

```
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
public class Chaines3 {
   public static void main( String[] args ) {
      final Dimension screenSize =
                     Toolkit.getDefaultToolkit().getScreenSize();
      final int largeurEcran = screenSize.width;
      final int hauteurEcran = screenSize.height + 2;
      int l = 300;
      int h = 100;
      String s = "Au fond de son âme, cependant, elle attendait un
événement. Comme les matelots en détresse, elle promenait sur la
solitude de sa vie des yeux désespérés, cherchant au loin quelque
voile blanche dans les brumes de l'horizon.";

      JFrame frame = new JFrame();
      frame.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }});
      frame.getContentPane().add(new TextPanel(0, 0, l, h, s));
      frame.setBounds ((largeurEcran - l) / 2,
                                  (hauteurEcran - h) / 2, l, h);
      frame.setResizable(false);
      frame.setVisible(true);
   }
}

class TextPanel extends Panel {
   Font labelFont = new Font("Helvetica", Font.BOLD, 11);
   FontMetrics fontMetrics = getFontMetrics(labelFont);
```

```
Color
   fColor = new Color(223, 223, 223),
   fColorC = new Color(255, 255, 255),
   fColorF = new Color(63, 63, 63),
   bColorF = new Color(32, 64, 96),
   bColorC = new Color(156, 171, 196),
   bColor = new Color(128, 144, 175);
String texte;
int
   largeur = 0,    // Largeur du composant
   hauteur = 0,    // Hauteur du composant
   xt = 8,    // Coordonnée horizontale du texte (marge gauche)
   yt = 13,   // Coordonnée verticale du texte (première ligne)
   ht = 13,   // Interligne
   lMax = 0;  // Longueur maximale d'une ligne (en tenant
              // compte des marges)
String[] mots = new String[300];
String[] lignes = new String[6];
int[] llignes = new int[6];
int numLignes;  // Nombre de lignes
int i;
int space;

TextPanel (int x, int y, int l, int h, String s) {
   super();
   largeur = l;
   lMax = l - (2 * xt);
   hauteur = h;
   setBounds(x, y, l, h);
   texte = s;
   setText();
}

public void setText() {
   space = fontMetrics.stringWidth(" ");
   for (i = 0; i < 6; i++) {
      lignes[i] = "";
   }
```

```
      int z = 0; // compteur de mots
      int y = 0;
      int x = texte.indexOf(32);
      int k = 0;
      while (x != -1) {  // mots[z] contient les (z + 1) mots
                         // de la phrase (de mots[0] à mots[z]
         mots[z] = texte.substring(y, x);
         z++;
         y = x + 1;
         x = texte.indexOf(32, y);
      }
      mots[z] = texte.substring(y, texte.length());
      i = 0;
      int l = 0; // Numéro de ligne
      int j = 0; // Longueur de la ligne en cours de création
      while (i <= z) { // Tant qu'il reste des mots
         if (j + fontMetrics.stringWidth(mots[i]) > lMax) {   // En
                    // ajoutant le mot suivant, on dépasse la ligne
            llignes[l] = j - space; // On enregistre la longueur de
                                    // la ligne (moins le dernier
                                    // espace)
            j = 0;  // On remet la longueur de ligne à 0
            l++; // On passe à la ligne suivante
         }
         else {
            j = j + fontMetrics.stringWidth(mots[i]) + space;
            lignes[l] = lignes[l] + mots[i] + " ";
            llignes[l] = j - space; // On enregistre la longueur de
                                    // la ligne (moins le dernier
                                    // espace)
            i ++;
         }
      }
      numLignes = l;  // nombre de lignes (exemple : 6 lignes,
                      // numérotées de 0 à 5)
      repaint();
   }
```

```
   public void paint(Graphics g) {
      g.setColor(bColor);
      g.fillRect(0, 0, largeur, hauteur);
      g.draw3DRect(0, 0, largeur - 1, hauteur - 1, true);

      g.setFont(labelFont);
      for (i = 0; i <= numLignes; i++) {
         g.setColor(fColorF);
         g.drawString(lignes[i], (xt + 1), ((yt + i * ht) + 1));
         g.setColor(fColorC);
         g.drawString(lignes[i], xt, (yt + i * ht));
      }
   }
}
```

La partie intéressante du programme est la méthode **setText**. Elle utilise la méthode **indexOf** pour trouver la position d'un caractère donné dans une chaîne, la méthode **subString** pour extraire une partie d'une chaîne, la méthode **length**, qui donne la longueur d'une chaîne en nombre de caractères et la méthode **StringWidth**, qui donne la longueur d'une chaîne en fonction de la police utilisée pour l'affichage. Les lignes de la méthode **setText** sont commentées et vous ne devriez avoir aucun mal à comprendre son fonctionnement.

La classe *StringBuffer*

La classe **StringBuffer** permet de créer des chaînes de caractères dynamiques, c'est-à-dire dont le contenu peut être modifié. La manipulation de telles structures est plus lente car l'espace qui leur est alloué doit être modifié en fonction des besoins. Cependant, la possibilité de modifier le contenu permet d'éviter la création d'une nouvelle instance à chaque modification. Pour optimiser les performances, il est préférable d'utiliser **StringBuffer** lorsque le contenu d'une chaîne doit être modifié, puis de créer une instance de **String** avec ce contenu une fois les modifications terminées.

Une instance de **StringBuffer** peut être créée de trois façons :

- **StringBuffer()** crée une chaîne vide en réservant la mémoire pour 16 caractères.
- **StringBuffer(int l)** crée une chaîne vide en réservant la mémoire pour l caractères.
- **StringBuffer(String chaîne)** crée une chaîne dont le contenu est identique à celui de la chaîne statique **chaîne**.

La classe **StringBuffer** contient diverses méthodes permettant de modifier le contenu d'une chaîne dynamique. La méthode **append**, par exemple, permet d'ajouter à la fin de la chaîne sur laquelle elle est invoquée une représentation sous forme de chaîne de son argument. La méthode **insert** fait la même chose à une position quelconque de la chaîne sur laquelle elle est invoquée. D'autres méthodes permettent d'obtenir le caractère correspondant à une position donnée, de remplacer un caractère, ou d'obtenir une sous-chaîne en indiquant la position de début et la position de fin de celle-ci.

Annexe B

Les mots réservés

Les mots suivants sont réservés en Java et ne peuvent donc pas être employés pour des identificateurs. Notez que certains de ces mots (apparaissant ici en italique) ne sont pas utilisés par Java.

abstract	boolean	break	byte
byvalue	case	*cast*	catch
char	class	*const*	continue
default	do	double	else
extends	false	final	finally

float	for	*future*	*generic*
goto	if	implements	import
inner	instanceof	int	interface
long	native	new	null
operator	*outer*	package	private
protected	public	*rest*	return
short	static	**strictfp**	super
switch	synchronized	this	throw
throws	transient	true	try
var	void	volatile	while
widefp			

Note : Les mots en gras ne sont pas encore définitivement adoptés. Ils sont donc susceptibles de disparaître dans une prochaîne version.

Annexe C

Configuration d'UltraEdit

Comme nous l'avons dit au début de ce livre, vous pouvez utiliser, pour écrire des programmes Java, de très nombreux outils allant de l'éditeur le plus simple, comme le bloc-notes de Windows, à l'environnement de développement intégré le plus sophistiqué. Quels que soient les avantages des environnement intégrés, ils ne sont pas adaptés à l'apprentissage d'un langage, dont ils ont tendance à masquer les particularités. D'un autre côté, un éditeur trop simple tel que le bloc-notes vous fera perdre beaucoup de temps (et probablement de cheveux).

Les qualités d'un bon éditeur sont sa simplicité, sa rapidité, son ergonomie, et la disponibilité de certaines fonctions qui simplifient beaucoup le travail des programmeurs :

- La numérotation des lignes : cette fonction est absolument indispensable car toutes les références renvoyées par le compilateur le sont sous forme de numéros de lignes
- La capacité à mettre la syntaxe en évidence à l'aide de couleurs différentes est utile, mais pas indispensable. Disons que c'est un élément de confort et de productivité très appréciable qui vous aidera de deux façons : en mettant en évidence vos fautes de frappe, et surtout en accélérant la lecture des programmes.
- La capacité à exécuter des commandes du système hôte est très intéressante. Elle vous permet de lancer la compilation et l'exécution du programme depuis l'éditeur.
- La possibilité de créer des macros et de configurer des commandes par menu ou au clavier est très utile pour simplifier encore la compilation et l'exécution des programmes.
- La possibilité de passer des paramètres à ces macros vous permet de compiler le programme se trouvant dans la fenêtre active sans avoir à spécifier son nom.
- La possibilité de capturer la sortie du compilateur ou du programme exécuté est inestimable. En effet, dans une fenêtre DOS, vous ne pouvez pas faire défiler l'affichage. Si la compilation de votre programme produit un nombre important d'erreurs, vous ne pourrez pas les voir affichées. Par ailleurs, le compilateur Java envoie les erreurs sur la sortie *Standard error* qui n'est pas redirigeable vers un fichier. Il n'y a donc aucune solution simple pour l'utilisateur.
- La capacité de gérer un ensemble de fichiers correpondant à un même projet est très appréciable.
- La possibilité de sélectionner un bloc délimité par deux caractères tels que parenthèses ou accolades est d'une aide précieuse pour détecter certaines erreurs.
- La possibilité d'ajouter ou de retirer automatiquement une tabulation au début de chaque ligne d'un bloc sélectionné est très utile.
- La possibilité de commenter automatiquement une partie du code est souhaitable.

Il existe de nombreux éditeurs sur le marché, mais aucun ne remplit toutes ces conditions. Il en existe un, cependant, qui les remplit toutes à la perfection sauf la dernière. Il s'agit d'UltraEdit, produit disponible en shareware pour un prix dérisoire. La dernière version d'UltraEdit est disponible sur le CD-ROM accompagnant ce livre.

UltraEdit n'est pas gratuit !

Attention : Le développement d'un tel produit représente un travail considérable. Sa mise à jour régulière est un gage de productivité maintenue pour tous ses utilisateurs. En conséquence, vous devez *impérativement* acheter la licence si vous décidez de l'utiliser au-delà de la période d'essai. Vous trouverez tous les renseignements vous permettant d'acquérir la licence sur le site de son éditeur, à l'adresse **www.idmcomp.com**. Le prix est de 30$ et vous pouvez régler par carte bancaire. La mise à jour est gratuite pendant une période d'un an et coûte ensuite 15$ par an.

Installation et configuration d'UltraEdit

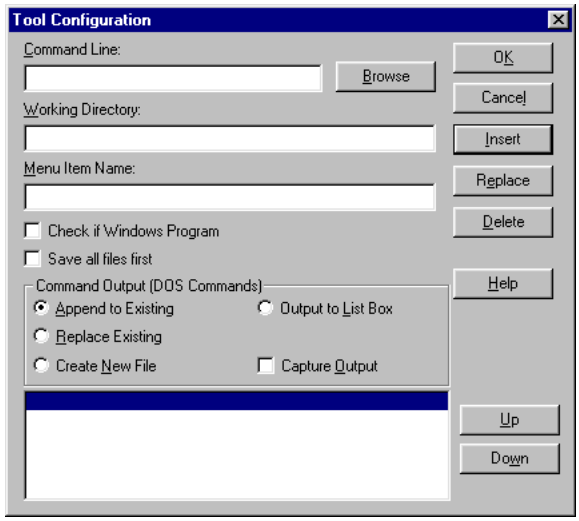
L'installation d'UltraEdit est extrêmement simple. Il vous suffit de lancer le programme exécutable **Uedit32i.exe** et de répondre aux questions posées.

Une fois le programme installé, il vous faudra le configurer pour pouvoir compiler et exécuter des programmes Java. Pour cela, il vous suffit de dérouler le menu *Advanced* et de sélectionner *Tool Configuration*. La boîte de dialogue de la page suivante est affichée.

Dans la zone *Command line*, tapez :

```
c:\jdk1.2\bin\javac "%f"
```

%f représente le nom du fichier en cours d'édition dans la fenêtre active. Attention à bien utiliser un **f** minuscule, qui désigne le nom long du fichier, alors qu'un **F** majuscule désigne le nom court.



Dans la zone *Working Directory*, tapez le chemin d'accès au répertoire de travail, par exemple :

```
c:\java
```

Dans la zone *Menu Item Name*, tapez le nom que vous voulez donner à la commande dans le menu, par exemple :

```
Compile
```

Cochez la case *Save all files first* de façon qu'UltraEdit enregistre les fichiers avant de les compiler ou de les exécuter.

Dans la zone *Command Output (DOS Commands)*, sélectionnez l'option qui vous convient. Vous pouvez :

- Ajouter les messages du compilateur à un fichier existant,
- Remplacer un fichier existant,
- Créer un nouveau fichier à chaque compilation,

- Afficher les messages dans une fenêtre (*List Box*).

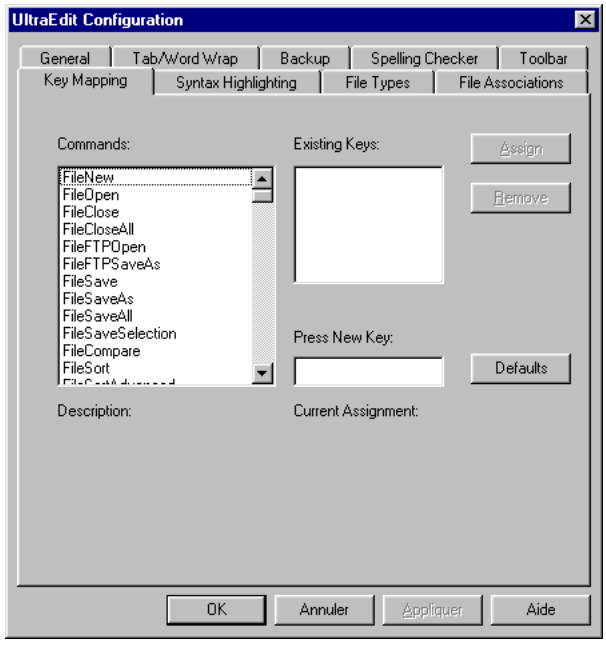
La quatrième option est la plus utile.

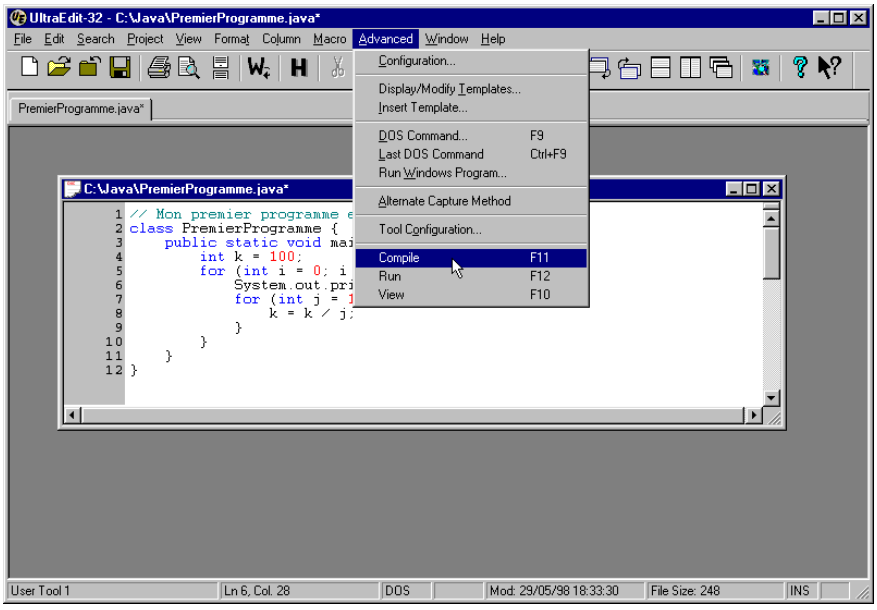
Cochez ensuite la case *Capture Output* puis cliquez sur *Insert*.

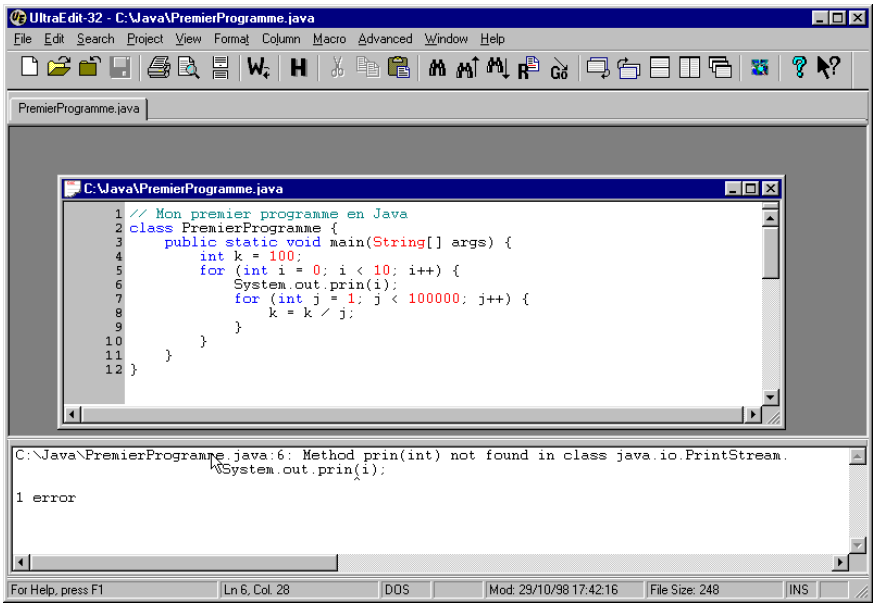
Recommencez la procédure pour créer une commande exécutant le programme et une autre lançant l'AppletViewer. Les lignes de commande à indiquer sont les suivantes :

- Exécuter : **c:\jdk1.2\bin\java "%n"**
- AppletViewer : **c:\jdk1.2\bin\AppletViewer "%n".html**

Il ne vous reste plus maintenant qu'à configurer des équivalents clavier pour ces commandes. Déroulez le menu *Advanced* et sélectionnez l'option *Configuration*. Dans la boîte de dialogue affichée, cliquez sur l'onglet *Key Mapping* pour obtenir l'affichage suivant :




UltraEdit-32 - C:\Java\PremierProgramme.java*
File Edit Search Project View Format Column Macro Advanced Window Help
Configuration...
Display/Modify Templates...
Insert Template...
DOS Command... F9
Last DOS Command Ctrl+F9
Run Windows Program...
Alternate Capture Method
Tool Configuration...
Compile F11
Run F12
View F10
PremierProgramme.java*
C:\Java\PremierProgramme.java*
1 // Mon premier programme e
2 class PremierProgramme {
3 public static void mai
4 int k = 100;
5 for (int i = 0; i
6 System.out.pri
7 for (int j = 1
8 k = k / j;
9 }
10 }
11 }
12 }
User Tool 1
Ln 6, Col. 28
DOS
Mod: 29/05/98 18:33:30
File Size: 248
INS


UltraEdit-32 - C:\Java\PremierProgramme.java
File Edit Search Project View Format Column Macro Advanced Window Help
PremierProgramme.java
C:\Java\PremierProgramme.java
1 // Mon premier programme en Java
2 class PremierProgramme {
3 public static void main(String[] args) {
4 int k = 100;
5 for (int i = 0; i < 10; i++) {
6 System.out.prin(i);
7 for (int j = 1; j < 100000; j++) {
8 k = k / j;
9 }
10 }
11 }
12 }
C:\Java\PremierProgramme.java:6: Method prin(int) not found in class java.io.PrintStream.
System.out.prin(i);
1 error
For Help, press F1
Ln 6, Col. 28
DOS
Mod: 29/10/98 17:42:16
File Size: 248
INS

- Dans la zone *Commands*, sélectionnez *AdvancedUserTool1*.
- Cliquez dans la zone *Press New Key* et tapez la touche ou la combinaison de touches que vous voulez affecter à la commande.
- Cliquez sur *Assign*.
- Recommencez pour les commandes *AdvancedUserTool1* et *AdvancedUserTool2*.

UltraEdit est maintenant configuré. Vous pouvez alors compiler et exécuter un programme en sélectionnant simplement une commande dans le menu *Advanced* ou en tapant une touche, comme indiqué sur la figure de la page ci-contre (en haut).

Si le programme comporte une erreur, celle-ci sera affichée dans la fenêtre réservée à cet effet (si vous avez choisi l'option *Output to List box*), comme dans l'exemple de la page ci-contre (en bas).

Notez qu'il vous suffit de faire un double clic sur le message d'erreur pour placer le point d'insertion au début de la ligne correspondante. (Si la fenêtre n'est pas affichée, déroulez le menu *View* et sélectionnez *Output Window*.)

Index

A

abstract 310, 339, 342
Accès direct (fichiers à) 564
Accesseurs 277, 300, 315, 389
Accessibilité 110, 118, 275
  et références 441
ActionEvent 732
ActionListener 355, 732
actionPerformed() 356, 730
ActiveX xxiii, 18, 27
Adapter 498
add() 51
Affectation
  des handles 79
  opérateur d' 185
Affichage
  de texte 799
  des images 808
  d'un message dans la barre d'état 835
aiff (format de fichier sons) 837
Aléatoires (nombres) 377
Algorithme 59
Allocation
  de la mémoire 76, 84, 428, 460
  de ressources 446
Anonymes
  classes 106, 495
    constructeurs des 165, 302
    initialisateurs des 497
    références aux 498
  instances
    externes 480
  interfaces 497
  objets 75
    dans les tableaux 368
    et garbage collector 432
    threads 614
  primitives 99
  threads 609
ANSI 53, 115
API
  3D 819
  ActiveX xxiii
  documentation 16, 41, 788
  dossier 6, 16, 41

append() 860
Applets 46, 673, 821, 822
  affichage des images dans les 836
  contexte d'exécution des 833
  création d' 822
  et autorisations d'accès 840
  et fichiers compressés 838
  fonctionnement des 826
  layout managers des 825
  limitations des 47
  optimisation des 555
  paramètres des 827
  signées 5
AppletViewer 2, 4, 23, 27, 48, 55, 450, 824, 829, 845, 867
Applications
  accès à un URL à partir d'une 845
  affichage d'images dans les 809
  arrêt des, en cas de boucle infinie 248
  boîtes de dialogue des 706
  compatibilité des xxii
  conception d' 134
  contexte d'exécution des 822
  création de clips dans les 837
  déploiement d' 12
  développement d' xxviii, 133
    rapide 24
  diffusion d' xxxiii, 13
  diffusion des xxiv
  échange de données entre 539
  et héritage 357
  fenêtrées 354, 677
  fonctionnement des 826
  lancement des 23
  look & feel des 773
  maintenance des 459, 668
  multifenêtrées 688
  multimédia 437, 695
  optimisation des 555
  portabilité des 695
  protection contre la copie 12
  réseaux 849
  sécurité des 845
  structure des 39
  test des 14
Archives
  fichiers d' 293
    création de 555
Arguments
  constructeurs sans 159, 160, 180, 658, 667
  de la méthode main() 39
  de la méthode println() 176
  de l'instruction if 90
  de type tableaux 381
  déclarés final 306
  des constructeurs 69, 73
  des méthodes 71
    surchargés 171
  handles 199, 206
  passage des 382, 389, 571
    au constructeur de la classe parente 615
ArithmeticException 506, 522
Arithmétique binaire (opérateurs d') 207, 218
Arithmétiques (opérateurs) 187
Arrays 427
Arrêt
  des applications en boucle infinie 248
Arrondi
  dans la division 188
  dans les castings 104
ASCII 115
Assembleur xxv, 83
Asynchrone
  méthode 809
  mode du garbage collector 428, 460
  processus 77
au (format de fichier sons) 837
AudioClip 837
Auto-décrémentation 194
Auto-incrémentation 194
autoexec.bat 8, 15, 34, 43
Autorisations d'accès (pour les applets) 840
Awt 674

B

Bags 398
Barre
  de défilement 670, 847
  de menus 675, 701, 741
    création d'une 745
  de progression 538
  de tâches (de Windows) 10
  de titre 674, 772
  d'espacement (pour la sélection des boutons) 732
  d'état 835
  d'outils 25
Base de registre 15
BasicStroke 798
BigDecimal 87
BigInteger 87
Binaires (opérateurs) 187
Binding 337
BitSet 393
Boîtes de dialogue
  dans les applications 706
Boolean 87
boolean 100, 211, 219
BorderLayout 696, 698, 700
Borland 26, 29

Boucles
for 244
infinies 248
sortie par return 256
while 259
Boutons 727
radio 760
BoxLayout 701
Branchements 232
vers des étiquettes 258
break 253, 256
BufferedReader 515, 548
ButtonGroup 760
Byte 86
Bytecode xxii, xxxi, 12
compilateurs de xxxii
compilation à la volée xxxiii
interpréteur de xxxii
portabilité du xxxiii

C

C (langage) xxvi, 184
C++ xxvii, 184
Canvas 778
Caractères
autorisés dans les identificateurs 104
autorisés dans les noms 40
chaînes de 41, 113
codage des 87, 115
de fin de lignes 553
jeu de 53
polices de 694, 800
streams de 540
Case à cocher 760
Cast. *Voir* Casting
Casting
des primitives 101
des valeurs littérales 223
d'un entier en flottant 103
d'un flottant en entier 104
explicite 102
opérateurs de 221
catch 508, 527
CD-ROM xxxii
hybride xxxiii
Chaîne de caractères
création de 852
Chaînes de caractères 41, 113, 851
concaténation de 851
dynamiques 114, 859
littérales 115
ChangeListener 775
char 86, 99, 215
Character 408
checkError() 540
Checksum (contrôle de) 539
Chemins d'accès
aux classes Java 42
configuration des 8, 10, 15
par défaut 33
séparateurs de niveaux 44
Class 655
Classes
abstraites 310
anonymes 106, 165, 495
et constructeurs 165, 302, 497
et initialiseurs 497
références aux 498
déclaration des 37
définition des 37
définitions des 61
extension de 65
extension des 52
externes
références aux 482
finales 308
imbriquées 463
importation des 471
noms des 470
instances de 65
internes 463
références aux 492
locales 488
et handles 491
noms des 490
références aux 490
membres 66, 471
et héritage 481
références aux 476
parentes 134
invocation des constructeurs des 143
référence aux 141
privées 315
protégées 314
publiques 313
représentation de la hiérarchie des 62
sans constructeurs 165
synchronisées 309
CLASSPATH 10, 15, 43, 282
Clonage 576
en profondeur 582
en surface 582
et héritage 585
interdire le 589
clone() 576
Cloneable 579, 587
CloneNotSupported-Exception 589
CloneNotSupportedException 579
Codage des caractères 87, 115
Code
ANSI 115
ASCII 115
UNICODE 87, 93
Collections 397, 413
Color 76

COM 27
command.com 10
Commentaires 37
  syntaxe des 36, 45, 50
Communication
  entre les threads 644
  stream de 541
Compactage
  de la mémoire 429
Comparable 402, 405
Comparateurs 396, 402, 405, 422
Compatibilité
  des applications xxii
  entre les versions de Java 13
Compilateurs
  de bytecode xxxii
  JIT xxxiii, 2, 26
  natifs xxxii
Compilation 32
  options de 285
Complément (à 1 et à 2) 213
Component 52, 675
Composants
  légers 671
  lourds 670
Composition 125
  et héritage 356
Compression 554
Concaténation
  de chaînes de caractères 851
Configuration
  de l'environnement 8
  des chemins d'accès 8, 10, 15
Consoles de jeu xxxiii
Constantes 117, 344, 357
  globales 346
Constructeurs 53, 138
  accessibilité des 316
  arguments des 69, 73
  classes sans 165
  définition 69
  des classes parentes 143
  et classes anonymes 165, 302, 497
  exceptions dans les 532
  explicites 122
  par défaut 122, 154, 160
  sans argument 159, 160, 180, 658, 667
  signatures des 151
  surcharger les 151
Container 52
contentPane 675
Contexte
  d'exécution 822, 833
  graphique 778
continue 257
Contrôle
  de checksum 539
  des threads 610
  structure de 231
Conversion
  des caractères de fin de lignes 553
Copie
  de tableaux 384
  d'objets 576
Corel xxii
Courbes
  cubiques 788
  de Béziers 788
  quadratiques 788
Création
  de fichiers d'archives 555
  de threads 601
  d'exceptions 520
Créationnisme 63
Cross-compilation xxxiii

D

Débordement
  de la pile xxx, 84
  d'un tableau xxx
  d'une variable numérique 104
Décalage (opérateurs de) 208
Déclaration
  des classes 37
  des handles 79
  des méthodes 38
  des primitives 87
  des tableaux 364
  méthodes 70
Décompression
  de fichiers 560
default 266
Définition
  des classes 37
  des méthodes 38, 71
DeflaterOutputStream 554
Démons 641
Démonstration
  de Java 3, 6
  des composants Swing 6
depend (option de compilation) 285
Déploiement d'applications 12
Deprecated 457
Désallocation
  de la mémoire 76, 428
  de ressources 446
Désérialisation 538
destroy() 826
Développement
  d'applications 133
    rapide 24
  environnements de 23
Dialog 674
Diffusion
  des applications xxiv, xxxiii,

13
Diminution des ressources xxx
dispose() 751
Division 187
arrondi dans la 188
et décalage 210
par zéro 506, 526
reste de la 187
do...while 262
doClick() 774
Documentation
du JDK 6
en ligne (utilisation de la) 41
installation de la 15
Données
compression de 554
échange entre applications 539
DOS 115, 281, 293, 600, 864
Double 87
double 100
drawImage() 808
Dymo 804

E

Early binding 337
Échanges de données entre applications 539
Écriture
dans un tampon indexé 539
de données, avec vérification 539
de données binaires 539
des données en mémoire xxix
d'un fichier 544
séquentiel 538
sur la sortie standard 45
Éditeur (choix d'un) 20
Effets de bord 38
Égalité
définition de l' 199
des objets 196, 198
des primitives 186
évaluation de l' 198
opérateur d' 80
test de l' 201, 229
else 237
elseif 241
Embossage 804
Encapsulation 276
Ensembles 401
ordonnés 402
Entrées/sorties 536
Enveloppeurs 85, 93
avec les vecteurs 388
de primitives 201
Environnement 1
configuration de l' 8
de développement 23
intégré 4
variables d' 9, 15, 282
equals() 198, 401
Équivalence
de handles 196
de primitives 196
Erreurs
surveillées 504, 507
traitement des 502
et finalisation 459
Errors 532
Étiquettes (pour les branchements) 258
Événements 354, 679
de souris 734
Évolutionnisme 63
Exceptions 161, 506
création d' 520
dans les constructeurs 532
et héritage 532
handlers d' 509
Explicite (casting) 102
extends 49, 52
Extension
de classes 52, 65, 275
et héritage 481
de signe 210
de zéro 208, 214
des noms de classes 281
des noms des fichiers sources 314
du code ASCII 115
packages constituant une 681, 726
sélective, des arbres 727

F

Fenêtres 673
d'applications 677
DOS 32
hiérarchie des 673
internes 746
structure des 674
Fermeture d'une fenêtre 679
Fichiers
à accès direct 564
d'archives 293, 555
décompression de 560
lecture et écriture de 544
longueur des 569
noms de, sous Windows 34
pointeurs de 569
File 545, 565
FileDescriptor 545
fillInStackTrace() 512
FilterOutputStream 554
final 117, 304, 337
Finalisation
des objets 429
et héritage 458
et traitement d'erreur 459
Finaliseurs 432
finalize() 429, 432, 453

finally 525, 527, 532
Fins de ligne 37
Float 87, 199
float 86, 100
FlowLayout 700
  utilisation du 703
Fonctions 206
FontMetrics 804
for 244
Format
  des identificateurs 104
  des nombres 85
  des valeurs littérales 99
Formatage
  du code Java 38
forName() 655
Forth 184
FORTRAN 58
fr.comp.java 30
Frame 674
friendly 313
Fuites de mémoire xxx, 76, 430

G

Garbage collector xxx, 76, 428
  contrôle du 437, 450
  et objets anonymes 432
  et paramètres de l'interpréteur 460
  optimisation du 430
Générateur de nombres aléatoires 377
getAppletContext() 833
getAudioClip() 837
getClass() 658
getCodeBase() 837
getDocumentBase() 837
getHeight() 813
getMaximumSize() 707
getMinimumSize() 707
getName() 632
getPreferredSize() 707
getRuntime() 457
getSize() 707
getStringBounds() 804
getWidth() 813
glassPane 674
Glissières 775
Graphics 778
Graphics2D 778
Graphisme 777
GridBagConstraints 713
GridBagLayout 698, 712
  utilisation du 711
GridLayout 700, 708
Groupes
  de boutons 760
  de composants 702
  de nouvelles 30
  de threads 613
GZIP 554
GZIPInputStream 538
GZIPOutputStream 539

H

Handlers d'exceptions 509, 522, 527, 529
Handles 83, 421
  affectation des 79
  création des 79
  de tableaux 369
  déclaration des 79
  définition 78
  équivalence de 196
  et classes locales 491
  initialisation des 79
  tableaux de 374
HashTable 396
Header C 6
Héritage 125, 135
  et classes membres 481
  et clonage 585
  et composition 356
  et exceptions 532
  et finalisation 458
  multiple 346, 352, 357
HotJava xxiv, 17, 824
HTML 2, 13, 17, 46, 55, 727, 809, 822
Hybride (CD-ROM) xxxiii
HyperlinkListener 849

I

IDE 4, 24
Identificateurs
  caractères autorisés dans les 104
  format des 104
  masquage des 109
  portée des 106
  visibilité des 107
Identité des objets 196
if 235
if...else 220
ImageObserver 808
Images
  affichage des 808
  dans les applets 836
imageUpdate() 815
Imbrication
  de boucles 250
  de classes 463, 476
import 287
Importation
  de classes imbriquées 471
  explicite 289
  implicite 289
Incrémentation 71
  d'une boucle for 245, 248
indexOf() 859

Indices (des boucles for) 249
Informations
  perte d', dans les castings 104
init() 49, 826
Initialisation
  des handles 79
  des instances 137
  des objets 139, 326
  des primitives 88
  des tableaux 365
    automatique 366
  des variables 137
    finales 305
    statiques 301
  d'une boucle for 245
  opérateur d' 89
  ordre d' 165
Initialiseurs 162
  dans les classes anonymes 497
  de variables d'instances 162
  d'instances 164
  statiques 302
InPrise 26, 29
InputStream 548
InputStreamReader 541, 548
insert() 860
Installation
  de la documentation en ligne 15
  du JDK 3
  du JRE (Windows) 14
instanceof 657
Instances 65
  externes anonymes 480
  initialisation des 137
  initialiseurs d' 164
  initialiseurs de variables d' 162
  variable d' 139
Instanciation 72
  d'une classe à l'aide de son objet Class 657
int 70, 86, 100
Integer 85, 86, 197
Integrated Development Environment 4, 24
Interfaces 312, 343
  anonymes 497
  et polymorphisme 347
  privées 315
  protégées 314
  publiques 313
InternalFrameAdapter 498
InternalFrameListener 498, 751
Internationalisation
  avec le JRE 13
Internet xxiv
Internet Explorer xxiii, xxiv, 19
  pour Macintosh xxv
Interpréteur
  allocation de la mémoire par l' 460
  de bytecode xxxii
  et contrôle du garbage collector 460
IOException 518, 539, 545
isInstance() 657
Itérateurs 402
  de listes 404
Iterator 402

J

JApplet 822
JAR 5, 42, 293, 538, 539, 555, 839
jar.exe 5
JarInputStream 538
JarOutputStream 539
jarsigner.exe 5
Java 1.0 xxiii
Java 1.1 xxiv
Java Boutique 29
Java Developer Connection 28
Java Developers Journal 29
Java Runtime Environment xxiv, 2, 12
Java Studio 26
Java Virtual Machine xxii, xxxi, 12
Java WorkShop 26
java.exe 4
java.lang.System 41
JavaBeans 751
  et les environnements de développement 25
javac.exe 2, 4, 34
javadoc.exe 2, 5
javah.exe 5
javap.exe 5
JavaScript 18
JavaWorld 29
JBuilder 26
JCheckBox 760
JDC 28
JDesktopPane 750
JDK (contenu du) 4
Jetables (objets) 532
JFrame 674, 696, 746
JInternalFrame 674, 746
JIT (compilateurs) xxxiii, 2, 26
JLabel 707
JMenuBar 675
join() 610
JRE xxiv, 2, 12
  installation du 14
  internationalisation 13
jre.exe 7
JRootPane 674
Just In Time xxxiii
JVM xxii, 12, 130, 427

L

Lancement des applications 23
Langages
  Assembleur xxv, 83
  C xxvi, 184
  C++ xxvii, 184
  de prototypage xxvii
  Forth 184
  FORTRAN 58
  Lisp 59
  machine xxvi, 58
  Pascal xxvi
  PostScript 184
  Smalltalk xxvii, 184
  structurés xxvi
Late binding 337
layeredPane 674
Layout managers 677, 693
  affichage sans 721
Lecture d'un fichier 544
length() 859
Libération
  de la mémoire 76, 429
  ressources 446
Ligne de commande 2
Lignes (caractères de fin de) 553
LineNumberReader 542
Lisp 59
List 399, 404
Listeners 355, 497, 680
Listes 399
  itérateurs de 404
ListIterator 404
Littérales (valeurs) 99
Logiques (opérateurs) 204
Long 86, 100, 199
Longueur
  des fichiers 569
  des noms en Java 40
Look & feel xxix, 670, 752
loop() 838

M

Machine virtuelle Java xxi, 12
Macintosh xxii, xxiv, xxv, xxix, 1, 553, 669, 752
MacOS xxii, 1
MacOs 695
main() 38
Maintenance
  des applications 459, 668
MalformedURLException 833
Map 394
Marquage des objets, par le garbage collector 428
Masquage
  des identificateurs 109
    de type static 126
Masques binaires 219
Math 292
MediaTracker 813
Membres
  définition 66
  référence aux 92
  statiques 67
Mémoire
  allocation de la 460
  compactage de la 429
  désallocation de la 428
  diminution des ressources 429
  fuites de xxx, 76, 430
  libération de la 76, 429
  pile 84
  récupération de la 428
  tas 84
Memory leaks 76
Menus
  barre de 675
  création de 741
Metal (look & feel) 672, 752
Méthodes
  abstraites 310
  arguments des 71
  asynchrones 809
  branchement par appel de 232
  déclaration de 38, 70
  définition des 38, 66, 71
  finales 306
  natives 184, 309, 671
  paramètres de 38, 70, 140
  portée des identificateurs de 112
  privées 315
  protégées 314
  publiques 313
  redéfinition de 141
  retour des 176
  signatures des 145
  statiques 300
  surcharge des 143, 329
    avec un type différent 172
  surchargées
    arguments des 171
  synchronisées 308
  valeurs de retour des 168
Microsoft xxii
midi (format de fichier sons) 837
Mise en forme du code Java 38
Mnémoniques xxv
Modulo 187
Motif 669, 695, 752
Mots réservés 861
MouseEvent 734
MouseListener 734
MS-DOS 45, 553, 554
Multi-processus 77
Multimédia 695
  applications 437

Multithreading 272
Mutateurs 277, 315

N

Natif (compilateur) xxxii
native 309
Natives (méthodes) 184
Navigateur 822
  choix d'un 17
Navigator xxiii, xxiv, 18, 823, 832
Négatifs (représentation des nombres) 213
Netscape xxiii, xxiv, 18, 823, 832
new 72, 78, 223
newAudioClip() 837
newLine() 554
Newsgroups 29
Nombres
  aléatoires 377
  format des 85
Noms
  caractères autorisés dans les 40
  de variables 40
  des classes anonymes 498
  des classes imbriquées 470
  des classes locales 490
  des objets 78
  des packages 295
  des polices de caractères 801
  des threads 608
  longueur des 40
Notation
  des nombres négatifs 213
  infixée 185
notifyAll() 644
null 431

O

ObjectOutputStream 563
Objets
  anonymes 75
    dans les tableaux 368
    et garbage collector 432
    threads 614
  banque d' 318
  copie d' 576
  définition 60
  égalité des 196
  finalisation des 429
  graphiques 778
  identité des 196
  inaccessibles 427
  initialisation des 139, 326
  jetables 532
  marquage, par le garbage collector 428
  noms des 78
  portée des 113
  références d' 421
  sous-casting des 334
  sur-casting des 325
    automatique 328
    explicite 332
    implicite 328, 334
  synchronisés 309
  tableaux d' 374
  verrouillage des 626
Opérandes 187, 193
  opérateur à trois 220
Opérateurs 183
  à deux opérandes 187
  à trois opérandes 220
  à un opérande 193
  affectation 185
  arithmétiques 187
  binaires 187
  d'arithmétique binaire 207, 218
  d'auto-décrémentation 194
  d'auto-incrémentation 194
  de casting 221
  de décalage 208
  d'intialisation 89
  et sur-casting automatique 188
  instanceof 224
  logiques 204
  new 223
  priorité des 190, 224
  raccourcis 191
  relationnels 196
  unaires 187
Optimisation 147
  des applications 555
  du garbage collector 430
Ordre
  d'initialisation 165
  naturel 405
  partiel 405
  total 405
OutOfMemoryError 445
OutputStreamWriter 543
Outrepasser. *Voir* Redéfinition de méthode
OverlayLayout 701

P

pack() 708
Packages 184, 279, 283, 314
  accessibles par défaut 292
  noms des 295
  par défaut 467
Pages de code 115
paint() 731
Panel 52

Paramètres
des applets 827
des méthodes 38, 70, 140
ordre des, dans la signature des méthodes 145
passage des 571
Pascal xxvi
Passage des paramètres 571
Path 15, 33, 281
PC xxix, 115, 669
Perte d'informations
dans les castings 104
PhantomReference 445
Pile 84, 392
débordement de la xxx, 84
Pkzip 293
Plaf 752
Plantage xxix
play() 838
Pluggable Look And Feel 752
Point-virgule 45
Pointeurs 129
de fichiers 569
Polices de caractères 694, 800
génériques 801
noms des 801
policytool 840
Polymorphisme 306, 308, 323, 360, 370, 448, 506, 780
des exceptions 522
et héritage 352
et interfaces 347
et traitement des exceptions 508
vs réflexion 660
Portabilité
des applications 695
des sources xxxii
du bytecode xxxiii
Portée
des identificateurs 106
des identificateurs de méthodes 112
des indices des boucles for 252
des objets 113
Post-décrémentation 194
Post-incrémentation 194
PostScript 184
PowerJ 26
Pré-décrémentation 194
Pré-incrémentation 194
preferredLayoutSize() 707
Primitives 83, 85
anonymes 99
casting des 101
déclaration des 87
enveloppeurs de 201
équivalence des 196
graphiques 777
initialisation des 88
tableaux de 369
valeur par défaut des 91
print()
et fonctionnement du tampon d'affichage 45
println() 41
arguments de 176
PrintStream 41, 540
PrintWriter 540
Priorité
des opérateurs 190, 224
des threads 621
private 118
Procédures 206
Processeur
manipulation des registres du 83
Processus 77, 600. *Voir aussi* Threads
Profondeur (clonage en) 582
PropertyVetoException 751
Protection des applications, contre la copie 12
Prototypage xxvii
public 39, 49, 313
Pure Java xxxi, xxxii
PushBackReader 541

Q

Queues 446

R

Raccourcis (pour les opérateurs) 191
RAD 23
Random 377
RandomAccessFile 565
Rapid Application Development 23
read() 554, 568
readBoolean() 568
readChar() 568
Reader 540
readFloat() 568
readInt() 568
readLine() 515, 554
readShort() 568
Récupération
de la mémoire 76, 428
des ressources 446
Redéfinition de méthodes 141
Reference 438
ReferenceQueue 446
Références
aux classes anonymes 498
aux classes externes 482
aux classes internes 492
aux classes locales 490
aux classes membres 476
aux classes parentes 141
aux membres d'une classe 92

circulaires 428
d'objets 421
en avant 166, 491
et accessibilité 441
faibles 442
limitées 438
passage par 572
Réflexion 659
et classes internes 666
repaint() 731
reshape() xxiv
Ressources
désallocation de 446
diminution des xxx
resume() 611
Retour
d'une méthode 176
valeur de 168, 235
return 169, 235
pour sortir d'une boucle 256
rmf (format de fichier sons) 837
round() 104
rt.jar 7, 43, 293
RTF 727
RTTI 494, 651
run() 601
runFinalization() 453
runFinalizersOnExit() 455
Runnable 601, 634
Runtime 457
RunTime Type Identification 494, 653
RuntimeException 507, 527

S

Sacs 398
scr.jar 49
ScrollPaneLayout 700
Sécurité xxix, 5, 839
des applications 845
Security manager 845
Sélecteurs 265
Séparateurs 37
de niveaux
dans les chemins d'accès 44
dans les packages 280
Séquences 232
Sérialisation 309, 540, 561, 589
Serializable 563, 590
Serif 801
Set 401
setBounds() xxiv, 678
setStroke() 798
setVisible() 678, 707
Short 86
Signatures 5
des constructeurs 151
des méthodes 145
Signe (extension de) 210
Simulation 3D xxxii
Sites Web
Java Boutique 29
Java Developer Connection 28
Java Developers Journal 29
java.sun.com 28
JavaWorld 29
relatifs à Java 27
Smalltalk xxvii, 184
SoftReference 438
Solaris 1, 247
Sons 837
SortedMap 396
SortedSet 402
Sources
de Java 3, 7, 43
portabilité des xxxii
Souris (événements de) 734
Sous-castings 101
des objets 334
Sous-classes 65
Sous-typage. *Voir* Sous-casting
src.jar 42
Stack 392
Standard error 44, 864
Standard output 44
start() 601, 826
static 39, 68, 297
Statiques
membres 67
variables 68
stop() 611, 826, 838
Streams 536
de caractères 540
de communication 537, 538, 541, 542, 543
de données binaires 537
de sortie 538, 542
de traitement 537, 539, 541, 543, 546
d'entrée 537, 540
String 113, 851
StringBuffer 114, 542, 859
StringWidth() 859
Structures de contrôle 231
subString() 859
super 159, 458, 584
super() 143
Sur-casting 651. *Voir aussi* Casting
automatique, avec les opérateurs 188
automatique, des primitives 214
des objets 325
automatique 328
explicite 332
implicite 328, 334
implicite des tableaux 369
Sur-typage. *Voir* Sur-casting

Surcharger
les constructeurs 151
les méthodes 143, 329
une méthode avec un type différent 172
Surface (clonage en) 582
suspend() 611
Swing xxix, 674, 727
démonstration des composants 6
Swingset 6
switch 265
Sybase 26
Symantec 26
Synchrone
garbage collector 77
mode du garbage collector 428, 460
Synchronisation
des classes 309
des méthodes 308
des objets 309
des threads 607, 623
du garbage collector 77
synchronized 272, 308, 627
System 41

T

Tableaux 364
copie de 384
de handles 374
de primitives 369
de tableaux 380
débordement de xxx
déclaration des 364
d'objets 374
anonymes 368
handles de 369
initialisation des 365
automatique 366
littéraux 367
multidimensionnels 379
sur-casting implicite des 369
taille des 365, 376
utilisés comme arguments 381
utilisés comme valeurs de retour 381
Tables 394
ordonnées 396
Tabulations 37
Tâche de fond 77
Taille
de la contentPane 676
de la pile 84
de l'écran 694
de tampon 541
par défaut 543
des boîtes de dialogue 695, 696
des caractères 695, 719, 802
des cellules des layout managers 708
des composants xxiv, 700
des fenêtres 695, 815
des images 812
des primitives 85
des tableaux 365, 376
des vecteurs 388
du tas 84
Tampons
avec la méthode print 45
pour l'accès aux fichiers 552
tailles des 541
Tas 84, 421, 460
Téléchargement du JDK 2
Temps réel xxx, xxxii, 77
Test
dans les boucles for 246
des applications 14
this 153, 477
Thread 600
ThreadGroup 609
Threads 77, 308, 600
anonymes 609
communication entre les 644
contrôle des 610
création de 601
groupes de 613
noms des 608
priorité des 621
sous formes d'objets anonymes 614
synchronisation des 607, 623
Throw 511
throw 522
Throwable 512, 532
ToolKit 809
toString() 122
Traitement d'erreur 502
et finalisation 459
transient 309, 563
Transtypage. *Voir* Casting
TreeSet 411, 422
Tris 416
Troncage (dans les castings) 104
try 162, 508
Types
des indices des boucles for 250
non signés 215
Typographie
dans l'écriture des programmes 66

U

UltraEdit 22, 32, 36, 863
Unaires (opérateurs) 187
UNICODE 87, 93, 115
Uniform Resource Locator 832

Unix xxii, xxix, 281, 553, 554, 610, 622, 669
URL 832
  accès à un 845

V

Valeurs
  de retour 168, 235
  littérales 41, 99
    syntaxe des 99
  par défaut, des primitives 91
  passage par 575
Variables
  définition 66
  d'environnement 9, 15, 282
  d'instances 139
    initialiseurs de 162
  finales 304
    initialisation des 305
  initialisation des 137
  privées 315
  protégées 314
  publiques 313
  statiques 68, 118, 297
    initialisation des 301
  transitoires 309
  volatiles 310
Vecteurs 388
Vector 388
Verrouillage des objets 626
ViewportLayout 700
Visibilité (des identificateurs) 107
Visual Café 26
Visual J++ 27
void 38, 71, 87, 168
volatile 310

W

wait() 644
waitForAll() 815
wav (format de fichier sons) 837
WeakReference 445
Web 822
while 259
Window 673
WindowAdapter 354, 685
WindowListener 354, 684
Windows xxii, xxx, 115, 281, 293, 554, 610, 622, 669, 694, 752
  base de registre 15
  jeu de caractères de 53
  noms de fichier sous 34
Windows 95 247
Windows 98 1
Windows NT 247
WindowsEvent 684
Winzip 16, 293
Wrapper 85
write() 568
Writer 540

Y

yield() 610, 619

Z

Zéro
  division par 506
  extension de 208, 214
Zip 293, 554
ZipInputStream 538
ZipOutputStream 539

Sun Microsystems, Inc.

Binary Code License Agreement

READ THE TERMS OF THIS AGREEMENT AND ANY PROVIDED SUPPLEMENTAL LICENSE TERMS (COLLECTIVELY "AGREEMENT") CAREFULLY BEFORE OPENING THE SOFTWARE MEDIA PACKAGE. BY OPENING THE SOFTWARE MEDIA PACKAGE, YOU AGREE TO THE TERMS OF THIS AGREEMENT. IF YOU ARE ACCESSING THE SOFTWARE ELECTRONICALLY, INDICATE YOUR ACCEPTANCE OF THESE TERMS BY SELECTING THE "ACCEPT" BUTTON AT THE END OF THIS AGREEMENT. IF YOU DO NOT AGREE TO ALL THESE TERMS, PROMPTLY RETURN THE UNUSED SOFTWARE TO YOUR PLACE OF PURCHASE FOR A REFUND OR, IF THE SOFTWARE IS ACCESSED ELECTRONICALLY, SELECT THE "DECLINE" BUTTON AT THE END OF THIS AGREEMENT.

1. LICENSE TO USE. Sun grants you a non-exclusive and non-transferable license for the internal use only of the accompanying software and documentation and any error corrections provided by Sun (collectively "Software"), by the number of users and the class of computer hardware for which the corresponding fee has been paid.

2. RESTRICTIONS Software is confidential and copyrighted. Title to Software and all associated intellectual property rights is retained by Sun and/or its licensors. Except as specifically authorized in any Supplemental License Terms, you may not make copies of Software, other than a single copy of Software for archival purposes. Unless enforcement is prohibited by applicable law, you may not modify, decompile, reverse engineer Software. Software is not designed or licensed for use in on-line control of aircraft, air traffic, aircraft navigation or aircraft communications; or in the design, construction, operation or maintenance of any nuclear facility. You warrant that you will not use Software for these purposes. You may not publish or provide the results of any benchmark or comparison tests run on Software to any third party without the prior written consent of Sun. No right, title or interest in or to any trademark, service mark, logo or trade name of Sun or its licensors is granted under this Agreement.

3. LIMITED WARRANTY. Sun warrants to you that for a period of ninety (90) days from the date of purchase, as evidenced by a copy of the receipt, the media on which Software is furnished (if any) will be free of defects in materials and workmanship under normal use. Except for the foregoing, Software is provided "AS IS". Your exclusive remedy and Sun's entire liability under this limited warranty will be at Sun's option to replace Software media or refund the fee paid for Software.

4. DISCLAIMER OF WARRANTY. UNLESS SPECIFIED IN THIS AGREEMENT, ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT THESE DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

5. LIMITATION OF LIABILITY. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. In no event

will Sun's liability to you, whether in contract, tort (including negligence), or otherwise, exceed the amount paid by you for Software under this Agreement. The foregoing limitations will apply even if the above stated warranty fails of its essential purpose.

6. Termination. This Agreement is effective until terminated. You may terminate this Agreement at any time by destroying all copies of Software. This Agreement will terminate immediately without notice from Sun if you fail to comply with any provision of this Agreement. Upon Termination, you must destroy all copies of Software.

7. Export Regulations. All Software and technical data delivered under this Agreement are subject to US export control laws and may be subject to export or import regulations in other countries. You agree to comply strictly with all such laws and regulations and acknowledge that you have the responsibility to obtain such licenses to export, re-export, or import as may be required after delivery to you.

8. U.S. Government Restricted Rights. Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in this Agreement and as provided in DFARS 227.7202-1 (a) and 227.7202-3(a) (1995), DFARS 252.227-7013 (c)(1)(ii)(Oct 1988), FAR 12.212 (a) (1995), FAR 52.227-19 (June 1987), or FAR 52.227-14(ALT III) (June 1987), as applicable.

9. Governing Law. Any action related to this Agreement will be governed by California law and controlling U.S. federal law. No choice of law rules of any jurisdiction will apply.

10. Severability. If any provision of this Agreement is held to be unenforceable, This Agreement will remain in effect with the provision omitted, unless omission would frustrate the intent of the parties, in which case this Agreement will immediately terminate.

11. Integration. This Agreement is the entire agreement between you and Sun relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification of this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

For inquiries please contact: Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California 94303

JAVATM DEVELOPMENT KIT VERSION 1.2 AND HOTJAVA BROWSER 1.1.X

SUPPLEMENTAL LICENSE TERMS

These supplemental terms ("Supplement") add to the terms of the Binary Code License Agreement ("Agreement"). Capitalized terms not defined herein shall have the same meanings ascribed to them in the Agreement. The Supplement terms shall supersede any inconsistent or conflicting terms in the Agreement.

1. Limited License Grant. Sun grants to you a non-exclusive, non-transferable limited license to use the Software without fee for the execution and development of Java? applets and applications provided that you: (i) may not redistribute the Software, in whole or in part, either separately or included with a product. (ii) may not create, or authorize your licensees to create additional classes, interfaces, or subpackages that are contained in the "java" or "sun" packages or similar as specified by Sun in any class file naming convention; and (iii) agree to the extent Programs are developed which utilize the Windows 95/98 style graphical user interface or components contained therein, such applets or applications may only be developed to run on a Windows 95/98 or Windows NT platform. Refer to the Java Runtime Environment Version 1.2 binary code license (http://java.sun.com/products/JDK/1.2/index.html) for the availability of runtime code which may be distributed with Java applets and applications.

2. Java Platform Interface. In the event that Licensee creates an additional API(s) which: (i) extends the functionality of a Java Environment; and, (ii) is exposed to third party software developers for the purpose of developing additional software which invokes such additional API, Licensee must promptly publish broadly an accurate specification for such API for free use by all developers.

3. Trademarks and Logos. This Agreement does not authorize Licensee to use any Sun name, trademark or logo. Licensee acknowledges as between it and Sun that Sun owns the Java trademark and all Java-related trademarks, logos and icons including the Coffee Cup and Duke ("Java Marks") and agrees to comply with the Java Trademark Guidelines at http://java.sun.com/trademarks.html.

4. High Risk Activities. Notwithstanding Section 2, with respect to high risk activities, the following language shall apply: the Software is not designed or intended for use in on-line control of aircraft, air traffic, aircraft navigation or aircraft communications; or in the design, construction, operation or maintenance of any nuclear facility. Sun disclaims any express or implied warranty of fitness for such uses.