

CampusPress Référence

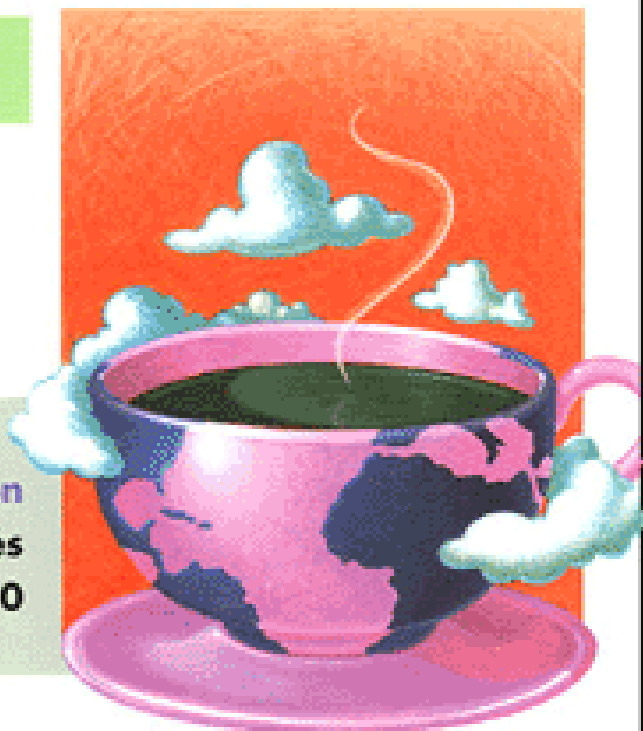
Volume
2

Au cœur de **Java™** Fonctions avancées

JDK 5.0



Nouvelle édition
Les dernières nouveautés
du J2SE™ 5.0



Cay S. Horstmann & Gary Cornell

Réseaux
et télécom

Développement

Génie logiciel

Sécurité

Système
d'exploitation


CampusPress
www.pearsoneducation.fr

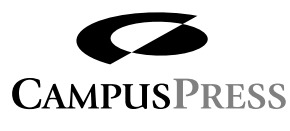


Au cœur de Java 2

volume 2

Fonctions avancées

**Cay S. Horstmann
et Gary Cornell**



CampusPress a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, CampusPress n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

CampusPress ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Publié par CampusPress
47 bis, rue des Vinaigriers
75010 PARIS
Tél. : 01 72 74 90 00

Titre original : *Core Java 2, volume 2 - Advanced Features*

Traduit de l'américain par :
Marie-Cécile Baland et Nathalie Le Guillou de Penanros

Mise en pages : TyPAO

ISBN : 2-7440-1962-3
Copyright © 2005 CampusPress
Tous droits réservés

CampusPress est une marque
de Pearson Education France

ISBN original : 0-13-111826-9
Copyright © 2005 Sun Microsystems, Inc.
Tous droits réservés

Sun Microsystems Inc.
4150 Network Circle, Santa Clara,
95054 Californie, USA

Aucune représentation ou reproduction, même partielle, autre que celles prévues à l'article L. 122-5 2° et 3° a) du code de la propriété intellectuelle ne peut être faite sans l'autorisation expresse de Pearson Education France ou, le cas échéant, sans le respect des modalités prévues à l'article L. 122-10 dudit code.

Table des matières

Introduction. Quelques mots au lecteur	1
A propos de ce livre	1
Conventions	3
Chapitre 1. Multithreads	5
Qu'est-ce qu'un thread ?	6
Utiliser des threads pour laisser une chance aux autres tâches	11
Interrompre des threads	17
Les états d'un thread	20
Threads morts	23
Propriétés d'un thread	24
Priorités d'un thread	24
Threads démons	25
Groupes de threads	25
Gestionnaire d'exceptions non récupérées	27
Synchronisation	28
Exemple de condition de course	28
Explication des conditions de course	32
Verrous d'objet	33
Objets de condition	36
Le mot clé synchronized	41
Blocs synchronisés	47
Verrous morts	48
Equité	50
Pourquoi les méthodes stop et suspend ne sont plus utilisées	50
Queues de blocage	52
Collections compatibles avec les threads	58
CopyOnWriteArray	58
Callable et Future	59

IV Table des matières

Executors	63
Pools de threads	63
Exécution programmée	67
Synchronizers	68
Barrières	69
Verrous Countdown	69
Exchanger	70
Queues synchrones	70
Sémaphores	70
Chapitre 2. Collections	77
Les interfaces de collection	78
Séparer les interfaces d'une collection et leur implémentation	78
Interfaces de collection et d'itération dans la bibliothèque Java	80
Suppression d'éléments	83
Méthodes utilitaires génériques	83
Les collections concrètes	85
Listes chaînées	86
Listes de tableaux	94
Tables de hachage	95
Arbres	98
Queues de priorité	104
Cartes	105
Classes de cartes et de set spécialisées	109
La structure des collections	114
Les vues et les emballages	117
Les opérations de masse	123
Conversion entre collections et tableaux	124
Extension du cadre	125
Algorithmes	127
Trier et mélanger	129
Recherche binaire	131
Algorithmes simples	132
Ecrire vos propres algorithmes	133
Les anciennes collections	135
La classe Hashtable	135
Les énumérations	135
Ensembles de propriétés	136
Piles	137
Les ensembles de bits	137

Chapitre 3. Programmation des bases de données	143
Conception de JDBC	144
Types de pilotes JDBC	146
Applications typiques de JDBC	147
SQL	148
Installation de JDBC	153
Principaux concepts de programmation JDBC	154
URL de bases de données	154
Etablir une connexion	154
Exécuter des commandes SQL	159
Types SQL avancés	161
Gestion des connexions, instructions et jeux de résultats	163
Remplir une base de données	164
Exécution de requêtes	168
Instructions préparées	168
Ensembles de résultats défilants et actualisables	177
Ensembles de résultats défilants	178
Ensembles de résultats actualisables	180
Métadonnées	184
RowSet	194
CachedRowSet	194
Transactions	202
Points de sauvegarde	203
Mises à jour automatisées	204
Gestion avancée des connexions	206
Introduction au LDAP	207
Configurer un serveur LDAP	208
Accéder aux informations du répertoire LDAP	212
Chapitre 4. Objets distribués	223
Les rôles du client et du serveur	224
Invocations de méthodes distantes	227
Stubs et encodage des paramètres	227
Charger des classes dynamiquement	229
Etablissement d'une invocation de méthode distante	230
Interfaces et implémentations	230
Génération de classe de stub	233
Localiser des objets de serveur	234
Du côté client	238

VI Table des matières

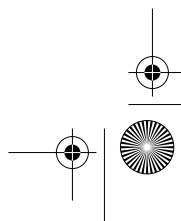
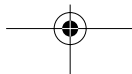
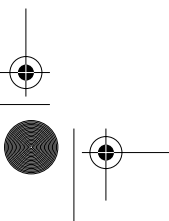
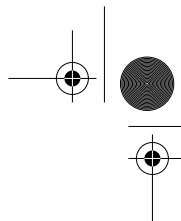
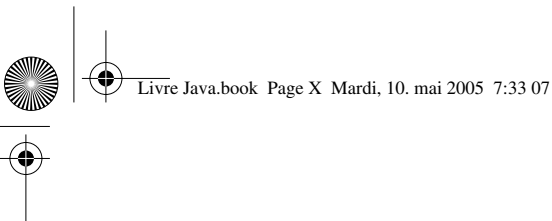
Préparer la mise en œuvre	243
Mise en œuvre du programme	246
Passage de paramètres aux méthodes distantes	248
Passer des objets locaux	248
Passer des objets distants	259
Objets distants et méthodes equals et hashCode	262
Cloner des objets distants	262
Activation des objets du serveur	263
IDL Java et CORBA	268
Langage de définition d'interfaces	270
Un exemple en CORBA	274
Implémenter des serveurs CORBA	283
Appels de méthode distante avec SOAP	288
Chapitre 5. Swing	295
Listes	295
Le composant JList	296
Modèles de listes	302
Insérer et supprimer des valeurs	306
Afficher des valeurs	308
Arbres	312
Exemples d'arbres	314
Enumération de nœuds	330
Afficher les nœuds	332
Ecouter les événements des arbres	338
Modèles d'arbre personnalisés	344
Tableaux	352
Un tableau simple	352
Modèles de tableaux	356
Affichage et modification des cellules	370
Travailler avec les lignes et les colonnes	384
Composants de texte stylisés	393
Indicateurs de progression	399
Barres de progression	399
Contrôleurs de progression	404
Surveiller la progression des flux d'entrée	408
Organisateurs de composants	414
Séparateurs	414
Onglets	418
Panneaux de bureau et fenêtres internes	423

Chapitre 6. JavaBeans™	441
Pourquoi les beans ?	442
Le processus d'écriture des beans	443
Construire une application à l'aide des beans	446
Intégrer des beans dans des fichiers JAR	446
Composer des beans dans un environnement de génération	448
Les modèles de nom pour les propriétés et événements de bean	453
Les types de propriétés de bean	456
Les propriétés simples	456
Les propriétés indexées	456
Les propriétés liées	457
Propriétés contraintes	459
Les classes BeanInfo	465
Les éditeurs de propriétés	470
Ecrire un éditeur de propriétés	477
Les "customizers"	491
Ecrire une classe Customizer	493
La persistance des JavaBeans	500
Utiliser la persistance des JavaBeans pour des données arbitraires	504
Un exemple complet de persistance des JavaBeans	510
Chapitre 7. La sécurité	523
Les chargeurs de classe	524
Utiliser des chargeurs de classe comme espaces de nom	526
Ecrire votre propre chargeur de classe	527
La vérification des bytecodes	532
Les gestionnaires de sécurité et les permissions	537
La sécurité de la plate-forme Java 2	539
Les fichiers de règles de sécurité	542
Les permissions personnalisées	549
Implémenter une classe de permissions	550
Un gestionnaire de sécurité personnalisé	556
Authentifier les utilisateurs	563
Modules d'identification JAAS	568
Signatures numériques	577
Les condensés de message	578
Les signatures numériques	583
L'authentification des messages	591
Le format de certification X.509	593

VIII Table des matières

Générer des certificats	596
Signer des certificats	599
La signature de code	606
Signer des fichiers JAR	606
Les certificats de développeur de logiciel	610
Le cryptage	612
Chiffres symétriques	612
Flux de chiffres	618
Chiffres de clés publiques	619
Chapitre 8. Internationalisation	625
Les paramètres régionaux	626
Le format des nombres	632
Devises	637
La date et l'heure	638
Classement	646
Formatage de message	653
Formats de choix	655
Fichiers texte et jeux de caractères	657
Codage de caractères des fichiers source	658
Les groupes de ressources	659
Trouver l'emplacement des groupes de ressources	659
Fichiers de propriétés	660
Classes de groupes	661
Un exemple complet	663
Chapitre 9. Méthodes natives	677
Appeler une fonction C à partir du langage Java	679
Travailler avec la fonction printf	680
Les paramètres numériques et les valeurs renvoyées	685
Utiliser printf pour le formatage de nombres	685
Les paramètres chaîne	686
Appeler sprintf dans une méthode native	690
Accéder aux champs	692
Accéder aux champs d'instance	692
Accéder aux champs statiques	697
Coder les signatures	697
Appeler les méthodes Java	699
Les méthodes non statiques	699
Les méthodes statiques	700

Les constructeurs	701
Autres possibilités d'invocation de méthodes	702
Accéder aux éléments de tableaux	706
La gestion des erreurs	711
L'API d'invocation	715
Un exemple exhaustif : accès à la base de registres de Windows	719
Aperçu de la base de registres de Windows	719
Une interface de plate-forme Java pour accéder à la base de registres	720
Implémenter les fonctions d'accès à la base de registres avec un code natif	721
Chapitre 10. XML	735
Une introduction à XML	736
La structure d'un document XML	738
Analyse d'un document XML	741
Valider les documents XML	752
Définitions du type de document (DTD)	753
XML Schema	760
Un exemple pratique	763
Localiser des informations avec XPath	776
Espaces de noms	782
L'analyseur SAX	784
Génération de documents XML	789
Transformations XSL	797
Chapitre 11. Annotations	807
Ajout de métadonnées aux programmes	808
Un exemple : annotation des gestionnaires d'événements	809
Syntaxe des annotations	814
Annotations standard	818
Annotations ordinaires	819
Méta-annotations	819
L'outil apt pour le traitement d'annotations au niveau de la source	822
Ingénierie des bytecodes	829
Modifier les bytecodes au moment du chargement	837
Index	841



Introduction

Quelques mots au lecteur

Le livre que vous tenez entre les mains est le second volume de la cinquième édition de l'ouvrage *Au cœur de Java*. Le premier volume traite les caractéristiques essentielles du langage. Il aborde notamment les sujets avancés qu'un programmeur doit connaître pour le développement d'un logiciel professionnel. Dans le présent volume, comme pour le premier volume et les éditions précédentes de ce livre, *nous nous adressons donc aux programmeurs qui veulent utiliser le langage Java sur des projets réels*.

Important : si vous faites partie des développeurs expérimentés qui sont à l'aise avec les nouveaux modèles d'événements et les caractéristiques avancées de ce langage, il n'est pas nécessaire de lire le premier volume pour profiter du présent volume. Cependant, en cas de besoin, nous faisons référence à des sections du premier volume, et nous espérons que vous l'achèterez si ce n'est déjà fait. De plus, vous trouverez toutes les informations générales nécessaires dans n'importe quel livre d'introduction à la plate-forme Java.

Enfin, lorsqu'on écrit un livre, les erreurs et les imprécisions sont inévitables. Nous serions très heureux que vous nous les rapportiez. Naturellement, nous préfererions ne les voir mentionnées qu'une seule fois, c'est pourquoi nous avons créé un site Web sur <http://www.horstmann.com/corejava.html>, avec une FAQ, des corrections d'erreurs et des remarques générales. Un formulaire, stratégiquement placé à la fin de la page Web des erreurs signalées (pour vous encourager à lire les erreurs déjà signalées), peut être utilisé pour indiquer les erreurs ou les problèmes et pour envoyer des suggestions destinées à améliorer les éditions futures.

A propos de ce livre

Les chapitres de ce livre sont, pour la plupart, indépendants les uns des autres. Vous devriez pouvoir passer directement au sujet qui vous intéresse le plus et donc lire les chapitres dans n'importe quel ordre.

Le Chapitre 1 aborde les *multithreads*, qui vous permettent d'exécuter des tâches en parallèle. Un thread est un flot de contrôle à l'intérieur d'un programme. Nous vous montrerons comment définir des threads et traiter leur synchronisation. Le multithreading a bien évolué dans le JDK 5.0, dont nous vous présenterons donc les nouveaux mécanismes.

Le sujet abordé dans le Chapitre 2 est le *cadre des collections* de la plate-forme Java 2. Lorsque vous voulez rassembler plusieurs objets pour les récupérer plus tard, vous pouvez utiliser une collection adaptée à vos besoins, au lieu de vous contenter d'assembler ces éléments dans un *Vector*.

2 Au cœur de Java 2 - Fonctions avancées

Ce chapitre vous montre comment tirer profit des collections standard qui ont été préétablies à votre intention. Le chapitre a été complètement révisé du fait des collections génériques du JDK 5.0.

Le Chapitre 3 se concentre sur la programmation des bases de données. Il parle de *JDBC*, l'API de connectivité de base de données de Java, qui permet de se connecter à des bases de données relationnelles. Nous vous montrerons comment écrire des programmes pratiques pour traiter les tâches courantes des bases de données, en utilisant un sous-ensemble de l'API JDBC. Une étude complète de l'API JDBC nécessiterait un livre entier, presque aussi long que celui-ci. Nous terminerons ce chapitre par une brève introduction aux bases de données hiérarchiques et verrons le JNDI et le protocole LDAP.

Le Chapitre 4 comprend les *objets distribués*. Nous parlerons en détail duRMI (*Remote Method Invocation*). Cette API vous permet de travailler avec des objets Java qui sont distribués sur plusieurs machines. Nous verrons aussi brièvement CORBA (*Common Object Request Broker Architecture*) et vous montrerons comment des objets écrits en C++ et en Java peuvent communiquer. Nous terminerons le chapitre par une discussion sur SOAP (*Simple Object Access Protocol*) et vous montrerons un exemple dans lequel un programme Java communique avec le service Web d'Amazon.

Le Chapitre 5 contient toutes les techniques de Swing qui n'avaient pas leur place dans le Volume 1, spécialement les composants d'arbres et de tables, importants mais complexes. Nous vous montrerons les utilisations principales des panneaux d'édition et l'implémentation Java d'une interface à "plusieurs documents" ainsi que les indicateurs de progression utilisés dans des programmes avec multithread. Là encore, nous nous concentrerons sur les constructions les plus utiles que vous pourriez rencontrer dans la pratique, puisqu'une présentation de la bibliothèque Swing dans son intégralité nécessiterait plusieurs volumes et ne serait intéressante que pour des taxinomistes spécialisés.

Le Chapitre 6 vous montre ce que vous devez savoir sur l'API des composants de la plate-forme Java : *JavaBeans*. Vous verrez comment écrire vos propres beans, que d'autres programmeurs pourront manipuler et intégrer dans leurs environnements de construction intégrés. Nous terminerons ce chapitre en vous montrant comment utiliser la persistance des JavaBeans pour stocker vos données dans un format adapté à une conservation à long terme, contrairement à la sérialisation.

Le Chapitre 7 aborde le modèle de sécurité. La plate-forme Java a été conçue dès le départ pour être sûre, et ce chapitre vous amène dans ses coulisses pour vous faire voir comment cette architecture a été implémentée. Nous vous montrerons comment écrire vos propres chargeurs de classe et des gestionnaires de sécurité pour des applications ciblées. Ensuite, nous étudierons la nouvelle API de sécurité, qui vous permet d'utiliser des caractéristiques aussi importantes que les messages et codes signés, l'autorisation, l'authentification et le cryptage. Ce chapitre a été totalement mis à jour pour le JDK 5.0, de manière à prendre en compte les algorithmes de cryptage AES et RSA.

Le Chapitre 8 parle d'une caractéristique spécialisée qui, nous semble-t-il, ne peut que prendre de l'importance : l'internationalisation. Le langage de programmation Java fait partie des quelques langages qui ont été conçus dès le départ pour gérer Unicode, mais le support de l'internationalisation de la plate-forme Java va bien plus loin. Par conséquent, vous pouvez internationaliser des applications Java de sorte qu'elles fonctionnent non seulement sur plusieurs plates-formes mais aussi dans plusieurs pays. Par exemple, nous vous montrerons comment écrire un applet de calcul de retraite en anglais, en allemand ou en chinois, en fonction de la langue du navigateur.

Le Chapitre 9 aborde les *méthodes natives*, qui vous permettent d'appeler des méthodes écrites pour une machine spécifique, comme l'API de Microsoft Windows. Bien sûr, cette caractéristique peut

être controversée : dès que des méthodes natives sont utilisées, la caractéristique interplate-forme de Java disparaît. Cependant, tous les programmeurs sérieux qui écrivent des applications Java pour des plates-formes spécifiques doivent connaître ces techniques. Pour écrire des applications sérieuses, il vous faudra parfois passer par l'API du système d'exploitation de votre plate-forme cible. Nous illustrerons cette technique en vous montrant comment accéder aux fonctions de la base de registre sous Windows.

Le Chapitre 10 traite du XML. Nous vous montrerons comment analyser les fichiers XML, générer du XML et utiliser les transformations XSL. A titre d'exemple, nous vous montrerons comment spécifier la mise en page d'un formulaire Swing en XML. Ce chapitre a été mis à jour pour inclure l'API XPath qui permet de retrouver plus facilement une aiguille dans la botte de foin XML.

Enfin, le Chapitre 11 a été ajouté à cette édition. Il traite des annotations et des métadonnées, des fonctionnalités ajoutées au JDK 5.0. Les annotations vous permettent d'ajouter des informations arbitraires (les métadonnées) à un programme Java. Nous vous montrerons la manière dont les outils de traitement collectent ces annotations à la source ou au niveau du fichier de classe et comment utiliser les annotations pour influencer le comportement des classes au moment de l'exécution. Les annotations ne servent qu'accompagnées d'outils, et nous espérons que cette discussion vous aidera à choisir ceux qui sont adaptés à vos besoins.

Conventions

Comme dans de nombreux livres d'informatique, nous utilisons une police courier pour représenter le code.

INFO

Ces informations sont signalées par une icône de bloc-notes qui ressemble à ceci.

ASTUCE

Les astuces utiles sont signalées par cette icône.

ATTENTION

Les notes qui vous mettent en garde contre un piège ou une situation dangereuse sont signalées par une icône "Attention".

INFO C++

Il existe un certain nombre de notes C++ qui expliquent la différence entre les langages de programmation Java et C++. Vous pouvez simplement les sauter si C++ ne vous intéresse pas.

Interface de programmation d'application (API)

La plate-forme Java possède une vaste bibliothèque de programmation, appelée aussi API (*Application Programming Interface*). Lorsque nous utilisons un appel d'API pour la première fois, nous

4 *Au cœur de Java 2 - Fonctions avancées*

ajoutons une petite description signalée par une icône API. Ces descriptions sont un peu moins formelles mais également plus riches que celles de la documentation de l'API officielle en ligne.

Les programmes dont le code source se trouve dans le code d'accompagnement de cet ouvrage sont présentés comme des exemples. Ainsi, **Exemple 4.8 : *WarehouseServer.java*** fait référence au code que vous pouvez télécharger à l'adresse **<http://www.pearsoneducation.fr>**.

1

Multithreads

Au sommaire de ce chapitre

- ✓ Qu'est-ce qu'un thread ?
- ✓ Interruption des threads
- ✓ Les états d'un thread
- ✓ Propriétés d'un thread
- ✓ Synchronisation
- ✓ Verrous morts
- ✓ Collections compatibles avec les threads
- ✓ Callable et Future
- ✓ Executor
- ✓ Synchronizer

Vous connaissez probablement déjà le *multitâche* : la possibilité d'avoir plusieurs programmes travaillant en même temps. Par exemple, avec un système multitâche, il est possible d'imprimer un document en même temps que vous en modifiez un autre ou que vous envoyez un fax. Naturellement, à moins que vous ne possédiez un ordinateur multiprocesseur, le système d'exploitation est obligé de partager les ressources du processeur, ce qui donne l'illusion d'une activité parallèle. Cette répartition des ressources est possible parce que la plupart des programmes ne se servent pas de l'intégralité du temps machine. Par exemple, lorsqu'un utilisateur saisit des données rapidement, il ne prend qu'un vingtième de seconde par caractère.

Il existe deux sortes de multitâches. Le premier, appelé multitâche *préemptif*, interrompt les programmes sans les consulter. Le second est appelé multitâche *coopératif*, ou *non préemptif* : chaque programme peut être interrompu lorsqu'il en fournit explicitement l'autorisation. Windows 3.x et Mac OS 9 constituent des systèmes d'exploitation coopératifs, alors que UNIX/Linux, Windows NT/XP (et Windows 9x pour les programmes 32 bits) et Mac OS X sont préemptifs. Bien que les systèmes d'exploitation préemptifs soient plus difficiles à implémenter, ils sont bien

6 Au cœur de Java 2 - Fonctions avancées

plus efficaces. Avec un multitâche coopératif, un programme mal conçu peut bloquer le système indéfiniment.

Les programmes utilisant plusieurs threads développent l'idée du multitâche en l'implémentant à un niveau plus bas : des programmes individuels peuvent effectuer plusieurs tâches en même temps. Chaque tâche est traditionnellement appelée un *thread*. Les programmes qui peuvent exécuter plusieurs threads en même temps sont appelés des programmes à *multithreads*.

Quelle est donc la différence entre plusieurs *processus* et plusieurs *threads* ? La différence essentielle est qu'un processus possède une copie unique de ses propres variables, alors que les threads partagent les mêmes données. Cela peut paraître risqué, et cela l'est parfois, comme nous allons le voir un peu plus loin dans ce chapitre. Mais il est bien plus rapide de créer et de détruire des threads individuels que de créer un nouveau processus. C'est pourquoi tous les systèmes d'exploitation modernes supportent les multithreads. De plus, la communication entre les processus est bien plus lente et plus restrictive qu'entre des threads.

Les multithreads sont très utiles dans la pratique. Par exemple, un navigateur doit pouvoir télécharger simultanément plusieurs pages, et un serveur Web doit pouvoir répondre à plusieurs requêtes simultanées. Le langage de programmation Java lui-même se sert d'un thread pour gérer le ramasse-miettes en tâche de fond, ce qui vous épargne la tâche de gérer la mémoire vous-même ! Les programmes ayant une interface utilisateur graphique possèdent un thread séparé pour récupérer les événements de l'interface utilisateur. Ce chapitre vous montrera comment ajouter des capacités de multithread dans vos applications Java.

Le multithread a considérablement changé dans le JDK 5.0, du fait de l'ajout d'un grand nombre de classes et d'interfaces apportant des implémentations de grande qualité. Celles-ci sont destinées aux mécanismes nécessaires à la plupart des programmeurs d'applications. Dans ce chapitre, nous verrons les nouvelles fonctionnalités du JDK 5.0 ainsi que les mécanismes classiques de synchronisation, et nous vous aiderons à faire votre choix.

Un petit conseil : la gestion de plusieurs threads peut rapidement devenir très complexe. Dans ce chapitre, nous présentons tous les outils fournis par le langage de programmation Java pour programmer ces threads. Cependant, pour les situations plus délicates, nous vous suggérons de passer par des ouvrages plus spécialisés, comme *Concurrent Programming in Java* de Doug Lea (Addison-Wesley, 1999).

Qu'est-ce qu'un thread ?

Commençons par étudier un programme qui ne se sert pas de plusieurs threads et qui, par conséquent, empêche l'utilisateur d'effectuer plusieurs tâches avec ce programme. Une fois que nous l'aurons disséqué, nous vous montrerons combien il est facile de l'exécuter dans plusieurs threads. Ce programme fait rebondir une balle en la déplaçant en permanence. Si la balle arrive contre un mur, il la fait rebondir (voir Figure 1.1).

Dès que vous cliquez sur le bouton "Démarrer", le programme lance la balle à partir du coin supérieur gauche de l'écran et celle-ci commence à rebondir. Le gestionnaire du bouton "Démarrer" appelle la méthode `addBall`. Cette méthode contient une boucle progressant sur 1 000 déplacements. Chaque appel à `move` déplace légèrement la balle, ajuste la direction si elle rebondit sur un mur, puis redessine l'écran.

```

Ball ball = new Ball();
panel.add(ball);

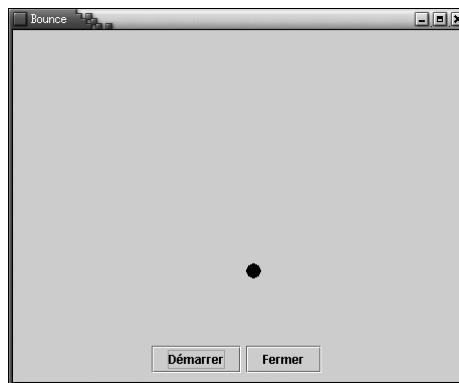
for (int i = 1; i <= STEPS; i++)
{
    ball.move(panel.getBounds());
    panel.paint(panel.getGraphics());
    Thread.sleep(DELAY);
}

```

La méthode `sleep` statique de la classe `Thread` effectue une pause du nombre de millisecondes donné.

Figure 1.1

Utilisation d'un thread pour animer une balle rebondissante.



L'appel à `Thread.sleep` ne crée pas un nouveau thread, `sleep` est une méthode statique de la classe `Thread` qui met temporairement le thread courant en sommeil.

La méthode `sleep` peut déclencher une exception `InterruptedException`. Nous reviendrons sur cette exception et sur son gestionnaire un peu plus tard. Pour l'instant, nous terminons simplement le rebond si l'exception survient.

Si vous exécutez ce programme, vous verrez que la balle rebondit très bien, mais qu'elle bloque complètement l'application. Si ces rebonds vous lassent et que vous ne voulez pas attendre les 1 000 déplacements, cliquez sur le bouton "Fermer". La balle continue quand même son trajet. Il est impossible d'interagir avec le programme tant que la balle rebondit.

INFO

Si vous étudiez attentivement le code à la fin de cette section, vous remarquerez l'appel

```
panel.paint(panel.getGraphics())
```

dans la méthode `move` de la classe `Ball`. Ceci est assez étrange, vous devriez normalement appeler `panel.repaint` et laisser le soin à AWT d'obtenir le contexte graphique et de dessiner l'écran. Toutefois, si vous essayez d'appeler `canvas.repaint()` dans ce programme, vous découvrirez que l'écran n'est jamais redessiné puisque la méthode `addBall` a totalement pris le pas sur le traitement. Dans le programme suivant, nous utiliserons un autre thread pour calculer la position de la balle, et nous emploierons à nouveau `repaint`.

8 Au cœur de Java 2 - Fonctions avancées

A l'évidence, ce n'est pas une situation acceptable, et l'on ne souhaite pas que les programmes utilisés se comportent de cette manière lorsqu'ils doivent effectuer des tâches longues. Après tout, lorsque vous lisez des données en passant par une connexion réseau, il est trop courant d'être bloqué par une tâche gourmande en temps machine, que vous souhaiteriez *vraiment* interrompre. Par exemple, supposons que vous chargiez une grande image et que vous décidiez, après en avoir vu une partie, que vous ne voulez pas voir le reste. Il peut alors être très intéressant de pouvoir cliquer sur un bouton "Arrêter" ou "Précédent" pour interrompre le chargement. Dans la prochaine section, nous vous montrerons comment laisser cette possibilité à l'utilisateur en exécutant les parties critiques d'un programme dans un *thread* séparé.

L'Exemple 1.1 présente le code de ce programme.

Exemple 1.1 : *Bounce.java*

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.util.*;
import javax.swing.*;

/**
 * Présente une balle rebondissante animée.
 */
public class Bounce
{
    public static void main(String[] args)
    {
        JFrame frame = new BounceFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Une balle qui bouge et rebondit sur les bords
 * d'un rectangle
 */
class Ball
{
    /**
     * Déplace la balle à la position suivante, en inversant la direction
     * si elle touche l'un des bords
     */
    public void move(Rectangle2D bounds)
    {
        x += dx;
        y += dy;
        if (x < bounds.getMinX())
        {
            x = bounds.getMinX();
            dx = -dx;
        }
        if (x + XSIZE >= bounds.getMaxX())
        {
            x = bounds.getMaxX() - XSIZE;
            dx = -dx;
        }
        if (y < bounds.getMinY())
        {
            y = bounds.getMinY();
            dy = -dy;
        }
        if (y + YSIZE >= bounds.getMaxY())
        {
            y = bounds.getMaxY() - YSIZE;
            dy = -dy;
        }
    }
}
```

```

        x = bounds.getMaxX() - XSIZE;
        dx = -dx;
    }
    if (y < bounds.getMinY())
    {
        y = bounds.getMinY();
        dy = -dy;
    }
    if (y + YSIZE >= bounds.getMaxY())
    {
        y = bounds.getMaxY() - YSIZE;
        dy = -dy;
    }
}

/**
 * Récupère la forme de la balle à sa position actuelle.
 */
public Ellipse2D getShape()
{
    return new Ellipse2D.Double(x, y, XSIZE, YSIZE);
}

private static final int XSIZE = 15;
private static final int YSIZE = 15;
private double x = 0;
private double y = 0;
private double dx = 1;
private double dy = 1;
}

/**
 * L'écran où se dessinent les balles.
 */
class BallPanel extends JPanel
{
    /**
     * Ajouter une balle à l'écran.
     * @param b la balle à ajouter
     */
    public void add(Ball b)
    {
        balls.add(b);
    }

    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        for (Ball b : balls)
        {
            g2.fill(b.getShape());
        }
    }

    private ArrayList<Ball> balls = new ArrayList<Ball>();
}

```


10 Au cœur de Java 2 - Fonctions avancées

```
/**
 * Le bloc avec l'écran et les boutons.
 */
class BounceFrame extends JFrame
{
    /**
     * Elabore le bloc de l'écran pour montrer la balle rebondissante et
     * les boutons Démarrer et Fermer.
     */
    public BounceFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
        setTitle("Bounce");

        panel = new BallPanel();
        add(panel, BorderLayout.CENTER);
        JPanel buttonPanel = new JPanel();
        addButton(buttonPanel, "Démarrer",
            new ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    addBall();
                }
            });

        addButton(buttonPanel, "Fermer",
            new ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    System.exit(0);
                }
            });
        add(buttonPanel, BorderLayout.SOUTH);
    }

    /**
     * Ajoute un bouton à un conteneur.
     * @param c le conteneur
     * @param title le nom du bouton
     * @param listener l'écouteur d'action pour le bouton
     */
    public void addButton(Container c, String title,
        ActionListener listener)
    {
        JButton button = new JButton(title);
        c.add(button);
        button.addActionListener(listener);
    }

    /**
     * Ajoute une balle rebondissante à l'écran et la fait
     * rebondir 1 000 fois.
     */
    public void addBall()
    {
        try
        {

```

```

        Ball ball = new Ball();
        panel.add(ball);

        for (int i = 1; i <= STEPS; i++)
        {
            ball.move(panel.getBounds());
            panel.paint(panel.getGraphics());
            Thread.sleep(DELAY);
        }
    }
    catch (InterruptedException e)
    {
    }
}

private BallPanel panel;
public static final int DEFAULT_WIDTH = 450;
public static final int DEFAULT_HEIGHT = 350;
public static final int STEPS = 1000;
public static final int DELAY = 3;
}

```

java.lang.Thread 1.0

- `static void sleep(long millis)`

Mise en veille pendant le nombre donné de millisecondes.

Paramètres : `millis` le nombre de millisecondes de mise en veille

Utiliser des threads pour laisser une chance aux autres tâches

Nous allons rendre notre programme de balles plus attentif à l'utilisateur en exécutant le code qui déplace la balle dans un thread séparé. Vous pourrez en fait lancer plusieurs balles. Chacune est déplacée par son propre thread. En outre, le thread de distribution d'événements AWT continue à fonctionner en parallèle, il s'occupe des événements de l'interface utilisateur. Chaque thread ayant une occasion de s'exécuter, le thread principal peut savoir si l'utilisateur clique sur le bouton Fermer alors que les balles rebondissent. Il peut alors traiter l'action correspondante.

Voici une procédure simple pour exécuter une tâche dans un thread séparé :

1. Placez le code de la tâche dans la méthode `run` d'une classe qui implémente l'interface `Runnable`. Cette interface est très simple, elle ne possède qu'une méthode :

```

public interface Runnable
{
    void run();
}

```

Vous implémentez simplement une classe comme ceci :

```

class MyRunnable implements Runnable
{
    public void run()
    {
        code de la tâche
    }
}

```

12 Au cœur de Java 2 - Fonctions avancées

2. Construisez un objet de votre classe :

```
Runnable r = new MyRunnable();
```

3. Construisez un objet Thread à partir de l'interface Runnable :

```
Thread t = new Thread(r);
```

4. Démarrez le thread :

```
t.start();
```

Pour exécuter notre programme dans un thread séparé, nous devons simplement implémenter une classe BallRunnable et placer le code de l'animation dans la méthode run, comme dans le code suivant :

```
class BallRunnable implements Runnable
{
    . . .
    public void run()
    {
        try
        {
            for (int i = 1; i <= STEPS; i++)
            {
                ball.move(component.getBounds());
                component.repaint();
                Thread.sleep(DELAY);
            }
        }
        catch (InterruptedException exception)
        {
        }
    }
    . . .
}
```

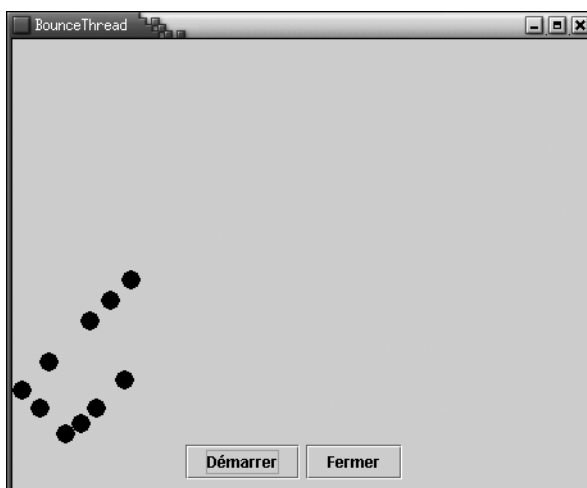
Une fois de plus, nous devons intercepter une exception appelée InterruptedException que la méthode sleep menace de déclencher. Nous verrons cette exception dans la prochaine section. Typiquement, un thread est interrompu pour y mettre fin. De même, notre méthode run se termine lorsqu'une InterruptedException se présente.

Dès que l'utilisateur clique sur le bouton Démarrer, la méthode addBall lance un nouveau thread (voir Figure 1.2) :

```
Ball b = new Ball();
panel.add(b);
Runnable r = new BallRunnable(b, panel);
Thread t = new Thread(r);
t.start
```

C'est tout ce qu'il faut savoir ! Vous savez maintenant lancer des tâches en parallèle. Le reste de ce chapitre vous montrera comment contrôler l'interaction entre les threads.

Figure 1.2
Exécuter plusieurs threads.



Le code complet de ce programme se trouve dans l'Exemple 1.2.

INFO

Vous pouvez aussi définir un thread en formant une sous-classe de la classe `Thread`, comme ceci :

```
class MyThread extends Thread
{
    public void run()
    {
        code de la tâche
    }
}
```

Vous construisez ensuite un objet de la sous-classe et appelez sa méthode `start`. Toutefois, cette approche n'est plus conseillée. Il vaut mieux découpler la *tâche* qui doit être exécutée en parallèle de son *mécanisme* d'exécution. Si vous disposez de plusieurs tâches, créer un thread séparé pour chacune serait trop onéreux. Vous pouvez plutôt utiliser un pool de threads (voir la section Pools de threads, plus loin dans ce chapitre).

ATTENTION

N'appellez pas la méthode `run` de la classe `Thread` ou de l'objet `Runnable` car cela exécute directement la tâche sur le *même* thread. Aucun nouveau thread n'est démarré. Appelez plutôt la méthode `Thread.start`. Elle créera un nouveau thread qui exécutera la méthode `run`.

Exemple 1.2 : *BounceThread.java*

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.util.*;
import javax.swing.*;
```

14 Au cœur de Java 2 - Fonctions avancées

```

/**
 * Présente une balle rebondissante animée.
 */
public class BounceThread
{
    public static void main(String[] args)
    {
        JFrame frame = new BounceFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Un exécutable qui anime une balle rebondissante.
 */
class BallRunnable implements Runnable
{
    /**
     * Construit l'exécutable.
     * @aBall la balle qui doit rebondir
     * @aPanel le composant dans lequel la balle rebondit
     */
    public BallRunnable(Ball aBall, Component aComponent)
    {
        ball = aBall;
        component = aComponent;
    }

    public void run()
    {
        try
        {
            for (int i = 1; i <= STEPS; i++)
            {
                ball.move(component.getBounds());
                component.repaint();
                Thread.sleep(DELAY);
            }
        }
        catch (InterruptedException e)
        {
        }
    }

    private Ball ball;
    private Component component;
    public static final int STEPS = 1000;
    public static final int DELAY = 5;
}

private ArrayList balls = new ArrayList();
}

/**
 * Une balle qui se déplace et rebondit sur les bords d'un
 * rectangle

```

```

*/
class Ball
{
    /**
     Déplace la balle à la position suivante, en inversant sa direction
     si elle touche l'un des bords
     */
    public void move(Rectangle2D bounds)
    {
        x += dx;
        y += dy;
        if (x < bounds.getMinX())
        {
            x = bounds.getMinX();
            dx = -dx;
        }
        if (x + XSIZE >= bounds.getMaxX())
        {
            x = bounds.getMaxX() - XSIZE;
            dx = -dx;
        }
        if (y < bounds.getMinY())
        {
            y = bounds.getMinY();
            dy = -dy;
        }
        if (y + YSIZE >= bounds.getMaxY())
        {
            y = bounds.getMaxY() - YSIZE;
            dy = -dy;
        }
    }

    /**
     Récupère la forme de la balle à sa position courante.
     */
    public Ellipse2D getShape()
    {
        return new Ellipse2D.Double(x, y, XSIZE, YSIZE);
    }

    private static final int XSIZE = 15;
    private static final int YSIZE = 15;
    private double x = 0;
    private double y = 0;
    private double dx = 1;
    private double dy = 1;
}

/**
 L'écran qui dessine les balles.
 */
class BallPanel extends JPanel
{
    /**
     Ajoute une balle à l'écran.

```

16 Au cœur de Java 2 - Fonctions avancées

```

        @param b la balle à ajouter
    */
    public void add(Ball b)
    {
        balls.add(b);
    }

    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        for (Ball b : balls)
        {
            g2.fill(b.getShape());
        }
    }

    private ArrayList<Ball> balls = new ArrayList<Ball>();
}

/**
 * Le cadre avec l'écran et les boutons.
 */
class BounceFrame extends JFrame
{
    /**
     * Construit le cadre avec l'écran pour afficher la
     * balle rebondissante et les boutons Démarrer et Fermer
     */
    public BounceFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
        setTitle("BounceThread");

        panel = new BallPanel();
        add(panel, BorderLayout.CENTER);
        JPanel buttonPanel = new JPanel();
        addButton(buttonPanel, "Démarrer",
            new ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    addBall();
                }
            });

        addButton(buttonPanel, "Fermer",
            new ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    System.exit(0);
                }
            });
        add(buttonPanel, BorderLayout.SOUTH);
    }
}

```

```

/**
 * Ajoute un bouton au conteneur.
 * @param c le conteneur
 * @param title le titre du bouton
 * @param listener l'écouteur d'action pour le bouton
 */
public void addButton(Container c, String title, ActionListener listener)
{
    JButton button = new JButton(title);
    c.add(button);
    button.addActionListener(listener);
}

/**
 * Ajoute une balle rebondissante sur le fond et lance un thread
 * pour la faire rebondir
 */
public void addBall()
{
    Ball b = new Ball();
    panel.add(b);
    Runnable r = new BallRunnable(b, panel);
    Thread t = new Thread(r);
    t.start();
}

private BallPanel panel;
public static final int DEFAULT_WIDTH = 450;
public static final int DEFAULT_HEIGHT = 350;
public static final int STEPS = 1000;
public static final int DELAY = 3;
}

```

java.lang.Thread 1.0

- Thread(Runnable target)
Construit un nouveau thread qui appelle la méthode run() de la cible spécifiée.
- void start()
Lance ce thread et appelle sa méthode run(). Cette méthode revient immédiatement. Le nouveau thread est exécuté en même temps.
- void run()
Appelle la méthode run de l'interface Runnable associée.

java.lang.Runnable 1.0

- void run()
Cette fonction doit être surchargée, et vous devez y ajouter les instructions qui doivent être exécutées dans le thread correspondant.

Interrompre des threads

Un thread s'arrête lorsque sa méthode run revient. Dans le JDK 1.0, il existait également une méthode stop qu'un autre thread aurait pu appeler pour terminer un thread. Mais cette méthode est aujourd'hui obsolète (voir plus loin dans ce chapitre).

18 Au cœur de Java 2 - Fonctions avancées

Il n'existe donc plus de méthode pour obliger un thread à se terminer. Toutefois, il est possible d'utiliser la méthode `interrupt` pour *exiger* la terminaison d'un thread.

Lorsque la méthode `interrupt` est appelée sur un thread, l'indication du thread est définie sur *interrompu* (*interrupted*). Il s'agit d'un indicateur booléen présent dans chaque thread. Chaque thread doit, à l'occasion, vérifier s'il a été interrompu.

Pour savoir si l'indication a été définie sur interrompu, appelez d'abord la méthode `Thread.currentThread` pour obtenir le thread actuel, puis appelez la méthode `isInterrupted` :

```
while (!Thread.currentThread().isInterrupted() && encore du travail)
{
    faire le travail
}
```

Cependant, si un thread est bloqué, il ne peut pas déterminer s'il est interrompu. C'est à ce moment que la méthode `InterruptedException` intervient. Lorsque la méthode `interrupt` est appelée sur un thread bloqué, l'appel bloquant (comme `sleep` ou `wait`) est terminé par une `InterruptedException`.

Il n'existe aucune spécification demandant qu'un thread interrompu doive être terminé. L'interruption d'un thread se contente d'attirer son attention. Le thread interrompu peut choisir comment réagir devant l'interruption. Certains threads sont tellement importants qu'ils peuvent tout simplement ignorer leur interruption en continuant leur travail. Mais, plus couramment, un thread essaiera d'interpréter une interruption comme une demande d'arrêt. La méthode `run` de ce type de thread a la forme suivante :

```
public void run()
{
    try
    {
        while (!Thread.currentThread().isInterrupted() && encore du travail)
        {
            faire le travail
        }
    }
    catch(InterruptedException e)
    {
        // le thread a été interrompu pendant sleep ou wait
    }
    finally
    {
        nettoyage, si nécessaire
    }
    // sort de la méthode run et met fin au thread
}
```

La vérification `isInterrupted` n'est pas nécessaire si vous appelez la méthode `sleep` après chaque itération de travail. Cette méthode déclenche une `InterruptedException` si vous l'appellez lorsque l'indication est définie sur interrompu. Ainsi, si votre boucle appelle `sleep`, ne vous inquiétez pas de vérifier l'interruption et interceptez simplement l'exception. Votre méthode `run` a ensuite la forme :

```
public void run()
{
    try
```

```

{
    while (effectuer du travail)
    {
        effectuer du travail
        Thread.sleep(delay);
    }
    catch(InterruptedException e)
    {
        // le thread a été interrompu pendant le sommeil ou l'attente
    }
    finally
    {
        nettoyage, si nécessaire
    }
    // quitter la méthode run met fin au thread
}

```

ATTENTION

Lorsque la méthode `sleep` déclenche une `InterruptedException`, elle *réinitialise* aussi l'indication *interrompu*.

INFO

Curieusement, il existe deux méthodes très similaires, `interrupted` et `isInterrupted`. La méthode `interrupted` est une méthode statique qui vérifie si le thread actuel a été interrompu. De plus, un appel à la méthode `interrupted` réinitialise l'indication "interrompu" d'un thread. D'un autre côté, la méthode `isInterrupted` est une méthode d'instance que vous pouvez utiliser pour vérifier que n'importe quel thread a été interrompu. L'indication "interrompu" de son argument n'est pas modifiée.

Vous trouverez de nombreux exemples de publication de codes où `InterruptedException` est silencieux, comme ceci :

```

void mySubTask()
{
    ...
    try { sleep(delay); }
    catch (InterruptedException e) {} NE L'IGNOREZ PAS !
    ...
}

```

Ne faites pas cela ! Si vous ne trouvez rien de bien à faire dans la clause `catch`, il vous reste deux choix raisonnables :

- Dans la clause `catch`, appelez `Thread.currentThread().interrupt()` pour définir l'indication sur *interrompu*. L'appelant peut alors le tester :

```

void mySubTask()
{
    ...
    try { sleep(delay); }
    catch (InterruptedException e) { Thread.currentThread().interrupt(); }
    ...
}

```

20 Au cœur de Java 2 - Fonctions avancées

- Mieux encore, vous pouvez baliser votre méthode avec `throws InterruptedException` et laisser le bloc `try`. L'appelant (ou, au final, la méthode `run`) peut l'intercepter :

```
void mySubTask() throws InterruptedException
{
    ...
    sleep(delay);
    ...
}
```

java.lang.Thread 1.0

- `void interrupt()`
Envoie une demande d'interruption à un thread. L'indication "interrompu" du thread est mise à `true`. Si le thread est actuellement bloqué par un appel à `sleep` ou à `wait`, une `InterruptedException` est déclenchée.
- `static boolean interrupted()`
Regarde si le thread *actuel* (c'est-à-dire le thread qui exécute cette instruction) a été interrompu. Notez qu'il s'agit d'une méthode statique. Cet appel possède un effet annexe : il met l'indication "interrompu" du thread courant à `false`.
- `boolean isInterrupted()`
Regarde si un thread a été interrompu. Contrairement à la méthode `static interrupted`, cet appel ne change pas l'indication "interrompu" du thread.
- `static Thread currentThread()`
Renvoie l'objet `Thread` représentant le thread en cours d'exécution.

Les états d'un thread

Les threads peuvent se trouver dans l'un des quatre états suivants :

- nouveau ;
- exécutable ;
- bloqué ;
- mort.

Chacun de ces états est expliqué dans les paragraphes suivants.

Nouveaux threads

Lorsque vous créez un thread avec l'opérateur `new` (par exemple `new Thread(r)`), le thread n'est pas encore exécuté. Cela signifie qu'il se trouve dans l'état *nouveau*. Lorsqu'un thread se trouve dans cet état, le programme n'a pas encore commencé à exécuter le code qui se trouve dans ce thread. Il faut encore faire quelques opérations avant que ce thread puisse être exécuté.

Threads exécutables

Une fois que vous avez invoqué la méthode `start`, le thread devient *exécutable*. Il se peut que le thread exécutable soit ou non en cours d'exécution. C'est au système d'exploitation de fournir au thread une fenêtre d'exécution. La spécification Java ne définit pas cet état comme un état séparé. Un thread en cours d'exécution reste dans l'état exécutable.

Lorsqu'un thread s'exécute, il ne poursuit pas obligatoirement l'opération dans le temps. En réalité, cela peut se révéler souhaitable si des threads sont mis en pause de temps à autre afin que d'autres threads aient une occasion de s'exécuter. Ce mécanisme dépend directement du système d'exploitation. Les systèmes de planification préemptifs accordent à chaque thread exécutable une tranche de temps pour effectuer sa tâche. Lorsque cette tranche de temps est écoulée, le système d'exploitation *préempte* le thread et passe à un autre (voir Figure 1.4). Lorsque vous sélectionnez le thread suivant, le système d'exploitation prend en compte les *priorités* des threads (voir plus loin pour en savoir plus sur les priorités).

Tous les systèmes d'exploitation modernes utilisent la planification préemptive. Toutefois, les petits appareils comme les téléphones portables peuvent utiliser la planification coopérative. Dans ces appareils, un thread ne perd le contrôle que lorsqu'il appelle une méthode comme `sleep` ou `yield`.

Sur une machine multiprocesseur, chaque processeur peut exécuter un thread, et plusieurs threads peuvent s'exécuter en parallèle. Bien entendu, s'il y a plus de threads que de processeurs, le gestionnaire est toujours responsable du découpage en tranches.

N'oubliez pas qu'un thread exécutable peut ou non s'exécuter à tout moment.

Threads bloqués

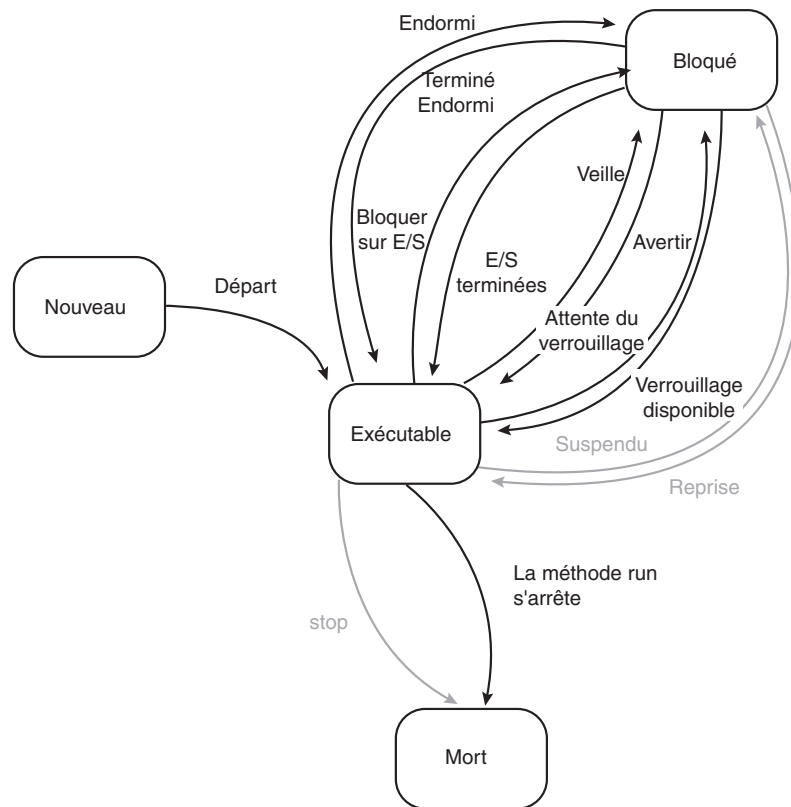
Un thread entre dans l'état *bloqué* lorsque l'une des actions suivantes se produit :

1. La méthode `sleep()` du thread est appelée, le thread se met en sommeil.
2. Le thread appelle une opération *bloquante sur les entrées/sorties*, c'est-à-dire une opération qui ne rendra pas la main à l'appelant tant que les opérations d'entrées ou de sorties ne sont pas terminées.
3. Le thread essaie d'obtenir un verrou actuellement détenu par un autre thread. Nous aborderons les verrous un peu plus loin dans ce chapitre.
4. Le thread attend une condition.
5. La méthode `suspend` du thread est appelée. Cependant, celle-ci commence à être désapprouvée, et il vaut mieux éviter de l'appeler dans votre code.

La Figure 1.3 montre les différents états d'un thread et les transitions possibles d'un état à un autre. Lorsqu'un thread est bloqué (ou, bien sûr, lorsqu'il est mort), un autre thread peut être prévu pour être exécuté. Lorsqu'un thread bloqué est réactivé (par exemple, parce qu'il s'est endormi pendant un délai donné, ou parce que l'opération d'entrée/sortie qu'il attendait est terminée), le gestionnaire regarde s'il possède une priorité supérieure à celle du thread actuellement en cours d'exécution. Dans ce cas, il interrompt le thread actuel et sélectionne un nouveau thread.

22 Au cœur de Java 2 - Fonctions avancées

Figure 1.3
Les états d'un thread.



Pour sortir de l'état bloqué et entrer à nouveau dans l'état exécutable, un thread doit emprunter l'un des chemins suivants :

1. Si un thread a été mis en sommeil, le nombre spécifié de millisecondes doit être écoulé.
2. Si un thread attend la fin d'une opération d'entrée ou de sortie, cette opération doit être terminée.
3. Si un thread attend un verrou qui est possédé par un autre thread, l'autre thread doit avoir rendu le verrou. Il est aussi possible d'attendre une temporisation. Le thread se débloquent alors à la fin de la temporisation.
4. Si un thread attend une condition, un autre thread doit signaler que la condition risque d'avoir changé (si le thread attend une temporisation, il est débloquent à la fin de la temporisation).
5. Si un thread a été suspendu, sa méthode `resume` doit être appelée. Cependant, comme la méthode `suspend` n'est plus utilisée, la méthode `resume` est également moins utilisée, il faut donc éviter de l'appeler dans votre code.

Un thread bloqué peut uniquement être activé par la même technique que celle qui l'a bloqué. En particulier, vous ne pouvez pas appeler sa méthode `resume` pour débloquent un thread bloquant.

ASTUCE

Si vous devez débloquent une opération d'E/S, utilisez le mécanisme de *canal* de la bibliothèque "new I/O". Lorsqu'un autre thread ferme le canal, le thread bloqué devient à nouveau exécutable, et l'opération bloquante lance une `ClosedChannelException`.

Threads morts

Un thread peut mourir pour l'une des deux raisons suivantes :

- Il meurt d'une mort naturelle parce que la méthode `run` s'est terminée normalement.
- Il meurt soudainement parce qu'une exception non récupérée a mis fin à la méthode `run`.

En particulier, il est possible de tuer un thread en invoquant sa méthode `stop`. Cette méthode déclenche une erreur `ThreadDeath` qui tue le thread. Cependant, la méthode `stop` n'est plus utilisée, et il faut éviter de l'appeler dans votre code.

Pour savoir si un thread est couramment actif, c'est-à-dire qu'il est soit exécutable, soit bloqué, utilisez la méthode `isAlive`. Celle-ci renvoie `true` si le thread est exécutable ou bloqué et `false` si le thread est nouveau et pas encore exécutable ou si le thread est mort.

INFO

Vous ne pouvez pas déterminer si un thread actif est exécutable ou bloqué ou si un thread exécutable est réellement en cours d'exécution. Il est également impossible de faire la différence entre un thread qui n'est pas encore exécutable et un thread qui est déjà mort.

java.lang.Thread 1.0

- `boolean isAlive()`
Renvoie `true` si le thread a démarré et n'est pas encore terminé.
- `void stop()`
Arrête le thread. Cette méthode est aujourd'hui obsolète.
- `void suspend()`
Suspend l'exécution du thread. Cette méthode est aujourd'hui obsolète.
- `void resume()`
Reprend ce thread. Cette méthode n'est valide qu'après un appel à `suspend()`. Cette méthode est aujourd'hui obsolète.
- `void join()`
Attend la mort du thread spécifié.
- `void join(long millis)`
Attend la mort du thread spécifié ou l'écoulement du nombre spécifié de millisecondes.

Propriétés d'un thread

Dans les sections suivantes, nous verrons les diverses propriétés des threads : priorités des threads, threads démons, groupes de threads et gestionnaires d'exceptions non récupérées.

Priorités d'un thread

Dans le langage de programmation Java, chaque thread possède une *priorité*. Par défaut, un thread hérite de la priorité de son thread parent. Vous pouvez augmenter ou baisser la priorité de n'importe quel thread avec la méthode `setPriority`. La priorité de n'importe quel thread peut être choisie entre `MIN_PRIORITY` (correspondant à 1 dans la classe `Thread`) et `MAX_PRIORITY` (correspondant à 10). `NORM_PRIORITY` vaut 5.

Lorsque le gestionnaire de threads peut choisir un nouveau thread, il choisit *généralement* le thread de la plus haute priorité. Toutefois, les priorités de threads sont *fortement dépendantes du système*. Lorsque la machine virtuelle compte sur l'implémentation des threads de la plate-forme hôte, les priorités de threads Java sont mises en correspondance avec les niveaux de priorité de la plate-forme hôte, qui peut en avoir plus ou moins.

A titre d'exemple, Windows NT/XP affiche sept niveaux de priorité. Certaines priorités Java concorderont avec le même niveau du système d'exploitation. Dans la machine virtuelle de Sun pour Linux, les priorités de threads sont totalement ignorées : tous les threads ont la même priorité.

Par conséquent, il vaut mieux considérer les priorités des threads comme des indications destinées au gestionnaire. Ne structurez jamais vos programmes en fonction des niveaux de priorité.

ATTENTION

Si vous souhaitez tout de même utiliser les priorités, soyez averti d'une erreur commune aux débutants. Si vous disposez de plusieurs threads à haute priorité qui bloquent rarement, ceux de moindre priorité risquent de ne *jamais* s'exécuter. Lorsque le gestionnaire décide d'exécuter un nouveau thread, il choisira d'abord parmi ceux de plus haute priorité, même si cela peut totalement annihiler les threads de moindre priorité.

java.lang.Thread 1.0

- `void setPriority(int newPriority)`
Définit la priorité de ce thread. Cette priorité doit être comprise entre `Thread.MIN_PRIORITY` et `Thread.MAX_PRIORITY`. Utilisez `Thread.NORM_PRIORITY` pour une priorité normale.
- `static int MIN_PRIORITY`
Définit la priorité minimale qu'un thread peut avoir. La valeur de la priorité minimale est 1.
- `static int NORM_PRIORITY`
Définit la priorité par défaut d'un thread. La valeur de la priorité par défaut est 5.
- `static int MAX_PRIORITY`
Définit la priorité maximale qu'un thread peut avoir. La valeur de la priorité maximale est 10.
- `static void yield()`
Cette méthode arrête le thread en cours d'exécution. S'il existe d'autres threads exécutables dont la priorité est au moins égale à celle de ce thread, ils seront traités par la suite. Notez qu'il s'agit d'une méthode statique.

Threads démons

Un thread peut être transformé en *thread démon* en appelant

```
t.setDaemon(true);
```

Il n'y a rien de démoniaque là-dedans ! Un démon est simplement un thread qui n'a aucun autre but dans la vie que de servir d'autres threads. On peut citer comme exemples des threads chronomètres qui envoient des "pulsations de temps" régulières à d'autres threads. Lorsqu'il ne reste plus que des threads démons, la machine virtuelle se ferme. Il n'y a en effet aucun intérêt à continuer d'exécuter un programme si tous les threads restants sont des démons.

java.lang.Thread 1.0

- void setDaemon(boolean isDaemon)

Signale ce thread comme un démon ou un thread utilisateur. Cette méthode doit être appelée avant le démarrage du thread.

Groupes de threads

Certains programmes contiennent un nombre important de threads. Il devient alors utile de les regrouper par fonctionnalités. Par exemple, considérons un navigateur Internet. Si plusieurs threads essaient d'obtenir des images à partir d'un serveur, et que l'utilisateur clique sur un bouton "Arrêter" pour interrompre le chargement de la page courante, il est alors pratique de disposer d'un moyen d'interrompre tous ces threads simultanément. Le langage de programmation Java vous permet de construire ce qu'il appelle un *groupe de threads* pour que vous puissiez travailler simultanément avec plusieurs threads.

Vous pouvez construire un groupe de threads avec le constructeur suivant :

```
String groupName = . . . ;  
ThreadGroup g = new ThreadGroup(groupName)
```

La chaîne passée en argument au constructeur ThreadGroup sert à identifier le groupe et doit être unique. Vous pouvez alors ajouter des threads au groupe de threads en spécifiant ce dernier dans le constructeur du thread.

```
Thread t = new Thread(g, threadName);
```

Pour déterminer si certains threads d'un groupe particulier sont toujours exécutables, utilisez la méthode activeCount.

```
if (g.activeCount() == 0)  
{  
    // tous les threads du groupe g ont été arrêtés  
}
```

Pour interrompre tous les threads d'un groupe de threads, appelez simplement interrupt sur l'objet du groupe.

```
g.interrupt(); // interrompt tous les threads du groupe g
```

Toutefois, les exécuteurs permettent d'effectuer la même tâche sans utiliser les groupes de threads.

26 Au cœur de Java 2 - Fonctions avancées

Les groupes de threads peuvent posséder des sous-groupes enfant. Par défaut, un groupe de threads nouvellement créé devient un enfant du groupe de threads courant. Mais vous pouvez aussi explicitement fournir le nom du groupe dans le constructeur (voir les notes d'API). Des méthodes comme `activeCount` et `interrupt` font référence à tous les threads de leur groupe et à tous les groupes enfant.

java.lang.Thread 1.0

- `Thread(ThreadGroup g, String name)`

Crée un nouveau thread qui appartient à un `ThreadGroup` donné.

Paramètres :

<code>g</code>	Le groupe de threads auquel appartient le nouveau thread
<code>name</code>	Le nom du nouveau thread

- `ThreadGroup getThreadGroup()`
Renvoie le groupe de threads de ce thread.

java.lang.ThreadGroup 1.0

- `ThreadGroup(String name)`

Cette méthode crée un nouveau `ThreadGroup`. Son parent sera le groupe de threads du thread courant.

Paramètres :

<code>name</code>	le nom du nouveau groupe de threads

- `ThreadGroup(ThreadGroup parent, String name)`

Cette méthode crée un nouveau `ThreadGroup`.

Paramètres :

<code>parent</code>	le groupe de threads parent du nouveau groupe de threads
<code>name</code>	le nom du nouveau groupe de threads

- `int activeCount()`

Cette méthode renvoie la borne supérieure correspondant au nombre de threads actifs dans le groupe de threads.

- `int enumerate(Thread[] list)`

Renvoie les références de tous les threads actifs dans ce groupe de threads. Vous pouvez utiliser la méthode `activeCount()` pour renvoyer la borne supérieure du tableau ; cette méthode renvoie le nombre de threads placés dans le tableau. Si le tableau est trop petit (par exemple parce que d'autres threads ont été créés après l'appel à `activeCount`), autant de threads que possible sont insérés.

Paramètres :

<code>list</code>	un tableau à remplir avec les références des threads

- `ThreadGroup getParent()`

Renvoie le parent de ce groupe de threads.

- `void interrupt()`

Interrompt tous les threads de ce groupe de threads et tous ses groupes enfant.

Gestionnaire d'exceptions non récupérées

La méthode `run` d'un thread ne peut pas déclencher d'exceptions sous contrôle, mais elle peut être terminée par une exception hors contrôle. Dans ce cas, le thread meurt.

Il n'existe toutefois pas de clause `catch` dans laquelle l'exception peut être propagée. Juste avant que le thread ne meure, l'exception est transférée à un gestionnaire d'exceptions non récupérées.

Ce gestionnaire doit appartenir à une classe qui implémente l'interface `Thread.UncaughtExceptionHandler`, qui ne possède qu'une seule méthode :

```
void uncaughtException(Thread t, Throwable e)
```

Depuis le JDK 5.0, vous pouvez installer un gestionnaire dans n'importe quel thread avec la méthode `setUncaughtExceptionHandler`. Vous pouvez aussi installer un gestionnaire par défaut pour tous les threads avec la méthode statique `setDefaultUncaughtExceptionHandler` de la classe `Thread`. Un gestionnaire de remplacement pourrait utiliser l'API d'identification pour envoyer des rapports sur les exceptions non récupérées à un fichier journal.

Si vous n'installez pas de gestionnaire par défaut, le gestionnaire par défaut vaut `null`. Mais, si vous n'installez pas de gestionnaire pour un thread particulier, le gestionnaire correspond à l'objet `ThreadGroup` du thread.

La classe `ThreadGroup` implémente l'interface `Thread.UncaughtExceptionHandler`. Sa méthode `uncaughtException` réalise l'action suivante :

1. Si le groupe de threads a un parent, la méthode `uncaughtException` du groupe parent est appelée.
2. Sinon, si la méthode `Thread.getDefaultExceptionHandler` renvoie un gestionnaire non nul, il est appelé.
3. Sinon, si `Throwable` est une instance de `ThreadDeath`, il ne se passe rien.
4. Sinon, le nom du thread et la trace de `Throwable` sont affichés sur `System.err`.

C'est une trace que vous avez certainement beaucoup vue dans vos programmes.

INFO

Avant le JDK 5.0, il était impossible d'installer un gestionnaire d'exceptions non récupérées dans chaque thread et de spécifier un gestionnaire par défaut. Pour installer un gestionnaire, il fallait sous-classer la classe `ThreadGroup` et surcharger la méthode `uncaughtException`.

java.lang.Thread 1.0

- `static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)` 5.0
- `static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()` 5.0

Définissent ou récupèrent le gestionnaire par défaut pour les exceptions non récupérées.

- `void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)` 5.0
- `Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()` 5.0

Définissent ou récupèrent le gestionnaire des exceptions non récupérées. Lorsque aucun gestionnaire n'est installé, l'objet du groupe de threads devient le gestionnaire.

28 Au cœur de Java 2 - Fonctions avancées

java.lang.Thread.UncaughtExceptionHandler 5.0

- void uncaughtException(Thread t, Throwable e)

Définit cette méthode de manière à consigner un rapport personnalisé lorsqu'un thread se termine avec une exception non récupérée.

Paramètres :

t	le thread terminé du fait d'une exception non récupérée
e	l'objet de l'exception non récupérée

java.lang.ThreadGroup 1.0

- void uncaughtException(Thread t, Throwable e)

Appelle cette méthode du groupe de threads parent (s'il existe un parent) ou le gestionnaire par défaut de la classe Thread (s'il existe un gestionnaire par défaut). Sinon affiche une trace du flux d'erreur standard (toutefois, si e est un objet ThreadDeath, la trace est supprimée. Les objets ThreadDeath sont générés par la méthode stop, aujourd'hui obsolète).

Synchronisation

Dans la plupart des applications pratiques à base de multithreads, il arrive que plusieurs threads doivent partager un accès aux mêmes objets. Que se passe-t-il si deux threads ont accès au même objet et que chacun appelle une méthode qui modifie l'état de cet objet ? Comme vous pouvez l'imaginer, les threads se font concurrence. Selon l'ordre dans lequel la donnée a été accédée, des objets corrompus peuvent apparaître. Ce type de situation est souvent appelé une *condition de course*.

Exemple de condition de course

Pour éviter que plusieurs threads ne corrompent des données partagées, vous devez apprendre comment synchroniser l'accès. Dans cette section, vous verrez ce qui se passe si vous n'utilisez pas de synchronisation. Dans la section suivante, vous verrez comment synchroniser les accès aux objets.

Dans le prochain programme de test, nous simulons une banque possédant plusieurs comptes. Nous générons au hasard des transactions qui déplacent de l'argent entre ces comptes. Chaque compte possède un thread. Chaque transaction déplace des quantités aléatoires d'argent, du compte desservi par le thread vers un autre compte choisi aléatoirement.

Le code de cette simulation est assez simple. Il est principalement constitué de la classe Bank et de la méthode transfer. Cette méthode transfère une certaine somme d'argent d'un compte vers un autre. Si le compte source ne possède pas assez d'argent, l'appel prend fin aussitôt. Voici le code de la méthode transfer de la classe Bank :

```
public void transfer(int from, int to, double amount)
// ATTENTION : cette méthode n'est pas sûre lorsqu'elle est appelée à
// partir de plusieurs threads
{
    System.out.print(Thread.currentThread());
    accounts[from] -= amount;
```

```

        System.out.print(" %10.2f de %d à %d", amount, from, to);
        accounts[to] += amount;
        System.out.printf(" Solde total : %10.2f%n", getTotalBalance());
    }

```

Voici le code de la classe `TransferRunnable`. Sa méthode `run` sort en permanence de l'argent d'un compte bancaire spécifié. A chaque itération, la méthode `run` choisit au hasard un compte cible et un montant d'argent aléatoire, puis elle appelle `transfer` sur l'objet banque, puis elle s'endort.

```

class TransferRunnable implements Runnable
{
    ...
    public void run()
    {
        int toAccount = (int) (bank.size() * Math.random());
        double amount = maxAmount * Math.random();
        bank.transfer(fromAccount, toAccount, amount);
        Thread.sleep(int) (DELAY * Math.random());
    }
    catch(InterruptedException e) {}
}

```

Lorsque cette simulation est exécutée, nous ne savons pas combien d'argent se trouve dans chaque compte bancaire. En revanche, nous savons que la somme de tous les comptes doit rester constante, puisque nous effectuerons uniquement des transferts d'argent entre ces comptes.

A la fin de chaque transaction, la méthode `transfer` recalcule le total et l'affiche.

Ce programme ne s'arrête jamais. Il suffit d'appuyer sur `Ctrl+C` pour l'arrêter.

Voici un résultat typique :

```

. . .
Thread[Thread-11,5,main]    588.48 de 11 à 44 Solde total : 100000.00
Thread[Thread-12,5,main]    976.11 de 12 à 22 Solde total : 100000.00
Thread[Thread-14,5,main]    521.51 de 14 à 22 Solde total : 100000.00
Thread[Thread-13,5,main]    359.89 de 13 à 81 Solde total : 100000.00
. . .
Thread[Thread-36,5,main]    401.71 de 36 à 73 Solde total : 99291.06
Thread[Thread-35,5,main]    691.46 de 35 à 77 Solde total : 99291.06
Thread[Thread-37,5,main]    78.64 de 37 à 3 Solde total : 99291.06
Thread[Thread-34,5,main]    197.11 de 34 à 69 Solde total : 99291.06
Thread[Thread-36,5,main]    85.96 de 36 à 4 Solde total : 99291.06
. . .
Thread[Thread-4,5,main]Thread[Thread-33,5,main]    7.31 de 31 à 32 Solde
total : 99979.24
627.50 de 4 à 5 Solde total : 99979.24
. . .

```

Comme vous pouvez le constater, il y a une erreur importante. Pour certaines transactions, le solde reste à 100 000 €, ce qui correspond au total correct pour 100 comptes de 10 000 € chacun. Mais après un certain moment, le solde total est légèrement modifié. Lorsque vous exécuterez ce programme, vous vous rendrez compte que des erreurs se produisent rapidement, ou au contraire qu'il faut un certain temps pour que le solde soit corrompu. Cette situation n'inspire pas confiance, et vous n'aurez probablement pas envie de déposer votre argent difficilement gagné dans cette banque.

30 Au cœur de Java 2 - Fonctions avancées

L'Exemple 1.3 fournit le code source complet de cette simulation. Regardez si vous pouvez repérer le problème posé par ce code. Nous vous dévoilerons ce mystère dans la prochaine section.

Exemple 1.3 : *UnsynchBankTest.java*

```
/**
 * Ce programme montre la corruption de données lorsque
 * plusieurs threads accèdent à une structure de données.
 */
public class UnsynchBankTest
{
    public static void main(String[] args)
    {
        Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
        int i;
        for (i = 0; i < NACCOUNTS; i++)
        {
            TransferRunnable r = new TransferRunnable(b, i,
                INITIAL_BALANCE);
            Thread t = new Thread(r);
            t.start();
        }

        public static final int NACCOUNTS = 100;
        public static final double INITIAL_BALANCE = 1000;
    }

    /**
     * Une banque avec plusieurs comptes bancaires
     */
    class Bank
    {
        /**
         * Construit la banque
         * @param n le nombre de comptes
         * @param initialBalance le solde initial
         * pour chaque compte
         */
        public Bank(int n, int initialBalance)
        {
            accounts = new double[n];
            for (int i = 0; i < accounts.length; i++)
                accounts[i] = initialBalance;
        }

        /**
         * Transfère l'argent d'un compte à l'autre
         * @param from le compte d'origine du transfert
         * @param to le compte vers lequel effectuer le transfert
         * @param amount la somme à transférer
         */
        public void transfer(int from, int to, double amount)
        {
            if (accounts[from] < amount) return;
            System.out.print(Thread.currentThread());
            accounts[from] -= amount;
            System.out.printf(" %10.2f de %d à %d", amount, from, to);
            accounts[to] += amount;
        }
    }
}
```

```
        System.out.printf(" Solde total : %10.2f%n", getTotalBalance());
    }

    /**
     * Récupère la somme de tous les soldes.
     * @return le solde
     */
    public double getTotalBalance()
    {
        double sum = 0;

        for (double a : accounts)
            sum += a;

        return sum;
    }

    /**
     * Obtient le nombre de comptes dans la banque
     * @return le nombre de comptes
     */
    public int size()
    {
        return accounts.length;
    }

    private final double[] accounts;
}

/**
 * Un exécutable qui transfère l'argent d'un compte à l'autre
 * dans une banque
 */
class TransferRunnable implements Runnable
{
    /**
     * Construit un exécutable de transfert
     * @param b la banque dont les comptes sont concernés par le transfert
     * @param from le compte d'origine du transfert
     * @param max la somme maximum d'argent pour chaque transfert
     */
    public TransferRunnable(Bank b, int from, double max)
    {
        bank = b;
        fromAccount = from;
        maxAmount = max;
    }

    public void run()
    {
        try
        {
            while (true)
            {
                int toAccount = (int) (bank.size() * Math.random());
                double amount = maxAmount * Math.random();
                bank.transfer(fromAccount, toAccount, amount);
                Thread.sleep((int) (DELAY * Math.random()));
            }
        }
    }
}
```

32 Au cœur de Java 2 - Fonctions avancées

```

    }
    catch(InterruptedException e) {}
}

private Bank bank;
private int fromAccount;
private double maxAmount;
private int DELAY = 10;
}

```

Explication des conditions de course

Dans la section précédente, nous avons exécuté un programme dans lequel plusieurs threads mettaient à jour des soldes de comptes bancaires. Après un certain temps, des erreurs apparaissaient, et une certaine quantité d'argent était soit perdue, soit ajoutée spontanément. Ce problème se présente lorsque deux threads essaient simultanément de mettre à jour un compte. Supposons que deux threads essaient d'exécuter l'instruction suivante simultanément :

```
accounts[to] += amount;
```

Le problème de cette instruction est qu'elle n'est pas composée d'opérations *atomiques*. Cette instruction peut en effet être décomposée comme suit :

1. Charger `accounts[to]` dans un registre.
2. Ajouter `amount`.
3. Placer le résultat dans `accounts[to]`.

Maintenant, supposons que le premier thread exécute les deux premières étapes, puis qu'il est interrompu. Supposons ensuite que le second thread se réveille et qu'il mette à jour la même entrée dans le tableau `account`. Le premier thread se réveille alors et termine sa troisième étape.

Cette action annule la modification de l'autre thread. Par conséquent, le total n'est plus correct (voir Figure 1.4).

Notre programme de test détecte cette erreur. Naturellement, il existe une légère possibilité que de fausses alarmes se présentent si le thread est interrompu alors qu'il effectue le test !

INFO

Vous pouvez en fait lire les codes d'octets binaires de la machine virtuelle qui correspondent à chaque instruction de notre classe. Exécutez la commande

```
javap -c -v Bank
```

pour décompiler le fichier `Bank.class`. Par exemple, la ligne

```
accounts[to] += amount;
```

est traduite par les codes binaires suivants :

```

aload_0
getfield #2 //comptes du champ: [D
iload_2
dup2

```

```

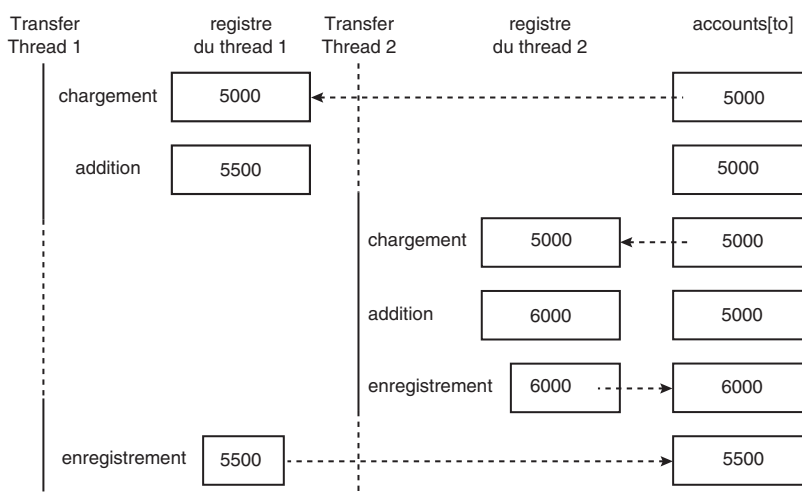
dload
dload_3
dadd
dastore

```

La signification réelle de ces codes n'a pas d'importance. Vous voyez maintenant que l'instruction d'addition est composée de plusieurs petites instructions, et que le thread qui les exécute peut être interrompu au milieu de ces instructions.

Figure 1.4

Deux accès simultanés.



Quelle est la probabilité que cette erreur se produise ? Nous avons déterminé que nous pouvons augmenter la probabilité d'erreur en entrelaçant les instructions d'affichage à celles qui actualisent le solde.

Si vous oubliez les instructions d'affichage, le risque de corruption diminue un peu car chaque thread travaille très peu avant de se rendormir. Il est donc peu probable que le gestionnaire le préempte au milieu du calcul. Mais le risque de corruption demeure. Si vous exécutez de très nombreux threads sur une machine très chargée, le programme échouera, même lorsque vous aurez éliminé les instructions d'affichage. L'échec peut survenir après quelques minutes, quelques heures ou quelques jours. Il n'y a rien de pire, ou presque, pour un programmeur qu'une erreur qui ne se manifeste que tous les deux ou trois jours.

Le vrai problème réside dans le fait que le travail de la méthode `transfer` peut être interrompu en plein milieu. Si nous pouvions nous assurer que la méthode s'exécute jusqu'à la fin, avant la perte de contrôle du thread, l'état de l'objet compte en banque ne serait jamais corrompu.

Verrous d'objet

Depuis le JDK 5.0, il existe deux mécanismes permettant de protéger un bloc de code d'un accès simultané. Les précédentes versions de Java utilisaient le mot clé `synchronized` dans ce but. Le JDK 5.0 a introduit la classe `ReentrantLock`. Le mot clé `synchronized` fournit automatiquement un verrou ainsi qu'une "condition" associée. Souvent, on comprend mieux le mot clé `synchronized`

34 Au cœur de Java 2 - Fonctions avancées

après avoir isolé les verrous et les conditions. Le JDK 5.0 propose des classes séparées pour ces mécanismes fondamentaux, que nous expliquerons ici et plus loin dans ce chapitre.

Le code principal permettant de protéger un bloc de code avec `ReentrantLock` est le suivant :

```
myLock.lock(); // un objet ReentrantLock
try
{
    Section principale
}
finally
{
    myLock.unlock(); // vérifier que le verrou est déverrouillé, même si
                    // une exception est déclenchée
}
```

Cette construction garantit que seul un thread à la fois puisse entrer dans la section principale. Dès qu'un thread verrouille l'objet verrou, aucun autre thread ne peut entrer dans l'instruction `lock`. Lorsque d'autres threads appellent `lock`, ils sont bloqués jusqu'à ce que le premier thread déverrouille l'objet verrou.

Utilisons un verrou pour protéger la méthode `transfer` de la classe `Bank` :

```
public class Bank
{
    public void transfer(int from, int to, int amount)
    {
        bankLock.lock();
        try
        {
            if (accounts[from] < amount) return;
            System.out.print(Thread.currentThread());
            accounts[from] -= amount;
            System.out.printf(" %10.2f de %d à %d", amount, from, to);
            accounts[to] += amount;
            System.out.printf(" Solde total : %10.2f%n", getTotalBalance());
        }
        finally
        {
            bankLock.unlock();
        }
    }
    . . .
    private Lock bankLock = new ReentrantLock(); // ReentrantLock
                                                // implémente l'interface Lock
}
```

Supposons qu'un thread appelle `transfer` et soit préempté avant d'avoir terminé. Supposons ensuite qu'un deuxième thread appelle aussi `transfer`. Le deuxième ne peut pas acquérir le verrou, il est bloqué dans l'appel à la méthode `lock`. Il est donc désactivé et doit attendre que le premier thread finisse d'exécuter la méthode `transfer`. Lorsque le premier thread débloque le verrou, le deuxième peut continuer (voir Figure 1.5).

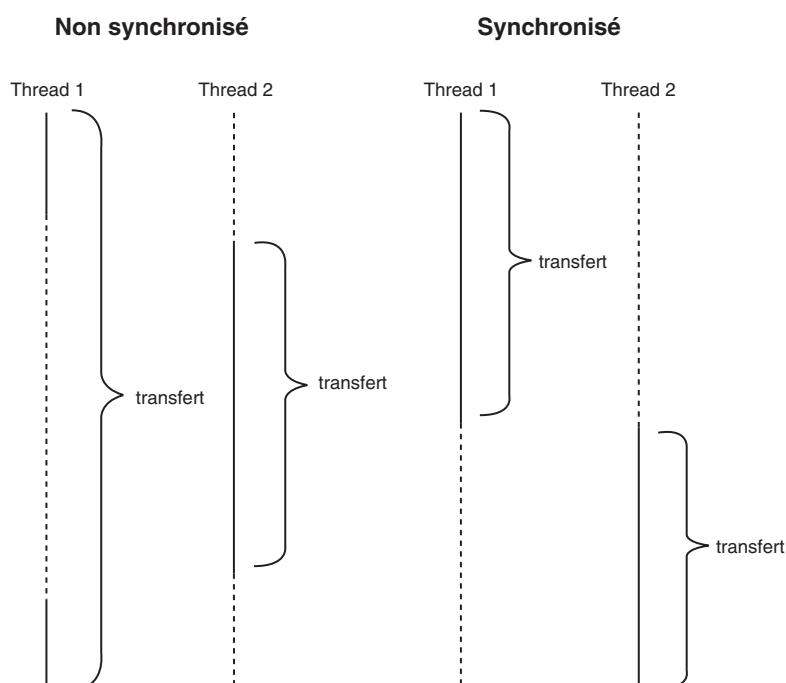
Testez-le. Ajoutez le code de verrouillage à la méthode `transfer` et exécutez à nouveau le programme. Vous pouvez l'exécuter à l'infini, le solde ne sera jamais corrompu.

Vous remarquerez que chaque objet Bank possède son propre objet `ReentrantLock`. Si deux threads tentent d'accéder à différents objets Bank, chaque thread acquiert un verrou différent et aucun thread n'est bloqué. C'est ce qui est prévu, car les threads ne doivent pas interférer les uns avec les autres lorsqu'ils manipulent différentes instances de Bank.

Le verrou est dit *rentrant* car un thread peut acquérir, à plusieurs reprises, un verrou qu'il possède déjà. Le verrou tient le *compte* des appels imbriqués à la méthode `lock`. Le thread doit appeler `unlock` pour chaque appel à `lock`, de manière à renoncer au verrou. Du fait de cette caractéristique, le code protégé par un verrou peut appeler une autre méthode qui utilise les mêmes verrous.

Figure 1.5

Comparaison entre threads synchronisés et non synchronisés.



Si, par exemple, la méthode `transfer` appelle la méthode `getTotalBalance`, celui-ci verrouille aussi l'objet `bankLock`, lequel affiche maintenant un compte de 2. Lorsque la méthode `getTotalBalance` se termine, le compte revient à 1. A la fin de la méthode `transfer`, le compte est à 0 et le thread renonce au verrou.

En général, il est souhaitable de protéger les blocs de code qui exigent plusieurs opérations pour mettre à jour ou inspecter une structure de données. Vous êtes alors assuré que ces opérations s'exécutent jusqu'au bout avant qu'un autre thread ne puisse utiliser le même objet.

ATTENTION

Vous devez prendre garde à ce que le code d'une section principale ne soit pas contourné par le déclenchement d'une exception. Si une exception est déclenchée avant la fin de la section, la clause `finally` renoncera au verrou, mais l'objet pourra être endommagé.

36 Au cœur de Java 2 - Fonctions avancées

java.util.concurrent.locks.Lock 5.0

- `void lock()`
Acquiert ce verrou ; se bloque si le verrou appartient à un autre thread.
- `void unlock()`
Libère ce bloc.

java.util.concurrent.locks.ReentrantLock 5.0

- `ReentrantLock()`
Construit un verrou rentrant pouvant être utilisé pour protéger une section principale.

Objets de condition

Très souvent, un thread entre dans une section principale, uniquement pour découvrir qu'il ne peut pas poursuivre tant qu'une condition n'est pas remplie. Vous utiliserez alors un *objet de condition* pour gérer les threads qui ont acquis un verrou mais ne peuvent pas procéder à un travail utile. Dans cette section, nous présentons l'implémentation d'objets de condition dans la bibliothèque Java (pour des raisons historiques, les objets de condition sont souvent appelés des *variables de condition*).

Précisons notre simulation de la banque. Il doit être impossible de faire sortir de l'argent d'un compte qui ne dispose pas de fonds suffisants. Sachez que nous ne pouvons pas utiliser un code du type

```
if (bank.getBalance(from) >= amount)
    bank.transfer(from, to, amount);
```

Il est tout à fait possible que le thread actuel soit désactivé entre le résultat positif du test et l'appel à `transfer`.

```
if (bank.getBalance(from) >= amount)
    // le thread risque d'être désactivé ici
    bank.transfer(from, to, amount);
```

D'ici à ce que le thread s'exécute à nouveau, le solde du compte risque d'être descendu sous le montant du retrait. Vous devez vous assurer que le thread ne soit pas interrompu entre le test et l'insertion. Pour ce faire, protégez le test et l'action de transfert avec un verrou :

```
public void transfer(int from, int to, int amount)
{
    bankLock.lock();
    try
    {
        while (accounts[from] < amount)
        {
            // attendre
            . . .
        }
        // transférer les fonds
        . . .
    }
    finally
```

```
    {  
        bankLock.unlock();  
    }  
}
```

Que faire maintenant lorsqu'il n'y a pas suffisamment d'argent sur le compte ? Nous attendons qu'un autre thread ajoute des fonds. Mais ce thread vient d'obtenir un accès exclusif à `bankLock`, aucun autre thread n'a donc l'occasion de réaliser un dépôt. C'est ici que les objets de condition entrent en jeu.

Un objet verrou peut se voir associer un ou plusieurs objets de condition. Pour obtenir un objet de condition, utilisez la méthode `newCondition`. Chaque objet de condition reçoit généralement un nom qui évoque la condition qu'il représente. Ici, par exemple, nous établissons un objet de condition pour représenter la condition "fonds suffisants".

```
class Bank  
{  
    public Bank()  
    {  
        .  
        .  
        .  
        sufficientFunds = bankLock.newCondition();  
    }  
    .  
    private Condition sufficientFunds;  
}
```

Si la méthode `transfer` découvre que les fonds ne sont pas suffisants, elle appelle `sufficientFunds.await()`;

Le thread actuel est maintenant bloqué et abandonne le verrou, ce qui permet à un autre thread d'entrer et, nous l'espérons, d'augmenter le solde.

Il existe une différence essentielle entre un thread qui attend d'acquiescer un verrou et un thread qui a appelé `await`. Lorsqu'un thread appelle la méthode `await`, il entre un *jeu d'attente* pour cette condition. Le thread n'est *pas* débloqué lorsque le verrou est disponible. Il reste bloqué jusqu'à ce qu'un autre thread appelle la méthode `signalAll` sur la même condition.

Lorsqu'un autre thread transfère de l'argent, il doit appeler

```
sufficientFunds.signalAll();
```

Cet appel débloque tous les threads qui attendent la condition. Lorsque les threads sont retirés du jeu d'attente, ils redeviennent exécutables et le gestionnaire finira par les réactiver. A ce moment-là, ils tenteront de rentrer dans l'objet. Dès que le verrou est disponible, l'un d'entre eux acquerra le verrou et *continuera là où il s'est arrêté*, en revenant de l'appel à `await`.

A ce moment-là, le thread doit à nouveau tester la condition. Il n'est pas garanti que la condition soit remplie. La méthode `signalAll` signale simplement aux threads en attente qu'elle *peut être* remplie à ce moment-là et qu'il faut à nouveau vérifier la condition.

INFO

En général, un appel à `await` doit toujours se trouver dans une boucle sous la forme

```
while (!(ok pour continuer))  
    condition.await();
```

38 Au cœur de Java 2 - Fonctions avancées

Il est particulièrement important qu'un autre thread appelle la méthode `signalAll`. Lorsqu'un thread appelle `await`, il n'a aucun moyen de se débloquent. Il met sa confiance dans les autres threads. Si aucun d'entre eux ne se préoccupe de débloquent le thread en attente, il ne sera jamais réexécuté. Ceci peut mener à des situations déplaisantes de *verrous morts*. Si tous les autres threads sont bloqués et que le dernier thread actif appelle `await` sans débloquent l'un des autres, il bloque aussi. Aucun thread n'ayant l'autorisation de débloquent les autres, le programme plante.

A quel moment appeler `signalAll` ? En règle générale, il vaut mieux l'appeler dès que l'état d'un objet change d'une manière qui pourrait être avantageuse pour les threads en attente. Par exemple, dès qu'un solde bancaire change, les threads en attente doivent avoir une autre occasion d'inspecter le solde. Dans notre exemple, nous appelons `signalAll` lorsque nous avons fini le transfert des fonds.

```
public void transfer(int from, int to, int amount)
{
    bankLock.lock();
    try
    {
        while (accounts[from] < amount)
            sufficientFunds.await();
        // transfert des fonds
        . . .
        sufficientFunds.signalAll();
    }
    finally
    {
        bankLock.unlock();
    }
}
```

Vous remarquerez que l'appel à `signalAll` n'active pas immédiatement un thread en attente. Il ne fait que débloquent les threads en attente de sorte qu'ils puissent entrer en concurrence pour entrer dans l'objet après que le thread actuel est sorti de la méthode synchronisée.

Une autre méthode, `signal`, ne débloquent qu'un thread choisi au hasard dans le jeu d'attente. Ceci est plus efficace que débloquent tous les threads, mais il y a un danger. Si le thread choisi aléatoirement découvre qu'il ne peut toujours pas poursuivre, il est à nouveau bloqué. Si aucun autre thread n'appelle `signal`, le système génère des *verrous morts*.

ATTENTION

Un thread ne peut appeler que `await`, `signalAll` ou `signal` sur une condition lorsqu'il possède le verrou de la condition.

Si vous exécutez le programme de l'Exemple 1.4, vous verrez que tout va bien. Le solde reste à 100 000 €. Aucun compte n'aura jamais de solde négatif (appuyez sur Ctrl+C pour mettre fin au programme). Vous pouvez aussi remarquer que le programme est un peu plus lent : c'est le prix à payer pour la charge supplémentaire impliquée par la synchronisation.

Exemple 1.4 : `SynchBankTest.java`

```
import java.util.concurrent.locks.*;
```

```

/**
 * Ce programme montre comment plusieurs threads peuvent
 * accéder en sécurité à une structure de données.
 */
public class SynchBankTest
{
    public static void main(String[] args)
    {
        Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
        int i;
        for (i = 0; i < NACCOUNTS; i++)
        {
            TransferRunnable r = new TransferRunnable(b, i,
                INITIAL_BALANCE);
            Thread t = new Thread(r);
            t.start();
        }

        public static final int NACCOUNTS = 100;
        public static final double INITIAL_BALANCE = 1000;
    }

/**
 * Une banque avec plusieurs comptes.
 */
class Bank
{
    /**
     * Construit la banque.
     * @param n le nombre de comptes
     * @param initialBalance le solde initial
     * pour chaque compte
     */
    public Bank(int n, double initialBalance)
    {
        accounts = new double[n];
        for (int i = 0; i < accounts.length; i++)
            accounts[i] = initialBalance;
        bankLock = new ReentrantLock();
        sufficientFunds = bankLock.newCondition();
    }

    /**
     * Transfère l'argent d'un compte à l'autre.
     * @param from le compte d'origine
     * @param to le compte de destination
     * @param amount le montant à transférer
     */
    public void transfer(int from, int to, double amount)
        throws InterruptedException
    {
        bankLock.lock();
        try
        {
            while (accounts[from] < amount)
                sufficientFunds.await();
            System.out.print(Thread.currentThread());
            accounts[from] -= amount;

```

40 Au cœur de Java 2 - Fonctions avancées

```

        System.out.printf(" %10.2f de %d à %d", amount, from, to);
        accounts[to] += amount;
        System.out.printf(" Solde Total : %10.2f%n", getTotalBalance());
        sufficientFunds.signalAll();
    }
    finally
    {
        bankLock.unlock();
    }
}

/**
 * Récupère la somme de tous les soldes.
 * @return le solde total
 */
public double getTotalBalance()
{
    bankLock.lock();
    try
    {
        double sum = 0;

        for (double a : accounts)
            sum += a;

        return sum;
    }
    finally
    {
        bankLock.unlock();
    }
}

/**
 * Récupère le nombre de comptes à la banque.
 * @return le nombre de comptes
 */
public int size()
{
    return accounts.length;
}

private final double[] accounts;
private Lock bankLock;
private Condition sufficientFunds;
}

/**
 * Un exécutable qui transfère l'argent d'un compte à un
 * autre dans une banque.
 */
class TransferRunnable implements Runnable
{
    /**
     * Construit un exécutable de transfert.
     * @param b la banque dont l'argent est transféré
     * @param from le compte de destination
     * @param max montant maximum de chaque transfert
     */

```

```
public TransferRunnable(Bank b, int from, double max)
{
    bank = b;
    fromAccount = from;
    maxAmount = max;
}

public void run()
{
    try
    {
        while (true)
        {
            int toAccount = (int) (bank.size() * Math.random());
            double amount = maxAmount * Math.random();
            bank.transfer(fromAccount, toAccount, amount);
            Thread.sleep((int) (DELAY * Math.random()));
        }
    }
    catch (InterruptedException e) {}
}

private Bank bank;
private int fromAccount;
private double maxAmount;
private int repetitions;
private int DELAY = 10;
}
```

java.util.concurrent.locks.Lock 5.0

- `Condition newCondition()`
Renvoie un objet de condition associé à ce verrou.
- `void await()`
Place ce thread dans le jeu d'attente pour cette condition.
- `void signalAll()`
Débloque tous les threads du jeu d'attente pour cette condition.
- `void signal()`
Débloque un thread choisi aléatoirement dans le jeu d'attente pour cette condition.

Le mot clé synchronized

Dans les sections précédentes, vous avez vu comment utiliser Lock et les objets Condition. Avant de poursuivre, résumons les points principaux concernant les verrous et les conditions :

- Un verrou protège des sections de code, ne permettant qu'à un seul thread d'exécuter du code à un moment donné.
- Un verrou gère les threads qui tentent d'entrer dans un segment de code protégé.
- Un verrou peut se voir associer un ou plusieurs objets de condition.
- Chaque objet de condition gère les threads qui sont entrés dans une section de code protégée mais ne peuvent pas poursuivre.

42 Au cœur de Java 2 - Fonctions avancées

Avant l'ajout des interfaces `Lock` et `Condition` au JDK 5.0, le langage Java utilisait un autre mécanisme pour la simultanéité. Depuis la version 1.0, *chaque objet* de Java possède un verrou implicite. Si une méthode est déclarée avec le mot clé `synchronized`, le verrou de l'objet protège toute la méthode. Pour appeler la méthode, un thread doit donc acquérir le verrou de l'objet.

Autrement dit,

```
public synchronized void method()
{
    corps de la méthode
}
```

équivalent à

```
public void method()
{
    implicitLock.lock();
    try
    {
        corps de la méthode
    }
    finally { implicitLock.unlock(); }
}
```

Par exemple, au lieu d'utiliser un verrou explicite, nous pouvons simplement déclarer comme `synchronized` la méthode `transfer` de la classe `Bank`.

Le verrou d'objet implicite ne s'est vu associer qu'une seule condition. La méthode `wait` ajoute un thread au jeu d'attente et les méthodes `notifyAll` et `notify` débloquent les threads en attente. Autrement dit, appeler `wait` ou `notifyAll` équivaut à

```
implicitCondition.await();
implicitCondition.signalAll();
```

INFO

Les méthodes `wait` et `signal` appartiennent à la classe `Object`. Les méthodes `Condition` ont dû être nommées `await` et `signalAll` pour ne pas entrer en conflit avec les précédentes.

Vous pouvez, par exemple, implémenter la classe `Bank` en Java comme suit :

```
class Bank
{
    public synchronized void transfer(int from, int to, int amount)
        throws InterruptedException
    {
        while (accounts[from] < amount)
            wait(); // attend la seule condition du verrou de l'objet
        accounts[from] -= amount;
        accounts[to] += amount;
        notifyAll(); // avertir tous les threads attendant la condition
    }
    public synchronized double getTotalBalance() { . . . }
    private double accounts[];
}
```

Comme vous le voyez, l'utilisation du mot clé `synchronized` permet un code bien plus concis. Pour le comprendre, vous devez savoir que chaque objet possède un verrou implicite et que le verrou a une condition implicite. Le verrou gère les threads qui tentent d'entrer dans une méthode `synchronized`. La condition gère les threads qui ont appelé `wait`.

Cependant, les verrous implicites et les conditions présentent certaines limites, et parmi elles :

- Vous ne pouvez pas interrompre un thread qui tente d'acquérir un verrou.
- Vous ne pouvez pas spécifier de temporisation lorsque vous tentez d'acquérir un verrou.
- Ne disposer que d'une seule condition par verrou peut se révéler insuffisant.
- Les primitives de verrouillage de la machine virtuelle ne concordent pas bien avec les mécanismes de verrouillage les plus efficaces disponibles sur le matériel.

On peut alors s'interroger sur ce qu'il faut utiliser dans le code : des objets `Lock` et `Condition` ou des méthodes synchronisées ? Voici quelques conseils :

1. Il vaut mieux n'utiliser ni `Lock/Condition` ni le mot clé `synchronized`. Dans de nombreux cas, vous pourrez utiliser l'un des mécanismes du paquetage `java.util.concurrent` qui gère le verrouillage. Vous verrez plus loin comment utiliser une queue de blocage pour synchroniser des threads qui agissent sur une même tâche.
2. Si le mot clé `synchronized` fonctionne pour vous, utilisez-le. Vous écrirez moins de code, ce qui réduit les risques d'erreur. Les exemples 1 à 5 illustrent l'exemple de la banque, implémenté avec des méthodes synchronisées.
3. Utilisez `Lock/Condition` si vous avez particulièrement besoin de la puissance offerte par ces constructions.

INFO

L'utilisation du mot clé `synchronized` présente, du moins pour l'instant, un avantage supplémentaire. Les outils de surveillance de la machine virtuelle peuvent signaler les verrous implicites et les conditions, ce qui aide au débogage des verrous morts. Il faudra attendre un peu pour que ces outils soient étendus aux mécanismes de `java.util.concurrent`.

Exemple 1.5 : `SynchBankTest2.java`

```
/**
 * Ce programme montre comment plusieurs threads peuvent accéder
 * en toute sécurité à une structure de données, à l'aide des
 * méthodes synchronisées.
 */
public class SynchBankTest2
{
    public static void main(String[] args)
    {
        Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
        int i;
        for (i = 0; i < NACCOUNTS; i++)
        {
            TransferRunnable r = new TransferRunnable(b, i,
                INITIAL_BALANCE);
            Thread t = new Thread(r);
```

44 Au cœur de Java 2 - Fonctions avancées

```

        t.start();
    }
}

public static final int NACCOUNTS = 100;
public static final double INITIAL_BALANCE = 1000;
}

/**
 * Une banque avec plusieurs comptes.
 */
class Bank
{
    /**
     * Construit la banque.
     * @param n le nombre de comptes
     * @param initialBalance le solde initial
     * pour chaque compte
     */
    public Bank(int n, double initialBalance)
    {
        accounts = new double[n];
        for (int i = 0; i < accounts.length; i++)
            accounts[i] = initialBalance;
    }

    /**
     * Transfère de l'argent d'un compte à l'autre.
     * @param from le compte d'origine du transfert
     * @param to le compte destinataire
     * @param amount le montant à transférer
     */
    public synchronized void transfer(int from, int to, double amount)
        throws InterruptedException
    {
        while (accounts[from] < amount)
            wait();
        System.out.print(Thread.currentThread());
        accounts[from] -= amount;
        System.out.printf(" %10.2f de %d à %d", amount, from, to);
        accounts[to] += amount;
        System.out.printf(" Solde total : %10.2f%n", getTotalBalance());
        notifyAll();
    }

    /**
     * Récupère la somme de tous les soldes.
     * @return le solde total
     */
    public synchronized double getTotalBalance()
    {
        double sum = 0;

        for (double a : accounts)
            sum += a;

        return sum;
    }
}

```

```

    /**
     * Récupère le nombre de comptes de la banque.
     * @return le nombre de comptes
     */
    public int size()
    {
        return accounts.length;
    }

    private final double[] accounts;
}

/**
 * Un exécutable qui transfère de l'argent d'un compte
 * à un autre dans une banque.
 */
class TransferRunnable implements Runnable
{
    /**
     * Construit un exécutable de transfert.
     * @param b la banque où s'effectuent les transferts
     * @param from le compte d'origine du transfert
     * @param max le montant maximum de chaque transfert
     */
    public TransferRunnable(Bank b, int from, double max)
    {
        bank = b;
        fromAccount = from;
        maxAmount = max;
    }

    public void run()
    {
        try
        {
            while (true)
            {
                int toAccount = (int) (bank.size() * Math.random());
                double amount = maxAmount * Math.random();
                bank.transfer(fromAccount, toAccount, amount);
                Thread.sleep((int) (DELAY * Math.random()));
            }
        }
        catch (InterruptedException e) {}
    }

    private Bank bank;
    private int fromAccount;
    private double maxAmount;
    private int repetitions;
    private int DELAY = 10;
}

```

46 Au cœur de Java 2 - Fonctions avancées

java.lang.Object 1.0

- `void notifyAll()`

Débloque les threads qui ont appelé `wait` sur cet objet. Cette méthode ne peut être appelée que depuis une méthode synchronisée ou un bloc. La méthode déclenche une `IllegalMonitorStateException` si le thread courant n'est pas propriétaire du verrou d'objet.

- `void notify()`

Débloque un thread choisi aléatoirement parmi ceux qui ont appelé `wait` sur cet objet. Cette méthode ne peut être appelée que depuis une méthode synchronisée ou un bloc. La méthode déclenche une `IllegalMonitorStateException` si le thread courant n'est pas propriétaire du verrou d'objet.

- `void wait()`

Amène un thread à patienter jusqu'à ce qu'il soit notifié. Cette méthode ne peut être appelée que depuis une méthode synchronisée. Elle déclenche une `IllegalMonitorStateException` si le thread courant n'est pas propriétaire du verrou d'objet.

- `void wait(long millis)`

- `void wait(long millis, int nanos)`

Amène un thread à patienter jusqu'à ce qu'il soit notifié ou jusqu'à ce que le délai spécifié se soit écoulé. Ces méthodes ne peuvent être appelées que depuis une méthode synchronisée. Elles déclenchent une `IllegalMonitorStateException` si le thread courant n'est pas propriétaire du verrou d'objet.

<i>Paramètres :</i>	<code>millis</code>	Le nombre de millisecondes
	<code>nanos</code>	Le nombre de nanosecondes, < 1 000 000

Moniteurs

Les verrous et les conditions constituent des outils puissants pour la synchronisation des threads, mais ils ne sont pas vraiment orientés objet. Pendant de nombreuses années, les chercheurs ont tenté de sécuriser le multithread, sans obliger les programmeurs à penser aux verrous explicites. L'une des solutions les plus performantes est le concept de *moniteur* présenté par Per Brinch Hansen et Tony Hoare dans les années 1970. En terminologie Java, un moniteur présente les propriétés suivantes :

- Un moniteur est une classe n'ayant que des champs privés.
- Chaque objet de cette classe se voit associer un verrou.
- Toutes les méthodes sont verrouillées par ce verrou. Autrement dit, si un client appelle `obj.method()`, le verrou de `obj` est automatiquement acquis au début de l'appel de méthode et abandonné à la fin. Tous les champs étant privés, cet arrangement permet de s'assurer qu'aucun thread n'a accès aux champs, alors qu'un autre thread les manipule.
- Le verrou peut posséder n'importe quel nombre de conditions associées.

Les précédentes versions des moniteurs possédaient une seule condition, avec une syntaxe plutôt agréable. Vous pouvez simplement appeler `await accounts[from] >= balance` sans utiliser de variable de condition explicite. Toutefois, les recherches ont montré qu'un nouveau test des conditions peut se révéler inefficace. Ce problème est résolu avec les variables de conditions explicites, chacune gérant un jeu de threads distinct.

Les concepteurs de Java ont adapté le concept du moniteur. *Chaque objet* de Java possède un verrou implicite et une condition implicite. Si une méthode est déclarée avec le mot clé `synchronized`, elle agit comme une méthode moniteur. La variable de condition est accessible par un appel à `wait/notify/notifyAll`.

Blocs synchronisés

Toutefois, un objet Java diffère d'un moniteur en trois points :

- Les champs n'ont pas à être privés.
- Les méthodes n'ont pas à être synchronisées.
- Le verrou n'a qu'une condition.

Si vous travaillez avec du code existant, vous devez savoir une chose sur les primitives de synchronisation intégrées. Souvenez-vous que chaque objet possède un verrou. Un thread peut acquérir le verrou en appelant une méthode synchronisée ou en entrant dans un *bloc synchronisé*. Si le thread appelle `obj.method()`, il acquiert le verrou pour `obj`. De même, si un thread entre dans un bloc de la forme

```
synchronized (obj) // syntaxe pour un bloc synchronisé
{
    Section principale
}
```

il acquiert le verrou pour `obj`. Le verrou est rentrant. Si un thread a acquis le verrou, il peut l'acquérir à nouveau, en incrémentant le compte. En particulier, une méthode synchronisée peut appeler d'autres méthodes synchronisées avec le même paramètre implicite, sans avoir à attendre le verrou.

Vous trouverez souvent des verrous *ad hoc* dans un code existant, comme

```
class Bank
{
    public void transfer(int from, int to, int amount)
    {
        synchronized (lock) // un verrou ad-hoc
        {
            accounts[from] -= amount;
            accounts[to] += amount;
        }
        System.out.println(. . .);
    }
    . . .
    private double accounts[];
    private Object lock = new Object(); }
```

Ici, l'objet `lock` n'est créé que pour utiliser le verrou que possède chaque objet Java.

Vous pouvez déclarer des méthodes statiques comme synchronisées. Si l'on appelle une telle méthode, elle acquiert le verrou de l'objet de classe associé. Par exemple, si la classe `Bank` possède une méthode statique synchronisée, le verrou de l'objet `Bank.class` est verrouillé lorsqu'il est appelé.

48 Au cœur de Java 2 - Fonctions avancées

Verrous morts

Les verrous et les conditions ne peuvent pas résoudre tous les problèmes qui se posent en multithreads. Considérons la situation suivante :

Compte 1 : 1 200 €.

Compte 2 : 1 300 €.

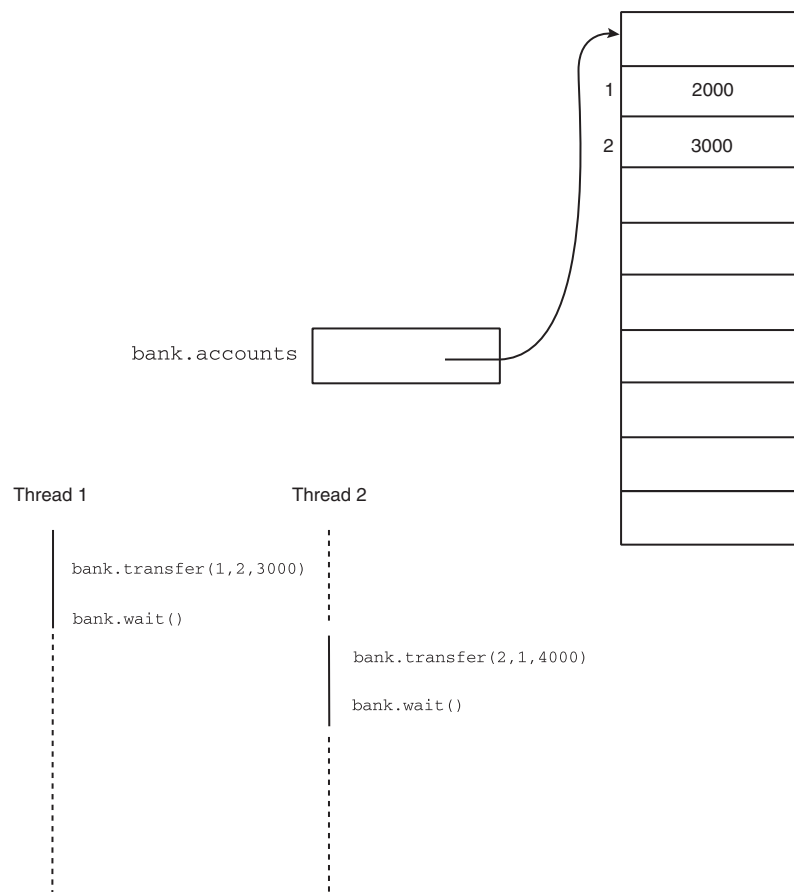
Thread 1 : transfère 300 € du compte 1 vers le compte 2.

Thread 2 : transfère 400 € du compte 2 vers le compte 1.

Comme le montre la Figure 1.6, les threads 1 et 2 sont clairement bloqués. Aucun d'entre eux ne peut continuer puisque les soldes des comptes 1 et 2 sont trop faibles.

Figure 1.6

Une situation de verrous morts.



Est-il possible que tous les threads soient bloqués parce que chacun d'entre eux attend un transfert d'argent ? Ce type de situation est appelé un *verrou mort*.

Dans notre programme, un verrou mort ne peut pas se produire pour une raison très simple. Chaque transfert d'argent s'élève au plus à 1 000 €. Comme il existe dix comptes et un total de 100 000 €, au

moins un des comptes doit avoir plus de 1 000 € à n'importe quel moment. Le thread qui sort de l'argent de ce compte peut par conséquent être traité.

Mais si vous modifiez la méthode `run` des threads pour supprimer la limite de 1 000 € pour les transactions, des verrous morts peuvent se produire rapidement. Vous pouvez essayer. Définissez `NACCOUNTS` sur 10. Construisez chaque thread de transfert avec un `maxAmount` de 2 000 € et exécutez le programme. Ce dernier s'exécutera pendant un moment, puis il se bloquera.

ASTUCE

Lorsque le programme se bloque, tapez `Ctrl+\`. Tous les threads seront listés. Chaque thread possède une trace, vous indiquant l'endroit de son blocage.

Une autre manière de créer un verrou mort est de demander au *i*-ème thread de placer de l'argent sur le *i*-ème compte, plutôt que de lui demander de retirer de l'argent de ce compte. Dans ce cas, il est possible que tous les threads soient bloqués sur un compte, chacun essayant de supprimer plus d'argent de ce compte qu'il n'en contient. Vous pouvez également essayer cela. Dans le programme `SynchBankTest`, examinez la méthode `run` de la classe `TransferRunnable`. Dans l'appel à `transfer`, échangez `fromAccount` et `toAccount`. Exécutez le programme, vous verrez qu'il se bloque presque immédiatement.

Voici une autre situation dans laquelle un verrou mort peut se produire très facilement : utilisez la méthode `signal` à la place de `signalAll` dans le programme `SynchBankTest`. Vous verrez que ce programme se bloque rapidement. Contrairement à `signalAll`, qui prévient tous les threads en attente que de nouveaux fonds sont disponibles, la méthode `signal` débloque un seul thread. Si ce thread ne peut pas continuer, tous les threads seront bloqués. Considérons le scénario suivant d'un verrou mort en cours de création :

Compte 1 : 1 990 €.

Tous les autres comptes : 990 € chacun.

Thread 1 : transfère 995 € du compte 1 vers le compte 2.

Tous les autres threads : transfèrent 995 € de leur compte vers un autre compte.

Il est alors évident que tous les threads sauf le thread 1 sont bloqués puisqu'il n'y a pas assez d'argent sur leur compte.

Le thread 1 continue. Par la suite, nous avons la situation suivante :

Compte 1 : 995 €.

Compte 2 : 1 985 €.

Tous les autres comptes : 990 € chacun.

Le thread 1 appelle alors `signal`. La méthode `signal` choisit aléatoirement un thread à débloquer. Supposons qu'elle choisisse le thread 3. Ce thread est réveillé, il détermine qu'il n'y a pas assez d'argent sur son compte et appelle `wait` une nouvelle fois. Mais le thread 1 est toujours en cours d'exécution. Une nouvelle transaction aléatoire est générée :

Thread 1 : transfère 997 € du compte 1 vers le compte 2.

50 Au cœur de Java 2 - Fonctions avancées

Maintenant, le thread 1 appelle également `await`, et *tous* les threads sont bloqués. Le système se trouve alors dans un verrou mort.

Le problème vient ici de la méthode `signal`. En effet, elle ne débloquent qu'un seul thread, et elle peut ne pas choisir le thread essentiel à la bonne marche du programme. Dans notre scénario, le thread 2 doit être activé pour retirer de l'argent du compte 2.

Malheureusement, il n'existe aucun mécanisme dans le langage de programmation Java pour éviter ou pour casser ces verrous morts. Vous devez concevoir des threads de sorte qu'une situation de verrou mort ne puisse survenir.

Équité

Lorsque vous construisez un `ReentrantLock`, vous pouvez spécifier que vous souhaitez adopter une *politique de verrouillage équitable*.

```
Lock fairLock = new ReentrantLock(true);
```

Un verrou équitable favorise le thread qui attend depuis le plus longtemps. Toutefois, ceci peut considérablement réduire les performances. Par défaut, les verrous ne sont donc pas équitables.

Même avec un verrou équitable, vous n'avez aucune garantie que le gestionnaire de threads soit lui-même équitable. S'il choisit de négliger un thread qui attend le verrou depuis longtemps, il n'aura aucune chance d'être traité équitablement par le verrou.

ATTENTION

L'équité est un concept plaisant, mais les verrous équitables sont *bien plus lents* que les verrous ordinaires. N'activez l'équité que si vous avez une bonne raison de le faire. C'est une technique véritablement avancée.

java.util.concurrent.locks.ReentrantLock 5.0

- `ReentrantLock(boolean fair)`

Construit un verrou avec la politique d'équité donnée.

Pourquoi les méthodes `stop` et `suspend` ne sont plus utilisées

La plate-forme Java 1.0 a défini une méthode `stop` qui se contente de mettre fin à un thread, et une méthode `suspend` qui bloque un thread jusqu'à ce qu'un autre thread appelle `resume`. Ces deux méthodes ont quelque chose en commun : elles tentent de contrôler le comportement d'un thread donné sans sa coopération.

Ces deux méthodes ne sont plus utilisées dans la plate-forme Java 2. La méthode `stop` est intrinsèquement non sûre et l'expérience a montré que la méthode `suspend` amène souvent à des situations de verrous morts. Dans cette section, nous allons voir pourquoi ces méthodes posent problème, et ce que nous pouvons faire pour éviter ces problèmes.

Commençons par étudier la méthode `stop`. Cette méthode met fin à toutes les méthodes en attente, y compris à la méthode `run`. Lorsqu'un thread est arrêté, il rend immédiatement les verrous sur tous les objets qu'il a verrouillés. Cela peut laisser les objets dans un état incohérent. Par exemple, supposons qu'un `TransferThread` soit arrêté en plein milieu d'un déplacement d'argent d'un compte vers

un autre, après le retrait et avant le dépôt. L'objet de banque est alors *endommagé*. Le verrou ayant été abandonné, les dommages peuvent être constatés sur les autres threads qui n'ont pas été arrêtés.

Lorsqu'un thread veut arrêter un autre thread, il n'a aucun moyen de savoir à quel moment l'emploi de la méthode `stop` est sûr, et à quel moment il risque d'endommager des objets. Par conséquent, cette méthode n'est plus utilisée. Il est conseillé d'interrompre un thread lorsque vous voulez l'arrêter. Le thread interrompu peut alors s'arrêter au bon moment.

INFO

Certains auteurs déclarent que la méthode `stop` n'est plus utilisée car elle peut verrouiller de manière permanente d'autres objets du fait d'un thread arrêté. Toutefois, ceci est faux. Un thread arrêté ferme toutes les méthodes synchronisées qu'il a appelées (par le traitement de l'exception `ThreadDeath`). En conséquence, le thread libère les verrous d'objet qu'il possède.

Voyons maintenant ce qui peut poser problème avec la méthode `suspend`. Contrairement à `stop`, `suspend` ne peut pas endommager les objets. Cependant, si vous suspendez un thread qui possède un verrou sur un objet, cet objet ne sera plus disponible jusqu'à ce que le thread ait repris son activité. Si le thread qui appelle la méthode `suspend` essaie d'obtenir le verrou sur le même objet, le programme se bloque : le thread interrompu attend qu'on lui demande de reprendre son activité, et le thread qu'il a mis en attente attend que l'objet soit déverrouillé.

Cette situation se produit fréquemment avec des interfaces utilisateurs graphiques. Supposons que nous ayons une simulation graphique de notre banque. Nous disposons d'un bouton "Pause" qui suspend les threads de transfert et d'un bouton "Reprise" qui les fait continuer.

```
pauseButton.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            for (int i = 0; i < threads.length; i++)
                threads[i].suspend(); // Ne pas faire cela
        }
    });
resumeButton.addActionListener(...); //appelle resume sur tous les
//threads de transfert
```

Une méthode `paintComponent` dessine un graphique pour chaque compte, en appelant la méthode `getBalances`.

Comme vous le verrez dans la prochaine section, les actions des boutons et l'action graphique se trouvent dans le même thread, appelé *thread de répartition des événements*.

Considérons maintenant le scénario suivant :

1. L'un des threads de transfert obtient un verrou sur l'objet `bank`.
2. L'utilisateur clique sur le bouton "Pause".
3. Tous les threads de transfert sont suspendus ; l'un d'entre eux possède toujours le verrou sur l'objet `bank`.
4. Pour une certaine raison, le graphique doit être redessiné.

52 Au cœur de Java 2 - Fonctions avancées

5. La méthode `paintComponent` appelle la méthode synchronisée `getBalances`.

6. Cette méthode tente d'acquieser le verrou de l'objet `bank`.

Le programme est alors bloqué.

Le thread de répartition des événements ne peut pas continuer parce que le verrou appartient à l'un des threads suspendus. Par conséquent, l'utilisateur ne peut pas cliquer sur le bouton "Reprise", et aucun thread ne continuera son exécution.

Si vous souhaitez interrompre un thread de manière sûre, il convient d'introduire une variable `suspendRequested` et de la tester dans un lieu sûr de votre méthode `run`, où votre thread ne verrouille pas des objets dont d'autres threads ont besoin. Lorsque cette variable a été définie, continuez à attendre jusqu'à ce qu'elle soit à nouveau disponible.

Le code suivant implémente cette conception :

```
public void run()
{
    while ( . . . )
    {
        . . .
        if (suspendRequested)
        {
            suspendLock.lock();
            try { while (suspendRequested) suspendCondition.await(); }
            finally { suspendLock.unlock(); }
        }
    }
}

public void requestSuspend() { suspendRequested = true; }
public void requestResume()
{
    suspendRequested = false;
    suspendLock.lock();
    try { suspendCondition.signalAll(); }
    finally { suspendLock.unlock(); }
}

private volatile boolean suspendRequested = false;
private Lock suspendLock = new ReentrantLock();
private Condition suspendCondition = suspendLock.newCondition();
```

Queues de blocage

Une *queue* est une structure de données qui réalise deux opérations fondamentales : l'ajout d'un élément à la *queue* de la queue et la suppression d'un élément à la *tête*. En fait, la queue gère les données sur le mode premier entré, premier sorti. Une *queue de blocage* fait bloquer un thread lorsque vous tentez d'ajouter un élément alors que la queue est pleine ou de supprimer un élément alors que la queue est vide. Les queues de blocage constituent un outil utile pour coordonner le travail de plusieurs threads. Les threads travailleurs peuvent déposer périodiquement des résultats intermédiaires dans une queue de blocage. D'autres threads travailleurs supprimeront les résultats intermédiaires et les modifieront. La queue équilibre automatiquement la charge de travail. Si le premier jeu de threads s'exécute plus lentement que le deuxième, le deuxième se bloque en attendant les résultats.

Si le premier jeu de threads s'exécute plus rapidement, la queue se remplit jusqu'à ce que le deuxième le rattrape. Le Tableau 1.1 montre le fonctionnement des queues de blocage.

Tableau 1.1 : Fonctionnement des queues de blocage

Méthode	Action normale	Action en cas d'erreur
add	Ajoute un élément	Déclenche une <code>IllegalStateException</code> si la queue est pleine
remove	Supprime et renvoie l'élément de tête	Déclenche une <code>NoSuchElementException</code> si la queue est vide
element	Revoie l'élément de tête	Déclenche une <code>NoSuchElementException</code> si la queue est vide
offer	Ajoute un élément et renvoie <code>true</code>	Revoie <code>false</code> si la queue est pleine
poll	Supprime et renvoie l'élément de tête	Revoie <code>null</code> si la queue était vide
peek	Revoie l'élément de tête	Revoie <code>null</code> si la queue était vide
put	Ajoute un élément	Bloque si la queue est pleine
take	Supprime et renvoie la tête	Bloque si la queue était vide

Le fonctionnement des queues de blocage est réparti en trois catégories, en fonction de la réponse apportée. Les opérations `add`, `remove` et `element` déclenchent une exception lorsque vous tentez d'ajouter des éléments à une queue pleine ou de récupérer la tête d'une queue vide. Bien entendu, dans un programme multithread, la queue peut se remplir ou se vider à tout moment, il faudra donc plutôt utiliser les méthodes `offer`, `poll` et `peek`. Elles renvoient simplement un indicateur d'échec au lieu de déclencher une exception si elles ne peuvent pas réaliser leurs tâches.

INFO

Les méthodes `poll` et `peek` renvoient `null` pour signaler une erreur. L'insertion de valeurs `null` dans ces queues n'est donc pas autorisée.

Il existe aussi des variantes des méthodes `offer` et `poll` avec temporisation. Par exemple, l'appel

```
boolean success = q.offer(x, 100, TimeUnit.MILLISECONDS);
```

tente pendant 100 millisecondes d'insérer un élément à la queue de la file. S'il réussit, il renvoie immédiatement `true`, sinon il renvoie `false` à la fin de la temporisation. De même, l'appel

```
Object head = q.poll(100, TimeUnit.MILLISECONDS)
```

renvoie `true` pendant 100 millisecondes pour supprimer la tête de la file. S'il réussit, il renvoie immédiatement la tête, sinon il renvoie `null` à la fin de la temporisation.

Enfin, il existe les opérations de blocage `put` et `take`. La méthode `put` bloque si la queue est pleine, la méthode `take` bloque si la queue est vide. Ce sont les équivalents d'`offer` et de `poll`, sans temporisation.


```

        String directory = in.nextLine();
        System.out.print("Entrez le mot clé (par ex. volatil) : ");
        String keyword = in.nextLine();

        final int FILE_QUEUE_SIZE = 10;
        final int SEARCH_THREADS = 100;

        BlockingQueue<File> queue = new
            ArrayBlockingQueue<File>(FILE_QUEUE_SIZE);

        FileEnumerationTask enumerator = new FileEnumerationTask(queue,
            new File(directory));
        new Thread(enumerator).start();
        for (int i = 1; i <= SEARCH_THREADS; i++)
            new Thread(new SearchTask(queue, keyword)).start();
    }
}

/**
 * Cette tâche recense tous les fichiers d'un répertoire et de
 * ses sous-répertoires.
 */
class FileEnumerationTask implements Runnable
{
    /**
     * Construit un FileEnumerationTask.
     * @param queue la file d'attente de blocage à laquelle sont ajoutés
     *             les fichiers recensés
     * @param startingDirectory le répertoire dans lequel commencer
     *             le recensement
     */
    public FileEnumerationTask(BlockingQueue<File> queue,
        File startingDirectory)
    {
        this.queue = queue;
        this.startingDirectory = startingDirectory;
    }

    public void run()
    {
        try
        {
            enumerate(startingDirectory);
            queue.put(DUMMY);
        }
        catch (InterruptedException e) {}
    }

    /**
     * Recense tous les fichiers d'un répertoire donné et de
     * ses sous-répertoires
     * @param directory le répertoire où commencer
     */
    public void enumerate(File directory) throws InterruptedException
    {
        File[] files = directory.listFiles();
        for (File file : files)
        {
            if (file.isDirectory()) enumerate(file);
            else queue.put(file);
        }
    }
}

```

56 Au cœur de Java 2 - Fonctions avancées

```
    }  
}  
  
public static File DUMMY = new File("");  
  
private BlockingQueue<File> queue;  
private File startingDirectory;  
}  
  
/**  
 * Cette tâche recherche les fichiers pour un mot clé donné.  
 */  
class SearchTask implements Runnable  
{  
    /**  
     * Construit un SearchTask.  
     * @param queue la file à partir de laquelle prendre les fichiers  
     * @param keyword le mot clé à rechercher  
     */  
    public SearchTask(BlockingQueue<File> queue, String keyword)  
    {  
        this.queue = queue;  
        this.keyword = keyword;  
    }  
  
    public void run()  
    {  
        try  
        {  
            boolean done = false;  
            while (!done)  
            {  
                File file = queue.take();  
                if (file == FileEnumerationTask.DUMMY) { queue.put(file);  
                    done = true; }  
                else search(file);  
            }  
        }  
        catch (IOException e) { e.printStackTrace(); }  
        catch (InterruptedException e) {}  
    }  
  
    /**  
     * Recherche un fichier pour un mot clé donné et affiche  
     * toutes les lignes correspondantes.  
     * @param file le fichier à rechercher  
     */  
    public void search(File file) throws IOException  
    {  
        Scanner in = new Scanner(new FileInputStream(file));  
        int lineNumber = 0;  
        while (in.hasNextLine())  
        {  
            lineNumber++;  
            String line = in.nextLine();  
            if (line.contains(keyword))  
                System.out.printf("%s:%d:%s\n", file.getPath(),  
                                    lineNumber, line);  
        }  
    }  
}
```

```

        in.close();
    }

    private BlockingQueue<File> queue;
    private String keyword;
}

```

java.util.concurrent.ArrayBlockingQueue<E> 5.0

- `ArrayBlockingQueue(int capacity)`
- `ArrayBlockingQueue(int capacity, boolean fair)`

Construisent une queue de blocage avec la capacité donnée et des paramètres d'équité. La queue est implémentée sous forme de tableau circulaire.

java.util.concurrent.LinkedBlockingQueue<E> 5.0

- `LinkedBlockingQueue()`

Construit une queue de blocage sans bornes, implémentée sous forme de liste chaînée.

- `LinkedBlockingQueue(int capacity)`

Construit une queue de blocage avec la capacité donnée, implémentée sous forme de liste chaînée.

java.util.concurrent.DelayQueue<E extends Delayed> 5.0

- `DelayQueue()`

Construit une queue de blocage d'éléments `Delayed`. Seuls les éléments dont le délai a expiré seront supprimés de la queue.

java.util.concurrent.Delayed 5.0

- `long getDelay(TimeUnit unit)`

Récupère le délai pour cet objet, mesuré dans l'unité de temps donnée.

java.util.concurrent.PriorityBlockingQueue<E> 5.0

- `PriorityBlockingQueue()`
- `PriorityBlockingQueue(int initialCapacity)`
- `PriorityBlockingQueue(int initialCapacity, Comparator<? super E> comparator)`

Construisent une queue d'attente de priorité de blocages sans bornes, implémentée sous forme de tas.

Paramètres :

<code>initialCapacity</code>	La capacité initiale de la queue de priorité. Vaut 11 par défaut.
<code>comparator</code>	Le comparateur utilisé pour les éléments. S'il n'est pas spécifié, les éléments doivent implémenter l'interface <code>Comparable</code> .

java.util.concurrent.BlockingQueue<E> 5.0

- `void put(E element)`
Ajoute l'élément, blocage si nécessaire.
- `boolean offer(E element)`

58 *Au cœur de Java 2 - Fonctions avancées*

- `boolean offer(E element, long time, TimeUnit unit)`
Ajoute l'élément donné et renvoie `true` en cas de succès ou se termine sans ajouter l'élément et renvoie `false` si la queue est pleine. La deuxième méthode bloque si nécessaire, jusqu'à ce que l'élément ait été ajouté ou que le délai soit écoulé.
- `E take()`
Supprime et renvoie l'élément de tête, blocage si nécessaire.
- `E poll(long time, TimeUnit unit)`
Supprime et renvoie l'élément de tête, blocage si nécessaire jusqu'à ce qu'un élément soit disponible ou que le délai soit écoulé. Renvoie `null` en cas d'échec.

java.util.Queue<E> 5.0

- `E poll()`
Supprime et renvoie l'élément de tête ou `null` si la queue est vide.
- `E peek()`
Renvoie l'élément de tête ou `null` si la queue est vide.

Collections compatibles avec les threads

Si plusieurs threads modifient simultanément une structure de données, telle qu'une table de hachage, cette structure de données peut facilement être endommagée (nous verrons les tables de hachage plus en détail au Chapitre 2). Un thread pourrait par exemple commencer à insérer un nouvel élément. Supposons qu'il soit alors préempté au milieu du réacheminement des liens entre les seaux de la table de hachage. Si un autre thread commence à parcourir la même liste, il risque de suivre des liens non valides et de créer un déséquilibre, par exemple en lançant des exceptions ou en s'engageant dans une boucle sans fin.

Vous pouvez protéger une structure de données partagées en fournissant un verrou, mais il est généralement plus simple de choisir une implémentation compatible avec les threads. Les queues de blocage vues précédemment sont, bien entendu, des collections compatibles avec les threads. Dans les sections suivantes, nous en verrons d'autres proposées par la bibliothèque Java.

CopyOnWriteArray

`CopyOnWriteArray` et `CopyOnWriteArraySet` sont des collections compatibles avec les threads, dans lesquelles toutes les méthodes de modification créent une copie du tableau sous-jacent. Cet arrangement est utile lorsque le nombre de threads à parcourir la collection dépasse de loin le nombre de ceux qui la modifient. Lorsque vous construisez un itérateur, il contient une référence au tableau actuel. Si le tableau est modifié par la suite, l'itérateur possède toujours l'ancien tableau, mais celui de la collection est remplacé. Par conséquent, l'itérateur le plus ancien possède une vue cohérente (mais peut-être obsolète) à laquelle il peut accéder sans frais de synchronisation.

Callable et Future

Un `Runnable` encapsule une tâche qui s'exécute de manière asynchrone. C'est en quelque sorte une méthode asynchrone sans paramètres ni valeur de retour. Un `Callable` est identique, mais il renvoie une valeur. L'interface `Callable` est un type à paramètres, avec une seule méthode `call` :

```
public interface Callable<V>
{
    V call() throws Exception;
}
```

Le paramètre de type correspond au type de la valeur renvoyée. Par exemple, `Callable<Integer>` représente un calcul asynchrone qui finit par renvoyer un objet `Integer`.

Un `Future` contient le *résultat* d'un calcul asynchrone. Un `Future` permet de démarrer un calcul, de donner le résultat à quelqu'un, puis de l'oublier. Lorsqu'il est prêt, le propriétaire de l'objet `Future` peut obtenir le résultat.

L'interface `Future` possède les méthodes suivantes :

```
public interface Future<V>
{
    V get() throws...;
    V get(long timeout, TimeUnit unit) throws...;
    void cancel(Boolean mayInterrupt);
    boolean isCancelled();
    boolean isDone();
}
```

Un appel à la première méthode `get` bloque jusqu'à ce que le calcul soit terminé. La deuxième méthode déclenche une `TimeoutException` si l'appel est arrivé à expiration avant la fin du calcul. Si le thread exécutant le calcul est interrompu, les deux méthodes déclenchent une `InterruptedException`. Si le calcul est terminé, `get` prend fin immédiatement.

La méthode `isDone` renvoie `false` si le calcul se poursuit, `true` s'il est terminé.

Vous pouvez annuler le calcul avec la méthode `cancel`. Si le calcul n'a pas encore commencé, il est annulé et ne commencera jamais. Si le calcul est en cours, il est interrompu lorsque le paramètre `mayInterrupt` vaut `true`.

L'emballage `FutureTask` est un mécanisme commode pour transformer un `Callable` à la fois en `Future` et en `Runnable` : il implémente les deux interfaces. Par exemple :

```
Callable<Integer> myComputation = ...;
FutureTask<Integer> task = new FutureTask<Integer>(myComputation);
Thread t = new Thread(task); // c'est un Runnable
t.start();
...
Integer result = task.get(); // c'est un Future
```

60 Au cœur de Java 2 - Fonctions avancées

Le programme de l'Exemple 1.7 illustre ces concepts. Ce programme est identique à l'exemple précédent qui trouvait les fichiers contenant un mot clé. Toutefois, nous allons maintenant simplement compter le nombre de fichiers correspondants. Nous avons donc une longue tâche qui produit une valeur entière ; un exemple de `Callable<Integer>` :

```
class MathCounter implements Callable<Integer>
{
    public MathCounter(File directory, String keyword) { ... }
    public Integer call() { ... } // renvoie le nb de fichiers concordants
}
```

Nous construisons ensuite un objet `FutureTask` depuis `MathCounter` et l'utilisons pour démarrer un thread :

```
FutureTask<Integer> task = new FutureTask<Integer>(counter);
Thread t = new Thread(task);
t.start();
```

Enfin, nous affichons le résultat :

```
System.out.println(task.get() + " fichiers concordants.");
```

Bien entendu, l'appel à `get` bloque jusqu'à ce que le résultat soit disponible.

Dans la méthode `call`, nous utilisons le même mécanisme à plusieurs reprises. Pour chaque sous-répertoire, nous produisons un nouveau `MatchCounter` et lançons un thread pour chacun. Nous répartissons aussi les objets `FutureTask` dans un `ArrayList<Future<Integer>>`. A la fin, nous ajoutons tous les résultats :

```
for (Future<Integer> result : results)
    count += result.get();
```

Chaque appel à `get` bloque jusqu'à ce que le résultat soit disponible. Bien sûr, les threads s'exécutent en parallèle, il y a donc de fortes chances pour que les résultats soient tous disponibles à peu près au même moment.

Exemple 1.7 : FutureTest.java

```
import java.io.*;
import java.util.*;
import java.util.concurrent.*;

public class FutureTest
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Entrez le répertoire de base
                        (par ex. /usr/local/jdk5.0/src): ");
        String directory = in.nextLine();
        System.out.print("Entrez le mot clé (par ex. volatile): ");
        String keyword = in.nextLine();

        MatchCounter counter = new MatchCounter(
            new File(directory), keyword);
        FutureTask<Integer> task = new FutureTask<Integer>(counter);
        Thread t = new Thread(task);
        t.start();
        try
```

```

        {
            System.out.println(task.get() + " fichiers concordants.");
        }
        catch (ExecutionException e)
        {
            e.printStackTrace();
        }
        catch (InterruptedException e) {}
    }
}

/**
 * Cette tâche compte les fichiers d'un répertoire et de ses
 * sous-répertoires qui contiennent un mot clé donné.
 */
class MatchCounter implements Callable<Integer>
{
    /**
     * Construit un MatchCounter.
     * @param directory le répertoire dans lequel commencer la recherche
     * @param keyword le mot clé à rechercher
     */
    public MatchCounter(File directory, String keyword)
    {
        this.directory = directory;
        this.keyword = keyword;
    }

    public Integer call()
    {
        count = 0;
        try
        {
            File[] files = directory.listFiles();
            ArrayList<Future<Integer>> results =
                new ArrayList<Future<Integer>>();

            for (File file : files)
            {
                if (file.isDirectory())
                {
                    MatchCounter counter = new MatchCounter(file, keyword);
                    FutureTask<Integer> task =
                        new FutureTask<Integer>(counter);
                    results.add(task);
                    Thread t = new Thread(task);
                    t.start();
                }
                else
                {
                    if (search(file)) count++;
                }
            }

            for (Future<Integer> result : results)
            {
                try
                {
                    count += result.get();
                }
                catch (ExecutionException e)
                {

```

62 Au cœur de Java 2 - Fonctions avancées

```

        e.printStackTrace();
    }
}
catch (InterruptedException e) {}
return count;
}

/**
 * Recherche un mot clé donné dans un fichier.
 * @param file Le fichier à parcourir
 * @return true si le mot clé figure dans le fichier
 */
public boolean search(File file)
{
    try
    {
        Scanner in = new Scanner(new FileInputStream(file));
        boolean found = false;
        while (!found && in.hasNextLine())
        {
            String line = in.nextLine();
            if (line.contains(keyword)) found = true;
        }
        in.close();
        return found;
    }
    catch (IOException e)
    {
        return false;
    }
}

private File directory;
private String keyword;
private int count;
}

```

java.util.concurrent.Callable<V> 5.0

- **V call()**
Exécute une tâche qui produit un résultat.

java.util.concurrent.Future<V> 5.0

- **V get()**
- **V get(long time, TimeUnit unit)**
Récupèrent le résultat, en bloquant jusqu'à ce qu'il soit disponible ou que le délai soit expiré. La deuxième méthode déclenche une `TimeoutException` en cas d'échec.
- **boolean cancel(boolean mayInterrupt)**
Tente d'annuler l'exécution de cette tâche. Si la tâche a déjà commencé et que le paramètre `mayInterrupt` vaut `true`, elle est interrompue. Renvoie `true` si l'annulation a réussi.
- **boolean isCancelled()**
Renvoie `true` si la tâche a été annulée avant la fin.
- **boolean isDone()**
Renvoie `true` si la tâche est terminée, suite à une fin normale, une annulation ou une exception.

java.util.concurrent.FutureTask<V> 5.0

- FutureTask(Callable<V> task)
- FutureTask(Runnable task, V result)

Construit un objet qui soit à la fois un Future<V> et un Runnable.

Executors

La construction d'un nouveau thread est assez onéreuse car elle implique une interaction avec le système d'exploitation. Si votre programme crée un grand nombre de threads courts, il devra plutôt utiliser un *pool de threads*. Ce pool contient plusieurs threads inactifs, prêts à s'exécuter. Vous attribuez un Runnable au pool et l'un des threads appelle la méthode run. A la fin de la méthode run, le thread ne meurt pas, il demeure pour répondre à la requête suivante.

Il existe une autre raison pour utiliser un pool de threads : l'étranglement de plusieurs threads simultanés. La création d'un grand nombre de threads peut considérablement dégrader les performances, voire bloquer la machine virtuelle. Pour un algorithme créateur de nombreux threads, utilisez un pool de threads "fixe" qui limite le nombre total de threads simultanés.

La classe Executors possède plusieurs méthodes factory statiques destinées à la création de pools de threads (voir Tableau 1.2).

Tableau 1.2 : Méthodes factory Executors

Méthode	Description
newCachedThreadPool	Les nouveaux threads sont créés en fonction des besoins. Les threads inactifs sont conservés pendant 60 secondes.
newFixedThreadPool	Le pool contient un jeu fixe de threads. Les threads inactifs sont conservés indéfiniment.
newSingleThreadExecutor	Un "pool" possédant un seul thread qui exécute les tâches envoyées de manière séquentielle.
newScheduledThreadPool	Un pool de threads fixe pour une exécution programmée.
newSingleThreadScheduledExecutor	Un "pool" d'un seul thread pour une exécution programmée.

Pools de threads

Etudions la première des trois méthodes du Tableau 1.2. Nous verrons les autres par la suite. La méthode newCachedThreadPool construit un pool de threads qui exécute chaque tâche immédiatement, à l'aide d'un thread inactif existant lorsqu'il est disponible ou en créant un nouveau thread dans le cas contraire. La méthode newFixedThreadPool construit un pool de threads de taille fixe. Si le nombre de tâches envoyées est supérieur au nombre de threads inactifs, les tâches non servies sont placées dans une queue. Elles sont exécutées à la fin des autres tâches. newSingleThreadExecutor est un pool dégénéré, de taille 1 : un seul thread exécute les tâches envoyées, l'une après

64 Au cœur de Java 2 - Fonctions avancées

l'autre. Ces trois méthodes renvoient un objet de la classe `ThreadPoolExecutor` qui implémente l'interface `ExecutorService`.

Vous pouvez envoyer un `Runnable` ou un `Callable` à un `ExecutorService` de l'une des manières suivantes :

```
Future<?> submit(Runnable task)
Future<T> submit(Runnable task, T result)
Future<T> submit(Callable<T> task)
```

Le pool exécutera la tâche envoyée le plus tôt possible. Lorsque vous appelez `submit`, vous recevez un objet `Future` disponible pour enquêter sur l'état de la tâche.

La première méthode `submit` renvoie un `Future<?>` à l'aspect étrange. Vous pouvez utiliser cet objet pour appeler `isDone`, `cancel` ou `isCancelled`. Or la méthode `get` renvoie simplement `null` à la fin.

La deuxième version de `submit` envoie aussi un `Runnable`, et la méthode `get` de `Future` renvoie l'objet `result` donné à la fin de l'opération.

La troisième version envoie un `Callable`. Le `Future` renvoyé obtient le résultat du calcul lorsqu'il est prêt.

Lorsque vous avez terminé avec un pool de connexion, appelez `shutdown`. Cette méthode lance la séquence de fermeture du pool. Un executor fermé n'accepte plus les nouvelles tâches. Lorsque toutes les tâches sont terminées, les threads du pool meurent. Vous pouvez aussi appeler `shutdownNow`. Le pool annule alors toutes les tâches qui n'ont pas encore commencé et tente d'interrompre les threads en cours d'exécution.

Voici, en bref, ce qu'il faut faire pour utiliser un pool de connexion :

1. Appelez la méthode statique `newCachedThreadPool` ou `newFixedThreadPool` de la classe `Executors`.
2. Appelez `submit` pour envoyer des objets `Runnable` ou `Callable`.
3. Pour pouvoir annuler une tâche ou si vous envoyez des objets `Callable`, restez sur les objets `Future` renvoyés.
4. Appelez `shutdown` lorsque vous ne voulez plus envoyer de tâches.

L'exemple précédent produisait un grand nombre de threads courts, un par répertoire. Le programme de l'Exemple 1.8 utilise un pool de threads pour lancer les tâches.

Le programme affiche la taille de pool la plus grande pendant l'exécution. Ces informations ne sont pas disponibles dans l'interface `ExecutorService`. Pour cette raison, nous avons dû transtyper l'objet `pool` sur la classe `ThreadPoolExecutor`.

Exemple 1.8 : `ThreadPoolTest.java`

```
import java.io.*;
import java.util.*;
import java.util.concurrent.*;

public class ThreadPoolTest
{
    public static void main(String[] args) throws Exception
    {
```

```

Scanner in = new Scanner(System.in);
System.out.print("Entrez le répertoire de base
                  par ex. /usr/local/jdk5.0/src): ");
String directory = in.nextLine();
System.out.print("Entrez le mot clé (par ex. volatile) : ");
String keyword = in.nextLine();

ExecutorService pool = Executors.newCachedThreadPool();

MatchCounter counter = new MatchCounter(
    new File(directory), keyword, pool);
Future<Integer> result = pool.submit(counter);

try
{
    System.out.println(result.get() + " fichiers concordants.");
}
catch (ExecutionException e)
{
    e.printStackTrace();
}
catch (InterruptedException e) {}
pool.shutdown();

int largestPoolSize = ((ThreadPoolExecutor)
    pool).getLargestPoolSize();
System.out.println("taille de pool la plus grande =" +
    largestPoolSize);
}
}

/**
 * Cette tâche compte les fichiers d'un répertoire et de ses
 * sous-répertoires qui contiennent un mot clé donné.
 */
class MatchCounter implements Callable<Integer>
{
    /**
     * Construit un MatchCounter.
     * @param directory Le répertoire dans lequel commencer la recherche
     * @param keyword Le mot clé à rechercher
     * @param pool Le pool de threads pour envoyer les sous-tâches
     */
    public MatchCounter(File directory, String keyword,
        ExecutorService pool)
    {
        this.directory = directory;
        this.keyword = keyword;
        this.pool = pool;
    }

    public Integer call()
    {
        count = 0;
        try
        {
            File[] files = directory.listFiles();
            ArrayList<Future<Integer>> results =
                new ArrayList<Future<Integer>>();

```



```

        for (File file : files)
            if (file.isDirectory())
            {
                MatchCounter counter = new MatchCounter(
                    file, keyword, pool);
                Future<Integer> result = pool.submit(counter);
                results.add(result);
            }
            else
            {
                if (search(file)) count++;
            }

        for (Future<Integer> result : results)
            try
            {
                count += result.get();
            }
            catch (ExecutionException e)
            {
                e.printStackTrace();
            }
    }
    catch (InterruptedException e) {}
    return count;
}

/**
 * Recherche un mot clé donné dans un fichier.
 * @param file Le fichier à parcourir
 * @return true si le mot clé figure dans le fichier
 */
public boolean search(File file)
{
    try
    {
        Scanner in = new Scanner(new FileInputStream(file));
        boolean found = false;
        while (!found && in.hasNextLine())
        {
            String line = in.nextLine();
            if (line.contains(keyword)) found = true;
        }
        in.close();
        return found;
    }
    catch (IOException e)
    {
        return false;
    }
}

private File directory;
private String keyword;
private ExecutorService pool;
private int count;
}

```

java.util.concurrent.Executors 5.0

- `ExecutorService newCachedThreadPool()`
Renvoie un pool de threads en cache qui crée des threads selon les besoins et met fin aux threads inactifs depuis 60 secondes.
- `ExecutorService newFixedThreadPool(int threads)`
Renvoie un pool de threads qui utilise le nombre donné de threads pour exécuter les tâches.
- `ExecutorService newSingleThreadExecutor()`
Renvoie un Executor qui exécute les tâches de manière séquentielle dans un seul thread.

java.util.concurrent.ExecutorService 5.0

- `Future<T> submit(Callable<T> task)`
- `Future<T> submit(Runnable task, T result)`
- `Future<?> submit(Runnable task)`
Envoient la tâche donnée pour exécution.
- `void shutdown()`
Arrête le service, en terminant les tâches déjà envoyées mais sans accepter les nouveaux envois.

java.util.concurrent.ThreadPoolExecutor 5.0

- `int getLargestPoolSize()`
Renvoie la plus grande taille du pool de threads pendant la vie de cet Executor.

Exécution programmée

L'interface `ScheduledExecutorService` possède des méthodes pour l'exécution programmée ou répétée des tâches. Il s'agit d'une généralisation de `java.util.Timer` permettant le pool de threads. Les méthodes `newScheduledThreadPool` et `newSingleThreadScheduledExecutor` de la classe `Executors` renvoient des objets qui implémentent l'interface `ScheduledExecutorService`.

Vous pouvez programmer un `Runnable` ou un `Callable` pour qu'ils ne s'exécutent qu'une fois, après un délai initial. Vous pouvez aussi programmer un `Runnable` pour qu'il s'exécute à intervalles réguliers. Voir les notes d'API pour plus de détails.

java.util.concurrent.Executors 5.0

- `ScheduledExecutorService newScheduledThreadPool(int threads)`
Renvoie un pool de threads qui utilise le nombre donné de threads pour programmer des tâches.
- `ScheduledExecutorService newSingleThreadScheduledExecutor()`
Renvoie un Executor qui programme des tâches sur un seul thread.

java.util.concurrent.ScheduledExecutorService 5.0

- `ScheduledFuture<V> schedule(Callable<V> task, long time, TimeUnit unit)`
- `ScheduledFuture<?> schedule(Runnable task, long time, TimeUnit unit)`
Programme la tâche donnée après expiration du délai donné.

68 Au cœur de Java 2 - Fonctions avancées

- `ScheduledFuture<?> scheduleAtFixedRate(Runnable task, long initialDelay, long period, TimeUnit unit)`
Programme la tâche donnée pour une exécution périodique, à chaque unité `period`, une fois le délai initial expiré.
- `ScheduledFuture<?> scheduleWithFixedDelay(Runnable task, long initialDelay, long delay, TimeUnit unit)`
Programme la tâche donnée pour une exécution périodique, avec des unités `delay` entre chaque réalisation d'un appel et le début du suivant, une fois le délai initial expiré.

Synchronizers

Le paquetage `java.util.concurrent` contient plusieurs classes qui permettent de gérer un jeu de threads de collaboration (voir Tableau 1.3). Ces mécanismes possèdent des fonctionnalités intégrées pour les patterns communs de rencontre des threads. Si vous disposez d'un jeu de threads de collaboration qui suit l'un de ces motifs de comportement, réutilisez simplement la classe de bibliothèque appropriée au lieu de concocter une collection artisanale de verrous.

Tableau 1.3 : Synchronizers

Classe	Action	Quand l'utiliser
<code>CyclicBarrier</code>	Permet à un jeu de threads de patienter jusqu'à ce qu'un nombre prédéfini d'entre eux ait atteint une barrière commune, puis exécute, en option, une action de barrière.	Lorsque plusieurs threads doivent se terminer avant que leurs résultats ne puissent être utilisés.
<code>CountDownLatch</code>	Permet à un jeu de threads de patienter jusqu'à ce qu'un compteur ait été ramené à 0.	Lorsqu'un ou plusieurs threads doivent patienter jusqu'à ce qu'un nombre spécifié de résultats soit disponible.
<code>Exchanger</code>	Permet à deux threads d'échanger des objets lorsque les deux sont prêts pour l'échange.	Lorsque deux threads agissent sur deux instances de la même structure de données, l'un en remplissant une instance, l'autre en vidant l'autre instance.
<code>SynchronousQueue</code>	Permet à un thread de donner un objet à un autre thread.	Pour envoyer un objet d'un thread à un autre lorsque les deux sont prêts, sans synchronisation explicite.
<code>Semaphore</code>	Permet à un jeu de threads de patienter jusqu'à ce que les autorisations de poursuivre soient disponibles.	Pour limiter le nombre total de threads ayant accès à une ressource. Si le compte des autorisations est à un, à utiliser pour bloquer les threads jusqu'à ce qu'un autre thread donne une autorisation.

Barrières

La classe `CyclicBarrier` implémente ce que l'on appelle une *barrière*. Imaginons que plusieurs threads travaillent sur certaines parties d'un calcul. Lorsque toutes les parties sont prêtes, il faut combiner les résultats. Ainsi, lorsqu'un thread a terminé avec une partie, nous le laissons s'exécuter contre la barrière. Lorsque tous les threads ont atteint la barrière, celle-ci cède et les threads peuvent poursuivre.

Pour cela, vous construisez d'abord une barrière, en indiquant le nombre de threads participants :

```
CyclicBarrier barrier = new CyclicBarrier(nthreads);
```

Chaque thread effectue un certain travail et appelle `await` sur la barrière lorsqu'il a terminé :

```
public void run()
{
    doWork();
    barrier.await();
    ...
}
```

La méthode `await` prend un paramètre optionnel de temporisation :

```
barrier.await(100, TimeUnit.MILLISECONDS);
```

Si l'un des threads attendant la barrière part, celle-ci se *rompt* (un thread peut partir s'il appelle `await` avec une temporisation ou parce qu'il a été interrompu). Dans ce cas, la méthode `await` de tous les autres threads déclenche une `BrokenBarrierException`. Les threads qui attendent déjà voient leur méthode `await` se terminer immédiatement.

Vous pouvez fournir une *action de barrière* optionnelle, qui s'exécutera lorsque tous les threads ont atteint la barrière :

```
Runnable barrierAction = ...;
CyclicBarrier barrier = new CyclicBarrier(nthreads, barrierAction);
```

L'action peut récupérer le résultat de chaque thread.

La barrière est dite *cyclique* car elle peut être réutilisée après libération de tous les threads en attente.

Verrous Countdown

Un `CountdownLatch` permet à un jeu de threads de patienter jusqu'à ce qu'un compteur ait atteint zéro. Il diffère d'une barrière à plusieurs égards :

- Tous les threads n'ont pas besoin d'attendre que le verrou puisse s'ouvrir.
- Le verrou peut faire l'objet d'un compte à rebours de la part d'événements externes.
- Le `CountdownLatch` ne sert qu'une fois. Lorsque le compteur a atteint zéro, vous ne pouvez pas le réutiliser.

Il existe un cas spécial, où le compteur du verrou est à 1. Ceci implémente une *porte* d'utilisation unique. Les threads sont maintenus à la porte jusqu'à ce qu'un autre thread définisse le compte sur 0.

Imaginons, par exemple, un jeu de threads qui ait besoin de données initiales pour effectuer son travail. Les threads travailleurs démarrent, puis patientent à la porte. Un autre thread prépare les données. Lorsqu'il est prêt, il appelle `countDown` et tous les threads travailleurs poursuivent.

Exchanger

Exchanger est utilisé lorsque deux threads travaillent sur deux instances du même tampon de données. Généralement, un thread remplit le tampon, tandis que l'autre consomme son contenu. Lorsque les deux ont terminé, ils échangent leurs tampons.

Queues synchrones

Une queue synchrone est un mécanisme qui apparie un thread producteur et un thread consommateur. Lorsqu'un thread appelle `put` sur un `SynchronousQueue`, il bloque jusqu'à ce qu'un autre thread appelle `take`, et *vice versa*. A la différence d'Exchanger, les données ne sont transférées que dans une direction, du producteur au consommateur.

Même si la classe `SynchronousQueue` implémente l'interface `BlockingQueue`, ce n'est pas vraiment une queue. Elle ne contient aucun élément. Sa méthode `size` renvoie toujours 0.

Sémaphores

Un sémaphore est un élément qui gère plusieurs *autorisations*. Pour passer au-delà du sémaphore, un thread demande une autorisation en appelant `acquire`. Seul un nombre fixe d'autorisations est disponible, ce qui limite le nombre de threads autorisés à passer. D'autres threads peuvent émettre des autorisations en appelant `release`. Mais il ne s'agit pas de vrais objets d'autorisation. Le sémaphore conserve simplement un compteur. De plus, une autorisation n'a pas à être libérée par le thread qui l'acquiert. En fait, n'importe quel thread peut émettre n'importe quel nombre d'autorisations. S'il en émet plus que le maximum disponible, le sémaphore est simplement défini sur le compte maximal. Ils sont donc flexibles, mais peuvent paraître confus.

Les sémaphores ont été inventés par Edsger Dijkstra en 1968 pour être utilisés sous la forme de *primitives de synchronisation*. M. Dijkstra a montré que les sémaphores pouvaient être efficacement implémentés et qu'ils sont suffisamment performants pour résoudre les problèmes communs de synchronisation des threads. Dans presque tous les systèmes d'exploitation, vous trouverez des implémentations de queues bornées utilisant des sémaphores. Bien entendu, les programmeurs d'applications n'ont pas à réinventer les queues bornées. Nous vous suggérons de n'utiliser les sémaphores que lorsque leur comportement concorde bien avec votre problème de synchronisation.

Par exemple, un sémaphore ayant un nombre d'autorisations égal à 1 servira de porte qu'un autre thread pourra ouvrir et fermer. Imaginons un programme qui effectue un travail, puis attend qu'un utilisateur étudie le résultat et appuie sur un bouton pour continuer. Le programme passe alors à la partie suivante du travail. Le thread travailleur appelle `acquire` dès qu'il est prêt à faire une pause. Le thread de la GUI appelle `release` dès que l'utilisateur clique sur le bouton "Continuer".

Que se passe-t-il alors si l'utilisateur clique plusieurs fois sur le bouton lorsque le thread travailleur est prêt ? Une seule autorisation étant disponible, le compteur reste à 1.

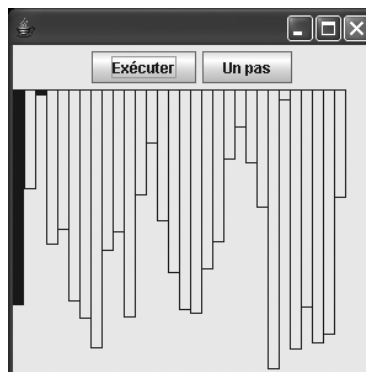
Le programme de l'Exemple 1.9 concrétise cette idée. Il anime un algorithme de tri. Un thread travailleur trie un tableau, il s'arrête à intervalles réguliers et attend que l'utilisateur donne l'autorisation de poursuivre. L'utilisateur peut admirer un dessin de l'état actuel de l'algorithme et appuyer sur le bouton "Continuer" pour permettre au thread travailleur de réaliser l'étape suivante.

Nous n'avons pas voulu vous ennuyer avec le code d'un algorithme de tri, nous appelons donc simplement `Arrays.sort`. Pour faire une pause dans l'algorithme, nous fournissons un objet

Comparator qui attend le sémaphore. L'animation fait donc une pause dès que l'algorithme compare deux éléments. Nous dessinons les valeurs actuelles du tableau et surlignons les éléments comparés (voir Figure 1.7).

Figure 1.7

Animation d'un algorithme de tri.



Exemple 1.9 : AlgorithmAnimation.java

```
import java.awt.*;
import java.awt.geom.*;
import java.awt.event.*;
import java.util.*;
import java.util.concurrent.*;
import javax.swing.*;

/**
 * Ce programme anime un algorithme de tri.
 */
public class AlgorithmAnimation
{
    public static void main(String[] args)
    {
        JFrame frame = new AnimationFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce cadre montre le tableau en cours de tri, ainsi que les
 * boutons pour avancer d'un pas dans l'animation
 * ou l'exécuter sans interruption.
 */
class AnimationFrame extends JFrame
{
    public AnimationFrame()
    {
        ArrayPanel panel = new ArrayPanel();
        add(panel, BorderLayout.CENTER);

        Double[] values = new Double[VALUES_LENGTH];
        final Sorter sorter = new Sorter(values, panel);
    }
}
```

72 Au cœur de Java 2 - Fonctions avancées

```

        JButton runButton = new JButton("Exécuter");
        runButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    sorter.setRun();
                }
            });

        JButton stepButton = new JButton("Un pas");
        stepButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    sorter.setStep();
                }
            });

        JPanel buttons = new JPanel();
        buttons.add(runButton);
        buttons.add(stepButton);
        add(buttons, BorderLayout.NORTH);
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        for (int i = 0; i < values.length; i++)
            values[i] = new Double(Math.random());

        Thread t = new Thread(sorter);
        t.start();
    }

    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 300;
    private static final int VALUES_LENGTH = 30;
}

/**
 * Cet exécutable exécute un algorithme de tri.
 * Lorsque deux éléments sont comparés, l'algorithme
 * fait une pause et actualise l'écran.
 */
class Sorter implements Runnable
{
    /**
     * Construit un Sorter.
     * @param values Le tableau à trier
     * @param panel L'écran sur lequel afficher la progression du tri
     */
    public Sorter(Double[] values, ArrayPanel panel)
    {
        this.values = values;
        this.panel = panel;
        this.gate = new Semaphore(1);
        this.run = false;
    }
}

```

```

    /**
     Règle le trieur sur le mode "exécuter".
    */
    public void setRun()
    {
        run = true;
        gate.release();
    }

    /**
     Règle le trieur sur le mode "un pas".
    */
    public void setStep()
    {
        run = false;
        gate.release();
    }

    public void run()
    {
        Comparator<Double> comp = new
            Comparator<Double>()
            {
                public int compare(Double i1, Double i2)
                {
                    panel.setValues(values, i1, i2);
                    try
                    {
                        if (run)
                            Thread.sleep(DELAY);
                        else
                            gate.acquire();
                    }
                    catch (InterruptedException exception)
                    {
                        Thread.currentThread().interrupt();
                    }
                    return i1.compareTo(i2);
                }
            };
        Arrays.sort(values, comp);
        panel.setValues(values, null, null);
    }

    private Double[] values;
    private ArrayPanel panel;
    private Semaphore gate;
    private static final int DELAY = 100;
    private boolean run;
}

/**
 Cet écran dessine un tableau et marque deux éléments dans
 le tableau.
 */
class ArrayPanel extends JPanel
{

```


74 Au cœur de Java 2 - Fonctions avancées

```

public void paintComponent(Graphics g)
{
    if (values == null) return;
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    int width = getWidth() / values.length;
    for (int i = 0; i < values.length; i++)
    {
        double height = values[i] * getHeight();
        Rectangle2D bar = new Rectangle2D.Double(width * i, 0,
            width, height);
        if (values[i] == marked1 || values[i] == marked2)
            g2.fill(bar);
        else
            g2.draw(bar);
    }
}

/**
 * Définit les valeurs à dessiner.
 * @param values Le tableau de valeurs à afficher
 * @param marked1 le premier élément marqué
 * @param marked2 le deuxième élément marqué
 */
public void setValues(Double[] values, Double marked1, Double marked2)
{
    this.values = values;
    this.marked1 = marked1;
    this.marked2 = marked2;
    repaint();
}

private Double marked1;
private Double marked2;
private Double[] values;
}

```

java.util.concurrent.CyclicBarrier 5.0

- `CyclicBarrier(int parties)`
- `CyclicBarrier(int parties, Runnable barrierAction)`
 Construisent une barrière cyclique pour le nombre de parties donné. `barrierAction` est exécuté lorsque toutes les parties ont appelé `await` sur la barrière.
- `int await()`
- `int await(long time, TimeUnit unit)`
 Attendent que toutes les parties aient appelé `await` sur la barrière ou jusqu'à la fin de la temporisation, auquel cas une `TimeoutException` est déclenchée. En cas de réussite, renvoient l'indice d'arrivée de cette partie. La première partie possède `parties - 1` indices, la dernière partie possède un indice 0.

java.util.concurrent.CountDownLatch 5.0

- `CountDownLatch(int count)`
 Construit un verrou avec un compte à rebours donné.

- `void await()`
Attend que le compteur de ce verrou ait atteint 0.
- `boolean await(long time, TimeUnit unit)`
Attend que le compteur de ce verrou ait atteint 0 ou que la temporisation ait expiré. Renvoie `true` si le compteur vaut 0, `false` si la temporisation a expiré.
- `public void countDown()`
Décompte du compte à rebours de ce verrou.

`java.util.concurrent.Exchanger<V> 5.0`

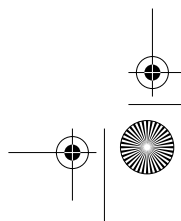
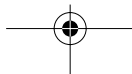
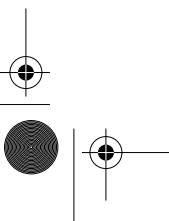
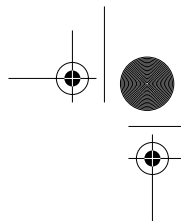
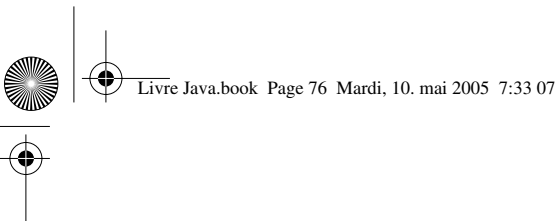
- `V exchange(V item)`
- `V exchange(V item, long time, TimeUnit unit)`
Bloquent jusqu'à ce qu'un autre thread appelle cette méthode, puis échangent l'élément avec l'autre thread et renvoient l'élément de l'autre thread. La deuxième méthode déclenche une `TimeoutException` après expiration de la temporisation.

`java.util.concurrent.SynchronousQueue<V> 5.0`

- `SynchronousQueue()`
- `SynchronousQueue(boolean fair)`
Construisent une queue synchrone qui permet aux threads de donner des éléments. Si `fair` vaut `true`, la queue favorise les threads attendant depuis le plus longtemps.
- `void put(V item)`
Bloquent jusqu'à ce qu'un autre thread appelle `take` pour prendre cet élément.
- `V take()`
Bloquent jusqu'à ce qu'un autre thread appelle `put`. Renvoie l'élément fourni par l'autre thread.

`java.util.concurrent.Semaphore 5.0`

- `Semaphore(int permits)`
- `Semaphore(int permits, boolean fair)`
Construisent un sémaphore avec le nombre maximal donné d'autorisations. Si `fair` vaut `true`, la queue favorise les threads attendant depuis le plus longtemps.
- `void acquire()`
Attend d'acquérir une autorisation.
- `boolean tryAcquire()`
Tente d'acquérir une autorisation ; renvoie `false` si aucune n'est disponible.
- `boolean tryAcquire(long time, TimeUnit unit)`
Tente d'acquérir une autorisation avec le délai donné ; renvoie `false` si aucune n'est disponible.
- `void release()`
Libère une autorisation.



2

Collections

Au sommaire de ce chapitre

- ✓ Les interfaces de collection
- ✓ Les collections concrètes
- ✓ Le cadre des collections
- ✓ Les algorithmes
- ✓ Les anciennes collections

La programmation orientée objet encapsule les données dans des classes, mais l'organisation de vos données dans ces classes reste tout aussi importante que pour d'autres langages de programmation traditionnels. Naturellement, le choix d'une structure de données dépend du problème que vous essayez de résoudre. Votre classe a-t-elle besoin de rechercher un élément parmi un millier (voire un million) d'entre eux ? Ses éléments doivent-ils être triés et votre classe doit-elle pouvoir insérer et supprimer des éléments en plein milieu de l'ensemble des données ? A-t-elle besoin d'une structure en tableau dans laquelle elle doit pouvoir accéder à n'importe quelle donnée parmi un ensemble de données de plus en plus important ? La structure de données que vous choisirez pour vos classes peut engendrer une grande différence, à la fois au moment de l'implémentation des méthodes et en termes de performances.

Ce chapitre met en évidence la façon dont la technologie Java peut vous aider à trouver une structure de données adaptée à une programmation sérieuse. Dans les écoles d'informatique, il existe un cours appelé *Structures de données* qui requiert en général un trimestre complet, et en conséquence un grand nombre de livres sont consacrés à ce sujet très important. Notre but dans ce chapitre n'est pas de parcourir la liste complète des structures de données existantes. Nous préférons examiner en détail les quelques structures fournies avec la bibliothèque standard de Java. Nous espérons ainsi qu'après avoir parcouru ce chapitre, il vous sera plus facile d'adapter n'importe quelle structure de données à la programmation en Java.

Les interfaces de collection

Avant la parution du JDK 1.2, la bibliothèque standard ne fournissait qu'un petit ensemble de classes pour les structures de données les plus utiles : `Vector`, `Stack`, `Hashtable`, `Bitset`, ainsi que l'interface `Enumeration` qui fournit un mécanisme abstrait pour répertorier des éléments dans un conteneur quelconque. Cela partait d'une bonne intention, car la compréhension d'une bibliothèque complète de classes de collections nécessite à la fois beaucoup de temps et de connaissances.

Avec l'arrivée du JDK 1.2, les concepteurs ont senti qu'il était temps de fournir un ensemble complet de structures de données. Ils ont notamment dû faire face à un certain nombre de conflits dans l'élaboration d'une telle bibliothèque. Ils voulaient en effet que cette bibliothèque reste petite et simple à comprendre. Ils rejetaient la complexité de la bibliothèque standard de modèles (ou *STL*, *Standard Template Library*) du C++, mais ils voulaient bénéficier de l'aspect générique des algorithmes mis en place par la STL. Ils voulaient également que toutes les anciennes classes fassent partie du nouvel ensemble de classes. Comme tous les concepteurs de bibliothèques, ils ont donc dû faire quelques choix difficiles et prendre des décisions de conception qui leur sont propres. Dans cette partie, nous explorerons le cadre des collections Java, nous vous montrerons comment l'exploiter et nous aborderons la logique cachée derrière les caractéristiques les plus controversées.

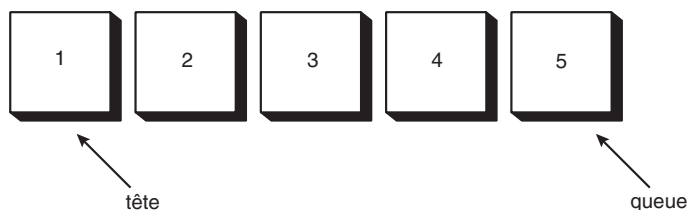
Séparer les interfaces d'une collection et leur implémentation

Comme pour la plupart des bibliothèques de structures de données récentes, la bibliothèque de collections Java fait la distinction entre les *interfaces* et les *implémentations*. Intéressons-nous à cette distinction avec une structure de données familière, la *queue de données*.

Une *interface de queue* indique que vous pouvez ajouter des éléments à la fin d'une queue, supprimer des éléments au début d'une queue, et déterminer le nombre d'éléments contenus dans une queue. Les queues sont utilisées si vous devez enregistrer des objets et les récupérer selon la règle "premier entré, premier sorti" (voir Figure 2.1).

Figure 2.1

Une queue.



Une forme minimaliste d'interface pour les queues pourrait ressembler à ceci :

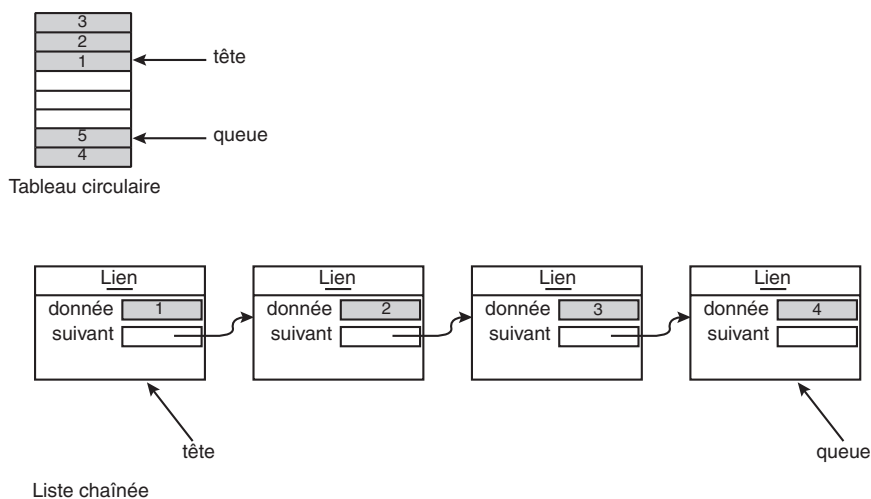
```
interface Queue<E> // Une forme simplifiée d'interface de la
                    // bibliothèque standard
{
    void add(E element);
    E remove();
    int size();
}
```

L'interface n'indique en aucune manière la façon dont la queue est implémentée. Il existe deux implémentations courantes des queues. La première se sert d'un tableau circulaire et la seconde, d'une liste chaînée (voir Figure 2.2).

INFO

Avec le JDK 5.0, les classes de collection sont devenues des classes génériques avec paramètres de type. Si vous utilisez une version antérieure de Java, vous devez ignorer les paramètres de type et remplacer les types génériques par le type `Object`. Pour en savoir plus sur les classes génériques, consultez le Volume 1, Chapitre 13.

Figure 2.2
Les implémentations
d'une queue.



Chacune de ces deux implémentations peut passer par une classe implémentant l'interface avec la queue :

```
class CircularArrayQueue<E> implements Queue<E> // pas une vraie classe
                                                // de bibliothèque
{
    CircularArrayQueue(int capacity) { . . . }
    public void add(E element) { . . . }
    public E remove() { . . . }
    public int size() { . . . }

    private E[] elements;
    private int head;
    private int tail;
}

class LinkedListQueue<E> implements Queue<E> // pas une vraie classe
                                                // de bibliothèque
{
    LinkedListQueue() { . . . }
    public void add(E element) { . . . }
```

80 *Au cœur de Java 2 - Fonctions avancées*

```

public E remove() { . . . }
public int size() { . . . }

private Link head;
private Link tail;
}

```

INFO

La bibliothèque Java ne possède pas de classes nommées `CircularArrayQueue` et `LinkedListQueue`. Nous les utilisons comme exemples pour expliquer la différence entre les interfaces de collection et les implémentations. Si vous avez besoin d'une queue de tableau circulaire, vous pouvez utiliser la classe `ArrayBlockingQueue` décrite au Chapitre 1 ou l'implémentation décrite plus loin. Pour une queue de liste chaînée, utilisez simplement la classe `LinkedList` : elle implémente l'interface `Queue`.

Lorsque vous vous servez d'une queue dans vos programmes, vous n'avez pas besoin de connaître l'implémentation réellement utilisée une fois que la collection est construite. Par conséquent, il est logique d'avoir recours à une classe existante *uniquement* lorsque vous construisez l'objet collection. Le *type d'interface* permet de faire référence à la collection.

```

Queue<Customer> expressLane = new CircularArrayQueue<Customer>(100);
expressLane.add(new Customer("Sandrine"));

```

Avec cette approche, si vous changez de point de vue, vous pouvez facilement utiliser une implémentation différente. Vous n'avez besoin de modifier votre programme qu'à un seul endroit : le constructeur. Si vous décidez qu'après tout, une `LinkedListQueue` constitue un meilleur choix, votre programme devient :

```

Queue<Customer> expressLane = new LinkedListQueue<Customer>();
expressLane.add(new Customer("Sandrine"));

```

Pourquoi choisir une implémentation plutôt qu'une autre ? Une interface ne fournit aucun détail quant à l'efficacité de son implémentation. De manière générale, un tableau circulaire est plus efficace qu'une liste chaînée, et il est donc préférable à cette dernière. Toutefois, comme d'habitude, il y a un prix à payer. Les tableaux circulaires font partie d'une collection *bornée*, qui a une capacité déterminée. Si vous ne connaissez pas la limite maximale du nombre d'objets stockés par votre programme, vous préférerez probablement une implémentation en liste chaînée.

En étudiant la documentation API, vous découvrirez un autre jeu de classes dont le nom commence par `Abstract`, comme `AbstractQueue`. Ces classes sont destinées aux implémentations de bibliothèque. Pour implémenter votre propre classe de queue, il sera plus facile d'étendre `AbstractQueue` que d'implémenter toutes les méthodes de l'interface `Queue`.

Interfaces de collection et d'itération dans la bibliothèque Java

L'interface fondamentale des classes de collections de la bibliothèque Java est l'interface `Collection`. Cette interface possède deux méthodes essentielles :

```

public interface Collection<E>
{
    boolean add(E element);
}

```

```

        Iterator<E> iterator();
        ...
    }

```

Il existe plusieurs autres méthodes, que nous aborderons plus loin.

La méthode `add` ajoute un élément dans la collection. La méthode `add` renvoie `true` si l'ajout de l'élément a effectivement modifié la collection. Elle renvoie `false` si la collection n'a pas été modifiée. Par exemple, si vous essayez d'ajouter un objet dans un ensemble et que cet objet en fasse déjà partie, la requête `add` est sans effet parce que les ensembles de données ne peuvent pas accepter deux fois la même donnée.

La méthode `iterator` renvoie un objet qui implémente l'interface `Iterator`. Un objet d'itération peut servir à parcourir les éléments d'une collection, un par un.

Itérateurs

L'interface `Iterator` possède trois méthodes :

```

public interface Iterator<E>
{
    E next();
    boolean hasNext();
    void remove();
}

```

En appelant plusieurs fois la méthode `next`, vous pouvez parcourir tous les éléments de la collection un par un. Lorsque la fin de la collection est atteinte, la méthode `next` déclenche une exception `NoSuchElementException`. Par conséquent, il convient d'appeler la méthode `hasNext` avant la méthode `next`. Cette méthode renvoie `true` si l'objet de l'itération possède encore au moins un élément. Si vous souhaitez parcourir tous les éléments d'un conteneur, il suffit de demander un objet `iterator` et d'appeler la méthode `next` tant que la méthode `hasNext` renvoie `true`. Par exemple :

```

Collection<String> c = ...;
Iterator<String> iter = c.iterator();
while (iter.hasNext())
{
    String element = iter.next();
    utilisation de element
}

```

Le JDK 5.0 a introduit un raccourci élégant pour cette boucle. Elle est bien plus concise avec la boucle "for each".

```

for (String element : c)
{
    utilisation de element
}

```

Le compilateur se contente de traduire la boucle "for each" en boucle avec un itérateur.

La boucle "for each" fonctionne comme tout objet qui implémente l'interface `Iterable`, une interface avec une seule méthode :

```

public interface Iterable<E>
{
    Iterator<E> iterator();
}

```


82 Au cœur de Java 2 - Fonctions avancées

L'interface `Collection` étend l'interface `Iterable`. Vous pouvez donc utiliser la boucle "for each" avec n'importe quelle collection de la bibliothèque standard.

L'ordre de visite des éléments dépend du type de la collection. Si vous parcourez un `ArrayList`, l'itérateur démarre à l'indice 0 et l'incrémente à chaque pas. Toutefois, si vous visitez les éléments dans un `HashSet`, vous les rencontrerez dans un ordre aléatoire. Vous pouvez être sûr de rencontrer tous les éléments d'une collection lors de l'itération, mais vous ne pouvez pas savoir dans quel ordre. Cela ne pose généralement pas problème car l'ordre importe peu pour des calculs comme les totaux ou les concordances.

INFO

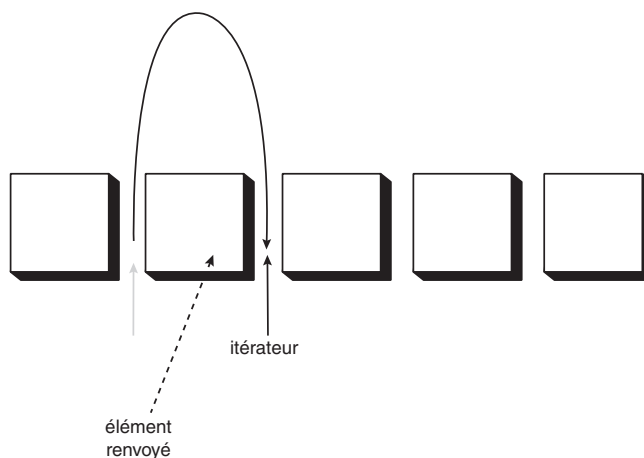
Les habitués de Java reconnaîtront que les méthodes `next` et `hasNext` de l'interface `Iterator` fonctionnent de la même manière que les méthodes `nextElement` et `hasMoreElements` d'une `Enumeration`. Les concepteurs de la bibliothèque de collections Java auraient pu choisir d'étendre l'interface `Enumeration`, mais ils n'aimaient pas les noms de méthode trop longs et ont choisi par conséquent d'introduire une nouvelle interface avec des noms plus courts.

Il existe une différence importante au niveau conceptuel entre les itérateurs de la bibliothèque de collections Java et les itérateurs des autres bibliothèques. Dans les bibliothèques de collections classiques (comme la STL du C++), les itérateurs correspondent plutôt à des indices de tableau. A partir d'un tel opérateur, il est possible de retrouver un élément qui est enregistré à la position correspondant à cet itérateur, de la même manière que vous accédez à un élément de tableau `a[i]` si vous possédez l'indice de tableau `i`. Indépendamment de la recherche, l'itérateur peut être avancé à la prochaine position. Ceci revient à incrémenter un indice de tableau en appelant `i++` sans effectuer de recherche dans le tableau. Mais les itérateurs Java ne fonctionnent pas de cette manière. Les recherches et les changements de position sont réellement couplés, et la seule façon de rechercher un élément consiste à appeler la méthode `next`, ce qui avance la position de l'itérateur.

Il vaut donc mieux considérer que les itérateurs Java se trouvent *entre les éléments*. Lorsque `next` est appelée, l'itérateur saute au-dessus de l'élément suivant et il renvoie une référence à l'élément qu'il vient de sauter (voir Figure 2.3).

Figure 2.3

La progression d'un itérateur.



INFO

Voici une autre analogie pratique. Vous pouvez considérer que `Iterator.next` est l'équivalent de `InputStream.read`. La lecture d'un octet à partir d'un flux de données "consomme" automatiquement cet octet. Le prochain appel à `read` consomme et renvoie l'octet suivant du flux de données. De la même manière, des appels répétés à `next` vous permettent de parcourir tous les éléments d'une collection.

Suppression d'éléments

La méthode `remove` de l'interface `Iterator` supprime l'élément renvoyé par le dernier appel à `next`. Dans de nombreux cas, cette méthode est pratique si vous avez besoin d'examiner chaque élément avant de savoir si vous devez le supprimer ou non. Mais, si vous souhaitez effacer un élément en fonction de sa position, il faut malgré tout le dépasser. Par exemple, voici comment supprimer le premier élément d'une collection de chaînes :

```
Iterator<String> it = c.iterator();
it.next(); // sauter le premier élément
it.remove(); // on peut maintenant le supprimer
```

Plus important encore, il existe une relation entre les appels aux méthodes `next` et `remove`. Il n'est pas permis d'appeler `remove` sans avoir au préalable appelé `next`. Si vous essayez, une `IllegalStateException` sera déclenchée.

Donc, si vous voulez supprimer deux éléments successifs, vous n'avez pas le droit d'appeler

```
it.remove();
it.remove(); // Erreur !
```

Il faut appeler `next` pour passer au-dessus de l'élément à supprimer.

```
it.remove();
it.next();
it.remove(); // Ok
```

Méthodes utilitaires génériques

Comme les interfaces `Iterator` et `Collection` sont générales, il est possible d'écrire des méthodes pratiques pouvant travailler sur n'importe quel type de collection. Par exemple, voici une méthode générale qui teste si une collection arbitraire contient un élément donné :

```
public static<E> boolean contains(Collection<E> c, Object obj)
{
    for (E element : c)
        if (element.equals(obj))
            return true;
    return false;
}
```

Les concepteurs de la bibliothèque Java ont décidé que certaines de ces méthodes générales étaient tellement pratiques qu'elles devraient faire partie de la bibliothèque. De cette façon, les utilisateurs n'ont pas besoin de réinventer la roue en permanence. La méthode `contains` en constitue un bon exemple.

84 Au cœur de Java 2 - Fonctions avancées

En fait, l'interface `Collection` déclare quelques méthodes pratiques que toutes les classes implémentées doivent fournir. Parmi elles, nous trouvons :

```
int size()
boolean isEmpty()
boolean contains(Object obj)
boolean containsAll(Collection<?> c)
boolean equals(Object autre)
boolean addAll(Collection<? extends E> source)
boolean remove(Object obj)
boolean removeAll(Collection<?> c)
void clear()
boolean retainAll(Collection<?> c)
Object[] toArray()
<T> T[] toArray(T[] arrayToFill)
```

La plupart de ces méthodes sont assez simples pour se passer d'une explication. Vous en trouverez la documentation complète dans les notes API à la fin de cette section.

Mais, comme il est plutôt fastidieux d'ajouter toutes ces méthodes dans chaque classe implémentant l'interface `Collection`, la bibliothèque fournit la classe `AbstractCollection`, qui définit les méthodes fondamentales (telles `size` et `iterator`) comme des méthodes abstraites et implémente le corps de ces méthodes à leur place. Par exemple,

```
public abstract class AbstractCollection<E>
    implements Collection<E>
{
    . . .
    public abstract Iterator<E> iterator();

    public boolean contains(Object obj)
    {
        for (E element : c) // Appelle iterator()
            if (element.equals(obj))
                return true;
        return false;
    }
    . . .
}
```

Une classe de collection concrète peut maintenant étendre la classe `AbstractCollection`. C'est donc à la classe de collection concrète de fournir une méthode `iterator`, mais la méthode `contains` est gérée par la superclasse `AbstractCollection`. De plus, si la sous-classe propose une meilleure implémentation de la méthode `contains`, elle est libre de l'utiliser.

Il s'agit là d'une bonne architecture pour un ensemble de classes. Les utilisateurs des classes de collections disposent ainsi d'un ensemble de méthodes très complet disponible au travers de l'interface générale, et les programmeurs chargés des structures de données n'ont pas besoin de se préoccuper de toutes les méthodes.

java.util.Collection<E> 1.2

- `Iterator<E> iterator()`
Renvoie un itérateur pouvant être utilisé pour parcourir les éléments d'une collection.
- `int size()`
Renvoie le nombre courant d'éléments stockés dans une collection.

- `boolean isEmpty()`
Renvoie `true` si la collection ne contient aucun élément.
- `boolean contains(Object obj)`
Renvoie `true` si la collection contient un objet correspondant à `obj`.
- `boolean containsAll(Collection<?> other)`
Renvoie `true` si cette collection contient tous les éléments de l'autre collection.
- `boolean add(Object element)`
Ajoute un élément à la collection. Renvoie `true` si la collection a été modifiée par cet appel.
- `boolean addAll(Collection<? extends E> other)`
Ajoute tous les éléments de l'autre collection dans cette collection. Renvoie `true` si la collection a été modifiée par cet appel.
- `boolean remove(Object obj)`
Supprime l'objet `obj` de cette collection. Renvoie `true` si l'objet correspondant a été trouvé.
- `boolean removeAll(Collection<?> other)`
Supprime de cette collection tous les éléments de l'autre collection. Renvoie `true` si la collection a été modifiée par cet appel.
- `void clear()`
Supprime tous les éléments de la collection.
- `boolean retainAll(Collection<?> other)`
Supprime dans cette collection tous les éléments qui ne figurent pas dans l'autre collection. Renvoie `true` si la collection a été modifiée par cet appel.
- `Object[] toArray()`
Renvoie un tableau contenant tous les objets de la collection.

java.util.Iterator<E> 1.2

- `boolean hasNext()`
Renvoie `true` s'il reste un élément à parcourir.
- `E next()`
Renvoie le prochain objet à parcourir. Déclenche une exception `NoSuchElementException` si la fin de la collection est atteinte.
- `void remove()`
Supprime le dernier objet lu. Cette méthode doit impérativement être appelée après une lecture. Si la collection a été modifiée depuis la dernière lecture, cette méthode renvoie une `IllegalStateException`.

Les collections concrètes

Plutôt que de voir en détail chaque interface, nous avons pensé qu'il serait plus utile de commencer par aborder les structures de données concrètes implémentées dans la bibliothèque Java. Une fois que vous aurez correctement décrit les classes que vous pourrez utiliser, nous reviendrons à des considérations abstraites et nous verrons comment la structure des collections organise toutes ces

86 Au cœur de Java 2 - Fonctions avancées

classes. Le Tableau 2.1 présente les collections de la bibliothèque Java et décrit brièvement l'objectif de chaque classe de collection (pour des raisons de simplicité, nous ne parlerons pas des collections compatibles avec les threads, traitées au Chapitre 1). Toutes les classes du Tableau 2.1 implémentent l'interface `Collection`, à l'exception de celles dont le nom se termine par `Map`. Celles-ci implémentent plutôt l'interface `Map` (nous la traiterons un peu plus loin).

Tableau 2.1 : Collections concrètes dans la bibliothèque Java

Type de collection	Description
<code>ArrayList</code>	Une séquence indexée qui grandit et se réduit de manière dynamique
<code>LinkedList</code>	Une séquence ordonnée qui permet des insertions et des retraits effectifs à n'importe quel endroit
<code>HashSet</code>	Une collection non ordonnée qui refuse les répliques
<code>TreeSet</code>	Un ensemble trié
<code>EnumSet</code>	Un ensemble de valeurs de type énuméré
<code>LinkedHashSet</code>	Un ensemble qui se souvient de l'ordre d'insertion des éléments
<code>PriorityQueue</code>	Une collection qui permet un retrait effectif de l'élément le plus petit
<code>HashMap</code>	Une structure de données qui stocke les associations clé/valeur
<code>TreeMap</code>	Une concordance dans laquelle les clés sont triées
<code>EnumMap</code>	Une concordance dans laquelle les clés appartiennent à un type énuméré
<code>LinkedHashMap</code>	Une concordance qui se souvient de l'ordre d'ajout des entrées
<code>WeakHashMap</code>	Une concordance avec des valeurs pouvant être réclamées par le ramasse-miettes si elles ne sont pas utilisées ailleurs
<code>IdentityHashMap</code>	Une concordance avec des clés comparées par <code>==</code> , et non par <code>equals</code>

Listes chaînées

Nous avons déjà abordé les tableaux et leur cousin dynamique, la classe `ArrayList`, dans un grand nombre d'exemples dans le Volume 1. Mais ces deux structures possèdent un inconvénient majeur. La suppression d'un élément au milieu d'un tableau nécessite beaucoup de temps machine, car tous les éléments situés après l'élément supprimé doivent être décalés d'une case (voir Figure 2.4). Le même problème se pose pour insérer des éléments au milieu d'un tableau.

Il existe une autre structure de données très répandue, la *liste chaînée*, qui permet de résoudre ces problèmes. Alors que les objets d'un tableau occupent des emplacements mémoire successifs, une liste chaînée stocke chaque objet avec un *lien* qui y fait référence. Chaque lien possède également une référence vers le lien suivant de la liste. Avec Java, chaque élément d'une liste chaînée possède en fait *deux liens*, c'est-à-dire que chaque élément est aussi relié à l'élément précédent (voir Figure 2.5).

Figure 2.4
Supprimer un élément
d'un tableau.

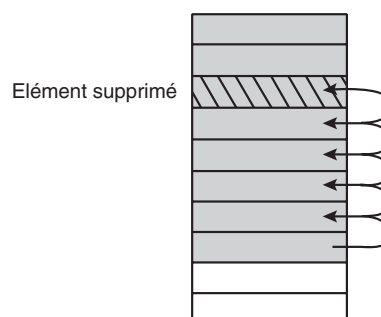
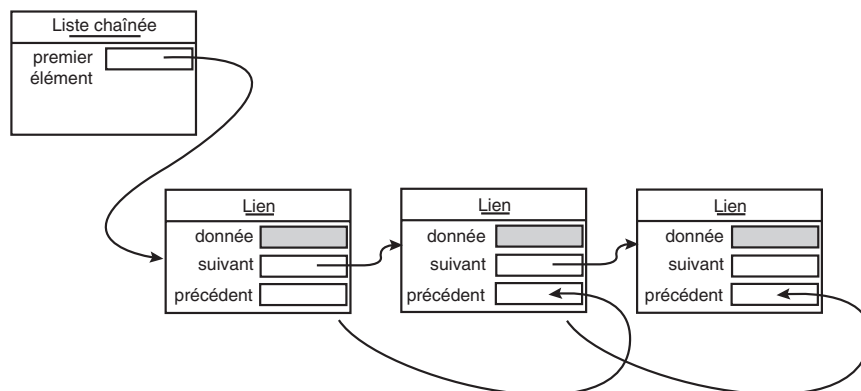
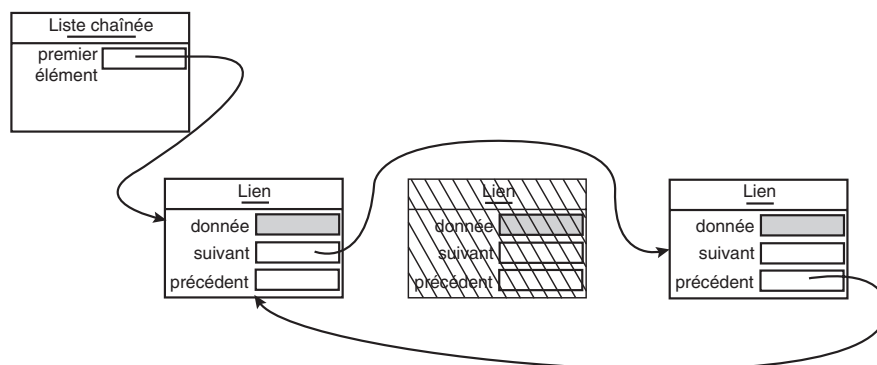


Figure 2.5
Une liste
doublement
chaînée.



La suppression d'un élément au milieu d'une liste chaînée est une opération très simple, car seuls les liens associés à l'élément supprimé doivent être modifiés (voir Figure 2.6).

Figure 2.6
Supprimer
un élément
d'une liste chaînée.



Vous avez peut-être déjà appris à implémenter des listes chaînées. Si vous avez oublié les manipulations de liens lors d'une suppression ou d'un ajout d'élément dans une liste chaînée, vous serez soulagé d'apprendre que la bibliothèque de collections Java possède une classe `LinkedList` prête à l'emploi.

88 Au cœur de Java 2 - Fonctions avancées

L'exemple de programme suivant ajoute trois éléments, puis supprime le deuxième.

```
List<String> staff = new LinkedList<String>(); // LinkedList
                                           // implémente List

staff.add("Claire");
staff.add("Laurent");
staff.add("Adrien");
Iterator iter = staff.iterator();
String first = iter.next(); // visiter le premier élément
String second = iter.next(); // visiter le deuxième élément
iter.remove(); // Supprime le dernier élément parcouru
```

Il y a une différence importante entre les listes chaînées et les collections générales. Une liste chaînée est en effet une *collection classée* dans laquelle la position des objets a une importance. La méthode `LinkedList.add` ajoute un objet à la fin de la liste. Mais vous aurez souvent besoin d'ajouter un élément quelque part au milieu d'une liste. Cette méthode `add`, qui dépend de la position courante, dépend aussi d'un itérateur, car ce sont les itérateurs qui sont chargés de stocker les positions dans une collection. L'emploi d'un itérateur pour ajouter des éléments n'a un sens que si les collections concernées sont classées selon un ordre implicite. Par exemple, le type de données *set* que nous aborderons dans la prochaine partie n'impose aucun ordre à ses éléments. C'est pourquoi il n'existe aucune méthode `add` dans l'interface `Iterator`. Pour compenser, la bibliothèque de collections fournit une sous-interface, `ListIterator`, qui contient une méthode `add` :

```
interface ListIterator<E> extends Iterator<E>
{
    void add(E element);
    . . .
}
```

Contrairement à `Collection.add`, cette méthode ne renvoie pas un boolean, c'est-à-dire que l'opération `add` est censée toujours modifier la liste.

De plus, l'interface `ListIterator` possède deux méthodes, dont vous pouvez vous servir pour parcourir une liste à l'envers.

```
E previous()
boolean hasPrevious()
```

Comme la méthode `next`, la méthode `previous` renvoie l'objet qu'elle a sauté.

La méthode `ListIterator` de la classe `LinkedList` renvoie un objet itérateur qui implémente l'interface `ListIterator`.

```
ListIterator<String> iter = staff.listIterator();
```

La méthode `add` ajoute le nouvel élément *avant* la position de l'itérateur. Par exemple, le code suivant saute le premier élément de la liste chaînée et ajoute la chaîne "Juliette" avant le deuxième élément (voir Figure 2.7) :

```
List<String> staff = new LinkedList<String>();
staff.add("Claire");
staff.add("Laurent");
```

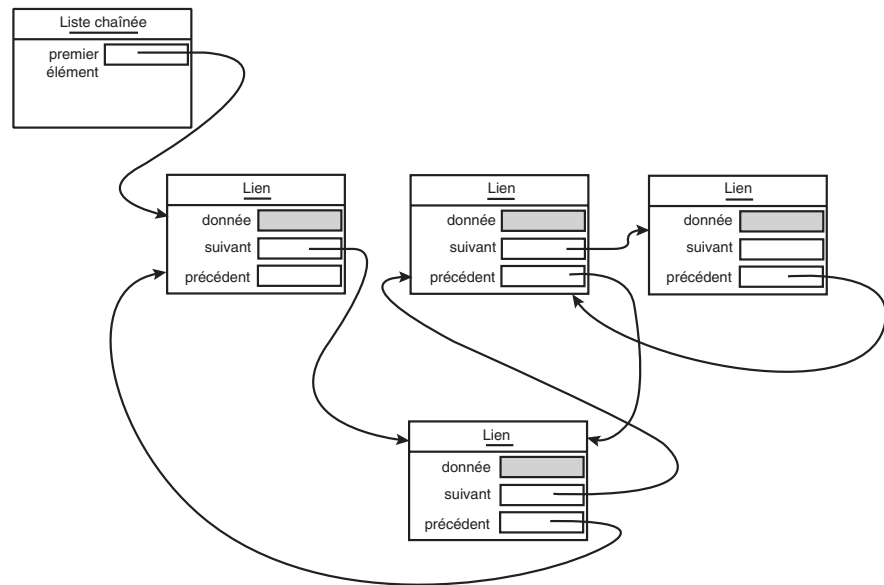
```

staff.add("Adrien");
ListIterator<String> iter = staff.listIterator();
iter.next(); // ignorer le premier élément
iter.add("Juliette");

```

Figure 2.7

Ajouter un élément dans une liste chaînée.



Si vous appelez la méthode `add` plusieurs fois, les éléments sont simplement ajoutés dans l'ordre dans lequel vous les passez. Ils sont tous ajoutés avant la position courante de l'itérateur.

Lorsque vous utilisez l'opération `add` avec un itérateur qui vient juste d'être renvoyé par la méthode `ListIterator` et qui fait référence au début de la liste chaînée, les éléments sont ajoutés au début de la liste chaînée. Lorsque l'itérateur dépasse le dernier élément de la liste (c'est-à-dire que `hasNext` renvoie `false`), l'élément est ajouté à la fin de la liste. Si la liste chaînée possède n éléments, il existe $n+1$ emplacements pour ajouter un nouvel élément. Ces emplacements correspondent aux $n+1$ positions possibles de l'itérateur. Par exemple, si une liste chaînée contient trois éléments, A, B et C, alors il existe quatre positions possibles pour insérer un nouvel élément (représentées par |) :

```

|ABC
A|BC
AB|C
ABC|

```

INFO

Faites attention à l'analogie avec un curseur de texte. L'opération `remove` ne fonctionne pas exactement comme la touche Retour-arrière. Juste après un appel à `next`, la méthode `remove` supprime en fait l'élément à gauche de l'itérateur, exactement comme la touche Retour-arrière. Cependant, si vous venez d'appeler `previous`, l'élément situé à droite de l'itérateur sera supprimé. De plus, il n'est pas possible d'appeler `remove` deux fois de suite.

Contrairement à la méthode `add`, qui ne dépend que de la position de l'itérateur, la méthode `remove` dépend de l'état de l'itérateur.

90 Au cœur de Java 2 - Fonctions avancées

De plus, il existe une méthode `set` qui remplace le dernier élément renvoyé par `next` ou `previous` par un nouvel élément. Par exemple, le code suivant remplace le premier élément d'une liste par une nouvelle valeur :

```
ListIterator<String> iter = list.listIterator();
String oldValue = iter.next(); // renvoie le premier élément
iter.set(newValue); // affecte une nouvelle valeur au premier élément
```

Comme vous pouvez l'imaginer, si un itérateur parcourt une liste alors qu'un autre itérateur est en train de la modifier, il peut en résulter une certaine confusion. Par exemple, supposons qu'un itérateur fasse référence à un élément qu'un autre itérateur vient juste de supprimer. L'itérateur est désormais invalide et ne devrait donc plus être utilisé. Les itérateurs de listes chaînées ont été conçus pour détecter de telles modifications. Si un itérateur se rend compte que sa liste a été modifiée par un autre itérateur ou par une méthode de la collection, il déclenche une exception `ConcurrentModificationException`. Par exemple, considérons le code suivant :

```
List<String> list = . . .;
ListIterator<String> iter1 = list.listIterator();
ListIterator<String> iter2 = list.listIterator();
iter1.next();
iter1.remove();
iter2.next(); // déclenche une ConcurrentModificationException
```

L'appel à `iter2.next` lance une `ConcurrentModificationException`, car `iter2` détecte que la liste a été modifiée.

Pour éviter ce type d'exception, il suffit de suivre cette règle simple : vous pouvez attacher autant d'itérateurs à une collection que vous le souhaitez, tant qu'ils effectuent uniquement des lectures dans la liste. Sinon, vous pouvez y attacher un seul itérateur qui peut à la fois lire et écrire.

Le déclenchement de ce type d'exception observe aussi une règle simple. La collection garde une trace du nombre d'opérations de transformations (comme l'ajout ou la suppression d'un élément). Chaque itérateur compte le nombre de transformations dont il est responsable. Au début de chaque méthode d'itérateur, ce dernier vérifie que son nombre de transformations est bien égal à celui de la collection. Dans le cas contraire, il déclenche l'exception `ConcurrentModificationException`.

Il s'agit là d'un excellent test et d'une grande amélioration par rapport aux itérateurs fondamentalement non sécurisés de la STL du C++.

INFO

Il existe une curieuse exception à la règle permettant de détecter une modification simultanée. La liste chaînée garde uniquement la trace des modifications *structurelles* de la liste, comme l'ajout et la suppression de liens. La méthode `set` n'est pas comptée comme une modification structurelle. Vous pouvez attacher plusieurs itérateurs à une liste chaînée, et ils peuvent tous appeler `set` pour modifier le contenu des liens existants. Cette capacité est en fait nécessaire pour un certain nombre d'algorithmes de la classe `Collections` que nous aborderons plus loin dans ce chapitre.

Vous connaissez désormais les méthodes fondamentales de la classe `LinkedList`. Vous pouvez vous servir d'un `ListIterator` pour parcourir les éléments d'une liste chaînée dans n'importe quelle direction et pour ajouter ou supprimer des éléments.

Comme nous l'avons vu dans la section précédente, plusieurs autres méthodes pratiques manipulent les listes chaînées, qui sont déclarées dans l'interface `Collection`. Ces méthodes, pour la plupart, sont implémentées dans la superclasse `AbstractCollection` de la classe `LinkedList`. Par exemple, la méthode `toString` invoque `toString` pour chaque élément et renvoie une seule longue chaîne au format `[A, B, C]`. Cela est très pratique pour déboguer. Utilisez la méthode `contains` pour vérifier si un élément est présent dans une liste chaînée. Par exemple, l'appel à `staff.contains("Georges")` renvoie `true` si la liste chaînée contient déjà une chaîne égale à la chaîne "Georges".

La bibliothèque fournit aussi un certain nombre de méthodes qui sont, d'un point de vue théorique, quelque peu douteuses. Les listes chaînées ne permettent pas d'accéder rapidement à n'importe quel élément. Si vous voulez connaître le n -ième élément d'une liste chaînée, il faut commencer par le début de la liste et sauter les $n-1$ premiers éléments. Il n'existe aucun moyen plus rapide. Pour cette raison, les programmeurs n'utilisent généralement pas de listes chaînées dans des situations où les éléments doivent être référencés par un index entier.

Néanmoins, la classe `LinkedList` fournit une méthode `get` qui vous permet d'accéder à un élément particulier :

```
LinkedList<String> list = ...;  
String obj = list.get(n);
```

Bien sûr, cette technique n'est pas très efficace. Si vous vous rendez compte que vous l'utilisez, c'est probablement que vous utilisez une structure de données qui n'est pas adaptée à votre problème.

Il ne faudrait *jamais* avoir recours à cette méthode pour parcourir les éléments d'une liste chaînée. L'efficacité du code suivant doit donc être vivement remise en question :

```
for (int i = 0; i < list.size(); i++)  
    utilisation de list.get(i);
```

Chaque fois que vous recherchez un autre élément, vous parcourez à nouveau la liste depuis le début. L'objet `LinkedList` ne fait aucun effort pour se rappeler la position courante du dernier élément lu.

INFO

La méthode `get` possède une légère optimisation : si l'indice vaut au moins `size() / 2`, la recherche commence par la fin de la liste.

L'interface de l'itérateur de liste possède en outre une méthode renvoyant l'indice de la position courante. En fait, comme les itérateurs Java pointent *entre* les éléments, il en existe deux : la méthode `nextIndex` renvoie l'indice de l'élément qui serait renvoyé par la méthode `next`, et la méthode `previousIndex` retourne l'indice de l'élément qui serait renvoyé par la méthode `previous`. Cet indice correspond naturellement à celui de la méthode `nextIndex` moins un. Ces méthodes sont assez efficaces, car l'itérateur garde en mémoire sa position courante. Enfin, si vous avez un indice n , la méthode `list.listIterator(n)` renvoie un itérateur qui pointe juste avant l'élément dont l'indice vaut n . C'est-à-dire qu'un appel à `next` donnera le même élément que `list.get(n)`. L'obtention de cet itérateur reste assez peu efficace.

92 Au cœur de Java 2 - Fonctions avancées

Si l'une de vos listes chaînées ne possède que quelques éléments, vous n'avez pas trop de soucis à vous faire à propos de l'efficacité des méthodes `get` et `set`. Mais dans ce cas, quel est l'avantage d'une liste chaînée ? La seule motivation permettant de choisir une liste chaînée consiste à minimiser le coût d'une insertion ou d'une suppression d'un élément en plein milieu de la liste. Si vous n'avez que quelques éléments à traiter, il vaut mieux utiliser une `ArrayList`.

Nous conseillons d'ailleurs d'éviter toutes les méthodes qui utilisent un indice d'entier permettant de noter la position dans une liste chaînée. Pour obtenir un accès aléatoire dans une collection, mieux vaut utiliser un tableau ou encore `ArrayList`.

Le programme de l'Exemple 2.1 permet de manipuler des listes chaînées. Il se contente de créer deux listes chaînées, de les fusionner, puis il supprime un élément sur deux dans la seconde liste, et teste finalement la méthode `removeAll`. Nous vous recommandons de suivre l'exécution de ce programme et d'accorder une attention particulière aux itérateurs. Il vous sera éventuellement pratique de dessiner un diagramme représentant les positions de l'itérateur, comme ceci :

```
|ACE   |BDFG
A|CE   |BDFG
AB|CE  B|DFG
. . .
```

Remarquez que l'appel à

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

affiche tous les éléments de la liste chaînée `a` en appelant la méthode `toString` dans `AbstractCollection`.

Exemple 2.1 : *LinkedListTest.java*

```
import java.util.*;

/**
 * Ce programme présente le fonctionnement des listes chaînées.
 */
public class LinkedListTest
{
    public static void main(String[] args)
    {
        List<String> a = new LinkedList<String>();
        a.add("Claire");
        a.add("Carl");
        a.add("Erica");

        List<String> b = new LinkedList<String>();
        b.add("Bob");
        b.add("Doug");
        b.add("Frances");
        b.add("Gloria");

        // fusionne les mots de b dans a

        ListIterator<String> aIter = a.listIterator();
        Iterator<String> bIter = b.iterator();

        while (bIter.hasNext())
        {
            if (aIter.hasNext()) aIter.next();
```

```

        aIter.add(bIter.next());
    }

    System.out.println(a);

    // supprime un mot sur deux dans b

    bIter = b.iterator();
    while (bIter.hasNext())
    {
        bIter.next(); // saute un élément
        if (bIter.hasNext())
        {
            bIter.next(); // saute un élément
            bIter.remove(); // supprime cet élément
        }
    }

    System.out.println(b);

    // Opération globale : supprime tous les mots dans b de a
    a.removeAll(b);

    System.out.println(a);
}
}

```

java.util.List<E> 1.2

- **ListIterator<E> listIterator()**
Renvoie un itérateur de liste permettant de parcourir les éléments de la liste.
- **ListIterator<E> listIterator(int index)**
Renvoie un itérateur de liste permettant de parcourir les éléments de la liste dont le premier appel à next doit renvoyer l'élément correspondant à l'indice spécifié.
- **void add(int i, E element)**
Ajoute un élément à la position spécifiée.
Paramètre : *element* élément à ajouter
- **void addAll(int i, Collection<? Extends E> elements)**
Ajoute tous les éléments d'une collection à la position spécifiée.
- **E remove(int i)**
Supprime et renvoie un élément à la position spécifiée.
- **E set(int i, E element)**
Remplace l'élément à la position spécifiée par un nouvel élément et renvoie l'ancien élément.
- **int indexOf(Object element)**
Renvoie la position de la première occurrence d'un élément égal à l'élément spécifié, ou -1 si cet élément n'a pu être trouvé.
- **int lastIndexOf(Object element)**
Renvoie la position de la dernière occurrence d'un élément égal à l'élément spécifié, ou -1 si cet élément n'a pu être trouvé.

94 *Au cœur de Java 2 - Fonctions avancées*

java.util.ListIterator<E> 1.2

- **void add(E newElement)**
Ajoute un élément avant la position courante.
- **void set(E newElement)**
Remplace le dernier élément renvoyé par `next` ou `previous` par un nouvel élément. Déclenche une `IllegalStateException` si la structure de la liste a été modifiée depuis le dernier appel à `next` ou à `previous`.
- **boolean hasPrevious()**
Renvoie `true` s'il existe un élément avant la position courante.
- **E previous()**
Renvoie l'objet précédent. Déclenche une `NoSuchElementException` lorsque le début de la liste est atteint.
- **int nextIndex()**
Renvoie l'indice de l'élément qui serait renvoyé par le prochain appel à `next`.
- **int previousIndex()**
Renvoie l'indice de l'élément qui serait renvoyé par le prochain appel à `previous`.

java.util.LinkedList<E> 1.2

- **LinkedList()**
Construit une liste chaînée vide.
- **LinkedList(Collection<? extends E> elements)**
Construit une liste chaînée et y ajoute tous les éléments d'une collection.
- **void addFirst(E element)**
- **void addLast(E element)**
Ajoute un élément au début ou à la fin d'une liste.
- **E getFirst()**
- **E getLast()**
Renvoie l'élément situé au début ou à la fin d'une liste.
- **E removeFirst()**
- **E removeLast()**
Supprime et renvoie l'élément situé au début ou à la fin d'une liste.

Listes de tableaux

Dans la section précédente, nous avons abordé l'interface `List` et la classe `LinkedList` qui l'implémente. L'interface `List` décrit une collection classée dans laquelle la position des éléments a une importance. Il existe deux protocoles pour parcourir ses éléments : avec un itérateur ou avec les méthodes d'accès aléatoire `get` et `set`. Ces deux méthodes ne sont pas vraiment adaptées à une structure en liste chaînée, mais `get` et `set` prennent toute leur importance avec des tableaux. La bibliothèque de collections fournit une classe `ArrayList` qui implémente aussi l'interface `List`. Une `ArrayList` encapsule un tableau dynamique d'objets relogeable.

INFO

Les anciens de la programmation Java auront peut-être déjà utilisé la classe `Vector` pour créer un tableau dynamique. Pourquoi utiliser une `ArrayList` à la place d'un `Vector` ? Il existe une raison très simple. Toutes les méthodes de la classe `Vector` sont *synchronisées*. Vous pouvez donc accéder en toute tranquillité à un objet `Vector` depuis deux threads. Mais si vous n'accédez à un vecteur que depuis un seul thread (ce qui est de loin le cas le plus fréquent), votre code perd beaucoup de temps avec la synchronisation. Au contraire, les méthodes `ArrayList` ne sont pas synchronisées. C'est pourquoi nous vous recommandons d'utiliser une `ArrayList` à la place d'un vecteur chaque fois que vous n'avez pas besoin d'une synchronisation.

Tables de hachage

Les listes chaînées et les tableaux vous permettent de spécifier l'ordre dans lequel vous voulez organiser vos éléments. Cependant, si vous recherchez un élément particulier et que vous ne vous rappelez pas sa position, vous aurez besoin de parcourir tous les éléments jusqu'à ce que vous trouviez l'élément recherché. Cela requiert parfois beaucoup de temps, surtout lorsque la collection contient beaucoup d'éléments. Si l'ordre des éléments n'a pas d'importance, il existe des structures de données qui vous permettent de retrouver un élément beaucoup plus rapidement. L'inconvénient est que ces structures de données ne vous permettent pas de contrôler l'ordre des éléments. En effet, elles les organisent selon un ordre qui leur permet de les retrouver facilement.

Une structure de données classique pour retrouver simplement un élément est la *table de hachage*. Une table de hachage calcule un nombre entier, appelé *code de hachage*, pour chacun des éléments. Un code de hachage est un entier dérivé des champs d'instance d'un objet, pour que les objets contenant différentes données produisent des codes différents. Le Tableau 2.2 présente quelques exemples de codes de hachage nés de la méthode `hashCode` de la classe `String`.

Tableau 2.2 : Codes de hachage résultant de la fonction `hashCode`

Chaîne	Code de hachage
"Lee"	76268
"lee"	107020
"eel"	100300

Si vous définissez vos propres classes, vous serez responsable de l'implémentation de votre propre méthode `hashCode` (voir Chapitre 5 du Volume 1 pour plus d'informations). Votre implémentation doit être compatible avec la méthode `equals` : si `a.equals(b)`, `a` et `b` doivent avoir le même code de hachage.

Pour l'instant, il suffit de savoir que les codes de hachage peuvent être calculés très rapidement et que ce calcul ne dépend que de l'état de l'objet à rechercher, et non des autres objets de la table de hachage.

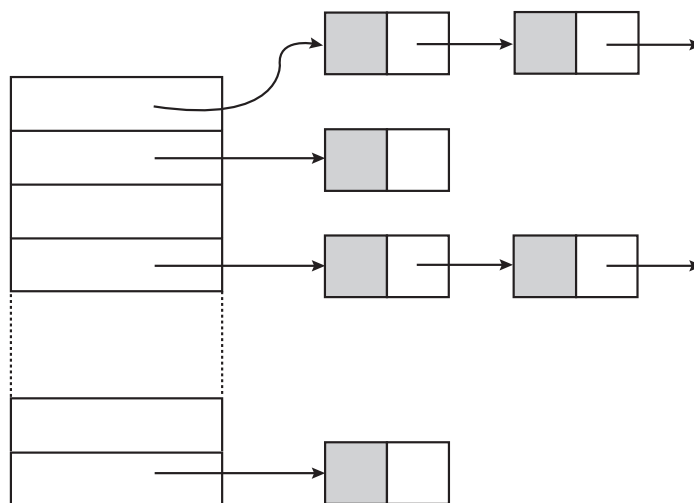
Une table de hachage est constituée d'un tableau de listes chaînées. Chaque liste est appelée un *seau* (ou *panier*, ou encore *bucket*) [Figure 2.8]. Pour retrouver la place d'un élément dans la table, il faut calculer son code de hachage et le réduire par un modulo du nombre total de seaux. Le nombre résultant correspond à l'indice du seau contenant l'élément. Par exemple, si un objet possède un code de

96 Au cœur de Java 2 - Fonctions avancées

hachage de 76268 et qu'il existe 128 seaux, l'objet sera placé dans le seau 108 (car le reste de la division entière $76268/128$ est 108). Avec un peu de chance, il n'existe aucun autre élément dans ce seau. Il suffit alors d'insérer l'élément dans le seau. Naturellement, il est inévitable de trouver parfois un seau déjà plein. Cela s'appelle une *collision de hachage*. Dans ce cas, vous comparez le nouvel objet avec tous les autres objets du seau pour déterminer s'il en fait déjà partie. En se fondant sur le fait que les codes de hachage sont distribués aléatoirement et que le nombre de seaux est suffisamment grand, il suffit généralement d'effectuer quelques comparaisons.

Figure 2.8

Une table de hachage.



Si vous voulez contrôler plus finement les performances d'une table de hachage, il est possible de spécifier le nombre initial de seaux, qui correspond au nombre de seaux utilisés pour tous les objets ayant le même code de hachage. Si trop d'éléments sont insérés dans la table de hachage, le nombre de collisions augmente et les performances de recherche baissent.

Si vous connaissez approximativement le nombre d'éléments de la table, il convient de choisir un nombre initial de seaux. Il doit généralement être compris entre 75 % et 150 % du nombre d'éléments prévus. Un certain nombre de chercheurs pensent qu'il est bon de choisir un nombre premier pour le nombre de seaux, afin d'éviter d'entasser trop de valeurs dans les mêmes seaux. Cette conclusion n'est cependant pas forcément évidente ! La bibliothèque standard utilise les nombres de seaux puissance de 2, avec un paramètre par défaut à 16 (toute valeur fournie pour la taille de la table est automatiquement arrondie au prochain nombre puissance de 2).

Naturellement, vous ne pouvez pas toujours connaître le nombre d'éléments que la table sera amenée à stocker, ou vos suppositions du départ peuvent se révéler erronées. Si la table de hachage se remplit trop, elle a besoin d'être réorganisée. Pour réorganiser une table de hachage, il faut créer une nouvelle table avec plus de seaux, insérer tous les éléments dans la nouvelle table, et supprimer l'ancienne table. Un *facteur de charge* est calculé pour savoir si une table doit être réorganisée ou non. Par exemple, si le facteur de charge vaut 0,75 (ce qui est la valeur par défaut) et que la table soit pleine à plus de 75 %, elle est automatiquement réorganisée avec le double de seaux. Pour la plupart des applications, il est raisonnable de conserver un facteur de charge autour de 0,75.

Les tables de hachage peuvent servir à implémenter plusieurs structures de données importantes. La plus simple d'entre elles est le type *set*. Un set correspond simplement à une collection d'éléments ne figurant qu'une seule fois chacun dans la collection. La méthode `add` d'un set commence par essayer de retrouver l'objet à ajouter et ne l'ajoute que s'il n'est pas encore présent.

La bibliothèque de collections Java fournit une classe `HashSet` qui implémente un set à partir d'une table de hachage.

Les éléments sont ajoutés avec la méthode `add`. La méthode `contains` est redéfinie pour effectuer une recherche rapide et vérifier si un élément fait déjà partie du set. Elle ne vérifie les éléments que d'un seul seau et non tous les éléments de la collection.

L'itérateur d'un set parcourt tous les seaux un par un. Comme les éléments d'une table de hachage sont dispersés dans toute la table, les seaux sont parcourus dans un ordre pseudo-aléatoire. C'est pourquoi il ne faut utiliser un set que si l'ordre des éléments dans la collection n'a aucune importance.

L'exemple de programme à la fin de cette section (voir Exemple 2.2) lit des mots à partir de `System.in`, les ajoute dans un set, puis imprime tous les mots du set. Par exemple, vous pouvez y entrer un texte issu d'*Alice au pays des merveilles* (que vous pouvez obtenir sur www.gutenberg.net), en l'exécutant à partir d'un shell, comme ceci :

```
java SetTest < alice30.txt
```

Le programme lit tous les mots qui lui sont passés et les ajoute dans le set. Il passe en revue tous les mots uniques et affiche pour terminer le nombre de mots uniques rencontrés. Ainsi, *Alice au pays des merveilles* comprend 5 909 mots uniques, en comptant la mention de copyright du début. Ces mots apparaissent selon un ordre aléatoire.

ATTENTION

Prenez garde lorsque vous faites muter les éléments set. Si le code de hachage d'un élément devait être modifié, l'élément ne se trouverait plus à la position correcte dans la structure de données.

Exemple 2.2 : *SetTest.java*

```
import java.util.*;

/**
 * Ce programme utilise un set pour imprimer tous les mots uniques dans
 * System.in.
 */
public class SetTest
{
    public static void main(String[] args)
    {
        Set<String> words = new HashSet<String>(); //HashSet implémente Set
        // définit un HashSet, LinkedHashSet or TreeSet
        long totalTime = 0;

        Scanner in = new Scanner(System.in);
        while (in.hasNext())
        {
            String word = in.next();
```


98 Au cœur de Java 2 - Fonctions avancées

```

        long callTime = System.currentTimeMillis();
        words.add(word);
        callTime = System.currentTimeMillis() - callTime;
        totalTime += callTime;

    }

    Iterator<String> iter = words.iterator();
    for (int i = 1; i <= 20; i++)
        System.out.println(iter.next());
    System.out.println("...");
    System.out.println(words.size()
        + " mots différents. " + totalTime + " millisecondes.");
}

```

java.util.HashSet<E> 1.2

- `HashSet()`
Construit un set vide.
- `HashSet(Collection<? extends E> elements)`
Construit un set et y ajoute tous les éléments d'une collection.
- `HashSet(int initialCapacity)`
Construit un set vide avec la capacité spécifiée.
- `HashSet(int initialCapacity, float loadFactor)`
Construit un set vide avec la capacité et le facteur de charge spécifiés (un nombre compris entre 0,0 et 1,0 qui détermine le pourcentage de remplissage d'une table de hachage au-delà duquel la table sera réorganisée).

java.lang.Object 1.0

- `int hashCode()`
Renvoie un code de hachage correspondant à cet objet, c'est-à-dire n'importe quel nombre entier, positif ou négatif. Les définitions de `equals` et `hashCode` doivent être compatibles. Si `x.equals(y)` vaut `true`, `x.hashCode()` doit avoir la même valeur que `y.hashCode()`.

Arbres

La classe `TreeSet` ressemble beaucoup à la classe `set`, avec une amélioration supplémentaire. Les arbres de hachage représentent des *collections triées*. Vous pouvez insérer des éléments dans la collection dans n'importe quel ordre, et lorsque vous parcourez ces éléments, ils sont automatiquement présentés selon un ordre croissant. Par exemple, supposons que vous ajoutiez trois chaînes, puis que vous parcouriez tous les éléments que vous venez d'ajouter.

```

SortedSet<String> sorter = new TreeSet<String>(); // TreeSet
// implémente SortedSet
sorter.add("Bénédicte");
sorter.add("Antoine");
sorter.add("Claire");
for (String s : sorter) System.println(s);

```

Les chaînes sont affichées par ordre croissant : Antoine Bénédicte Claire. Comme le nom de la classe le laisse entendre, le tri est accompli par une structure de données en arbre. L'implémentation

courante se sert d'un *arbre rouge-noir*. Pour une description détaillée des arbres rouge-noir, consultez, par exemple, le livre (en langue anglaise) *Introduction to Algorithms* de Thomas Cormen, Charles Leiserson, Ronald Rivest et Clifford Stein (The MIT Press, 2001). Chaque fois qu'un élément est ajouté à un arbre, il est placé à un endroit correspondant à son rang parmi les éléments triés. Par conséquent, l'itérateur parcourt toujours les éléments d'un arbre selon un ordre croissant.

L'ajout d'un élément dans un arbre est plus lent que s'il fallait l'ajouter dans une table de hachage, mais cela reste bien plus rapide que de l'ajouter dans un tableau trié ou dans une liste chaînée. Si l'arbre contient n éléments, une moyenne de $\log_2 n$ comparaisons sera nécessaire pour trouver la position d'un nouvel élément dans l'arbre. Par exemple, si l'arbre contient déjà 1 000 éléments, l'ajout d'un nouvel élément nécessitera environ 10 comparaisons.

Par conséquent, il est moins rapide d'ajouter un élément dans un `TreeSet` que dans un `HashSet` (voir le Tableau 2.3 pour une comparaison), mais le `TreeSet` trie automatiquement ses éléments.

Tableau 2.3 : Ajouter des éléments dans un `HashSet` et dans un `TreeSet`

<i>Document</i>	<i>Nombre total de mots</i>	<i>Nombre de mots uniques</i>	<i>HashSet</i>	<i>TreeSet</i>
<i>Alice au pays des merveilles</i>	28195	5909	5 sec	7 sec
<i>Le Comte de Monte-Cristo</i>	466300	37545	75 sec	98 sec

java.util.TreeSet<E> 1.2

- `TreeSet()`
Construit un `TreeSet` vide.
- `TreeSet(Collection<? extends E> elements)`
Construit un `TreeSet` et y ajoute tous les éléments d'une collection.

Comparaison d'objets

Comment le `TreeSet` sait-il dans quel ordre trier les éléments ? Par défaut, cette structure de données part de l'hypothèse que vous insérez des éléments qui implémentent l'interface `Comparable`. Cette interface définit une seule méthode :

```
public interface Comparable<T>
{
    int compareTo(T other);
}
```

L'appel `a.compareTo(b)` doit renvoyer 0 si `a` et `b` sont égaux, un nombre entier négatif si `a` est inférieur à `b` (selon l'ordre de tri choisi), et un nombre entier positif si `a` est supérieur à `b`. La valeur exacte renvoyée par cette méthode n'a pas grande importance. Seul son signe (>0 , 0 ou <0) a une signification. Plusieurs classes de la plate-forme Java standard implémentent l'interface `Comparable`. La classe `String` en est un bon exemple. Sa méthode `compareTo` compare des chaînes selon l'ordre du dictionnaire (aussi appelé ordre *lexicographique*).

Si vous insérez vos propres objets, il vous faut définir un ordre de tri en implémentant l'interface `Comparable`. Il n'existe aucune implémentation par défaut de `compareTo` dans la classe `Object`.

100 Au cœur de Java 2 - Fonctions avancées

Par exemple, voici comment trier des objets `Item` en fonction de leur numéro de code :

```
class Item implements Comparable<Item>
{
    public int compareTo(Item other)
    {
        return partNumber - other.partNumber;
    }
    . . .
}
```

Si vous comparez deux entiers *positifs*, comme les numéros de code de notre exemple, il vous suffit de renvoyer leur différence, qui sera négative si le premier article est inférieur au second, nulle si les deux articles sont identiques, et positive dans le dernier cas.

ATTENTION

Cette astuce ne fonctionne que si les nombres entiers proviennent d'une gamme assez petite. En effet, si x est un grand nombre positif et que y soit un grand nombre négatif, la différence $x - y$ pourra provoquer un débordement.

Il est bien évident que l'utilisation de l'interface `Comparable` pour définir un ordre de tri possède certaines limitations. Cette interface ne peut être implémentée qu'une seule fois. Mais que peut-on faire pour trier un ensemble d'articles selon leur numéro de code dans une collection, et en fonction de leur description dans une autre ? De plus, que pouvez-vous faire pour trier des objets d'une classe dont le concepteur n'a pas pris le soin d'implémenter l'interface `Comparable` ?

Dans ces situations, il faut demander à l'arbre de hachage de recourir à une autre méthode de comparaison, en passant un objet `Comparator` dans le constructeur `TreeSet`. L'interface `Comparator` déclare une méthode `compare`, avec deux paramètres explicites :

```
public interface Comparator<T>
{
    int compare(T a, T b);
}
```

Comme la méthode `compareTo`, la méthode `compare` renvoie une valeur entière négative si a est inférieur à b , nulle si a et b sont identiques, et positive dans le dernier cas.

Pour trier des articles selon leur description, il suffit de définir une classe qui implémente l'interface `Comparator` :

```
class ItemComparator implements Comparator<Item>
{
    public int compare(Item a, Item b)
    {
        String descrA = a.getDescription();
        String descrB = b.getDescription();
        return descrA.compareTo(descrB);
    }
}
```

Il faut ensuite passer un objet de cette classe au constructeur de l'arbre :

```
ItemComparator comp = new ItemComparator();
SortedSet sortByDescription = new TreeSet<Item>(comp);
```

Si vous construisez un arbre avec un comparateur, il se sert de cet objet même s'il doit comparer deux éléments.

Remarquez que le comparateur d'articles ne possède aucune donnée. Il s'agit juste d'un emplacement pour la méthode de comparaison. Ce type d'objet est parfois appelé un *objet de fonction*.

Les objets de fonction sont souvent définis "en cours de route", comme des instances de classes internes anonymes :

```
SortedSet<Item> sortByDescription = new TreeSet<Item>(new
    Comparator<Item>()
    {
        public int compare(Item a, Item b)
        {
            String descrA = a.getDescription();
            String descrB = b.getDescription();
            return descrA.compareTo(descrB);
        }
    });
```

INFO

En réalité, l'interface `Comparator<T>` est déclarée comme ayant deux méthodes : `compare` et `equals`. Bien entendu, chaque classe possède une méthode `equals` ; il y aurait donc peu d'avantages à ajouter la méthode à la déclaration de l'interface. La documentation de l'API explique que vous devez surcharger la méthode `equals` mais que cela peut améliorer les performances dans certains cas. La méthode `addAll` de la classe `TreeSet`, par exemple, fonctionnera plus efficacement si vous ajoutez des éléments d'un autre set qui utilise le même comparateur.

Si vous revenez un instant au Tableau 2.3, vous pourriez vous demander s'il convient d'utiliser systématiquement un `TreeSet` à la place d'un `HashSet`. Après tout, l'ajout d'un élément n'a pas l'air de prendre beaucoup plus de temps, et les éléments sont triés automatiquement. La réponse à cette question dépend en fait des données traitées. Si vous n'avez pas besoin que les données soient triées, il n'y a aucune raison de ralentir votre programme pour le plaisir de les trier. De plus, il est parfois très difficile de définir un ordre de tri pour certains types de données. Supposons que vous traitiez un ensemble de rectangles. Comment allez-vous les trier ? Par aire ? Il se peut que deux rectangles distincts occupent des emplacements différents, mais que leurs aires soient identiques. Si vous choisissez de les trier par aire, le second ne sera pas pris en considération. L'ordre de tri d'un arbre doit nécessairement être un *ordre total*. Cela signifie que deux éléments quelconques doivent être comparables, et que la comparaison de deux éléments ne peut renvoyer zéro que si ces deux éléments sont identiques. Il existe bien un tel ordre applicable aux rectangles (l'ordre lexicographique appliqué sur ses coordonnées), mais il n'est pas très naturel et plutôt compliqué à calculer. Par opposition, les fonctions de hachage sont généralement plus simples à définir. Il leur suffit d'effectuer une bonne dispersion des objets, alors que les fonctions de comparaison doivent classer les objets avec une précision parfaite.

Le programme de l'Exemple 2.3 génère deux `TreeSet` d'objets `Item`. Le premier est trié par numéro de code, l'ordre de tri par défaut des objets `Item`. Le second set est trié par description, à l'aide d'un comparateur particulier.

102 Au cœur de Java 2 - Fonctions avancées**Exemple 2.3 : TreeSetTest.java**

```
import java.util.*;

/**
 * Ce programme trie un set d'éléments en comparant
 * leurs descriptions.
 */
public class TreeSetTest
{
    public static void main(String[] args)
    {
        SortedSet<Item> parts = new TreeSet<Item>();
        parts.add(new Item("Grille-pain", 1234));
        parts.add(new Item("Accessoire", 4562));
        parts.add(new Item("Modem", 9912));
        System.out.println(parts);

        SortedSet<Item> sortByDescription = new TreeSet<Item>(new
            Comparator<Item>()
            {
                public int compare(Item a, Item b)
                {
                    String descrA = a.getDescription();
                    String descrB = b.getDescription();
                    return descrA.compareTo(descrB);
                }
            });

        sortByDescription.addAll(parts);
        System.out.println(sortByDescription);
    }
}

/**
 * Un élément avec une description et un numéro de code.
 */
class Item implements Comparable<Item>
{
    /**
     * Construit un élément.
     * @param aDescription la description de l'élément
     * @param aPartNumber le numéro de code de l'élément
     */
    public Item(String aDescription, int aPartNumber)
    {
        description = aDescription;
        partNumber = aPartNumber;
    }

    /**
     * Obtient la description de cet élément.
     * @return la description
     */
    public String getDescription()
    {
        return description;
    }
}
```

```

public String toString()
{
    return "[description=" + description
        + ", partNumber=" + partNumber + "]\n";
}

public boolean equals(Object otherObject)
{
    if (this == otherObject) return true;
    if (otherObject == null) return false;
    if (getClass() != otherObject.getClass()) return false;
    Item other = (Item) otherObject;
    return description.equals(other.description)
        && partNumber == other.partNumber;
}

public int hashCode()
{
    return 13 * description.hashCode() + 17 * partNumber;
}

public int compareTo(Item other)
{
    return partNumber - other.partNumber;
}

private String description;
private int partNumber;
}

```

java.lang.Comparable<T> 1.2

- `int compareTo(T other)`

Compare cet objet avec un autre objet et renvoie une valeur négative si `this` est inférieur à `other`, nulle si les deux objets sont identiques au sens de l'ordre de tri, et positive si `this` est supérieur à `other`.

java.util.Comparator<T> 1.2

- `int compare(T a, T b)`

Compare deux objets et renvoie une valeur négative si `a` est inférieur à `b`, nulle si les deux objets sont identiques au sens de l'ordre de tri, et positive si `a` est supérieur à `b`.

java.util.SortedSet<E> 1.2

- `Comparator<? super E> comparator()`

Renvoie le comparateur utilisé pour trier les éléments, ou `null` si les éléments sont comparés avec la méthode `compareTo` de l'interface `Comparable`.

- `E first()`
- `E last()`

Renvoient le plus grand ou le plus petit élément de la collection triée.

java.util.TreeSet<E> 1.2

- `TreeSet()`

Construit un `TreeSet` pour stocker les objets `Comparable`.

104 Au cœur de Java 2 - Fonctions avancées

- `TreeSet(Comparator<? super E> c)`
Construit un `TreeSet` et se sert du comparateur spécifié pour trier ses éléments.
- `TreeSet(SortedSet<? extends E> elements)`
Construit un `TreeSet`, ajoute tous les éléments d'un ensemble trié, et se sert du même comparateur que celui de l'ensemble spécifié.

Queues de priorité

Une queue de priorité récupère des éléments triés, après qu'ils ont été insérés dans un ordre arbitraire. Ainsi, dès que vous appelez la méthode `remove`, vous obtenez le plus petit élément se trouvant dans la queue. Celle-ci ne trie toutefois pas tous ses éléments. Si vous parcourez les éléments, ils ne sont pas nécessairement triés. La queue de priorité fait usage d'une structure de données efficace et élégante, appelée un *tas*. Il s'agit d'une arborescence binaire dans laquelle les opérations `add` et `remove` amènent l'élément le plus petit à graviter autour de la racine, sans perdre du temps à trier tous les éléments.

A l'instar d'un `TreeSet`, une queue de priorité peut contenir soit les éléments d'une classe qui implémente l'interface `Comparable`, soit un objet `Comparator` que vous fournissez dans le constructeur.

Ces queues servent généralement à planifier les tâches. Chaque tâche se voit attribuer une priorité, et elles sont ajoutées dans un ordre aléatoire. Dès le démarrage possible d'une nouvelle tâche, celle ayant la plus haute priorité est supprimée de la queue (la priorité 1 étant généralement la plus forte, l'opération de suppression concerne le plus petit élément).

L'Exemple 2.4 présente une queue de priorité. A la différence de l'itération dans un `TreeSet`, celle-ci ne parcourt pas les éléments dans l'ordre de tri. Cependant, le retrait concerne toujours le plus petit élément restant.

Exemple 2.4 : `PriorityQueueTest.java`

```
import java.util.*;

/**
 * Ce programme présente une queue de priorité.
 */
public class PriorityQueueTest
{
    public static void main(String[] args)
    {
        PriorityQueue<GregorianCalendar> pq =
            new PriorityQueue<GregorianCalendar>();
        pq.add(new GregorianCalendar(1906, Calendar.DECEMBER, 9));
            // G. Hopper
        pq.add(new GregorianCalendar(1815, Calendar.DECEMBER, 10));
            // A. Lovelace
        pq.add(new GregorianCalendar(1903, Calendar.DECEMBER, 3));
            // J. von Neumann
        pq.add(new GregorianCalendar(1910, Calendar.JUNE, 22));
            // K. Zuse
    }
}
```

```

        System.out.println("Itération sur les éléments...");
        for (GregorianCalendar date : pq)
            System.out.println(date.get(Calendar.YEAR));
        System.out.println("Suppression d'éléments...");
        while (!pq.isEmpty())
            System.out.println(pq.remove().get(Calendar.YEAR));
    }
}

```

java.util.PriorityQueue 5.0

- `PriorityQueue()`
- `PriorityQueue(int initialCapacity)`
Construisent un `TreeSet` pour stocker des objets `Comparable`.
- `PriorityQueue(int initialCapacity, Comparator<? super E> c)`
Construit un `TreeSet` et utilise le comparateur spécifié pour trier ses éléments.

Cartes

Un set est une collection qui vous permet de retrouver rapidement un élément. Pour cela, il faut le plus souvent le connaître exactement, ce qui n'est pas souvent le cas. En fait, on dispose généralement de certaines informations importantes sur l'élément à rechercher, et il faut le retrouver à partir de ces informations. La structure de données *cartes* (ou *map*) permet d'effectuer ce genre de recherche. Une carte enregistre des paires clé / valeur. Une valeur peut être retrouvée à partir de la clé correspondante. Par exemple, vous pouvez disposer d'un tableau de données sur des salariés, dans lequel les clés sont les numéros de salariés et les valeurs des objets `Employee`.

La bibliothèque Java propose deux implémentations générales des cartes : `HashMap` et `TreeMap`. Les deux classes implémentent l'interface `Map`.

Une `HashMap` répartit les clés dans plusieurs catégories, alors que la `TreeMap` se sert d'un ordre total pour organiser les clés dans un arbre de recherche. Les fonctions de hachage ou de comparaison ne sont appliquées qu'aux clés. Les valeurs associées à ces clés ne sont ni comparées ni réparties.

Comment choisir entre une `HashMap` et une `TreeMap` ? Comme pour les sets, les cartes de hachage sont légèrement plus rapides et elles sont à privilégier si vous n'avez pas besoin de parcourir les éléments selon un ordre particulier.

Voici comment mettre en place une carte de hachage pour enregistrer des données sur des salariés :

```

Map<String, Employee> staff = new HashMap<String, Employee>();
// HashMap implémente Map
Employee claire = new Employee("Claire Tessier");
staff.put("987-98-9996", claire);
. . .

```

Dès qu'un nouvel objet est ajouté à une carte, il faut également fournir une clé associée. Dans le cas qui nous intéresse, la clé est constituée d'une chaîne et la valeur correspondante est un objet `employee`.

Pour retrouver un objet, il faut se servir de la clé (et par conséquent, il faut la connaître).

```

String s = "987-98-9996";
e = staff.get(s); // retrouve les informations sur Claire

```


106 Au cœur de Java 2 - Fonctions avancées

Si aucune information n'est stockée dans la carte pour une clé spécifiée, `get` renvoie `null`.

Les clés doivent être uniques. Il n'est pas permis d'associer deux valeurs à la même clé. Si vous appelez la méthode `put` deux fois avec la même clé, la seconde valeur remplace la première. En fait, `put` renvoie la dernière valeur correspondant à la clé fournie.

La méthode `remove` supprime de la carte un élément ayant une clé donnée. La méthode `size` renvoie quant à elle le nombre d'éléments de la carte.

Les cartes ne sont pas vraiment considérées comme des collections. Au sens de certains ensembles de structures de données, les cartes sont des collections de *paires*, c'est-à-dire des collections de valeurs indexées par des clés. Il reste néanmoins possible d'obtenir une *vue* d'une carte, c'est-à-dire un objet qui implémente l'interface `Collection` ou l'une de ses sous-interfaces.

Il existe trois vues différentes : l'ensemble des clés, l'ensemble des valeurs et l'ensemble des paires clé / valeur. Les ensembles des clés et des paires clé / valeur forment un set parce qu'il ne peut exister qu'un seul exemplaire d'une clé dans une carte. Les méthodes

```
Set<K> keySet()
Collection<K> values()
Set<Map.Entry<K, V>> entrySet()
```

permettent d'afficher ces trois vues. Les éléments de la troisième méthode font partie de la classe interne statique `Map.Entry`.

Notons que le `keySet` n'est pas un `HashSet` ni un `TreeSet`, mais il s'agit d'un objet d'une autre classe qui implémente l'interface `Set`, laquelle étend l'interface `Collection`. Vous pouvez donc utiliser un `keySet` comme n'importe quelle autre collection.

Par exemple, vous pouvez énumérer toutes les clés d'une carte :

```
Set<String> keys = map.keySet();
for (String key : keys)
{
    Utilisation de key
}
```

ASTUCE

Si vous voulez examiner à la fois les clés et les valeurs associées, vous pouvez éviter de rechercher les valeurs en énumérant les *entrées*. Vous pouvez vous servir de l'exemple suivant :

```
for (Map.Entry<String, Employee> entry : staff.entrySet())
{
    String key = entry.getKey();
    Employee value = entry.getValue();
    Utilisation de key et de value
}
```

Si vous invoquez la méthode `remove` de l'itérateur, vous supprimez en fait la *clé* et la *valeur associée*. Il n'est toutefois pas possible d'ajouter un élément à l'affichage du set de la clé. L'ajout d'une clé sans valeur associée n'a en effet aucun sens dans une carte. Si vous essayez d'invoquer la méthode `add`, cela déclenchera une `UnsupportedOperationException`. Le set d'entrées possède la même restriction, bien que l'ajout d'une nouvelle paire clé / valeur ait un sens pour ce set.

L'Exemple 2.5 présente un exemple d'utilisation de carte. Nous commençons par ajouter des paires clé / valeur dans la carte, puis nous supprimons une des clés de la carte, ce qui supprime également la valeur associée. Ensuite, nous modifions la valeur associée à une clé et nous appelons la méthode `get` pour retrouver la valeur. Pour terminer, toutes les entrées du set sont passées en revue.

Exemple 2.5 : *MapTest.java*

```
import java.util.*;

/**
 * Ce programme montre l'utilisation d'une carte avec le type de clé
 * String et le type de valeur Employee.
 */
public class MapTest
{
    public static void main(String[] args)
    {
        Map<String, Employee> staff = new HashMap<String, Employee>();
        staff.put("144-25-5464", new Employee("Nicolas Babled"));
        staff.put("567-24-2546", new Employee("Ludovic Bertin"));
        staff.put("157-62-7935", new Employee("Nicolas Serres"));
        staff.put("456-62-5527", new Employee("Pierre Dumas"));

        // imprime toutes les entrées

        System.out.println(staff);

        // supprime une entrée

        staff.remove("567-24-2546");

        // remplace une entrée

        staff.put("456-62-5527", new Employee("David Fontanella"));

        // recherche une valeur

        System.out.println(staff.get("157-62-7935"));

        // parcourt toutes les entrées

        for (Map.Entry<String, Employee> entry : staff.entrySet())
        {
            String key = entry.getKey();
            Employee value = entry.getValue();
            System.out.println("key=" + key + ", value=" + value);
        }
    }
}

/**
 * Une classe d'employés minimaliste pour un objectif de test.
 */
class Employee
```

108 Au cœur de Java 2 - Fonctions avancées

```
{
    /**
     * Génère un employé avec un salaire de $0.
     * @param n le nom de l'employé
     */
    public Employee(String n)
    {
        name = n;
        salary = 0;
    }

    public String toString()
    {
        return "[name=" + name + ", salary=" + salary + "]";
    }

    private String name;
    private double salary;
}
```

java.util.Map<K, V> 1.2

- **V get(K key)**
Cherche la valeur associée à une clé, puis renvoie l'objet associé, ou null si la clé n'a pas été trouvée dans la carte. La clé peut être nulle.
- **V put(K key, V value)**
Associe une clé et une valeur dans une carte. Si la clé est déjà présente dans la carte, le nouvel objet remplace celui qui était associé à la clé. Cette méthode renvoie l'ancienne valeur associée à la clé, ou null si la clé n'existait pas dans la carte. La clé peut être nulle, mais la valeur doit être non nulle.
- **void putAll(Map<? extends K, ? extends V> entries)**
Ajoute toutes les entrées d'une carte spécifiée à cette carte.
- **boolean containsKey(Object key)**
Renvoie true si la clé est présente dans la carte.
- **boolean containsValue(Object value)**
Renvoie true si la valeur est présente dans la carte.
- **Set<Map, Entry<K, V>> entrySet()**
Renvoie une vue des objets Map.Entry, contenant les paires clé / valeur de la carte. Vous pouvez supprimer des éléments de cette vue (ils sont alors éliminés de la carte), mais vous ne pouvez pas en ajouter de nouveaux.
- **Set<K> keySet()**
Renvoie une vue de toutes les clés de la carte. Lorsque vous supprimez des éléments de ce set, la clé et la valeur associée sont supprimées de la carte. En revanche, vous ne pouvez pas ajouter de nouveaux éléments.
- **Collection<V> values()**
Renvoie une vue de toutes les valeurs de la carte. Vous pouvez supprimer des éléments de ce set (la clé et la valeur associée sont alors supprimées de la carte), mais vous n'avez pas le droit d'y ajouter de nouveaux éléments.

java.util.Map.Entry<K, V> 1.2

- `K getKey()`
Renvoie la clé ou la valeur de cette entrée.
- `V getValue()`
Renvoie la valeur de la *carte associée* avec la nouvelle valeur et renvoie l'ancienne valeur.
- `V setValue(V newValue)`
Modifie la valeur de la *carte associée* avec la nouvelle valeur et renvoie l'ancienne valeur.

java.util.HashMap<K, V> 1.2

- `HashMap()`
- `HashMap(int initialCapacity)`
- `HashMap(int initialCapacity, float loadFactor)`
Construisent une carte de hachage vide avec la capacité et le facteur de charge spécifiés (un nombre compris entre 0,0 et 1,0 qui détermine le pourcentage de remplissage au-delà duquel la table de hachage sera réorganisée). Le facteur de charge par défaut vaut 0,75.

java.util.TreeMap<K, V> 1.2

- `TreeMap(Comparator<? super K> c)`
Construit une carte en arbre et se sert du comparateur spécifié pour trier ses clés.
- `TreeMap(Map<? extends K, ? extends V> entries)`
Construit une carte en arbre et y ajoute toutes les entrées de la carte spécifiée.
- `TreeMap(SortedMap<? extends K, ? extends V> entries)`
Construit un arbre, ajoute toutes les entrées de la carte triée, et se sert du même comparateur d'éléments que la carte triée.

java.util.SortedMap<K, V> 1.2

- `Comparator<? super K> comparator()`
Renvoie le comparateur servant à trier les clés, ou `null` si les clés sont comparées à l'aide de la méthode `compareTo` de l'interface `Comparable`.
- `K firstKey()`
- `K lastKey()`
Renvoient la plus petite ou la plus grande clé de la carte.

Classes de cartes et de set spécialisées

La bibliothèque de classes de collections possède plusieurs classes de cartes pour les besoins spécialisés que nous traiterons brièvement dans cette section.

Cartes de hachage faibles

La classe `WeakHashMap` a été conçue dans le but de résoudre un problème intéressant. Que se passe-t-il avec une valeur lorsque sa clé n'est plus utilisée nulle part dans votre programme ? Supposons que la dernière référence à une clé ait disparu. Il n'y a alors plus aucune manière de se référer à l'objet de la valeur. Toutefois, étant donné qu'aucune partie du programme ne possède plus la clé, la paire clé / valeur ne peut être supprimée de la carte. Pourquoi alors le ramasse-miettes ne peut-il pas la supprimer ? Ce serait en effet son rôle de supprimer les objets inutilisés.

110 Au cœur de Java 2 - Fonctions avancées

Malheureusement, la situation n'est pas aussi simple. Le ramasse-miettes suit les objets *vivants*. Tant que l'objet de carte est vivant, *tous* les seaux qu'il contient sont également vivants et ils ne seront pas réclamés. Ainsi, votre programme doit supprimer les valeurs inutilisées des cartes vivantes. Vous pouvez également utiliser à la place `WeakHashMap`. Cette structure de données coopère avec le ramasse-miettes pour supprimer les paires clé / valeur lorsque la seule référence à la clé est la référence de l'entrée de la table de hachage.

Voici en fait le fonctionnement interne de ce mécanisme. `WeakHashMap` utilise des *références faibles* pour conserver les clés. Un objet `WeakReference` contient une référence à un autre objet, dans notre cas, une clé de table de hachage. Les objets de ce type sont considérés de manière particulière par le ramasse-miettes. Normalement, s'il découvre qu'un objet particulier ne possède pas de références vers lui, il réclame simplement l'objet. Toutefois, si l'objet peut être atteint *uniquement* par un `WeakReference`, le ramasse-miettes réclame toujours l'objet, mais place la référence faible qui y menait dans une queue. Les opérations du `WeakHashMap` vérifient périodiquement cette queue pour y retrouver des références faibles nouvellement arrivées. L'arrivée d'une référence faible dans la queue signifie que la clé n'est plus utilisée par personne et qu'elle a été collectée. `WeakHashMap` supprime alors l'entrée associée.

Cartes et sets de hachage liés

Le JDK 1.4 ajoute les classes `LinkedHashSet` et `LinkedHashMap` qui se souviennent de l'ordre dans lequel vous avez inséré des éléments. De cette manière, vous évitez l'ordre assez aléatoire des éléments dans une table de hachage. A mesure que des entrées sont insérées dans la table, elles sont simplement reliées ensemble dans une liste doublement liée (voir Figure 2.9).

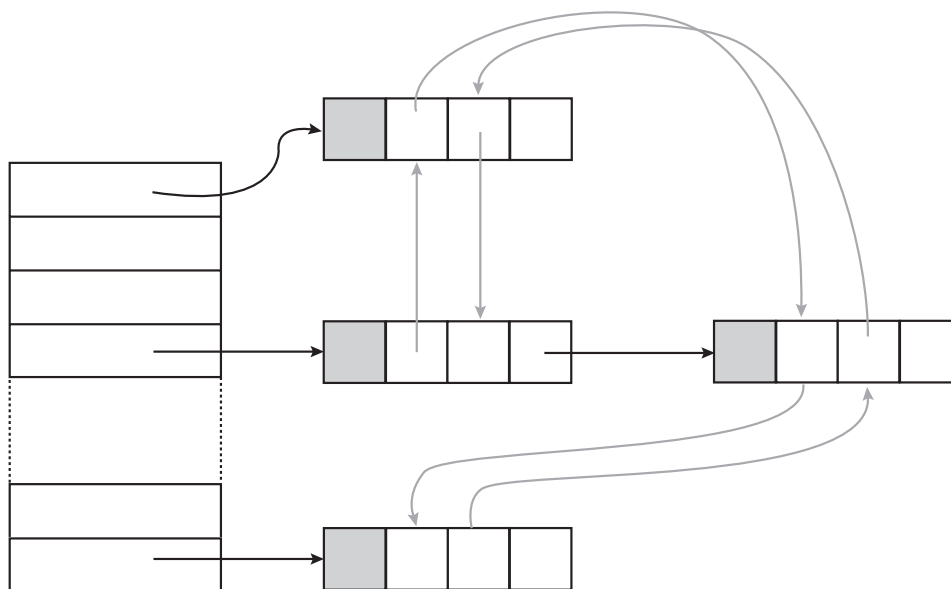


Figure 2.9

Une table de hachage liée.

Par exemple, considérez les insertions de cartes suivantes de l'Exemple 2.5 :

```
Map staff = new LinkedHashMap();
staff.put("144-25-5464", new Employee("Nicolas Babled"));
    staff.put("567-24-2546", new Employee("Ludovic Bertin"));
    staff.put("157-62-7935", new Employee("Nicolas Serres"));
    staff.put("456-62-5527", new Employee("Pierre Dumas"));
```

A ce moment-là, `staff.keySet().iterator()` énumère les clés dans cet ordre :

```
144-25-5464
567-24-2546
157-62-7935
456-62-5527
```

et `staff.values().iterator()` énumère les valeurs dans l'ordre :

```
Nicolas Babled
Ludovic Bertin
Nicolas Serres
Pierre Dumas
```

Une carte de hachage liée peut utiliser *l'ordre d'accès*, et non l'ordre d'insertion, pour parcourir les entrées de la carte. A chaque fois que vous appelez `get` ou `put`, l'entrée affectée est supprimée de sa position et placée à la *fin* de la liste chaînée des entrées (seule la position dans la liste chaînée est concernée, et non le seau de la table de hachage. Une entrée reste toujours dans le seau qui correspond au code de hachage de la clé). Pour construire une carte de hachage, appelez

```
LinkedHashMap<K, V>(initialCapacity, loadFactor, true)
```

L'ordre d'accès est utile pour implémenter le système de "la dernière utilisée" pour un cache. Par exemple, vous pouvez vouloir conserver les entrées fréquemment lues en mémoire et lire des objets lus moins souvent à partir d'une base de données. Lorsqu'une entrée d'une table est introuvable et que la table est déjà assez pleine, vous pouvez obtenir un itérateur dans la table, puis supprimer les premiers éléments qu'elle énumère. Ces entrées étaient les dernières utilisées.

Vous pouvez même automatiser cette procédure. Formez une sous-classe de `LinkedHashMap` et surchargez la méthode

```
protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
```

Ajoutez ensuite une nouvelle entrée pour supprimer l'entrée la plus ancienne lorsque votre méthode renvoie `true`. Par exemple, le cache suivant est conservé à une taille maximale de 100 éléments.

```
Map<K, V> cache = new
    LinkedHashMap<K, V>(128, 0.75F, true)
    {
        protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
        {
            return size() > 100;
        }
    };
```

Vous pouvez également envisager l'entrée `eldest` pour décider si vous devez le supprimer. Par exemple, vous voudrez peut-être vérifier une indication de la date et de l'heure stockée avec l'entrée.

112 Au cœur de Java 2 - Fonctions avancées

Sets et cartes d'énumération

EnumSet est une implémentation efficace de sets, dont les éléments appartiennent à un type énuméré. Un type énuméré ayant un nombre fini d'instances, EnumSet est implémenté en interne sous la forme d'une suite de bits. Un bit est activé si la valeur correspondante est présente dans le set.

La classe EnumSet ne possède pas de constructeurs publics. Utilisez une méthode factory statique pour construire le set :

```
enum Weekday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
                SATURDAY, SUNDAY };
EnumSet<Weekday> always = EnumSet.allOf(Weekday.class);
EnumSet<Weekday> never = EnumSet.noneOf(Weekday.class);
EnumSet<Weekday> workday = EnumSet.range(Weekday.MONDAY, Weekday.FRIDAY);
EnumSet<Weekday> mwf = EnumSet.of(Weekday.MONDAY,
                                   Weekday.WEDNESDAY, Weekday.FRIDAY);
```

Pour modifier un EnumSet, vous pouvez utiliser les méthodes habituelles de l'interface Set.

EnumMap est une carte dont les clés appartiennent à un type énuméré. Il est implémenté de manière simple et efficace sous forme de tableau de valeurs. Le type de la clé doit être spécifié dans le constructeur :

```
EnumMap<Weekday, Employee> personInCharge = new EnumMap<Weekday, Employee>
(Weekday.class);
```

INFO

Dans la documentation API de EnumSet, vous découvrirez d'étranges paramètres de type, de la forme E extends Enum<E>. Ceci signifie simplement que "E est un type énuméré". Tous les types énumérés étendent la classe générique Enum. Weekday, par exemple, étend Enum<Weekday>.

Cartes de hachage d'identité

Le JDK 1.4 ajoute une autre classe IdentityHashMap pour un autre objectif assez spécialisé, où les valeurs de hachage pour les clés ne doivent pas être calculées par la méthode hashCode mais par la méthode System.identityHashCode. C'est la méthode utilisée par Object.hashCode pour calculer un code de hachage à partir de l'adresse mémoire de l'objet. De même, pour la comparaison des objets, IdentityHashMap utilise ==, et non equals.

En d'autres termes, différents objets de clé sont considérés comme distincts, même s'ils ont un contenu égal. Cette classe est utile pour implémenter des algorithmes object traversal (comme la sérialisation d'objets), dans lesquels vous souhaitez effectuer le suivi des objets auxquels on a déjà appliqué traversed.

java.util.WeakHashMap<K, V> 1.2

- WeakHashMap()
- WeakHashMap(int initialCapacity)
- WeakHashMap(int initialCapacity, float loadFactor)

Construisent une carte de hachage vide avec la capacité et le facteur de charge spécifiés.

java.util.LinkedHashMap<E> 1.4

- LinkedHashMap()

- `LinkedHashSet(int initialCapacity)`
 - `LinkedHashSet(int initialCapacity, float loadFactor)`
- Construisent un set de hachage lié et vide, avec la capacité et le facteur de charge spécifiés.

java.util.LinkedHashMap<K, V> 1.4

- `LinkedHashMap()`
 - `LinkedHashMap(int initialCapacity)`
 - `LinkedHashMap(int initialCapacity, float loadFactor)`
 - `LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)`
- Construisent une carte de hachage liée vide avec la capacité, le facteur de charge et l'ordre spécifiés. Le paramètre `accessOrder` vaut `true` pour l'ordre d'accès, `false` pour l'ordre d'insertion.
- `protected boolean removeEldestEntry(Map.Entry<K, V> eldest)`
- Doit être surchargé pour renvoyer `true` si vous souhaitez supprimer l'entrée la plus ancienne. Le paramètre le plus ancien correspond à l'entrée dont on envisage la suppression. Cette méthode est appelée après qu'une entrée a été ajoutée à la carte. L'implémentation par défaut renvoie `false`, les anciens éléments ne sont pas supprimés par défaut. Vous pouvez toutefois redéfinir cette méthode pour renvoyer de manière sélective `true`, par exemple si l'entrée la plus ancienne répond à une certaine condition ou si la carte dépasse une certaine taille.

java.util.EnumSet<E extends Enum<E>> 5.0

- `static <E extends Enum<E>> EnumSet<E> allOf(Class<E> enumType)`
- Renvoie un set contenant toutes les valeurs du type énuméré donné.
- `static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> enumType)`
- Renvoie un set vide capable de contenir des valeurs du type énuméré donné.
- `static <E extends Enum<E>> EnumSet<E> range(E from, E to)`
- Renvoie un set contenant toutes les valeurs comprises entre `from` et `to` (compris).
- `static <E extends Enum<E>> EnumSet<E> of(E value)`
 - `static <E extends Enum<E>> EnumSet<E> of(E value, E... values)`
- Renvoient un set contenant les valeurs données.

java.util.EnumMap<K extends <K>, V> 5.0

- `EnumMap(Class<K> keyType)`
- Construit une carte vide dont les clés ont le type donné.

java.util.IdentityHashMap<K, V> 1.4

- `IdentityHashMap()`
 - `IdentityHashMap(int expectedMaxSize)`
- Construisent une carte de hachage d'identité vide dont la capacité est la plus petite puissance de 2 dépassant `1,5 x expectedMaxSize` (le paramètre par défaut pour `expectedMaxSize` vaut 21).

java.lang.System 1.0

- `static int identityHashCode(Object obj) 1.1`
- Renvoie le même code de hachage (dérivé de l'adresse mémoire de l'objet) que `Object.hashCode` calcule, même si la classe à laquelle appartient `obj` a redéfini la méthode `hashCode`.

La structure des collections

Les classes sont regroupées dans une structure appelée *cadre* (ou *framework*), qui constitue la base commune pour créer des fonctionnalités avancées. Un cadre contient des superclasses renfermant des fonctionnalités pratiques, des méthodes et des mécanismes. L'utilisateur d'un cadre crée des sous-classes pour étendre les fonctionnalités sans devoir réinventer les mécanismes de base. Par exemple, Swing est un cadre pour les interfaces utilisateur.

La bibliothèque de collections Java forme un cadre pour des classes de collections. Ce cadre définit un certain nombre d'interfaces et de classes abstraites pour des implémentations de collections (voir Figure 2.10), et il propose certains mécanismes, comme un protocole d'itération. Vous pouvez vous servir des classes de collections sans connaître grand-chose au cadre. C'est exactement ce que nous avons fait dans les sections précédentes. En revanche, si vous souhaitez implémenter des algorithmes généraux qui fonctionnent sur plusieurs types de collections, ou si vous voulez créer un nouveau type de collection, il vaut mieux comprendre le fonctionnement du cadre.

Il existe deux interfaces fondamentales pour les collections : `Collection` et `Map`. Des éléments peuvent être insérés dans une collection avec la méthode suivante :

```
boolean add(E element)
```

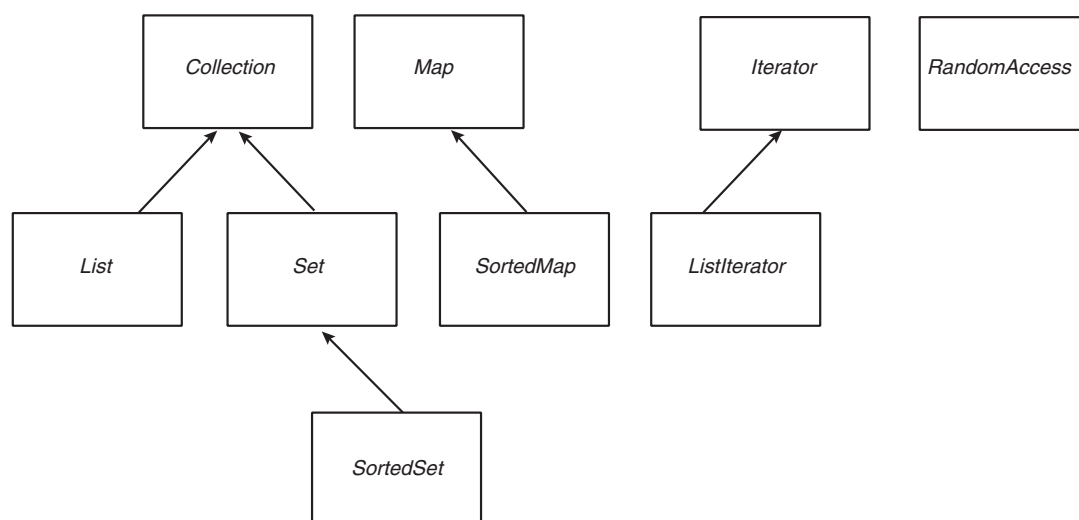


Figure 2.10

Les interfaces du cadre de collections.

Les cartes renferment des paires clé / valeur, et la méthode `put` sert à les ajouter dans une carte :

```
V put(K key, V value)
```

Pour lire les éléments d'une collection, il faut les parcourir avec un itérateur. Pour lire les valeurs d'une carte, il faut avoir recours à la méthode `get` :

```
V get(K key)
```

Une `List` est une *collection triée*. Les éléments sont ajoutés à une position particulière du conteneur. Un objet peut être inséré à la position adéquate de deux manières : par un indice entier ou par un

itérateur de liste. L'interface `List` définit des méthodes pour accéder à n'importe quelle donnée d'une liste :

```
void add(int index, E element)
E get(int index)
void remove(int index)
```

Comme nous l'avons déjà indiqué, l'interface `List` fournit ces méthodes d'accès aléatoire qu'elles soient ou non efficaces pour une implémentation particulière. Pour permettre d'éviter de réaliser des opérations d'accès aléatoires coûteuses, le JDK 1.4 introduit une interface de balisage, `RandomAccess`. Cette interface ne possède pas de méthodes, mais vous pouvez l'utiliser pour tester si une collection particulière prend en charge un accès aléatoire efficace :

```
if (c instanceof RandomAccess)
{
    utiliser un algorithme d'accès aléatoire
}
else
{
    utiliser un algorithme d'accès séquentiel
}
```

Les classes `ArrayList` et `Vector` implémentent l'interface `RandomAccess`.

INFO

D'un point de vue théorique, il aurait été intéressant de posséder une interface `Array` séparée qui étendrait l'interface `List` et qui déclarerait des méthodes d'accès aléatoires. Si cette interface existait, les algorithmes nécessitant ce type d'accès se serviraient de paramètres `Array`, et il vous serait alors impossible de les appliquer accidentellement à des collections pour lesquelles ce type d'accès n'est pas optimisé. Quoi qu'il en soit, les concepteurs du cadre des collections ont choisi de ne pas définir d'interface séparée. Ils voulaient en effet conserver un petit nombre d'interfaces dans la bibliothèque. De plus, ils souhaitaient éviter de prendre une attitude paternaliste envers les programmeurs. Vous êtes donc libre de passer une liste chaînée à des algorithmes accédant à des données par leur indice. Vous devez juste savoir que ce genre de technique n'est pas très rapide. L'interface `ListIterator` définit une méthode pour ajouter un élément juste avant la position de l'itérateur :

```
void add(E element)
```

Pour lire et supprimer des éléments à une position particulière, il suffit d'utiliser les méthodes `next` et `remove` de l'interface `Iterator`.

L'interface `Set` est identique à l'interface `Collection`, mais le comportement des méthodes y est défini de manière plus précise. La méthode `add` d'un set doit rejeter les valeurs doubles. La méthode `equals` d'un set doit être définie de telle sorte que deux sets sont identiques s'ils possèdent les mêmes éléments, mais pas nécessairement disposés dans le même ordre. La méthode `hashCode` doit être définie pour que deux sets possédant les mêmes éléments soient associés au même code de hachage.

Pourquoi définir une interface séparée si les signatures des méthodes sont les mêmes ? Conceptuellement, toutes les collections ne sont pas des sets. La définition d'une interface `Set` permet aux programmeurs d'écrire des méthodes qui n'acceptent que des sets.

En somme, les interfaces `SortedSet` et `SortedMap` mettent en évidence l'objet de comparaison utilisé pour trier les éléments, et elles définissent des méthodes pour obtenir des vues des sous-ensembles des collections. Nous reviendrons sur ces vues dans la prochaine section.

116 Au cœur de Java 2 - Fonctions avancées

Passons maintenant des interfaces aux classes qui les implémentent. Nous avons déjà vu que les interfaces de collections possèdent quelques méthodes qui peuvent être implémentées très simplement à partir d'un nombre plus important de méthodes fondamentales. Les classes abstraites fournissent certaines implémentations de ces routines :

```
AbstractCollection
AbstractList
AbstractSequentialList
AbstractSet
AbstractQueue
AbstractMap
```

Si vous implémentez votre propre classe de collection, vous voudrez probablement étendre l'une de ces classes pour récupérer les implémentations de ses opérations.

La bibliothèque Java fournit des classes concrètes :

```
LinkedList
ArrayList
HashSet
TreeSet
PriorityQueue
HashMap
TreeMap
```

La Figure 2.11 montre les relations entre ces classes.

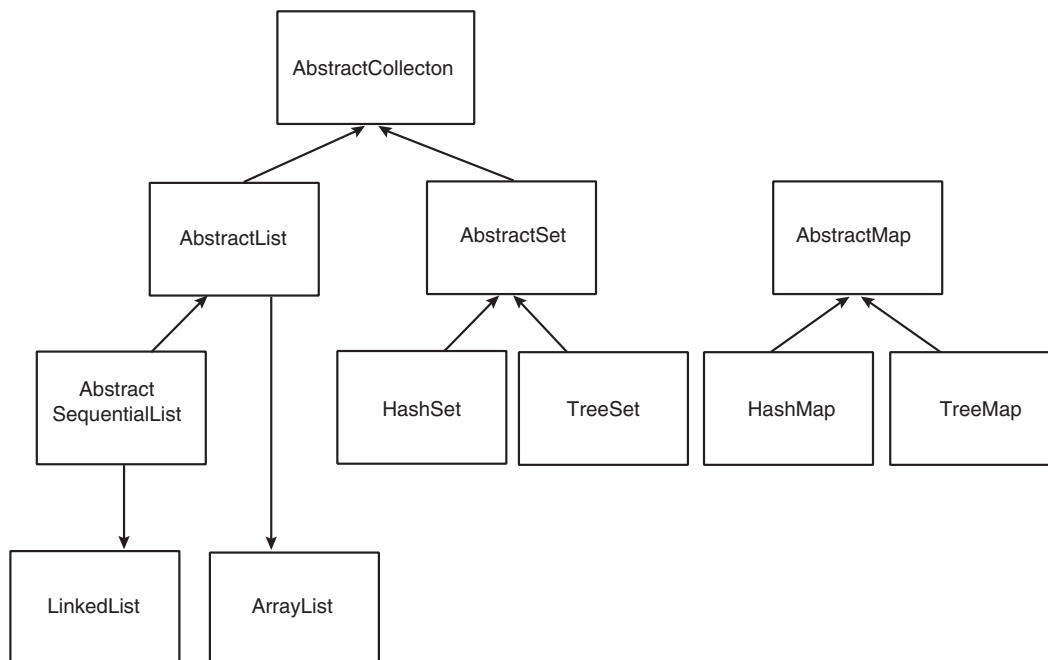


Figure 2.11

Les classes du cadre des collections.

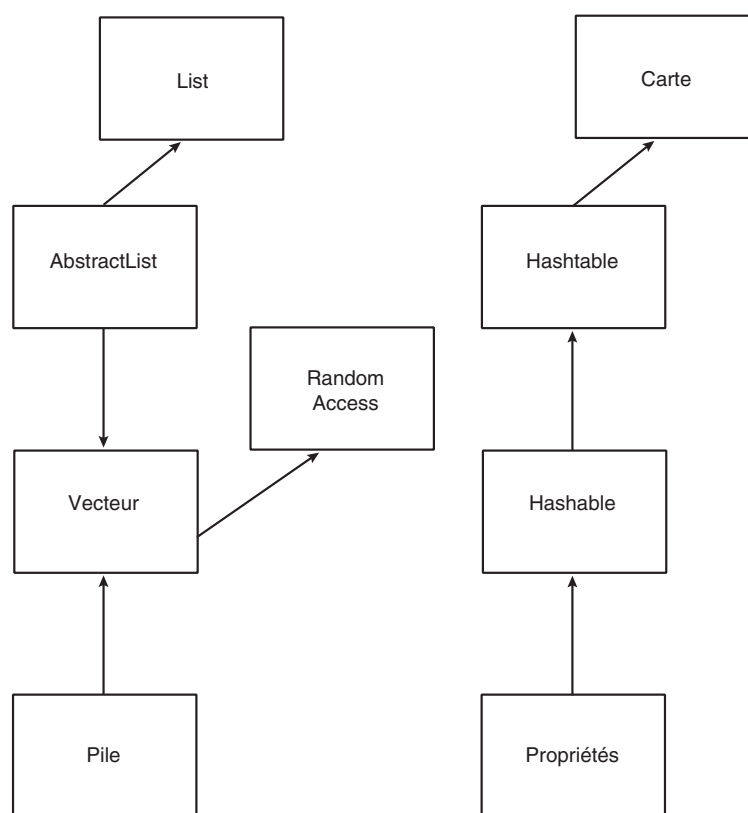
Finalement, il existe un certain nombre de classes conteneur "anciennes" qui sont présentes depuis le JDK 1.0, avant même qu'il n'y ait un cadre de collections :

Vector
Stack
Hashtable
Properties

Elles ont été intégrées dans le cadre des collections (voir Figure 2.12). Nous reviendrons sur ces classes un peu plus loin dans ce chapitre.

Figure 2.12

Les anciennes classes du cadre des collections.



Les vues et les emballages

Si vous regardez les Figures 2.10 et 2.11, vous trouverez peut-être qu'il est exagéré d'avoir de nombreuses interfaces et classes abstraites pour implémenter un nombre modeste de collections concrètes. Mais ces figures n'expliquent pas tout. En ayant recours à des *vues*, vous pouvez obtenir d'autres objets qui implémentent les interfaces `Collection` ou `Map`. Vous en avez déjà vu un exemple avec la méthode `keySet` des classes de cartes. Au premier abord, il apparaît que la méthode crée un nouveau set, le remplit avec toutes les clés de la carte, puis renvoie le set. Mais ce n'est pas exactement ce qui se passe. En fait, la méthode `keySet` renvoie un objet d'une classe qui implémente

118 Au cœur de Java 2 - Fonctions avancées

l'interface `Set` et dont les méthodes permettent de manipuler la carte d'origine. Ce type de collection est appelé une *vue*.

La technique des vues possède un certain nombre d'applications pratiques dans le cadre des collections. Nous verrons ces applications dans les sections suivantes.

Emballages de collection légers

La méthode statique `asList` de la classe `Arrays` renvoie un emballage `List` autour d'un tableau Java brut. Cette méthode vous permet de passer le tableau à une méthode qui attend un argument de liste ou de collection. Par exemple :

```
Card[] cardDeck = new Card[52];  
.  
.  
.  
List<Card> cardList = Arrays.asList(cardDeck);
```

L'objet renvoyé n'est pas un `ArrayList`, c'est un objet *vue* avec des méthodes `get` et `set` qui accèdent au tableau sous-jacent. Toutes les méthodes capables de modifier la taille du tableau (comme `add` et la méthode `remove` de l'itérateur associé) déclenchent une `UnsupportedOperationException`.

Depuis le JDK 5.0, la méthode `asList` est déclarée comme ayant un nombre variable d'arguments. Au lieu de passer un tableau, vous pouvez passer des éléments individuels. Par exemple :

```
List<String> names = Arrays.asList("Claire", "Julien", "Antoine");
```

L'appel de méthode doit être totalement terminé avant qu'un autre thread ne puisse appeler une autre méthode.

```
Collections.nCopies(n, anObject)
```

renvoie un objet inaltérable qui implémente l'interface `List` et donne l'illusion de posséder `n` éléments, chacun apparaissant comme un `anObject`.

Par exemple, l'appel suivant crée une liste contenant 100 chaînes, toutes définies sur "DEFAULT" :

```
List<String> settings = Collections.nCopies(100, "DEFAULT");
```

Le stockage se révèle peu coûteux, l'objet n'est stocké qu'une fois. C'est une bonne application de la technique d'affichage.

INFO

La classe `Collections` contient un certain nombre de méthodes commodes dont les paramètres ou les valeurs renvoyées constituent des collections. Attention à ne pas les confondre avec l'interface `Collection`.

L'appel de méthode

```
Collections.singleton(anObject)
```

renvoie un objet *vue* qui implémente l'interface `Set` (à la différence de `ncopies`, qui produit une `List`). L'objet renvoyé implémente un set d'éléments uniques inaltérable, sans avoir la charge de la structure de données. Les méthodes `singletonList` et `singletonMap` se comportent de la même manière.

Sous-ensembles

Vous pouvez créer des vues de sous-ensembles pour un certain nombre de collections. Par exemple, supposons que vous ayez une liste `staff` et que vous vouliez en extraire les éléments 10 à 19. Vous pouvez vous servir de la méthode `subList` pour obtenir une vue dans le sous-ensemble de la liste.

```
List group2 = staff.subList(10, 20);
```

Le premier indice est inclusif et le second exclusif, exactement comme les paramètres de l'opération `substring` de la classe `String`.

N'importe quelle opération peut être appliquée à un sous-ensemble, et elle reflète automatiquement la liste entière. Par exemple, vous pouvez effacer entièrement un sous-ensemble :

```
group2.clear(); // réduction de staff
```

Les éléments sont automatiquement effacés de la liste `staff` et `group2` est vide.

Pour les sets et les cartes triés, il faut se servir de l'ordre relatif et pas de la position de l'élément pour former un sous-ensemble. L'interface `SortedSet` déclare trois méthodes :

```
subSet(from, to)
headSet(to)
tailSet(from)
```

Celles-ci renvoient les sous-ensembles de tous les éléments supérieurs ou égaux à l'argument `from`, et strictement inférieurs à `to`. Pour les cartes triées, les méthodes similaires

```
subMap(from, to)
headMap(to)
tailMap(from)
```

renvoient des vues de la carte composées de toutes les entrées dont les *clés* se trouvent dans l'ensemble spécifié.

Vues non modifiables

La classe `Collections` possède des méthodes qui produisent des vues non modifiables de collections. Ces vues ajoutent à une collection existante une vérification se produisant en cours d'exécution. Si une tentative de modification de la collection est détectée, une exception est déclenchée et la collection demeure inchangée.

Les vues non modifiables s'obtiennent par six méthodes :

```
Collections.unmodifiableCollection
Collections.unmodifiableList
Collections.unmodifiableSet
Collections.unmodifiableSortedSet
Collections.unmodifiableMap
Collections.unmodifiableSortedMap
```

Chaque méthode est définie de manière à fonctionner sur une interface. `Collection.unmodifiableList`, par exemple, fonctionne avec un `ArrayList`, un `LinkedList` ou toute autre classe qui implémente l'interface `List`.

120 Au cœur de Java 2 - Fonctions avancées

Par exemple, supposons que vous souhaitiez qu'une partie de votre programme puisse examiner (mais pas modifier) le contenu d'une collection. Voici un exemple de ce que vous pourriez envisager :

```
List<String> staff = new LinkedList<String>();  
...  
lookAt(new Collection.unmodifiableList(staff));
```

La méthode `Collections.unmodifiableList` renvoie un objet d'une classe implémentant l'interface `List`. Sa méthode d'accès recherche des valeurs dans la collection `staff`. Naturellement, la méthode `lookAt` peut appeler toutes les méthodes de l'interface `List`, et pas seulement les méthodes d'accès. Mais toutes les méthodes de modification (comme `add`) ont été redéfinies pour lancer une `UnsupportedOperationException` au lieu de propager l'appel à la collection sous-jacente.

La vue non modifiable ne rend pas la collection inaltérable. Vous pouvez toujours la modifier *via* sa référence initiale (`staff`, dans notre cas). Vous pouvez toujours appeler des méthodes de modification sur les éléments de la collection.

ATTENTION

La méthode `unmodifiableCollection` (ainsi que les méthodes `synchronizedCollection` et `checkedCollection`, sur lesquelles nous reviendrons un peu plus tard dans cette section) renvoie une collection dont la méthode `equals` n'invoque pas la méthode `equals` de la collection sous-jacente. En fait, elle hérite de la méthode `equals` de la classe `Object`, qui se contente de tester si deux objets sont identiques. Si vous transformez un set ou une liste en simple collection, vous ne pouvez plus tester l'identité de leur contenu. La vue se comporte de cette manière parce qu'un test d'égalité n'est pas bien défini à ce niveau de hiérarchie. La vue traite la méthode `hashCode` de la même manière.

Cependant, les classes `unmodifiableSet` et `unmodifiableList` ne cachent pas les méthodes `equals` et `hashCode` des collections sous-jacentes.

Vues synchronisées

Si vous accédez à une collection depuis plusieurs threads, il est très important que la collection ne soit pas accidentellement endommagée. Par exemple, il serait désastreux qu'un thread tente d'ajouter un élément dans une table de hachage alors qu'un autre thread essaierait d'en réorganiser les éléments.

Au lieu d'implémenter des classes de collection compatibles avec les threads, les concepteurs de la bibliothèque ont créé un mécanisme qui génère des collections ordinaires compatibles avec les threads. Par exemple, la classe statique `synchronizedMap` de la classe `Collections` peut transformer n'importe quelle carte en carte possédant des méthodes d'accès synchronisées :

```
HashMap<String, Employee> hashMap = new HashMap<String, Employee>();  
Map<String, Employee> = Collections.synchronizedMap(hashMap);
```

Vous pouvez dorénavant accéder à l'objet map depuis plusieurs threads. Les méthodes comme `get` et `put` sont synchronisées, c'est-à-dire que chaque appel à l'une d'elles doit être entièrement terminé avant qu'un autre thread puisse appeler une autre méthode.

Vérifiez qu'aucun thread n'accède à la structure de données en passant par les méthodes d'origine non synchronisées. Le moyen le plus simple pour s'en assurer consiste à ne stocker aucune référence à l'objet d'origine :

```
map = Collections.synchronizedMap(new HashMap<String, Employee>());
```

Sachez que les vues ne synchronisent que les méthodes de la *collection*. Si vous utilisez un itérateur, vous devez acquérir manuellement le verrou sur l'objet de collection. Par exemple :

```
synchronized (map)
{
    Iterator<String> iter = map.keySet().iterator();
    while (iter.hasNext()) . . .
}
```

Vous devez utiliser le même code si vous utilisez une boucle "for each" car la boucle utilise un itérateur. Sachez que l'itérateur échouera avec une *ConcurrentModificationException* si un autre thread modifie la collection au cours de l'itération. La synchronisation reste nécessaire pour que la modification simultanée puisse être correctement détectée.

En termes pratiques, les emballages de synchronisation ont une utilité limitée. Il vaut généralement mieux utiliser les collections définies dans le paquetage *java.util.concurrent* (voir Chapitre 1 pour en savoir plus). En particulier, la carte *ConcurrentHashMap* a été soigneusement implémentée de sorte que plusieurs threads puissent y accéder sans se bloquer les uns les autres, à condition qu'ils accèdent à différents seaux.

Vues sous contrôle

Le JDK 5.0 ajoute un set de vues "sous contrôle" destinées à déboguer un problème survenant avec les types génériques. Comme nous l'avons vu au Chapitre 13 du Volume 1, il est en fait possible de faire entrer des éléments du mauvais type dans une collection générique. Par exemple :

```
ArrayList<String> strings = new ArrayList<String>();
ArrayList rawList = strings; // un avertissement uniquement, non
                             // une erreur, pour des raisons de
                             // compatibilité avec le code existant
rawList.add(new Date()); // maintenant, strings contient un objet Date !
```

La commande *add* erronée n'est pas détectée au moment de l'exécution. Une exception de transtypage de classe surviendra plus tard lorsqu'une autre partie du code appellera *get* et transtypera le résultat sur un *String*.

Une vue sous contrôle peut détecter ce problème. Définissez

```
List<String> safeStrings = Collections.checkedList(strings, String.class);
```

La méthode *add* de la vue vérifie que l'objet inséré appartient à la classe donnée et déclenche immédiatement une *ClassCastException* si ce n'est pas le cas. Avantage, l'erreur est signalée à l'emplacement correct :

```
ArrayList rawList = safeStrings;
rawList.add(new Date()); // La liste sous contrôle déclenche
                          // une ClassCastException
```

ATTENTION

Les vues sous contrôle sont limitées par les vérifications d'exécution que la machine virtuelle peut réaliser. Si vous avez par exemple un *ArrayList<Pair<String>>*, vous ne pouvez pas le protéger de l'insertion d'un *Pair<Date>* puisque la machine virtuelle possède une seule classe *Pair* "brute".

122 Au cœur de Java 2 - Fonctions avancées

Une dernière note sur les opérations optionnelles

Les vues possèdent généralement quelques restrictions : elles peuvent être accessibles uniquement en lecture, leur taille n'est parfois pas modifiable. D'autres encore peuvent supprimer l'un de leurs éléments, mais ne peuvent pas en insérer de nouveaux, comme pour une vue de clé d'une carte. Une vue restreinte déclenche une `UnsupportedOperationException` si vous tentez une opération inappropriée.

Dans la documentation de l'API pour les interfaces de collection et d'itérateur, certaines méthodes sont décrites comme des "opérations optionnelles". Ceci semble contredire la notion même d'interface, car c'est bien son but que de fournir les méthodes qu'une classe *doit* implémenter. Cet agencement est en fait insatisfaisant du point de vue théorique. Une meilleure solution aurait consisté à concevoir des interfaces séparées pour les vues en lecture seule et celles qui ne peuvent pas modifier la taille d'une collection. Mais cela aurait triplé le nombre d'interfaces, ce que les concepteurs de la bibliothèque ont jugé inacceptable.

Pourquoi ne pas étendre la technique des méthodes "optionnelles" à vos propres interfaces ? Nous pensons que cela n'est pas conseillé. Bien que les collections soient utilisées très fréquemment, le style de programmation inhérent à leur utilisation ne correspond pas forcément à tous les types de problèmes. Les concepteurs d'une bibliothèque de classes de collections doivent généralement résoudre un ensemble complexe de conflits. Les utilisateurs veulent qu'une bibliothèque soit simple à apprendre, facile d'emploi, suffisamment générale, sans incohérences, et en même temps aussi efficace qu'un algorithme optimisé à la main. Il est parfaitement impossible d'atteindre simultanément tous ces buts, voire de s'en rapprocher. En fait, vous rencontrerez rarement des problèmes aussi complexes lorsque vous écrirez vos propres programmes. Vous devriez trouver des solutions qui ne sont pas fondées sur des mesures aussi extrêmes que des opérations d'interfaces "optionnelles".

java.util.Collections 1.2

- `static <E> Collection unmodifiableCollection(Collection<E> c)`
- `static <E> List unmodifiableList(List<E> c)`
- `static <E> Set unmodifiableSet(Set<E> c)`
- `static <E> SortedSet unmodifiableSortedSet(SortedSet<E> c)`
- `static <K, V> Map unmodifiableMap(Map<K, V> c)`
- `static <K, V> SortedMap unmodifiableSortedMap(SortedMap<K, V> c)`

Construisent une vue de la collection dont les méthodes de modification peuvent déclencher une `UnsupportedOperationException`.

- `static <E> Collection<E> synchronizedCollection(Collection<E> c)`
- `static <E> List<E> synchronizedList(List<E> c)`
- `static <E> Set<E> synchronizedSet(Set<E> c)`
- `static <E> SortedSet<E> synchronizedSortedSet(SortedSet<E> c)`
- `static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)`
- `static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)`

Construisent une vue de la collection dont les méthodes sont synchronisées.

- `static <E> Collection checkedCollection(Collection<E> c, Class<E> elementType)`
- `static <E> List checkedList(List<E> c, Class<E> elementType)`
- `static <E> Set checkedSet(Set<E> c, Class<E> elementType)`
- `static <E> SortedSet checkedSortedSet(SortedSet<E> c, Class<E> elementType)`

- `static <K, V> Map checkedMap(Map<K, V> c, Class<K> keyType, Class<V> valueType)`
- `static <K, V> SortedMap checkedSortedMap(SortedMap<K, V> c, Class<K> keyType, Class<V> valueType)`
Construisent une vue de la collection dont les méthodes déclenchent une `ClassCastException` en cas d'insertion d'un élément d'un mauvais type.
- `static <E> List<E> nCopies(int n, E value)`
- `static <E> Set<E> singleton(E value)`
Construisent une vue de l'objet soit comme une liste non modifiable comprenant `n` éléments identiques, soit comme un set contenant un seul élément.

java.util.Arrays 1.2

- `static <E> List<E> asList(E... array)`
Renvoie une vue en liste des éléments d'un tableau, modifiable mais non redimensionnable.

java.util.List<E> 1.2

- `List<E> subList(int firstIncluded, int firstExcluded)`
Renvoie une vue de liste à partir des éléments compris entre deux positions.

java.util.SortedSet<E> 1.2

- `SortedSet<E> subSet(E firstIncluded, E firstExcluded)`
- `SortedSet<E> headSet(E firstExcluded)`
- `SortedSet<E> tailSet(E firstIncluded)`
Renvoient une vue des éléments compris entre deux positions.

java.util.SortedMap<K, V> 1.2

- `SortedMap<K, V> subMap(K firstIncluded, K firstExcluded)`
- `SortedMap<K, V> headMap(K firstExcluded)`
- `SortedMap<K, V> tailMap(K firstIncluded)`
Renvoient une vue de la carte dont les entrées sont comprises entre deux positions.

Les opérations de masse

Jusqu'à maintenant, la plupart de nos exemples se servaient d'un itérateur pour parcourir un par un les éléments d'une collection. Mais il est possible d'éviter de parcourir tous ces éléments en ayant recours à l'une des *opérations de masse* (ou *bulk operation*) de la bibliothèque.

Supposons que vous cherchiez l'*intersection* de deux ensembles, c'est-à-dire les éléments communs à chacun des deux ensembles. Commençons par créer un nouvel ensemble destiné à contenir le résultat :

```
Set<String> result = new HashSet<String>(a);
```

Nous utilisons ici le fait que toutes les collections possèdent un constructeur dont les paramètres sont une autre collection renfermant les valeurs d'initialisation.

Appelons maintenant la méthode `retainAll` :

```
result.retainAll(b);
```

124 Au cœur de Java 2 - Fonctions avancées

Cette méthode conserve tous les éléments qui sont aussi dans b. Vous venez de déterminer l'intersection sans programmer de boucle.

Pourquoi ne pas continuer sur cette lancée et appliquer une opération de masse à une *vue* ? Supposons par exemple que vous disposiez d'une carte associant des numéros de salariés à des objets *employee* et que vous possédiez un ensemble de numéros de salariés qui vont être licenciés.

```
Map<String, Employee> staffMap = . . . ;  
Set<String> terminatedIDs = . . . ;
```

Il suffit de créer un ensemble de clés et de supprimer tous les numéros de salariés licenciés.

```
staffMap.keySet().removeAll(terminatedIDs);
```

Comme l'ensemble de clés constitue une vue de la carte, les clés et les noms de salariés associés sont automatiquement supprimés de la carte.

En utilisant une vue d'un sous-ensemble, vous pouvez restreindre les opérations de masse à des sous-ensembles et à des sous-listes. Par exemple, supposons que vous vouliez ajouter les dix premiers éléments d'une liste dans un autre conteneur. Il suffit de former une sous-liste correspondant aux dix premiers éléments :

```
relocated.addAll(staff.subList(0, 10));
```

Le sous-ensemble peut aussi être la cible d'une opération de mutation :

```
staff.subList(0, 10).clear();
```

Conversion entre collections et tableaux

Comme une grande partie de l'API de la plate-forme Java a été conçue avant l'introduction du cadre des collections, vous aurez parfois besoin d'adapter les anciens tableaux en collections plus récentes.

Si vous possédez un tableau, vous devez le convertir en collection. L'emballage `Arrays.asList` sert à cela. Par exemple :

```
String[] values = . . . ;  
HashSet<String> staff = new HashSet<String>(Arrays.asList(values));
```

L'obtention d'un tableau à partir d'une collection est légèrement plus astucieuse. Naturellement, vous pouvez vous servir de la méthode `toArray` :

```
Object[] values = staff.toArray();
```

Mais il en résulte un tableau d'*objets*. Même si vous savez que votre collection contient des objets d'un type spécifique, vous ne pouvez pas recourir à une conversion de type :

```
String[] values = (String[]) staff.toArray(); // Erreur !
```

Le tableau renvoyé par la méthode `toArray` a été créé comme un tableau `Object[]`, et son type ne peut pas être modifié. Pour contourner ce problème, vous utilisez une variante de la méthode `toArray`. Passez-lui un tableau de longueur nulle et du type qui vous intéresse. Le tableau créé est du même type que le type spécifié :

```
String[] values = staff.toArray(new String[0]);
```

Si vous le souhaitez, vous pouvez construire le tableau, de sorte qu'il ait la bonne taille :

```
staff.toArray(new String[staff.size()]);
```

Dans ce cas, aucun nouveau tableau n'est créé.

INFO

Vous vous demandez peut-être pourquoi vous ne pouvez pas passer directement un objet `Class` (comme `String.class`) à la méthode `toArray`. Cette méthode est à double emploi, c'est-à-dire qu'elle peut à la fois remplir un tableau existant (en supposant qu'il est assez grand) et créer un nouveau tableau.

`java.util.Collection<E>` 1.2

- `<T> T[] toArray(T[] array)`

Compare le tableau passé en paramètre et la taille de la collection. Si le tableau est plus grand que la collection, cette méthode ajoute tous les éléments de la collection dans le tableau, suivi d'un terminateur `null`, et renvoie le tableau. Si la longueur du tableau est égale à la taille de la collection, la méthode ajoute tous les éléments de la collection dans le tableau, mais n'ajoute pas le terminateur `null`. S'il n'y a pas assez de place dans le tableau, la méthode crée un nouveau tableau *du même type que le tableau*, et le remplit avec tous les éléments de la collection.

Extension du cadre

Le cadre des collections contient toutes les structures de données dont ont besoin les programmeurs. Mais, si vous avez besoin d'une structure de données spécialisée, vous pouvez facilement étendre le cadre. A titre d'exemple, nous implémentons une queue de tableau circulaire (voir Exemple 2.6).

Le cadre contient une classe `AbstractQueue` qui implémente toutes les méthodes de l'interface `Queue`, à l'exception de `size`, `offer`, `poll`, `peek` et `iterator`. Nous utilisons cette classe et n'implémentons que les méthodes manquantes. Nous héritons alors automatiquement de toutes les méthodes restantes de la classe `AbstractQueue`.

La plupart des méthodes sont simples, à l'exception d'`iterator`. Nous implémentons l'itérateur de queue sous la forme d'une classe interne. Ceci permet aux méthodes d'itération d'accéder aux champs de l'objet queue englobant, c'est-à-dire l'objet qui a construit l'itérateur (comme nous l'avons vu au Chapitre 5 du Volume 1, chaque objet d'une classe interne non statique possède une référence à l'objet de classe externe qui l'a créé).

Vous remarquerez également la protection contre la modification simultanée. La queue effectue le comptage de toutes les modifications. A la construction de l'itérateur, elle crée une copie de ce comptage. Dès que l'itérateur est utilisé, elle vérifie que le compte concorde toujours. Si ce n'est pas le cas, elle déclenche une `ConcurrentModificationException`.

Exemple 2.6 : `CircularArrayQueueTest.java`

```
import java.util.*;

public class CircularArrayQueueTest
{
    public static void main(String[] args)
    {
        Queue<String> q = new CircularArrayQueue<String>(5);
        q.add("Annie");
        q.add("Bernard");
        q.add("Charles");
        q.add("Didier");
        q.add("Emile");
        q.remove();
        q.add("Fifi");
    }
}
```

126 Au cœur de Java 2 - Fonctions avancées

```

        q.remove();
        for (String s : q) System.out.println(s);
    }
}

/**
 * Une collection bornée, du style premier entré, premier sorti.
 */
class CircularArrayQueue<E> extends AbstractQueue<E>
{
    /**
     * Construit une queue vide.
     * @param capacity la capacité maximum de la queue
     */
    public CircularArrayQueue(int capacity)
    {
        elements = (E[]) new Object[capacity];
        count = 0;
        head = 0;
        tail = 0;
    }

    public boolean offer(E newElement)
    {
        assert newElement != null;
        if (count < elements.length)
        {
            elements[tail] = newElement;
            tail = (tail + 1) % elements.length;
            count++;
            modcount++;
            return true;
        }
        else
            return false;
    }

    public E poll()
    {
        if (count == 0) return null;
        E r = elements[head];
        head = (head + 1) % elements.length;
        count--;
        modcount++;
        return r;
    }

    public E peek()
    {
        if (count == 0) return null;
        return elements[head];
    }

    public int size()
    {
        return count;
    }

    public Iterator<E> iterator()

```

```

    {
        return new QueueIterator();
    }

    private class QueueIterator implements Iterator<E>
    {
        public QueueIterator()
        {
            modcountAtConstruction = modcount;
        }

        public E next()
        {
            if (!hasNext()) throw new NoSuchElementException();
            E r = elements[(head + offset) % elements.length];
            offset++;
            return r;
        }

        public boolean hasNext()
        {
            if (modcount != modcountAtConstruction)
                throw new ConcurrentModificationException();
            return offset < elements.length;
        }

        public void remove()
        {
            throw new UnsupportedOperationException();
        }

        private int offset;
        private int modcountAtConstruction;
    }

    private E[] elements;
    private int head;
    private int tail;
    private int count;
    private int modcount;
}

```

Algorithmes

Les interfaces de collection générales possèdent un grand avantage : il suffit d'implémenter vos algorithmes une seule fois. Par exemple, considérons un algorithme simple qui calcule l'élément maximal d'une collection. Les programmeurs sont tentés par l'approche traditionnelle qui consiste à implémenter ce genre d'algorithme dans une boucle. Voici comment trouver le plus grand élément d'un tableau :

```

if (a.length == 0) throw new NoSuchElementException();
T largest = a[0];
for (int i = 1; i < a.length; i++)
    if (largest.compareTo(a[i]) < 0)
        largest = a[i];

```

128 Au cœur de Java 2 - Fonctions avancées

Naturellement, pour trouver l'élément maximal d'une liste de tableau, le code doit être légèrement modifié.

```
if (v.size() == 0) throw new NoSuchElementException();
T largest = v.get(0);
for (int i = 1; i < v.size(); i++)
    if (largest.compareTo(v.get(i)) < 0)
        largest = v.get(i);
```

Et pour une liste chaînée ? Comme vous ne disposez pas d'un accès aléatoire aux éléments d'une liste chaînée, vous pouvez vous servir d'un itérateur :

```
if (l.isEmpty()) throw new NoSuchElementException();
Iterator<T> iter = l.iterator();
T largest = iter.next();
while (iter.hasNext())
{
    T next = iter.next();
    if (largest.compareTo(next) < 0)
        largest = next;
}
```

Ces boucles sont plutôt pénibles à écrire et elles peuvent engendrer des erreurs. N'existe-t-il pas une méthode plus simple ? Est-ce que ces boucles fonctionnent correctement avec des conteneurs vides ? Avec des conteneurs ne contenant qu'un seul élément ? Vous n'avez probablement pas envie de tester et de déboguer ce code pour chaque cas particulier, mais vous voulez probablement aussi éviter d'implémenter l'une des méthodes lentes suivantes :

```
static <T extends Comparable> T max(T[] a)
static <T extends Comparable> T max(ArrayList<T> v)
static <T extends Comparable> T max(LinkedList<T> l)
```

C'est dans ce contexte que les interfaces de collection entrent en jeu. Imaginez une interface de collection *minimale* pour gérer efficacement l'algorithme. Les méthodes d'accès aléatoire comme `get` et `set` sont à considérer comme des méthodes de plus haut niveau qu'une simple itération. Comme nous l'avons vu dans le calcul de la valeur maximale d'une liste chaînée, ces méthodes d'accès aléatoire ne sont pas forcément nécessaires. Le calcul de l'élément maximal peut être effectué par une simple itération sur tous les éléments. Par conséquent, vous pouvez implémenter la méthode `max` pour prendre en compte *n'importe quel* objet qui implémente l'interface `Collection`.

```
public static <T extends Comparable> T max(Collection<T> c)
{
    if (c.isEmpty()) throw new NoSuchElementException();
    Iterator<T> iter = c.iterator();
    T largest = iter.next();
    while (iter.hasNext())
    {
        T next = iter.next();
        if (largest.compareTo(next) < 0)
            largest = next;
    }
    return largest;
}
```

Vous pouvez maintenant calculer la valeur maximale d'une liste chaînée, d'une liste de tableau ou d'un tableau avec une seule méthode.

Il s'agit d'un concept puissant. En fait, la bibliothèque standard du C++ possède des douzaines d'algorithmes pratiques, qui fonctionnent tous à partir d'une collection générale. La bibliothèque Java n'est pas aussi riche, mais elle contient les algorithmes essentiels : un tri, une recherche binaire, et d'autres algorithmes pratiques.

Trier et mélanger

Les vieux routiers de l'informatique se rappellent peut-être les cartes perforées qu'ils devaient utiliser et les algorithmes de tri qu'ils devaient programmer à la main. Heureusement, de nos jours, les algorithmes de tri font partie des bibliothèques standard de la plupart des langages de programmation, et Java n'y fait pas exception.

La méthode `sort` de la classe `Collections` sert à trier une collection qui implémente l'interface `List`.

```
List<String> staff = new LinkedList<String>();  
// remplissage de la collection . . . ;  
Collections.sort(staff);
```

Cette méthode part de l'hypothèse que les éléments de la liste implémentent l'interface `Comparable`. Si vous voulez trier une liste d'une autre manière, vous pouvez spécifier un objet `Comparator` en second argument. Nous avons abordé les comparateurs dans la section traitant des ensembles triés. Voici un exemple pour trier une liste d'éléments :

```
Comparator<Item> itemComparator = new  
    Comparator<Item>()  
    {  
        public int compare(Item a, Item b)  
        {  
            Return a.partNumber - b.partNumber  
        }  
    });  
Collections.sort(items, itemComparator);
```

Si vous souhaitez trier une liste par ordre *décroissant*, il suffit d'utiliser la méthode statique `Collections.reverseOrder()`. Elle renvoie un comparateur qui renvoie `b.compareTo(a)`. Par exemple,

```
Collections.sort(staff, Collections.reverseOrder())
```

trie les éléments de la liste `staff` par ordre décroissant, selon l'ordre fourni par la méthode `compareTo` du type d'élément. De même,

```
Collections.sort(items, Collections.reverseOrder(itemComparator))
```

inverse l'ordre d'`itemComparator`.

Vous vous demandez peut-être comment la méthode `sort` trie une liste. Typiquement, les algorithmes de tri que vous pourrez trouver dans des livres sur les algorithmes se servent de tableaux et accèdent à leurs éléments avec une méthode d'accès aléatoire. Mais nous savons maintenant que les accès aléatoires ne sont pas optimisés pour les listes. En fait, vous pouvez trier efficacement une liste grâce à un algorithme appelé *tri fusion* (voir, par exemple, *Algorithms in C++*, par Robert Sedgewick, Addison-Wesley, 1998, p. 366 à 369). Cependant, l'implémentation Java ne se sert pas de cet algorithme. Elle se contente de recopier tous les éléments de la liste dans un tableau, de trier le tableau et de recopier le tableau trié dans la liste d'origine.

130 Au cœur de Java 2 - Fonctions avancées

L'algorithme de tri fusion utilisé dans la bibliothèque de collections est légèrement plus lent que le *quick sort*, qui est l'algorithme retenu généralement pour trier les données les plus générales. Il présente pourtant un avantage majeur : il est *stable*, c'est-à-dire qu'il n'intervertit pas des éléments identiques. En quoi l'ordre des éléments identiques est-il important ? Voici un scénario classique : supposons que vous disposiez d'une liste d'employés que vous avez déjà triée par noms et que vous vouliez maintenant la trier par salaires. Que se passe-t-il pour les employés ayant le même salaire ? Avec un tri stable, le classement d'origine par noms est conservé. En d'autres termes, le résultat est une liste triée d'abord par salaires, puis par noms.

Comme les collections n'ont pas besoin d'implémenter toutes leurs méthodes "optionnelles", toutes les méthodes recevant des collections en paramètre doivent indiquer s'il est possible de passer sans danger une collection à un algorithme. Par exemple, il est évident que vous ne pouvez pas passer une liste `unmodifiableList` à l'algorithme `sort`. Quelle sorte de liste peut-on alors lui passer ? Si l'on s'en réfère à la documentation, la liste doit être modifiable, mais sa taille n'a pas besoin d'être modifiable.

Voici la définition de ces deux critères :

- Une liste est *modifiable* si elle supporte la méthode `set`.
- La taille d'une liste est *modifiable* si elle supporte les opérations `add` et `remove`.

La classe `Collections` possède un algorithme `shuffle` qui effectue le contraire d'un tri, c'est-à-dire qui permute aléatoirement les éléments d'une liste. Il faut lui passer en argument la liste à mélanger et un nombre aléatoire. Par exemple,

```
ArrayList<Card> cards = . . . ;
Collections.shuffle(cards);
```

Si vous fournissez une liste qui n'implémente pas l'interface `RandomAccess`, la méthode `shuffle` copie les éléments dans un tableau, mélange le tableau et recopie les éléments mélangés dans la liste. Le programme de l'Exemple 2.7 remplit une liste avec 49 objets `Integer` contenant les nombres 1 à 49. Puis il mélange la liste aléatoirement et sélectionne les six premières valeurs de la liste. Pour terminer, il trie les valeurs sélectionnées et les affiche.

Exemple 2.7 : *ShuffleTest.java*

```
import java.util.*;

/**
 * Ce programme montre les algorithmes de tri et de mélange
 * aléatoire.
 */
public class ShuffleTest
{
    public static void main(String[] args)
    {
        List<Integer> numbers = new ArrayList<Integer>();
        for (int i = 1; i <= 49; i++)
            numbers.add(i);
        Collections.shuffle(numbers);
        List<Integer> winningCombination = numbers.subList(0, 6);
        Collections.sort(winningCombination);
        System.out.println(winningCombination);
    }
}
```

java.util.Collections 1.2

- `static <T extends Comparable<? super T>> void sort(List<T> elements)`
Trie les éléments de la liste avec un algorithme de tri stable. La complexité de cet algorithme est $O(n \log n)$, où n correspond au nombre d'éléments de la liste.
- `static <T> void sort(List<T> elements, Comparator<? super T> c)`
Trie les éléments de la liste avec un algorithme de tri stable. La complexité de cet algorithme est $O(n \log n)$, où n correspond au nombre d'éléments de la liste.
- `static void shuffle(List<?> elements)`
- `static void shuffle(List<?> elements, Random r)`
Mélange aléatoirement les éléments de la liste. La complexité de cet algorithme est $O(n a(n))$, où n correspond à la longueur de la liste et où $a(n)$ est le temps moyen d'accès à un élément.
- `static <T> Comparator<T> reverseOrder()`
Renvoie un comparateur qui trie les éléments par ordre décroissant par rapport à la méthode `compareTo` de l'interface `Comparable`.
- `static <T> Comparator<T> reverseOrder(Comparator<T> comp)`
Renvoie un comparateur qui trie les éléments dans l'ordre inverse de celui donné par `comp`.

Recherche binaire

En principe, pour trouver un objet dans un tableau, il faut parcourir tous ses éléments jusqu'à trouver celui que vous cherchez. Cependant, lorsqu'un tableau est trié, vous pouvez commencer par examiner la valeur de l'élément situé au milieu du tableau et vérifier si cette valeur est supérieure à celle de l'objet que vous cherchez. Dans ce cas, il vous suffit de chercher l'objet dans la première moitié du tableau. Sinon, l'élément cherché se trouve dans la seconde moitié du tableau. Cette opération divise le problème en deux. Vous pouvez alors continuer à chercher l'objet en utilisant la même technique. Ainsi, si le tableau comprend 1 024 éléments, vous trouverez n'importe lequel de ses éléments en 10 étapes au maximum, alors qu'une recherche séquentielle aurait nécessité en moyenne 512 étapes si l'élément cherché fait bien partie du tableau, et il aurait fallu 1 024 étapes pour confirmer l'absence d'un élément dans ce tableau.

La recherche binaire (`binarySearch`) de la classe `Collections` implémente cet algorithme. Notons que la collection doit déjà être triée, sinon cet algorithme renvoie une réponse erronée. Pour trouver un élément, il suffit de fournir une collection qui doit implémenter l'interface `List` (voir la note ci-après) et l'élément à rechercher. Si la collection n'a pas été triée par l'élément `compareTo` de l'interface `Comparable`, il faut également fournir le comparateur utilisé.

```
i = Collections.binarySearch(c, element);  
i = Collections.binarySearch(c, element, comparator);
```

Si la valeur renvoyée par la recherche binaire est supérieure ou égale à zéro, cette valeur correspond à l'indice de l'élément trouvé. C'est-à-dire que `c.get(i)` est égal à l'élément au sens du comparateur. Si la valeur est négative, cela signifie que l'élément n'a pas pu être trouvé dans la collection. De plus, vous pouvez vous servir de la valeur renvoyée pour calculer l'emplacement où il *faudrait* insérer l'élément pour que la collection reste triée. Cet indice correspond à :

```
insertionPoint = -i - 1;
```

132 Au cœur de Java 2 - Fonctions avancées

Ce n'est pas simplement `-i`, car cela mènerait à des résultats ambigus si la valeur renvoyée était nulle. En d'autres termes, l'opération

```
if (i < 0)
    c.add(-i - 1, element);
```

ajoute l'élément à la bonne place.

Pour rester valable, une recherche binaire doit bénéficier d'un accès aléatoire aux données de la collection. Si vous devez parcourir tous les éléments jusqu'à la moitié du tableau pour trouver l'élément du milieu, l'avantage principal de la recherche binaire est perdu. Par conséquent, l'algorithme `binarySearch` se transforme en recherche linéaire si vous lui fournissez une liste chaînée.

INFO

Le JDK 1.3 ne possédait aucune interface séparée pour une collection triée avec un accès aléatoire efficace : la méthode `binarySearch` se sert d'une technique très grossière pour déterminer si le paramètre de la liste prolonge la classe `AbstractSequentialList`. Ceci a été résolu dans la version 1.4. Désormais, la méthode `binarySearch` vérifie si le paramètre de la liste implémente l'interface `RandomAccess`. Si c'est le cas, la méthode réalise une recherche binaire. Sinon, elle effectue une recherche linéaire.

java.util.Collections 1.2

- `static <T extends Comparable<? super T>> int binarySearch(List<T> elements, T key)`
- `static <T> int binarySearch(List<T> elements, T key, Comparator<? super T> c)`

Cherchent une clé dans une liste chaînée avec une recherche linéaire si les éléments étendent la classe `AbstractSequentialList`. Une recherche binaire est utilisée dans tous les autres cas. La complexité de ces méthodes est $O(a(n) \log n)$, où n est la longueur de la liste et où $a(n)$ correspond au temps moyen d'accès à un élément quelconque. Ces méthodes renvoient soit l'indice de la clé dans la liste, soit une valeur négative i si la clé ne fait pas partie de la liste. Dans ce cas, la clé devrait être insérée à la position `-i - 1` pour que la liste reste triée.

Algorithmes simples

La classe `Collections` contient plusieurs algorithmes simples, mais très utiles. Nous retrouvons notamment parmi ces algorithmes l'exemple du début de cette section, la recherche de la valeur maximale d'une collection. Les autres algorithmes couvrent des sujets aussi variés que la copie des éléments d'une liste dans une autre liste, le remplissage d'un conteneur avec une valeur constante, ou l'inversion d'une liste. Pourquoi fournir des algorithmes aussi simples dans une bibliothèque standard ? Il est probable que la plupart des programmeurs pourraient les implémenter facilement avec des boucles simples. Nous aimons bien ces algorithmes, car ils simplifient la vie des programmeurs et surtout parce qu'ils augmentent la lisibilité des programmes. Lorsque vous lisez une boucle qui a été implémentée par un autre programmeur, vous devez commencer par décrypter les intentions de ce programmeur. Alors que si vous voyez un appel à une méthode comme `Collections.max`, vous devinez immédiatement ce que réalise cette instruction.

Les notes d'API suivantes décrivent les algorithmes simples de la classe `Collections`.

java.util.Collections 1.2

- `static <T extends Comparable<? super T>> T min(Collection<T> elements)`
- `static <T extends Comparable<? super T>> T max(Collection<T> elements)`
- `static <T> min(Collection<T> elements, Comparator<? super T> c)`
- `static <T> max(Collection<T> elements, Comparator<? super T> c)`
Renvoient le plus petit ou le plus grand élément de la collection (les bornes des paramètres sont simplifiées pour plus de clarté).
- `static <T> void copy(List<? super T> to, List<T> from)`
Copie tous les éléments de la liste source à la même position dans la liste cible. La liste cible doit être au moins aussi longue que la liste source.
- `static <T> void fill(List<? Super T> l, T value)`
Remplit toutes les positions d'une liste avec la même valeur.
- `static <T> boolean addAll(Collection<? super T>, c, T... values) 5.0`
Ajoute toutes les valeurs à la collection donnée et renvoie true si la collection a changé en conséquence.
- `static <T> boolean replaceAll(List<T> l, T oldValue, T newValue) 1.4`
Remplace tous les éléments égaux à `oldValue` par `newValue`.
- `static int indexOfSubList(List<?> l, List<?> s) 1.4`
- `static int lastIndexOfSubList(List<?> l, List<?> s) 1.4`
Renvoie l'indice de la première ou de la dernière sous-liste de `l`, égale à `s` ou `-1` lorsqu'aucune sous-liste de `l` égale `s`. Par exemple, si `l` est égal à `[s, t, a, r]` et `s` à `[t, a, r]`, les deux méthodes renvoient l'indice 1.
- `static void swap(List<?> l, int i, int j) 1.4`
Echange les éléments aux décalages donnés.
- `static void reverse(List<?> l)`
Inverse l'ordre des éléments d'une liste. Par exemple, l'inversion de la liste `[t, a, r]` produit la liste `[r, a, t]`. La complexité de cette méthode est $O(n)$, où n correspond à la longueur de la liste.
- `static void rotate(List<?> l, int d) 1.4`
Fait tourner les éléments dans la liste, en déplaçant les entrées avec indice `i` à la position `(i + d) % l.size()`. Par exemple, le déplacement de la liste `[t, a, r]` de 2 produit la liste `[a, r, t]`. La complexité de cette méthode est $O(n)$, où n correspond à la longueur de la liste.
- `static int frequency(Collection<?> c, Object o) 5.0`
Renvoie le compte d'éléments dans `c`, égal à l'objet `o`.
- `boolean disjoint(Collection<?> c1, Collection<?> c2) 5.0`
Renvoie true si les collections n'ont pas d'éléments en commun.

Ecrire vos propres algorithmes

Si vous écrivez vos propres algorithmes (ou en fait n'importe quelle méthode prenant une collection en paramètre), vous devriez travailler avec des *interfaces*, et non avec des implémentations concrètes, lorsque cela est possible. Par exemple, supposons que vous vouliez remplir un `JMenu` avec un

134 Au cœur de Java 2 - Fonctions avancées

ensemble d'éléments de menu. Normalement, une telle méthode peut être implémentée comme ceci :

```
void fillMenu(JMenu menu, ArrayList<JMenuItem> items)
{
    for (JMenuItem item : items)
        menu.addItem(item);
}
```

Cependant, vous obligez les utilisateurs de cette méthode à placer les chaînes dans un `ArrayList`. Si ces chaînes se trouvent être dans un autre conteneur, elles devraient être adaptées en vecteur. En fait, il vaut mieux accepter des collections beaucoup plus générales en paramètre.

Il faut vous demander quelle est l'interface de collection la plus générale pouvant faire l'affaire. Dans notre cas, il suffit de parcourir tous les éléments, ce qui est l'une des caractéristiques de base de l'interface `Collection`. Voici comment réécrire une version de la méthode `fillMenu` qui accepte n'importe quel type de collection :

```
void fillMenu(JMenu menu, Collection<JMenuItem> items)
{
    for (JMenuItem item : items)
        menu.addItem(item);
}
```

Désormais, n'importe qui peut appeler cette méthode avec un `ArrayList`, un `LinkedList` ou même un tableau, emballé avec l'emballage `Arrays.asList`.

INFO

Si l'utilisation d'interfaces de collection en paramètres des méthodes est une si bonne idée, pourquoi la bibliothèque Java ne suit-elle pas plus souvent cette règle ? Par exemple, la classe `JComboBox` possède deux constructeurs :

```
JComboBox(Object[] items)
JComboBox(Vector<?> items)
```

La raison en est très simple. La bibliothèque Swing a été conçue avant la bibliothèque de collections.

Si vous écrivez une méthode qui *renvoie* une collection, vous pouvez aussi vouloir renvoyer une interface au lieu d'une classe car vous pourrez alors changer d'avis et réimplanter la méthode par la suite avec une collection différente.

Par exemple, écrivons une méthode `getAllItems` qui renvoie tous les éléments d'un menu :

```
List<MenuItem> getAllItems(JMenu menu)
{
    ArrayList<MenuItem> items = new ArrayList<MenuItem>();
    for (int i = 0; i < menu.getItemCount(); i++)
        items.add(menu.getItem(i));
    return items;
}
```

Vous pouvez ensuite décider que vous ne voulez pas *copier* les éléments mais simplement en fournir une vue. Cela peut être réalisé en renvoyant une sous-classe anonyme de `AbstractList` :

```
List<MenuItem> getAllItems(final JMenu menu)
{
```

```
return new
    AbstractList<MenuItem>()
    {
        public MenuItem get(int i)
        {
            return item.getItem(i);
        }
        public int size()
        {
            return item.getItemCount();
        }
    };
}
```

Naturellement, il s'agit d'une technique avancée. Si vous vous en servez, prenez soin de documenter avec exactitude quelles opérations "optionnelles" sont supportées. Dans ce cas, vous devez avertir l'utilisateur que l'objet renvoyé est une liste non modifiable.

Les anciennes collections

Dans cette partie, nous abordons les classes de collections qui sont présentes dans le langage de programmation Java depuis son début : la classe `Hashtable` et sa sous-classe utile `Properties`, la sous-classe `Stack` de `Vector`, et la classe `BitSet`.

La classe `Hashtable`

La classe `Hashtable` a le même objectif que la classe `HashMap`, et possède essentiellement la même interface. Comme pour les méthodes de la classe `Vector`, les méthodes `Hashtable` sont synchronisées. Si vous n'avez aucun besoin d'une synchronisation ou d'une compatibilité avec les anciens programmes, vous devriez utiliser `HashMap` à la place de cette classe.

INFO

Le nom exact de cette classe est `Hashtable`, avec un `t` minuscule. Sous Windows, vous obtiendrez des messages d'erreur étranges si vous écrivez `HashTable`, parce que le système de fichiers de Windows ne fait pas de différence entre les majuscules et les minuscules, contrairement au compilateur Java.

Les énumérations

Les anciennes collections se servent de l'interface `Enumeration` pour parcourir des séquences d'éléments. L'interface `Enumeration` possède deux méthodes, `hasMoreElements` et `nextElement`. Ces méthodes sont les équivalents des méthodes `hasNext` et `next` de l'interface `Iterator`.

Par exemple, la méthode `elements` de la classe `Hashtable` fournit un objet pour passer en revue les valeurs de la table :

```
Enumeration<Employee> e = staff.elements();
while (e.hasMoreElements())
{
    Employee e = e.nextElement();
    . . .
}
```

136 Au cœur de Java 2 - Fonctions avancées

Vous rencontrerez parfois une ancienne méthode qui attend un paramètre d'énumération. La méthode statique `Collections.enumeration` fournit un objet d'énumération qui passe en revue les éléments d'une collection. Par exemple :

```
ArrayList<InputStream> streams = . . . ;
SequenceInputStream in
    = new SequenceInputStream(Collections.enumeration(streams));
// Le constructeur SequenceInputStream attend une énumération.
```

INFO

En C++, il est assez courant d'utiliser des itérateurs en paramètre. Heureusement, lorsque vous programmez pour la plate-forme Java, très peu de programmeurs se servent de ce concept. Il est beaucoup plus rusé de passer une collection que de passer un itérateur : une collection est bien plus utile. Les conteneurs peuvent toujours obtenir un itérateur de la collection s'ils en ont réellement besoin. De plus, ils ont toutes les méthodes de collection à leur disposition. Cependant, vous trouverez des énumérations dans certains vieux programmes parce qu'ils étaient les seuls outils disponibles pour des collections générales jusqu'à l'apparition du cadre des collections dans le JDK 1.2.

java.util.Enumeration<E> 1.0

- `boolean hasMoreElements()`
Renvoie `true` s'il reste d'autres éléments à parcourir.
- `E nextElement()`
Renvoie le prochain élément à parcourir. N'appellez pas cette méthode si `hasMoreElements()` renvoie `false`.

java.util.Hashtable<K, V> 1.0

- `Enumeration<K> keys()`
Renvoie un objet énumération qui parcourt les clés d'une table de hachage.
- `Enumeration<V> elements()`
Renvoie un objet énumération qui parcourt les éléments d'une table de hachage.

java.util.Vector<E> 1.0

- `Enumeration<E> elements()`
Renvoie un objet énumération qui parcourt les éléments d'un vecteur.

Ensembles de propriétés

Un *ensemble de propriétés* est une structure de carte d'un type très spécial, affichant trois caractéristiques particulières :

- Les clés et les valeurs sont des chaînes.
- La table peut être enregistrée dans un fichier et chargée à partir d'un fichier.
- Il existe une table secondaire contenant les paramètres par défaut.

La classe Java qui implémente un ensemble de propriétés est appelée `Properties`.

Les ensembles de propriétés permettent souvent de spécifier des options de configuration pour des programmes (voir Chapitre 10 du Volume 1).

java.util.Properties 1.0

- `Properties()`
Crée une liste de propriétés vide.
- `Properties(Properties defaults)`
Crée une liste de propriétés avec un ensemble de valeurs par défaut.
- `String getProperty(String key)`
Renvoie la chaîne associée à une clé, éventuellement dans la table de valeurs par défaut si la clé ne figure pas dans la table spécifiée.
- `String getProperty(String key, String defaultValue)`
Trouve une propriété ou sa valeur par défaut si la clé n'a pas été trouvée. Renvoie une chaîne associée à la clé, ou une valeur par défaut si la clé ne figure pas dans la table.
- `void load(InputStream in)`
Charge un ensemble de propriétés à partir d'un `InputStream`.
- `void store(OutputStream out, String commentString)`
Stocke un set de propriétés dans un `OutputStream`.

Piles

Depuis la version 1.0, la bibliothèque standard possède une classe `Stack` dotée des méthodes familières `push` et `pop`. Toutefois, la classe `Stack` prolonge la classe `Vector`, qui n'est pas satisfaisante d'un point de vue théorique : vous pouvez en effet appliquer des opérations de désempilage comme `insert` et `remove` en vue d'insérer et de supprimer des valeurs à n'importe quel endroit, et non pas simplement en haut de la pile.

java.util.Stack<E> 1.0

- `E push(E item)`
Place un élément sur la pile et le renvoie.
- `E pop()`
Récupère l'élément situé sur le dessus de la pile. N'appellez pas cette méthode si la pile est vide.
- `E peek()`
Renvoie l'élément situé sur le dessus de la pile sans le dépiler. N'appellez pas cette méthode si la pile est vide.

Les ensembles de bits

La classe `BitSet` sert à stocker des séquences de données binaires. Il ne s'agit pas d'un *ensemble* au sens mathématique du terme. Les termes *tableau de bits* ou *vecteur de bits* auraient été plus appropriés. Vous pouvez vous servir d'un `BitSet` si vous devez enregistrer une séquence de bits de manière efficace. Comme les ensembles de bits réunissent leurs informations dans des octets, il est bien plus efficace d'utiliser un `BitSet` qu'un `ArrayList` d'objets booléens.

La classe `BitSet` fournit une interface pratique pour lire, écrire ou réinitialiser un bit. Vous pouvez vous servir de cette interface pour éviter les opérations de manipulation de bits (comme les masques) qui seraient nécessaires si vous stockiez les bits dans des variables `int` ou `long`.

138 *Au cœur de Java 2 - Fonctions avancées*

Par exemple, pour un `BitSet` appelé `BucketOfBits`,

```
bucketOfBits.get(i)
```

renvoie `true` si le *i*-ème bit est à 1, et `false` dans le cas contraire. De même,

```
bucketOfBits.set(i)
```

met le *i*-ème bit à 1. Pour terminer,

```
bucketOfBits.clear(i)
```

met le *i*-ème bit à 0.

INFO C++

Le modèle `bitset` du C++ possède les mêmes fonctionnalités que le `BitSet` Java.

java.util.BitSet 1.0

- `BitSet(int initialCapacity)`
Construit un ensemble de bits.
- `int length()`
Renvoie la longueur logique d'un ensemble de bits : l'indice du dernier bit plus 1.
- `boolean get(int bit)`
Lit un bit.
- `void set(int bit)`
Ecrit un bit.
- `void clear(int bit)`
Efface un bit.
- `void and(BitSet set)`
Effectue un ET logique entre deux ensembles de bits.
- `void or(BitSet set)`
Effectue un OU logique entre deux ensembles de bits.
- `void xor(BitSet set)`
Effectue un OU exclusif logique entre deux ensembles de bits.
- `void andNot(BitSet set)`
Efface tous les bits de cet ensemble de bits qui sont à 1 dans l'autre ensemble de bits.

Le test du crible d'Eratosthène

Comme exemple d'utilisation d'ensembles de bits, nous voulons vous montrer une implémentation du crible d'Eratosthène, un algorithme permettant de trouver les nombres premiers. Un nombre premier est un nombre qui n'est divisible que par 1 et par lui-même, et le crible d'Eratosthène est l'une des premières méthodes découvertes pour énumérer ces nombres premiers. En réalité, il ne s'agit pas d'un excellent algorithme pour trouver des nombres premiers, mais il est devenu un test

très populaire pour mesurer les performances d'un compilateur. En fait, ce n'est pas vraiment un excellent test, car il effectue beaucoup d'opérations sur les bits.

Bon, d'accord, respectons la tradition : voici une implémentation de ce programme qui compte tous les nombres premiers entre 2 et 2 000 000. Comme il y en a 148 933, vous n'avez probablement pas envie de tous les afficher.

Sans vouloir entrer dans les détails de ce programme, l'idée principale est de parcourir un ensemble de bits comprenant 2 millions de bits. Nous commençons par mettre tous ces bits à 1. Puis, nous mettons à 0 tous les bits qui sont des multiples des nombres premiers que nous avons déjà identifiés. La position des bits qui restent à 1 à la fin du processus fournit directement les nombres premiers. L'Exemple 2.8 illustre ce programme en Java, et l'Exemple 2.9 correspond au code C++.

INFO

Même si ce crible ne constitue pas un excellent test, nous n'avons pas pu résister au plaisir de chronométrer ces deux implémentations. Voici donc les performances observées sur un Thinkpad IBM 1,7 GHz avec 1 gigaoctets de RAM, fonctionnant sous Red Hat Linux 9.

C++ (g++ 3.2.2) : 330 millisecondes.

Java (JDK 5.0) : 105 millisecondes.

Nous avons effectué ce test pour six versions de ce livre, et c'est la troisième fois que l'implémentation Java supplante facilement la version C++. Dans les éditions précédentes, nous signalions, pour rester honnêtes, que le mauvais score du C++ venait des piètres performances du modèle standard `bitset`. Lorsque nous avons réimplémenté `bitset`, le temps d'exécution de la version C++ était plus rapide que Java. Ce n'est plus le cas. Avec un `bitset` bricolé, le temps du C++ était de 140 millisecondes dans notre dernière tentative.

Exemple 2.8 : *Sieve.java*

```
import java.util.*;

/**
 * Ce programme exécute le test du crible d'Eratosthène.
 * Il calcule tous les nombres premiers jusqu'à 2 000 000.
 */
public class Sieve
{
    public static void main(String[] s)
    {
        int n = 2000000;
        long start = System.currentTimeMillis();
        BitSet b = new BitSet(n + 1);
        int count = 0;
        int i;
        for (i = 2; i <= n; i++)
            b.set(i);
        i = 2;
        while (i * i <= n)
        {
            if (b.get(i))
            {

```

140 Au cœur de Java 2 - Fonctions avancées

```

        count++;
        int k = 2 * i;
        while (k <= n)
        {
            b.clear(k);
            k += i;
        }
        i++;
    }
    while (i <= n)
    {
        if (b.get(i))
            count++;
        i++;
    }
    long end = System.currentTimeMillis();
    System.out.println(count + " nombres premiers");
    System.out.println((end - start) + " millisecondes");
}

```

Exemple 2.9 : Sieve.cpp

```

#include <bitset>
#include <iostream>
#include <ctime>

using namespace std;

int main()
{
    const int N = 2000000;
    clock_t cstart = clock();

    bitset<N + 1> b;
    int count = 0;
    int i;
    for (i = 2; i <= N; i++)
        b.set(i);
    i = 2;
    while (i * i <= N)
    {
        if (b.test(i))
        {
            count++
            int k = 2 * i;
            while (k <= N)
            {
                b.reset(k);
                k += i;
            }
        }
        i++;
    }
    while (i <= N)
    {

```

```

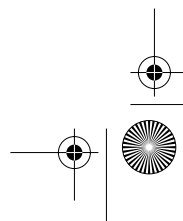
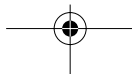
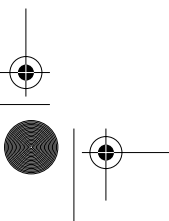
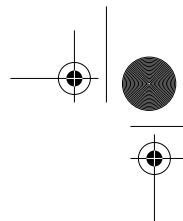
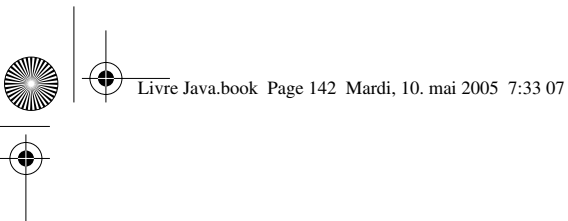
        if (b.test(i))
            count++;
        i++;
    }

    clock_t cend = clock();
    double millis = 1000.0
        * (cend - cstart) / CLOCKS_PER_SEC;

    cout << count << " nombres premiers\n"
        << millis << " millisecondes\n";

    return 0;
}w

```



3

Programmation des bases de données

Au sommaire de ce chapitre

- ✓ La conception de JDBC
- ✓ La structure du langage de requêtes
- ✓ Installation de JDBC
- ✓ Principaux concepts de programmation JDBC
- ✓ Exécution de requêtes
- ✓ Ensembles de résultats défilants et actualisables
- ✓ Métadonnées
- ✓ RowSet
- ✓ Transactions
- ✓ Gestion avancée des connexions
- ✓ Introduction au LDAP

Au cours de l'été 1996, Sun a diffusé la première version de l'API du kit JDBC (Java Database Connectivity). Elle permet aux programmeurs de se connecter à une base de données, d'y effectuer une recherche ou de la mettre à jour grâce au langage SQL (Structured Query Language, ou langage de requêtes structurées), qui fait partie des standards industriels pour l'accès aux bases de données.

L'avantage essentiel de Java et de JDBC sur les autres environnements de programmation pour les bases de données est le suivant : les programmes développés avec Java et JDBC fonctionnent sur n'importe quel ordinateur et peuvent être distribués par n'importe qui.

Un même programme écrit avec le langage de programmation Java peut fonctionner sous NT, sur un serveur Solaris ou sur un ordinateur dédié aux bases de données, à partir du moment où celui-ci possède la plate-forme Java. Même si vos données sont déplacées d'une base de données à une autre, comme par exemple d'un serveur Microsoft SQL vers Oracle, ou encore vers une petite base de données intégrée dans un petit appareil portatif, le même programme pourra toujours lire vos données. Cette approche est diamétralement opposée à la programmation de bases de données traditionnelle.

144 Au cœur de Java 2 - Fonctions avancées

Il est malheureusement trop courant de voir une personne écrire des applications de bases de données dans un langage propriétaire, avec un système de gestion de base de données propre à un seul distributeur.

JDBC a été mis à jour plusieurs fois. Une seconde version de JDBC a été intégrée dans le JDK 1.2, en 1998. Au moment où ce livre est écrit, JDBC 3 est la version la plus courante et JDBC 4 est en cours de développement. JDBC 3 est inclus dans les JDK 1.4 et 5.0.

Nous devons également vous avertir que le JDK ne fournit aucun outil de programmation de bases de données avec le langage de programmation Java. Pour concevoir des formulaires, générer des requêtes et créer des rapports, vous devrez avoir recours à des logiciels d'autres entreprises.

Dans ce chapitre, nous expliquerons certaines idées propres à JDBC (l'API de connectivité de bases de données de Java).

- Nous vous présenterons SQL (ou nous vous en rappellerons le fonctionnement), le langage de requêtes structuré standard sur le marché pour les bases de données relationnelles.
- Nous vous fournirons suffisamment de détails et d'exemples pour que vous puissiez réellement commencer à utiliser JDBC.
- Nous terminerons par une brève introduction des bases de données hiérarchiques, du protocole LDAP et de JNDI (l'interface de répertoire et de nommage de Java).

INFO

Au cours des années, plusieurs techniques ont été inventées pour rendre les accès aux bases de données plus efficaces et plus sûrs. Les bases de données relationnelles standard supportent des indices, des déclencheurs, des procédures enregistrées et une gestion de transactions. JDBC gère également toutes ces caractéristiques, mais nous ne les aborderons pas dans ce chapitre. Un livre entier pourrait être écrit à propos des caractéristiques avancées de programmation de bases de données en Java, et en fait, plusieurs livres semblables ont déjà été écrits. Pour aller plus loin avec JDBC, nous vous suggérons le livre (en langue anglaise) *JDBC API Tutorial and Reference*, de Maydene Fisher, Jon Ellis et Jonathan Bruce (Addison-Wesley, 2003).

Conception de JDBC

Dès le début, les concepteurs de Java chez Sun étaient conscients du potentiel de Java pour travailler sur les bases de données. A partir de 1995, ils ont commencé à agrandir la bibliothèque standard Java pour traiter les accès SQL aux bases de données. Ils avaient tout d'abord espéré étendre Java pour communiquer avec n'importe quelle base de données, en utilisant uniquement du code Java. Il leur a fallu très peu de temps pour réaliser qu'il s'agissait d'une tâche impossible : il y a tout simplement trop de bases de données différentes, utilisant trop de protocoles différents. De plus, même si tous les concepteurs de bases de données étaient d'accord pour que Sun fournisse un protocole réseau standard d'accès aux bases de données, ils voulaient tous que Sun choisisse *leur* protocole réseau.

Ces concepteurs sont cependant tombés d'accord sur un point : Sun devrait fournir une API Java pour les accès SQL avec un gestionnaire de pilotes, pour permettre aux pilotes développés par les autres entreprises de se connecter aux différentes bases de données. Les concepteurs de bases de données pourraient alors fournir leur propre pilote au gestionnaire de pilotes. Un mécanisme simple

suffirait alors pour incorporer les pilotes de ces entreprises dans le gestionnaire de pilotes, le plus important étant que les pilotes auraient uniquement à suivre les spécificités définies dans l'API du gestionnaire de pilotes.

Deux interfaces ont été créées en conséquence. Les programmeurs d'applications utilisent l'API JDBC, les concepteurs de bases de données et d'outils utilisent l'API JDBC Driver.

Ce protocole reprend le modèle très réussi d'ODBC de Microsoft, qui fournit une interface de langage de programmation en C pour les accès aux bases de données. JDBC et ODBC sont fondés sur la même idée : les programmes compatibles avec l'API communiquent avec le gestionnaire de pilotes, qui, en retour, se sert des pilotes auxquels il est relié au moment où la communication avec la base de données est établie.

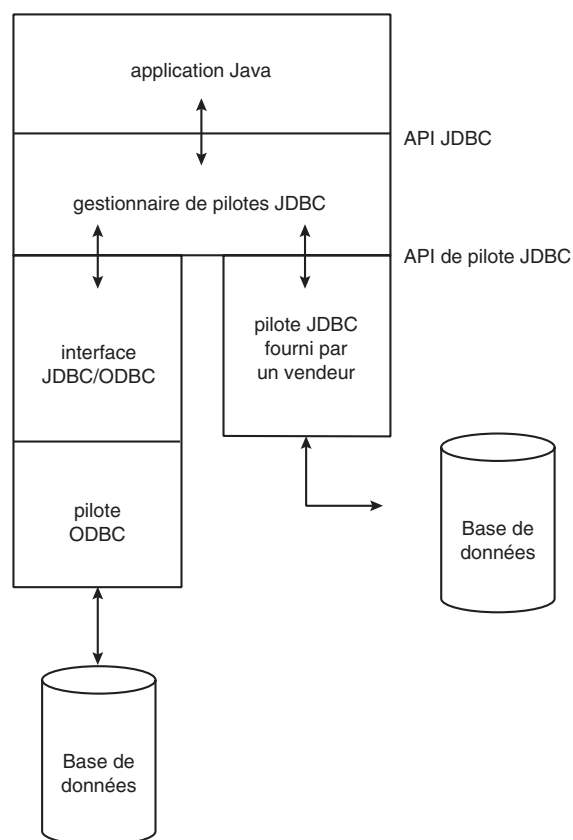
Tout cela implique que l'API JDBC est la seule chose qu'auront à gérer la plupart des programmeurs (voir Figure 3.1).

INFO

La liste des pilotes disponibles actuellement se trouve sur le site Web <http://industry.java.sun.com/products/jdbc/drivers>.

Figure 3.1

La communication entre JDBC et une base de données.



Types de pilotes JDBC

Les pilotes JDBC sont classés en plusieurs *types* :

- *Un pilote de type 1* traduit JDBC en ODBC et se sert d'un pilote ODBC pour communiquer avec la base de données. Sun fournit un pilote de ce type dans le JDK : *l'interface JDBC/ODBC*. Cependant, cette interface nécessite un développement et une configuration particulière d'un pilote ODBC. A sa sortie, l'interface se voulait pratique pour le test, mais n'a jamais été destinée à être produite en gros volumes. Aujourd'hui, il existe de nombreux pilotes, mieux adaptés. Nous déconseillons l'utilisation de l'interface JDBC/ODBC.
- *Un pilote de type 2* est un pilote écrit partiellement en Java et partiellement dans un langage natif, qui communique avec l'API cliente d'une base de données. Lorsque vous vous servez de ce type de pilote, vous devez installer des programmes spécifiques à votre plate-forme en plus de la bibliothèque Java.
- *Un pilote de type 3* est une bibliothèque client en Java pur qui se sert d'un protocole indépendant de la base de données pour communiquer des requêtes de base de données à un serveur, qui se charge ensuite de traduire la requête dans un protocole propre à la base de données. La bibliothèque client est en fait indépendante de la base de données, ce qui simplifie son développement.
- *Un pilote de type 4* est une bibliothèque en Java pur qui traduit directement des requêtes JDBC en un protocole spécifique à la base de données.

La plupart des concepteurs de bases de données fournissent un pilote de type 3 ou 4 avec leur base de données. De plus, un certain nombre d'entreprises se sont spécialisées dans la production de pilotes qui se conforment mieux aux standards, supportent plus de plates-formes, sont plus performants ou, dans certains cas, sont plus fiables que les pilotes fournis par les concepteurs de bases de données.

Pour résumer, l'objectif fondamental de JDBC est de rendre réalisables les concepts suivants :

- Les programmeurs peuvent écrire des applications dans le langage de programmation Java pour accéder à n'importe quelle base de données, grâce aux instructions standard SQL, ou même des extensions particulières de SQL, en respectant toujours les conventions du langage Java.
- Les concepteurs de bases de données et d'outils pour bases de données peuvent fournir leurs pilotes de bas niveau. Par conséquent, ils peuvent optimiser leurs pilotes pour leurs produits spécifiques.

INFO

Si vous vous demandez pourquoi Sun n'a pas adopté le modèle ODBC, voici quelle a été sa réponse, donnée au cours de la conférence JavaOne en mai 1996 :

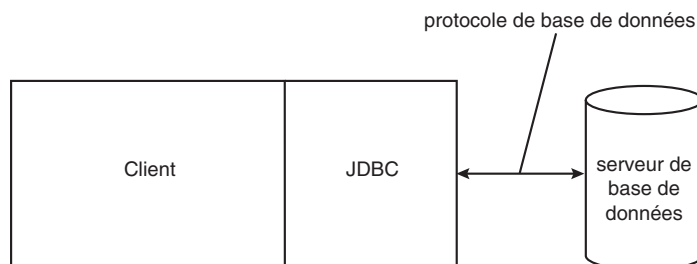
- ODBC est complexe à apprendre.
- ODBC possède quelques commandes nécessitant beaucoup d'options compliquées. La philosophie privilégiée en Java est d'avoir un grand nombre de méthodes simples et intuitives.
- ODBC se sert de pointeurs `void*` et d'autres caractéristiques du C qui ne sont pas naturelles dans Java.
- Une solution fondée sur ODBC est intrinsèquement moins fiable et plus complexe à développer qu'une solution en Java pur.

Applications typiques de JDBC

Le modèle client/serveur traditionnel possède une GUI riche sur le client et une base de données sur le serveur (voir Figure 3.2). Dans ce modèle, un pilote JDBC est déployé sur le client.

Figure 3.2

Une application client-serveur.

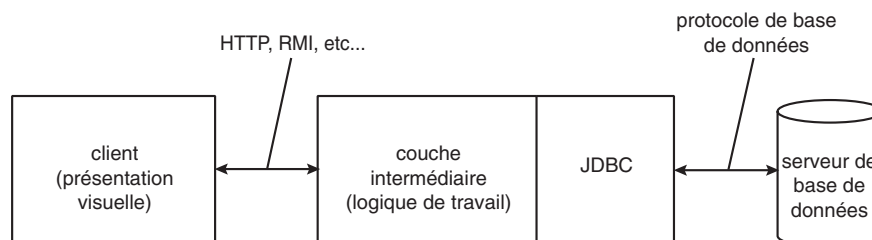


Cependant, la mode de l'architecture client-serveur est en train de disparaître, au profit des modèles en trois niveaux, ou même des modèles en n niveaux. Dans un modèle en trois niveaux, le client ne fait aucun appel à la base de données : il appelle une couche intermédiaire du serveur, qui exécute à son tour la requête de base de données. Ce modèle possède quelques avantages. Il distingue la *présentation visuelle* (du client), la *logique fonctionnelle* (dans la couche intermédiaire) et les données brutes (de la base de données). Par conséquent, il devient possible d'accéder aux mêmes données et aux mêmes règles de travail à partir de plusieurs clients, comme une application Java, un applet ou un formulaire Web.

La communication entre le client et la couche intermédiaire peut se faire par l'intermédiaire du HTTP (lorsque vous utilisez un navigateur Web comme client), d'une RMI (lorsque vous utilisez une application ou un applet ; voir le Chapitre 5) ou d'un autre mécanisme. JDBC est utilisé pour gérer la communication entre la couche intermédiaire et la base de données. La Figure 3.3 en représente l'architecture principale. Il existe naturellement plusieurs variantes de ce modèle. En particulier, l'édition Entreprise de Java 2 définit une structure pour les *serveurs d'application* qui gère des modules de code appelés *Enterprise JavaBeans*, et qui fournit des services intéressants comme la répartition du taux de charge, la mise en mémoire intermédiaire des requêtes, la sécurité et des accès simples de base de données. Dans cette architecture, JDBC joue toujours un rôle important pour résoudre des requêtes complexes de base de données. Pour plus de renseignements sur l'édition Entreprise, consultez le site <http://java.sun.com/j2ee>.

Figure 3.3

Une application en trois niveaux.



148 Au cœur de Java 2 - Fonctions avancées

INFO

Vous pouvez utiliser JDBC dans les applets, même si vous n'en avez pas forcément envie. Par défaut, le gestionnaire de sécurité n'autorise la connexion à la base de données qu'au serveur à partir duquel l'applet est téléchargée. Cela signifie que le serveur Web et le serveur de base de données doivent se trouver sur le même ordinateur, ce qui n'est pas la configuration standard. Avec les applets signées, cette restriction peut être levée. De plus, l'applet aurait à inclure le pilote JDBC.

SQL

JDBC permet de communiquer avec les bases de données grâce à SQL, le langage de commande vers presque toutes les bases de données relationnelles modernes. Les bases de données grand public possèdent généralement une interface utilisateur graphique qui permet à l'utilisateur de manipuler les données directement, mais les bases de données hébergées sur un serveur sont uniquement accessibles au travers de SQL. La plupart des bases de données grand public possèdent également une interface SQL, mais bien souvent cette dernière ne supporte pas toutes les caractéristiques SQL standard.

JDBC peut être considéré comme étant uniquement une API (*Application Programming Interface*, ou interface de programmation d'une application) permettant d'envoyer des requêtes SQL aux bases de données. Cette section renferme une brève introduction à SQL. Si vous ne connaissez pas ce type de requêtes, vous voudrez peut-être en apprendre davantage. Dans ce cas, vous pourrez consulter l'un des nombreux livres traitant de ce sujet. Nous vous recommandons (en langue anglaise) *Client/Server Databases*, de James Martin et Joe Leben (Prentice-hall, 1998) ou la bible de référence, *A Guide to the SQL Standard*, de C. J. Date et Hugh Darwen (Addison-Wesley, 1997).

Vous pouvez considérer qu'une base de données est un ensemble de tableaux étiquetés de lignes et de colonnes. Chaque colonne possède un *nom de colonne*, et chaque ligne renferme des données, qui sont parfois appelées des *enregistrements*.

Comme exemple de base de données, nous nous servons d'un ensemble de tableaux décrivant un ensemble de livres américains sur HTML.

Tableau 3.1 : Le tableau Auteurs

<i>Auteur_ID</i>	<i>Nom</i>	<i>Prénom</i>
ALEX	Alexander	Christopher
BROO	Brooks	Frederick P.
...	...	...

Tableau 3.2 : Le tableau Livres

<i>Titre</i>	<i>ISBN</i>	<i>Editeur_ID</i>	<i>Prix</i>
A Guide to the SQL Standard	0-201-96426-0	0201	47,95
A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	019	65.00
...	...	...	...

Tableau 3.3 : Le tableau *LivresAuteurs*

ISBN	Auteur_ID	Nu_Séq
0-201-96426-0	DATE	1
0-201-96426-0	DARW	2
0-19-501919-9	ALEX	1
...	...	...

Tableau 3.4 : Le tableau *Éditeurs*

Éditeur_ID	Nom	URL
0201	Addison-Wesley	www.aw-bc.com
0407	John Wiley & Sons	www.wiley.com
...	...	...

La Figure 3.4 fournit une représentation du tableau *Livres*. La Figure 3.5 montre la *fusion* de ce tableau avec le tableau *Éditeurs*. Ces deux tableaux contiennent un numéro de code pour chaque éditeur. Lorsque ces deux tableaux sont fusionnés en fonction de ce numéro de code, nous obtenons un *résultat de requête* contenant les données des deux tableaux. Chaque ligne du résultat contient les informations relatives à un livre, au nom de l'éditeur et à l'URL de sa page Web. Notez que les noms des éditeurs et les URL sont dupliqués sur plusieurs lignes puisque plusieurs lignes parlent d'un même éditeur.

Figure 3.4

Un simple tableau contenant des livres.

ISBN	Price	Publisher	Title	URL
0-07-882198-3	27.95	00788	Beyond HTML	
0-78970541-9	15.00	07897	10 Minute Guide to HTML	www.mcp.com/cgi-bin/bag?isbn=1-07897-0541-9&userID=4&last=bookstore
1-56-894454-9	19.99	15689	Creating Cool Web Pages	db:www.idgbooks.com/database/book/isbn/generic-book.html?query=1-56894
1-56894545-2	20.00	15688	Creating Web Pages for	db:www.idgbooks.com/database/book/isbn/generic-book.html?query=1-56894
1-56276390-3	24.99	15627	How to Use HTML 3	www.mcp.com/169486041707673/cgi-bin/bag?isbn=1-56276-390-3&last=book
1-57-163050-7	39.99	15716	HTML 3 How-To	w3.waite.com/waite/waite/books/new/HTML_3_How_To/html/html3htcov.htm
1-57163066-2	39.99	15716	HTML 3 Interactive Cou	w3.waite.com/waite/waite/books/new/HTML_3_Interactive_Course/html/h3ic
1-56-276352-0	24.95	15627	HTML 3.0 Manual of St	www.mcp.com/zdpress/features/html3.html
0-78970812-4	34.99	07897	HTML by Example	www.mcp.com/26417520761436/cgi-bin/bag?isbn=0-7897-0812-4&last=book
0-13-232331-1	40.00	01356	HTML CD - An Internet F	www.prenhall.com/013/232330/ptr/23233-0.html
1-56-894647-9	30.00	15688	HTML for Dummies 2nd	db:www.idgbooks.com/database/book/isbn/generic-book.html?query=1-56894
1-56894590-7	12.99	15688	HTML For Dummies Qui	db:www.idgbooks.com/database/book/isbn/generic-book.html?query=1-56894
0-13-242498-6	40.00	01356	HTML For Fun and Profi	www.sun.com/html/scilpress/books/Mom/Mom.html
0-02189448-6	17.95	00218	HTML for the World Wic	www.peachpit.com/peachpit/Html/catalog/98448.html
1-55615948-X	29.95	15561	HTML In Action	www.microsoft.com/MSPRESS/BOOKS/DES/5-948-xa.HTM
0-53451626-2	36.00	05345	HTML Plus!	scholar.lib.vt.edu/powell/htmlplus/
1-56-604288-7	39.95	15660	HTML Publishing for Net	ephraim.vmedia.com/catalog/
0-07-812100-4	45.00	00781	HTML Publishing on the	
0-78970867-1	17.99	07897	HTML Quick Reference	www.mcp.com/282320801523718/cgi-bin/bag?isbn=0-7897-0867-1&last=book
0-47-114242-5	30.00	04711	HTML Sourcebook, 2nd	www.wiley.com/80/compbooks/k19.html

L'intérêt de fusionner plusieurs tableaux est d'éviter une redondance de données. Par exemple, une base de données à l'architecture simpliste peut posséder des colonnes correspondant au nom de

150 Au cœur de Java 2 - Fonctions avancées

l'éditeur et à l'URL dans le tableau Livres. Mais dans ce cas, la base de données elle-même (et pas seulement le résultat de la requête) possédera ces entrées en plusieurs exemplaires. Si l'adresse Web d'un éditeur est amenée à changer, *toutes* les entrées devront être mises à jour. Il s'agit donc clairement d'une technique qui peut entraîner des erreurs. Avec un modèle relationnel, les données sont réparties dans plusieurs tableaux, de sorte qu'aucune information ne soit dupliquée sans raison. Par exemple, chaque URL d'éditeur ne figure qu'une seule fois dans le tableau des éditeurs. Si des informations doivent être combinées, les tableaux correspondants doivent être fusionnés.

Figure 3.5
Deux tableaux
fusionnés.

ISBN	Price	Publisher	Name	URL	Title
0-07-682198-3	27.95	00788	Osborne/McGraw-Hill	www.osborne.com	Beyond HTML
0-78970541-9	15.00	07897	Que	www.mcp.com/que/	10 Minute Guide to HTML
1-56-884454-9	19.99	15688	IDG Books	www.idgbooks.com/	Creating Cool Web Pages with HTML
1-56884545-2	20.00	15688	IDG Books	www.idgbooks.com/	Creating Web Pages for Dummies
1-56276390-3	24.99	15627	Ziff-Davis	www.mcp.com/zdpress/	How to Use HTML 3
1-57163050-7	39.99	15716	W/ale	www.waite.com/waite/	HTML 3 How-To
1-57163066-2	39.99	15716	W/ale	www.waite.com/waite/	HTML 3 Interactive Course
1-56-276352-0	24.95	15627	Ziff-Davis	www.mcp.com/zdpress/	HTML 3.0 Manual of Style (2nd Ed.)
0-78970812-4	34.99	07897	Que	www.mcp.com/que/	HTML by Example
0-13-232331-1	40.00	01356	Prentice Hall	www.prenhall.com/	HTML CD - An Internet Publishing Toolkit - Windows V
1-56-884454-9	30.00	15688	IDG Books	www.idgbooks.com/	HTML for Dummies 2nd Edition
1-56884990-7	12.99	15688	IDG Books	www.idgbooks.com/	HTML For Dummies Quick Reference
0-13-242488-6	40.00	01356	Prentice Hall	www.prenhall.com/	HTML For Fun and Profit: Gold Signature Edition
0-02188448-8	17.95	00218	Peachpit Press	www.peachpit.com/	HTML for the World Wide Web: Visual Quick Start Gu
1-55615948-X	23.95	15561	Microsoft Press	www.microsoft.com/micr	HTML in Action
0-53451525-2	36.00	05345	Integrated Media Group	www.thomson.com/icer	HTML Plus!
1-56-604288-7	39.95	15660	Netscape Press	www.netscapepress.com	HTML Publishing for Netscape, Windows Edition
0-07-912100-4	45.00	00791	McGraw-Hill	www.mcgraw-hill.com/	HTML Publishing on the Internet
0-78970867-1	17.99	07897	Que	www.mcp.com/que/	HTML Quick Reference
0-47-114242-5	30.00	04711	Wiley	www.wiley.com/	HTML Sourcebook, 2nd Ed.

Dans les figures, nous nous sommes servis d'un outil graphique pour parcourir et lier les tableaux. Plusieurs entreprises proposent des outils qui permettent de spécifier une requête dans un simple formulaire, en y connectant des noms de colonnes et en saisissant des informations dans des formulaires. Ce type d'outil est souvent appelé QBE (*query by example*, ou requête à base d'exemple). Par opposition, une requête SQL est écrite en texte, en respectant la syntaxe propre à SQL. Par exemple :

```
SELECT Livres.Prix, Livres.Titre,
       Livres.Editeur_Id, Editeurs.Nom, Editeurs.URL
FROM Livres, Editeurs
WHERE Livres.Editeur_Id = Editeurs.Editeur_Id
```

Dans le reste de cette section, nous allons voir comment écrire ce type de requête. Si vous connaissez déjà bien SQL, passez directement à la section suivante.

Par convention, les mots clés SQL sont écrits en majuscules, bien que cela ne soit pas strictement nécessaire.

L'opération SELECT est très souple. La requête suivante permet de sélectionner tous les éléments du tableau Livres :

```
SELECT * FROM Livres
```

L'instruction FROM doit être spécifiée dans chaque instruction SELECT. La clause FROM indique à la base de données quels sont les tableaux à examiner pour trouver les données.

Vous pouvez choisir les colonnes que vous souhaitez.

```
SELECT ISBN, Prix, Titre
FROM Livres
```

Les lignes de la réponse peuvent être filtrées grâce à l'instruction WHERE.

```
SELECT ISBN, Prix, Titre
FROM Livres
WHERE Prix <= 29.95
```

Faites attention aux comparaisons d'égalité. SQL se sert de = et <>, et pas de == ou !=, comme le langage de programmation Java.

INFO

Certaines bases de données supportent le signe != pour tester la différence. Il ne s'agit pas d'une instruction standard de SQL, donc nous vous recommandons de ne pas l'utiliser.

La clause WHERE peut également chercher des éléments connus, lorsqu'elle est utilisée avec l'opérateur LIKE. Contrairement à ce que l'on pourrait attendre, les jokers ne sont pas * et ?. Utilisez un % pour zéro ou plusieurs caractères et un caractère souligné pour un seul caractère. Par exemple,

```
SELECT ISBN, Prix, Titre
FROM Livres
WHERE Titre NOT LIKE '%n_x%'
```

exclut les livres dont les titres contiennent les mots UNIX ou Linux.

Notez que les chaînes sont entourées d'apostrophes, et non de guillemets. Les apostrophes dans une chaîne sont remplacées par deux guillemets dans la requête. Par exemple,

```
SELECT Titre
FROM Livres
WHERE Titre LIKE '%''%'
```

renvoie tous les titres qui contiennent une apostrophe.

Il est également possible de sélectionner des données parmi plusieurs tableaux.

```
SELECT * FROM Livres, Editeurs
```

Sans clause WHERE, cette requête n'est pas très intéressante. Elle établit la liste de *toutes les combinaisons* de lignes dans les deux tableaux. Dans notre cas, où Livres possède 20 lignes et Editeurs en comprend 8, le résultat de cette requête est un tableau de 20 × 8 entrées, renfermant beaucoup de données redondantes. Il faut donc réellement restreindre cette requête pour n'afficher que certains livres avec leur éditeur.

```
SELECT * FROM Livres, Editeurs
WHERE Livres.Editeur_Id = Editeurs.Editeur_Id
```

Le résultat de cette requête comprend 20 lignes, une pour chaque livre, puisque chaque livre correspond à un éditeur dans le tableau Editeurs.

Lorsqu'une requête travaille sur plusieurs tableaux, il peut arriver qu'un nom de colonne figure en plusieurs endroits. C'est ce qui s'est produit dans notre exemple. Il existe une colonne Editeur_Id, à la fois dans le tableau Livres et dans le tableau Editeurs. Lorsque ce genre d'ambiguïté risque de

152 *Au cœur de Java 2 - Fonctions avancées*

se produire, il faut faire précéder chaque nom de colonne du nom du tableau auquel il appartient, comme `Livres.Editeur_Id`.

SQL peut aussi modifier les données à l'intérieur d'une base de données, en appelant des requêtes d'action (c'est-à-dire des requêtes qui déplacent ou qui modifient les données). Par exemple, supposons que vous vouliez réduire de 5 € le prix courant de tous les livres dont le titre contient la chaîne "C++".

```
UPDATE Livres
SET Prix = Prix - 5.00
WHERE Titre NOT LIKE '%C++%'
```

L'action la plus importante à part UPDATE est probablement DELETE, qui permet à la requête de supprimer les enregistrements qui correspondent à certains critères de la clause WHERE.

De plus, SQL contient certaines fonctions intégrées pour calculer des moyennes, trouver les maxima et les minima, etc. Pour en savoir plus, consultez le site <http://sqlzoo.net> (ce site contient aussi un bon didacticiel interactif sur SQL).

Typiquement, pour insérer des valeurs dans un tableau, il faut avoir recours à l'instruction INSERT :

```
INSERT INTO Livres
VALUES ('A Guide to the SQL Standard', '0-201-96426-0', '0201', 47.95)
```

Il vous faut une instruction INSERT pour chaque ligne à insérer dans le tableau.

Naturellement, avant de pouvoir effectuer une requête, de modifier ou d'insérer des données, il vous faut un emplacement pour vos données. La commande CREATE TABLE permet de créer un nouveau tableau. Le nom et le type de chaque colonne doivent être spécifiés. Par exemple :

```
CREATE TABLE Livres
(
    Titre CHAR(60),
    ISBN CHAR(13),
    Editeur_Id CHAR(6),
    Prix DECIMAL(10,2)
)
```

Le Tableau 3.5 représente les types de données SQL les plus courants.

Tableau 3.5 : Les types de données SQL

<i>Types de données</i>	<i>Description</i>
INTEGER ou INT	Typiquement, un entier 32 bits
SMALLINT	Typiquement, un entier 16 bits
NUMERIC(<i>m</i> , <i>n</i>) , DECIMAL(<i>m</i> , <i>n</i>) ou DEC(<i>m</i> , <i>n</i>)	Nombre décimal à virgule fixe avec <i>m</i> chiffres au total, dont <i>n</i> chiffres après la virgule
FLOAT(<i>n</i>)	Un nombre à virgule flottante de <i>n</i> bits de précision
REAL	Typiquement, un nombre à virgule flottante de 32 bits
DOUBLE	Typiquement, un nombre à virgule flottante de 64 bits

Tableau 3.5 : Les types de données SQL (suite)

<i>Types de données</i>	<i>Description</i>
CHARACTER(n) ou CHAR(n)	Chaîne de caractères statique de n caractères
VARCHAR(n)	Chaîne de caractères de longueur variable dont la longueur maximale est n
BOOLEAN	Une valeur booléenne
DATE	Date du calendrier, dépendant de l'implémentation
TIME	Heure, dépendant de l'implémentation
TIMESTAMP	Date et heure, dépendant de l'implémentation
BLOB	Un grand objet binaire
CLOB	Un grand objet contenant des caractères

Dans ce livre, nous n'aborderons pas les clauses additionnelles, comme les clés et les contraintes, qui sont utilisables avec la commande CREATE TABLE.

Installation de JDBC

Pour commencer, il vous faudra un programme de bases de données compatible avec JDBC. Il en existe beaucoup d'excellents, comme IBM DB2, Microsoft SQL Server, MySQL, Oracle et PostgreSQL.

Il vous faudra également une base de données pour effectuer quelques expériences. Nous supposons que vous appellerez cette base de données COREJAVA. Créez une nouvelle base de données ou demandez à votre administrateur de bases de données de vous en créer une avec les droits appropriés. Vous devez être capable de créer, de mettre à jour et de supprimer des tableaux.

Si vous n'avez jamais installé de base de données client-serveur auparavant, vous trouverez peut-être que la mise en service de ce type de bases de données est assez complexe, et qu'il est parfois assez difficile d'identifier la cause d'une erreur. Le mieux est de demander de l'aide à un expert si votre configuration ne fonctionne pas correctement.

Si vous débutez dans les bases de données, nous vous recommandons d'installer une base de données Java pure comme McKoi (<http://mckoi.com/database>), HSQLDB (<http://hsqldb.sourceforge.net>) ou Derby (<http://incubator.apache.org/derby>). Ces bases de données sont moins performantes, mais plus simples à configurer.

Pour l'essentiel, tous les créateurs de bases de données possèdent déjà des pilotes JDBC. Vous devez donc retrouver les instructions du créateur pour charger le pilote dans votre programme et pour établir une connexion à la base de données. Dans la prochaine section, nous vous expliquerons comment configurer les deux principales bases de données, disponibles sur plusieurs plates-formes :

- McKoi ;
- PostgreSQL.

154 Au cœur de Java 2 - Fonctions avancées

Les instructions relatives aux autres bases de données sont identiques même si, bien entendu, certains détails peuvent varier.

Nous déconseillons l'utilisation du pilote d'interface JDBC/ODBC livré avec Java 2 SDK. Nous déconseillons encore plus fermement l'utilisation de ce pilote avec une base de données comme Microsoft Access. Non seulement l'installation et la configuration se révèlent assez complexes, mais le pilote de l'interface et les bases de données présentent également des restrictions qui peuvent facilement tromper l'utilisateur. Enfin, cette configuration ne permet pas d'en apprendre beaucoup sur les véritables bases de données.

Principaux concepts de programmation JDBC

La programmation à base de classes JDBC est, par essence, assez proche de l'utilisation des classes de la plate-forme Java : vous construisez des objets à partir des principales classes JDBC, en les étendant par héritage en cas de besoin. Cette section en examine les détails.

INFO

Les classes utilisées pour JDBC se trouvent dans `java.sql` et `javax.sql`.

URL de bases de données

Lorsque vous vous connectez à une base de données, vous devez spécifier la source des données et éventuellement d'autres paramètres. Par exemple, des pilotes de protocole réseau peuvent demander un numéro de port, et des pilotes ODBC peuvent nécessiter certains attributs.

Comme vous pouvez vous y attendre, JDBC se sert d'une syntaxe semblable à celle des URL classiques pour décrire des sources de données. En voici deux exemples :

```
jdbc:mckoi://localhost/  
jdbc:postgresql:COREJAVA
```

Ces URL JDBC spécifient une base de données McKoi ou PostgreSQL appelée COREJAVA. La syntaxe générale est la suivante :

```
jdbc:sous-protocole:paramètres
```

Dans cet exemple, le sous-protocole sélectionne le pilote spécifique qui sera en charge d'établir la connexion à la base de données.

Le format du paramètre *paramètres* dépend du sous-protocole utilisé.

Recherchez la documentation de votre fournisseur pour en connaître le format spécifique.

Etablir une connexion

Retrouvez les noms des classes de pilotes JDBC utilisées par votre fournisseur. Les noms habituels des pilotes sont les suivants :

```
org.postgresql.Driver  
com.mckoi.JDBCdriver
```

Retrouvez ensuite la bibliothèque dans laquelle se trouve le pilote, comme `pg74jdbc3.jar` ou `mkjdbc.jar`. Utilisez l'un des mécanismes suivants :

- Lancez vos programmes de bases de données avec l'argument de ligne de commande `-classpath`.
- Modifiez la variable d'environnement `CLASSPATH`.
- Copiez la bibliothèque de base de données dans le répertoire `jre/lib/ext`.

La classe `DriverManager` sélectionne les pilotes de base de données et crée une nouvelle connexion à la base de données. Cependant, avant que le gestionnaire de pilotes puisse activer un pilote, ce dernier doit être enregistré.

Votre programme peut définir une liste de pilotes dans la propriété système `jdbc.drivers`. Deux méthodes peuvent définir cette propriété.

Vous pouvez spécifier les propriétés à l'aide d'un argument de ligne de commande, comme

```
java -Djdbc.drivers=org.postgresql.Driver MyProg
```

Ou votre application peut définir les propriétés système avec un appel tel que

```
System.setProperty("jdbc.drivers", "org.postgresql.Driver");
```

Il est également possible de fournir plusieurs pilotes ; séparez-les par des deux-points, comme suit :

```
org.postgresql.Driver:com.mckoi.JDBCdriver
```

INFO

Vous pouvez aussi enregistrer manuellement un pilote en chargeant sa classe. Par exemple :

```
Class.forName("org.postgresql.Driver"); // forcer l'enregistrement du pilote
```

Vous devrez peut-être utiliser cette approche si le gestionnaire de pilotes ne parvient pas à charger le pilote, par exemple du fait des limitations d'un pilote particulier ou parce que votre programme s'exécute dans un conteneur, comme un moteur de servlet.

Une fois que les pilotes sont enregistrés, il faut encore ouvrir une connexion vers la base de données grâce à un code analogue à l'exemple suivant :

```
String url = "jdbc:postgresql:COREJAVA";
String username = "dbuser";
String password = "secret";
Connection conn = DriverManager.getConnection(url, username, password);
```

Le gestionnaire de pilotes parcourt alors les pilotes disponibles actuellement enregistrés pour trouver le pilote pouvant utiliser le sous-protocole spécifié dans l'URL de la base de données.

Pour notre programme d'exemple, il nous a semblé plus pratique d'avoir recours à un fichier de propriétés pour spécifier l'URL, le nom de l'utilisateur et le mot de passe, en plus du pilote de la base de données. Un fichier de propriétés typique renferme les informations suivantes :

```
jdbc.drivers=org.postgresql.Driver
jdbc.url=jdbc:postgresql:COREJAVA
jdbc.username=dbuser
jdbc.password=secret
```

156 *Au cœur de Java 2 - Fonctions avancées*

Voici le code permettant de lire ce fichier de propriétés et d'ouvrir la connexion vers la base de données :

```
Properties props = new Properties();
FileInputStream in = new FileInputStream("database.properties");
props.load(in);
in.close();
String drivers = props.getProperty("jdbc.drivers");
if (drivers != null) System.setProperty("jdbc.drivers", drivers);
String url = props.getProperty("jdbc.url");
String username = props.getProperty("jdbc.username");
String password = props.getProperty("jdbc.password");
return DriverManager.getConnection(url, username, password);
```

La méthode `getConnection` renvoie un objet `Connection`. Dans les sections à venir, vous verrez comment l'utiliser pour exécuter des requêtes SQL.

ASTUCE

Une bonne méthode pour déboguer les problèmes liés à JDBC est d'activer le mode trace de JDBC. Appelez la méthode `DriverManager.setLogWriter` pour envoyer des messages de trace à un `PrintWriter`. La sortie de la trace reflète l'activité détaillée de JDBC.

Tester l'installation de votre base de données

La première configuration JDBC peut se révéler quelque peu sinieuse. Il vous faudra disposer de plusieurs informations spécifiques à votre fournisseur, et la plus légère erreur pourra alors générer des messages d'erreur surprenants. Il est recommandé de tester en premier lieu la configuration de votre base de données sans JDBC. Procédez comme suit :

Etape 1. Démarrez la base de données. Avec McKoi, exécutez

```
java -jar mckoidb.jar
```

depuis le répertoire d'installation. Avec PostgreSQL, exécutez

```
postmaster -i -D /usr/share/pgsql/data
```

Etape 2. Configurez un utilisateur et une base de données. Avec McKoi, appelez

```
java -jar mckoidb.jar -create dbuser secret
```

Avec PostgreSQL, réalisez les commandes

```
createuser -d -U dbuser
```

```
createdb -U dbuser COREJAVA
```

Etape 3. Démarrez l'interpréteur SQL pour votre base de données. Avec McKoi, appelez

```
java -classpath mckoidb.jar com.mckoi.tools.JDBCQueryTool
```

Avec PostgreSQL, appelez

```
psql COREJAVA
```

Etape 4. Tapez les commandes SQL suivantes :

```
CREATE TABLE Salutations (Message CHAR(20))
INSERT INTO Salutations VALUES ('Bonjour')
SELECT * FROM Salutations
```

Désormais, vous devriez voir apparaître le message "Bonjour".

Etape 5. Procédez maintenant au nettoyage :

```
DROP TABLE Salutations
```

Dès que vous savez que l'installation de la base de données fonctionne, et que vous pouvez vous connecter à la base de données, cinq informations vous sont nécessaires :

- le nom d'utilisateur et le mot de passe de la base de données ;
- le nom de la base de données à utiliser (comme COREJAVA) ;
- le format de l'URL JDBC ;
- le nom du pilote JDBC ;
- l'emplacement des fichiers de bibliothèque contenant le code du pilote.

Les deux premiers éléments dépendent de la configuration de votre base de données. Les trois autres sont fournis dans la documentation JDBC du créateur de la base de données.

Pour McKoi, les valeurs habituelles sont :

- nom de l'utilisateur de la base de données = dbuser, mot de passe = secret ;
- nom de la base de données = (aucun) ;
- format de l'URL JDBC = jdbc:mckoi://localhost/ ;
- pilote JDBC = com.mckoi.JDBCdriver ;
- fichier de bibliothèque = mkjdbc.jar.

Pour PostgreSQL, vous pourriez avoir :

- nom de l'utilisateur de la base de données = dbuser, mot de passe = (aucun) ;
- nom de la base de données = COREJAVA ;
- format de l'URL JDBC = jdbc:postgresql:COREJAVA ;
- pilote JDBC = org.postgresql.Driver ;
- fichier de bibliothèque = pg74jdbc3.jar.

L'Exemple 3.1 constitue un petit programme de test que vous pouvez utiliser sur votre configuration JDBC. Préparez le fichier database.properties avec les informations rassemblées. Démarrez ensuite le programme de test avec la bibliothèque de pilotes sur le chemin des classes, comme

```
java -classpath .:cheminPilote TestDB
```

N'oubliez pas d'utiliser un deux-points au lieu d'un point-virgule comme séparateur de chemin sous Windows.

Ce programme exécute les mêmes instructions SQL que le test manuel. Si un message d'erreur SQL s'affiche, vous devrez continuer à travailler votre configuration. Il arrive très fréquemment que l'on commette quelques petites erreurs dans les majuscules, les noms de chemins, le format de l'URL JDBC ou dans la configuration de la base de données. Une fois que le programme de test affiche "Bonjour", vous pouvez passer à la section suivante.

158 *Au cœur de Java 2 - Fonctions avancées*

Exemple 3.1 : *TestDB.java*

```
import java.sql.*;
import java.io.*;
import java.util.*;

/**
 * Ce programme teste si la base de données et le pilote
 * JDBC sont correctement configurés.
 */
class TestDB
{
    public static void main (String args[])
    {
        try
        {
            runTest();
        }
        catch (SQLException ex)
        {
            while (ex != null)
            {
                ex.printStackTrace();
                ex = ex.getNextException();
            }
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }

    /**
     * Exécute un test en créant un tableau, en ajoutant une valeur, en
     * affichant le contenu du tableau, puis en supprimant le tableau.
     */
    public static void runTest()
        throws SQLException, IOException
    {
        Connection conn = getConnection();
        try
        {
            Statement stat = conn.createStatement();

            stat.execute("CREATE TABLE Salutations (Message CHAR(20))");
            stat.execute("INSERT INTO Salutations VALUES ('Bonjour')");

            ResultSet result = stat.executeQuery("
                SELECT * FROM Salutations");
            result.next();
            System.out.println(result.getString(1));
            stat.execute("DROP TABLE Salutations");
        }
        finally
    }
}
```

```
        {
            conn.close();
        }
    }

    /**
     * Obtient une connexion à partir des propriétés spécifiées
     * dans le fichier database.properties
     * @return la connexion à la base de données
     */
    public static Connection getConnection()
        throws SQLException, IOException
    {
        Properties props = new Properties();
        FileInputStream in = new FileInputStream("database.properties");
        props.load(in);
        in.close();

        String drivers = props.getProperty("jdbc.drivers");
        if (drivers != null)
            System.setProperty("jdbc.drivers", drivers);
        String url = props.getProperty("jdbc.url");
        String username = props.getProperty("jdbc.username");
        String password = props.getProperty("jdbc.password");

        return DriverManager.getConnection(url, username, password);
    }
}
```

Exécuter des commandes SQL

Pour exécuter une commande SQL, il faut commencer par créer un objet `Statement`. Pour créer des objets `Statement`, utilisez l'objet `Connection` que vous avez obtenu de l'appel à `DriverManager.getConnection`.

```
Statement stat = conn.createStatement();
```

Il faut ensuite placer l'instruction à exécuter dans une chaîne, par exemple :

```
String command = "UPDATE Livres "
    + "SET Prix = Prix - 5.00"
    + "WHERE Titre NOT LIKE '%Introduction%'";
```

Appelez alors la méthode `executeUpdate` de la classe `Statement` :

```
stat.executeUpdate(command);
```

La méthode `executeUpdate` renvoie le nombre de lignes qui ont été affectées par la commande SQL. Par exemple, l'appel à `executeUpdate` dans l'exemple précédent renvoie le nombre de livres dont le prix a été diminué de 5 €.

Les commandes peuvent être des actions comme `INSERT`, `UPDATE` et `DELETE`, ou bien des commandes de définition de données comme `CREATE TABLE` ou `DROP TABLE`. Cependant, vous devez utiliser la méthode `executeQuery` pour exécuter des requêtes `SELECT`. Il existe également une instruction `execute` permettant d'exécuter des instructions SQL arbitraires. Elle est souvent utilisée, mais uniquement pour les requêtes fournies de manière interactive par un utilisateur.

160 Au cœur de Java 2 - Fonctions avancées

Lorsque vous exécutez une requête, c'est le résultat qui vous intéresse. L'objet `executeQuery` renvoie un objet du type `ResultSet` que vous utilisez pour parcourir le résultat, ligne par ligne.

```
ResultSet rs = stat.executeQuery("SELECT * FROM Livres")
```

La boucle permettant d'analyser les résultats ressemble à ceci :

```
while (rs.next())
{
    examine une ligne de l'ensemble de résultats
}
```

ATTENTION

Le protocole d'itération de la classe `ResultSet` est légèrement différent du protocole de l'interface `Iterator` que nous avons abordé au Chapitre 2. Ici, l'itérateur est placé *avant* la première ligne. Vous devez appeler la méthode `next` une première fois pour vous positionner sur la première ligne.

INFO

L'ordre des lignes dans un jeu de résultats est totalement arbitraire. A moins que vous n'ordonniez spécifiquement le résultat à l'aide de la clause `ORDER BY`, n'attachez aucune importance à l'ordre des lignes.

Pour examiner chaque ligne, il faudra lire le contenu des champs. Il existe beaucoup de méthodes accessoires qui vous fourniront ces informations.

```
String isbn = rs.getString(1);
double Prix = rs.getDouble("Prix");
```

Il existe des accessoires pour divers *types*, comme `getString` et `getDouble`. Chaque accessoire possède deux formes. La première accepte un argument numérique et la seconde prend une chaîne en paramètre. Lorsque vous fournissez un argument numérique, vous faites référence à la colonne correspondant à cette valeur. Par exemple, `rs.getString(1)` renvoie la valeur de la première colonne de la ligne courante.

ATTENTION

Contrairement aux indices de tableau, les numéros de colonnes de base de données commencent à 1.

Lorsque vous passez une chaîne en argument, vous faites référence à la colonne portant le même nom. Par exemple, `rs.getDouble("Prix")` renvoie la valeur de la colonne dont le nom est "Prix". L'utilisation d'un argument numérique est un peu plus efficace, mais les chaînes de caractères rendent le code plus lisible et plus facile à maintenir.

Chaque méthode `get` effectue une conversion de type adéquate lorsque le type de la méthode ne correspond pas au type de la colonne. Par exemple, `rs.getString("Prix")` convertit la valeur à virgule flottante de la colonne `Prix` en chaîne de caractères.

INFO

Les types de données de SQL et de Java ne sont pas strictement les mêmes. Consultez le Tableau 3.6 pour obtenir la liste des types de données SQL et leur équivalent Java.

Tableau 3.6 : Les types de données SQL et leur équivalent Java

Type de données SQL	Type de données Java
INTEGER ou INT	int
SMALLINT	short
NUMERIC(<i>m, n</i>), DECIMAL(<i>m, n</i>) ou DEC(<i>m, n</i>)	java.sql.Numeric
FLOAT(<i>n</i>)	double
REAL	float
DOUBLE	double
CHARACTER(<i>n</i>) ou CAR(<i>n</i>)	String
VARCHAR(<i>n</i>)	String
BOOLEAN	boolean
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BLOB	java.sql.Blob
CLOB	java.sql.Clob
ARRAY	java.sql.Array

Types SQL avancés

En plus des nombres, des chaînes, et des dates, certaines bases de données peuvent enregistrer de *grands objets* comme des images. En SQL, ces grands objets binaires sont appelés des BLOB, et les grands objets contenant des caractères sont appelés des CLOB. Les méthodes `getBlob` et `getClob` renvoient des objets de type `java.sql.Blob` et `java.sql.Clob`. Ces classes possèdent des méthodes pour lire les octets ou les caractères des grands objets.

Un objet SQL ARRAY est constitué d'une suite de valeurs. Par exemple, dans un tableau `Student`, vous pouvez avoir une colonne `Scores` qui est un `ARRAY OF INTEGER`. La méthode `getArray` renvoie un objet de type `java.sql.Array` (qui est différent de la classe `java.lang.reflect.Array` que nous avons vue dans le Volume 1). L'interface `java.sql.Array` possède des méthodes pour lire les valeurs d'un Array.

Lorsque vous travaillez avec un `blob` ou un `array` dans une base de données, ils ne sont lus dans la base de données que lorsque vous demandez une valeur particulière en faisant partie. Il s'agit d'une

162 Au cœur de Java 2 - Fonctions avancées

amélioration très importante au niveau des performances, puisque les données manipulées peuvent être très grosses.

Certaines bases de données sont même capables de stocker des structures de données définies par l'utilisateur. JDBC supporte un mécanisme pour transformer automatiquement les types structurés SQL en objets Java.

Nous n'étudierons pas plus les blob, les array ou les types définis par l'utilisateur. Vous trouverez de plus amples informations sur ces sujets dans l'ouvrage (en langue anglaise) *JDBC(TM) API Tutorial and Reference: Universal Data Access for the Java 2 Platform* (2^e édition) par Seth White, Maydene Fisher, Rick Cattell, Graham Hamilton et Mark Hapner (Addison-Wesley, 1999).

java.sql.DriverManager 1.1

- `static Connection getConnection(String url, String user, String password)`
Etablit une connexion à la base de données spécifiée et renvoie un objet `Connection`.

java.sql.Connection 1.1

- `Statement createStatement()`
Crée un objet `Statement` qui peut être utilisé pour exécuter des requêtes SQL ou des mises à jour, sans paramètre.
- `void close()`
Ferme immédiatement la connexion courante et les ressources JDBC qu'il a créées.

java.sql.Statement 1.1

- `ResultSet executeQuery(String sqlQuery)`
Exécute l'instruction SQL spécifiée dans la chaîne et renvoie un objet `ResultSet` comme résultat de la requête.
- `int executeUpdate(String sqlStatement)`
Exécute la commande INSERT, UPDATE ou DELETE spécifiée dans la chaîne. Exécute aussi des DLL (*Data Definition Language*, ou langage de définition de données) comme CREATE TABLE. Renvoie le nombre d'enregistrements modifiés ou -1 pour une instruction sans compte de mise à jour.
- `boolean execute(String sqlStatement)`
Exécute l'instruction SQL spécifiée par la chaîne. Renvoie `true` si l'instruction renvoie un ensemble de résultats et `false` dans le cas contraire. Utilisez la méthode `getResultSet` ou `getUpdateCount` pour obtenir le résultat de l'instruction.
- `int getUpdateCount()`
Renvoie le nombre d'enregistrements affectés par l'instruction de mise à jour précédente ou -1 si l'instruction précédente ne possédait pas de compte de mises à jour. Appelez cette méthode une fois seulement pour chaque instruction exécutée.
- `ResultSet getResultSet()`
Renvoie l'ensemble de résultats de l'instruction de requête précédente, ou `null` si l'instruction précédente ne possédait pas d'ensemble de résultats. Appelez cette méthode une fois seulement pour chaque instruction exécutée.

- `void close()`
Ferme cet objet `Statement` et son jeu de résultats associé.

`java.sql.ResultSet` 1.1

- `boolean next()`
Avance d'une ligne dans l'ensemble de résultats. Renvoie `false` après la dernière ligne. Notez que vous devez appeler cette méthode pour vous positionner sur la première ligne.
- `Xxx getXxx(int columnNumber)`
- `Xxx getXxx(String columnName)`
(`Xxx` est un type comme `int`, `double`, `String`, `Date`, etc.)
Renvoie la valeur située dans le numéro ou le nom de colonne donné, convertie dans le type spécifié. Toutes les conversions de type ne sont pas autorisées. Consultez la documentation pour plus de détails.
- `int findColumn(String columnName)`
Fournit l'indice de la colonne associée à un nom de colonne spécifié.
- `void close()`
Ferme immédiatement l'ensemble de résultats courant.

`java.sql.SQLException` 1.1

- `String getSQLState()`
Récupère "l'état SQL", un code à cinq chiffres associé à l'erreur.
- `int getErrorCode()`
Fournit le code d'exception spécifique du vendeur.
- `SQLException getNextException()`
Renvoie l'exception suivant celle-ci. Elle peut contenir plus d'informations sur la cause de l'erreur.

Gestion des connexions, instructions et jeux de résultats

Chaque objet `Connection` est en mesure de créer un ou plusieurs objets `Statement`. Vous pouvez utiliser le même objet `Statement` pour plusieurs commandes et requêtes qui ne soient pas liées. Or une instruction possède *au moins* un jeu de résultats ouvert. Si vous émettez plusieurs requêtes dont vous analyserez les résultats simultanément, vous devez utiliser plusieurs objets `Statement`.

Sachez toutefois qu'au moins une bibliothèque de base de données standard (Microsoft SQL Server) possède un pilote JDBC n'autorisant qu'une instruction active à la fois. Utilisez la méthode `getMaxStatements` de la classe `DatabaseMetaData` pour trouver le nombre d'instructions simultanément ouvertes acceptées par votre pilote JDBC.

Cela peut vous paraître limitatif mais, dans la pratique, il vaut mieux éviter les jeux de résultats simultanés. Si les jeux de résultats sont liés, vous devriez pouvoir émettre une requête combinée et n'analyser qu'un seul résultat. Il vaut bien mieux laisser la base de données combiner les requêtes que permettre à un programme Java de passer en revue plusieurs jeux de résultats.

164 Au cœur de Java 2 - Fonctions avancées

Une fois terminé votre travail avec `ResultSet`, `Statement` ou `Connection`, appelez immédiatement la méthode `close`. Ces objets utilisent de grandes structures de données, il ne faut donc pas attendre que le ramasse-miettes s'en charge.

La méthode `close` d'un objet `Statement` ferme automatiquement le jeu de résultats associé si l'instruction dispose d'un jeu de résultats ouvert. De même, la méthode `close` de la classe `Connection` ferme toutes les instructions de la connexion.

Si vos connexions sont courtes, vous devrez vous préoccuper de la fermeture des instructions et des jeux de résultats. Vérifiez qu'aucun objet de connexion ne puisse rester ouvert, en plaçant l'instruction `close` dans un bloc `finally` :

```
Connection conn = . . . ;
try
{
    Statement stat = conn.createStatement();
    ResultSet result = stat.executeQuery(queryString);
    résultat de la requête de traitement
}
finally
{
    conn.close();
}
```

ASTUCE

N'utilisez le bloc `try/finally` que pour fermer la connexion et utilisez un bloc `try/catch` séparé pour traiter les exceptions. La séparation des blocs `try` simplifie la lecture du code.

Remplir une base de données

Nous allons maintenant écrire notre premier vrai programme JDBC. Il serait d'ailleurs intéressant de tester quelques-unes des requêtes que nous venons de voir. Malheureusement, il y a un problème : il n'existe pour l'instant aucune donnée dans la base de données. Et vous ne trouverez aucun fichier de base de données sur le CD-ROM que vous pourriez simplement copier sur votre disque dur, pour que le programme de base de données le lise, car aucun format de fichier de base de données ne vous permet d'échanger des bases de données relationnelles SQL d'un vendeur à un autre. SQL n'a strictement rien à voir avec les fichiers. Il s'agit d'un langage pour effectuer des requêtes de mises à jour dans une base de données. L'exécution de ces instructions et les formats de fichier utilisés dépendent entièrement de l'implémentation de la base de données. Les concepteurs de bases de données font des efforts pour parvenir à des stratégies de requêtes optimisées et pour l'enregistrement des données, et chacune de leurs approches est différente des autres. Par conséquent, les instructions SQL sont portables, mais la représentation des données sous-jacentes ne l'est pas.

Pour supprimer notre problème, nous fournissons un petit ensemble de données présentées dans une série de fichiers texte. Nous vous fournissons également un programme capable de lire un fichier composé d'instructions SQL, une instruction sur chaque ligne, puis de les exécuter.

Plus particulièrement, le programme lit des données dans un fichier texte respectant le format suivant :

```
CREATE TABLE Editeur (Editeur_Id CHAR(6), Nom CHAR(30), URL CHAR(80))
INSERT INTO Editeurs VALUES ('0201', 'Addison-Wesley', 'www.aw-bc.com')
INSERT INTO Editeurs VALUES ('0471', 'John Wiley & Sons', 'www.wiley.com')
. . .
```

A la fin de cette section, vous trouverez le code de ce programme qui lit un fichier d'instructions SQL et qui exécute ces instructions. Même si son implémentation ne vous intéresse pas, vous devez l'exécuter pour pouvoir exécuter les exemples plus intéressants de la fin du chapitre. Pour exécuter ce programme, suivez les instructions suivantes :

```
java -classpath .:cheminPilote ExecSQL Livres.sql
java -classpath .:cheminPilote ExecSQL Auteurs.sql
java -classpath .:cheminPilote ExecSQL Editeurs.sql
java -classpath .:cheminPilote ExecSQL LivresAuteurs.sql
```

Avant d'exécuter le programme, vérifiez que le fichier `database.properties` soit correctement configuré pour votre environnement.

Les étapes suivantes décrivent brièvement le programme `ExecSQL` :

1. Se connecte à la base de données. La méthode `getConnection` lit les propriétés dans le fichier `database.properties` et ajoute la propriété `jdbc.drivers` aux propriétés système. Le gestionnaire de pilotes se sert de la propriété `jdbc.drivers` pour charger le pilote de base de données approprié. La méthode `getConnection` se sert des propriétés `jdbc.url`, `jdbc.username` et `jdbc.password` pour ouvrir une connexion vers la base de données.
2. Ouvre le fichier contenant les commandes SQL. Lorsque aucun nom de fichier n'est fourni, demande à l'utilisateur de saisir les commandes sur la console.
3. Exécute chaque commande avec la méthode générique `execute`. Si une commande renvoie `true`, cela signifie qu'elle disposait d'un ensemble de résultats. Les quatre fichiers SQL que nous avons fournis pour la base de données des livres se terminaient tous par l'instruction `SELECT *` afin que vous puissiez voir que les données ont bien été insérées.
4. S'il existe un ensemble de résultats, affiche le résultat. Etant donné qu'il s'agit d'un ensemble de résultats générique, nous devons utiliser des *métadonnées* pour retrouver le nombre de colonnes dont dispose le résultat. Vous en saurez plus sur les métadonnées en lisant la suite de ce chapitre.
5. En cas d'exceptions SQL, affiche l'exception et toute exception chaînée qui peut y être contenue.
6. Ferme la connexion à la base de données.

L'Exemple 3.2 donne le code correspondant à ces étapes.

Exemple 3.2 : *ExecSQL.java*

```
import java.io.*;
import java.util.*;
import java.sql.*;

/**
 * Exécute toutes les instructions SQL d'un fichier.
 * Appelez ce programme comme suit
 * java -classpath driverPath:. ExecSQL commandFile
 */
```

166 *Au cœur de Java 2 - Fonctions avancées*

```

class ExecSQL
{
    public static void main (String args[])
    {
        try
        {
            Scanner in;
            if (args.length == 0)
                in = new Scanner(System.in);
            else
                in = new Scanner(new File(args[0]));

            Connection conn = getConnection();
            try
            {
                Statement stat = conn.createStatement();

                while (true)
                {
                    if (args.length == 0) System.out.println(
                        "Entrer une commande ou EXIT pour quitter :");

                    if (!in.hasNextLine()) return;

                    String line = in.nextLine();
                    if (line.equalsIgnoreCase("EXIT")) return;
                    try
                    {
                        boolean hasResultSet = stat.execute(line);
                        if (hasResultSet)
                            showResultSet(stat);
                    }
                    catch (SQLException e)
                    {
                        while (e != null)
                        {
                            e.printStackTrace();
                            e = e.getNextException();
                        }
                    }
                }
            }
            finally
            {
                conn.close();
            }
        }
        catch (SQLException e)
        {
            while (e != null)
            {
                e.printStackTrace ();
                e = e.getNextException();
            }
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}

```

```

    }

    /**
     * Obtient une connexion des propriétés spécifiées
     * dans le fichier database.properties
     * @return la connexion à la base de données
     */
    public static Connection getConnection()
        throws SQLException, IOException
    {
        Properties props = new Properties();
        FileInputStream in = new FileInputStream("database.properties");
        props.load(in);
        in.close();

        String drivers = props.getProperty("jdbc.drivers");
        if (drivers != null) System.setProperty("jdbc.drivers", drivers);
        String url = props.getProperty("jdbc.url");
        String username = props.getProperty("jdbc.username");
        String password = props.getProperty("jdbc.password");

        return DriverManager.getConnection(url, username, password);
    }

    /**
     * Affiche un ensemble de résultats.
     * @param stat l'instruction dont l'ensemble de résultats doit
     * être affiché
     */
    public static void showResultSet(Statement stat)
        throws SQLException
    {
        ResultSet result = stat.getResultSet();
        ResultSetMetaData metaData = result.getMetaData();
        int columnCount = metaData.getColumnCount();

        for (int i = 1; i <= columnCount; i++)
        {
            if (i > 1) System.out.print(", ");
            System.out.print(metaData.getColumnLabel(i));
        }
        System.out.println();

        while (result.next())
        {
            for (int i = 1; i <= columnCount; i++)
            {
                if (i > 1) System.out.print(", ");
                System.out.print(result.getString(i));
            }
            System.out.println();
        }
        result.close();
    }
}

```

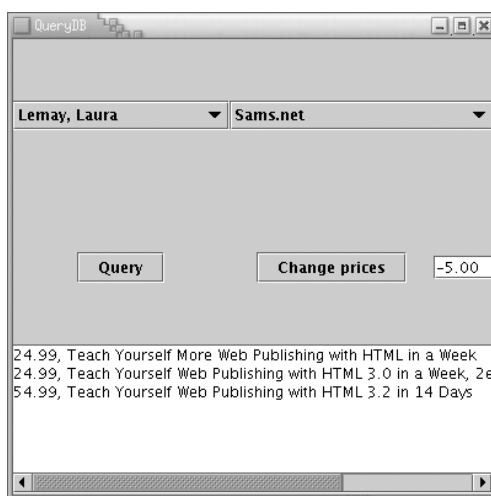
Exécution de requêtes

Dans cette section, nous allons écrire un programme qui exécute des requêtes dans la base de données COREJAVA. Pour que ce programme puisse fonctionner correctement, vous devez avoir rempli cette base de données avec des tableaux, comme dans la section précédente.

La Figure 3.6 montre une action de l'application QueryDB.

Figure 3.6

L'application QueryDB.



Vous pouvez sélectionner un auteur et un éditeur, ou ne pas remplir l'un de ces deux champs. Cliquez sur "Requête" (Query) ; tous les livres correspondant à votre sélection seront affichés dans la boîte de texte.

Vous pouvez également modifier les données de la base de données. Sélectionnez un éditeur et tapez une somme dans la zone de texte près du bouton "Changer les prix". Lorsque vous cliquez sur ce bouton, tous les prix de cet éditeur sont modifiés en fonction de la valeur saisie et la zone de texte affiche un message indiquant le nombre de données modifiées. Cependant, pour éviter les erreurs de manipulation, vous ne pouvez pas modifier tous les prix d'un seul coup. Le champ Auteur est ignoré lorsque vous saisissez une modification de prix. Après un changement de prix, vous pourrez exécuter une nouvelle requête pour vérifier les nouveaux prix.

Instructions préparées

Dans ce programme, nous faisons appel à une nouvelle technique, les *instructions préparées*. Considérons par exemple une requête cherchant tous les livres d'un éditeur donné, indépendamment de l'auteur. La requête SQL correspondante est :

```
SELECT Livres.Prix, Livres.Titre
FROM Livres, Editeurs
WHERE Livres.Editeur_Id = Editeurs.Editeur_Id
AND Editeurs.Nom = le nom issu du menu déroulant
```

Plutôt que générer une nouvelle instruction de requête chaque fois que l'utilisateur exécute ce type de requête, nous pouvons *préparer* une requête paramétrée, et nous en servir chaque fois avec une chaîne différente. Cette technique apporte une certaine amélioration au niveau des performances. Chaque fois que la base de données exécute une requête, elle commence par déterminer une stratégie lui permettant d'exécuter la requête de manière efficace. En préparant une requête et en l'utilisant par la suite, vous vous assurez que la tâche de répartition n'est faite qu'une seule fois.

Chaque paramètre d'une requête préparée est spécifié par un ?. S'il y a plus d'une variable, vous devez garder une trace de la position des ? lorsque vous choisissez leurs valeurs. Par exemple, notre requête préparée devient :

```
String EditeurQuery =  
    "SELECT Livres.Prix, Livres.Titre" +  
    " FROM Livres, Editeurs" +  
    " WHERE Livres.Editeur_Id = Editeurs.Editeur_Id AND Editeurs.Nom = ?";  
PreparedStatement EditeurQueryStat = conn.prepareStatement(EditeurQuery);
```

Avant d'exécuter une instruction préparée, vous devez affecter les valeurs aux variables avec la méthode `set`. Comme pour les méthodes `ResultSetGet`, il existe une méthode `set` pour plusieurs types de données. Ici, nous voulons affecter une chaîne à un nom d'éditeur.

```
EditeurQueryStat.setString(1, Editeur);
```

Le premier argument correspond au numéro de position de la variable que nous voulons définir. La position 1 représente le premier ?. Le second argument est la valeur que nous voulons affecter à la variable.

Si vous réutilisez une requête préparée que vous avez déjà exécutée, et que cette requête possède plusieurs variables, toutes ces variables restent inchangées tant qu'elles ne sont pas modifiées par une méthode `set`. Cela signifie que vous avez uniquement besoin d'appeler une méthode `setXxx` pour les variables qui doivent être modifiées d'une requête à l'autre.

Une fois que toutes les variables ont été définies, vous pouvez exécuter la requête :

```
ResultSet rs = EditeurQueryStat.executeQuery();
```

ASTUCE

Même si vous n'êtes pas préoccupé par les performances, utilisez les instructions préparées dès que votre requête implique des variables. Si vous construisez une requête à la main, vous devrez prendre garde aux caractères spéciaux (comme les apostrophes). Cela pose plus de problèmes qu'utiliser une instruction préparée.

La modification des prix est implémentée avec une instruction `UPDATE`. Notez que nous appelons `executeUpdate` et non `executeQuery`, puisque l'instruction `UPDATE` ne renvoie aucun ensemble de résultats. La valeur de retour d'`executeUpdate` est le nombre de lignes modifiées. Nous affichons ce nombre dans la zone de texte.

```
int r = priceUpdateStmt.executeUpdate();  
result.setText(r + " enregistrements modifiés");
```


170 Au cœur de Java 2 - Fonctions avancées

INFO

Un objet `PreparedStatement` devient non valide après la fermeture de l'objet `Connection` associé. Toutefois, de nombreux pilotes de bases de données placent automatiquement en *cache* des instructions préparées. Si la même requête est préparée deux fois, la base de données réutilise simplement la stratégie des requêtes. Ne vous inquiétez donc pas de la surcharge de l'appel de `prepareStatement`.

Les étapes suivantes décrivent brièvement le fonctionnement de ce programme.

1. Arrange les composants du cadre, avec une mise en page en quadrillage (voir le Chapitre 9 du Volume 1).
2. Remplit les zones de texte Auteur et Editeur en exécutant deux requêtes qui renvoient tous les noms d'auteurs et d'éditeurs de la base de données.
3. Lorsque l'utilisateur clique sur le bouton "Requête", le programme identifie le type de requête à effectuer, parmi les quatre requêtes préparées. Si la requête est exécutée pour la première fois, la variable d'instruction préparée vaut `null`, et l'instruction préparée est construite. Puis, les valeurs sont placées dans la requête et celle-ci est exécutée.

Les requêtes concernant les auteurs sont plus complexes. Comme un livre peut être écrit par plusieurs auteurs, le tableau `LivresAuteurs` fournit la correspondance entre les auteurs et les livres. Par exemple, le livre dont le numéro d'ISBN est 0-201-96426-0 possède deux auteurs dont les codes sont DATE et DARW. Le tableau `LivresAuteurs` possède en effet les deux lignes suivantes :

```
0-201-96426-0, DATE, 1
0-201-96426-0, DARW, 2
```

La troisième colonne fournit l'ordre des auteurs. Nous ne pouvons pas nous contenter d'utiliser la position des enregistrements dans ce tableau : il n'existe aucun ordre particulier pour les lignes dans un tableau relationnel. Par conséquent, la requête doit associer les tableaux `Livres`, `LivresAuteurs` et `Auteurs` pour comparer le nom de l'auteur avec celui qui a été sélectionné par l'utilisateur.

```
SELECT Livres.Prix, Livres.Titre FROM Livres, LivresAuteurs, Auteurs,
      Editeurs
WHERE Auteurs.Auteur_Id = LivresAuteurs.Auteur_Id AND
      LivresAuteurs.ISBN = Livres.ISBN
AND Livres.Editeur_Id = Editeurs.Editeur_Id AND Auteurs.Nom = ? AND
      Editeurs.Nom = ?
```

ASTUCE

Certains programmeurs Java évitent d'utiliser des instructions SQL complexes comme celle-ci. Un raccourci bizarrement commun, mais très inefficace, consiste à écrire une grande quantité de code Java qui parcourt plusieurs ensembles de résultats. Toutefois, la base de données parvient bien mieux à exécuter un code de requête qu'un programme Java. Suivez donc ce conseil : si vous pouvez le faire en SQL, ne le faites pas en Java.

4. Les résultats de cette requête sont placés dans la zone de texte des résultats.
5. Lorsque l'utilisateur clique sur "Modification des prix", la commande de modification est construite puis exécutée. Cette commande est assez complexe parce que la clause `WHERE` de l'instruction `UPDATE` a besoin d'un *code* d'éditeur alors que nous ne connaissons que le *nom* de l'éditeur. Ce problème est résolu par une sous-requête imbriquée.

```
UPDATE Livres
SET Prix = Prix + ?
WHERE Livres.Editeur_Id = (SELECT Editeur_Id FROM Editeurs WHERE Nom = ?)
```

6. Nous initialisons la connexion et les objets d'instructions dans le constructeur. Nous les conserverons pour toute la durée du programme. Juste avant la fin du programme, nous appelons la méthode "de fermeture de fenêtre" et ces objets sont fermés.

```
class QueryDBFrame extends JFrame
{
    public QueryDBFrame()
    {
        conn =getConnection();
        stat = conn.createStatement();
        . . .
        add(new
            WindowAdapter()
            {
                public void windowClosing(WindowEvent event)
                {
                    try
                    {
                        stat.close();
                        conn.close();
                    }
                    catch(SQLException e)
                    {
                        while (e != null)
                        {
                            e.printStackTrace();
                            e = e.getNextException();
                        }
                    }
                }
            });
    }
    . . .
    private Connection conn;
    private Statement stat;
}
```

L'Exemple 3.3 fournit le code complet de ce programme.

Exemple 3.3 : *QueryDB.java*

```
import java.net.*;
import java.sql.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;

/**
 * Ce programme présente plusieurs requêtes complexes de bases de
 * données.
 */
public class QueryDB
{
    . . .
}
```

172 Au cœur de Java 2 - Fonctions avancées

```
public static void main(String[] args)
{
    JFrame frame = new QueryDBFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
}

/**
 * Ce bloc affiche des menus déroulants pour les paramètres de requête,
 * une zone de texte pour les résultats de la commande et des boutons
 * pour lancer une requête et une mise à jour.
 */
class QueryDBFrame extends JFrame
{
    public QueryDBFrame()
    {
        setTitre("QueryDB");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
        setLayout(new GridBagLayout());

        Auteurs = new JComboBox();
        Auteurs.setEditable(false);
        Auteurs.addItem("Non précisé");

        Editeurs = new JComboBox();
        Editeurs.setEditable(false);
        Editeurs.addItem("Non précisé");

        result = new JTextArea(4, 50);
        result.setEditable(false);

        PrixChange = new JTextField(8);
        PrixChange.setText("-5.00");

        try
        {
            conn = getConnection();
            Statement stat = conn.createStatement();

            String query = "SELECT Nom FROM Auteurs";
            ResultSet rs = stat.executeQuery(query);
            while (rs.next())
                Auteurs.addItem(rs.getString(1));
            rs.close();

            query = "SELECT Nom FROM Editeurs";
            rs = stat.executeQuery(query);
            while (rs.next())
                Editeurs.addItem(rs.getString(1));
            rs.close();
            stat.close();
        }
        catch(SQLException e)
        {
            result.setText("");
            while (e != null)
            {
                result.append("" + e);
            }
        }
    }
}
```

```

        e = e.getNextException();
    }
}
catch (IOException e)
{
    result.setText("" + e);
}

// Nous utilisons la classe GBC du Chapitre 9 du Volume 1
add(Auteurs, new GBC(0, 0, 2, 1));

add(Editeurs, gbc, 2, 0, 2, 1);

JButton queryButton = new JButton("Requête");
queryButton.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            executeQuery();
        }
    });
add(queryButton, new GBC(0, 1, 1, 1).setInsets(3));

JButton changeButton = new JButton("Modification des prix");
changeButton.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            changePrices();
        }
    });
add(changeButton, new GBC(2, 1, 1, 1).setInsets(3));

add(PrixChange, new GBC(3, 1, 1, 1).setFill(GBC.HORIZONTAL));

add(new JScrollPane(result), new GBC(0, 2, 4, 1).
    setFill(GBC.BOTH).setWeight(100, 100));

addWindowListener(new
    WindowAdapter()
    {
        public void windowClosing(WindowEvent event)
        {
            try
            {
                if (conn != null) conn.close();
            }
            catch (SQLException e)
            {
                while (e != null)
                {
                    e.printStackTrace();
                    e = e.getNextException();
                }
            }
        }
    });

```

174 Au cœur de Java 2 - Fonctions avancées

```

    }

    /**
     * Exécute la requête sélectionnée.
     */
    private void executeQuery()
    {
        ResultSet rs = null;
        try
        {
            String Auteur = (String) Auteurs.getSelectedItemAt();
            String Editeur = (String) Editeurs.getSelectedItemAt();
            if (!Auteur.equals("Non précisé")
                && !Editeur.equals("Non précisé"))
            {
                if (AuteurEditeurQueryStmt == null)
                    AuteurEditeurQueryStmt
                        = conn.prepareStatement(AuteurEditeurQuery);
                AuteurEditeurQueryStmt.setString(1, Auteur);
                AuteurEditeurQueryStmt.setString(2, Editeur);
                rs = AuteurEditeurQueryStmt.executeQuery();
            }
            else if (!Auteur.equals("Non précisé")
                && Editeur.equals("Non précisé"))
            {
                if (AuteurQueryStmt == null)
                    AuteurQueryStmt
                        = conn.prepareStatement(AuteurQuery);
                AuteurQueryStmt.setString(1, Auteur);
                rs = AuteurQueryStmt.executeQuery();
            }
            else if (Auteur.equals("Non précisé")
                && !Editeur.equals("Non précisé"))
            {
                if (EditeurQueryStmt == null)
                    EditeurQueryStmt
                        = conn.prepareStatement(EditeurQuery);
                EditeurQueryStmt.setString(1, Editeur);
                rs = EditeurQueryStmt.executeQuery();
            }
            else
            {
                if (allQueryStmt == null)
                    allQueryStmt = conn.prepareStatement(allQuery);
                rs = allQueryStmt.executeQuery();
            }

            result.setText("");
            while (rs.next())
            {
                result.append(rs.getString(1));
                result.append(", ");
                result.append(rs.getString(2));
                result.append("\n");
            }
            rs.close();
        }
        catch (Exception e)
        {
    
```

```

        result.setText("");
        while (e != null)
        {
            result.append("" + e);
            e = e.getNextException();
        }
    }

    /**
     * Exécute une instruction de mise à jour pour modifier les prix.
     */
    public void changePrices()
    {
        String Editeur = (String)Editeurs.getSelectedItemAt();
        if (Editeur.equals("Non précisé"))
        {
            result.setText
                ("Je suis désolé, je ne peux pas faire cela.");
            return;
        }
        try
        {
            if (prixUpdateStmt == null)
                prixUpdateStmt = conn.prepareStatement(prixUpdate);
            prixUpdateStmt.setString(1, prixChange.getText());
            prixUpdateStmt.setString(2, Editeur);
            int r = prixUpdateStmt.executeUpdate();
            result.setText(r + " enregistrements mis à jour.");
        }
        catch(SQLException e)
        {
            result.setText("");
            while (e != null)
            {
                result.append("" + e);
                e = e.getNextException();
            }
        }
    }

    /**
     * Obtient une connexion depuis les propriétés spécifiées
     * dans le fichier database.properties
     * @return la connexion à la base de données
     */
    public static Connection getConnection()
        throws SQLException, IOException
    {
        Properties props = new Properties();
        FileInputStream in
            = new FileInputStream("database.properties");
        props.load(in);
        in.close();

        String drivers = props.getProperty("jdbc.drivers");
        if (drivers != null)
            System.setProperty("jdbc.drivers", drivers);
        String url = props.getProperty("jdbc.url");
    }

```

176 Au cœur de Java 2 - Fonctions avancées

```

        String username = props.getProperty("jdbc.username");
        String password = props.getProperty("jdbc.password");

        return DriverManager.getConnection(url, username, password);
    }

    public static final int DEFAULT_WIDTH = 400;
    public static final int DEFAULT_HEIGHT = 400;

    private JComboBox Auteurs;
    private JComboBox Editeurs;
    private JTextField PrixChange;
    private JTextArea result;
    private Connection conn;
    private PreparedStatement AuteurQueryStmt;
    private PreparedStatement AuteurEditeurQueryStmt;
    private PreparedStatement EditeurQueryStmt;
    private PreparedStatement allQueryStmt;
    private PreparedStatement priceUpdateStmt;

    private static final String AuteurEditeurQuery =
        "SELECT Livres.Prix, Livres.Titre FROM Livres, LivresAuteurs,
        Auteurs, Editeurs" +
        " WHERE Auteurs.Auteur_Id = LivresAuteurs.Auteur_Id AND
        LivresAuteurs.ISBN = Livres.ISBN" +
        " AND Livres.Editeur_Id = Editeurs.Editeur_Id AND Auteurs.Nom
        = ?" +
        " AND Editeurs.Nom = ?";

    private static final String AuteurQuery =
        "SELECT Livres.Prix, Livres.Titre FROM Livres, LivresAuteurs,
        Auteurs" +
        " WHERE Auteurs.Auteur_Id = LivresAuteurs.Auteur_Id AND
        LivresAuteurs.ISBN = Livres.ISBN" +
        " AND Auteurs.Nom = ?";

    private static final String EditeurQuery =
        "SELECT Livres.Prix, Livres.Titre FROM Livres, Editeurs" +
        " WHERE Livres.Editeur_Id = Editeurs.Editeur_Id AND Editeurs.Nom
        = ?";

    private static final String allQuery = "SELECT Livres.Prix,
        Livres.Titre FROM Livres";

    private static final String prixUpdate =
        "UPDATE Livres " + "SET Prix = Prix + ? " +
        " WHERE Livres.Editeur_Id = (SELECT Editeur_Id FROM Editeurs WHERE
        Nom = ?)";
}

```

java.sql.Connection 1.1

- `PreparedStatement prepareStatement(String sql)`

Renvoie un objet `PreparedStatement` contenant l'instruction précompilée. La chaîne `sql` contient une instruction SQL qui renferme un ou plusieurs emplacements de paramètres représentés par des caractères `?`.

java.sql.PreparedStatement 1.1

- `void setXxx(int n, Xxx x)`
(Xxx est un type comme `int`, `double`, `String`, `Date`, etc.)
Met la valeur du n-ième paramètre à x.
- `void clearParameters()`
Efface tous les paramètres courants d'une instruction préparée.
- `ResultSet executeQuery()`
Exécute une requête SQL préparée et renvoie un objet `ResultSet`.
- `int executeUpdate()`
Exécute une instruction SQL `INSERT`, `UPDATE` ou `DELETE` représentée par l'objet `PreparedStatement`. Renvoie le nombre de lignes affectées ou 0 pour les instructions de DDL (Data Definition Language), comme `CREATE TABLE`.

Ensembles de résultats défilants et actualisables

Comme vous l'avez vu, la méthode `next` de la classe `ResultSet` parcourt les lignes d'un ensemble de résultats. Ceci convient certainement pour un programme devant analyser des données. Toutefois, envisagez un affichage de données visuelles qui présente un tableau ou des résultats de requêtes (voir Figure 3.7). Il est généralement souhaitable que l'utilisateur puisse avancer ou reculer dans l'ensemble des résultats. Or JDBC 1 n'avait pas de méthode `previous`. Les programmeurs qui souhaitaient implémenter la recherche vers l'arrière devaient placer manuellement le résultat en cache. La fonction de *défilement* de JDBC 2 vous permet d'avancer et de reculer dans un ensemble de résultats et de passer à n'importe quelle position dans celui-ci.

Figure 3.7

La vue d'une interface graphique utilisateur d'un résultat de requête.

F	TITLE	ISBN	PUBLISHER_ID	URL	PRICE
8	HTML 3 Interacti	1-57169066-	15716	w3.waite.com/w	39.99
9	HTML 3.0 Manua	1-56-276352	15627	www.mcp.com/z	24.95
10	HTML by Examl	0-78970812-	07897	www.mcp.com/z	34.99
11	HTML CD- An In	0-13-232331	01356	www.prenhall.co	40
12	HTML for Dumm	1-56-884647	15688	db.www.idgbook	30
13	HTML For Dumm	1-56884990-	15688	db.www.idgbook	12.99
14	HTML For Fun ar	0-13-242488	01356	www.sun.com/s	40
15	HTML for the Wo	0-02188448-	00218	www.peachpit.cc	17.95
16	HTML In Action	1-55615948-	15561	www.microsoft.c	29.95
17	HTML Plus!	0-53451626-	05345	scholar.lib.vt.ed	36
18	HTML Publishing	1-56-604288	15660	ephraim.vmedia	39.95
19	HTML Publishing	0-07-912100	00791		45
20	HTML Quick Ref	0-78970867-	07897	www.mcn.com/z	17.99

De plus, lorsque les utilisateurs voient le contenu d'un ensemble de résultats pour les utilisateurs, ceux-ci peuvent être tentés de le modifier. Ainsi, si vous leur proposez une vue modifiable, vous

178 Au cœur de Java 2 - Fonctions avancées

devrez vous assurer que les modifications sont renvoyées à la base de données. Dans JDBC 1, il fallait programmer des instructions UPDATE. Dans JDBC 2, vous pouvez simplement mettre à jour les entrées de l'ensemble de résultats et la base de données se met automatiquement à jour.

JDBC 2 permet également de mettre à jour un ensemble de résultats avec les données les plus récentes si elles ont été modifiées par une autre connexion simultanée à la base de données. JDBC 3 spécifie le comportement des ensembles de résultats lorsqu'une transaction est envisagée. Toutefois, ces fonctions n'entrent pas dans le cadre de ce chapitre d'introduction. Nous vous conseillons donc de consulter l'ouvrage (en langue anglaise) *JDBC API Tutorial and Reference*, ainsi que les documents de spécifications JDBC à l'adresse <http://java.sun.com/products/jdbc> pour obtenir de plus amples informations.

Ensembles de résultats défilants

Pour obtenir des ensembles de résultats pouvant défiler à partir de vos requêtes, vous devez obtenir un objet Statement différent à l'aide de la méthode :

```
Statement stat = conn.createStatement(type, concurrency);
```

Pour une instruction préparée, utilisez l'appel :

```
PreparedStatement stat = conn.prepareStatement(command,
    type, concurrency);
```

Les valeurs possibles de type et concurrency sont recensées dans les Tableaux 3.7 et 3.8. Vous disposez des choix suivants :

- Souhaitez-vous pouvoir faire défiler l'ensemble de résultats ? Si ce n'est pas le cas, utilisez `ResultSet.TYPE_FORWARD_ONLY`.
- S'il doit être possible de faire défiler l'ensemble de résultats, souhaitez-vous pouvoir refléter les changements dans la base de données résultant de la requête ? Dans cette discussion, nous prendrons pour hypothèse le paramètre `ResultSet.TYPE_SCROLL_INSENSITIVE` pour les résultats défilants. Ceci suppose que l'ensemble des résultats ne "capte" pas les changements de base de données survenus après l'exécution de la requête.
- Souhaitez-vous pouvoir mettre à jour la base de données en modifiant l'ensemble des résultats ? Voir la section suivante pour obtenir de plus amples détails.

Par exemple, si vous souhaitez simplement pouvoir faire défiler un ensemble de résultats, mais que vous ne souhaitiez pas modifier ses données, vous utiliserez :

```
Statement stat
    = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
```

Tableau 3.7 : Valeurs du type *ResultSet*

TYPE_FORWARD_ONLY	L'ensemble de résultats ne défile pas.
TYPE_SCROLL_INSENSITIVE	L'ensemble de résultats défile mais n'est pas sensible aux changements de la base de données.
TYPE_SCROLL_SENSITIVE	L'ensemble de résultats défile et il est sensible aux changements de la base de données.

Tableau 3.8 : Valeurs de simultanéité *ResultSet*

CONCUR_READ_ONLY	L'ensemble de résultats ne peut pas être utilisé pour mettre à jour la base de données.
CONCUR_UPDATABLE	L'ensemble de résultats peut être utilisé pour mettre à jour la base de données.

Tous les ensembles de résultats qui sont renvoyés par des appels à la méthode

```
ResultSet rs = stat.executeQuery(query)
```

peuvent maintenant défiler. Un ensemble de résultats défilant possède un *curseur* qui indique la position actuelle.

INFO

En fait, un pilote de base de données pourrait ne pas être capable d'honorer votre requête d'un curseur défilant ou actualisable. Les méthodes `supportResultSetType` et `supportResultSetConcurrency` de la classe `DatabaseMetaData` vous indiquent les types et les modes de concurrence supportés par une base de données particulière. Même si une base de données supporte tous les modes d'ensembles de résultats, une requête particulière pourrait ne pas pouvoir produire un ensemble de résultats avec toutes les propriétés demandées (par exemple, l'ensemble de résultats d'une requête complexe peut ne pas être actualisable. Dans ce cas, la méthode `executeQuery` renvoie un `ResultSet` de capacités moindres et ajoute un `SQLWarning` à l'objet de connexion. Vous pouvez récupérer des avertissements avec les méthodes `getType` et `getConcurrency` de la classe `ResultSet` pour découvrir le mode dont dispose un ensemble de résultats. Si vous ne vérifiez pas les capacités de l'ensemble de résultats et que vous émettiez une opération non prise en charge, comme `previous` sur un ensemble de résultats qui n'est pas défilant, l'opération déclenche une `SQLException`.

ATTENTION

Dans les pilotes JDBC 1, la classe `Connection` ne possède pas de méthode.

```
Statement createStatement(int type, int concurrency);
```

Si un programme compilé avec JDBC 2 charge par inadvertance un pilote JDBC 1 et appelle cette méthode non existante, le programme se bloque. Malheureusement, il n'existe pas de mécanisme dans JDBC 2 permettant de demander si un pilote est compatible JDBC 2.

Dans JDBC 3, vous pouvez utiliser les méthodes `getJDBCMinorVersion` et `getJDBCMinorVersion` de la classe `DatabaseMetaData` pour trouver le numéro de version JDBC du pilote.

Le défilement est très simple. Vous utilisez :

```
if (rs.previous()) . . .
```

pour défiler vers l'arrière. La méthode renvoie `true` si le curseur est positionné sur une ligne réelle et `false` s'il est positionné avant la première ligne.

Vous pouvez déplacer le curseur vers l'avant ou l'arrière de plusieurs lignes avec la commande

```
rs.relative(n);
```

180 *Au cœur de Java 2 - Fonctions avancées*

Si n est positif, le curseur avance, s'il est négatif, il recule. Si n est égal à zéro, l'appel reste sans effet. Si vous tentez de déplacer le curseur hors de l'ensemble de lignes actuel, il est défini pour pointer soit après la dernière ligne soit avant la première ligne, en fonction du signe de n . Alors, la méthode renvoie `false` et le curseur ne se déplace pas. La méthode renvoie `true` si le curseur s'est positionné sur une ligne réelle.

De même, vous pouvez régler le curseur sur un numéro de ligne particulier :

```
rs.absolute( $n$ );
```

Vous obtenez le numéro de ligne actuel avec l'appel

```
int currentRow = rs.getRow();
```

La première ligne de l'ensemble de résultats possède le numéro 1. Si la valeur renvoyée est 0, le curseur ne se trouve pas actuellement sur une ligne : il se trouve soit avant la première ligne, soit après la dernière ligne.

Il existe des méthodes commodes pour déplacer le curseur à la première position, à la dernière, avant la première ou après la dernière position :

```
first  
last  
beforeFirst  
afterLast
```

Les méthodes

```
isFirst  
isLast  
isBeforeFirst  
isAfterLast
```

testent si le curseur se trouve dans l'une de ces positions spéciales.

Utiliser un résultat défilant se révèle très simple. Le difficile travail de mise en mémoire cache des données de la requête est réalisé en coulisses par le pilote de la base de données.

Ensembles de résultats actualisables

Si vous souhaitez pouvoir modifier les données de l'ensemble de résultats et que les changements soient automatiquement reflétés dans la base de données, vous devez créer un ensemble de résultats actualisable. Ceux-ci n'ont pas forcément besoin de défiler, mais si vous présentez des données à un utilisateur pour modification, cette option est généralement souhaitable.

Pour obtenir des ensembles de résultats actualisables, vous allez créer une instruction comme celle-ci :

```
Statement stat  
= conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,  
    ResultSet.CONCUR_UPDATABLE);
```

Les ensembles de résultats renvoyés par un appel à `executeQuery` pourront ensuite être mis à jour.

INFO

Toutes les requêtes ne renvoient pas des ensembles de résultats actualisables. Si votre requête constitue un ensemble conjoint qui implique plusieurs tables, le résultat peut ne pas être actualisable. Si votre requête n'implique qu'une seule table ou si elle associe plusieurs tableaux par leurs clés primaires, l'ensemble de résultats risque fort d'être actualisable. Appelez la méthode `getConcurrency` de la classe `ResultSet` pour vous en assurer.

Par exemple, supposons que vous souhaitiez augmenter les prix de certains livres, mais que vous n'ayez pas de critère simple pour émettre une commande `UPDATE`. Vous pouvez alors parcourir tous les livres et mettre à jour les prix, en fonction de conditions arbitraires.

```
String query = "SELECT * FROM Livres";
ResultSet rs = stat.executeQuery(query);
while (rs.next())
{
    if ( . . . )
    {
        double increase = . . .
        double price = rs.getDouble("Prix");
        rs.updateDouble("Prix", price + increase);
        rs.updateRow();
    }
}
```

Il existe des méthodes de type `updateXxx` pour tous les genres de données qui correspondent au format SQL, comme `updateDouble`, `updateString`, etc. Comme avec les méthodes `getXxx`, vous spécifiez le nom ou le numéro de la colonne. Puis vous spécifiez la nouvelle valeur du champ.

INFO

Si vous utilisez la méthode `updateXxx` dont le premier paramètre correspond au numéro de la colonne, sachez qu'il s'agit du numéro de la colonne dans l'ensemble de résultats. Il peut être tout à fait différent du numéro de colonne dans la base de données.

La méthode `updateXxx` ne change que les valeurs de ligne, et non la base de données. Lorsque vous avez terminé avec les mises à jour de champs dans une ligne, vous devez appeler la méthode `updateRow`. Cette méthode envoie toutes les mises à jour dans la ligne en cours à la base de données. Si vous déplacez le curseur vers une autre ligne sans appeler `updateRow`, toutes les mises à jour sont effacées de l'ensemble de lignes et ne sont jamais communiquées à la base de données. Vous pouvez également appeler la méthode `cancelRowUpdates` pour annuler les mises à jour de la ligne en cours.

L'exemple qui précède montre comment modifier une ligne existante. Si vous souhaitez ajouter une nouvelle ligne à la base de données, vous utiliserez tout d'abord la méthode `moveToInsertRow` pour déplacer le curseur à une position spéciale, appelée *ligne d'insertion*. Une nouvelle ligne s'intègre à la position d'insertion de ligne grâce à des instructions de type `updateXxx`. Enfin, lorsque vous avez terminé, appelez la méthode `insertRow` pour passer la nouvelle ligne à la base de données. Une fois l'insertion terminée, appelez `moveToCurrentRow` pour ramener le curseur à sa position initiale. En voici un exemple :

```
rs.moveToInsertRow();
rs.updateString("Titre", title);
rs.updateString("ISBN", isbn);
```

182 *Au cœur de Java 2 - Fonctions avancées*

```
rs.updateString("Editeur_ID", pubid);
rs.updateDouble("Prix", price);
rs.insertRow();
rs.moveToCurrentRow();
```

Vous remarquerez que vous n'avez aucune influence sur *l'emplacement* où vous allez ajouter les nouvelles données dans l'ensemble de résultats ou dans la base de données.

Enfin, vous pouvez effacer la ligne située sous le curseur.

```
rs.deleteRow();
```

La méthode `deleteRow` supprime immédiatement la ligne de l'ensemble de résultats et de la base de données.

Les méthodes `updateRow`, `insertRow` et `deleteRow` de la classe `ResultSet` vous offrent les mêmes possibilités que l'exécution des commandes SQL `UPDATE`, `INSERT` et `DELETE`. Toutefois, les programmeurs habitués au langage de programmation Java trouveront plus naturel de manipuler le contenu de la base de données par le biais d'ensembles de résultats qu'en construisant des instructions SQL.

ATTENTION

Faute de précautions, vous risquez d'aboutir à un code qui ne fonctionnera pas, avec des ensembles de résultats actualisables. Sachez qu'il est *bien plus* efficace d'exécuter une instruction `UPDATE` que d'effectuer une requête et de parcourir le résultat, en modifiant les données tout au long du chemin. Les ensembles de résultats actualisables font sens dans des programmes interactifs dans lesquels un utilisateur peut effectuer des changements arbitraires ; toutefois, en ce qui concerne les changements plus axés sur la programmation, une commande `UPDATE SQL` conviendra mieux.

javax.sql.Connection 1.1

- `Statement createStatement(int type, int concurrency)` 1.2
 - `PreparedStatement prepareStatement(String command, int type, int concurrency)` 1.2
- Crée une instruction ou une instruction préparée qui produit des ensembles de résultats avec le type et la concurrence donnés.

Paramètres :

<code>command</code>	la commande à préparer
<code>type</code>	l'une des constantes <code>TYPE_FORWARD_ONLY</code> , <code>TYPE_SCROLL_INSENSITIVE</code> ou <code>TYPE_SCROLL_SENSITIVE</code> de l'interface <code>ResultSet</code>

`concurrency` l'une des constantes `CONCUR_READ_ONLY` ou `CONCUR_UPDATABLE` de l'interface `ResultSet`

- `SQLWarning getWarnings()`

Renvoie le premier des avertissements en attente sur cette connexion ou `null` si aucun avertissement n'est en attente. Les avertissements sont chaînés, ils continuent à appeler `getNextWarning` sur l'objet `SQLWarning` renvoyé jusqu'à ce que la méthode renvoie `null`. Cet appel ne consomme pas les avertissements. La classe `SQLWarning` prolonge la `SQLException`. Utilisez `getErrorCode` et `getSQLState` hérités pour analyser les avertissements.

- `void clearWarnings()`
Efface tous les avertissements qui ont été reportés sur cette connexion.

java.sql.ResultSet 1.1

- `int getType()` 1.2
Renvoie le type de cet ensemble de résultats, entre `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE` et `TYPE_SCROLL_SENSITIVE`.
- `int getConcurrency()` 1.2
Renvoie le paramètre de concurrence de cet ensemble de résultats, entre `CONCUR_READ_ONLY` et `CONCUR_UPDATABLE`.
- `boolean previous()` 1.2
Déplace le curseur sur la ligne précédente. Renvoie `true` si le curseur est positionné sur une ligne.
- `int getRow()` 1.2
Obtient le nombre de la ligne courante. Les lignes sont numérotées à partir de 1.
- `boolean absolute(int r)` 1.2
Déplace le curseur à la ligne `r`. Renvoie `true` si le curseur est positionné sur une ligne.
- `boolean relative(int d)` 1.2
Déplace le curseur par de `d` lignes. Si `d` est négatif, le curseur est déplacé vers l'arrière. Renvoie `true` si le curseur est positionné sur une ligne.
- `boolean first()` 1.2
- `boolean last()` 1.2
Déplace le curseur à la première ou dernière ligne. Renvoie `true` si le curseur est positionné sur une ligne.
- `void beforeFirst()` 1.2
- `void afterLast()` 1.2
Déplace le curseur avant la première ligne ou après la dernière ligne.
- `boolean isFirst()` 1.2
- `boolean isLast()` 1.2
Teste si le curseur se trouve sur la première ligne ou sur la dernière ligne.
- `boolean isBeforeFirst()` 1.2
- `boolean isAfterLast()` 1.2
Teste si le curseur se trouve avant la première ligne ou après la dernière ligne.
- `void moveToInsertRow()` 1.2
Déplace le curseur dans la *ligne d'insertion*. La ligne d'insertion est une ligne spéciale utilisée pour insérer de nouvelles données avec les méthodes `updateXxx` et `insertRow`.
- `void moveToCurrentRow()` 1.2
Ramène le curseur depuis la ligne d'insertion jusqu'à la ligne qu'il occupait lors de l'appel de la méthode `moveToInsertRow`.
- `void insertRow()` 1.2
Insère le contenu de la ligne d'insertion dans la base de données et dans l'ensemble de résultats.

184 Au cœur de Java 2 - Fonctions avancées

- `void deleteRow()` 1.2
Supprime la ligne en cours à partir de la base de données et l'ensemble de résultats.
- `void updateXxx(int column, Xxx data)` 1.2
- `void updateXxx(String columnName, Xxx data)` 1.2
(Xxx est un type comme `int`, `double`, `String`, `Date`, etc.)
Actualise un champ dans la ligne en cours de l'ensemble de résultats.
- `void updateRow()` 1.2
Envoie les mises à jour de la ligne en cours à la base de données.
- `void cancelRowUpdates()` 1.2
Annule les mises à jour de la ligne en cours.

***java.sql.DatabaseMetaData* 1.1**

- `boolean supportsResultSetType(int type)` 1.2
Renvoie `true` si la base de données peut supporter les ensembles de résultats du type donné.
Paramètres : `type` l'une des constantes `TYPE_FORWARD_ONLY`,
 `TYPE_SCROLL_INSENSITIVE` ou
 `TYPE_SCROLL_SENSITIVE` de l'interface `ResultSet`.
- `boolean supportsResultSetConcurrency(int type, int concurrency)` 1.2
Renvoie `true` si la base de données peut supporter les ensembles de résultats de la combinaison donnée du type et de la concurrence.
Paramètres : `type` l'une des constantes `TYPE_FORWARD_ONLY`,
 `TYPE_SCROLL_INSENSITIVE` ou
 `TYPE_SCROLL_SENSITIVE` de l'interface `ResultSet`.
 `concurrency` l'une des constantes `CONCUR_READ_ONLY`
 ou `CONCUR_UPDATABLE` de l'interface `ResultSet`.

Métadonnées

Dans les sections précédentes, nous avons vu comment remplir les tableaux d'une base de données, comment effectuer une requête et comment modifier une base de données. Cependant, JDBC peut vous fournir plus d'informations sur la *structure* d'une base de données et de ses tableaux. Par exemple, vous pouvez obtenir une liste des tableaux d'une base de données particulière ou le nom de chaque colonne et le type de données associées de chaque tableau. Ces informations ne sont pas essentielles pour implémenter une application professionnelle particulière avec une base de données prédéfinies. Après tout, si vous concevez vous-même les tableaux, vous connaissez déjà leur structure. En revanche, ces informations de structure sont extrêmement importantes pour les programmeurs qui doivent écrire des outils pour travailler sur n'importe quelle base de données.

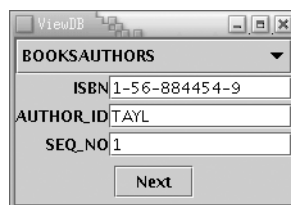
Dans cette section, nous allons vous expliquer comment écrire ce type d'outil. Il vous permettra de parcourir tous les tableaux d'une base de données.

Le menu déroulant en haut de l'application affiche tous les tableaux de la base de données. Sélectionnez l'un d'entre eux, et le cadre central sera rempli avec les noms des champs de ce tableau et

avec les valeurs du premier enregistrement, comme le montre la Figure 3.8. Cliquez sur "Suivant" pour parcourir les enregistrements du tableau.

Figure 3.8

L'application ViewDB.



De nombreuses bases de données sont livrées avec des outils bien plus sophistiqués pour afficher et modifier les tableaux. Si ce n'est pas le cas de la vôtre, consultez iSQL-Viewer (<http://isql.sourceforge.net>). Ces programmes sont capables d'afficher les tableaux dans n'importe quelle base de données JDBC. Notre programme d'exemple n'a pas pour but de remplacer ces outils, mais il montre comment implémenter un outil qui travaillera avec des tableaux arbitraires.

En SQL, les données qui décrivent la base de données ou l'une de ses parties sont appelées *métadonnées* (pour les distinguer des données elles-mêmes qui sont stockées dans la base de données). Vous pourrez rencontrer trois types de métadonnées : les métadonnées relatives à la base de données, celles concernant un ensemble de résultats et celles liées aux paramètres des instructions préparées.

Pour obtenir plus d'informations sur une base de données, il faut demander un objet de type `DatabaseMetaData` à la connexion de la base de données.

```
DatabaseMetaData meta = conn.getMetaData();
```

Vous êtes maintenant prêt à obtenir certaines métadonnées. Par exemple, l'appel

```
ResultSet mrs = meta.getTables(null, null, null, new String[]
{ "TABLE" });
```

renvoie un ensemble de résultats contenant des informations sur tous les tableaux d'une base de données. Consultez la note d'API pour connaître les paramètres de cette méthode.

Chaque ligne de l'ensemble de résultats contient des informations sur un tableau de la base de données. Seule la troisième colonne nous intéresse (reportez-vous encore à la note d'API pour la signification des autres colonnes). Les noms de tableaux se trouvent donc dans `rs.getString(3)`. Voici le code permettant de remplir le menu déroulant :

```
while (mrs.next())
    tableNames.addItem(mrs.getString(3));
rs.close();
```

La classe `DatabaseMetaData` propose des informations sur la base de données. Une deuxième classe de métadonnées, `ResultSetMetaData`, rapporte des informations sur un ensemble de résultats. Lorsque vous obtenez un ensemble de résultats à partir d'une requête, vous pouvez demander le nombre de colonnes total, ainsi que le nom, le type et le nombre d'éléments de chaque colonne.

Nous nous servons de ces informations pour définir une étiquette pour chaque nom de colonne et un champ de texte suffisamment grand pour chaque valeur.

```
ResultSet mrs = stat.executeQuery("SELECT * FROM " + tableName);
ResultSetMetaData meta = mrs.getMetaData();
```


186 Au cœur de Java 2 - Fonctions avancées

```
for (int i = 1; i <= meta.getColumnCount(); i++)
{
    String columnName = meta.getColumnLabel(i);
    int columnWidth = meta.getColumnDisplaySize(i);
    JLabel l = new JLabel (columnName);
    JTextField tf = new TextField (columnWidth);
    . . .
}
```

Il existe une deuxième utilisation importante pour les métadonnées de la base de données. Les bases de données sont complexes et la norme SQL autorise de nombreuses variations. Plus d'une centaine de méthodes dans la classe DatabaseMetaData peuvent enquêter sur la base de données, y compris des appels avec des noms exotiques tels que

```
meta.supportsCatalogsInPrivilegeDefinitions()
```

et

```
meta.nullPlusNonNullIsNull()
```

A l'évidence, elles sont destinées aux utilisateurs avancés possédant des besoins particuliers, notamment ceux qui doivent écrire des codes fortement transposables qui fonctionnent avec plusieurs bases de données. Dans notre programme, nous ne donnons qu'un seul exemple de cette technique. Nous demandons aux métadonnées de la base de données si le pilote JDBC supporte des ensembles de résultats défilants. Si c'est le cas, nous ouvrons un ensemble de résultats défilant et ajoutons un bouton "Précédente" pour défiler vers l'arrière.

```
if (meta.supportsResultSetType(
    ResultSet.TYPE_SCROLL_INSENSITIVE)) . . .
```

Les étapes suivantes décrivent brièvement le programme.

1. Ajout du menu déroulant contenant le nom du tableau, de l'écran contenant les valeurs du tableau et du panneau de boutons.
2. Connexion à la base de données. Découvre si elle supporte les ensembles de résultats défilants. Dans ce cas, crée l'objet Statement pour produire des ensembles de résultats défilants, sinon crée simplement une instruction par défaut.
3. Obtient les noms du tableau et les place dans le composant choisi.
4. Si le défilement est supporté, ajoute le bouton "Précédente". Ajoute toujours le bouton "Suivante".
5. Lorsque l'utilisateur sélectionne un tableau, le programme effectue une requête pour visualiser toutes ses valeurs, puis il lit la métadonnée. Les anciens composants du centre sont supprimés et remplacés par un quadrillage d'étiquettes et de boîtes de texte. Ces boîtes de texte sont enregistrées dans un vecteur. La méthode validate est ensuite appelée pour recalculer la fenêtre. Puis, un appel à showNextRow permet d'afficher la première ligne.
6. Appel de la méthode showNextRow pour afficher la première ligne, et à chaque fois que l'utilisateur clique sur le bouton "Suivante". La méthode showNextRow lit la ligne suivante dans le tableau et place les valeurs de ses colonnes dans les boîtes de texte.
7. Il existe une légère subtilité dans la détection de la fin de l'ensemble de résultats. Lorsque cet ensemble défile, nous pouvons simplement utiliser la méthode isLast. Mais lorsqu'il ne défile pas, l'appel à cette méthode entraînera une exception (ou même une erreur JVM si le pilote est

un pilote JDBC 1). Ainsi, nous utilisons une stratégie différente pour les ensembles de résultats qui ne défilent pas. Lorsque `rs.next()` renvoie `false`, nous fermons l'ensemble des résultats et définissons `rs` sur `null`.

8. Le bouton "Précédente" appelle `showPreviousRow`, qui déplace l'ensemble de résultats vers l'arrière. Etant donné que ce bouton n'est installé que si l'ensemble de résultats peut défiler, nous savons que les méthodes `previous` et `isFirst` sont supportées.
9. La méthode `showRow` remplit simplement les champs de l'ensemble de résultats dans les champs de texte du panneau de données.

L'Exemple 3.4 donne le code de ce programme.

Exemple 3.4 : *ViewDB.java*

```
import java.net.*;
import java.sql.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;

/**
 * Ce programme utilise des métadonnées pour afficher des tableaux
 * arbitraires dans une base de données.
 */
public class ViewDB
{
    public static void main(String[] args)
    {
        JFrame frame = new ViewDBFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.set(Visible);
    }
}

/**
 * Le cadre qui contient le panneau de données et les boutons de
 * navigation.
 */
class ViewDBFrame extends JFrame
{
    public ViewDBFrame()
    {
        setTitre("ViewDB");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        tableNames = new JComboBox();
        tableNames.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    showTable((String) tableNames.getSelectedItem());
                }
            });
    }
}
```

188 *Au cœur de Java 2 - Fonctions avancées*

```

add(tableNames, BorderLayout.NORTH);

try
{
    conn = getConnection();
    meta = conn.getMetaData();
    createStatement();
    getTableNames();
}
catch(SQLException e)
{
    JOptionPane.showMessageDialog(this, e);
}
catch (IOException e)
{
    JOptionPane.showMessageDialog(this, e);
}

JPanel buttonPanel = new JPanel();
add(buttonPanel, BorderLayout.SOUTH);

if (scrolling)
{
    previousButton = new JButton("Précédente");
    previousButton.addActionListener(new
        ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                showPreviousRow();
            }
        });
    buttonPanel.add(previousButton);
}

nextButton = new JButton("Suivante");
nextButton.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            showNextRow();
        }
    });
buttonPanel.add(nextButton);

addWindowListener(new
    WindowAdapter()
    {
        public void windowClosing(WindowEvent event)
        {
            try
            {
                if (conn != null) conn.close();
            }
            catch(SQLException e)
            {
                while (e != null)
                {

```

```

        e.printStackTrace();
        e = e.getNextException();
    }
}

});
}

/**
 * Crée l'objet instruction utilisé pour exécuter des requêtes.
 * Si la base de données supporte les curseurs de défilement,
 * l'instruction est créée pour les produire.
 */
public void createStatement() throws SQLException
{
    if (meta.supportsResultSetType(
        ResultSet.TYPE_SCROLL_INSENSITIVE))
    {
        stat = conn.createStatement(
            ResultSet.TYPE_SCROLL_INSENSITIVE,
            ResultSet.CONCUR_READ_ONLY);
        scrolling = true;
    }
    else
    {
        stat = conn.createStatement();
        scrolling = false;
    }
}

/**
 * Obtient tous les noms de tableaux de cette base de données et les
 * ajoute au menu déroulant.
 */
public void getTableNames() throws SQLException
{
    ResultSet mrs = meta.getTables(null, null, null,
        new String[] { "TABLE" });
    while (mrs.next())
        tableNames.addItem(mrs.getString(3));
    mrs.close();
}

/**
 * Prépare les champs de texte pour montrer un nouveau tableau, et
 * affiche la première ligne.
 * @param tableName le nom du tableau à afficher
 */
public void showTable(String tableName)
{
    try
    {
        if (rs != null) rs.close();
        rs = stat.executeQuery("SELECT * FROM " + tableName);
        if (scrollPane != null)
            remove(scrollPane);
        dataPanel = new DataPanel(rs);
        scrollPane = new JScrollPane(dataPanel);
        add(scrollPane, BorderLayout.CENTER);
    }
}

```

190 *Au cœur de Java 2 - Fonctions avancées*

```

        validate();
        showNextRow();
    }
    catch (SQLException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }
}

/**
 * Passe à la ligne de tableau précédente.
 */
public void showPreviousRow()
{
    try
    {
        if (rs == null || rs.isFirst()) return;
        rs.previous();
        dataPanel.showRow(rs);
    }
    catch (SQLException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }
}

/**
 * Passe à la prochaine ligne de tableau.
 */
public void showNextRow()
{
    try
    {
        if (rs == null || scrolling && rs.isLast()) return;

        if (!rs.next() && !scrolling)
        {
            rs.close();
            rs = null;
            return;
        }

        dataPanel.showRow(rs);
    }
    catch (SQLException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }
}

/**
 * Obtient une connexion à partir des propriétés spécifiées
 * dans un fichier database.properties
 * @return la connexion à la base de données
 */
public static Connection getConnection()
    throws SQLException, IOException

```

```

{
    Properties props = new Properties();
    FileInputStream in
        = new FileInputStream("database.properties");
    props.load(in);
    in.close();

    String drivers = props.getProperty("jdbc.drivers");
    if (drivers != null)
        System.setProperty("jdbc.drivers", drivers);
    String url = props.getProperty("jdbc.url");
    String username = props.getProperty("jdbc.username");
    String password = props.getProperty("jdbc.password");

    return
        DriverManager.getConnection(url, username, password);
}

public static final int DEFAULT_WIDTH = 300;
public static final int DEFAULT_HEIGHT = 200;

private JButton previousButton;
private JButton nextButton;
private DataPanel dataPanel;
private Component scrollPane;
private JComboBox tableNames;

private Connection conn;
private Statement stat;
private DatabaseMetaData meta;
private ResultSet rs;
private boolean scrolling;
}

/**
 * Ce panneau affiche le contenu d'un ensemble de résultats.
 */
class DataPanel extends JPanel
{
    /**
     * Construit le panneau de données.
     * @param rs l'ensemble de résultats affiché par ce panneau
     */
    public DataPanel(ResultSet rs) throws SQLException
    {
        fields = new ArrayList<JTextField>();
        setLayout(new GridBagLayout());
        GridBagConstraints gbc = new GridBagConstraints();
        gbc.gridwidth = 1;
        gbc.gridheight = 1;

        ResultSetMetaData rsmd = rs.getMetaData();
        for (int i = 1; i <= rsmd.getColumnCount(); i++)
        {
            gbc.gridy = i - 1;

            String columnName = rsmd.getColumnLabel(i);
            gbc.gridx = 0;

```

192 Au cœur de Java 2 - Fonctions avancées

```

        gbc.anchor = GridBagConstraints.EAST;
        add(new JLabel(columnName), gbc);

        int columnWidth = rsmd.getColumnDisplaySize(i);
        JTextField tb = new JTextField(columnWidth);
        fields.add(tb);

        gbc.gridx = 1;
        gbc.anchor = GridBagConstraints.WEST;
        add(tb, gbc);
    }
}

/**
 * Affiche une ligne de base de données en remplissant tous les champs
 * texte avec les valeurs de colonnes.
 */
public void showRow(ResultSet rs) throws SQLException
{
    for (int i = 1; i <= fields.size(); i++)
    {
        String field = rs.getString(i);
        JTextField tb
            = (JTextField)fields.get(i - 1);
        tb.setText(field);
    }
}

private ArrayList<JTextFields> fields;
}

```

java.sql.Connection 1.1

- DatabaseMetaData getMetaData()

Renvoie une métadonnée de la connexion sous la forme d'un objet DataBaseMetaData.

java.sql.DatabaseMetaData

- ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String types[])

Récupère la description de tous les tableaux d'un catalogue qui correspondent à un modèle de nom de tableau et de schéma, ainsi qu'à des critères de type. Un *schéma* décrit un groupe de tableaux associés et des droits d'accès. Un *catalogue* décrit un groupe de schémas associés. Ces concepts sont très importants pour structurer les grandes bases de données.

Les paramètres catalog et schema peuvent être vides ("") pour retrouver des tableaux sans catalogue ou sans schéma, ou null pour renvoyer des tableaux indépendamment d'un catalogue ou d'un schéma.

Le tableau types contient les noms des types de tableaux à inclure. Les types les plus courants sont TABLE, VIEW, SYSTEM TABLE, GLOBAL TEMPORARY, LOCAL TEMPORARY, ALIAS, et SYNONYM. Si types vaut null, les tableaux de n'importe quel type peuvent être renvoyés.

L'ensemble de résultats possède cinq colonnes, qui sont toutes du type `String`, comme le montre le Tableau 3.9.

Tableau 3.9 : Les cinq colonnes de l'ensemble de résultats

1	TABLE_CAT	Catalogue de tableau (peut être null)
2	TABLE_SCHEM	Schéma de tableau (peut être null)
3	TABLE_NAME	Nom de tableau
4	TABLE_TYPE	Type de tableau
5	REMARKS	Commentaires sur le tableau

- `int getJDBCMinorVersion()`
 - `int getJDBCMajorVersion()`
- (JDBC 3) Renvoient les numéros de version JDBC principaux et secondaires pour le pilote qui a établi la connexion à la base de données. Par exemple, un pilote JDBC 3.0 possède le numéro de version principal 3 et le numéro de version secondaire 0.
- `int getMaxConnections()`
- Renvoie le nombre maximal de connexions simultanées à cette base de données.
- `int getMaxStatements()`
- Renvoie le nombre optimal d'instructions ouvertes simultanément par connexion à la base de données ou 0 si le nombre est illimité ou inconnu.

`java.sql.ResultSet 1.1`

- `ResultSetMetaData getMetaData()`
- Fournit la métadonnée associée avec les colonnes courantes `ResultSet`.

`java.sql.ResultSetMetaData 1.1`

- `int getColumnCount()`
- Renvoie le nombre de colonnes de l'objet courant `ResultSet`.
- `int getColumnDisplaySize(int column)`
- Indique la largeur maximale de colonne spécifiée par le paramètre d'indice.
- Paramètres :* `column` le numéro de colonne
- `String getColumnLabel(int column)`
- Fournit le titre suggéré de la colonne.
- Paramètres :* `column` le numéro de colonne
- `String getColumnName(int column)`
- Renvoie le nom de la colonne associée à l'indice de colonne spécifié.
- Paramètres :* `column` le numéro de colonne

RowSet

Les ensembles de résultats défilants constituent un outil performant, mais ils présentent un inconvénient majeur. Vous devez maintenir ouverte la connexion à la base de données pendant l'interaction avec l'utilisateur. Les utilisateurs peuvent toutefois quitter leur ordinateur pendant un long moment, en laissant la connexion occupée. Ceci doit être évité, les connexions à la base de données représentant des ressources rares. Dans cette situation, vous utiliserez un *RowSet*. L'interface *RowSet* étend l'interface *ResultSet*, mais ces *RowSet* n'ont pas à être liés à une connexion de base de données.

Les *RowSet* sont également disponibles lorsque vous devez déplacer un résultat de requête à un autre niveau d'une application complexe ou vers un autre appareil, tel un téléphone portable. Il ne faut pas, en revanche, vouloir déplacer un jeu de résultats : ses structures de données peuvent être lourdes, et il est lié à la connexion de la base de données.

Le paquetage `javax.sql.rowset` propose les interfaces suivantes, qui étendent l'interface *RowSet* :

- Un *CachedRowSet* autorise un fonctionnement hors connexion. Nous traiterons de cette interface dans la section suivante.
- Un *WebRowSet* est un *CachedRowSet* pouvant être enregistré dans un fichier XML. Celui-ci peut ensuite être déplacé à un autre niveau d'une application Web, où il est ouvert par un autre objet *WebRowSet*.
- Les interfaces *FilteredRowSet* et *JoinRowSet* acceptent des opérations légères sur les *RowSet*, équivalant aux opérations SQL `SELECT` et `JOIN`. Ces opérations sont réalisées sur les données stockées dans des *RowSet*, sans qu'il soit besoin de se connecter à une base de données.
- Un *JdbcRowSet* est un emballage fin qui entoure un *ResultSet*. Il ajoute des éléments de récupération et de définition utiles depuis l'interface *RowSet*, ce qui transforme un jeu de résultats en "bean" (voir Chapitre 8 pour en savoir plus sur les beans).

Sun Microsystems espère que les fabricants de bases de données produiront des implémentations efficaces de ces interfaces. Heureusement, ils fournissent aussi des implémentations de référence, de sorte que vous puissiez utiliser les *RowSet* même si votre fournisseur de base de données ne les prend pas en charge. Les implémentations de référence font partie du JDK 5.0, mais vous pouvez les télécharger depuis <http://java.sun.com/jdbc>. Les implémentations de référence se trouvent dans le paquetage `com.sun.rowset`. Les noms de classe se terminent par `Impl`, comme *CachedRowSetImpl*.

CachedRowSet

Un *CachedRowSet* contient toutes les données d'un ensemble de résultats. Puisqu'il étend l'interface *ResultSet*, vous l'utilisez exactement comme un ensemble de résultats. Les *CachedRowSet* présentent un gros avantage : vous pouvez fermer la connexion et continuer à utiliser le *RowSet*. Comme vous le verrez dans notre exemple, ceci simplifie grandement l'implémentation des applications interactives. Chaque commande utilisateur se contente d'ouvrir la connexion à la base de données, émet une requête, place le résultat dans un *RowSet*, puis met fin à la connexion.

Vous pouvez même modifier les données d'un *CachedRowSet*. Bien entendu, les modifications ne sont pas immédiatement appliquées à la base de données : vous devez réaliser une requête explicite

visant à accepter les changements accumulés. Le `CachedRowSet` se reconnecte alors à la base de données et émet des commandes SQL pour appliquer ces changements.

Bien entendu, les `CachedRowSet` ne conviennent pas aux grands résultats de requêtes. Le déplacement de grandes quantités d'enregistrements de la base de données en mémoire serait tout à fait inefficace, en particulier si les utilisateurs n'en consultent que quelques-uns.

Vous pouvez remplir un `CachedRowSet` à partir d'un ensemble de résultats :

```
ResultSet result = stat.executeQuery(queryString);
CachedRowSet rowset = new com.sun.rowset.CachedRowSetImpl();
// ou utiliser une implémentation de votre fournisseur de bdd
rowset.populate(result);
conn.close(); // ok pour fermer la connexion à la bdd
```

Vous pouvez aussi autoriser l'objet `CachedRowSet` à établir automatiquement la connexion. Configurez les paramètres de la base de données :

```
rowset.setURL("jdbc:mckoi://localhost/");
rowset.setUsername("dbuser");
rowset.setPassword("secret");
```

Définissez ensuite la commande de requête :

```
rowset.setCommand("SELECT * FROM Livres");
```

Enfin, remplit le `RowSet` du résultat de la requête :

```
rowset.execute();
```

Cet appel établit la connexion à la base de données, émet la requête, remplit le `RowSet` puis procède à la déconnexion.

L'inspection et la modification du `RowSet` se font à l'aide des mêmes commandes que celles utilisées pour les ensembles de résultats. Si vous avez modifié son contenu, vous devez le réécrire dans la base de données en appelant

```
rowset.acceptChanges(conn);
```

ou

```
rowset.acceptChanges();
```

Le deuxième appel ne fonctionnera que si vous avez configuré le `RowSet` avec les informations exigées pour la connexion à la base de données (URL, nom d'utilisateur, mot de passe...).

Vous avez vu précédemment que les ensembles de résultats ne sont pas tous actualisables. De même, un `RowSet` qui contient le résultat d'une requête complexe ne pourra pas réécrire les changements dans la base de données. Cela ne fonctionnera que si le `RowSet` contient des données d'un seul tableau.

ATTENTION

Si vous avez rempli le `RowSet` à partir d'un ensemble de résultats, il ne connaîtra pas le nom du tableau à actualiser. Appelez `setTable` pour définir le nom du tableau.

Une modification de la base de données ultérieure au remplissage du `RowSet` pose aussi problème et pourrait générer des données incohérentes. L'implémentation de référence vérifie si les valeurs

196 *Au cœur de Java 2 - Fonctions avancées*

initiales du `RowSet` (avant la modification) sont identiques aux valeurs de la base de données. Si c'est le cas, elles sont remplacées par les valeurs modifiées. Dans le cas contraire, une `SyncProviderException` est déclenchée et les changements ne sont pas appliqués. D'autres implémentations peuvent faire appel à d'autres stratégies de synchronisation.

Le programme de l'Exemple 3.5 est identique à la visionneuse de base de données de l'Exemple 3.4, mais nous utilisons maintenant un `CachedRowSet`. La logique du programme en est considérablement simplifiée.

- Il suffit d'ouvrir et de fermer la connexion dans chaque écouteur d'action.
- Il n'est plus nécessaire de piéger l'événement de "fermeture de fenêtre" pour mettre fin à la connexion.
- Nous ne nous préoccupons plus de savoir si l'ensemble de résultats est défilant. Les `RowSet` le sont toujours.

Exemple 3.5 : `RowSetTest.java`

```
import com.sun.rowset.*;
import java.net.*;
import java.sql.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.sql.*;
import javax.sql.rowset.*;

/**
 * Ce programme utilise des métadonnées pour afficher des tableaux
 * arbitraires dans une base de données.
 */
public class RowSetTest
{
    public static void main(String[] args)
    {
        JFrame frame = new RowSetFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Le cadre contenant le panneau de données et les
 * boutons de navigation.
 */
class RowSetFrame extends JFrame
{
    public RowSetFrame()
    {
        setTitle("RowSetTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        tableNames = new JComboBox();
        tableNames.addActionListener(new
            ActionListener()
```

```

        {
            public void actionPerformed(ActionEvent event)
            {
                showTable((String) tableNames.getSelectedItem());
            }
        });
add(tableNames, BorderLayout.NORTH);

try
{
    Connection conn = getConnection();
    try
    {
        DatabaseMetaData meta = conn.getMetaData();
        ResultSet mrs = meta.getTables(null, null, null, new String[]
            { "TABLE" });
        while (mrs.next())
            tableNames.addItem(mrs.getString(3));
    }
    finally
    {
        conn.close();
    }
}
catch (SQLException e)
{
    JOptionPane.showMessageDialog(this, e);
}
catch (IOException e)
{
    JOptionPane.showMessageDialog(this, e);
}

JPanel buttonPanel = new JPanel();
add(buttonPanel, BorderLayout.SOUTH);

previousButton = new JButton("Précédent");
previousButton.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            showPreviousRow();
        }
    });
buttonPanel.add(previousButton);

nextButton = new JButton("Suivant");
nextButton.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            showNextRow();
        }
    });
buttonPanel.add(nextButton);

deleteButton = new JButton("Supprimer");
deleteButton.addActionListener(new

```

198 *Au cœur de Java 2 - Fonctions avancées*

```

        ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                deleteRow();
            }
        });
        buttonPanel.add(deleteButton);

        saveButton = new JButton("Enregistrer");
        saveButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    saveChanges();
                }
            });
        buttonPanel.add(saveButton);
    }

    /**
     * Prépare les champs de texte pour afficher un nouveau tableau,
     * affiche la première ligne.
     * @param tableName Le nom du tableau à afficher
     */
    public void showTable(String tableName)
    {
        try
        {
            // ouvrir la connexion
            Connection conn = getConnection();
            try
            {
                // récupérer l'ensemble de résultats
                Statement stat = conn.createStatement();
                ResultSet result = stat.executeQuery("SELECT * FROM " + *
                    tableName);
                // copier dans le RowSet
                rs = new CachedRowSetImpl();
                rs.setTableName(tableName);
                rs.populate(result);
            }
            finally
            {
                conn.close();
            }

            if (scrollPane != null)
                remove(scrollPane);
            dataPanel = new DataPanel(rs);
            scrollPane = new JScrollPane(dataPanel);
            add(scrollPane, BorderLayout.CENTER);
            validate();
            showNextRow();
        }
        catch (SQLException e)
        {
            JOptionPane.showMessageDialog(this, e);
        }
    }

```

```

    }
    catch (IOException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }
}

/**
 * Déplace vers la ligne précédente du tableau.
 */
public void showPreviousRow()
{
    try
    {
        if (rs == null || rs.isFirst()) return;
        rs.previous();
        dataPanel.showRow(rs);
    }
    catch (SQLException e)
    {
        System.out.println("Erreur " + e);
    }
}

/**
 * Déplace vers la ligne suivante du tableau.
 */
public void showNextRow()
{
    try
    {
        if (rs == null || rs.isLast()) return;
        rs.next();
        dataPanel.showRow(rs);
    }
    catch (SQLException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }
}

/**
 * Efface la ligne courante du tableau.
 */
public void deleteRow()
{
    try
    {
        rs.deleteRow();
        if (!rs.isLast()) rs.next();
        else if (!rs.isFirst()) rs.previous();
        else rs = null;
        dataPanel.showRow(rs);
    }
    catch (SQLException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }
}

```

200 *Au cœur de Java 2 - Fonctions avancées*

```

/**
 * Enregistre toutes les modifications.
 */
public void saveChanges()
{
    try
    {
        Connection conn = getConnection();
        try
        {
            rs.acceptChanges(conn);
        }
        finally
        {
            conn.close();
        }
    }
    catch (SQLException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }
    catch (IOException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }
}

/**
 * Récupère une connexion à partir des propriétés spécifiées
 * dans le fichier database.properties
 * @return La connexion à la base de données
 */
public static Connection getConnection()
    throws SQLException, IOException
{
    Properties props = new Properties();
    FileInputStream in
        = new FileInputStream("database.properties");
    props.load(in);
    in.close();

    String drivers = props.getProperty("jdbc.drivers");
    if (drivers != null) System.setProperty("jdbc.drivers", drivers);
    String url = props.getProperty("jdbc.url");
    String username = props.getProperty("jdbc.username");
    String password = props.getProperty("jdbc.password");

    return DriverManager.getConnection(url, username, password);
}

public static final int DEFAULT_WIDTH = 400;
public static final int DEFAULT_HEIGHT = 200;

private JButton previousButton;
private JButton nextButton;
private JButton deleteButton;
private JButton saveButton;
private DataPanel dataPanel;

```

```

        private Component scrollPane;
        private JComboBox tableNames;

        private CachedRowSet rs;
    }

    /**
     * Ce panneau affiche le contenu d'un ensemble de résultats.
     */
    class DataPanel extends JPanel
    {
        /**
         * Construit le panneau de données.
         * @param rs L'ensemble de résultats dont le contenu est affiché
         */
        public DataPanel(RowSet rs) throws SQLException
        {
            fields = new ArrayList<JTextField>();
            setLayout(new GridBagLayout());
            GridBagConstraints gbc = new GridBagConstraints();
            gbc.gridwidth = 1;
            gbc.gridheight = 1;

            ResultSetMetaData rsmd = rs.getMetaData();
            for (int i = 1; i <= rsmd.getColumnCount(); i++)
            {
                gbc.gridy = i - 1;

                String columnName = rsmd.getColumnLabel(i);
                gbc.gridx = 0;
                gbc.anchor = GridBagConstraints.EAST;
                add(new JLabel(columnName), gbc);

                int columnWidth = rsmd.getColumnDisplaySize(i);
                JTextField tb = new JTextField(columnWidth);
                fields.add(tb);

                gbc.gridx = 1;
                gbc.anchor = GridBagConstraints.WEST;
                add(tb, gbc);
            }
        }

        /**
         * Affiche une ligne de base de données en remplissant tous les
         * champs de texte avec les valeurs des colonnes.
         */
        public void showRow(ResultSet rs) throws SQLException
        {
            for (int i = 1; i <= fields.size(); i++)
            {
                String field = rs.getString(i);
                JTextField tb = (JTextField) fields.get(i - 1);
                tb.setText(field);
            }
        }

        private ArrayList<JTextField> fields;
    }

```


202 *Au cœur de Java 2 - Fonctions avancées*

javax.sql.RowSet 1.4

- `String getURL()`
- `void setURL(String url)`
Récupèrent ou définissent l'URL de la base de données.
- `String getUsername()`
- `void setUsername(String username)`
Récupèrent ou définissent le nom d'utilisateur pour la connexion à la base de données.
- `String getPassword()`
- `void setPassword(String password)`
Récupèrent ou définissent le mot de passe pour la connexion à la base de données.
- `String getCommand()`
- `void setCommand(String command)`
Récupèrent ou définissent la commande exécutée pour remplir ce `RowSet`.
- `void execute()`
Remplit ce `RowSet` en émettant le jeu de commandes avec `setCommand`. Pour que le gestionnaire de pilotes obtienne la connexion, il convient de définir l'URL, le nom d'utilisateur et le mot de passe.

javax.sql.rowset.CachedRowSet 5.0

- `void execute(Connection conn)`
Remplit ce `RowSet` en émettant le jeu de commandes avec `setCommand`. Cette méthode utilise la connexion donnée et y *met fin*.
- `void populate(ResultSet result)`
Remplit ce `CachedRowSet` avec les données de l'ensemble de résultats.
- `String getTableName()`
- `void setTableName(String tableName)`
Récupèrent ou définissent le nom du tableau à partir duquel ce `RowSet` a été rempli.
- `void acceptChanges()`
- `void acceptChanges(Connection conn)`
Se reconnectent à la base de données et écrivent les changements résultant de la modification du `RowSet`. Peuvent déclencher une `SyncProviderException` s'il est impossible de réécrire les données suite aux changements de la base de données.

Transactions

Vous pouvez grouper un ensemble d'instructions pour former une *transaction* qui peut être effectuée lorsque tout s'est bien passé ou annulée si aucune des commandes n'a pu être menée à bien, dans le cas où une erreur apparaît.

La raison principale pour grouper des commandes dans des transactions est l'*intégrité de la base de données*. Par exemple, supposons que vous vouliez transférer de l'argent d'un compte bancaire à un autre. Il est alors important de débiter un compte et de créditer l'autre simultanément. Si le système bloque avant d'avoir crédité l'autre compte, le débit doit être annulé.

Si vous placez une mise à jour dans une transaction, cette dernière pourra réussir dans son intégralité, auquel cas elle pourra être validée. Si vous demandez une annulation (*rollback*), la base de données supprimera toutes les modifications apportées depuis la dernière validation de transaction. De plus, les requêtes ne sont validées que lorsque la base de données est validée.

Par défaut, une connexion de base de données est en mode *validation automatique*, et chaque commande SQL est validée dès qu'elle est exécutée. Une fois qu'une commande est validée, vous ne pouvez plus l'annuler.

Pour vérifier le mode de validation automatique courant, appelez la méthode `getAutoCommit` de la classe `Connection`.

Le mode de validation automatique peut être désactivé avec la commande suivante :

```
conn.setAutoCommit(false);
```

Vous pouvez maintenant créer un objet `Statement` de manière classique :

```
Statement stat = conn.createStatement();
```

Appelez `executeUpdate` autant de fois que vous le souhaitez :

```
stat.executeUpdate(commande1);
stat.executeUpdate(commande2);
stat.executeUpdate(commande3);
. . .
```

Lorsque toutes les commandes ont été exécutées, appelez la méthode `commit` :

```
conn.commit();
```

Mais si une erreur survient, appelez

```
conn.rollback();
```

A ce moment, toutes les commandes depuis la dernière validation sont automatiquement annulées. Ce mécanisme est typiquement appelé lorsqu'une `SQLException` a interrompu votre transaction.

Points de sauvegarde

Vous obtiendrez un meilleur contrôle sur le processus d'annulation en utilisant des *points de sauvegarde*. Ils marquent un emplacement auquel vous pourrez revenir par la suite, sans avoir à retourner au début de la transaction. Par exemple :

```
Statement stat = conn.createStatement(); // début de la transaction;
                                         // rollback() vient ici
stat.executeUpdate(command1);
Savepoint svpt = conn.setSavepoint(); // établir le point de sauvegarde;
                                         // rollback(svpt) vient ici
stat.executeUpdate(command2);
if (. . .) conn.rollback(svpt); // annuler l'effet de command2
. . .
conn.commit();
```

Nous faisons appel ici à un point de sauvegarde anonyme. Mais vous pouvez aussi lui donner un nom, comme

```
Savepoint svpt = conn.setSavepoint("étape1");
```

Lorsque vous avez fini de travailler avec un point de sauvegarde, libérez-le :

```
stat.releaseSavepoint(svpt);
```

204 Au cœur de Java 2 - Fonctions avancées

Mises à jour automatisées

Imaginons qu'un programme doive exécuter plusieurs instructions INSERT, en vue de remplir un tableau de base de données. Avec JDBC 2, vous pouvez améliorer les performances de ce programme en ayant recours à une *mise à jour automatisée*. Grâce à cette technique, un ensemble de commandes est rassemblé puis validé comme une seule commande.

INFO

Utilisez la méthode `supportsBatchUpdates` de la classe `DatabaseMetaData` pour savoir si votre base de données prend cette fonction en charge.

Les commandes que vous pouvez automatiser peuvent être des actions comme INSERT, UPDATE et DELETE, ou des commandes de définition de données comme CREATE TABLE et DROP TABLE. Cependant, il est impossible d'ajouter des commandes SELECT dans un traitement automatisé, puisque l'exécution d'une instruction SELECT renvoie un ensemble de résultats.

Pour exécuter un traitement automatisé, il faut commencer par créer un objet `Statement` :

```
Statement stat = conn.createStatement();
```

Puis, au lieu d'appeler `executeUpdate`, il faut appeler la méthode `addBatch` :

```
String command = "CREATE TABLE . . .";
stat.addBatch(command);

while (. . .)
{
    command = "INSERT INTO . . . VALUES (" + . . . + ")";
    stat.addBatch(command);
}
```

Pour terminer, il faut valider l'ensemble :

```
int[] counts = stat.executeBatch();
```

L'appel à `executeBatch` renvoie un tableau contenant les nombres de lignes modifiées par chaque commande venant d'être validée. Rappelez-vous qu'un seul appel à `executeUpdate` renvoie un entier correspondant au nombre de lignes ayant été modifiées par la commande. Dans notre exemple, la méthode `executeBatch` renvoie un tableau dont le premier élément vaut 0 (puisque la commande CREATE TABLE renvoie un nombre de lignes valant 0), et dont tous les autres éléments valent 1 (puisque chaque commande INSERT affecte une seule ligne).

Pour une bonne gestion des erreurs en mode automatisé, il convient de traiter l'exécution de l'ensemble des commandes comme une seule transaction. Si un traitement automatisé est interrompu en cours d'exécution, vous pouvez l'annuler et revenir à l'état précédant l'exécution de ce traitement automatisé.

Pour commencer, désactivez le mode de validation automatique, puis mettez en place le traitement automatisé, exécutez-le, validez-le et, pour terminer, activez à nouveau le mode de validation automatique :

```
boolean autoCommit = conn.getAutoCommit();
conn.setAutoCommit(false);
Statement stat = conn.createStatement();
```

```
...  
// appels successifs à stat.addBatch(. . .);  
...  
stat.executeBatch();  
conn.commit();  
conn.setAutoCommit(autoCommit);
```

INFO

Vous ne pouvez émettre d'instructions de mise à jour que dans un lot. Si vous émettez une requête SELECT, vous déclenchez une exception.

java.sql.Connection 1.1

- `void setAutoCommit (boolean b)`
Définit le mode d'attribution automatique de cette connexion sur `b`. Si l'attribution automatique vaut `true`, toutes les instructions sont attribuées dès la fin de leur exécution.
- `boolean getAutoCommit()`
Récupère le mode d'attribution automatique de cette connexion.
- `void commit()`
Attribue toutes les instructions qui ont été émises depuis la dernière attribution.
- `void rollback()`
Annule l'effet de toutes les instructions émises depuis la dernière attribution.
- `Savepoint setSavepoint() 1.4`
Définit un point de sauvegarde anonyme.
- `Savepoint setSavepoint(String name) 1.4`
Définit un point de sauvegarde nommé.
- `void rollback(Savepoint svpt) 1.4`
Procède à une annulation jusqu'au point de sauvegarde donné.
- `void releaseSavepoint(Savepoint svpt) 1.4`
Libère le point de sauvegarde donné.

java.sql.Savepoint 1.4

- `int getSavepointId()`
Récupère l'ID de ce point de sauvegarde anonyme ou déclenche une `SQLException` s'il s'agit d'un point de sauvegarde nommé.
- `String getSavepointName()`
Récupère le nom de ce point de sauvegarde ou déclenche une `SQLException` s'il s'agit d'un point de sauvegarde anonyme.

java.sql.Statement 1.1

- `void addBatch(String command) 1.2`
Ajoute la commande à l'ensemble des commandes automatisées pour cette instruction.

206 Au cœur de Java 2 - Fonctions avancées

- `int[] executeBatch()` 1.2

Exécute toutes les commandes du traitement automatisé courant. Renvoie un tableau contenant le nombre de lignes modifiées par chaque commande du script.

java.sql.DatabaseMetaData 1.1

- `boolean supportsBatchUpdates()` 1.2

Renvoie `true` si le pilote supporte les mises à jour automatisées.

Gestion avancée des connexions

La configuration plutôt simpliste des connexions de bases de données, effectuée à l'aide d'un fichier `database.properties`, comme évoquée dans les sections précédentes, convient malgré tout aux petits programmes de test. Toutefois, elle ne pourra être appliquée aux plus grandes applications.

Lors du déploiement d'une application JDBC dans un environnement d'entreprise, la gestion des connexions à la base de données est totalement intégrée dans l'interface JNDI (*Java Naming and Directory Interface*). Les propriétés des sources de données dans la totalité de l'entreprise peuvent être stockées dans un répertoire. Ceci autorise une gestion centralisée des noms d'utilisateurs, des mots de passe, des noms de bases de données, mais également des URL JDBC.

Dans un tel environnement, le code qui suit permet d'établir une connexion à une base de données :

```
Context jndiContext = new InitialContext();
DataSource source
    = (DataSource) jndiContext.lookup("java:comp/env/jdbc/corejava");
Connection conn = source.getConnection();
```

Vous remarquerez que `DriverManager` n'est plus utilisé. A sa place, le service JNDI localise une *source de données*. Celle-ci correspond à une interface qui permet de simples connexions JDBC ainsi que des services plus avancés, notamment l'exécution de transactions distribuées qui impliquent plusieurs bases de données. L'interface `DataSource` est définie dans le module d'extension standard `javax.sql`.

Bien entendu, la source des données doit être configurée à un moment ou à un autre. Si vous écrivez des programmes de bases de données qui s'exécutent dans un conteneur de servlet comme Apache Tomcat ou dans un serveur d'application comme BEA WebLogic, placez la configuration de la base de données (y compris l'URL JDBC, le nom d'utilisateur et le mot de passe) dans un fichier de configuration.

La gestion des noms d'utilisateur et des identifiants ne constitue que l'un des aspects qui nécessitent une attention particulière, le deuxième aspect à prendre en compte étant le coût de l'établissement des connexions.

Nos programmes simples de base de données établissent une seule connexion de base de données au début du programme et la ferment à la fin du programme. Toutefois, dans de nombreuses situations de programmation, cette approche ne fonctionnera pas. Envisagez une application Web ordinaire ; elle répond en parallèle à de nombreuses requêtes de pages. Ces multiples requêtes peuvent avoir besoin d'un accès simultané à la base de données. Avec plusieurs bases de données, une connexion n'a pas pour objectif d'être partagée par plusieurs threads. Ainsi, chaque requête a besoin de sa propre connexion. Une approche simpliste consisterait à établir une connexion pour chaque requête de page et à la fermer par la suite. Toutefois, ceci se révélerait trop coûteux. L'établissement d'une

connexion à une base de données peut être assez long. Les connexions doivent donc pouvoir être utilisées pour plusieurs requêtes, et non fermées après une ou deux requêtes.

La solution consiste à *mettre les connexions en commun*. Cela revient à ne pas fermer physiquement les connexions à la base de données, mais à les conserver dans une file d'attente et à les réutiliser. La mise en commun des connexions constitue un service important et la spécification JDBC propose des liens pour les personnes chargées de la mettre en place. Malgré tout, Java SDK ne fournit pas de support pour ce service et les fabricants de bases de données n'en incluent généralement pas dans leur pilote JDBC. Au lieu de cela, les fournisseurs de serveurs d'applications, comme BEA WebLogic ou IBM WebSphere, proposent des implémentations de mise en commun des connexions dans le cadre du module du serveur d'applications.

L'utilisation d'une mise en commun des connexions demeure totalement transparente pour le programmeur. Vous pouvez ainsi acquérir une connexion à partir d'une source de connexions mises en commun en vous procurant une source de données et en appelant `getConnection`. Lorsque vous aurez terminé d'utiliser la connexion, appelez `close`. Ceci ne ferme pas la connexion physique mais indique à l'ensemble des connexions que vous avez fini de l'utiliser.

Vous avez maintenant appris les bases du JDBC et vous savez implémenter de simples applications de base de données. Toutefois, comme nous l'avons mentionné au début de ce chapitre, les bases de données sont complexes et quelques sujets avancés n'ont pas été abordés dans ce chapitre d'introduction. Pour obtenir un aperçu des fonctions avancées de JDBC, retrouvez les spécifications JDBC à l'adresse :

<http://java.sun.com/products/jdbc>.

Introduction au LDAP

Dans les sections précédentes, vous avez appris à interagir avec une base de données relationnelle. Nous allons maintenant brièvement étudier les bases de données hiérarchiques qui utilisent LDAP (*Lightweight Directory Access Protocol*). Cette section est adaptée de l'ouvrage *Core JavaServer Faces*, de Geary et Horstmann (Sun Microsystems Press, 2004).

Le LDAP est préféré aux bases de données relationnelles lorsque les données d'application respectent naturellement une structure en arbre et que les opérations de lecture dépassent de loin les opérations d'écriture. Le LDAP convient mieux pour le stockage de répertoires contenant des données comme les noms d'utilisateur, les mots de passe et les autorisations.

INFO

Pour une explication détaillée du LDAP, nous vous recommandons chaudement l'ouvrage (en langue anglaise) *Understanding and Deploying LDAP Directory Services*, 2^e édition, de Timothy Howes et al. (Macmillan, 2003).

Une base de données LDAP conserve toutes les données dans une arborescence, et non dans l'ensemble de tableaux des bases relationnelles. Chaque entrée de l'arbre possède les éléments suivants :

- **zéro attribut ou plus.** Un attribut possède une identité et une valeur, par exemple `cn=Jean C. Public` (l'identifiant `cn` indique le "nom commun" ; voir Tableau 3.10 pour connaître la signification des attributs LDAP les plus habituels).

208 *Au cœur de Java 2 - Fonctions avancées*

- **Une ou plusieurs classes d'objet.** Une classe d'objet définit l'ensemble des attributs obligatoires et facultatifs pour cet élément. Par exemple, la classe d'objet `person` définit un attribut obligatoire `cn` et un attribut facultatif `telephoneNumber`. Bien entendu, les classes d'objet diffèrent des classes Java, mais elles acceptent aussi la notion d'héritage. Par exemple, `organizationalPerson` est une sous-classe de `person`, avec d'autres attributs.
- **Un nom distinctif (par exemple, `uid=jqpublic,ou=people,dc=mycompany,dc=com`).** Le nom distinctif est une suite d'attributs qui constitue un chemin reliant l'entrée à la racine de l'arbre. Il peut exister d'autres chemins, mais l'un d'entre eux doit être défini comme distinctif.

Tableau 3.10 : Attributs LDAP fréquemment utilisés

<i>ID de l'attribut</i>	<i>Signification</i>
dc	Composant de domaine
cn	Nom commun
sn	Nom de famille
dn	Nom distinctif
o	Organisation
ou	Unité organisationnelle
uid	Identifiant unique

La Figure 3.9 présente un exemple d'arborescence de répertoire.

L'organisation d'une arborescence de répertoires de même que le caractère des informations à y placer peuvent donner lieu à discussion. Nous ne traiterons pas de ce sujet ici, nous supposons simplement qu'un schéma organisationnel a été établi et que le répertoire a été rempli par les données idoines de l'utilisateur.

Configurer un serveur LDAP

Plusieurs options permettent d'exécuter un serveur LDAP pour tester les programmes de cette section. Voici les choix les plus fréquents :

- IBM Tivoli Directory Server ;
- Microsoft Active Directory ;
- Novell eDirectory ;
- OpenLDAP (<http://openldap.org>), un serveur gratuit, disponible pour Linux et Windows et intégré dans Mac OS X ;
- Sun Java System Directory Server.

Vous trouverez ici quelques brèves instructions pour configurer OpenLDAP. Si vous utilisez un autre serveur, sachez que les étapes de base sont similaires.

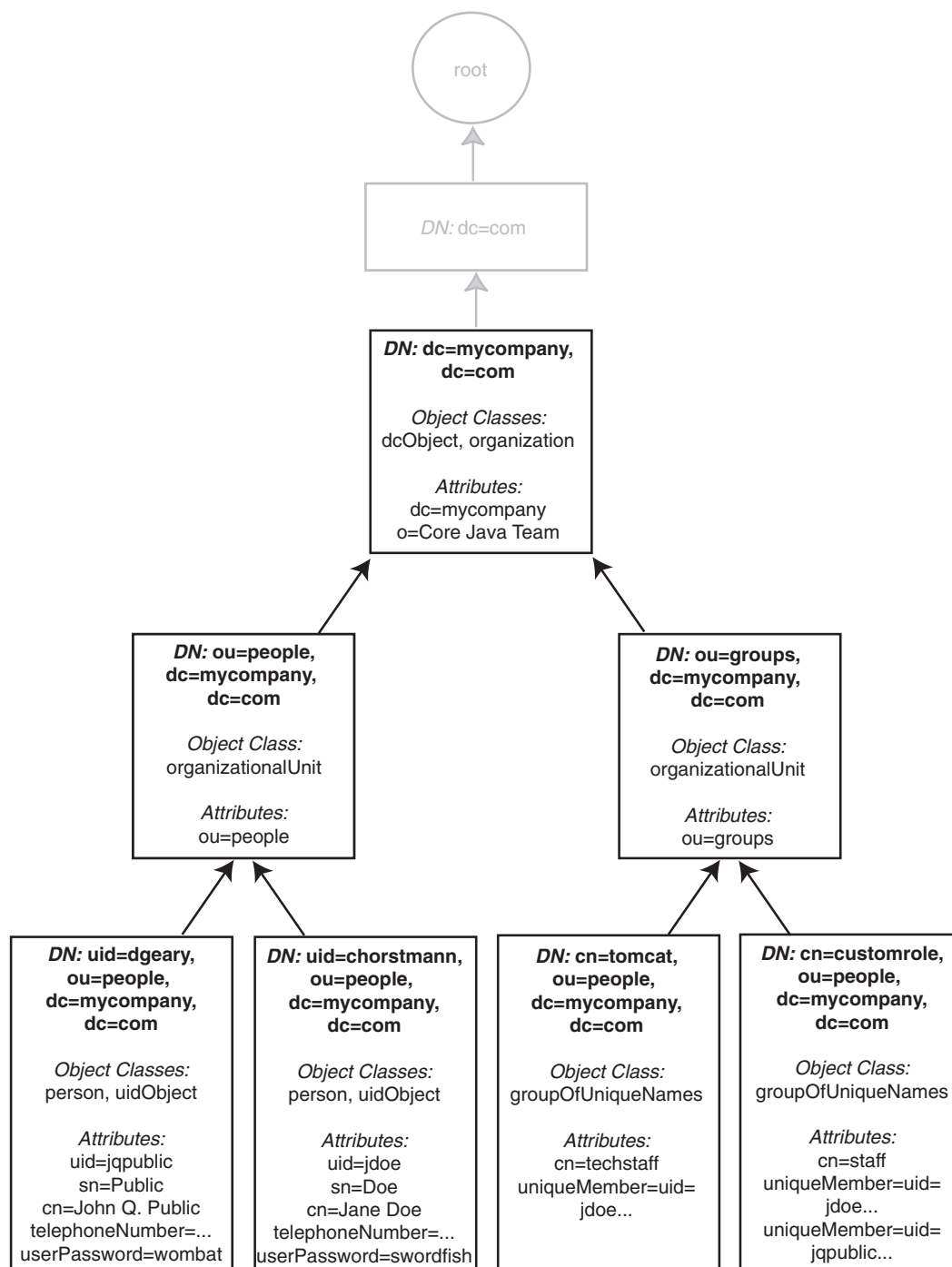


Figure 3.9

Une arborescence de répertoire.

210 Au cœur de Java 2 - Fonctions avancées

Si vous utilisez OpenLDAP, vous devez modifier le fichier `slapd.conf` avant de lancer le serveur LDAP (sous Linux, le fichier `slapd.conf` se trouve par défaut dans `/usr/local/etc/openldap`). Modifiez l'entrée du suffixe pour la faire concorder avec l'ensemble de données d'exemple. Cette entrée spécifie le suffixe du nom distinctif pour ce serveur. Elle doit indiquer

```
suffix "dc=mycompany,dc=com"
```

Vous devez aussi configurer un utilisateur LDAP ayant des droits d'administrateur pour la modification des données du répertoire. Dans OpenLDAP, ajoutez ces lignes à `slapd.conf` :

```
rootdn "cn=Manager,dc=mycompany,dc=com"
rootpw secret
```

Vous pouvez maintenant démarrer le serveur LDAP. Sous Linux, exécutez `/usr/local/libexec/slapd`.

Remplissez ensuite le serveur avec les données d'exemple. La plupart des serveurs LDAP permettent l'importation de données LDIF (*Lightweight Directory Interchange Format*). LDIF est un format lisible par l'homme qui énumère simplement toutes les entrées du répertoire, y compris leurs noms distinctifs, les classes d'objet et les attributs. L'Exemple 3.6 montre un fichier LDIF décrivant nos données d'exemple.

Exemple 3.6 : sample.ldif

```
# Définir une entrée de haut niveau
dn: dc=mycompany,dc=com
objectClass: dcObject
objectClass: organization
dc: mycompany
o: Core Java Team

# Définir une entrée pour stocker des personnes
# Les recherches d'utilisateurs se basent sur cette entrée
dn: ou=people,dc=mycompany,dc=com
objectClass: organizationalUnit
ou: people

# Définir une entrée utilisateur pour Jean C. Public
dn: uid=jpublic,ou=people,dc=mycompany,dc=com
objectClass: person
objectClass: uidObject
uid: jpublic
sn: Public
cn: Jean C. Public
telephoneNumber: +1 408 555 0017
userPassword: wombat

# Définir une entrée utilisateur pour Jane Doe
dn: uid=jdoe,ou=people,dc=mycompany,dc=com
objectClass: person
objectClass: uidObject
uid: jdoe
sn: Doe
cn: Jane Doe
telephoneNumber: +1 408 555 0029
userPassword: heffalump

# Définir une entrée qui contiendra les groupes LDAP
```

```
# Les recherches de rôles se basent sur cette entrée
dn: ou=groups,dc=mycompany,dc=com
objectClass: organizationalUnit
ou: groups

# Définir une entrée pour le groupe "techstaff"
dn: cn=techstaff,ou=groups,dc=mycompany,dc=com
objectClass: groupOfUniqueNames
cn: techstaff
uniqueMember: uid=jdoe,ou=people,dc=mycompany,dc=com

# Définir une entrée pour le groupe "staff"
dn: cn=staff,ou=groups,dc=mycompany,dc=com
objectClass: groupOfUniqueNames
cn: staff
uniqueMember: uid=jpublic,ou=people,dc=mycompany,dc=com
uniqueMember: uid=jdoe,ou=people,dc=mycompany,dc=com
```

Par exemple, avec OpenLDAP, vous utilisez l'outil `ldapadd` pour ajouter les données au répertoire :

```
ldapadd -f sample.ldif -x -D "cn=Manager,dc=mycompany,dc=com" -w secret
```

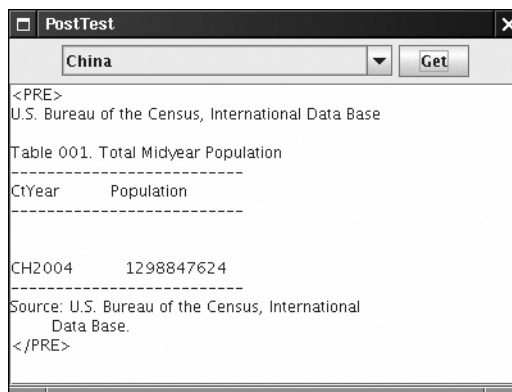
Avant de continuer, il est conseillé de vérifier que le répertoire contient bien les données dont vous avez besoin. Nous vous suggérons de télécharger le navigateur/éditeur LDAP de Jarek Gawor à l'adresse <http://www-unix.mcs.anl.gov/~gawor/ldap/>. Ce programme Java vous permet de naviguer dans le contenu de tout serveur LDAP. Lancez le programme et configurez-le avec les options suivantes :

```
Host: localhost
Base DN: dc=mycompany,dc=com
Anonymous bind: unchecked
User DN: cn=Manager
Append base DN: checked
Password: secret
```

Vérifiez que le serveur LDAP a démarré, puis procédez à la connexion. Si tout va bien, vous devriez voir apparaître une arborescence de répertoires semblable à celle affichée à la Figure 3.10.

Figure 3.10

*Inspection
d'une arborescence
de répertoires LDAP.*



Accéder aux informations du répertoire LDAP

Une fois votre base de données LDAP remplie, connectez-vous-y avec un programme Java. Vous utilisez ainsi l'interface JNDI (*Java Naming and Directory Interface*), qui unifie les divers protocoles de répertoires.

Commencez par obtenir un contexte du répertoire LDAP, grâce à l'instruction suivante :

```
Hashtable env = new Hashtable();
env.put(Context.SECURITY_PRINCIPAL, username);
env.put(Context.SECURITY_CREDENTIALS, password);
DirContext initial = new InitialDirContext(env);
DirContext context = (DirContext) initial.lookup("ldap://localhost:389");
```

Ici, nous nous connectons au serveur LDAP, sur l'hôte local. Le numéro de port 389 correspond au port LDAP par défaut.

Si vous vous connectez à la base de données LDAP avec une combinaison non valide de nom d'utilisateur et de mot de passe, le système déclenche une `AuthenticationException`.

INFO

Le didacticiel de Sun sur le JNDI suggère une autre manière pour se connecter au serveur :

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:389");
env.put(Context.SECURITY_PRINCIPAL, userDN);
env.put(Context.SECURITY_CREDENTIALS, password);
DirContext context = new InitialDirContext(env);
```

Il n'est toutefois pas souhaitable de coder en dur le fournisseur de Sun LDAP. JNDI possède un mécanisme adéquat pour configurer les fournisseurs, et il ne vaut mieux pas le contourner à la légère.

Pour répertorier les attributs d'une entrée donnée, indiquez son nom distinctif, puis utilisez la méthode `getAttributes` :

```
Attributes attrs =
context.getAttributes("uid=jqpublic,ou=people,dc=mycompany,dc=com");
```

Vous pouvez obtenir un attribut spécifique grâce à la méthode `get`, par exemple

```
Attribute commonNameAttribute = attrs.get("cn");
```

Pour énumérer tous les attributs, utilisez la classe `NamingEnumeration`. Ses concepteurs ont pensé qu'ils pourraient aussi améliorer le protocole d'itération standard de Java et nous ont fourni ce pattern d'utilisation :

```
NamingEnumeration<? extends Attribute> attrEnum = attrs.getAll();
while (attrEnum.hasMore())
{
    Attribute attr = attrEnum.next();
    String id = attr.getID();
    . . .
}
```

Remarquez l'utilisation de `hasMore` au lieu de `hasNext`.

Si vous savez qu'un attribut ne possède qu'une seule valeur, vous pouvez appeler la méthode `get` pour la récupérer :

```
String commonName = (String) commonNameAttribute.get();
```

Lorsqu'un attribut peut avoir plusieurs valeurs, utilisez un autre `NamingEnumeration` pour les énumérer toutes :

```
NamingEnumeration<?> valueEnum = attr.getAll();
while (valueEnum.hasMore())
{
    Object value = valueEnum.next();
    . . .
}
```

INFO

Depuis le JDK 5.0, `NamingEnumeration` est un type générique. La borne de type `<? extends Attribute>` signifie que l'énumération produit des objets d'un type inconnu qui est un sous-type de `Attribute`. Vous n'avez donc pas besoin de transtyper la valeur renvoyée par `next`, elle est de type `Attribute`. En l'absence du générique, vous écririez :

```
NamingEnumeration attrEnum = attrs.getAll();
Attribute attr = (Attribute) attrEnum.next();
```

Toutefois, `NamingEnumeration<?>` ne sait pas ce qu'il énumère. Sa méthode `next` renvoie un `Object`.

Vous savez maintenant comment interroger le répertoire à la recherche de données utilisateur. Avançons maintenant jusqu'à la modification du contenu du répertoire.

Pour ajouter une entrée, rassemblez les jeux d'attributs dans un objet `BasicAttributes` (la classe `BasicAttributes` implémente l'interface `Attributes`).

```
Attributes attrs = new BasicAttributes();
attrs.put("uid", "alain");
attrs.put("sn", "Louis");
attrs.put("cn", "Anne-Lise");
attrs.put("telephoneNumber", "+1 408 555 0033");
String password = "redqueen";
attrs.put("userPassword", password.getBytes());
// l'attribut suivant a deux valeurs
Attribute objclass = new BasicAttribute("objectClass");
objclass.add("uidObject");
objclass.add("person");
attrs.put(objclass);
```

Appelez ensuite la méthode `createSubcontext`. Indiquez le nom distinctif de la nouvelle entrée et l'ensemble d'attributs :

```
context.createSubcontext("uid=alee,ou=people,dc=mycompany,dc=com", attrs);
```

ATTENTION

Lors de l'assemblage des attributs, n'oubliez pas qu'ils sont contrôlés par rapport au schéma. Ne fournissez pas d'attributs inconnus et assurez-vous de la présence de tous les attributs nécessaires à la classe d'objet. Par exemple, si vous oubliez le `sn` de `person`, la méthode `createSubcontext` échouera.

214 *Au cœur de Java 2 - Fonctions avancées*

Pour supprimer une entrée, appelez `destroySubcontext` :

```
context.destroySubcontext("uid=jdoeF,ou=people,dc=mycompany,dc=com");
```

Enfin, pour modifier les attributs d'une entrée, appelez la méthode

```
context.modifyAttributes(distinguishedName, flag, attrs);
```

Ici, `flag` vaudra l'une des valeurs suivantes :

```
DirContext.ADD_ATTRIBUTE
DirContext.REMOVE_ATTRIBUTE
DirContext.REPLACE_ATTRIBUTE
```

Le paramètre `attrs` contient un ensemble d'attributs à ajouter, à supprimer ou à remplacer.

Le constructeur `BasicAttributes(String, Object)` élabore un jeu d'attributs ne contenant qu'un seul élément. Par exemple :

```
context.modifyAttributes("uid=alee,ou=people,dc=mycompany,dc=com",
    DirContext.ADD_ATTRIBUTE,
    new BasicAttributes("title", "CTO"));

context.modifyAttributes("uid=alee,ou=people,dc=mycompany,dc=com",
    DirContext.REMOVE_ATTRIBUTE,
    new BasicAttributes("telephoneNumber", "+1 408 555 0033"));

context.modifyAttributes("uid=alee,ou=people,dc=mycompany,dc=com",
    DirContext.REPLACE_ATTRIBUTE,
    new BasicAttributes("userPassword", password.getBytes()));
```

Enfin, lorsque vous avez terminé avec un contexte, mettez-y fin :

```
context.close();
```

Le programme de l'Exemple 3.7 montre comment accéder à une base de données hiérarchique *via* LDAP. Le programme permet d'afficher, de modifier et de supprimer des informations d'une base de données, en faisant appel aux données de l'Exemple 3.6.

Entrez un `uid` dans le champ de texte et cliquez sur le bouton "Rechercher" pour trouver une entrée. Si vous modifiez l'entrée et que vous cliquez sur "Enregistrer", vos changements seront sauvegardés. Si vous modifiez le champ `uid`, vous créez une nouvelle entrée, sinon l'entrée existante est actualisée. Vous pouvez également effacer l'entrée en cliquant sur le bouton "Supprimer" (voir Figure 3.11).

Les points suivants décrivent brièvement le programme.

1. La configuration du serveur LDAP figure dans le fichier `ldapservlet.properties`. Il définit l'URL, le nom d'utilisateur et le mot de passe du serveur, comme ceci :

```
ldap.username=cn=Manager,dc=mycompany,dc=com
ldap.password=secret
ldap.url=ldap://localhost:389
```

La méthode `getContext` lit le fichier et récupère le contexte du répertoire.

2. Lorsque l'utilisateur clique sur le bouton "Rechercher", la méthode `findEntry` rapporte le jeu d'attributs de l'entrée ayant l'`uid` donné. Il permet de construire un nouveau `DataPanel`.
3. Le constructeur `DataPanel` passe en revue le jeu d'attributs et ajoute une étiquette et un champ de texte pour chaque paire ID/valeur.

Figure 3.11

Accéder à une base de données hiérarchique.

4. Lorsque l'utilisateur clique sur le bouton "Supprimer", la méthode `deleteEntry` efface l'entrée ayant l'uid donné et rejette le panneau de données.
5. Lorsque l'utilisateur clique sur le bouton "Enregistrer", le `DataPanel` construit un objet `BasicAttributes` avec le contenu actuel des champs de texte. La méthode `saveEntry` vérifie si l'uid a changé. Si l'utilisateur l'a modifié, une nouvelle entrée est créée, sinon les attributs modifiés sont mis à jour. Le code de modification est simple car nous n'avons qu'un attribut avec plusieurs valeurs, à savoir `objectClass`. En général, il faudra fournir plus d'efforts pour gérer les diverses valeurs de chaque attribut.
6. Comme dans le programme de l'Exemple 3.4, nous fermons le contexte de répertoires à la fermeture de la fenêtre du cadre.

Vous en savez maintenant suffisamment sur le fonctionnement des répertoires pour réaliser les tâches courantes d'un travail avec les répertoires LDAP. Pour en savoir plus sur le JNDI, consultez son didacticiel à l'adresse <http://java.sun.com/products/jndi/tutorial>.

Exemple 3.7 : LDAPTest.java

```
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.naming.*;
import javax.naming.directory.*;
import javax.swing.*;

/**
 * Ce programme présente un accès à une base de données
 * hiérarchique via LDAP.
 */
public class LDAPTest
{
    public static void main(String[] args)
    {
        JFrame frame = new LDAPFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

216 *Au cœur de Java 2 - Fonctions avancées*

```

    }
}

/**
 * Le cadre qui contient le panneau de données et
 * les boutons de navigation.
 */
class LDAPFrame extends JFrame
{
    public LDAPFrame()
    {
        setTitle("LDAPTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        JPanel northPanel = new JPanel();
        northPanel.setLayout(new java.awt.GridLayout(1, 2, 3, 1));
        northPanel.add(new JLabel("uid", SwingConstants.RIGHT));
        uidField = new JTextField();
        northPanel.add(uidField);
        add(northPanel, BorderLayout.NORTH);

        JPanel buttonPanel = new JPanel();
        add(buttonPanel, BorderLayout.SOUTH);

        findButton = new JButton("Rechercher");
        findButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    findEntry();
                }
            });
        buttonPanel.add(findButton);

        saveButton = new JButton("Enregistrer");
        saveButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    saveEntry();
                }
            });
        buttonPanel.add(saveButton);

        deleteButton = new JButton("Supprimer");
        deleteButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    deleteEntry();
                }
            });
        buttonPanel.add(deleteButton);

        addWindowListener(new
            WindowAdapter()

```

```

        {
            public void windowClosing(WindowEvent event)
            {
                try
                {
                    if (context != null) context.close();
                }
                catch (NamingException e)
                {
                    e.printStackTrace();
                }
            }
        });
    }

    /**
     * Retrouve l'entrée de l'uid dans le champ de texte.
     */
    public void findEntry()
    {
        try
        {
            if (scrollPane != null) remove(scrollPane);
            String dn = "uid=" + uidField.getText() +
                ",ou=people,dc=mycompany,dc=com";
            if (context == null) context = getContext();
            attrs = context.getAttributes(dn);
            dataPanel = new DataPanel(attrs);
            scrollPane = new JScrollPane(dataPanel);
            add(scrollPane, BorderLayout.CENTER);
            validate();
            uid = uidField.getText();
        }
        catch (NamingException e)
        {
            JOptionPane.showMessageDialog(this, e);
        }
        catch (IOException e)
        {
            JOptionPane.showMessageDialog(this, e);
        }
    }

    /**
     * Enregistre les changements apportés par l'utilisateur.
     */
    public void saveEntry()
    {
        try
        {
            if (dataPanel == null) return;
            if (context == null) context = getContext();
            if (uidField.getText().equals(uid)) // actualiser l'entrée
            {
                String dn = "uid=" + uidField.getText() +
                    ",ou=people,dc=mycompany,dc=com";
                Attributes editedAttrs = dataPanel.getEditedAttributes();
                NamingEnumeration<? extends Attribute> attrEnum =
                    attrs.getAll();
            }
        }
    }

```


218 Au cœur de Java 2 - Fonctions avancées

```

        while (attrEnum.hasMore())
        {
            Attribute attr = attrEnum.next();
            String id = attr.getID();
            Object value = attr.get();
            Attribute editedAttr = editedAttrs.get(id);
            if (editedAttr != null &&
                !attr.get().equals(editedAttr.get()))
                context.modifyAttributes(dn,
                    DirContext.REPLACE_ATTRIBUTE,
                    new BasicAttributes(id, editedAttr.get()));
        }
    }
    else // créer une nouvelle entrée
    {
        String dn = "uid=" + uidField.getText() +
            ",ou=people,dc=mycompany,dc=com";
        attrs = dataPanel.getEditedAttributes();
        Attribute objclass = new BasicAttribute("objectClass");
        objclass.add("uidObject");
        objclass.add("person");
        attrs.put(objclass);
        attrs.put("uid", uidField.getText());
        context.createSubcontext(dn, attrs);
    }

    findEntry();
}
catch (NamingException e)
{
    JOptionPane.showMessageDialog(LDAPFrame.this, e);
    e.printStackTrace();
}
catch (IOException e)
{
    JOptionPane.showMessageDialog(LDAPFrame.this, e);
    e.printStackTrace();
}
}

/**
 * Efface l'entrée de l'uid du champ de texte.
 */
public void deleteEntry()
{
    try
    {
        String dn = "uid=" + uidField.getText() +
            ",ou=people,dc=mycompany,dc=com";
        if (context == null) context = getContext();
        context.destroySubcontext(dn);
        uidField.setText("");
        remove(scrollPane);
        scrollPane = null;
        repaint();
    }
    catch (NamingException e)
    {
        JOptionPane.showMessageDialog(LDAPFrame.this, e);
    }
}

```

```

        e.printStackTrace();
    }
    catch (IOException e)
    {
        JOptionPane.showMessageDialog(LDAPFrame.this, e);
        e.printStackTrace();
    }
}

/**
 * Récupère un contexte des propriétés spécifiées
 * dans le fichier ldapserver.properties
 * @return Le contexte de répertoires
 */
public static DirContext getContext()
    throws NamingException, IOException
{
    Properties props = new Properties();
    FileInputStream in = new FileInputStream("ldapserver.properties");
    props.load(in);
    in.close();

    String url = props.getProperty("ldap.url");
    String username = props.getProperty("ldap.username");
    String password = props.getProperty("ldap.password");

    Hashtable<String, String> env = new Hashtable<String, String>();
    env.put(Context.SECURITY_PRINCIPAL, username);
    env.put(Context.SECURITY_CREDENTIALS, password);
    DirContext initial = new InitialDirContext(env);
    DirContext context = (DirContext) initial.lookup(url);

    return context;
}

public static final int DEFAULT_WIDTH = 300;
public static final int DEFAULT_HEIGHT = 200;

private JButton findButton;
private JButton saveButton;
private JButton deleteButton;

private JTextField uidField;
private DataPanel dataPanel;
private Component scrollPane;

private DirContext context;
private String uid;
private Attributes attrs;
}

/**
 * Ce panneau affiche le contenu d'un jeu de résultats.
 */
class DataPanel extends JPanel
{
    /**
     * Construit le panneau de données.
     * @param attributes Les attributs de l'entrée

```

220 *Au cœur de Java 2 - Fonctions avancées*

```

*/
public DataPanel(Attributes attrs) throws NamingException
{
    setLayout(new java.awt.GridLayout(0, 2, 3, 1));

    NamingEnumeration<? extends Attribute> attrEnum = attrs.getAll();
    while (attrEnum.hasMore())
    {
        Attribute attr = attrEnum.next();
        String id = attr.getID();

        NamingEnumeration<?> valueEnum = attr.getAll();
        while (valueEnum.hasMore())
        {
            Object value = valueEnum.next();
            if (id.equals("userPassword"))
                value = new String((byte[]) value);

            JLabel idLabel = new JLabel(id, SwingConstants.RIGHT);
            JTextField valueField = new JTextField("" + value);
            if (id.equals("objectClass"))
                valueField.setEditable(false);
            if (!id.equals("uid"))
            {
                add(idLabel);
                add(valueField);
            }
        }
    }
}

public Attributes getEditedAttributes()
{
    Attributes attrs = new BasicAttributes();
    for (int i = 0; i < getComponentCount(); i += 2)
    {
        JLabel idLabel = (JLabel) getComponent(i);
        JTextField valueField = (JTextField) getComponent(i + 1);
        String id = idLabel.getText();
        String value = valueField.getText();
        if (id.equals("userPassword"))
            attrs.put("userPassword", value.getBytes());
        else if (!id.equals("") && !id.equals("objectClass"))
            attrs.put(id, value);
    }
    return attrs;
}
}

```

javax.naming.directory.InitialDirContext 1.3

- **InitialDirContext(Hashtable env)**

Construit un contexte de répertoire, à l'aide des paramètres d'environnement donnés. La table de hachage peut contenir des liaisons pour `Context.SECURITY_PRINCIPAL`, `Context.SECURITY_CREDENTIALS` et d'autres clés (voir la documentation de l'API pour en savoir plus sur l'interface `javax.naming.Context`).

javax.naming.Context 1.3

- `Object lookup(String name)`
Recherche l'objet ayant le nom donné. La valeur renvoyée dépend de la nature de ce contexte. C'est généralement un contexte de sous-arbre ou un objet feuille.
- `Context createSubcontext(String name)`
Crée un sous-contexte avec le nom donné. Il devient enfant de ce contexte. Tous les composants de chemin du nom doivent exister, à part le dernier.
- `void destroySubcontext(String name)`
Détruit le sous-contexte ayant le nom donné. Tous les composants de chemin du nom doivent exister, à part le dernier.
- `void close()`
Ferme ce contexte.

javax.naming.directory.DirContext 1.3

- `Attributes getAttributes(String name)`
Récupère les attributs de l'entrée ayant le nom donné.
- `void modifyAttributes(String name, int flag, Attributes modes)`
Modifie les attributs de l'entrée ayant le nom donné. La valeur `flag` vaudra `DirContext.ADD_ATTRIBUTE`, `DirContext.REMOVE_ATTRIBUTE` ou `DirContext.REPLACE_ATTRIBUTE`.

javax.naming.directory.Attributes 1.3

- `Attribute get(String id)`
Récupère l'attribut ayant l'ID donné.
- `NamingEnumeration<? extends Attribute> getAll()`
Produit une énumération qui passe en revue tous les attributs du jeu d'attributs.
- `Attribute put(Attribute attr)`
- `Attribute put(String id, Object value)`
Ajoutent un attribut à cet ensemble d'attributs.

javax.naming.directory.BasicAttributes 1.3

- `BasicAttributes(String id, Object value)`
Construit un jeu d'attributs qui contient un seul attribut ayant l'ID et la valeur données.

javax.naming.directory.Attribute 1.3

- `String getID()`
Récupère l'ID de cet attribut.
- `Object get()`
Récupère la première valeur de cet attribut si les valeurs sont ordonnées ou une valeur arbitraire si elles ne le sont pas.
- `NamingEnumeration<?> getAll()`
Produit une énumération qui passe en revue toutes les valeurs de cet attribut.

222 *Au cœur de Java 2 - Fonctions avancées*

***javax.naming.NamingEnumeration<T>* 1.3**

- **boolean hasMore()**
Renvoie true si cet objet d'énumération possède d'autres éléments.
- **T next()**
Renvoie l'élément suivant de cette énumération.

4

Objets distribués

Au sommaire de ce chapitre

- ✓ Les rôles du client et du serveur
- ✓ Invocations de méthodes distantes
- ✓ Etablissement d'une invocation de méthode distante
- ✓ Passage de paramètres aux méthodes distantes
- ✓ Activation des objets de serveur
- ✓ IDL Java et CORBA
- ✓ Appels de méthode distante avec SOAP

Périodiquement, la communauté informatique tente d'utiliser des objets intensivement pour tenter de résoudre tous les problèmes du moment. L'idée dissimulée derrière cette tentative est de réunir une gentille petite famille d'objets qui collaboreraient tous entre eux et qui pourraient se trouver à n'importe quel endroit. Ces objets sont naturellement censés communiquer entre eux au moyen de protocoles standard et par l'intermédiaire d'un réseau. Par exemple, un objet se trouvant sur un client peut permettre à l'utilisateur de remplir une requête pour obtenir certaines informations. L'objet client envoie un message correspondant à cette requête à un objet du serveur. Ce dernier collecte les informations de la requête, en passant éventuellement par une base de données ou en appelant un autre objet. Une fois que toutes les informations nécessaires sont réunies, la réponse est envoyée au client. Comme la plupart des nouvelles tendances en programmation, l'intérêt de ce concept est parfois obscurci par l'effet de mode qui s'y rattache. Ce chapitre :

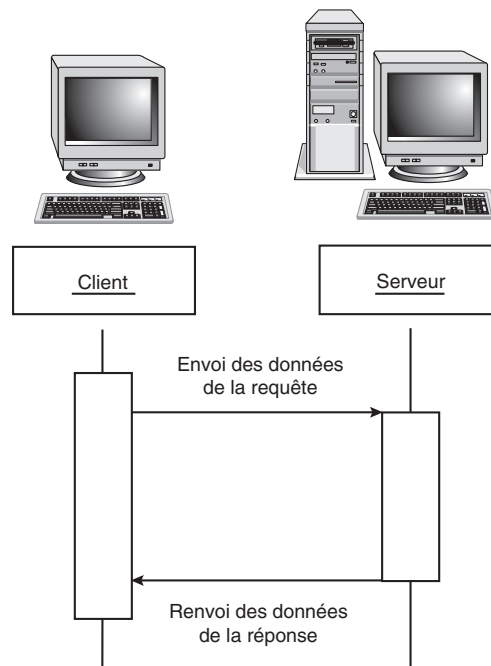
- explique les modèles qui rendent possible la communication entre les objets.
- passe en revue les situations dans lesquelles les objets répartis peuvent être utiles.
- vous montre comment utiliser les objets distants et les *invocations de méthodes distantes* (RMI, ou *Remote Method Invocation*) pour communiquer entre deux machines virtuelles Java (qui peuvent éventuellement se trouver sur deux ordinateurs différents).
- présente CORBA et SOAP, des technologies qui établissent une communication entre plusieurs objets écrits dans des langages de programmation différents.

Les rôles du client et du serveur

Revenons un instant sur ce concept qui consiste à réunir des informations sur un ordinateur client et à les envoyer à un serveur sur un réseau. Nous partons de l'hypothèse qu'un utilisateur sur une machine locale va remplir un formulaire de requête d'informations. Les données sont envoyées au serveur du vendeur, qui traite la requête et envoie la réponse au client, qui peut l'afficher, comme le montre la Figure 4.1.

Figure 4.1

Transmission des objets entre un client et un serveur.



Dans le modèle traditionnel client-serveur, le client traduit la requête dans un format de transmission intermédiaire et envoie la requête au serveur. Le serveur analyse le format de la requête, calcule la réponse et la formate avant de la transmettre au client. Puis le client analyse la réponse obtenue et l'affiche pour l'utilisateur.

Si vous implémentez manuellement cette approche, vous rencontrerez un problème de codage : vous devez concevoir un format de transmission et écrire un code pour convertir vos données dans le format de transmission.

INFO

Dans cet exemple, nous supposons que le client est un ordinateur qui interagit avec un utilisateur humain. Mais il pourrait aussi s'agir d'un ordinateur qui exécute une application Web et exige un service d'un programme qui s'exécute sur un autre ordinateur. Vous avez à nouveau deux objets qui communiquent, l'objet du client se trouve sur l'application Web et l'objet du serveur qui implémente le service. Dans la pratique, il s'agit d'un modèle très répandu. Dans les exemples de ce chapitre, nous nous concentrons sur un modèle plus traditionnel de client-serveur, dans lequel les rôles du client et du serveur sont plus intuitifs.

Ce que nous voulons, c'est un mécanisme permettant au programmeur du client d'effectuer un appel de méthode ordinaire, sans s'inquiéter d'envoyer des données sur le réseau ou d'analyser la réponse. Le problème réside, bien entendu, dans le fait que l'objet qui réalise la tâche n'est pas situé sur la même machine virtuelle, laquelle n'est peut-être même pas implémentée en Java.

La solution consiste à installer un *proxy* pour l'objet de serveur sur le client. Le client appelle le proxy, réalisant ainsi un appel de méthode ordinaire. C'est alors au proxy du client de contacter le serveur.

De même, le programmeur de l'objet de serveur ne veut pas s'encombrer de la communication avec le client. La solution consiste à installer un deuxième objet proxy sur le serveur, qui communique avec le proxy du client et effectue les appels de méthode ordinaires sur l'objet de serveur (voir Figure 4.2).

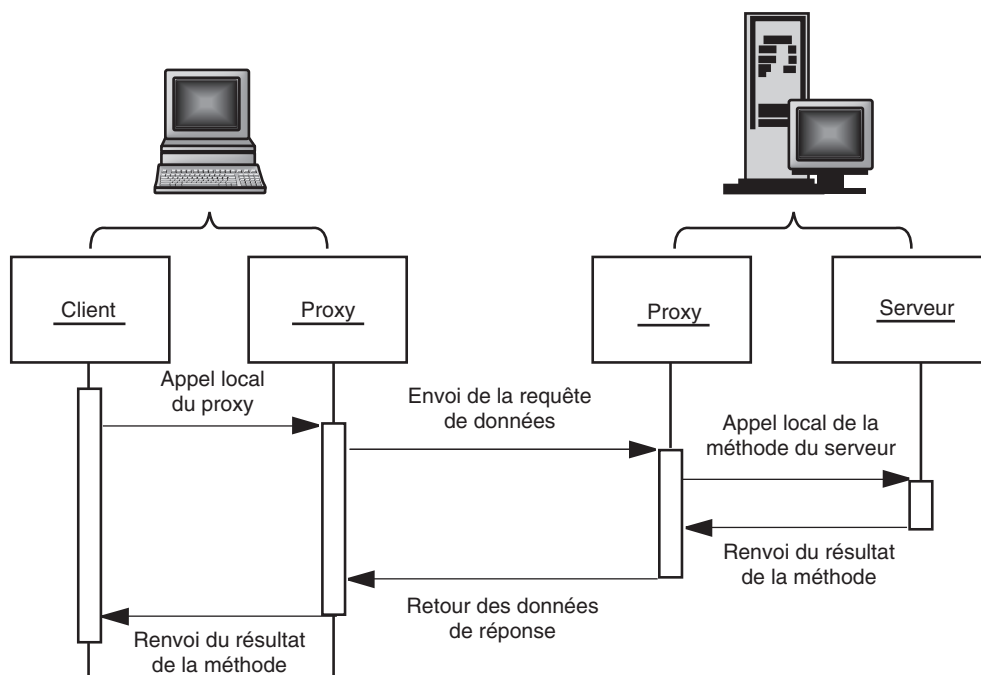


Figure 4.2

Appel de méthode distant avec des proxies.

La communication entre les proxies dépend de la technologie d'implémentation. Il existe trois choix communs :

- RMI, la technologie d'appel de méthode distante de Java, prend en charge les appels de méthode entre les objets Java distribués.
- CORBA prend en charge les appels de méthodes entre des objets, indépendamment de leur langage de programmation. CORBA utilise le protocole Internet Inter-ORB aussi appelé IIOP pour communiquer avec les objets.

226 Au cœur de Java 2 - Fonctions avancées

- SOAP est également indépendant du langage de programmation mais utilise un format de transmission fondé sur le XML.

INFO

Pour la communication entre les objets, Microsoft se sert d'un autre protocole, appelé COM, qui est situé à un niveau plus bas. Des services comparables à ceux d'un ORB sont implémentés directement dans Windows. Auparavant, Microsoft positionnait COM en concurrent de CORBA. Malgré tout, Microsoft met plutôt l'accent sur SOAP.

CORBA et SOAP sont complètement indépendants des langages. Les programmes du client et du serveur peuvent être écrits en C, C++, Java ou dans n'importe quel autre langage. Vous devez fournir une *description d'interface* pour spécifier les signatures des méthodes et les types de données que vos objets peuvent gérer. Ces descriptions sont mises en forme dans un langage spécial appelé Interface Definition Language (IDL) pour CORBA et Web Services Description Language (WSDL) pour SOAP.

Certaines personnes sont convaincues que CORBA deviendra très rapidement très important. Quoi qu'il en soit, et pour parler franchement, CORBA a eu une réputation (parfois bien méritée) d'implémentations complexes et de problèmes d'interopérabilité.

SOAP a peut-être été simple à ses débuts, mais il est lui aussi devenu complexe, car il acquiert de nombreuses fonctions de CORBA. Le protocole XML possède l'avantage d'être (à peu près) lisible par l'homme, ce qui aide au débogage. En revanche, le traitement XML ralentit considérablement les performances. CORBA est généralement plus efficace, même si SOAP convient mieux à une architecture Web.

Si les deux objets communicants sont implémentés en Java, la complexité de CORBA et de SOAP et leur caractère universel ne sont pas nécessaires. Sun a développé un mécanisme plus simple appelé RMI (*Remote Method Invocation*, invocation de méthodes distantes), spécialement pour traiter les communications entre les applications Java.

INFO

Les partisans de CORBA n'aimaient pas trop les RMI à leurs débuts parce qu'elles ne prenaient pas du tout en compte le standard CORBA. Certains efforts sont faits cependant, dans la version JDK 1.3, pour que CORBA et les RMI puissent travailler ensemble. En particulier, vous pouvez utiliser RMI avec le protocole IIOP à la place du protocole propriétaire Java Remote Method Protocol. Pour plus d'informations sur RMI et IIOP, consultez l'article d'introduction en anglais à l'adresse <http://www.javaworld.com/javaworld/jw-12-1999/jw-12-iiop.html> et le didacticiel à l'adresse <http://java.sun.com/j2se/1.4/docs/guide/rmi-iiop/rmiiopexample.html>. Vous en trouverez un très bon exemple à l'adresse <http://www.ocweb.com/jnb/jnbApr2004.html>.

RMI étant plus simple à comprendre que CORBA et SOAP, nous commencerons ce chapitre par une discussion sur ce sujet. Dans la dernière section de ce chapitre, nous introduirons CORBA et SOAP. Nous vous indiquerons comment utiliser CORBA pour communiquer entre des programmes Java et C++ et comment accéder à un service Web avec SOAP.

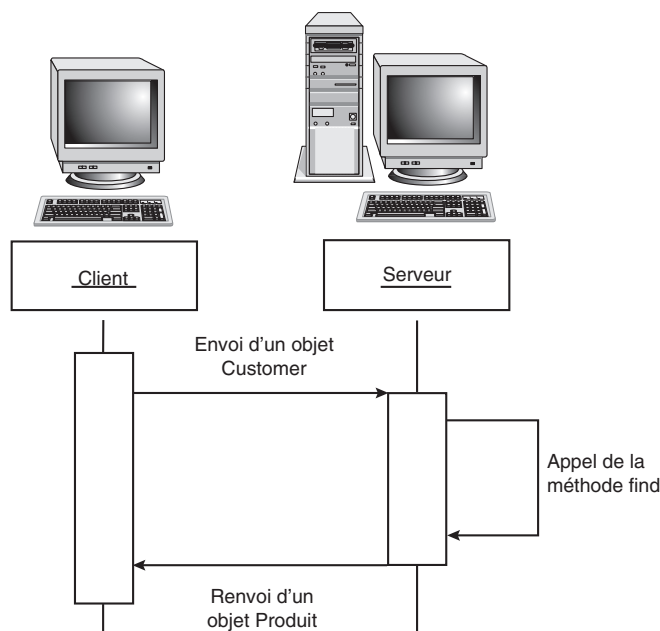
Invocations de méthodes distantes

Le mécanisme RMI vous permet de faire une chose qui peut paraître simple au premier abord. Si vous avez accès à un objet d'une machine distante, vous pouvez appeler des méthodes de cet objet distant. Naturellement, les paramètres de ces méthodes doivent être fournis d'une manière ou d'une autre à l'ordinateur distant, le serveur doit être informé de la méthode à exécuter et la valeur de retour doit être renvoyée correctement. RMI permet de gérer tous ces détails automatiquement.

Par exemple, un client qui cherche des informations sur certains produits peut effectuer sa requête auprès d'un objet Warehouse (entrepôt) du serveur. Il appelle donc une méthode distante, `find`, qui prend un seul paramètre : un objet Customer (client). La méthode `find` renvoie un objet au client : l'objet Product (voir Figure 4.3).

Figure 4.3

Invoquer une méthode distante sur un objet de serveur.



Selon la terminologie RMI, l'objet dont la méthode effectue un appel à distance est appelé l'*objet de client*. L'objet distant est appelé l'*objet de serveur*. Il est important de se rappeler que la terminologie client-serveur ne s'applique qu'à un seul appel de méthode. L'ordinateur qui exécute le code Java qui appelle la méthode distante est le client pour *cet* appel et l'ordinateur qui héberge l'objet qui traite cet appel est le serveur pour *cet* appel. Il est donc tout à fait possible que ces rôles soient inversés un peu plus tard. Le serveur d'un appel précédent peut se retrouver à la place du client lorsqu'il invoque une méthode distante d'un objet situé sur un autre ordinateur.

Stubs et encodage des paramètres

Lorsque le code d'un client veut invoquer une méthode distante sur un objet distant, il commence en fait par appeler une méthode ordinaire sur un proxy, appelé un *stub*. Ce stub se trouve sur la machine

228 *Au cœur de Java 2 - Fonctions avancées*

client et non sur le serveur. Il rassemble tous les paramètres utilisés par la méthode distante dans un bloc d'octets, ce qui lui permet d'obtenir pour chaque paramètre un codage indépendant de la machine. Par exemple, dans le protocole RMI, les nombres sont toujours envoyés en respectant l'ordre big-endian. Les objets, quant à eux, sont encodés selon un procédé de sérialisation décrit dans le Chapitre 12 du Volume 1. L'intérêt de l'encodage des paramètres est de convertir les paramètres en un format adapté à leur transport entre plusieurs machines virtuelles.

Pour résumer, la méthode du stub sur le client construit un bloc de données composé de :

- un identificateur de l'objet distant à utiliser ;
- une description de la méthode à appeler ;
- les paramètres encodés.

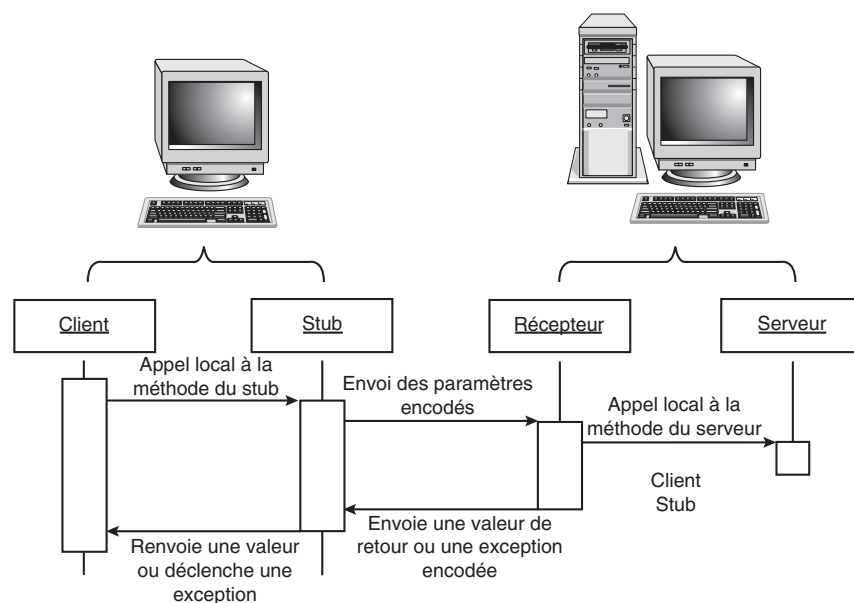
Le stub envoie alors ces informations au serveur. Du côté du serveur, un objet de réception effectue les actions suivantes pour chaque appel de méthode distante :

- Il décode les paramètres encodés.
- Il situe l'objet à appeler.
- Il appelle la méthode spécifiée.
- Il capture et encode la valeur de retour ou l'exception renvoyée par l'appel.
- Il envoie un bloc de données correspondant à la valeur de retour encodée au stub du client.

Le stub client décode la valeur de retour ou l'exception du serveur. Cette valeur devient la valeur de retour de l'appel au stub. Cependant, si la méthode distante a déclenché une exception, le stub la renvoie à l'appelant. La Figure 4.4 représente le flux d'informations lors d'une invocation de méthode distante.

Figure 4.4

Encodage des paramètres.



Ce processus est évidemment assez complexe, mais la bonne nouvelle est qu'il est entièrement automatique, et de manière plus générale, complètement transparent pour le programmeur. De plus, les concepteurs des objets Java distants ont fait de leur mieux pour que ceux-ci aient le même comportement que les objets locaux.

La syntaxe pour un appel à une méthode distante est la même que pour une méthode locale. Si `centralWarehouse` référence un objet de stub pour un objet de l'entrepôt central sur une machine distante, et si la méthode à invoquer est `getQuantity`, l'appel ressemblera typiquement à ceci :

```
int q = centralWarehouse.getQuantity("SuperAspi 100 Aspirateur");
```

Le code du client se sert toujours de variables d'objet dont le type est une interface, dans le but d'accéder à des objets distants :

```
interface Warehouse
{
    int getQuantity(String description)
        throws RemoteException;
    Product getProduct(Customer cust)
        Throws RemoteException;
    . . .
}
```

Une déclaration d'objet pour une variable implémentant une interface serait :

```
Warehouse centralWarehouse = . . .;
```

Naturellement, les interfaces sont des entités abstraites qui se contentent d'indiquer quelles méthodes peuvent être appelées, avec les signatures correspondantes. Les variables dont le type est une interface doivent toujours se trouver à l'intérieur d'un objet réel du même type. Lorsque vous appelez des méthodes distantes, la variable d'objet fait référence à un *objet de stub*. Le programme client ne connaît en fait pas exactement le type de ces objets. Les classes de stub et les objets associés sont créés automatiquement.

Bien que les concepteurs aient fait un très bon travail pour cacher la plupart des détails de l'invocation de méthodes distantes au programmeur, un certain nombre de techniques restent à maîtriser.

INFO

Les objets distants sont récupérés automatiquement par le ramasse-miettes, exactement comme les objets locaux. Cependant, le récupérateur actuel compte des références et ne peut pas détecter des cycles d'objets non référencés. Ces cycles doivent être explicitement cassés par le programmeur avant que les objets distants ne puissent être réclamés.

Charger des classes dynamiquement

Lorsque vous passez un objet distant à un autre programme, soit comme paramètre, soit comme valeur de retour, ce programme doit posséder le fichier de classe de cet objet. Envisageons par exemple une méthode ayant le type de retour `Product`. Bien entendu, le programme client a besoin du fichier de classe `Product.class` pour se compiler. Supposons maintenant que le serveur construise et renvoie un objet `Book` et que `Book` soit un sous-type de `Product`. Le client peut ne jamais avoir vu la classe `Book` et n'avoir aucune idée de l'endroit où trouver le fichier de classe `Book.class`.

230 Au cœur de Java 2 - Fonctions avancées

Un chargeur de classe est donc nécessaire pour charger les classes requises du serveur. Ce procédé est analogue au chargement des classes par des applets qui fonctionnent dans un navigateur.

Lorsqu'un programme charge un nouveau code à partir d'un autre emplacement sur le réseau, il se pose un problème de sécurité. Pour cette raison, vous devez passer par un *gestionnaire de sécurité* pour des applications de client RMI. Il s'agit d'un mécanisme sûr qui protège le programme des virus qui pourraient se trouver dans le code du stub. Pour des applications spécialisées, les programmeurs peuvent fournir leur propre chargeur de classes et leur propre gestionnaire de sécurité, mais ceux qui sont fournis par le système RMI sont suffisants pour un usage classique. Reportez-vous au Chapitre 9 pour plus d'informations sur les chargeurs de classes et les gestionnaires de sécurité.

Etablissement d'une invocation de méthode distante

L'exécution du plus simple des programmes d'exemples d'invocation de méthode distante nécessite une préparation bien plus poussée que pour une application autonome ou pour une applet. Vous devez en effet exécuter des programmes à la fois sur le client et sur le serveur. Les informations nécessaires sur les objets doivent être séparées correctement dans des interfaces au niveau du client et dans des implémentations au niveau du serveur. Il existe également un mécanisme particulier pour effectuer des recherches et permettre au client de localiser des objets sur le serveur.

Pour commencer à implémenter notre exemple, nous devons passer par toutes ces étapes intermédiaires. Dans notre premier exemple, nous générons deux objets de type `Product` sur le serveur. Nous exécutons alors un programme sur le client, pour localiser ces objets et leur envoyer des requêtes.

Interfaces et implémentations

Notre programme client doit pouvoir manipuler des objets de serveur, mais il ne dispose pas en réalité de leurs copies. Les objets eux-mêmes résident sur le serveur. Le code du client doit en permanence savoir ce qu'il est en mesure de faire avec ces objets. Leurs caractéristiques sont détaillées dans une interface partagée entre le client et le serveur, interface qui se trouve par conséquent sur chacune des deux machines.

```
interface Product // partagée par le client et le serveur
    extends Remote
{
    String getDescription() throws RemoteException;
}
```

Comme dans cet exemple, *toutes* les interfaces des objets distants doivent étendre l'interface `Remote` définie dans `java.rmi`. Toutes les méthodes de ces interfaces doivent également déclarer qu'elles peuvent déclencher une exception `RemoteException`. Ces déclarations doivent être effectuées parce que les appels de méthodes distantes sont intrinsèquement moins fiables que les appels locaux : il est toujours possible qu'un appel distant échoue. Par exemple, le serveur ou la connexion peuvent être temporairement hors service, ou encore le réseau peut avoir un problème. Votre code de client doit être prêt à faire face à ces possibilités. Pour ces raisons, le langage de programmation Java vous oblige à récupérer les `RemoteException` pour *chaque* appel de méthode distante et à spécifier les actions à effectuer lorsqu'un appel échoue.

Le client accède à un objet de serveur au travers d'un stub qui implémente cette interface.

```
Product p = ...;
// voir ci-après la manière dont le client obtient une référence
// de stub
String d = p.getDescription();
System.out.println(d);
```

Dans la prochaine section, vous verrez comment le client peut obtenir une référence vers ce type d'objet standard.

Puis, au niveau du serveur, vous devez implémenter la classe qui gère réellement les méthodes présentées dans l'interface distante.

```
public class ProductImpl // serveur
    extends UnicastRemoteObject
    implements Product
{
    public ProductImpl(String d)
        throws RemoteException
    {
        descr = d;
    }

    public String getDescription()
        throws RemoteException
    {
        return "Je suis un " + descr + ". Achetez-moi !";
    }

    private String descr;
}
```

INFO

Le constructeur `ProductImpl` est déclaré en vue de déclencher une `RemoteException` car `UnicastRemoteObject` pourrait déclencher cette exception s'il ne parvient pas à se connecter au service de réseau qui effectue le suivi des objets du serveur.

Cette classe possède une seule méthode, `getDescription`, qui peut être appelée à partir du client distant.

Vous pouvez considérer que la classe est un serveur pour les méthodes distantes parce qu'elle étend `UnicastRemoteObject`, qui est une classe concrète de la plate-forme Java et qui rend les objets accessibles à distance.

INFO

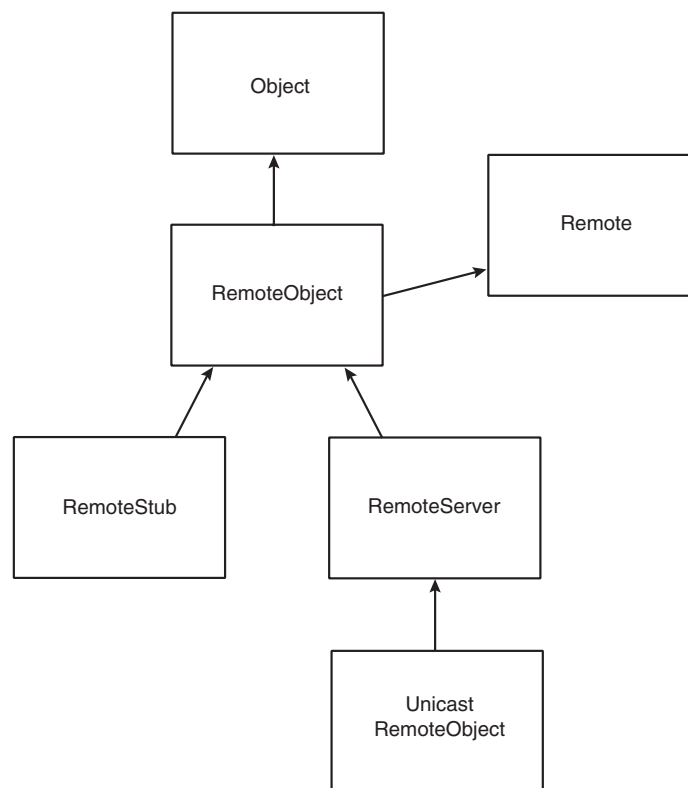
La classe `ProductImpl` n'est pas une classe typique de serveur parce qu'elle effectue très peu de travail. Normalement, les classes de serveur sont destinées à prendre en charge les travaux les plus complexes, ceux que le client ne peut pas effectuer en local. Nous nous servons donc uniquement de l'exemple `Product` pour examiner les mécanismes intervenant dans les appels de méthodes distantes.

232 Au cœur de Java 2 - Fonctions avancées

Toutes les classes de serveur doivent étendre la classe `RemoteServer` de `java.rmi.server`, mais `RemoteServer` est elle-même une classe abstraite qui ne définit que les mécanismes importants pour la communication entre les objets de serveur et leurs stubs distants. La classe `UnicastRemoteObject` fournie avec RMI étend la classe abstraite `RemoteServer`. De plus, cette classe est concrète et vous pouvez l'utiliser sans écrire de code supplémentaire. Le "chemin de moindre coût" pour une classe de serveur est de dériver la classe `UnicastRemoteObject` ; c'est d'ailleurs la stratégie retenue pour toutes les classes de serveur de ce chapitre. La Figure 4.5 montre les relations d'héritage entre ces classes.

Figure 4.5

Classes RMI fondamentales.



Un objet `UnicastRemoteObject` réside sur un serveur. Il doit être actif lorsqu'un service est demandé et il doit être disponible au travers du protocole TCP/IP. Il s'agit de la classe que nous étendons pour toutes les classes de serveur dans ce livre et c'est la seule classe de serveur disponible dans la version courante de RMI. Dans le futur, Sun ou d'autres vendeurs pourront concevoir d'autres classes utilisables par les serveurs pour RMI.

INFO

Par moments, vous ne souhaitez pas utiliser de classe serveur qui étende la classe `UnicastRemoteObject`, par exemple parce qu'elle étend déjà une autre classe. Dans ce cas, il convient d'instancier manuellement les objets du

serveur et de les transmettre à la méthode statique `exportObject`. Au lieu d'étendre `UnicastRemoteObject`, appelez

```
UnicastRemoteObject.exportObject(this, 0);
```

dans le constructeur de l'objet de serveur. Le deuxième paramètre est 0, ce qui indique que tout port adapté peut être utilisé pour écouter les connexions du client.

Lorsque vous vous servirez de RMI (ou de n'importe quel mécanisme d'objets distribués), vous devrez commencer par maîtriser un nombre déconcertant de classes. Dans ce chapitre, nous avons recours à une convention de noms uniforme pour tous nos exemples. Nous espérons que cette convention vous permettra d'identifier plus facilement le but de chaque classe (voir Tableau 4.1).

Tableau 4.1 : Convention de noms pour les classes RMI

pas de suffixe (ex : <code>Product</code>)	Une interface distante
suffixe <code>Impl</code> (ex : <code>ProductImpl</code>)	Une classe de serveur qui implémente cette interface
suffixe <code>Server</code> (ex : <code>ProductServer</code>)	Un serveur qui crée des objets de serveur
suffixe <code>Client</code> (ex : <code>ProductClient</code>)	Un client qui appelle des méthodes distantes
suffixe <code>Stub</code> (ex : <code>ProductImpl_Stub</code>)	Une classe de stub générée automatiquement par le programme <code>rmic</code>
suffixe <code>Skel</code> (ex : <code>ProductImpl_Skel</code>)	Une classe de squelette générée automatiquement par le programme <code>rmic</code> et nécessaire pour le SDK 1.1

Génération de classe de stub

Depuis le JDK 5.0, toutes les classes de stub sont générées automatiquement, grâce au mécanisme de proxy expliqué au Chapitre 6 du Volume 1. Toutefois, avant cela, leur génération se faisait manuellement à l'aide de l'outil `rmic`, comme dans l'exemple suivant :

```
rmic -v1.2 ProductImpl
```

Cet appel à `rmic` génère un fichier de classe appelé `ProductImpl_Stub.class`. Si votre classe se trouve dans un ensemble de programmes, vous devez appeler `rmic` avec le nom complet de cet ensemble.

Si vous continuez à utiliser le JDK 1.1, il faudra appeler :

```
rmic -v1.1 ProductImpl
```

Dans ce cas, deux fichiers sont générés : le fichier de stub et un second fichier de classe appelé `ProductImpl_Skel.class`.

INFO

N'oubliez pas de compiler votre fichier source avec `javac` avant d'exécuter `rmic`. Si vous générez des stubs pour un ensemble de programmes, vous devez fournir à `rmic` le nom complet de cet ensemble.

234 Au cœur de Java 2 - Fonctions avancées

Localiser des objets de serveur

Pour accéder à un objet distant existant sur un serveur, le client a besoin d'un objet de stub local. Comment le client peut-il demander un tel stub ? La méthode la plus courante consiste à appeler une méthode distante d'un autre objet de serveur et à obtenir un objet de stub comme valeur de retour. Cette technique pose cependant un petit problème. Le *premier* objet de serveur doit être localisé d'une autre manière. La bibliothèque RMI de Sun fournit un service pour localiser le premier objet de serveur, appelé *bootstrap registry service*, ou service de base de registres de démarrage.

Un programme de serveur peut alors enregistrer des objets avec ce service, et le client obtient des stubs pour ces objets. Vous pouvez enregistrer un objet de serveur en fournissant au service de base de registres une référence vers cet objet et un *nom*. Ce nom doit correspondre à une chaîne de caractères unique.

```
// serveur
ProductImpl p1 = new ProductImpl("grille-pain Moulinex");
Context namingContext = new InitialContext();
namingContext.bind("rmi:grille-pain", p1);
```

Le code du client obtient un stub pour accéder à cet objet de serveur en spécifiant le nom du serveur et le nom de l'objet de la manière suivante :

```
// client
Product p
    = (Product) namingContext.lookup("rmi://votre_serveur.com/grille-pain");
```

Les URL de RMI commencent par `rmi:/` et sont suivies par un serveur, un numéro de port optionnel, un autre slash et le nom de l'objet distant. En voici un autre exemple :

```
rmi://hôte_local:99/entrepôt_central
```

Par défaut, le serveur est `hôte_local` et le numéro de port vaut 1099.

INFO

Comme il est assez difficile de conserver des noms uniques dans une base de registres globale, il est déconseillé d'utiliser cette technique comme méthode générale pour localiser des objets sur un serveur. Il vaut donc mieux garder peu de noms dans le service de base de registres, et créer des objets qui pourront localiser d'autres objets pour vous. Dans notre exemple, nous transgressons temporairement cette règle et nous enregistrons des objets triviaux pour vous montrer le mécanisme d'enregistrement et de localisation des objets.

Pour des raisons de sécurité, une application peut lier, délier ou relier des références d'objets de base de registres uniquement dans le cas où cette application est exécutée sur le même ordinateur que le service de base de registres. Cela empêche des clients hostiles de modifier les informations de la base de registres. Cependant, n'importe quel client peut rechercher ces objets.

Le service de nommage RMI est intégré dans le service JNDI (*Java Naming and Directory Information*). Dans le JDK 1.3 et versions antérieures, utilisez un service de nommage RMI indépendant, comme ceci :

```
Naming.bind("grille-pain", p1); // sur le serveur
Product p = (Product) Naming.lookup("rmi://votre_serveur.com/grille-pain");
```

Les Exemples 4.1 à 4.3 fournissent le code d'un programme de serveur complet qui enregistre deux objets `Product` sous les noms grille-pain et micro-ondes.

ASTUCE

Si vous comparez notre serveur avec les exemples de serveur de la documentation d'apprentissage de Sun, vous verrez que nous n'installons pas de gestionnaire de sécurité sur ce serveur. Contrairement aux instructions de la documentation, un gestionnaire de sécurité n'est pas nécessaire aux serveurs RMI. Il vous faudra un gestionnaire de sécurité si le client envoie au serveur des objets qui appartiennent à des classes inconnues du serveur. Toutefois, dans notre service, le client n'envoie que des paramètres `String`. Il est généralement conseillé de limiter le chargement des classes dynamiques sur les serveurs.

Exemple 4.1 : *ProductServer.java*

```
import java.rmi.*;
import java.rmi.server.*;
import javax.naming.*;

/**
 * Ce programme de serveur instancie deux objets distants,
 * les enregistre auprès du service de nom et attend les
 * clients pour invoquer les méthodes sur les objets distants.
 */
public class ProductServer
{
    public static void main(String args[])
    {
        try
        {
            System.out.println
                ("Construction des implémentations du serveur...");

            ProductImpl p1
                = new ProductImpl("grille-pain Moulinex");
            ProductImpl p2
                = new ProductImpl("micro-ondes Philips");

            System.out.println
                ("Liaison des implémentations du serveur à la base de registres...");
            Context namingContext = new InitialContext();
            namingContext.bind("rmi:grille-pain", p1);
            namingContext.bind("rmi:micro-ondes", p2);

            System.out.println
                ("Attente des invocations des clients...");
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

236 *Au cœur de Java 2 - Fonctions avancées*

Exemple 4.2 : *ProductImpl.java*

```
import java.rmi.*;
import java.rmi.server.*;

/**
 * Il s'agit d'une classe d'implémentation pour les objets
 * du produit distant.
 */
public class ProductImpl
    extends UnicastRemoteObject
    implements Product
{
    /**
     * Construit une implémentation de produit
     * @param n le nom du produit
     */
    public ProductImpl(String n) throws RemoteException
    {
        name = n;
    }

    public String getDescription() throws RemoteException
    {
        return "Je suis un " + name + ". Achetez-moi!";
    }

    private String name;
}
```

Exemple 4.3 : *Product.java*

```
import java.rmi.*;

/**
 * L'interface des objets de produits distants.
 */
public interface Product extends Remote
{
    /**
     * Récupère la description de ce produit.
     * @return la description du produit
     */
    String getDescription() throws RemoteException;
}
```

Démarrer le serveur

Notre programme de serveur n'est pas encore tout à fait prêt à fonctionner. Comme il se sert de la base de registres RMI de démarrage, ce service doit être disponible. Pour lancer la base de registres RMI sous UNIX, exécutez l'instruction :

```
rmiregistry &
```

Sous Windows, exécutez l'instruction suivante :

```
start rmiregistry
```

sous DOS ou à partir de la boîte de dialogue Exécuter. La commande start est une commande de Windows qui exécute un programme dans une nouvelle fenêtre.

Vous êtes maintenant prêt à démarrer le serveur. Sous Windows, lancez la commande :

```
start java ProductServer
```

Sous Unix, il faudra utiliser la commande suivante :

```
java ProductServer &
```

Si vous vous contentez de lancer le serveur avec :

```
java ProductServer
```

le programme ne se terminera jamais de manière normale. Cela peut sembler étrange, après tout ce programme se contente de créer deux objets et de les enregistrer. En fait, la fonction `main` prend fin immédiatement après la fin de l'enregistrement, comme vous pouvez vous y attendre. Mais si vous créez un objet à partir d'une classe qui étend `UnicastRemoteObject`, un nouveau thread est lancé pour garder le programme activé indéfiniment. Par conséquent, le programme ne se termine pas pour permettre aux clients de s'y connecter.

ASTUCE

La version Windows du JDK contient une commande, `javaw`, qui démarre l'interpréteur de code dans un processus Windows indépendant. Certaines sources recommandent d'utiliser `javaw` et non `start java` pour démarrer une session java en tâche de fond pour RMI. Ce n'est pas vraiment une bonne idée, pour les deux raisons suivantes.

Windows ne possède aucun outil pour fermer une tâche de fond `javaw`, parce qu'il ne l'affiche pas dans la liste de tâches. Il apparaît donc que vous avez besoin d'arrêter le service de base de registres de démarrage et de le redémarrer, lorsque vous modifiez le stub d'une classe enregistrée. Pour arrêter un processus que vous avez lancé avec la commande `start`, il vous suffit de cliquer sur la fenêtre et d'appuyer sur `Ctrl+C`.

Il existe une autre raison importante pour utiliser la commande `start`. Lorsque vous lancez un processus de serveur avec `javaw`, les messages envoyés dans les flux de sortie ou dans les flux d'erreur ne sont pas pris en compte. En particulier, ils ne sont affichés *nulle part*. Si vous voulez voir les messages d'erreur ou les messages standard, utilisez la commande `start`. Dans ce cas, les messages d'erreur sont au moins affichés dans la console. Et croyez-nous, vous aurez souvent envie de voir ces messages. Il existe un certain nombre de problèmes qui peuvent se produire lorsque vous faites des essais avec RMI. L'une des erreurs les plus courantes se produit lorsque vous oubliez de lancer `rmic`. Dans ce cas, le serveur réclame des stubs manquants. Si vous vous servez de `javaw`, vous ne verrez pas ces messages d'erreur, et il vous sera très difficile de découvrir la raison pour laquelle le client ne peut pas trouver les objets de serveur.

Listage des objets distants

Avant d'écrire le programme client, vérifions que nous avons correctement enregistré les objets distants. Appelez

```
NamingEnumeration<NameClassPair> e = namingContext.list("rmi:");
```

pour obtenir une énumération de tous les objets de serveur ayant l'URL `rmi:`. L'énumération produit des objets du type `NameClassPair`, une classe d'assistance qui contient à la fois le nom de l'objet lié et le nom de sa classe. Seuls les noms nous intéressent :

```
while (e.hasMore()) System.out.println(e.next().getName());
```

L'Exemple 4.4 contient le programme complet. Le résultat devrait être le suivant :

```
grille-pain  
micro-ondes
```

238 Au cœur de Java 2 - Fonctions avancées

Exemple 4.4 : *ShowBindings.java*

```
import java.rmi.*;
import java.rmi.server.*;
import javax.naming.*;

/**
 * Ce programme présente toutes les liaisons RMI.
 */
public class ShowBindings
{
    public static void main(String[] args)
    {
        try
        {
            Context namingContext = new InitialContext();
            NamingEnumeration<NameClassPair> e = namingContext.list("rmi:");
            while (e.hasMore())
                System.out.println(e.next().getName());
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Du côté client

Nous pouvons maintenant écrire un programme client qui demanderait à chaque nouveau produit enregistré d'afficher sa description.

Les programmes client qui se servent de RMI devraient installer un gestionnaire de sécurité pour contrôler les activités des stubs chargés dynamiquement. `RMISecurityManager` est un bon exemple d'un tel gestionnaire de sécurité. L'instruction suivante permet de l'installer :

```
System.setSecurityManager(new RMISecurityManager());
```

INFO

Si toutes les classes (et notamment les stubs) sont disponibles en local, vous n'avez pas réellement besoin d'un gestionnaire de sécurité. Si vous connaissez tous les fichiers de classes de votre programme au moment de sa conception, il vous suffit de les développer en local. Cependant, vous vous rendrez compte qu'un programme de serveur est souvent amené à évoluer et que de nouvelles classes doivent y être ajoutées au fur et à mesure. Vous pourrez alors tirer profit du chargement dynamique des classes. Chaque fois que vous devrez charger du code à partir d'un nouvel emplacement, vous devrez passer par un gestionnaire de sécurité.

INFO

Les applets possèdent déjà un gestionnaire de sécurité capable de contrôler le chargement des classes de stub. Lorsque vous utilisez RMI dans une applet, vous n'avez pas besoin d'installer un autre gestionnaire de sécurité.

L'Exemple 4.5 fournit le code complet du programme client. Ce client obtient simplement les références de deux objets `Product` dans la base de registres RMI et invoque la méthode `getDescription` de chaque objet.

Exemple 4.5 : *ProductClient.java*

```
import java.rmi.*;
import java.rmi.server.*;
import javax.naming.*;

/**
 * Ce programme montre comment appeler une méthode distante
 * sur deux objets localisés grâce au service de nom.
 */
public class ProductClient
{
    public static void main(String[] args)
    {
        System.setProperty("java.security.policy", "client.policy");
        System.setSecurityManager(new RMISecurityManager());
        String url = "rmi://hôte_local/";
        // utilisez "rmi://votre_serveur.com/"
        // lorsque le serveur se trouve sur la machine distante
        // votre_serveur.com
        try
        {
            Context namingContext = new InitialContext();
            Product c1 = (Product) namingContext.lookup(url + "grille-pain");
            Product c2 = (Product) namingContext.lookup(url + "micro-ondes");
            System.out.println(c1.getDescription());
            System.out.println(c2.getDescription());
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Exécuter le client

Par défaut, `RMISecurityManager` empêche n'importe quelle partie du programme d'établir des connexions réseau. Mais notre programme doit ouvrir des connexions réseau :

- pour atteindre la base de registres RMI ;
- pour contacter les objets du serveur.

INFO

Une fois que le programme client est mis en place, il faut également lui fournir une autorisation pour charger ses classes de stub. Nous reviendrons sur ce problème un peu plus loin, lorsque nous aborderons les mises en œuvre.

Pour permettre à un client de se connecter à une base de registres RMI et aux objets du serveur, vous devrez fournir un *fichier de règles de sécurité*. Nous approfondirons les fichiers de règles de sécurité

240 *Au cœur de Java 2 - Fonctions avancées*

dans le Chapitre 9. Pour l'instant, vous pouvez vous contenter d'utiliser et de modifier les exemples que nous vous fournissons. Voici un fichier de règles de sécurité qui permet à une application d'effectuer n'importe quelle connexion réseau vers un port dont le numéro est supérieur ou égal à 1024. Le port RMI par défaut est 1099, et les objets du serveur se servent également de ports supérieurs à 1024.

```
grant
{
    permission java.net.SocketPermission
        " *:1024-65535", "connect";
};
```

INFO

Plusieurs objets serveur disposés sur le même serveur sont en mesure de partager un port mais, en cas d'appel distant alors que le port est occupé, un autre port est automatiquement ouvert. Ainsi, attendez-vous à utiliser moins de ports que les objets distants sur le serveur.

Dans le programme client, nous demandons au gestionnaire de sécurité de lire des règles, en définissant la propriété `java.security.policy` au nom du fichier (dans nos exemples, nous utilisons le fichier `client.policy`).

```
System.setProperty("java.security.policy", "client.policy");
```

Vous pouvez également régler les propriétés système à l'aide d'une ligne de commande :

```
java -Djava.security.policy=client.policy ProductClient
```

INFO

Avec le JDK 1.1, les fichiers de règles de sécurité n'étaient pas nécessaires pour les clients RMI. Vous n'en avez pas besoin non plus pour les applets, à condition que le serveur et la base de registre RMI soient tous deux situés sur l'hôte qui répond au code de l'applet.

Si la base de registres RMI et le serveur fonctionnent, vous pouvez exécuter le programme client. Autrement, si vous préférez repartir de zéro, arrêtez le serveur et la base de registres RMI, puis suivez ces étapes :

1. Compilez les fichiers source des classes de l'interface, de l'implémentation, du client et du serveur.

```
javac Product*.java
```
2. Si vous utilisez le JDK 1.4 ou version antérieure, lancez `rmic` sur la classe d'implémentation :

```
rmic -v1.2 ProductImpl
```
3. Démarrez la base de registres RMI :

```
start rmiregistry
```

ou

```
rmiregistry &
```
4. Lancez le serveur :

```
start java ProductServer
```

ou

```
java ProductServer &
```

5. Exécutez le programme client :

```
java ProductClient
```

Assurez-vous que le fichier `client.policy` se trouve bien dans le répertoire en cours.

Le programme affiche simplement :

```
Je suis un grille-pain Moulinex. Achetez-moi !
Je suis un micro-ondes Philips. Achetez-moi !
```

Ce résultat n'est pas très impressionnant, mais n'oubliez pas tout ce qui se passe en arrière-plan lorsque Java exécute l'appel à la méthode `getDescription`. Le programme client possède une référence vers un objet stub qui est obtenu à partir d'une méthode `lookup`. Cet objet appelle la méthode `getDescription`, qui envoie un message réseau à un objet récepteur, au niveau du serveur. Ce récepteur invoque à son tour la méthode `getDescription` sur l'objet `ProductImpl` localisé sur le serveur. Cette méthode calcule une chaîne de caractères, que le récepteur envoie sur le réseau. Le stub reçoit alors cette chaîne et la renvoie comme résultat (voir Figure 4.6).

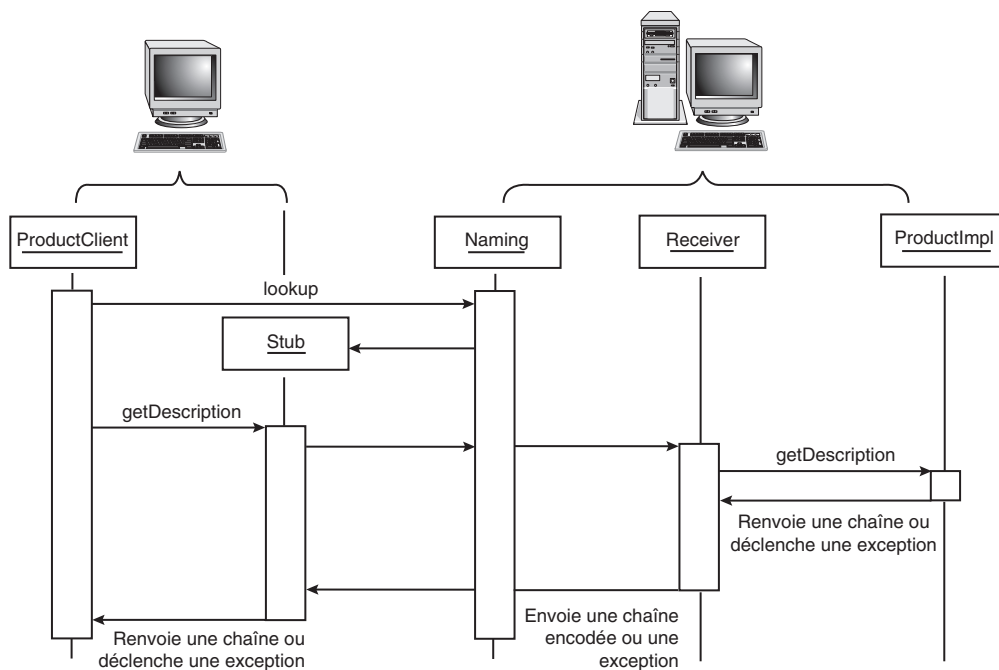


Figure 4.6

Un appel à la méthode distante `getDescription`.

javax.naming.InitialContext 1.3

- `InitialContext()`

Construit un contexte de nom pouvant être utilisé pour accéder au registre des RMI.

242 *Au cœur de Java 2 - Fonctions avancées*

javax.naming.Context 1.3

- `static Object lookup(String name)`
Renvoie l'objet pour le nom donné. Déclenche une `NamingException` si le nom n'est pas lié actuellement.
- `static void bind(String name, Object obj)`
Lie le nom à l'objet `obj`. Déclenche une `NameAlreadyBoundException` si l'objet est déjà lié.
- `static void unbind(String name)`
Délie le nom. Vous pouvez délier un nom qui n'existe pas.
- `static void rebind(String name, Object obj)`
Lie `name` à l'objet `obj`. Remplace toute liaison existante.
- `NamingEnumeration<NameClassPair> list(String name)`
Renvoie une énumération répertoriant tous les objets liés concordants. Pour énumérer tous les objets RMI, utilisez `"rmi:"`.

javax.naming.NamingEnumeration<T> 1.3

- `boolean hasMore()`
Renvoie `true` si cette énumération possède d'autres éléments.
- `T next()`
Renvoie l'élément suivant de cette énumération.

javax.naming.NameClassPair 1.3

- `String getName()`
Récupère le nom de l'objet nommé.
- `String getClassName()`
Récupère le nom de la classe à laquelle appartient l'objet nommé.

java.rmi.Naming 1.1

- `static Remote lookup(String url)`
Renvoie l'objet distant correspondant à l'URL. Déclenche une `NotBoundException` si le nom n'est pas lié actuellement.
- `static void bind(String name, Remote obj)`
Lie `name` à l'objet distant `obj`. Déclenche une `AlreadyBoundException` si l'objet est déjà lié.
- `static void unbind(String name)`
Délie le nom. Déclenche une exception `NotBound` si le nom n'est pas lié actuellement.
- `static void rebind(String name, Remote obj)`
Lie `name` à l'objet distant `obj`. Remplace n'importe quel lien existant.
- `static String[] list(String url)`
Renvoie un tableau de chaînes contenant les URL de la base de registres correspondant à l'URL spécifiée. Le tableau contient un aperçu des noms présents dans la base de registres.

Préparer la mise en œuvre

La mise en œuvre d'une application utilisant RMI peut parfois être assez délicate parce qu'un certain nombre de choses peuvent mal se passer et parce que les messages d'erreur obtenus sont très peu explicites. Il nous semble qu'il est donc réellement intéressant de préparer la mise en œuvre en local. Dans cette étape de préparation, il convient de séparer les fichiers de classes en trois sous-répertoires :

```
server
download
client
```

Le répertoire `server` contient tous les fichiers nécessaires à l'*exécution du serveur*. Vous placerez plus tard les fichiers de ce répertoire sur la machine hébergeant le serveur. Dans notre exemple, le répertoire `server` contient les fichiers suivants :

```
server/
  ProductServer.class
  ProductImpl.class
  Product.class
```

ATTENTION

Si vous utilisez le JDK 1.4 ou version antérieure, ajoutez les classes de stub (comme `ProductImpl_Stub.class`) dans le répertoire `server`. Elles sont en effet nécessaires lorsque le serveur enregistre l'objet d'implémentation. Contrairement à ce que vous pourriez penser, le serveur ne les place pas dans le répertoire `download`, même si vous prenez la peine de définir un répertoire source.

Le répertoire `client` contient les fichiers nécessaires au *démarrage du client*. Ces fichiers sont :

```
client/
  ProductClient.class
  Product.class
  client.policy
```

Vous devrez par la suite placer ces fichiers sur l'ordinateur client.

Enfin, le répertoire `download` contient les fichiers de classe nécessaires au registre RMI, au client et au serveur, ainsi que les classes dont ils dépendent. Dans notre exemple, le répertoire `download` ressemble à ceci :

```
download/
  Product.class
```

Si vos clients exécutent le JDK 1.4 ou version antérieure, fournissez également les classes de stub (comme `ProductImpl_Stub.class`). Si le serveur exécute le JDK 1.1, fournissez les classes squelette (comme `ProductImpl_Skel.class`). Vous placerez ensuite ces fichiers sur un serveur Web.

ATTENTION

N'oubliez pas que *trois* machines virtuelles utilisent le répertoire `download` : le client, le serveur et le registre RMI. Ce dernier a besoin des fichiers de classe pour les interfaces distantes des objets qu'il lie. Le client et le serveur ont besoin des fichiers de classe pour les paramètres et les valeurs de retour.

244 Au cœur de Java 2 - Fonctions avancées

Tous vos fichiers de classes sont maintenant répartis correctement et vous pouvez vérifier qu'ils peuvent bien être chargés.

Vous devez exécuter un serveur Web pour donner les fichiers de classe sur votre ordinateur. Si vous n'avez pas installé de serveur, téléchargez Tomcat à l'adresse <http://jakarta.apache.org/tomcat> et installez-le. Créez un répertoire `tomcat/webapps/download`, où *tomcat* correspond au répertoire de base de votre installation Tomcat. Créez un sous-répertoire `tomcat/webapps/download/WEB-INF` et placez-y le fichier `Web.xml` minimal suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/Web-app_2_3.dtd">
<Web-app>
</Web-app>
```

Copiez ensuite les fichiers de classe du répertoire intermédiaire `download` dans le répertoire `tomcat/webapps/download`.

Puis éditez le fichier `client.policy`. Il doit fournir au client les permissions suivantes :

- se connecter aux ports supérieurs à 1024 pour atteindre la base de registres RMI et les implémentations du serveur ;
- se connecter au port 80 (port HTTP standard) pour charger les fichiers de classes stub. Vous pouvez omettre cette autorisation si vous utilisez Tomcat comme serveur Web (il utilise le port 8080 par défaut).

Modifiez ce fichier en fonction du modèle suivant :

```
grant
{
    permission java.net.SocketPermission
        "*:1024-65535", "connect";
    permission java.net.SocketPermission
        "*:80", "connect";
};
```

Vous êtes maintenant prêt à tester votre installation.

1. Lancez le serveur Web.
2. Pointez un navigateur Web vers l'URL de téléchargement (<http://localhost:8080/download/Product.class> pour Tomcat) afin de vérifier que le serveur Web fonctionne.
3. Ouvrez un nouveau terminal. Vérifiez que le chemin d'accès aux classes est correct. Placez-vous dans un répertoire qui ne contient aucun fichier de classe, puis exécutez la base de registres RMI.

ATTENTION

Si vous souhaitez uniquement tester votre programme et placer les fichiers de classes du client, du serveur et des stubs dans un seul répertoire, vous pouvez lancer la base de registres RMI dans ce répertoire. Cependant, pour la mise en œuvre, vérifiez bien que vous lancez la base de registres dans un terminal ne possédant aucun chemin d'accès aux classes et dans un répertoire ne contenant aucun fichier de classe. Dans le cas contraire, la base de registres RMI trouvera des fichiers de classes parasites qui la gêneront lorsqu'elle tentera de charger des classes de stub à partir d'un autre répertoire. Pour connaître la raison de ce comportement, consultez <http://java.sun.com/products/j2se/5.0/docs/guide/rmi/codebase.html>.

Le manque de consistance des bases de registres RMI constitue une source d'erreurs fréquentes lors d'une mise en œuvre RMI. Le meilleur moyen de vous protéger contre ce genre de problème est de vous assurer que la base de registres RMI ne peut trouver aucune classe.

4. Lancez un nouveau terminal et placez-vous dans le répertoire `server`. Démarrez le serveur en fournissant une URL correspondant au répertoire `download` à la propriété `java.rmi.server.codebase` :

```
java
-Djava.rmi.server.codebase=http://localhost:8080/download/
ProductServer &
```

ATTENTION

Il est très important de bien vérifier que l'URL se termine par un slash (/).

5. Placez-vous dans le répertoire `client`. Assurez-vous que le fichier `client.policy` se trouve dans ce répertoire. Démarrez le programme client :

```
java -Djava.security.policy=client.policy ProductClient
```

Si le client et le serveur ont pu être démarrés sans problème, vous êtes prêt pour passer à la prochaine étape et pour mettre en œuvre les classes dans un client et un serveur indépendants. Dans le cas contraire, vous devrez avoir recours à quelques astuces.

ASTUCE

Si vous ne voulez pas installer de serveur Web en local, vous pouvez vous servir de l'URL d'un fichier pour tester le chargement des classes, mais la configuration devient légèrement plus complexe. Ajoutez la ligne suivante :

```
permission java.io.FilePermission
"downloadDirectory", "read";
```

à votre fichier de règles de sécurité client. Ici, `downloadDirectory` correspond au chemin d'accès complet du répertoire `download`, entre guillemets et se terminant obligatoirement par un signe moins (pour spécifier tous les fichiers de ce répertoire et de ses sous-répertoires). Par exemple :

```
permission java.io.FilePermission
"/home/test/download/-", "read";
```

Avec les noms de fichiers Windows, vous devez doubler les barres obliques. Par exemple :

```
permission java.io.FilePermission
"c:\\home\\test\\download\\-", "read";
```

Lancez la base de registres RMI, puis le serveur avec :

```
tart java
-Djava.rmi.server.codebase=file:/c:/home/test/download/
ProductServer
```

ou

```
java -Djava.rmi.server.codebase=file://home/test/download/
ProductServer &
```

N'oubliez pas d'ajouter un slash à la fin de l'URL. Vous pouvez ensuite démarrer le client.

246 *Au cœur de Java 2 - Fonctions avancées*

Mise en œuvre du programme

Maintenant que vous avez testé la mise en œuvre de votre programme, vous pouvez le distribuer sur les clients et les serveurs finaux.

Déplacez le répertoire `download` vers le serveur Web. Vérifiez que vous vous servez de cette URL lorsque vous démarrez le serveur. Mettez les classes du répertoire `server` dans votre serveur et lancez la base de registres RMI et le serveur.

Votre configuration de serveur est maintenant terminée. Mais vous devez encore apporter deux modifications au client. Tout d'abord, ouvrez le fichier de règles de sécurité et remplacez `*` par le nom du serveur :

```
grant
{
    permission java.net.SocketPermission
        "yourserver.com:1024-65535", "connect";
    permission java.net.SocketPermission
        "yourserver.com:80", "connect";
};
```

Pour terminer, remplacez `localhost` dans l'URL RMI du programme client par le serveur réel.

```
String url = "rmi://yourserver.com/";
Product c1 = (Product) namingContext.lookup(url + "toaster");
. . .
```

Recompilez ensuite le client et testez-le. Si tout fonctionne correctement, vous pouvez être fier de vous ! Sinon la liste suivante pourra peut-être vous aider. Elle répertorie une série de vérifications à effectuer pour supprimer un certain nombre de problèmes courants.

ASTUCE

En pratique, vous voudrez sûrement éviter de coder les URL RMI directement dans votre programme. Une meilleure approche consiste à les enregistrer dans un fichier de propriétés. Nous utilisons cette technique dans l'Exemple 4.13.

Vérifications d'une mise en œuvre RMI

Une mise en œuvre RMI est toujours très délicate. De deux choses l'une : soit tout se passe bien immédiatement, soit vous obtenez un message d'erreur incompréhensible. D'après les échanges de la liste de discussion sur <http://archives.java.sun.com/archives/rmi-users.html>, un grand nombre de programmeurs se heurtent à des difficultés au départ. Si c'est également votre cas, cela vous aidera peut-être de vérifier les points suivants. Nous avons réussi à obtenir une erreur pour chacun d'entre eux au moins une fois lors de la phase de test.

- Avez-vous mis les fichiers de classes stub dans le répertoire du serveur ?
- Avez-vous mis les fichiers de classes d'interface dans les répertoires `client` et `download` ?
- Avez-vous inclus les classes dépendantes pour chaque classe ? Par exemple, dans la prochaine section, vous découvrirez l'interface `Warehouse` qui possède une méthode avec un paramètre du type `Customer`. N'oubliez pas d'inclure `Customer.class` dès que vous mettez en œuvre `Warehouse.class`.

- Lorsque vous avez démarré `rmiregistry`, le `CLASSPATH` était-il vide ? Le répertoire courant contenait-il des fichiers de classes ?
- Vous servez-vous d'un fichier de règles de sécurité pour démarrer le client ? Ce fichier contient-il les noms corrects du serveur (ou `*` pour se connecter à n'importe quel hôte) ?
- Si vous vous servez d'une URL `file:` pour effectuer vos tests, avez-vous spécifié le bon nom dans le fichier de règles de sécurité ? Se termine-t-il par `\\` ou `/` ? N'avez-vous pas oublié d'utiliser `\\` pour les noms sous Windows ?
- Votre URL de base se termine-t-elle par un slash ?

Pour terminer, remarquez que la base de registres RMI garde une trace de tous les fichiers de classes qu'elle a rencontrés. Si vous supprimez souvent des fichiers de classes pour déterminer lesquels sont réellement nécessaires, n'oubliez pas de redémarrer `rmiregistry` chaque fois.

Consignation de l'activité des RMI

L'implémentation des RMI de Sun permet de produire des messages de consignation à l'aide de l'API de consignation standard de Java (voir Chapitre 11 du Volume 1 pour en savoir plus sur la consignation).

Pour étudier la consignation, créez un fichier `logging.properties` ayant le contenu suivant :

```
handlers=java.util.logging.ConsoleHandler
.level=FINEST
java.util.logging.ConsoleHandler.level=FINEST
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter
```

Vous pouvez affiner les réglages en définissant des niveaux individuels pour chaque consignation au lieu d'utiliser le niveau général (FINEST). Le Tableau 4.2 énumère les systèmes de consignation des RMI.

Tableau 4.2 : Consignation des RMI

Nom	Activité consignée
<code>sun.rmi.server.call</code>	Appels distants côté serveur
<code>sun.rmi.server.ref</code>	Références distantes côté serveur
<code>sun.rmi.client.call</code>	Appels distants côté client
<code>sun.rmi.client.ref</code>	Références distantes côté client
<code>sun.rmi.dgc</code>	Ramasse-miettes distribué
<code>sun.rmi.loader</code>	<code>RMICClassLoader</code>
<code>sun.rmi.transport.misc</code>	Couche du transport
<code>sun.rmi.transport.tcp</code>	Liaison et connexion TCP
<code>sun.rmi.transport.proxy</code>	Tunnelisation HTTP

248 Au cœur de Java 2 - Fonctions avancées

Lancez le registre RMI à l'aide de l'option

```
-J-Djava.util.logging.config.file=répertoire/logging.properties
```

Lancez le client et le serveur avec

```
-Djava.util.logging.config.file=répertoire/logging.properties
```

Il est conseillé de lancer le registre RMI, le client et le serveur dans des fenêtres différentes. Vous pouvez aussi consigner les messages dans un fichier (voir Chapitre 11 du Volume 1).

Voici un exemple de message consigné montrant un problème de chargement de classe : le registre RMI ne trouve pas la classe `Product`, celle-ci devant construire un proxy dynamique.

```
Aug 15, 2004 10:44:07 AM sun.rmi.server.LoaderHandler loadProxyClass
FINE: RMI TCP Connection(1)-127.0.0.1: interfaces = [java.rmi.Remote, Product],
codebase = "http://localhost:8080/download/"
Aug 15, 2004 10:44:07 AM sun.rmi.server.LoaderHandler loadProxyClass
FINER: RMI TCP Connection(1)-127.0.0.1: (thread context class loader:
java.net.URLClassLoader@6ca1c)
Aug 15, 2004 10:44:07 AM sun.rmi.server.LoaderHandler loadProxyClass
FINE: RMI TCP Connection(1)-127.0.0.1: proxy class resolution failed
java.lang.ClassNotFoundException: Product
```

Passage de paramètres aux méthodes distantes

Vous serez souvent amené à passer des paramètres aux objets distants. Cette section passe en revue quelques techniques sur ce sujet, ainsi que certains défauts connus.

Passer des objets locaux

Lorsqu'un objet distant est passé d'un serveur à un client, le client reçoit un stub. A partir de ce stub, il peut manipuler l'objet du serveur en invoquant des méthodes distantes. Cependant, l'objet lui-même reste sur le serveur. Il est également possible de passer et de renvoyer n'importe quel objet en utilisant un appel de méthode distante, et pas seulement ceux qui implémentent l'interface `Remote`. Par exemple, la méthode `getDescription` de la section précédente renvoyait un objet `String`. Cette chaîne a été créée sur le serveur et devait être amenée sur le client. Comme `String` n'implémente pas l'interface `Remote`, le serveur ne peut pas renvoyer une chaîne de stub. Il doit donc se contenter d'obtenir une *copie* de la chaîne. Puis, après l'appel, le client possède sa propre version de l'objet `String`, sur lequel il peut travailler. Cela signifie que cette chaîne n'a plus besoin d'aucune connexion vers un objet du serveur.

Lorsqu'un objet qui n'est pas un objet distant doit être transporté d'une machine virtuelle Java à une autre, cet objet est copié et la copie est envoyée au travers d'une connexion réseau. Cette technique est très différente d'un passage de paramètres en local. Lorsque vous passez un objet à une méthode locale, ou que vous le renvoyez comme résultat d'une méthode, seules des *références* vers cet objet sont échangées. Ces références sont en fait des adresses mémoire correspondant aux objets de la machine virtuelle Java locale, et elles n'ont aucune signification pour une autre machine virtuelle Java.

Il n'est pas très difficile d'imaginer comment une copie d'une chaîne peut être transportée sur un réseau. Le mécanisme RMI peut également effectuer des copies d'objets plus complexes, à partir du moment où ces objets peuvent être mis en série. RMI se sert du mécanisme de sérialisation décrit au

Chapitre 12 du Volume 1 pour envoyer des objets au travers d'une connexion réseau. Cela signifie que seules les informations des classes implémentant l'interface `Serializable` peuvent être copiées. Le programme suivant montre le principe des copies de paramètres et des valeurs de retour en action. Ce programme est une simple application permettant à un utilisateur d'acheter un cadeau. Sur le client, l'utilisateur exécute un programme qui collecte des informations sur le destinataire du cadeau, c'est-à-dire dans notre cas son âge, son sexe et ses loisirs (voir Figure 4.7).

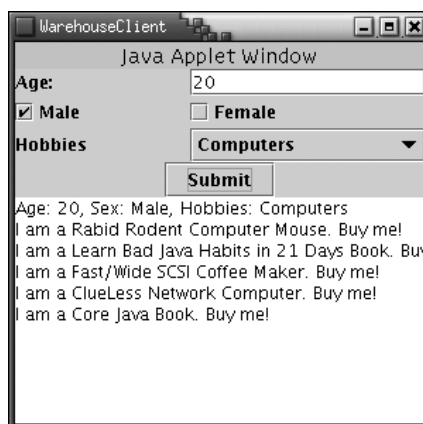
INFO

La Figure 4.7 présente une curieuse bannière indiquant une fenêtre d'applet Java. Ceci est dû à l'exécution du programme avec un gestionnaire de sécurité. Cet avertissement permet de se prémunir des applets de "phishing". Une applet hostile pourrait faire apparaître une fenêtre, demander un mot de passe ou un numéro de carte de crédit, puis renvoyer les informations à son hôte. Pour désactiver l'avertissement, ajoutez les lignes suivantes au fichier `client.policy` :

```
permission java.awt.AWTPermission
    "showWindowWithoutWarningBanner";
```

Figure 4.7

Obtenir des suggestions
de produits à partir
d'un serveur.



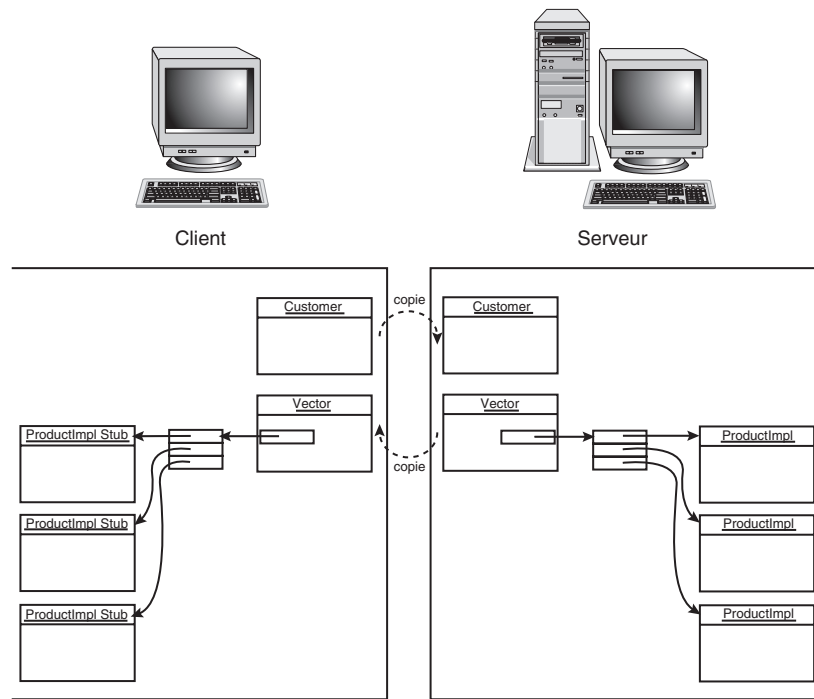
Un objet de type `Customer` est alors envoyé au serveur. Comme `Customer` n'est pas un objet distant, une copie de cet objet est créée sur le serveur. Le serveur renvoie alors une liste de tableau de produits. Elle contient les produits correspondant au profil du client, et notamment un article qui réjouira tout le monde, à savoir une copie du livre *Au cœur de Java*. De même, `ArrayList` n'est pas une classe distante, donc la liste de tableau sera copiée avant d'être renvoyée au client. Comme l'explique le Chapitre 12 du Volume 1, le mécanisme de sérialisation effectue des copies de tous les objets référencés dans un objet copié. Dans notre cas, il effectue aussi une copie de toutes les entrées de la liste de tableau. Nous avons ajouté un petit raffinement : les entrées sont en fait des objets distants `Product`. Par conséquent, le destinataire obtient une copie de la liste de tableau, remplie avec des objets stub correspondant aux produits du serveur (voir Figure 4.8).

Pour résumer, les objets distants sont transférés sur le réseau sous la forme de stubs, et les objets locaux sont copiés. Tout ce processus est entièrement automatique et ne nécessite aucune intervention du programmeur.

250 Au cœur de Java 2 - Fonctions avancées

Figure 4.8

Copier des paramètres locaux et des objets de résultat.



Lorsque votre code appelle une méthode distante, le stub crée un ensemble contenant des copies de toutes les valeurs des paramètres et les envoie au serveur, en utilisant le mécanisme de sérialisation des objets pour encoder les paramètres. Le serveur doit alors les décoder. Naturellement, ce processus peut être relativement lent, tout particulièrement lorsque les paramètres sont assez gros.

Examinons maintenant le programme complet. Tout d'abord, nous avons les interfaces des produits et des entrepôts, comme le montrent les Exemples 4.6 et 4.7.

Exemple 4.6 : *Product.java*

```
import java.rmi.*;

/**
 * L'interface des objets product distants.
 */
public interface Product extends Remote
{
    /**
     * Récupère la description de ce produit.
     * @return la description du produit
     */
    String getDescription() throws RemoteException;
}
```

Exemple 4.7 : *Warehouse.java*

```
import java.rmi.*;
import java.util.*;
```

```
/**
 * L'interface distante d'un entrepôt avec des produits.
 */
public interface Warehouse extends Remote
{
    /**
     * Récupère les produits correspondants à la demande du client.
     * @param c le client demandeur
     * @return une liste de tableau des produits correspondants
     */
    ArrayList<Product> find(Customer c) throws RemoteException;
}
```

L'Exemple 4.8 montre l'implémentation du service de produits. Les produits contiennent une description, une classe d'âge, un genre ciblé (masculin, féminin ou les deux), et le loisir correspondant. Vous remarquerez que cette classe implémente la méthode `getDescription` annoncée dans l'interface `Product` et qu'elle implémente également une autre méthode, `match`, qui ne fait pas partie de cette interface. La méthode `match` représente un exemple de *méthode locale*, c'est-à-dire de méthode qui ne peut être appelée que par le programme local, et non à distance. Comme la méthode `match` est locale, il faut qu'elle ne soit pas préparée pour déclencher une `RemoteException`.

L'Exemple 4.9 fournit le code de la classe `Customer`. Notez une fois encore que `Customer` n'est pas une classe distante, c'est-à-dire qu'aucune de ses méthodes ne peut être exécutée à distance. Cependant, cette classe est sérialisable. Par conséquent, les objets de cette classe peuvent être transportés d'une machine virtuelle à une autre.

Les Exemples 4.10 et 4.11 représentent l'interface et l'implémentation du service d'entrepôt. Comme la classe `ProductImpl`, la classe `WarehouseImpl` possède des méthodes distantes et des méthodes locales. La méthode `add` est locale et est utilisée par le serveur pour ajouter des produits dans le magasin. En revanche, la méthode `find` est une méthode distante utilisée par le client pour trouver des articles dans l'entrepôt.

Pour mettre en évidence le fait que l'objet `Customer` est réellement copié, la méthode `find` de la classe `WarehouseImpl` efface l'objet `Customer` qu'elle reçoit. Lorsque la méthode est terminée, `WarehouseClient` affiche l'objet `Customer` qu'il envoie au serveur. Comme vous pourrez le constater, cet objet n'a pas été modifié. Le serveur n'a effacé que *sa copie*. Dans ce cas, l'opération `clear` ne réalise aucune tâche fondamentalement nécessaire, mais elle permet de démontrer que les objets locaux sont bien copiés lorsqu'ils sont passés en paramètres.

Il deviendrait possible pour plusieurs stubs de clients d'effectuer des appels simultanés à un objet du serveur, même si l'une des méthodes venait à modifier l'état du serveur. Dans l'Exemple 4.11, nous utilisons un `ReadWriteLock` dans la classe `WarehouseImpl` parce qu'il est possible que la méthode locale `add` et la méthode distante `find` soient appelées simultanément. L'Exemple 4.12 montre le programme de serveur qui crée un objet entrepôt et qui l'enregistre avec le service de démarrage de la base de registres.

INFO

N'oubliez pas que vous devez exécuter la base de registres et le programme du serveur et que ces deux programmes doivent encore être en cours d'exécution lorsque vous lancez le client.

252 Au cœur de Java 2 - Fonctions avancées

L'Exemple 4.13 fournit le code du client. Lorsque l'utilisateur clique sur le bouton "Envoyer", un nouvel objet `Customer` est généré et passé à la méthode distante `find`. Puis, le champ du `Customer` est affiché dans une zone de texte, pour prouver que l'appel à la méthode `reset` du serveur ne l'a pas affecté. Pour terminer, les descriptions des produits renvoyés dans la liste du tableau sont ajoutées à la zone de texte. Vous remarquerez que chaque appel à `getDescription` constitue à nouveau une invocation de méthode distante. Ceci ne serait pas particulièrement adapté dans la pratique, il conviendrait plus naturellement de transférer de petits objets, comme les descriptions de produits, en fonction de leur valeur. Nous souhaitons toutefois montrer qu'un objet distant est automatiquement remplacé par un stub au cours de l'encodage.

ASTUCE

Si vous avez lancé le serveur avec

```
java -Djava.rmi.server.logCalls=true WarehouseServer
```

le serveur enregistrera tous les appels de méthodes distantes dans sa console. Vous pouvez essayer cette option pour obtenir une idée représentative du trafic RMI.

Exemple 4.8 : *ProductImpl.java*

```
import java.rmi.*;
import java.rmi.server.*;

/**
 * Ceci est la classe d'implémentation des objets product
 * distants.
 */
public class ProductImpl
    extends UnicastRemoteObject
    implements Product
{
    /**
     * Construit une implémentation de produit
     * @param n le nom du produit
     */
    public ProductImpl(String n) throws RemoteException
    {
        name = n;
    }

    public String getDescription() throws RemoteException
    {
        return "Je suis un " + name + ". Achetez-moi !";
    }

    public String name;
}
```

Exemple 4.9 : *Customer.java*

```
import java.io.*;

/**
 * Description d'un client. Sachez que les objets customer ne sont pas
 * distants--la classe n'implémente pas d'interface distante.
 */
```

```

*/
public class Customer implements Serializable
{
    /**
     Construit un client.
     @param theAge l'âge du client
     @param theSex le sexe du client (MASCULIN ou FEMININ)
     @param theHobbies les loisirs du client
    */
    public Customer(int theAge, int theSex, String[] theHobbies)
    {
        age = theAge;
        sex = theSex;
        hobbies = theHobbies;
    }

    /**
     Récupère l'âge du client.
     @return l'âge
    */
    public int getAge() { return age; }

    /**
     Récupère le sexe du client
     @return MASCULIN ou FEMININ
    */
    public int getSex() { return sex; }

    /**
     Teste si ce client a un loisir particulier.
     @param aHobby le loisir à tester
     @return true si ce client a choisi le loisir
    */
    public boolean hasHobby(String aHobby)
    {
        if (aHobby == "") return true;
        for (int i = 0; i < hobbies.length; i++)
            if (hobbies[i].equals(aHobby)) return true;

        return false;
    }

    /**
     Réinitialise cet enregistrement client sur ses valeurs par défaut.
    */
    public void reset()
    {
        age = 0;
        sex = 0;
        hobbies = null;
    }

    public String toString()
    {
        String result = "Age: " + age + ", Sexe: ";
        if (sex == Product.MALE) result += "masculin";
        else if (sex == Product.FEMALE) result += "féminin";
        else result += " masculin ou féminin ";
        result += ", Hobbies:";
    }
}

```

254 *Au cœur de Java 2 - Fonctions avancées*

```

        for (int i = 0; i < hobbies.length; i++)
            result += " " + hobbies[i];
        return result;
    }

    private int age;
    private int sex;
    private String[] hobbies;
}

```

Exemple 4.10 : *Warehouse.java*

```

import java.rmi.*;
import java.util.*;

/**
 * L'interface distante d'un entrepôt avec des produits.
 */
public interface Warehouse extends Remote
{
    /**
     * Récupère les produits correspondant au client.
     * @param c le client concerné
     * @return une liste de tableau des produits correspondants
     */
    ArrayList<Product> find(Customer c) throws RemoteException;
}

```

Exemple 4.11 : *WarehouseImpl.java*

```

import java.io.*;
import java.rmi.*;
import java.util.*;
import java.rmi.server.*;
import java.util.*;
import java.util.concurrent.locks.*;

/**
 * Cette classe correspond à l'implémentation de l'interface
 * distante de l'entrepôt.
 */
public class WarehouseImpl
    extends UnicastRemoteObject
    implements Warehouse
{
    /**
     * Construit une implémentation d'entrepôt.
     */
    public WarehouseImpl()
        throws RemoteException
    {
        products = new ArrayList<ProductImpl>();
        add(new ProductImpl("Core Java Book",
            0, 200, Product.BOTH, "Ordinateurs"));
    }
}

```

```

/**
 * Ajoute un produit à l'entrepôt. Ceci est une méthode locale.
 * @param p Le produit à ajouter
 */
public void add(ProductImpl p)
{
    Lock wlock = rwlock.writeLock();
    wlock.lock();
    try
    {
        products.add(p);
    }
    finally
    {
        wlock.unlock();
    }
}

public ArrayList<Product> find(Customer c)
    throws RemoteException
{
    Lock rlock = rwlock.readLock();
    rlock.lock();
    try
    {
        ArrayList<Product> result = new ArrayList<Product>();
        //ajoute tous les produits correspondants
        for (ProductImpl p : products)
        {
            if (p.match(c)) result.add(p);
        }
        //ajoute le produit correspondant à tous, une copie de CoreJava
        if (!result.contains(products.get(0)))
            result.add(products.get(0));

        //nous réinitialisons c simplement pour montrer que c est
        //une copie de l'objet client
        c.reset();
        return result;
    }
    finally
    {
        rlock.unlock();
    }
}

private ArrayList<ProductImpl> products;
private ReadWriteLock rwlock = new ReentrantReadWriteLock();
}

```

Exemple 4.12 : WarehouseServer.java

```

import java.rmi.*;
import java.rmi.server.*;
import javax.naming.*;

```

256 Au cœur de Java 2 - Fonctions avancées

```

/**
 * Ce programme serveur instancie un objet d'un entrepôt
 * distant, l'enregistre auprès du service de nom et attend
 * que les clients invoquent les méthodes.
 */
public class WarehouseServer
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println
                ("Construction des implémentations du serveur...");

            WarehouseImpl w = new WarehouseImpl();
            w.add(new ProductImpl("Grille-pain Moulinex", Product.BOTH,
                18, 200, "Electroménager"));
            w.add(new ProductImpl("Micro-ondes Philips", Product.BOTH,
                18, 200, "Electroménager"));
            w.add(new ProductImpl("Pelle à vapeur MécaTerrassement",
                Product.MALE, 20, 60, "Jardinage"));
            w.add(new ProductImpl("Désherbant U238", Product.BOTH,
                20, 200, "Jardinage"));
            w.add(new ProductImpl("Fragrance Java persistante",
                Product.FEMALE, 15, 45, "Beauté"));
            w.add(new ProductImpl("Souris informatique BlackRongeur",
                Product.BOTH, 6, 40, "Ordinateurs"));
            w.add(new ProductImpl("Mon premier Espresso", Product.FEMALE,
                6, 10, "Electroménager"));
            w.add(new ProductImpl("Eau de Cologne JavaJungle", Product.MALE,
                15, 45, "Beauté"));
            w.add(new ProductImpl("Machine à Espresso FireWire",
                Product.BOTH, 20, 50, "Ordinateurs"));
            w.add(new ProductImpl("Livre Les mauvaises habitudes de Java en
                21 jours", Product.BOTH, 20, 200, "Ordinateurs"));

            System.out.println
                ("Liaison des implémentations du serveur à la base
                ↳ de registres...");
            Context namingContext = new InitialContext();
            namingContext.bind("rmi:central_warehouse", w);

            System.out.println
                ("Attente des invocations des clients...");
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}

```

Exemple 4.13 : WarehouseClient.java

```

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;

```

```
import java.util.*;
import javax.naming.*;
import javax.swing.*;

/**
 * Le client du programme de l'entrepôt.
 */
public class WarehouseClient
{
    public static void main(String[] args)
    {
        try
        {
            System.setProperty("java.security.policy", "client.policy");
            System.setSecurityManager(new RMISecurityManager());

            Properties props = new Properties();
            String fileName = "WarehouseClient.properties";
            FileInputStream in = new FileInputStream(fileName);
            props.load(in);
            String url = props.getProperty("warehouse.url");
            if (url == null)
                url = "rmi://localhost/central_warehouse";

            Context namingContext = new InitialContext();
            Warehouse centralWarehouse = (Warehouse)
                namingContext.lookup(url);
            JFrame frame = new WarehouseClientFrame(centralWarehouse);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}

/**
 * Un bloc pour sélectionner l'âge, le sexe et les hobbies du client
 * et pour montrer les produits correspondants naissant d'un appel
 * distant à l'entrepôt.
 */
class WarehouseClientFrame extends JFrame
{
    public WarehouseClientFrame(Warehouse warehouse)
    {
        this.warehouse = warehouse;
        setTitle("WarehouseClient");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        JPanel panel = new JPanel();
        panel.setLayout(new GridLayout(0, 2));

        panel.add(new JLabel("Age:"));
        age = new JTextField(4);
        age.setText("20");
        panel.add(age);
    }
}
```


258 Au cœur de Java 2 - Fonctions avancées

```

female = new JRadioButton("féminin", true);
male = new JRadioButton("masculin", true);
ButtonGroup group = new ButtonGroup();
panel.add(female); group.add(female);
panel.add(male); group.add(male);

panel.add(new JLabel("Hobbies: "));
hobbies = new ArrayList<JCheckBox>();
for (String h : new String[] { "jardinage", "beauté",
    "ordinateurs", "électroménager", "sports" })
    JCheckBox checkBox = new JCheckBox(h);
    hobbies.add(checkBox);
    panel.add(checkBox);
}

result = new JTextArea(4, 40);
result.setEditable(false);

JPanel buttonPanel = new JPanel();
JButton submitButton = new JButton("Envoyer");
buttonPanel.add(submitButton);
submitButton.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            callWarehouse();
        }
    });

add(panel, BorderLayout.NORTH);
add(result, BorderLayout.CENTER);
add(buttonPanel, BorderLayout.SOUTH);
}

/**
 * Appelle l'entrepôt distant pour trouver des produits
 * correspondants.
 */
private void callWarehouse()
{
    try
    {
        ArrayList<String> selected = new ArrayList<String>();
        for (JCheckBox checkBox : hobbies)
            if (checkBox.isSelected()) selected.add(checkBox.getText());
        Customer c = new Customer(Integer.parseInt(age.getText()),
            (male.isSelected() ? Product.MALE : 0)
            + (female.isSelected() ? Product.FEMALE : 0),
            selected.toArray(new String[selected.size()]));
        ArrayList<Product> recommendations = warehouse.find(c);
        result.setText(c + "\n");
        for (Product p : recommendations)
        {
            String t = p.getDescription() + "\n";
            result.append(t);
        }
    }
}

```

```

        catch(Exception e)
        {
            e.printStackTrace();
            result.setText("Exception: " + e);
        }
    }

    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 300;

    private Warehouse warehouse;
    private JTextField age;
    private JRadioButton male;
    private JRadioButton female;
    private ArrayList<JCheckBox> hobbies;
    private JTextArea result;
}

```

Passer des objets distants

Le passage d'objets distants du serveur au client est assez simple. Le client reçoit un objet stub, le sauvegarde dans une variable dont le type est le même que celui de l'interface distante. Le client peut alors accéder à l'objet d'origine sur le serveur en passant par cette variable. Le client peut aussi copier cette variable sur sa machine locale, toutes les copies effectuées restant de simples références au même stub. Il est important de remarquer que seules les interfaces distantes peuvent être atteintes par ce stub. Une interface distante est une interface qui étend l'interface `Remote`, alors qu'une méthode locale est une méthode qui n'est définie dans aucune interface distante. Les méthodes locales ne peuvent fonctionner que sur la machine virtuelle contenant l'objet réel.

Ensuite, les stubs sont générés uniquement à partir de classes qui implémentent une interface distante, et seules les méthodes spécifiées dans les interfaces sont fournies aux classes de stub. Si une sous-classe n'implémente pas d'interface distante, mais qu'une superclasse le fait, et qu'un objet de la sous-classe est passé à une méthode distante, seules les méthodes de la superclasse sont accessibles. Pour mieux comprendre ce fonctionnement, considérons l'exemple suivant, dans lequel nous dérivons une classe `BookImpl` à partir de `ProductImpl`.

```

class BookImpl extends ProductImpl
{
    public BookImpl(String title, String theISBN,
        int sex, int age1, int age2, String hobby)
    {
        super(title + " livre", sex, age1, age2, hobby);
        ISBN = theISBN;
    }
    public String getStockCode() { return ISBN; }
    private String ISBN;
}

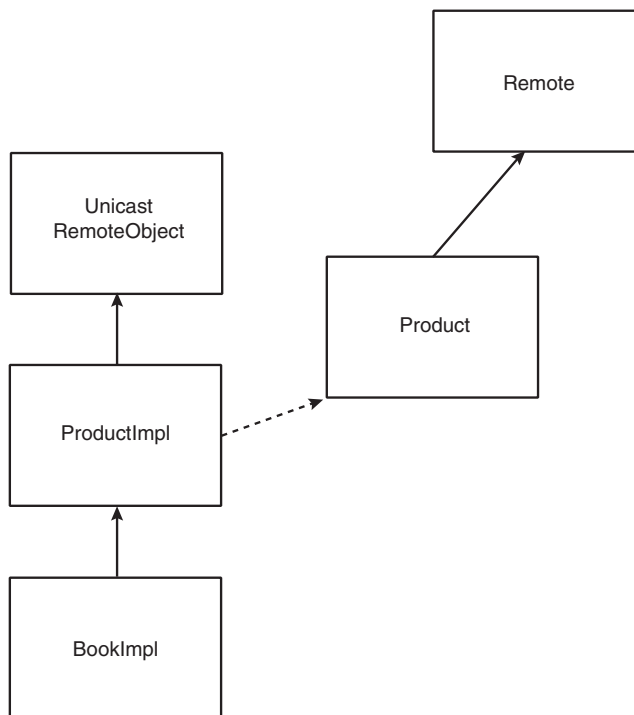
```

Supposons maintenant que nous passions un objet `Book` à une méthode distante, soit comme paramètre soit comme valeur de retour. Le destinataire reçoit un objet stub. Mais ce stub n'est pas un stub de `book`. Il s'agit en fait d'un stub de la superclasse `ProductImpl`, puisque cette classe implémente une interface distante (voir Figure 4.9). Par conséquent, dans ce cas, la méthode `getStockCode` n'est pas accessible à distance.

260 Au cœur de Java 2 - Fonctions avancées

Figure 4.9

Seules les méthodes de ProductImpl sont accessibles à distance.



Une classe distante peut implémenter plusieurs interfaces. Par exemple, la classe BookImpl peut implémenter une seconde interface en plus de Product. Ici, nous définissons une interface distante StockUnit qui est implémentée par la classe BookImpl :

```

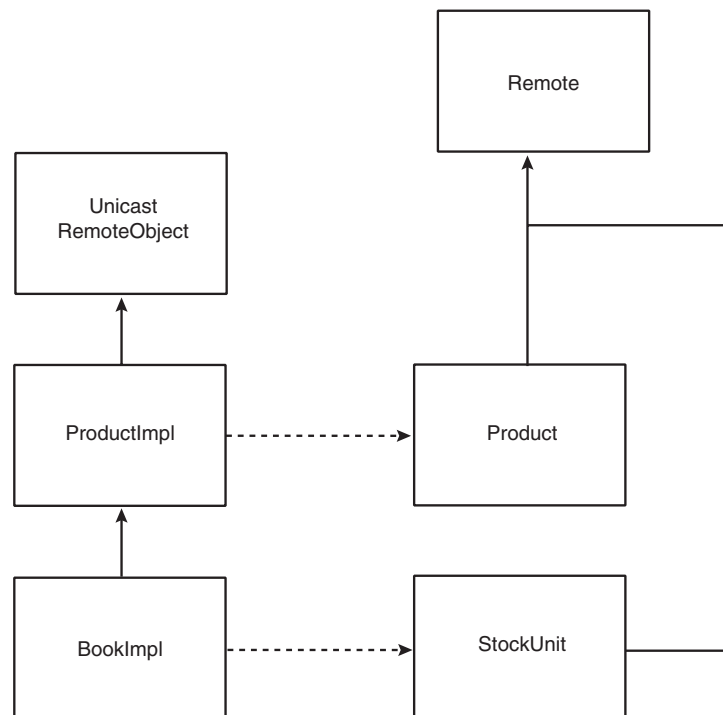
interface StockUnit extends Remote
{
    public String getStockCode() throws RemoteException;
}

class BookImpl extends ProductImpl implements StockUnit
{
    public BookImpl(String title, String theISBN,
        int sex, int age1, int age2, String hobby)
        throws RemoteException
    {
        super(title + " livre", sex, age1, age2, hobby);
        ISBN = theISBN;
    }
    public String getStockCode() throws RemoteException
    {
        return ISBN;
    }
    private String ISBN;
}
  
```

La Figure 4.10 fournit le diagramme d'héritage.

Figure 4.10

BookImpl possède d'autres méthodes distantes.



Maintenant, lorsqu'un objet book est passé à une méthode distante, le destinataire obtient un stub possédant un accès aux méthodes distantes des classes **Product** et **StockUnit**. En fait, vous pouvez vous servir de l'opérateur `instanceof` pour déterminer si un objet distant particulier implémente une interface. Voici une situation typique où vous serez amené à vous servir de cette technique. Supposons que vous receviez un objet distant par une variable de type **Product**.

```

ArrayList<Product> result = centralWarehouse.find(c);
for (Product p : result)
{
    . . .
}

```

L'objet distant peut ou non correspondre à un livre. Nous pouvons donc nous servir de `instanceof` pour déterminer si c'en est bien un, ou non. Mais nous ne pouvons pas tester

```

if (p instanceof BookImpl) // faux
{
    BookImpl b = (BookImpl) p;
    . . .
}

```

L'objet **p** fait référence à un objet stub, et **BookImpl** est la classe de l'objet du serveur. Nous choisissons le type de la seconde interface :

```

if (p instanceof StockUnit)
{
    StockUnit s = (StockUnit) p;
}

```

262 Au cœur de Java 2 - Fonctions avancées

```
String c = s.getStockCode();  
.  
.  
.  
}
```

Ce code examine si l'objet stub auquel `p` fait référence implémente l'interface `StockUnit`. Dans ce cas, il appelle la méthode distante `getStockCode` de cette interface.

Pour résumer :

- Si un objet appartenant à une classe qui implémente une interface distante est passé à une méthode distante, cette méthode distante reçoit un objet stub.
- Vous pouvez modifier le type de cet objet stub et choisir le type de n'importe quelle interface distante qui est implémenté par la classe d'implémentation.
- Vous pouvez appeler toutes les méthodes distantes définies dans ces interfaces, mais vous ne pouvez appeler aucune méthode locale au travers d'un stub.

Objets distants et méthodes `equals` et `hashCode`

Comme nous l'avons vu au cours du Chapitre 2, les objets insérés dans des ensembles (*sets*) doivent surcharger la méthode `equals`. Dans le cas d'un ensemble de hachage ou d'une carte de hachage, la méthode `hashCode` doit également être définie. Sinon, vous aurez un problème lorsque vous essaie-
rez de comparer des objets distants. Pour vérifier que deux objets distants possèdent le même contenu, un appel à la méthode `equals` aurait besoin de contacter les serveurs contenant les objets et de comparer leur contenu. Et cet appel pourrait échouer. Mais la méthode `equals` de la classe `Object` n'est pas définie pour déclencher une `RemoteException`, alors que toutes les méthodes d'une interface distante doivent déclencher cette exception. Comme une méthode d'une sous-classe ne peut pas déclencher plus d'exceptions que la superclasse qu'elle remplace, vous ne pourrez pas trouver de méthode `equals` dans une interface distante. Cela est également vrai pour `hashCode`.

Les méthodes `equals` et `hashCode` sur les objets stub examinent simplement l'emplacement des objets de serveur. La méthode `equals` considère que deux stubs sont égaux s'ils font référence au même objet du serveur. Deux stubs faisant référence à des objets différents du serveur ne peuvent jamais être égaux, même si les contenus de ces deux objets sont identiques. De même, le code de hachage est calculé uniquement à partir de l'identificateur de l'objet.

Pour résumer : vous pouvez utiliser des objets de stubs dans des tables de hachage, mais vous devez vous rappeler que les tests d'égalité et le calcul des codes de hachage ne prennent pas en compte le contenu des objets distants.

Cloner des objets distants

Les stubs ne possèdent pas de méthode `clone`, vous ne pouvez donc pas cloner un objet distant en invoquant la méthode `clone` d'un stub. La raison en est assez technique. Si `clone` devait effectuer un appel distant pour demander au serveur de cloner l'objet d'implémentation, la méthode `clone` devrait pouvoir déclencher une `RemoteException`. Mais la méthode `clone` de la superclasse `Object` ne peut déclencher aucune exception à part `CloneNotSupportedException`. C'est exactement la même limitation que vous avez rencontrée dans la section précédente, lorsque nous avons vu que les méthodes `equals` et `hashCode` n'examinent pas la valeur des objets distants, mais qu'elles se contentent de comparer les références des stubs. Mais cela n'a aucun sens pour `clone` de faire un

autre clone d'un stub. Si vous souhaitez obtenir une autre référence à un objet distant, il vous suffit de copier la variable du stub. Par conséquent, `clone` est tout simplement non défini pour les stubs.

Activation des objets du serveur

Dans le précédent exemple de programme, nous avons utilisé un programme de serveur dans le but d'instancier et d'enregistrer des objets, de manière que les clients soient en mesure d'effectuer des appels distants sur eux. Toutefois, dans certains cas, le fait d'instancier un grand nombre d'objets de serveur peut se révéler inutile et leur faire attendre des connexions, que les objets du client les utilisent ou non. Le mécanisme d'*activation* vous permet de retarder la construction de l'objet, afin qu'un objet du serveur ne soit construit que lorsqu'un client au moins appelle une méthode distante dessus.

Pour profiter de l'activation, le code client est totalement inchangé. Le client demande simplement une référence distante et effectue ses appels à travers elle.

Malgré tout, le programme du serveur est remplacé par un programme d'activation qui construit des *descripteurs d'activation* des objets devant être construits par la suite, et lie les récepteurs des appels de méthode distants grâce au service de dénomination. Au premier lancement d'un appel, les informations contenues dans le descripteur d'activation permettent de construire l'objet.

Un objet du serveur utilisé de cette manière devrait prolonger la classe `Activatable` et, bien entendu, implémenter une ou plusieurs interfaces distantes. Par exemple :

```
class ProductImpl
    extends Activatable
    implements Product
{
    . . .
}
```

Etant donné que la construction de l'objet est retardée, elle doit se réaliser de manière standardisée. Vous devez donc fournir un constructeur qui prenne deux paramètres :

- une ID d'activation (que vous passez simplement au constructeur de la superclasse) ;
- un seul objet contenant toutes les informations de construction, emballées dans un `MarshaledObject`.

Dans les cas où vous avez besoin de plusieurs paramètres de construction, il conviendra de les assembler dans un objet unique. Vous pourrez ainsi toujours utiliser un tableau `Object[]` ou un `ArrayList`. Comme vous le constaterez dans un moment, vous allez placer une copie sérialisée (ou encodée) des informations de construction à l'intérieur du descripteur de l'activation. Le constructeur de votre objet serveur doit utiliser la méthode `get` de la classe `MarshaledObject` pour désérialiser les informations de construction.

En ce qui concerne la classe `ProductImpl`, la situation est assez simple : une seule information est nécessaire à sa construction, à savoir le nom du produit.

Cette information peut être emballée dans un `MarshaledObject` et développée dans le constructeur.

```
public ProductImpl(ActivationID id, MarshaledObject data)
{
    super(id, 0);
```

264 *Au cœur de Java 2 - Fonctions avancées*

```

        name = (String) data.get();
        System.out.println("Construit " + name);
    }

```

En passant 0 comme second paramètre du constructeur de la superclasse, nous signifions que la bibliothèque RMI doit assigner un numéro de port adapté au port de l'écouteur.

Ce constructeur imprime un message vous permettant de voir que les objets product sont activés sur demande.

INFO

Vos objets de serveur n'ont pas nécessairement à étendre la classe `Activatable`. S'ils ne le font pas, placez l'appel de méthode statique

```
Activatable.exportObject(this, id, 0)
```

dans le constructeur de la classe du serveur.

Intéressons-nous maintenant au programme d'activation. Vous devez tout d'abord définir un groupe d'activation qui décrit les paramètres ordinaires du lancement de la machine virtuelle hébergeant les objets du serveur. Les paramètres les plus importants à prendre en compte sont les règles de sécurité.

Comme avec les autres objets du serveur, nous n'effectuerons pas de vérifications de sécurité et supposons qu'ils proviennent d'une source de confiance. Toutefois, la machine virtuelle sur laquelle les objets activés s'exécutent a installé un gestionnaire de sécurité. Pour activer toutes les autorisations, il vous faut fournir un fichier `server.policy` contenant les éléments suivants :

```

grant
{
    permission java.security.AllPermission;
};

```

Construisez un descripteur de groupe d'activation comme suit :

```

Properties props = new Properties();
props.put("java.security.policy", "/server/server.policy");
ActivationGroupDesc group = new ActivationGroupDesc(props, null);

```

Le deuxième paramètre permet de décrire des options de commande spéciales ; nous n'en aurons pas besoin dans cet exemple, nous passerons donc une référence `null`.

Vous allez maintenant créer une ID de groupe, grâce à l'appel :

```

ActivationGroupID id
    = ActivationGroup.getSystem().registerGroup(group);

```

Vous pouvez maintenant construire les descripteurs d'activation. Pour chaque objet qui doit être construit à la demande, vous avez besoin de :

- l'ID du groupe d'activation pour la machine virtuelle sur laquelle l'objet doit être construit ;
- le nom de la classe (comme `"ProductImpl"` ou `"com.mycompany.MyClassImpl"`) ;
- la chaîne d'URL à partir de laquelle vous allez charger les fichiers de classe. Il doit s'agir de l'URL de base et elle ne doit pas inclure les chemins du package ;
- les informations sur la construction encodée.

Par exemple :

```
MarshaledObject param
    = new MarshaledObject("Grille-pain Moulinex");
ActivationDesc desc = new ActivationDesc(id, "ProductImpl",
    "http://myserver.com/download/", param);
```

Passez le descripteur à la méthode statique `Activatable.register`. Elle renvoie un objet d'une classe qui implémente les interfaces distantes de la classe d'implémentation. Vous pouvez lier cet objet au service de dénomination :

```
Product p = (Product)Activatable.register(desc);
namingContext.bind("grille-pain", p);
```

A la différence des programmes du serveur des exemples précédents, le programme d'activation se ferme après enregistrement et liaison des récepteurs d'activation. Les objets du serveur sont uniquement construits lors du premier appel à la méthode.

Les Exemples 4.14 et 4.15 présentent le code d'implémentation du produit activatable ainsi que le programme d'activation. L'interface produit et le programme client demeurent inchangés.

Pour lancer ce programme, procédez comme suit :

1. Compilez tous les fichiers source.
2. Si vous utilisez le JDK 1.4 ou version antérieure, lancez `rmic` pour générer un stub pour la classe `ProductImpl` :
3. Lancez la base de registres RMI.
4. Démarrez le démon d'activation RMI.

```
rmid -J-Djava.security.policy=rmid.policy &
ou
start rmid -J-Djava.security.policy=rmid.policy
```

Le programme `rmid` écoute les requêtes d'activation et active les objets dans une machine virtuelle séparée. Pour lancer une machine virtuelle, le programme `rmid` a besoin de certaines autorisations, spécifiées dans un fichier des règles de sécurité (voir Exemple 4.16). L'option `-J` permet de passer une option à la machine virtuelle qui exécute le démon d'activation.

5. Lancez le programme d'activation. Dans cette configuration, nous supposons que vous démarrez le programme dans le répertoire qui contient les fichiers de classe et le fichier des règles de sécurité.

```
Java ProductActivator
```

Le programme se ferme après enregistrement des récepteurs d'activation auprès du service de dénomination.

6. Exécutez le programme client

```
java -Djava.security.policy=client.policy ProductClient
```

Le client affichera les descriptions des produits habituels. Lorsque vous exécuterez le client pour la première fois, les messages du constructeur s'afficheront dans la fenêtre du serveur.

266 *Au cœur de Java 2 - Fonctions avancées*

Exemple 4.14 : *ProductImpl.java*

```
import java.rmi.*;
import java.rmi.server.*;

/**
 * Ceci est la classe d'implémentation pour les objets du produit
 * distant.
 */
public class ProductImpl
    extends UnicastRemoteObject
    implements Product
{
    /**
     * Construit une implémentation de produit
     * @param n le nom du produit
     */
    public ProductImpl(String n) throws RemoteException
    {
        name = n;
    }

    public String getDescription() throws RemoteException
    {
        return "Je suis un " + name + ". Achetez-moi !";
    }

    private String name;
}
```

Exemple 4.15 : *ProductActivator.java*

```
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.activation.*;
import java.util.*;
import javax.naming.*;

/**
 * Ce programme serveur active deux objets distants et
 * les enregistre auprès du service de noms.
 */
public class ProductActivator
{
    public static void main(String args[])
    {
        try
        {
            System.out.println
                ("Construction des descripteurs d'activation...");

            Properties props = new Properties();
            // utilise le fichier server.policy du répertoire courant
            props.put("java.security.policy",
                new File("server.policy").getCanonicalPath());
            ActivationGroupDesc group = new ActivationGroupDesc(props, null);
            ActivationGroupID id
                = ActivationGroup.getSystem().registerGroup(group);
            MarshalledObject p1param
```

```

        = new MarshalledObject("Grille-pain Moulinex");
MarshalledObject p2param
        = new MarshalledObject("Micro-ondes Philips");

String classDir = ".";
// transforme le répertoire de classe en URL de fichier
// pour cette démo, nous supposons que les classes sont dans le
// rép. Courant
// Nous utilisons toURI pour que les espaces et autres
// caractères spéciaux des noms de fichiers soient ignorés
String classURL
        = new File(classDir).getCanonicalFile().toURI().toString();

ActivationDesc desc1 = new ActivationDesc
        (id, "ProductImpl", classURL, p1param);
ActivationDesc desc2 = new ActivationDesc
        (id, "ProductImpl", classURL, p2param);

Product p1 = (Product) Activatable.register(desc1);
Product p2 = (Product) Activatable.register(desc2);

System.out.println
        ("Liaison des implémentations activables à la base de
registre...");
Context namingContext = new InitialContext();
namingContext.bind("rmi:grille-pain", p1);
namingContext.bind("rmi:micro-ondes", p2);

System.out.println ("Fermeture...");
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
}

```

Exemple 4.16 : *rmid.policy*

```

grant
{
    permission com.sun.rmi.rmid.ExecPermission
        "${java.home}${/}bin${/}java";
    permission com.sun.rmi.rmid.ExecOptionPermission
        "-Djava.security.policy=*";
};

```

Exemple 4.17 : *server.policy*

```

grant
{
    permission java.security.AllPermission;
};

```

java.rmi.activation.Activatable 1.2

- `protected Activatable(ActivationID id, int port)`
 Construit l'objet activatable et établit un écouteur sur le port donné. Utilisez 0 pour qu'un port soit automatiquement affecté.

268 *Au cœur de Java 2 - Fonctions avancées*

- `static Remote exportObject (Remote obj, ActivationID id, int port)`
Crée un objet distant activatable. Renvoie le récepteur d'activation qui doit être mis à la disposition des appelants distants. Utilisez 0 pour qu'un port soit automatiquement affecté.
- `static Remote register (ActivationDescriptor desc)`
Enregistre le descripteur d'un objet activatable et le prépare à recevoir des appels distants. Renvoie le récepteur d'activation qui doit être mis à la disposition des appelants distants.

java.rmi.MarshalledObject 1.2

- `MarshalledObject(Object obj)`
Construit un objet contenant les données sérialisées d'un objet donné.
- `Object get()`
Déséréalise les données de l'objet stocké et renvoie l'objet.

java.rmi.activation.ActivationGroupDesc 1.2

- `ActivationGroupDesc(Properties props, ActivationGroupDesc.CommandEnvironment env)`
Construit un descripteur de groupe d'activation qui spécifie les propriétés d'une machine virtuelle hébergeant des objets activés. Le paramètre `env` contient le chemin jusqu'à la machine virtuelle exécutable et les options de ligne de commande, ou `null` si aucun paramètre particulier n'est requis.

java.rmi.activation.ActivationGroup 1.2

- `Static ActivationSystem getSystem()`
Renvoie une référence au système d'activation.

java.rmi.activation.ActivationSystem 1.2

- `ActivationGroupID registerGroup(ActivationGroupDesc group)`
Enregistre un groupe d'activation et renvoie l'ID du groupe.

java.rmi.activation.ActivationDesc 1.2

- `ActivationDesc(ActivationGroupID id, String className, String classFileURL, MarshalledObject data)`
Construit un descripteur d'activation.

IDL Java et CORBA

Contrairement à RMI, CORBA vous permet d'effectuer des appels entre des objets Java et des objets écrits dans d'autres langages. CORBA nécessite cependant qu'un ORB (*Object Request Broker*) soit disponible à la fois sur le client et sur le serveur. Vous pouvez considérer que ORB est une sorte de traducteur universel permettant à des objets CORBA de communiquer. La spécification CORBA 2 définit plus d'une douzaine de services que l'ORB peut utiliser lors d'un certain nombre de tâches d'organisation. Ces tâches varient d'un "service de démarrage" permettant l'exécution d'un processus, à un "service de durée de vie", que vous pouvez utiliser pour créer, copier, déplacer ou supprimer des objets, ou encore à un "service de désignation" qui vous permet de chercher des objets en fonction de leur nom.

Le JDK 1.2 a introduit une implémentation complète d'un ORB compatible CORBA 2. Par conséquent, toutes les applications et les applets ont la possibilité de se connecter à des objets CORBA distants.

INFO

Sun désigne la compatibilité avec CORBA sous le terme "Java IDL". Ce terme est vraiment mal choisi. IDL fait en effet référence à un langage de définition d'interfaces, c'est-à-dire à un langage pour décrire les interfaces des classes. L'aspect important de la technologie apportée par Sun est la connectivité avec CORBA, et pas seulement le support de IDL.

Voici la liste des étapes à suivre pour implémenter des objets CORBA :

1. Ecrivez l'interface qui spécifie le fonctionnement de l'objet en utilisant IDL, le langage de définition d'interfaces permettant de définir les interfaces CORBA. IDL est un langage spécial pour définir des interfaces sous une forme indépendante du langage d'origine.
2. En utilisant le compilateur pour le langage cible, générez les classes d'aide et de stub nécessaires.
3. Ajoutez le code d'implémentation pour les objets du serveur, en utilisant le langage de votre choix. Le squelette fourni par le compilateur IDL n'est qu'un code de base. Vous devrez toujours fournir le code de l'implémentation réelle pour les méthodes du serveur. Compilez ce code d'implémentation.
4. Ecrivez un programme de serveur qui crée et enregistre des objets de serveur. La méthode la plus simple pour l'enregistrement consiste à utiliser le *service de désignation* de CORBA, qui est en fait un service analogue à `rmiregistry`.
5. Ecrivez un programme client qui localise les objets du serveur et invoque des services sur ces objets.
6. Lancez le service de désignation et le programme du serveur sur le serveur, et lancez le programme client sur le client.

Ces étapes sont relativement comparables aux étapes nécessaires à la construction d'applications distribuées avec RMI. Il existe cependant deux différences importantes :

- Vous pouvez utiliser n'importe quel langage relié à CORBA pour implémenter des clients et des serveurs.
- Vous vous servez de IDL pour spécifier les interfaces.

Dans les sections suivantes, vous verrez comment utiliser IDL pour définir des interfaces CORBA, et comment connecter des clients écrits en Java avec des serveurs écrits en C++, et des clients C++ avec des serveurs implémentés avec Java.

Cependant, CORBA est un sujet assez complexe et nous nous contenterons de vous fournir quelques exemples fondamentaux qui vous mettront sur une bonne voie pour commencer.

270 Au cœur de Java 2 - Fonctions avancées

Langage de définition d'interfaces

Pour introduire la syntaxe IDL, revenons un instant sur l'exemple que nous avons utilisé pour RMI. Avec RMI, nous avons commencé par une interface en Java. Avec CORBA, le point de départ est une interface en syntaxe IDL :

```
interface Product
{
    string getDescription();
};
```

Il existe quelques différences subtiles entre IDL et Java. En IDL, la définition de l'interface se termine par un point-virgule. Vous remarquerez que `string` est écrit en minuscules. En fait, la classe `string` fait référence à la notion de chaînes de CORBA, qui est différente de la notion de chaînes en Java. Avec le langage de programmation Java, les chaînes contiennent des caractères Unicode sur 16 bits. Sous CORBA, les chaînes ne contiennent que des caractères de 8 bits. Si vous envoyez une chaîne de caractères sur 16 bits à l'ORB et que cette chaîne possède des caractères spéciaux dont l'octet de poids fort est différent de zéro, une exception est déclenchée. Ce genre de problème provoqué par des erreurs de types est le prix à payer pour une interopérabilité entre les langages de programmation.

INFO

CORBA possède également les types `wchar` et `wstring` pour les caractères étendus (*wide*). Cependant, il n'existe aucune garantie que les chaînes de caractères étendus se servent bien d'un encodage Unicode.

Le compilateur "Java vers IDL" (aussi appelé compilateur Java IDL) traduit les définitions IDL en définitions d'interfaces pour Java. Par exemple, si vous placez la définition IDL de `Product` dans un fichier `Product.idl` et que vous lancez

```
idlj Product.idl
```

vous obtiendrez un fichier `ProductOperations.java` avec le contenu suivant :

```
interface ProductOperations
{
    String getDescription();
}
```

ainsi qu'un fichier `Product.java` qui définira une interface

```
public interface Product extends
    ProductOperations,
    org.omg.CORBA.Object,
    org.omg.CORBA.portable.IDLEntity
{
}
```

INFO

Avec le JDK 1.2, le programme `idltojava` permet de traduire les fichiers IDL.

Le compilateur IDL génère aussi plusieurs autres fichiers source, dont la classe de stub permettant de communiquer avec ORB et trois classes d'aide sur lesquelles nous reviendrons dans cette section et dans la suivante.

INFO

Il n'est pas possible de programmer en IDL. IDL sert uniquement à définir des interfaces. Les objets CORBA décrits par IDL doivent toujours être implémentés, par exemple en C++ ou en Java.

Les règles qui gouvernent la traduction de IDL en Java sont couramment appelées *liaisons (ou enchaînements) du langage de programmation Java*. Les liaisons de langages sont standardisées par l'OMG. Tous les vendeurs de solutions CORBA doivent utiliser les mêmes règles pour adapter les constructions IDL dans un langage de programmation particulier.

Nous n'aborderons pas tous les aspects de la liaison entre IDL et Java. Reportez-vous à la documentation CORBA sur le site Web de l'OMG (<http://www.omg.org>) pour une description plus complète. Cependant, il existe certains concepts importants que chaque utilisateur de IDL devrait connaître.

Lorsque vous définissez une méthode, vous disposez de plus d'options pour le passage des paramètres que n'en offre Java. Chaque paramètre peut être déclaré en entrée, en sortie ou en entrée et en sortie (in, out ou inout). Les paramètres en entrée sont simplement passés à la méthode concernée, comme dans Java. Cependant, le langage de programmation Java ne possède pas d'équivalent pour les paramètres en sortie. Une méthode enregistre une valeur dans chaque paramètre out avant de se terminer. L'appelant peut ensuite retrouver les valeurs enregistrées dans ces paramètres out.

Par exemple, une méthode find peut enregistrer l'objet Product qu'elle a trouvé :

```
interface Warehouse
{
    boolean locate(in String descr, out Product p);
    . . .
};
```

Si le paramètre est déclaré uniquement en sortie, la méthode ne doit pas s'attendre que ce paramètre soit initialisé. Cependant, si le paramètre est déclaré en inout, l'appelant doit fournir une valeur à la méthode. Cette dernière peut à son tour modifier cette valeur pour que l'appelant récupère cette valeur modifiée. Avec Java, ces paramètres sont simulés avec des classes particulières appelées *classes propriétaires (holder)* qui sont générées par le compilateur IDL Java.

Le compilateur IDL génère une classe avec le suffixe Holder pour chaque interface. Par exemple, lorsque vous compilez l'interface Product, il génère automatiquement une classe ProductHolder. Chaque classe propriétaire possède une variable d'instance publique appelée value.

Lorsqu'une méthode possède un paramètre out, le compilateur IDL change la signature de la méthode pour utiliser une classe propriétaire, comme dans l'exemple suivant :

```
interface Warehouse
{
    boolean locate(String descr, ProductHolder p);
    . . .
};
```

272 *Au cœur de Java 2 - Fonctions avancées*

Lorsque vous appelez la méthode, vous devez passer dans un objet propriétaire. Lorsque la méthode est terminée, vous pouvez récupérer la valeur à partir du paramètre out de l'objet propriétaire. Voici comment appeler la méthode locate :

```
Warehouse w = . . . ;
String descr = . . . ;
Product p;
ProductHolder pHolder = new ProductHolder();
if (w.locate(descr, pHolder))
    p = pHolder.value;
```

Il existe des classes propriétaires prédéfinies pour les types fondamentaux (comme `IntHolder`, `DoubleHolder`, etc.).

INFO

IDL ne supporte pas les méthodes surchargées. Vous serez donc amené à fournir un nom différent pour chaque méthode.

Avec IDL, vous devez utiliser une construction `sequence` pour définir des tableaux de tailles variables. Il faut commencer par définir un type avant de déclarer des paramètres `sequence` ou des valeurs de retour. Par exemple, voici une définition d'un type "séquence de produits" :

```
typedef sequence<Product> ProductSeq;
```

Vous pouvez alors utiliser ce type dans des déclarations de méthodes :

```
interface Warehouse
{
    ProductSeq find(in Customer c);
    . . .
};
```

Avec le langage de programmation Java, les séquences correspondent à des tableaux. Par exemple, la méthode `find` est transformée en

```
Product[] find(Customer c)
```

Si une méthode peut déclencher une exception, il faut commencer par définir le type de l'exception, puis utiliser une déclaration `raises`. Dans l'exemple suivant, la méthode `find` peut déclencher une exception `BadCustomer` :

```
interface Warehouse
{
    exception BadCustomer { string reason; };
    ProductSeq find(in Customer c) raises BadCustomer;
    . . .
};
```

Le compilateur IDL traduit les types d'exceptions par des classes.

```
final public class BadCustomer
    extends org.omg.CORBA.UserException
{
    public BadCustomer() {}
```

```
public BadCustomer(String __reason) { reason = __reason; }
public String reason;
}
```

Si vous interceptez ce type d'exception, vous pouvez examiner ses variables d'instance publique.

L'instruction `raises` se transforme en `throws` pour une méthode Java :

```
ProductSeq find(Customer c) throws BadCustomer
```

Les interfaces peuvent contenir des constantes, comme le montre l'exemple suivant :

```
interface Warehouse
{
    const int SOLD_OUT = 404;
    . . .
};
```

Les interfaces peuvent également renfermer des attributs. Un attribut est comparable à une variable d'instance, mais il constitue en fait un raccourci pour certaines méthodes accessoires et de mutation. Par exemple, voici une interface `Book` avec un attribut `isbn` :

```
interface Book
{
    attribute string isbn;
    . . .
};
```

L'équivalent en Java serait une paire de méthodes possédant chacune le nom `isbn` :

```
String isbn() // accessoire
void isbn(String __isbn) // mutateur
```

Si l'attribut a été déclaré en lecture seule, aucune méthode de mutation n'est générée.

Vous ne pouvez pas spécifier de variables dans des interfaces CORBA, puisque la représentation des données pour les objets fait partie de la stratégie d'implémentation retenue, et que IDL ne gère pas du tout les implémentations.

CORBA supporte les héritages d'interfaces, comme ceci :

```
interface Book : Product { /* . . . */ };
```

Il faut utiliser les deux-points (:) pour signaler un héritage. Une interface peut hériter de plusieurs interfaces.

Avec IDL, vous pouvez grouper les définitions des interfaces, des types, des constantes et des exceptions dans des *modules*.

```
module corejava
{
    interface Product
    {
        . . .
    };

    interface Warehouse
    {
        . . .
    };
};
```

Les modules sont transformés en *packages* en Java.

274 *Au cœur de Java 2 - Fonctions avancées*

Lorsque vous avez votre fichier IDL, il suffit d'exécuter le compilateur IDL fourni par votre vendeur d'ORB pour obtenir des classes de stub et d'aide en fonction de votre langage de programmation cible (comme Java ou C++).

Par exemple, pour convertir des fichiers IDL en Java, vous devez lancer le programme `idlj`, et fournir le nom du fichier IDL sur la ligne de commande :

```
idltojava Product.idl
```

Ce programme crée cinq fichiers source :

- `Product.java`, la définition de l'interface ;
- `ProductOperations.java`, l'interface qui contient les opérations réelles (`Product` étend `ProductOperations` ainsi que deux interfaces spécifiques à CORBA) ;
- `ProductHolder.java`, la classe propriétaire pour les paramètres out ;
- `ProductHelper.java`, une classe d'aide, dont vous verrez l'intérêt dans la section suivante ;
- `_ProductStub.java`, la classe de stub pour communiquer avec ORB.

Le même fichier IDL peut être compilé en C++. Nous nous servons d'un ORB disponible gratuitement pour nos exemples. Le package `omniORB` contient un compilateur IDL vers C++ appelé `omniidl2`. Pour générer des stubs C++, invoquez-le de la manière suivante :

```
omniidl -bcxx Product.idl
```

Vous obtenez deux fichiers C++ :

- `Product.hh`, un fichier d'en-tête qui définit les classes `Product`, `ProductHelper`, et `POA_Product` (la superclasse d'implémentation du serveur).
- `ProductSK.cc`, un fichier C++ contenant le code source de ces classes.

INFO

Bien que la liaison du langage soit standardisée, il revient à chaque vendeur de générer et de concevoir le code qui réalise la liaison. Les compilateurs IDL vers C++ d'autres vendeurs généreront un autre ensemble de fichiers.

Un exemple en CORBA

Dans notre premier exemple, nous vous montrerons comment appeler un objet d'un serveur C++ à partir d'un client Java, en tirant profit du support CORBA intégré au JDK. Au niveau du serveur, nous nous servons d'`omniORB`, un ORB disponible gratuitement disponible sur le site <http://omniorb.sourceforge.net>.

INFO

Vous pouvez utiliser n'importe quel ORB compatible CORBA 2 sur le serveur. Cependant, vous devrez apporter quelques modifications au code C++ si vous vous servez d'un autre ORB.

Notre exemple d'objet de serveur C++ se contente de renvoyer la valeur d'une variable d'environnement du serveur. L'interface est la suivante :

```
interface Env
{
    string getenv(in string name);
};
```

Par exemple, le fragment de programme Java suivant récupère la valeur de la variable d'environnement PATH du processus dans lequel l'objet du serveur tourne.

```
Env env = . . . ;
String value = env.getenv("PATH")
```

Le code de l'implémentation C++ de cette interface va de soi. Nous appelons la méthode `getenv` de la bibliothèque C standard.

```
class EnvImpl
: public POA_Env, public PortableServer::RefCountServantBase
{
public:
    virtual char* getenv(const char *name)
    {
        char* value = std::getenv(name);
        return CORBA::string_dup(value);
    }
};
```

Vous n'avez pas besoin de comprendre le code C++ pour poursuivre cette section, il vous suffit de le considérer comme un ancien programme qu'il faut encapsuler dans un objet CORBA, de sorte que vous pourrez l'appeler à partir d'autres programmes écrits en Java.

Au niveau du serveur, vous devez écrire un programme C++ qui réalise les tâches suivantes :

1. Démarrer l'ORB.
2. Créer un objet de la classe `EnvImpl` et l'enregistrer avec l'ORB.
3. Utiliser le nom du serveur pour lier l'objet à un nom.
4. Attendre une invocation du client.

Vous trouverez ce programme dans l'Exemple 4.19 à la fin de cette section. Nous n'examinerons pas le code C++ en détail. Si vous êtes intéressé, consultez la documentation d'omniORB pour plus d'informations. Cette documentation contient un cours très bien présenté qui explique chaque étape.

Passons maintenant au code du client. Vous savez déjà comment invoquer une méthode d'un objet du serveur une fois que vous possédez une référence à l'objet distant. Cependant, pour obtenir cette référence, vous devrez vous débrouiller sans RMI.

Tout d'abord, initialisez l'ORB. Ce dernier est simplement une bibliothèque de codes qui sait comment communiquer avec d'autres ORB, et comment encoder et décoder des paramètres.

```
ORB orb = ORB.init(args, null);
```

Puis, vous devrez localiser le service d'identification qui vous aidera à localiser d'autres objets. Cependant, avec CORBA, le service d'identification est juste un autre objet CORBA. Pour appeler ce service, vous devrez d'abord le localiser. A l'époque de CORBA 1, cela posait énormément de problèmes, puisqu'il n'existait aucun moyen standard d'en obtenir la référence. Mais un ORB

276 Au cœur de Java 2 - Fonctions avancées

CORBA 2 vous permet de localiser certains services standard en fonction de leur nom. L'appel suivant :

```
String[] services = orb.list_initial_services();
```

établit la liste des noms des services standard auxquels l'ORB peut se connecter. Le service d'identification possède le nom standard `NameService`. La plupart des ORB possèdent d'autres services de démarrage, comme le service `RootPOA`, qui accède au POA (*Portable Object Adaptor*) racine.

Pour obtenir une référence d'objet vers ce service, vous devez utiliser la méthode `resolve_initial_references`. Elle renvoie un objet générique CORBA, une instance de la classe `org.omg.corba.Object`. Il convient d'employer le préfixe complet du package : si vous utilisez simplement `Object`, le compilateur supposera que vous faites référence à `java.lang.Object`.

```
org.omg.CORBA.Object object  
= orb.resolve_initial_references("NameService");
```

Ensuite, convertissez cette référence en référence `NamingContext` pour que vous puissiez invoquer les méthodes de l'interface `NamingContext`. Avec RMI, il suffit de modifier le type de cette référence. Cependant, avec CORBA, il n'est pas possible de modifier simplement le type d'une référence.

```
NamingContext namingContext  
= (NamingContext)object; // ERREUR
```

Il faut donc avoir recours à la méthode `narrow` de la classe d'aide de l'interface cible.

```
NamingContext namingContext  
= NamingContextHelper.narrow(object);
```

ATTENTION

Vous réussirez *parfois* à transformer une référence d'objet CORBA en sous-type. En effet, un grand nombre de références `org.omg.CORBA.Object` pointent déjà sur des objets qui implémentent l'interface appropriée. Mais une référence d'objet peut également renfermer une référence vers un autre objet qui implémentera à son tour l'interface. Comme vous ne disposez d'aucun moyen pour déterminer comment les objets de stub ont été générés, vous devriez toujours utiliser la méthode `narrow` pour convertir une référence d'objet CORBA dans un sous-type.

Maintenant que vous connaissez le contexte de définition des noms, vous pouvez vous en servir pour localiser l'objet déclaré par le serveur. Le contexte de noms associe des noms avec les objets du serveur. Ces noms sont en fait des séquences imbriquées de *composants de noms*. Vous pouvez tirer profit des niveaux des séquences imbriquées pour organiser une hiérarchie de noms, un peu comme des répertoires dans un système de fichiers.

Un composant de nom contient un *identificateur* (ID) et un *type*. L'ID correspond au nom du composant et doit être unique parmi tous les noms du composant parent. Les types des composants ne sont pas standardisés. Nous utilisons "Context" comme noms de composants comprenant des noms imbriqués, et "Object" pour les noms d'objets.

Dans notre exemple, le programme de serveur a placé l'objet `EnvImpl` dans le nom défini par la séquence suivante :

```
(id="corejava", kind="Context"), (id="Env", kind="Object")
```

Nous en obtenons une référence à distance en construisant un tableau de composants de noms et en le passant à la méthode `resolve` de l'interface `NamingContext`.

```

NameComponent[] path =
{
    new NameComponent("corejava", "Context"),
    new NameComponent("Env", "Object")
};
org.omg.CORBA.Object envObj = namingContext.resolve(path);

```

Une fois de plus, nous devons restreindre la référence d'objet résultante :

```
Env env = EnvHelper.narrow(envObj);
```

Nous sommes maintenant prêts à appeler la méthode distante :

```
String value = env.getenv("PATH");
```

Vous trouverez le code complet dans l'Exemple 4.18.

Cet exemple montre les étapes à suivre pour un programme de client typique :

1. Démarrez l'ORB.
2. Localisez le service de noms en retrouvant une référence de départ à "NameService" et en la restreignant à une référence `NamingContext`.
3. Localisez l'objet dont vous souhaitez appeler les méthodes en assemblant son nom et en appelant la méthode `resolve` du `NamingContext`.
4. Restreignez l'objet retourné en fonction d'un type donné et invoquez vos méthodes.

Pour tester réellement ce programme, suivez les étapes suivantes. Les instructions C++ dépendent de l'ORB. Nos instructions concernent `omniORB`. Modifiez-les si vous utilisez un autre ORB :

1. Compilez le fichier IDL, en utilisant à la fois les compilateurs C++ et IDL :

```

omniidl -bcxx Env.idl
idlj Env.idl

```

2. Compilez le programme du serveur en C++. Les instructions de compilation dépendent de l'ORB. Par exemple, avec `omniORB` sous Linux, vous utiliseriez :

```

g++
-o EnvServer
-D__x86__ -D__linux__ -D__OSVERSION__=2
-I/usr/local/include/omniORB4
EnvServer.cpp EnvSK.cc
-lomniORB4 -lomnithread -lpthread

```

Pour connaître les éléments nécessaires à votre ORB particulier, compilez l'un des programmes d'exemple fournis avec votre installation et apportez les modifications nécessaires pour compiler votre propre programme.

3. Compilez le programme du client Java.
4. Démarrez le service d'identification de noms sur le serveur. Vous pouvez utiliser le programme `orbd` livré avec le JDK ou le service d'identification de votre ORB (par exemple `omniNames` si vous utilisez `omniORB`). Ce service reste actif jusqu'à ce que vous le fermiez.

278 Au cœur de Java 2 - Fonctions avancées

Pour lancer orbd, exécutez

```
orbd -ORBInitialPort 2809 &
```

Pour lancer omniNames, exécutez

```
omniNames -ORBsupportBootstrapAgent 1 &
```

5. Démarrez le serveur :

```
./EnvServer -ORBInitRef NameService=corbaname::localhost:2809 &
```

De même, ce serveur reste actif jusqu'à ce que vous le fermiez.

6. Lancez le client :

```
java EnvClient -ORBInitialPort 2809
```

Il devrait renvoyer le PATH du processus du serveur.

Si le serveur se trouve sur une machine distante ou si le port initial du serveur ORB ne correspond pas au port par défaut IDL Java (900), vous devez définir les propriétés `ORBInitialHost` et `ORBInitialPort`. Par exemple, omniORB utilise le port 2809. Avec orbd, nous avons aussi choisi le port 2809 car nous avons besoin de privilèges racine pour lancer un service sur un port inférieur à 1024 sous UNIX/Linux.

Il existe deux méthodes pour définir ces propriétés. Vous pouvez définir les propriétés système

```
org.omg.CORBA.ORBInitialHost  
org.omg.CORBA.ORBInitialPort
```

par exemple, en lançant l'interpréteur java avec l'option `-D`. Ces valeurs peuvent aussi être définies sur la ligne de commande :

```
java EnvClient -ORBInitialHost warthog -ORBInitialPort 2809
```

Les paramètres de la ligne de commande sont passés à l'ORB par l'appel :

```
ORB orb = ORB.init(args, null);
```

En principe, votre vendeur d'ORB devrait vous indiquer très clairement comment fonctionne son processus de démarrage. En pratique, il nous est apparu que les vendeurs partent tranquillement de l'hypothèse que vous ne serez jamais amené à mélanger leur précieux ORB avec celui d'un autre vendeur, et par conséquent ils ont tendance à laisser ces informations de côté. Si votre client n'arrive pas à trouver le service d'identification de noms, essayez de forcer les ports initiaux du serveur et du client à la même valeur.

ASTUCE

Si vous rencontrez des problèmes pour vous connecter au service de noms, affichez la liste des services initiaux que votre ORB peut localiser.

```
public class ListServices  
{  
    public static void main(String args[]) throws Exception  
    {  
        ORB orb = ORB.init(args, null);  
        String[] services = orb.list_initial_services();  
    }  
}
```

```

        for (int i = 0; i < services.length; i++)
            System.out.println(services[i]);
    }
}

```

Avec certains ORB, NameService ne fait pas partie des services répertoriés, quelles que soient les modifications que vous pourrez apporter à la configuration. Dans ce cas, vous devriez passer au plan B et localiser le serveur en passant par son IOR (*Interoperable Object Reference*). Consultez l'encart suivant pour plus d'informations.

Dans cette section, vous avez appris à vous connecter à un serveur implémenté en C++. Nous pensons qu'il s'agit là d'un scénario particulièrement utile. Vous pouvez transformer d'anciens services en objets CORBA et y accéder à partir de vos programmes Java 2, sans avoir à mettre en œuvre le moindre programme supplémentaire au niveau du client. Dans la prochaine section, nous aborderons le scénario opposé, dans lequel le serveur est implémenté en Java et le client, en C++.

Exemple 4.18 : *EnvClient.java*

```

import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class EnvClient
{
    public static void main(String args[])
    {
        try
        {
            ORB orb = ORB.init(args, null);
            org.omg.CORBA.Object namingContextObj
                = orb.resolve_initial_references("NameService");
            NamingContext namingContext
                = NamingContextHelper.narrow(namingContextObj);

            NameComponent[] path =
            {
                new NameComponent("corejava", "Context"),
                new NameComponent("Env", "Object")
            };
            org.omg.CORBA.Object envObj
                = namingContext.resolve(path);
            Env env = EnvHelper.narrow(envObj);
            System.out.println(env.getenv("PATH"));
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}

```

Exemple 4.19 : *EnvServer.cpp*

```

#include <iostream>
#include <cstdlib>

#include "Env.hh"

```

280 *Au cœur de Java 2 - Fonctions avancées*

```
using namespace std;

class EnvImpl :
    public POA_Env,
    public PortableServer::RefCountServantBase
{
public:
    virtual char* getenv(const char *name);
};

char* EnvImpl::getenv(const char *name)
{
    char* value = std::getenv(name);
    return CORBA::string_dup(value);
}

static void bindObjectToName(CORBA::ORB_ptr orb,
    const char name[], CORBA::Object_ptr objref)
{
    CosNaming::NamingContext_var rootContext;

    try
    {
        // Obtient une référence au contexte racine du service Name :
        CORBA::Object_var obj;
        obj = orb->resolve_initial_references("NameService");

        // Restreint la référence renvoyée
        rootContext = CosNaming::NamingContext::_narrow(obj);
        if(CORBA::is_nil(rootContext))
        {
            cerr << "Echec dans la restriction du contexte de nom." << endl;
            return;
        }
    }
    catch (CORBA::ORB::InvalidName& ex)
    {
        //Ceci ne devrait pas arriver !
        cerr << "Le service n'existe pas." << endl;
        return;
    }

    try
    {
        CosNaming::Name contextName;
        contextName.length(1);
        contextName[0].id = (const char*) "corejava";
        contextName[0].kind = (const char*) "Context";

        CosNaming::NamingContext_var corejavaContext;
        try
        {
            // Lie le contexte à la racine :
            corejavaContext
                = rootContext->bind_new_context(contextName);
        }
        catch (CosNaming::NamingContext::AlreadyBound& ex)
        {
            // Si le contexte existe déjà, cette exception sera déclenchée.

```

```

        // Dans ce cas, il suffit de résoudre le nom et d'assigner
        // testContext à l'objet retourné :
        CORBA::Object_var obj;
        obj = rootContext->resolve(contextName);
        corejavaContext
            = CosNaming::NamingContext::_narrow(obj);
        if( CORBA::is_nil(corejavaContext) )
        {
            cerr << "Echec dans la restriction du contexte de nom."
                << endl;
            return;
        }
    }

    // Lie objref à name dans le contexte :
    CosNaming::Name objectName;
    objectName.length(1);
    objectName[0].id = name;
    objectName[0].kind = (const char*) "Object";

    try
    {
        corejavaContext->bind(objectName, objref);
    }
    catch (CosNaming::NamingContext::AlreadyBound& ex)
    {
        corejavaContext->rebind(objectName, objref);
    }
}
catch (CORBA::COMM_FAILURE& ex)
{
    cerr
        << "Exception système détournée COMM_FAILURE - impossible de "
        << "contacter le service de noms." << endl;
}
catch (CORBA::SystemException&)
{
    cerr << "Déournement d'une CORBA::SystemException pendant "
        << "l'utilisation du service de nom." << endl;
}
}

int main(int argc, char *argv[])
{
    cout << "Création et initialisation de l'ORB..." << endl;

    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "omniORB4");

    CORBA::Object_var obj
        = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa
        = PortableServer::POA::_narrow(obj);
    poa->the_POAManager()->activate();

    EnvImpl* envImpl = new EnvImpl();
    poa->activate_object(envImpl);

    // Récupère une référence à l'objet et l'enregistre dans
    // le service de nom.

```


282 Au cœur de Java 2 - Fonctions avancées

```

obj = envImpl->_this();

cout << orb->object_to_string(obj) << endl;
cout << "Liaison des implémentations serveur à la base de registre..."
    << endl;
bindObjectName(orb, "Env", obj);
envImpl->_remove_ref();

cout << "Attente des invocations des clients..." << endl;
orb->run();

return 0;
}

```

Localiser des objets avec des IOR

Si vous n'arrivez pas à configurer votre serveur ORB et le service d'identification de noms pour que votre client puisse l'invoquer, vous pouvez néanmoins localiser des objets CORBA en utilisant une référence d'objet interopérable ou IOR (*Interoperable Object Reference*). Une IOR est une longue chaîne commençant par IOR: et constituée d'un grand nombre de chiffres hexadécimaux, comme :

```

IOR:012020201000000049444c3a4163636f756e743a312e3000010000000000 00004e000000010
100200f0000003231362e31352e3131322e31373900203504 20202e00000001504d4300000000010
00000049444c3a4163636f756e743a312e 30000e00000004a61636b20422e20517569636b00

```

Une IOR décrit un objet de manière unique. Par convention, plusieurs classes de serveurs affichent les IOR de tous les objets qu'elles enregistrent, pour permettre aux clients de les localiser. Vous pouvez alors coller l'IOR du serveur dans le programme client. Plus précisément, vous pouvez utiliser le code suivant :

```

String ref = "IOR:012020201000000049444c3a4163636f...";
// collez l'IOR à partir du serveur
org.omg.CORBA.Object object = orb.string_to_object(ref);

```

Vous devez ensuite restreindre l'objet retourné en fonction du type approprié, comme ceci :

```

Env env = EnvHelper.narrow(object);

```

ou

```

NamingContext context = NamingContextHelper.narrow(object);

```

Lorsque nous avons testé le code de ce livre, nous nous sommes servis de cette méthode avec succès pour connecter des clients à VisiBroker et omniORB (certaines versions de Java présentent des bugs qui les empêchent de reconnaître le service de nom omniORB).

org.omg.CORBA.ORB 1.2

- `static ORB init(String[] commandLineArgs, Properties orbConfigurationprops)`
Crée un nouvel ORB et l'initialise.
- `String[] list_initial_services()`
Renvoie une liste des services disponibles au démarrage, comme "NameService".

- `org.omg.CORBA.Object resolve_initial_references(String initialServiceName)`
Renvoie un objet qui prend en charge l'un des services de démarrage.
- `org.omg.CORBA.Object string_to_object(String ior)`
Localise l'objet correspondant à une IOR spécifiée.

org.omg.CosNaming.NamingContext 1.2

- `org.omg.CORBA.Object resolve(NameComponent[] name)`
Renvoie l'objet lié au nom spécifié.

org.omg.CosNaming.NameComponent 1.2

- `NameComponent(String componentId, String componentType)`
Construit un nouveau composant de nom.

Implémenter des serveurs CORBA

Si vous mettez en place une infrastructure CORBA, vous vous rendrez compte que Java est un bon langage d'implémentation pour des objets de serveur CORBA. Les liaisons de langage sont naturelles, et il est plus simple de construire des programmes de serveur robustes qu'avec C++. Cette section montre comment implémenter un serveur CORBA avec le langage de programmation Java.

Le programme d'exemple de cette section est comparable à celui de la section précédente. Nous fournissons un service pour chercher une propriété système d'une machine virtuelle Java. Voici la description IDL :

```
interface SysProp
{
    string getProperty(in string name);
};
```

Par exemple, notre programme client appelle le serveur de la manière suivante :

```
CORBA::String_var key = "java.vendor";
CORBA::String_var value = sysProp->getProperty(key);
```

Il en résulte une chaîne qui décrit le vendeur de la machine virtuelle Java qui exécute le programme du serveur. Nous n'examinerons pas les détails du programme client C++. Vous en trouverez le code dans l'Exemple 4.21.

Pour implémenter le serveur, vous devez exécuter le compilateur `idlj` avec l'option `-fall`. Par défaut, `idlj` ne peut créer que des stubs côté client :

```
idlj -fall SysProp.idl
```

Vous étendez ensuite la classe `SysPropPOA` que le compilateur `idlj` a généré à partir du fichier IDL. En voici l'implémentation :

```
class SysPropImpl extends _SysPropPOA
{
    public String getProperty(String key)
    {
        return System.getProperty(key);
    }
}
```

284 Au cœur de Java 2 - Fonctions avancées

INFO

Vous pouvez choisir n'importe quel nom pour la classe d'implémentation. Dans ce livre, nous observons la convention RMI et nous utilisons le suffixe `Impl` pour le nom de la classe d'implémentation. D'autres programmeurs se servent du suffixe `Servant` ou `_i`.

INFO

Si votre classe d'implémentation étend déjà une autre classe, vous ne pouvez pas étendre simultanément la classe de base d'implémentation. Dans ce cas, vous pouvez demander au compilateur `idlj` de créer une classe *égalité* (*tie*). Votre classe de serveur implémente alors l'interface *opérations* au lieu d'étendre la classe de base d'implémentation. Cependant, tous les objets de serveur doivent être créés au moyen de cette classe d'égalité. Pour plus de détails, consultez le cours sur les produits IDL à l'adresse suivante : <http://java.sun.com/j2se/5.0/docs/guide/rmi-iiop/toJava-PortableUG.html>.

Ecrivez ensuite un programme de serveur qui gère les tâches suivantes :

1. Démarrer l'ORB.
2. Localiser et activer le POA racine.
3. Créer une implémentation de serveur.
4. Utiliser le POA pour convertir la référence servant en référence d'objet CORBA (la classe d'implémentation du serveur étend `SysPropPOA` qui étend lui-même `org.omg.PortableServer.Servant`).
5. Afficher ses IOR (pour les clients affrontant des problèmes de service d'identification de noms).
6. Lier l'implémentation du serveur au service de noms.
7. Attendre les invocations des clients.

Vous trouverez le code complet dans l'Exemple 4.20. En voici les points essentiels.

Vous démarrez l'ORB comme pour un programme client :

```
ORB orb = ORB.init(args, null);
```

Activez ensuite le POA racine :

```
POA rootpoa
    = (POA)orb.resolve_initial_references("RootPOA");
rootpoa.the_POAManager().activate();
```

Construisez l'objet serveur et transformez-le en objet CORBA :

```
SysPropImpl impl = new SysPropImpl();
org.omg.CORBA.Object ref
    = rootpoa.servant_to_reference(impl);
```

Puis, trouvez l'IOR avec la méthode `object_to_string` et affichez-le :

```
System.out.println(orb.object_to_string(impl));
```

Vous obtenez alors une référence au service de noms exactement comme pour un programme client :

```
org.omg.CORBA.Object namingContextObj =
    orb.resolve_initial_references("NameService");
NamingContext namingContext
    = NamingContextHelper.narrow(namingContextObj);
```

Vous pouvez maintenant construire un nom pour l'objet. Ici, nous appelons l'objet SysProp :

```
NameComponent[] path =
{
    new NameComponent("SysProp", "Object")
};
```

Utilisez ensuite la méthode `rebind` pour lier l'objet au nom :

```
namingContext.rebind(path, impl);
Pour terminer, attendez les invocations du client.
orb.run();
```

Pour tester ce programme, suivez les étapes ci-après :

1. Compilez le fichier IDL en utilisant les compilateurs C++ et Java IDL.

```
omniidl -bcxx SysProp.idl
idlj -fall SysProp.idl
```

2. Compilez le programme du serveur.

```
javac SysPropServer.java
```

3. Compilez le programme du client C++. Sous Linux, utilisez la commande :

```
g++ -o SysPropClient -D__x86__ -D__linux__ -D__OSVERSION__=2
-I /usr/local/include/omniORB4/
SysPropClient.cpp SysPropSK.cc -lomniORB4 -lomnithread -lpthread
```

4. Lancez le service de noms `orbd` sur le serveur. Ce programme fait partie du JDK.

```
orbd - ORBInitialPort 2809 &
```

5. Démarrez le serveur :

```
java SysPropServer -ORBInitialPort 2809 &
```

Le serveur reste également actif jusqu'à ce que vous le fermiez.

6. Lancez le client :

```
./SysPropClient -ORBInitRef NameService=corbaname::localhost
```

Le vendeur de la machine virtuelle Java du serveur devrait s'afficher.

Vous savez maintenant comment utiliser CORBA pour connecter des clients et des serveurs écrits dans différents langages de programmation. Cela termine notre sujet sur CORBA. CORBA possède un certain nombre d'autres caractéristiques intéressantes, comme des invocations de méthodes dynamiques et certains services standard comme une gestion des transactions. Nous vous renvoyons à l'ouvrage *CampusPress Référence Développer avec CORBA en Java et C++* pour une présentation approfondie des caractéristiques avancées de CORBA.

Exemple 4.20 : SysPropServer.java

```
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;

class SysPropImpl extends _SysPropPOA
{
```

286 *Au cœur de Java 2 - Fonctions avancées*

```

    public String getProperty(String key)
    {
        return System.getProperty(key);
    }
}

public class SysPropServer
{
    public static void main(String args[])
    {
        try
        {
            System.out.println(
                "Création et initialisation de l'ORB...");

            ORB orb = ORB.init(args, null);

            System.out.println(
                "Enregistrement de l'implémentation du serveur avec l'ORB...");

            POA rootpoa
                = (POA) orb.resolve_initial_references("RootPOA");
            rootpoa.the_POAManager().activate();

            SysPropImpl impl = new SysPropImpl();
            org.omg.CORBA.Object ref
                = rootpoa.servant_to_string(ref));
            System.out.println(orb.object_to_string(ref));

            org.omg.CORBA.Object namingContextObj =
                orb.resolve_initial_references("NameService");
            NamingContext namingContext
                = NamingContextHelper.narrow(namingContextObj);

            NameComponent[] path =
                {
                    new NameComponent("SysProp", "Object")
                };

            System.out.println(
                "Liaison de l'implémentation du serveur au service de définition
                ➡ de noms...");

            namingContext.rebind(path, impl);

            System.out.println(
                "Attente des invocations des clients...");
            orb.run();

        }
        catch (Exception e)
        {
            e.printStackTrace(System.out);
        }
    }
}

```

Exemple 4.21 : SysPropClient.cpp

```
#include <iostream>

#include "SysProp.hh"

using namespace std;

CORBA::Object_ptr getObjectReference(CORBA::ORB_ptr orb,
    const char serviceName[])
{
    CosNaming::NamingContext_var rootContext;

    try
    {
        // Obtient une référence au contexte racine du service Name:
        CORBA::Object_var initServ;
        initServ = orb->resolve_initial_references("NameService");

        // Restreint l'objet renvoyé par resolve_initial_references()
        // à un objet CosNaming::NamingContext object
        rootContext = CosNaming::NamingContext::_narrow(initServ);
        if (CORBA::is_nil(rootContext))
        {
            cerr << "Echec dans la restriction du contexte de définition
                ↳ de noms." << endl;
            return CORBA::Object::_nil();
        }
    }
    catch(CORBA::ORB::InvalidName&)
    {
        cerr << "Le service de définition de noms n'existe pas." << endl;
        return CORBA::Object::_nil();
    }

    // Crée un objet name, contenant le nom corejava/SysProp:
    CosNaming::Name name;
    name.length(1);

    name[0].id = serviceName;
    name[0].kind = "Object";

    CORBA::Object_ptr obj;
    try
    {
        // Résout le nom à une référence d'objet, et y assigne la référence
        // renvoyée à un CORBA::Object:
        obj = rootContext->resolve(name);
    }
    catch(CosNaming::NamingContext::NotFound&)
    {
        // Cette exception est déclenchée si l'un des composants
        // du chemin [contextes ou l'objet] n'est pas trouvé :
        cerr << "Contexte non trouvé." << endl;
        return CORBA::Object::_nil();
    }
    return obj;
}
```

288 Au cœur de Java 2 - Fonctions avancées

```

int main (int argc, char *argv[])
{
    CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv, "omniORB4");

    CORBA::Object_var obj = getObjectReference(orb, "SysProp");

    if (CORBA::is_nil(sysProp))
    {
        cerr << "Impossible d'invoquer une référence d'objet nulle."
              << endl;
        return 1;
    }

    CORBA::String_var key = "java.vendor";
    CORBA::String_var value = sysProp->getProperty(key);

    cerr << key << "=" << value << endl;

    return 0;
}

```

org.omg.CORBA.ORB 1.2

- `Void connect(org.omg.CORBA.Object obj)`
Connecte l'objet d'implémentation spécifié à cet ORB, en lui permettant de transmettre des appels aux méthodes des objets.
- `String object_to_string(org.omg.CORBA.Object obj)`
Renvoie la chaîne de l'IOR de l'objet spécifié.

org.omg.CosNaming.NamingContext 1.2

- `void bind(NameComponent[] name, org.omg.CORBA.Object obj)`
- `void rebind(NameComponent[] name, org.omg.CORBA.Object obj)`
Lie un objet à un nom. La méthode `bind` déclenche une exception `AlreadyBound` si l'objet a déjà été lié. La méthode `rebind` remplace n'importe quel lien existant.

Appels de méthode distante avec SOAP

Cette section est adaptée de l'ouvrage *Core JavaServer Faces*, de Geary et Horstmann (Sun Microsystems Press, 2004).

Ces dernières années, les *services Web* se sont confirmés comme une technologie populaire pour les appels de méthode distante. Au plan technique, un service Web possède deux composants :

- un serveur accessible à l'aide du protocole de transport SOAP (*Simple Object Access Protocol*) ;
- une description du service au format WDSL (*Web Service Description Language*).

SOAP est un protocole XML qui, à l'instar du IIOP de CORBA, permet d'invoquer des méthodes distantes. Vous pouvez programmer des clients et des serveurs CORBA sans rien savoir sur IIOP, de même que vous n'avez pas réellement besoin de connaître les détails de SOAP pour appeler un service Web.

WSDL est analogue à IDL, il décrit l'interface du service Web, à savoir les méthodes pouvant être appelées ainsi que leurs types de paramètres et de retour. Nous ne verrons ici que l'implémentation d'un client qui se connecte à un service Web. Une autre personne a déjà préparé le fichier WSDL.

L'implémentation d'un service Web va au-delà de l'objet de cet ouvrage. Pour en savoir plus, consultez le Chapitre 8 du didacticiel du J2EE à l'adresse <http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html>.

Afin de simplifier l'étude des services Web, nous étudierons un exemple concret : le service Web d'Amazon, décrit à l'adresse <http://www.amazon.com/gp/aws/landing.html>. Ce service Web permet à un programmeur d'interagir avec le système Amazon à diverses fins. Vous pouvez, par exemple, obtenir des listings de tous les ouvrages d'un auteur donné ou ayant tel ou tel titre ou bien encore remplir un chariot et commander. Amazon met ce service à la disposition des sociétés qui souhaitent vendre des articles à leurs clients, le système Amazon jouant le rôle de système d'arrière-guichet. Pour exécuter notre programme d'exemple, vous devrez vous identifier chez Amazon et obtenir un jeton de développeur gratuit assurant la connexion au service.

Vous pouvez également adapter la technique décrite dans cette section à tout autre service Web. Le site <http://www.xmethods.com> répertorie de nombreux services Web libres que vous aurez tout loisir de tester.

Premier attrait des services Web : ils sont indépendants du langage. Nous accédons aux services Web d'Amazon en utilisant des programmes Java, mais d'autres développeurs pourront employer le C++ ou Visual Basic. Le descripteur WSDL identifie les services indépendamment du langage. Le WSDL des services Web d'Amazon, par exemple (<http://soap.amazon.com/schemas3/AmazonWebServices.wsdl>), décrit une opération AuthorSearchRequest comme suit :

```
<operation name="AuthorSearchRequest">
  <input message="typens:AuthorSearchRequest" />
  <output message="typens:AuthorSearchResponse" />
</operation>
. . .
<message name="AuthorSearchRequest">
  <part name="AuthorSearchRequest" type="typens:AuthorRequest" />
</message>
<message name="AuthorSearchResponse">
  <part name="return" type="typens:ProductInfo" />
</message>
```

Partout ailleurs, il définit les types de données. Voici la définition d'AuthorRequest :

```
<xsd:complexType name="AuthorRequest">
  <xsd:all>
    <xsd:element name="author" type="xsd:string"/>
    <xsd:element name="page" type="xsd:string"/>
    <xsd:element name="mode" type="xsd:string"/>
    <xsd:element name="tag" type="xsd:string"/>
    <xsd:element name="type" type="xsd:string"/>
    <xsd:element name="devtag" type="xsd:string"/>
    <xsd:element name="sort" type="xsd:string" minOccurs="0"/>
    <xsd:element name="locale" type="xsd:string" minOccurs="0"/>
    <xsd:element name="keywords" type="xsd:string" minOccurs="0"/>
    <xsd:element name="price" type="xsd:string" minOccurs="0"/>
  </xsd:all>
</xsd:complexType>
```


290 Au cœur de Java 2 - Fonctions avancées

Une fois cette description traduite en Java, le type `AuthorRequest` devient une classe :

```
public class AuthorRequest
{
    public AuthorRequest(String author, String page, String mode,
        String tag, String type,
        String devtag, String sort, String locale, String keyword,
        String price) { . . . }
    public String getAuthor() { . . . }
    public void setAuthor(String newValue) { . . . }
    public String getPage() { . . . }
    public void setPage(String) { . . . }
    . . .
}
```

Pour appeler le service de recherche, construisez un objet `AuthorRequest` et appelez l'`authorSearchRequest` d'un objet "port" :

```
AmazonSearchPort port = (AmazonSearchPort) (
    new AmazonSearchService_Impl().getAmazonSearchPort());
AuthorRequest request = new AuthorRequest(name, "1", "books", "",
    "lite", "", token, "", "", "");
ProductInfo response = port.authorSearchRequest(request);
```

L'objet de port traduit l'objet Java en message SOAP, le passe au serveur d'Amazon et traduit le message renvoyé en objet `ProductInfo`. Les classes du port sont générées automatiquement.

INFO

Le fichier WSDL ne spécifie pas ce *que fait* le service. Il n'indique que le paramètre et les types de retours.

Téléchargez le pack développeur des services Web Java (JWSDP) à l'adresse <http://java.sun.com/webservices/webservicespack.html> et installez-le.

Pour générer les classes Java requises, placez un fichier `config.xml` ayant le contenu suivant dans le répertoire du programme client :

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
    <wsdl
        location="http://soap.amazon.com/schemas3/
            AmazonWebServices.wsdl"
        packageName="com.amazon" />
    </wsdl>
</configuration>
```

Exécutez ensuite ces commandes :

```
jwsdp/jaxrpc/bin/wscompile.sh -import config.xml
jwsdp/jaxrpc/bin/wscompile.sh -gen -keep config.xml
```

Ici, `jwsdp` correspond au répertoire dans lequel vous avez installé le JWSDP, par exemple `/usr/local/jwsdp-1.4` ou `c:\jwsdp-1.4` (les utilisateurs de Windows doivent bien sûr utiliser `\` au lieu de `/` et `.bat` au lieu de `.sh`).

La première commande importe le fichier WSDL depuis un emplacement donné. La seconde génère les classes nécessaires au client. L'option `-keep` permet de conserver les fichiers source. Ceci n'est pas strictement nécessaire, mais il peut être intéressant de les étudier. En conséquence de ces commandes, un grand nombre de classes est créé dans le répertoire spécifié (ici, `com/amazon`). Etudiez certaines d'entre elles, en particulier `com/amazon/AuthorRequest.java` et `com/amazon/Details.java`. Nous les utilisons dans l'application d'exemple.

Notre application d'exemple (Exemple 4.22) est assez simple. L'utilisateur indique un nom d'auteur et clique sur le bouton Search. Nous n'affichons que la première page de la réponse (voir Figure 4.11). Elle montre que le service Web a réussi. Nous laissons le soin au lecteur d'étendre les fonctionnalités de cette application.

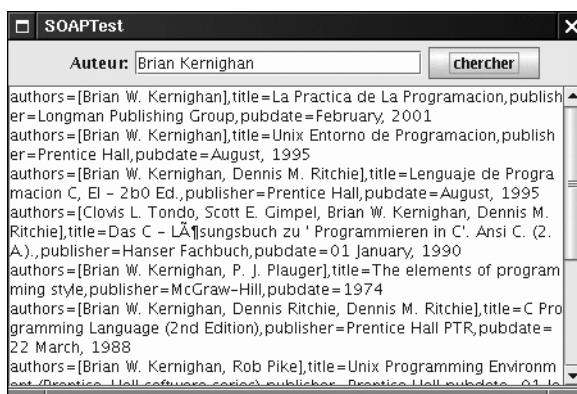
Pour la compiler, ajoutez le fichier de bibliothèque `jwsdp/jaxrpc/lib/jaxrpc-impl.jar` à votre chemin de classe.

Pour exécuter cette application, votre fichier de classe doit contenir ceci :

```
. (le répertoire courant)
jwsdp/jaxrpc/lib/jaxrpc-api.jar
jwsdp/jaxrpc/lib/jaxrpc-impl.jar
jwsdp/jaxrpc/lib/jaxrpc-spi.jar
jwsdp/jwsdp-shared/lib/activation.jar
jwsdp/jwsdp-shared/lib/mail.jar
jwsdp/saaj/lib/saaj-api.jar
jwsdp/saaj/lib/saaj-impl.jar
```

Figure 4.11

Connexion
à un service Web.



Si vous compilez le programme et que vous le compilez depuis la ligne de commande, utilisez un script de shell pour automatiser cette tâche.

Cet exemple montre que l'appel d'un service Web est quasiment identique à la réalisation d'un autre appel de méthode distante. Le programmeur appelle une méthode locale sur un objet proxy et le proxy se connecte à un serveur. Les services Web apparaissant un peu partout, c'est là une technologie tout à fait intéressante pour les programmeurs d'application. Les services Web seront de plus en plus faciles à utiliser à mesure que les outils et la bibliothèque s'intégreront dans les prochaines versions du JDK.

292 *Au cœur de Java 2 - Fonctions avancées*

Exemple 4.22 : SOAPTest.java

```
import com.amazon.*;
import java.awt.*;
import java.awt.event.*;
import java.rmi.*;
import java.util.*;
import javax.swing.*;

/**
 * Le client du programme d'entrepasage.
 */
public class SOAPTest
{
    public static void main(String[] args)
    {
        JFrame frame = new SOAPTestFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Une fenêtre permettant de sélectionner l'auteur et d'afficher
 * la réponse du serveur.
 */
class SOAPTestFrame extends JFrame
{
    public SOAPTestFrame()
    {
        setTitle("SOAPTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        JPanel panel = new JPanel();

        panel.add(new JLabel("Auteur :"));
        author = new JTextField(20);
        panel.add(author);

        JButton searchButton = new JButton("Rechercher");
        panel.add(searchButton);
        searchButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    result.setText("Veuillez patienter...");
                    new Thread(new
                        Runnable()
                        {
                            public void run()
                            {
                                String name = author.getText();
                                String books = searchByAuthor(name);
                                result.setText(books);
                            }
                        }
                    ).start();
                }
            }
        );
    }
}
```

```

        result = new JTextArea();
        result.setLineWrap(true);
        result.setEditable(false);

        add(panel, BorderLayout.NORTH);
        add(new JScrollPane(result), BorderLayout.CENTER);
    }

    /**
     * Appelle le service Web d'Amazon pour trouver les titres
     * correspondant à l'auteur.
     * @param name le nom de l'auteur
     * @return une description des titres concordants
     */
    private String searchByAuthor(String name)
    {
        try
        {
            AmazonSearchPort port = (AmazonSearchPort)
                (new AmazonSearchService_Impl().getAmazonSearchPort());

            AuthorRequest request
                = new AuthorRequest(name, "1", "books", "", "lite", "",
                    token, "", "", "");
            ProductInfo response = port.authorSearchRequest(request);

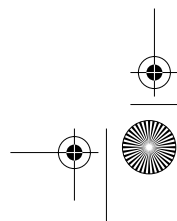
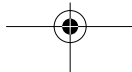
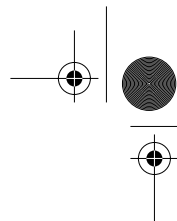
            Details[] details = response.getDetails();
            StringBuilder r = new StringBuilder();
            for (Details d : details)
            {
                r.append("authors=");
                String[] authors = d.getAuthors();
                if (authors == null) r.append("[]");
                else r.append(Arrays.asList(d.getAuthors()));
                r.append(",title=");
                r.append(d.getProductName());
                r.append(",publisher=");
                r.append(d.getManufacturer());
                r.append(",pubdate=");
                r.append(d.getReleaseDate());
                r.append("\n");
            }
            return r.toString();
        }
        catch (RemoteException e)
        {
            return "Exception : " + e;
        }
    }

    private static final int DEFAULT_WIDTH = 450;
    private static final int DEFAULT_HEIGHT = 300;

    private static final String token = "votre jeton vient ici";

    private JTextField author;
    private JTextArea result;
}

```



5

Swing

Au sommaire de ce chapitre

- ✓ Listes
- ✓ Arbres
- ✓ Tableaux
- ✓ Composants de texte stylisés
- ✓ Indicateurs de progression
- ✓ Organismes de composants

Dans ce chapitre, nous continuons notre étude des outils de l'interface utilisateur Swing du Volume 1. Swing est en fait un ensemble d'outils très vaste et très complexe dont nous n'avons abordé que quelques-uns des composants les plus importants au cours du Volume 1. Ce qui nous laisse pour ce volume l'étude de trois composants, les listes, les arbres et les tableaux, dont l'exploration occupera la majeure partie de ce chapitre. Les composants de texte stylisés, en particulier le HTML, sont encore plus complexes et nous vous indiquerons comment les mettre en pratique. Nous terminerons ce chapitre par l'étude des derniers composants Swing, comme les onglets et les panneaux de bureau avec des fenêtres internes.

Listes

Si vous souhaitez présenter un ensemble de choix à un utilisateur et qu'un ensemble de boutons ou de cases à cocher prenne trop de place, vous pouvez utiliser un menu déroulant ou une liste. Les menus déroulants ont été traités dans le Volume 1 du fait de leur relative simplicité. Le composant `JList` possède bien d'autres fonctions, et sa conception est semblable à celle des composants arbre et tableau. C'est la raison pour laquelle nous commencerons par lui dans notre discussion sur les composants Swing complexes.

Bien entendu, vous pouvez disposer de listes de chaînes, mais également de listes d'objets arbitraires et posséder un contrôle complet sur leur affichage. L'architecture interne du composant de la liste

296 Au cœur de Java 2 - Fonctions avancées

qui permet cette généralité est plutôt agréable mais est inutile au programmeur qui souhaite simplement utiliser le composant. Vous découvrirez également que le contrôle sur la liste est assez étrange à utiliser dans certains cas car vous devrez manipuler quelques instruments qui permettent l'application des cas généraux. Nous étudierons les cas les plus simples et les plus communs, une liste de chaînes, puis nous vous proposerons un exemple plus complexe montrant la flexibilité du composant de liste.

Le composant JList

Le composant JList est un ensemble de cases à cocher ou de boutons, mais les éléments sont placés dans une seule boîte. Vous pouvez les sélectionner en cliquant dessus (et non en cliquant sur les boutons). Si vous autorisez la sélection multiple dans une liste, l'utilisateur aura la possibilité de sélectionner n'importe quelle combinaison des éléments de la liste.

La Figure 5.1 présente un exemple de situation possible. L'utilisateur a ainsi la possibilité de sélectionner les attributs du renard : "quick" (rapide), "brown" (marron), "hungry" (affamé), "wild" (sauvage), "static" (statique), "private" (privé) et "final" (définitif).

Pour construire ce composant de liste, vous commencez par un tableau de chaînes, puis vous passez le tableau au constructeur JList :

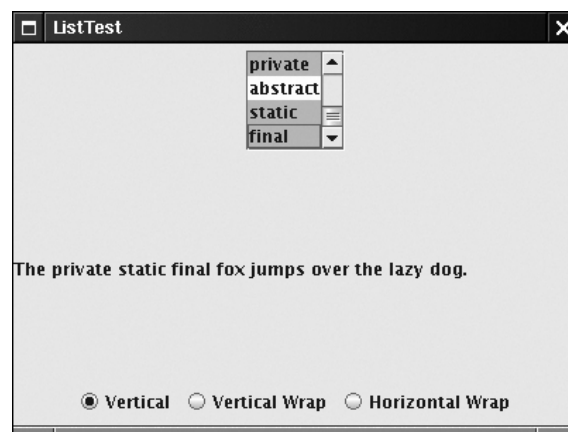
```
String[] words= { "quick", "brown", "hungry", "wild", ... };
JList wordList = new JList(words);
```

Vous pouvez également utiliser un tableau anonyme :

```
JList wordList = new JList(new String[]
    {"quick", "brown", "hungry", "wild", ... });
```

Figure 5.1

Une liste.



Les listes ne défilent pas automatiquement. Pour y parvenir, vous devez l'insérer dans un volet de défilement :

```
JScrollPane scrollPane = new JScrollPane(wordList);
```

Vous devez ensuite ajouter le volet de défilement, et non la liste, dans le panneau qui l'entoure.

Nous devons admettre que la séparation de l'affichage de la liste et que le mécanisme de défilement sont assez plaisants en théorie, mais qu'ils se révèlent peu pratiques. Pour l'essentiel, il était nécessaire que toutes les listes que nous avons rencontrées défilent. Pourtant, il semble inutile d'obliger les programmeurs à devoir écrire la situation par défaut, simplement pour qu'ils puissent apprécier cette fonction.

Par défaut, le composant de liste affiche huit éléments ; utilisez la méthode `setVisibleRowCount` pour modifier cette valeur :

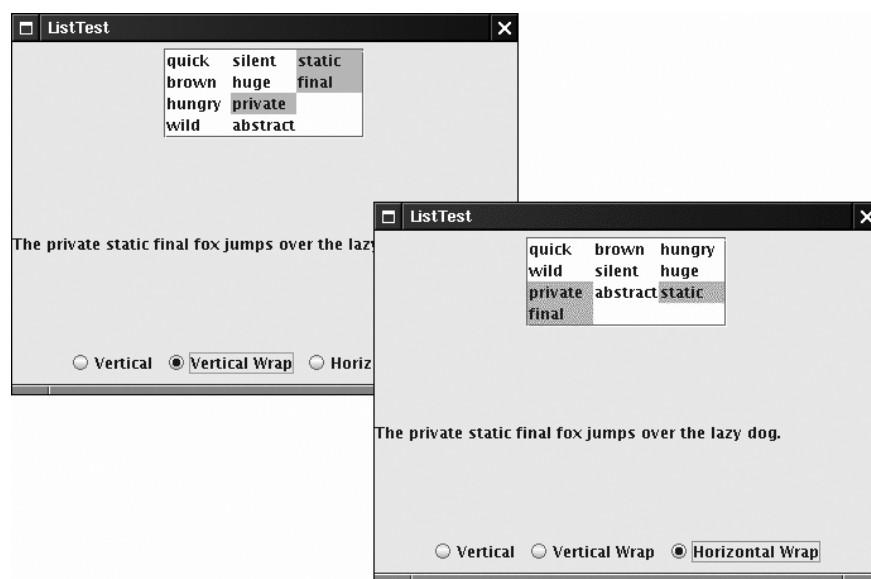
```
wordList.setVisibleRowCount(4); // affiche 4 choix
```

Vous pouvez définir l'orientation de la disposition sur l'une de ces trois valeurs :

- `JList.VERTICAL` (le paramètre par défaut). Agence verticalement tous les éléments.
- `JList.VERTICAL_WRAP`. Démarre de nouvelles colonnes s'il y a plus d'éléments que le nombre de lignes visibles (voir Figure 5.2).
- `JList.HORIZONTAL_WRAP`. Démarre de nouvelles colonnes s'il y a plus d'éléments que le nombre de lignes visibles, mais les remplit horizontalement. Constatez le placement des mots "quick", "brown" et "hungry" à la Figure 5.2 pour voir la différence entre le positionnement horizontal et vertical.

Figure 5.2

Listes avec placement vertical et horizontal.



Par défaut, un utilisateur peut sélectionner plusieurs éléments, à condition qu'il ait une certaine connaissance du fonctionnement de la souris. Pour ajouter plusieurs éléments à une sélection, appuyez sur la touche Ctrl, tout en cliquant sur chaque élément. Pour sélectionner une plage contiguë d'éléments, cliquez sur le premier d'entre eux, puis maintenez la touche Maj enfoncée et cliquez sur le dernier.

298 Au cœur de Java 2 - Fonctions avancées

Vous pouvez également restreindre les possibilités de l'utilisateur grâce à la méthode `setSelectionMode` :

```
wordList.setSelectionMode
(ListSelectionMode.SINGLE_SELECTION);
// sélectionne un choix à la fois
wordList.setSelectionMode
(ListSelectionMode.SINGLE_INTERVAL_SELECTION);
// sélectionne un choix ou une plage de choix
```

Vous vous souvenez peut-être, comme vous l'avez vu au Volume 1, que les composants de base de l'interface utilisateur envoient des événements d'action lorsque l'utilisateur les active. Les zones de liste utilisent par contre un mécanisme de notification différent. Plutôt que d'écouter des événements d'action, vous devez écouter des événements de sélection de liste. Ajoutez donc un écouteur de sélection de liste au composant de la liste et implémentez la méthode suivante dans l'écouteur :

```
public void valueChanged(ListSelectionEvent evt)
```

Lorsque l'utilisateur sélectionne des éléments, toute une série d'événements de sélection de liste sont générés. Supposons, par exemple, que l'utilisateur clique sur un nouvel élément. Lorsqu'il enfonce le bouton de la souris, un événement signale un changement de la sélection. Il s'agit d'un événement de transition ; l'appel :

```
event.isAdjusting()
```

renvoie `true` si la sélection n'est pas encore terminée. Puis, lorsque l'utilisateur relâche le bouton de la souris, un autre événement, cette fois-ci accompagné de `isAdjusting`, renvoie `false`. Si les événements de transition ne vous intéressent pas, vous pouvez attendre l'événement pour lequel `isAdjusting` vaut `false`. Toutefois, si vous souhaitez fournir à l'utilisateur un retour immédiat, dès qu'il clique sur la souris, vous devez traiter tous les événements.

Une fois averti de la survenue d'un événement, vous voudrez connaître les éléments sélectionnés. La méthode `getSelectedValues` renvoie un *tableau d'objets* contenant tous les éléments sélectionnés.

Vous devez diffuser *chaque* élément de tableau à une chaîne :

```
Object[] values = list.getSelectedValues();
for (Object value : values)
    faire quelque chose avec (String) value;
```

ATTENTION

Vous ne pouvez pas diffuser la valeur de retour de `getSelectedValues` depuis un tableau `Object[]` dans un tableau `String[]`. La valeur de retour n'a pas été créée sous forme de tableau de chaînes, mais sous forme de tableau d'objets, chacun se trouvant être une chaîne. Si vous souhaitez traiter la valeur de retour sous forme de tableau de chaînes, vous pouvez utiliser le code suivant :

```
int length = values.length;
String[] words = new String[length];
System.arraycopy(values, 0, words, 0, length);
```

Si votre liste ne permet pas d'effectuer des sélections multiples, vous pouvez appeler la méthode `getSelectedValue`. Elle renvoie la première valeur sélectionnée (que vous savez être la seule valeur lorsque les sélections multiples sont interdites).

```
String value = (String) list.getSelectedValue();
```

INFO

Les composants de liste ne réagissent pas aux doubles-clics d'une souris. Comme le souhaitent les concepteurs de Swing, vous utilisez une liste pour sélectionner un élément, puis vous devez cliquer sur un bouton pour que quelque chose se produise. Toutefois, certaines interfaces utilisateur permettent à un utilisateur de double-cliquer sur une liste pour signaler un choix d'un élément de la liste et l'acceptation d'une action par défaut. Si vous souhaitez implémenter ce comportement, vous devrez ajouter un écouteur de souris à la zone de liste, puis saisir l'événement de souris comme suit :

```
public void mouseClicked(MouseEvent evt)
{
    if (evt.getClickCount() == 2)
    {
        JList source = (JList)evt.getSource();
        Object[] selection = source.getSelectedValues();
        doAction(selection);
    }
}
```

L'Exemple 5.1 représente le code du programme qui présente une zone de liste remplie de chaînes. Vous remarquerez que la méthode `valueChanged` élabore la chaîne de message à partir des éléments sélectionnés.

Exemple 5.1 : *ListTest.java*

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Ce programme présente une liste de chaînes fixes.
 */
public class ListTest
{
    public static void main(String[] args)
    {
        JFrame frame = new ListFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc contient une liste de mots et une étiquette présentant une
 * phrase constituée des mots choisis. Vous remarquerez que vous pouvez
 * sélectionner plusieurs mots avec Ctrl+clic et Maj+clic.
 */
class ListFrame extends JFrame
{
    public ListFrame()
    {
        setTitle("ListTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
    }
}
```

300 *Au cœur de Java 2 - Fonctions avancées*

```
String[] words =
{
    "quick", "brown", "hungry", "wild", "silent",
    "huge", "private", "abstract", "static", "final"
};

wordList = new JList(words);
wordList.setVisibleRowCount(4);
JScrollPane scrollPane = new JScrollPane(wordList);

listPanel = new JPanel();
listPanel.add(scrollPane);
wordList.addListSelectionListener(new
    ListSelectionListener()
    {
        public void valueChanged(ListSelectionEvent event)
        {
            Object[] values = wordList.getSelectedValues();

            StringBuilder text = new StringBuilder(prefix);
            for (int i = 0; i < values.length; i++)
            {
                String word = (String) values[i];
                text.append(word);
                text.append(" ");
            }
            text.append(suffix);

            label.setText(text.toString());
        }
    });

buttonPanel = new JPanel();
group = new ButtonGroup();
makeButton("Vertical", JList.VERTICAL);
makeButton("Vertical Wrap", JList.VERTICAL_WRAP);
makeButton("Horizontal Wrap", JList.HORIZONTAL_WRAP);

add(listPanel, BorderLayout.NORTH);
label = new JLabel(prefix + suffix);
add(label, BorderLayout.CENTER);
add(buttonPanel, BorderLayout.SOUTH);
}

/**
 * Crée un bouton pour définir l'orientation de la liste.
 * @param label L'intitulé du bouton
 * @param orientation L'orientation de la liste
 */
private void makeButton(String label, final int orientation)
{
    JRadioButton button = new JRadioButton(label);
    buttonPanel.add(button);
    if (group.getButtonCount() == 0) button.setSelected(true);
    group.add(button);
    button.addActionListener(new
        ActionListener()
        {
            public void actionPerformed(ActionEvent event)
```

```

        {
            wordList.setLayoutOrientation(orientation);
            listPanel.revalidate();
        }
    });
}

private static final int DEFAULT_WIDTH = 400;
private static final int DEFAULT_HEIGHT = 300;
private JPanel listPanel;
private JList wordList;
private JLabel label;
private JPanel buttonPanel;
private ButtonGroup group;
private String prefix = "The ";
private String suffix = "fox jumps over the lazy dog.";
}

```

javax.swing.JList 1.2

- **JList(Object[] items)**
Construit une liste qui affiche ces éléments.
- **int getVisibleRowCount()**
- **void setVisibleRowCount(int c)**
Récupèrent ou définissent le nombre de lignes préférées dans la liste qui peuvent être affichées sans barre de défilement.
- **int getLayoutOrientation() 1.4**
- **void setLayoutOrientation() 1.4**
Récupèrent ou définissent l'orientation de l'élément.
Paramètres : orientation VERTICAL, VERTICAL_WRAP ou HORIZONTAL_WRAP
- **int getSelectionMode()**
- **void setSelectionMode(int mode)**
Récupèrent ou déterminent si les sélections d'un ou de plusieurs éléments sont autorisées.
Paramètres : mode entre SINGLE_SELECTION, SINGLE_INTERVAL_SELECTION, MULTIPLE_INTERVAL_SELECTION
- **void addListSelectionListener(ListSelectionListener listener)**
Ajoute à la liste un écouteur qui est averti à chaque changement de sélection.
- **Object[] getSelectedValues()**
Renvoie les valeurs sélectionnées ou un tableau vide si la sélection est vide.
- **Object getSelectedValue()**
Renvoie la première valeur sélectionnée ou null si la sélection est vide.

javax.swing.event.ListSelectionListener 1.2

- **void valueChanged(ListSelectionEvent e)**
Appelé en cas de changement de sélection de la liste.

Modèles de listes

Dans la section précédente, vous avez vu la méthode la plus habituelle pour utiliser un composant de liste :

- spécifier un ensemble de chaînes fixe pour l’afficher dans la liste ;
- ajouter une barre de défilement ;
- capturer les événements de sélection de liste.

Dans le reste de la section sur les listes, nous traiterons de situations plus complexes qui exigent un peu plus de finesse :

- de très longues listes ;
- des listes dont le contenu change ;
- des listes qui ne contiennent pas de chaînes.

Dans le premier exemple, nous avons construit un composant `JList` qui contenait une suite fixe de chaînes. Toutefois, la suite de choix d’une zone de liste n’est pas toujours fixe. Comment alors ajouter ou supprimer des éléments d’une zone de liste ? De manière assez surprenante, il n’existe pas de méthodes dans la classe `JList` pour y parvenir. Au lieu de cela, il vous faudra connaître un peu mieux la conception interne d’un composant de liste. Comme pour les composants de texte, le composant de liste utilise un motif de conception modèle-vue-contrôleur pour séparer l’apparence visuelle (une colonne d’éléments affichée d’une certaine manière) des données sous-jacentes (une suite d’objets).

La classe `JList` est responsable de l’apparence visuelle des données. Elle connaît en fait très peu la manière dont les données sont stockées. Elle peut seulement récupérer les données par un objet qui implémente l’interface `ListModel` :

```
public interface ListModel
{
    int getSize();
    Object getElementAt(int i);
    void addListDataListener(ListDataListener l);
    void removeListDataListener(ListDataListener l);
}
```

Par le biais de cette interface, `JList` peut obtenir le nombre des éléments et récupérer chacun d’entre eux. De même, l’objet `JList` peut s’ajouter sous forme *d’écouteur de données de liste*. Il est alors averti en cas de changement de la série d’éléments, afin qu’il puisse réactualiser la liste.

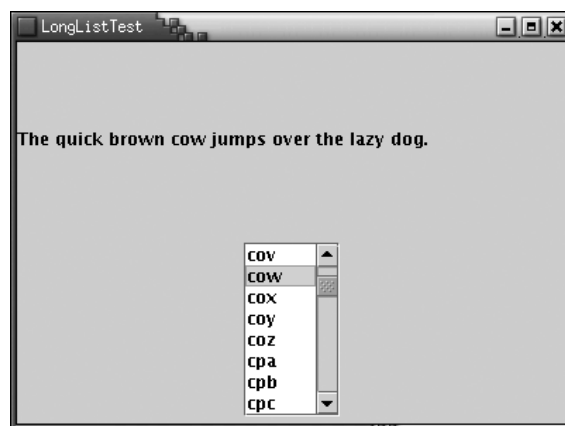
En quoi cette généralité est-elle utile ? Pourquoi l’objet `JList` ne stocke-t-il pas simplement un vecteur des objets ?

Vous remarquerez que l’interface ne spécifie pas la manière dont les objets sont stockés. En particulier, elle ne les oblige pas à se stocker du tout ! La méthode `getElementAt` est libre de recalculer chaque valeur dès qu’elle est appelée. Ceci peut être utile si vous souhaitez présenter une grande série, sans avoir à stocker les valeurs.

Dans l'exemple suivant, nous laissons l'utilisateur choisir parmi *tous les mots de trois lettres* dans une zone de liste (voir Figure 5.3).

Figure 5.3

Choisir dans une très longue liste de choix.



Il existe $26 \times 26 \times 26 = 17\,576$ combinaisons à trois lettres. Plutôt que de stocker toutes ces combinaisons, nous les recalculerons lorsque l'utilisateur fera défiler la liste.

Ceci se révèle très facile à implémenter. La partie difficile, l'ajout et la suppression d'écouteurs, a déjà été réalisée dans la classe `AbstractListModel` que nous étendons. Nous avons simplement besoin de fournir les méthodes `getSize` et `getElementAt` :

```
class WordListModel extends AbstractListModel
{
    public WordListModel(int n) { length = n; }
    public int getSize() { return (int) Math.pow(26, length); }
    public Object getElementAt(int n)
    {
        // calcule la n-ième chaîne
        . . .
    }
    . . .
}
```

Le calcul de la n -ième chaîne est un peu technique : vous en trouverez les détails dans le code de l'Exemple 5.2.

Maintenant que nous vous avons donné un modèle, nous pouvons simplement construire une liste qui permette à l'utilisateur de parcourir les éléments fournis par le modèle :

```
JList wordList = new JList(new WordListModel(3));
wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
JScrollPane scrollPane = new JScrollPane(wordList);
```

Le fait est que les chaînes ne sont jamais *stockées*. Seules celles que l'utilisateur demande réellement à voir sont générées.

Nous devons réaliser un réglage pour indiquer au composant de la liste que tous les éléments possèdent une largeur et une hauteur fixes. La manière la plus simple de définir les dimensions d'une cellule consiste à spécifier une *valeur de cellule prototype* :

```
wordList.setPrototypeCellValue("www");
```

304 *Au cœur de Java 2 - Fonctions avancées*

Cette valeur permet de déterminer la taille de toutes les cellules. Vous pouvez aussi définir une taille de cellule fixe :

```
wordList.setFixedCellWidth(50);  
wordList.setFixedCellHeight(15);
```

Faute de quoi, le composant de liste calculerait chaque élément pour mesurer sa largeur et sa hauteur. Ceci prendrait un long moment.

D'un point de vue pratique, des listes aussi longues sont rarement utiles. Il est en effet extrêmement pénible pour un utilisateur de parcourir une très grande sélection. C'est pour cette raison que nous pensons que le contrôle de la liste est véritablement trop perfectionné. Une sélection gérée de manière confortable par l'utilisateur à l'écran doit être assez petite pour pouvoir être stockée directement dans le composant de la liste. Cet arrangement épargne aux programmeurs de devoir traiter le modèle de liste comme une entité séparée. D'autre part, la classe `JList` est cohérente avec les classes `JTree` et `JTable` où cette généralité est utile.

Exemple 5.2 : *LongListTest.java*

```
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
import javax.swing.event.*;  
  
/**  
 * Ce programme montre une liste qui calcule des entrées de liste de  
 * manière dynamique.  
 */  
public class LongListTest  
{  
    public static void main(String[] args)  
    {  
        JFrame frame = new LongListFrame();  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        frame.setVisible(true);  
    }  
}  
  
/**  
 * Ce bloc contient une longue liste de mots ainsi qu'une étiquette qui  
 * montre une phrase constituée du mot choisi.  
 */  
class LongListFrame extends JFrame  
{  
    public LongListFrame()  
    {  
        setTitle("LongListTest");  
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);  
  
        wordList = new JList(new WordListModel(3));  
        wordList.setSelectionMode  
            (ListSelectionModel.SINGLE_SELECTION);  
        wordList.setPrototypeCellValue("www");  
        JScrollPane scrollPane = new JScrollPane(wordList);
```

```

JPanel p = new JPanel();
p.add(scrollPane);
wordList.addListSelectionListener(new
    ListSelectionListener()
    {
        public void valueChanged(ListSelectionEvent evt)
        {
            StringBuilder word
                = (StringBuilder) wordList.getSelectedValue();
            setSubject(word.toString());
        }
    });

Container contentPane = getContentPane();
contentPane.add(p, BorderLayout.NORTH);
label = new JLabel(prefix + suffix);
contentPane.add(label, BorderLayout.CENTER);
setSubject("fox");
}

/**
 * Définit le sujet de l'étiquette.
 * @param word le nouveau sujet de la phrase
 */
public void setSubject(String word)
{
    StringBuilder text = new StringBuilder(prefix);
    text.append(word);
    text.append(suffix);
    label.setText(text.toString());
}

private static final int DEFAULT_WIDTH = 400;
private static final int DEFAULT_HEIGHT = 300;
private JList wordList;
private JLabel label;
private String prefix = "The quick brown ";
private String suffix = " jumps over the lazy dog.";
}

/**
 * Un modèle qui génère un mot à n lettres de manière dynamique.
 */
class WordListModel extends AbstractListModel
{
    /**
     * Construit le modèle.
     * @param n la longueur du mot
     */
    public WordListModel(int n) { length = n; }

    public int getSize()
    {
        return (int) Math.pow(LAST - FIRST + 1, length);
    }
}

```


306 *Au cœur de Java 2 - Fonctions avancées*

```

public Object getElementAt(int n)
{
    StringBuilder r = new StringBuilder();
    for (int i = 0; i < length; i++)
    {
        char c = (char)(FIRST + n % (LAST - FIRST + 1));
        r.insert(0, c);
        n = n / (LAST - FIRST + 1);
    }
    return r;
}

private int length;
public static final char FIRST = 'a';
public static final char LAST = 'z';
}

```

javax.swing.JList 1.2

- `JList(ListModel dataModel)`
Construit une liste qui affiche les éléments dans le modèle spécifié.
- `void setPrototypeCellValue(Object newValue)`
- `Object getPrototypeCellValue()`
Définissent ou récupèrent la valeur de la cellule prototype utilisée pour déterminer la largeur et la hauteur de chaque cellule de la liste. La valeur par défaut vaut null, ce qui oblige à mesurer la taille de chaque cellule.
- `void setFixedCellWidth(int width)`
Si la largeur est supérieure à zéro, spécifie la largeur de chaque cellule de la liste. La valeur par défaut est de -1, ce qui oblige à mesurer la taille de chaque cellule.
- `void setFixedCellHeight(int height)`
Si la hauteur est supérieure à zéro, spécifie la hauteur de chaque cellule dans la liste. La valeur par défaut est de -1, ce qui oblige à mesurer la taille de chaque cellule.

javax.swing.ListModel 1.2

- `int getSize()`
Renvoie le nombre d'éléments du modèle.
- `Object getElementAt(int position)`
Renvoie un élément du modèle à la position donnée.

Insérer et supprimer des valeurs

Il est impossible de modifier directement la série des valeurs de liste. Vous devrez plutôt accéder au *modèle* et ajouter et supprimer des éléments. Une fois de plus, ceci est plus simple à dire qu'à faire. Supposons que vous souhaitiez ajouter d'autres valeurs à une liste. Vous pouvez obtenir une référence au modèle :

```
ListModel model = list.getModel();
```

Mais ceci ne fonctionne pas. Comme vous l'avez vu dans la section précédente, l'interface `ListModel` ne possède pas de méthodes pour insérer ou supprimer des éléments puisque, après tout, un modèle de liste n'a pas besoin de *stocker* les éléments.

Essayons autrement. L'un des constructeurs de `JList` prend un vecteur d'objets :

```
Vector<String> values = new Vector<String>();
values.addElement("quick");
values.addElement("brown");
. . .
JList list = new JList(values);
```

Bien entendu, vous pouvez maintenant modifier le vecteur et ajouter ou supprimer des éléments, mais la liste ne sait pas ce qui survient, elle ne peut donc pas réagir aux changements. En particulier, la liste ne peut pas mettre à jour son affichage lorsque vous ajoutez les valeurs. Le constructeur n'est donc pas très utile.

Au lieu de cela, vous devez construire un modèle particulier, le `DefaultListModel`, le remplir avec les valeurs initiales et l'associer à la liste.

```
DefaultListModel model = new DefaultListModel();
model.addElement("quick");
model.addElement("brown");
. . .
JList list = new JList(model);
```

Vous pouvez maintenant ajouter ou supprimer des valeurs de l'objet `model`. Celui-ci avertit alors la liste des changements et la liste se redessine elle-même.

```
model.removeElement("quick");
model.addElement("slow");
```

Comme vous pouvez le voir, la classe `DefaultListModel` n'utilise pas les mêmes noms de méthodes que les classes de collection.

Le modèle de liste par défaut utilise un vecteur en interne pour stocker les valeurs.

ATTENTION

Il existe des constructeurs `JList` qui construisent une liste à partir d'un tableau, d'un vecteur d'objets ou de chaînes. Vous pourriez penser que ces constructeurs utilisent un `DefaultListModel` pour stocker ces valeurs. Or, ce n'est pas le cas : ils élaborent un modèle trivial capable d'accéder aux valeurs sans aucune provision de notification si le contenu change. Par exemple, voici le code du constructeur qui élabore un `JList` à partir d'un `Vector` :

```
public JList(final Vector<?> listData)
{
    this (new AbstractListModel()
    {
        public int getSize() { return listData.size(); }
        public Object getElementAt(int i)
        { return listData.elementAt(i); }
    });
}
```

Cela signifie que, si vous modifiez le contenu du vecteur après la construction de la liste, cette dernière pourrait présenter un mélange déroutant d'anciennes et de nouvelles valeurs jusqu'à ce qu'elle soit totalement rafraîchie (le mot clé `final` dans le constructeur qui précède ne vous empêche pas de modifier le vecteur, il signifie simplement que le constructeur lui-même ne modifiera pas la valeur de la référence `listData` ; le mot clé est nécessaire car l'objet `listData` est utilisé dans la classe intérieure).

308 Au cœur de Java 2 - Fonctions avancées

javax.swing.JList 1.2

- `ListModel getModel()`
Récupère le modèle de cette liste.

javax.swing.DefaultListModel 1.2

- `void addElement(Object obj)`
Ajoute l'objet à la fin du modèle.
- `boolean removeElement(Object obj)`
Supprime la première occurrence de l'objet dans le modèle. Renvoie `true` si l'objet était contenu dans le modèle, `false` dans le cas contraire.

Afficher des valeurs

Toutes les listes que vous avez vues dans ce chapitre contenaient uniquement des chaînes. Il est en fait tout aussi simple de présenter une liste d'icônes : passez simplement un tableau ou un vecteur rempli d'objets `Icon`. De manière plus intéressante, vous pouvez facilement représenter vos valeurs de liste avec n'importe quel dessin.

Alors que la classe `JList` peut afficher des chaînes et des icônes automatiquement, vous devrez installer un *afficheur de cellules de liste* dans l'objet `JList` pour tous les dessins personnalisés. Un afficheur de cellules de liste représente n'importe quelle classe qui implémente l'interface suivante :

```
interface ListCellRenderer
{
    Component getListCellRendererComponent(JList list,
        Object value, int index,
        boolean isSelected, boolean cellHasFocus);
}
```

Cette méthode est appelée pour chaque cellule. Elle renvoie un composant qui dessine le contenu de la cellule, lequel est ensuite placé à l'endroit qui convient dès qu'une cellule doit être affichée.

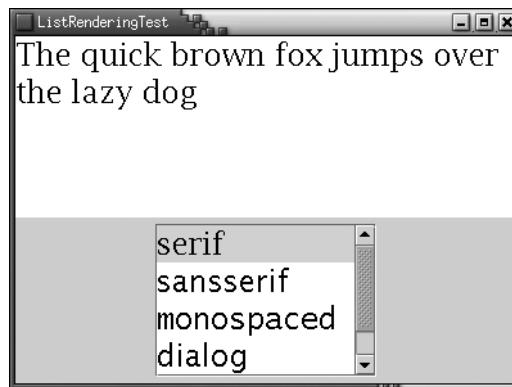
Pour créer un afficheur de cellule, créez une classe qui étend `JPanel`, comme ceci :

```
class MyCellRenderer extends JPanel implements ListCellRenderer
{
    public Component getListCellRendererComponent(JList list,
        Object value, int index, boolean isSelected, boolean cellHasFocus)
    {
        // informations nécessaires pour le dessin et
        // la mesure de la taille
        return this;
    }
    public void paintComponent(Graphics g)
    {
        // le code de dessin vient ici
    }
    public Dimension getPreferredSize()
    {
        // le code de mesure de la taille vient ici
    }
    // champs d'instance
}
```

Dans l'Exemple 5.3, nous présentons les choix de polices, de manière graphique, en présentant l'apparence véritable de chacune (voir Figure 5.4). Dans la méthode `paintComponent`, nous affichons chaque nom dans sa propre police. Nous devons également nous assurer de faire correspondre les couleurs habituelles de l'apparence de la classe `JList`. Nous obtenons ces couleurs en appelant `getForeground/getBackground` et les méthodes `getSelectionForeground/getSelectionBackground` de la classe `JList`. Dans la méthode `getPreferredSize`, nous devons mesurer la taille de la chaîne, à l'aide des techniques envisagées au Chapitre 7 du Volume 1.

Figure 5.4

Une zone de liste avec des cellules affichées.



Pour installer l'afficheur de cellules, appelez simplement la méthode `setCellRenderer` :

```
fontList.setCellRenderer(new FontCellRenderer());
```

Désormais, toutes les cellules de liste sont dessinées avec l'afficheur personnalisé.

Il existe en fait une méthode plus simple pour écrire des afficheurs personnalisés qui fonctionnent dans de nombreux cas. Si l'image affichée contient simplement du texte, une icône et un changement de couleur, vous pouvez vous en sortir en configurant un `JLabel`. Par exemple, pour afficher le nom de la police dans sa propre police, nous pouvons utiliser l'afficheur suivant :

```
class FontCellRenderer extends JLabel implements ListCellRenderer
{
    public Component getListCellRendererComponent(JList list,
        Object value, int index, boolean isSelected,
        boolean cellHasFocus)
    {
        JLabel label = new JLabel();
        Font font = (Font) value;
        setText(font.getFamily());
        setFont(font);
        setOpaque(true);
        setBackground(isSelected
            ? list.getSelectionBackground()
            : list.getBackground());
        setForeground(isSelected
            ? list.getSelectionForeground()
            : list.getForeground());
        return this;
    }
}
```

310 *Au cœur de Java 2 - Fonctions avancées*

Vous remarquerez que nous n'écrivons aucune méthode `paintComponent` ou `getPreferredSize` ; la classe `JLabel` implémente déjà ces méthodes, à notre plus grande satisfaction. Tout ce dont nous avons besoin, c'est de configurer l'étiquette correctement en définissant son texte, sa police et sa couleur.

Ce code constitue un raccourci commode dans les cas où un composant existant, dans ce cas `JLabel`, fournit déjà toutes les fonctionnalités nécessaires à l'affichage d'une valeur de cellule.

Exemple 5.3 : *ListRenderingTest.java*

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Ce programme présente l'utilisation des afficheurs de cellules dans
 * une zone de liste.
 */
public class ListRenderingTest
{
    public static void main(String[] args)
    {
        JFrame frame = new ListRenderingFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc contient une liste avec un ensemble de polices et une zone
 * de texte définie sur la police sélectionnée.
 */
class ListRenderingFrame extends JFrame
{
    public ListRenderingFrame()
    {
        setTitle("ListRenderingTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        ArrayList<Font> fonts = new ArrayList<Font>();
        final int SIZE = 24;
        fonts.add(new Font("Serif", Font.PLAIN, SIZE));
        fonts.add(new Font("SansSerif", Font.PLAIN, SIZE));
        fonts.add(new Font("Monospaced", Font.PLAIN, SIZE));
        fonts.add(new Font("Dialog", Font.PLAIN, SIZE));
        fonts.add(new Font("DialogInput", Font.PLAIN, SIZE));
        fontList = new JList(fonts.toArray());
        fontList.setVisibleRowCount(4);
        fontList.setSelectionMode
            (ListSelectionModel.SINGLE_SELECTION);
        fontList.setCellRenderer(new FontCellRenderer());
        JScrollPane scrollPane = new JScrollPane(fontList);
    }
}
```

```

JPanel p = new JPanel();
p.add(scrollPane);
fontList.addListSelectionListener(new
    ListSelectionListener()
    {
        public void valueChanged(ListSelectionEvent evt)
        {
            Font font = (Font) fontList.getSelectedValue();
            text.setFont(font);
        }
    });

Container contentPane = getContentPane();
contentPane.add(p, BorderLayout.SOUTH);
text = new JTextArea(
    "The quick brown fox jumps over the lazy dog");
text.setFont((Font) fonts.get(0));
text.setLineWrap(true);
text.setWrapStyleWord(true);
contentPane.add(text, BorderLayout.CENTER);
}

private JTextArea text;
private JList fontList;
private static final int DEFAULT_WIDTH = 400;
private static final int DEFAULT_HEIGHT = 300;
}

/**
 * Un afficheur de cellules pour les objets Font qui affiche le nom de la
 * police dans sa propre police.
 */
class FontCellRenderer extends JPanel implements ListCellRenderer
{
    public Component getListCellRendererComponent
        (JList list, Object value, int index, boolean isSelected,
         boolean cellHasFocus)
    {
        font = (Font) value;
        background = isSelected
            ? list.getSelectionBackground()
            : list.getBackground();
        foreground = isSelected
            ? list.getSelectionForeground()
            : list.getForeground();
        return this;
    }

    public void paintComponent(Graphics g)
    {
        String text = font.getFamily();
        FontMetrics fm = g.getFontMetrics(font);
        g.setColor(background);
        g.fillRect(0, 0, getWidth(), getHeight());
        g.setColor(foreground);
        g.setFont(font);
        g.drawString(text, 0, fm.getAscent());
    }
}

```

312 *Au cœur de Java 2 - Fonctions avancées*

```

public Dimension getPreferredSize()
{
    String text = font.getFamily();
    Graphics g = getGraphics();
    FontMetrics fm = g.getFontMetrics(font);
    return new Dimension(fm.stringWidth(text),
                        fm.getHeight());
}

private Font font;
private Color background;
private Color foreground;
}

```

javax.swing.JList 1.2

- `Color getBackground()`
Renvoie la couleur d'arrière-plan pour les cellules non sélectionnées.
- `Color getSelectionBackground()`
Renvoie la couleur d'arrière-plan pour les cellules sélectionnées.
- `Color setForeground()`
Renvoie la couleur de premier plan pour les cellules non sélectionnées.
- `Color getSelectionForeground()`
Renvoie la couleur de premier plan pour les cellules sélectionnées.
- `void setCellRenderer(ListCellRenderer cellRenderer)`
Définit l'afficheur utilisé pour dessiner les cellules dans la liste.

javax.swing.ListCellRenderer 1.2

- `Component getListCellRendererComponent(JList list, Object item, int index, boolean isSelected, boolean hasFocus)`
Renvoie un composant dont la méthode `paint` dessinera le contenu de la cellule. Si les cellules de la liste n'ont pas de taille fixe, ce composant doit également implémenter `getPreferredSize`.

<i>Paramètres :</i>	<code>list</code>	la liste dont la cellule est dessinée
	<code>item</code>	l'élément à dessiner
	<code>index</code>	l'indice où est stocké l'élément dans le modèle
	<code>isSelected</code>	renvoie <code>true</code> si la cellule spécifiée a été sélectionnée
	<code>hasFocus</code>	renvoie <code>true</code> si la cellule spécifiée possède le focus

Arbres

Tous les utilisateurs d'ordinateurs possédant un système de fichiers hiérarchique ont déjà rencontré des arbres, comme celui de la Figure 5.5. Naturellement, les répertoires et les fichiers représentent seulement une forme possible d'organisation en arbre. Les programmeurs connaissent bien les

héritages de classes en arbres. Il existe un grand nombre de structures en arbres dans la vie de tous les jours, comme la hiérarchie des pays, des régions, des départements et des villes représentée dans la Figure 5.6.

Figure 5.5

Un arbre de répertoires.

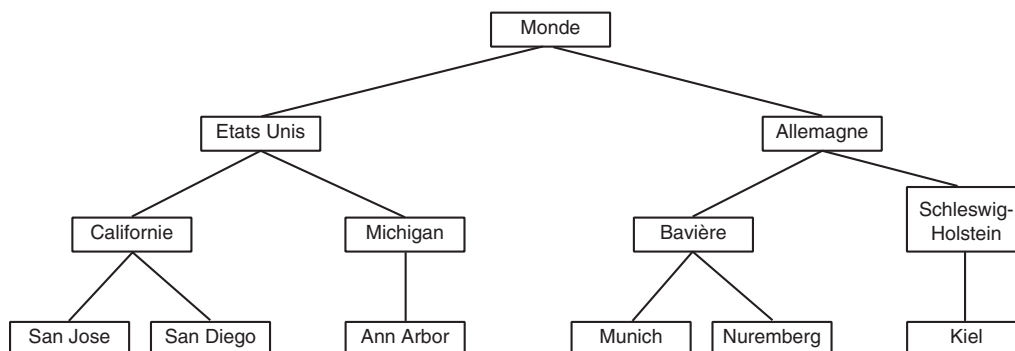
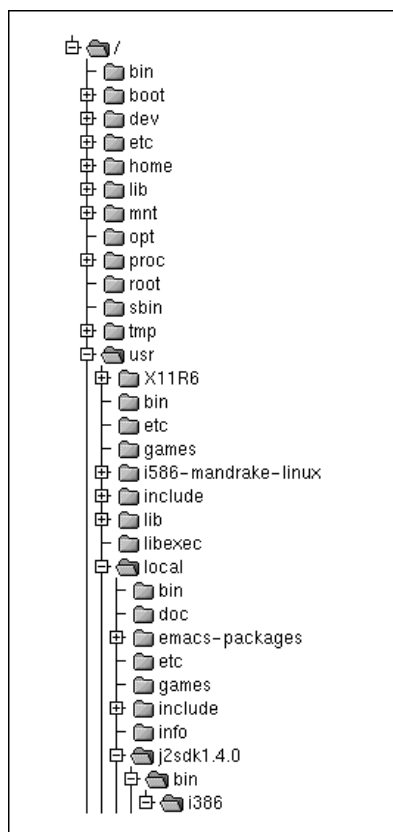


Figure 5.6

Une hiérarchie de pays, de régions et de villes.

314 Au cœur de Java 2 - Fonctions avancées

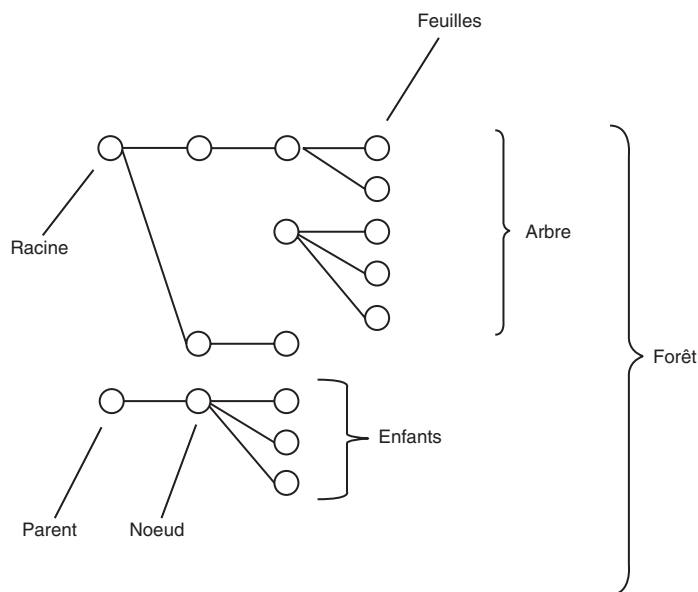
En tant que programmeurs, il nous faut souvent afficher ces structures en arbres. Heureusement, la bibliothèque Swing possède une classe `JTree` spécialement conçue dans ce but. La classe `JTree` (ainsi que ses classes d'aide) prend en charge l'organisation des arbres et le traitement des requêtes de l'utilisateur visant à ajouter et à supprimer des nœuds. Dans cette section, vous apprendrez à utiliser la classe `JTree`.

Comme pour d'autres composants complexes de Swing, nous devons nous concentrer sur les cas les plus fréquents et les plus pratiques, puisque nous ne pouvons pas appréhender toutes les subtilités de ce sujet. Si vous souhaitez obtenir un effet très particulier, nous vous recommandons de consulter *Core Java Foundation Classes*, de Kim Topley (Prentice-Hall, 1998) ou *Core Swing: Advanced Programming*, du même auteur (Prentice-Hall, 1999).

Avant de poursuivre, mettons-nous d'accord sur quelques éléments de terminologie (voir Figure 5.7). Un arbre est composé de *nœuds*. Un nœud peut soit être une *feuille*, soit posséder des nœuds *enfant*. Chaque nœud, à l'exception du nœud de départ (la *racine*), possède un seul *parent*. Un arbre possède un seul nœud de départ. Des arbres peuvent être assemblés dans un groupe, chaque arbre possédant sa propre racine. Ce type de groupe est appelé une *forêt*.

Figure 5.7

La terminologie des arbres.



Exemples d'arbres

Dans notre premier programme d'exemple, nous nous contentons d'afficher un arbre possédant quelques nœuds (voir Figure 5.9). Comme la plupart des autres composants Swing, le composant `JTree` respecte une architecture de modèle/affichage/contrôles. Un modèle des données hiérarchiques doit être fourni, et le composant affiche ces données pour vous. Pour construire un `JTree`, vous devez spécifier le modèle d'arbre dans le constructeur :

```

TreeModel model = . . . ;
JTree tree = new JTree(model);

```

INFO

Il existe également des constructeurs qui permettent de créer des arbres à partir d'un ensemble d'éléments.

```
JTree(Object[] nodes)
JTree(Vector<?> nodes)
JTree(Hashtable<?, ?> nodes) // les valeurs sont transformées en nœuds
```

Ces constructeurs ne sont pas très utiles. Ils permettent surtout de générer des forêts d'arbres, chaque arbre possédant un seul nœud. Le troisième constructeur semble particulièrement inutile puisque les nœuds sont organisés selon l'ordre aléatoire fourni par les codes de hachage des clés.

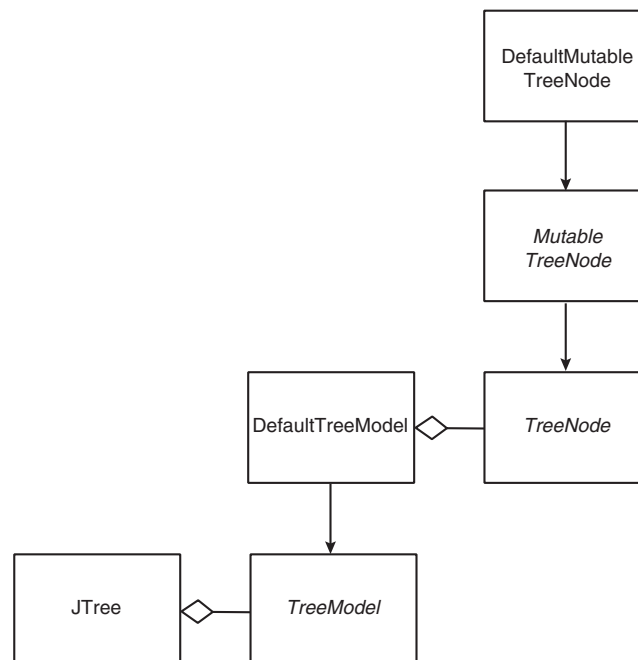
Comment obtenir un modèle d'arbre ? Vous pouvez construire votre propre modèle en créant une classe qui implémente l'interface `TreeModel`. Vous verrez plus loin dans ce chapitre comment procéder. Pour l'instant, nous allons reprendre le modèle par défaut `DefaultTreeModel` fourni dans la bibliothèque Swing.

Pour construire un modèle d'arbre par défaut, vous devez fournir un nœud racine (*root*).

```
TreeNode root = . . . ;
DefaultTreeModel model = new DefaultTreeModel(root);
```

`TreeNode` est une autre interface. Vous pouvez placer dans ce modèle d'arbre par défaut des objets de n'importe quelle classe qui implémente cette interface. Pour l'instant, nous nous servons des classes de nœuds concrètes fournies dans Swing, c'est-à-dire `DefaultMutableTreeNode`. Cette classe implémente l'interface `MutableTreeNode`, une sous-interface de `TreeNode` (voir Figure 5.8).

Figure 5.8
Classes d'arbres.



316 Au cœur de Java 2 - Fonctions avancées

Un nœud d'arbre mutable par défaut renferme un objet, et plus précisément un *objet de l'utilisateur*. L'arbre peut transformer les objets de l'utilisateur contenus dans chaque nœud. A moins que vous ne spécifiez une méthode de transformation, l'arbre se contente d'afficher une chaîne résultant de la méthode `toString`.

Dans notre premier exemple, nous nous servons de chaînes comme objets de l'utilisateur. En pratique, vous remplirez probablement des arbres avec des objets d'utilisateur plus importants. Par exemple, pour afficher un arbre de répertoire, il convient de le remplir avec des objets `File`.

Vous pouvez spécifier le type des objets d'utilisateur dans le constructeur, mais vous pouvez également le définir par la suite grâce à la méthode `setUserObject`.

```
DefaultMutableTreeNode node
    = new DefaultMutableTreeNode("Texas");
node.setUserObject("Californie");
```

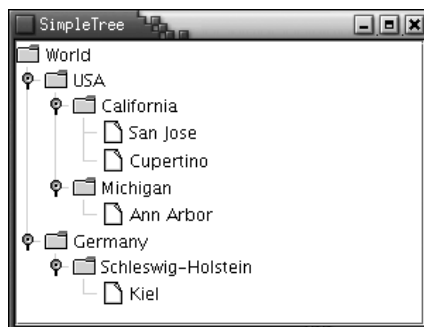
Ensuite, il faut établir les relations hiérarchiques entre les parents et les enfants pour chaque nœud. Commencez par le nœud racine, et utilisez la méthode `add` pour ajouter des enfants :

```
DefaultMutableTreeNode root
    = new DefaultMutableTreeNode("Monde");
DefaultMutableTreeNode country
    = new DefaultMutableTreeNode("USA");
root.add(country);
DefaultMutableTreeNode state
    = new DefaultMutableTreeNode("Californie");
country.add(state);
```

La Figure 5.9 illustre cet exemple d'arbre.

Figure 5.9

Exemple d'arbre.



Vous devez relier tous les nœuds de cette manière. Construisez ensuite un `DefaultTreeModel` avec le nœud racine. Pour terminer, construisez un `JTree` avec le modèle de l'arbre.

```
DefaultTreeModel treeModel = new DefaultTreeModel(root);
JTree tree = new JTree(treeModel);
```

Plus simplement, il suffit de passer le nœud racine au constructeur `JTree`. L'arbre construit alors automatiquement un modèle d'arbre par défaut :

```
JTree tree = new JTree(root);
```

L'Exemple 5.4 fournit le code complet.

Exemple 5.4 : SimpleTree.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;

/**
 * Ce programme présente un exemple d'arbre.
 */
public class SimpleTree
{
    public static void main(String[] args)
    {
        JFrame frame = new SimpleTreeFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc contient un exemple d'arbre qui affiche un
 * modèle d'arbre construit manuellement.
 */
class SimpleTreeFrame extends JFrame
{
    public SimpleTreeFrame()
    {
        setTitle("SimpleTree");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // définition des données du modèle d'arbre

        DefaultMutableTreeNode root
            = new DefaultMutableTreeNode("Monde");
        DefaultMutableTreeNode country
            = new DefaultMutableTreeNode("USA");
        root.add(country);
        DefaultMutableTreeNode state
            = new DefaultMutableTreeNode("Californie");
        country.add(state);
        DefaultMutableTreeNode city
            = new DefaultMutableTreeNode("San José");
        state.add(city);
        city = new DefaultMutableTreeNode("Cupertino");
        state.add(city);
        state = new DefaultMutableTreeNode("Michigan");
        country.add(state);
        city = new DefaultMutableTreeNode("Ann Arbor");
        state.add(city);
        country = new DefaultMutableTreeNode("Allemagne");
        root.add(country);
        state = new DefaultMutableTreeNode("Schleswig-Holstein");
        country.add(state);
        city = new DefaultMutableTreeNode("Kiel");
        state.add(city);

        // construit l'arbre et le place dans un panneau déroulant
    }
}
```

318 Au cœur de Java 2 - Fonctions avancées

```

    JTree tree = new JTree(root);
    Container contentPane = getContentPane();
    contentPane.add(new JScrollPane(tree));
}
private static final int DEFAULT_WIDTH = 300;
private static final int DEFAULT_HEIGHT = 200;
}

```

Lorsque vous exécutez ce programme, l'arbre possède l'aspect représenté par la Figure 5.10. Seuls le nœud racine et ses enfants sont visibles. Cliquez sur les icônes rondes (les *poignées*) pour ouvrir les arbres de niveau inférieur. Le segment dépassant des poignées se trouve sur la droite lorsque le sous-répertoire est caché, et il pointe vers le bas lorsque le sous-répertoire est affiché (voir Figure 5.11). Nous ne savons pas exactement ce que les concepteurs de l'aspect Metal avaient en tête, mais il nous semble que ces icônes représentent des poignées de porte. Il faut appuyer sur la poignée pour ouvrir le sous-répertoire.

INFO

Naturellement, l'affichage de l'arbre dépend de l'aspect sélectionné. Nous venons de voir l'aspect de Metal. Avec les aspects de Windows et de Motif, les poignées ont une forme plus familière : un "+" ou un "-" dans un carré (voir Figure 5.12).

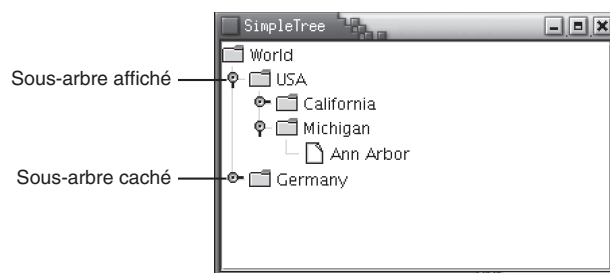
Figure 5.10

L'affichage initial d'un arbre.



Figure 5.11

Les sous-répertoires cachés et affichés.



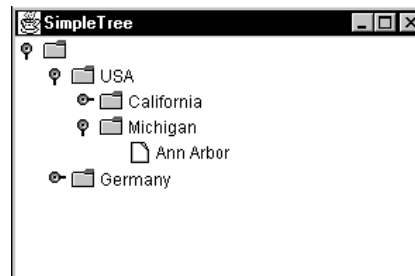
Jusqu'au JDK 1.3, l'aspect Metal n'affiche pas par défaut la structure de l'arbre (voir Figure 5.13). Quant à la version JDK 1.4, le style de trait par défaut est "angled".

Dans le JDK 1.4, vous devrez avoir recours à la formule magique suivante pour afficher des lignes entre les parents et leurs enfants :

```
tree.putClientProperty("JTree.lineStyle", "None");
```

Figure 5.12

Un arbre avec l'aspect de Windows.



A l'inverse, pour vous assurer que les lignes sont affichées, utilisez

```
tree.putClientProperty("JTree.lineStyle", "Angled");
```

Figure 5.13

Un arbre avec un style de lignes brisées.



Un autre style appelé `Horizontal` est représenté à la Figure 5.14. L'arbre est affiché avec des lignes horizontales séparant uniquement les enfants du nœud racine. Nous ne sommes pas absolument certains de connaître l'intérêt de ce style.

Figure 5.14

Un arbre avec un style de lignes horizontales.



Par défaut, il n'existe aucune poignée pour cacher la racine d'un arbre. Si vous le désirez, vous pouvez en ajouter une avec la ligne suivante :

```
tree.setShowsRootHandles(true);
```

La Figure 5.15 montre l'effet produit. Vous pouvez maintenant cacher l'arbre entier dans le nœud racine.

320 Au cœur de Java 2 - Fonctions avancées

Figure 5.15

Un arbre avec une poignée sur la racine.



Inversement, la racine peut être entièrement cachée. Cela peut être utile si vous souhaitez afficher une *forêt*, c'est-à-dire un ensemble d'arbres possédant chacun leur propre racine. Vous devez cependant regrouper tous les arbres de la forêt avec une seule racine commune. Vous pouvez alors cacher cette racine grâce à l'instruction suivante :

```
tree.setRootVisible(false);
```

Examinez la Figure 5.16. Il semble qu'il y ait deux racines, appelées "USA" et "Allemagne". En fait, la racine qui regroupe ces deux racines est rendue invisible.

Figure 5.16

Une forêt.



Passons maintenant de la racine aux feuilles de l'arbre. Notez que les feuilles possèdent une icône différente de celle des autres nœuds (voir Figure 5.17).

Lorsque l'arbre est affiché, chaque nœud est représenté par une icône. Il existe en fait trois sortes d'icônes : les icônes de feuilles, les icônes de nœuds intermédiaires ouverts et les icônes de nœuds intermédiaires fermés. Pour des raisons de simplicité, nous appellerons les deux dernières des icônes de répertoires.

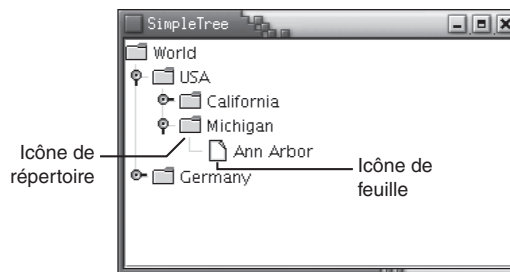
L'afficheur de nœuds doit savoir quelle icône utiliser pour chaque nœud. Par défaut, cette décision est prise de la façon suivante : si la méthode `isLeaf` d'un nœud renvoie `true`, l'icône de feuille est utilisée. Sinon, une icône de répertoire est utilisée.

La méthode `isLeaf` de la classe `DefaultMutableTreeNode` renvoie `true` si le nœud ne possède aucun enfant. Par conséquent, les nœuds possédant des enfants sont associés à des icônes de répertoires, et les nœuds sans enfants sont associés à des icônes de feuilles.

Parfois, cette technique n'est pas toujours appropriée. Supposons que vous ajoutiez un nœud "Montana" dans notre exemple d'arbre, mais que nous ne sachions pas quelles villes ajouter. Il convient cependant d'éviter d'affecter une icône de feuille à ce nœud puisque seules les villes correspondent à des feuilles.

Figure 5.17

*Les icônes des feuilles
et des dossiers.*



La classe `JTree` ne possède aucune information lui permettant de déterminer si un nœud doit être considéré comme une feuille ou comme un répertoire. Elle le demande donc au modèle de l'arbre. Si un nœud sans enfant n'est pas toujours interprété au plan conceptuel comme une feuille, vous pouvez demander au modèle d'utiliser différents critères pour vérifier qu'un nœud est bien une feuille, en interrogeant la propriété `AllowsChildren` d'un nœud.

Pour les nœuds qui ne devraient pas avoir d'enfants, appelez :

```
node.setAllowsChildren(false);
```

Puis indiquez au modèle d'arbre qu'il faut examiner la propriété `AllowsChildren` d'un nœud pour savoir s'il doit être affiché avec une icône de feuille ou non. La méthode `setAsksAllowsChildren` de la classe `DefaultTreeModel` permet de définir ce comportement :

```
model.setAsksAllowsChildren(true);
```

A partir de ces critères de décision, les nœuds susceptibles d'avoir des enfants sont associés à des icônes de répertoires, et les autres à des icônes de feuilles.

Sinon, si vous construisez un arbre en fournissant un nœud racine, vous devez spécifier ce comportement dans le constructeur :

```
JTree tree = new JTree(root, true);
// les nœuds qui n'ont pas d'enfants ont des icônes de feuilles
```

javax.swing.JTree 1.2

- `JTree(TreeModel model)`
Construit un arbre à partir d'un modèle d'arbre.
 - `JTree(TreeNode root)`
 - `JTree(TreeNode root, boolean asksAllowChildren)`
Construisent un arbre à partir du modèle d'arbre par défaut qui affiche la racine et ses enfants.
- Paramètres :*
- | | |
|---------------------------------|---|
| <code>root</code> | le nœud de départ (racine) |
| <code>asksAllowsChildren</code> | <code>true</code> pour utiliser la propriété de nœud "Allows Children" pour déterminer si un nœud est une feuille |
- `void setShowsRootHandles(boolean b)`
Si `b` vaut `true`, le nœud racine possède une poignée pour cacher ou afficher ses enfants.
 - `void setRootVisible(boolean b)`
Si `b` vaut `true`, le nœud racine est affiché, sinon il est caché.

322 Au cœur de Java 2 - Fonctions avancées

***javax.swing.tree.TreeNode* 1.2**

- `boolean isLeaf()`
Renvoie true si le nœud est une feuille au plan conceptuel.
- `boolean getAllowsChildren()`
Renvoie true si ce nœud peut avoir des nœuds enfant.

***javax.swing.tree.MutableTreeNode* 1.2**

- `void setUserObject(Object userObject)`
Définit l'objet de l'utilisateur dont l'arbre se sert pour l'affichage.

***javax.swing.tree.TreeModel* 1.2**

- `boolean isLeaf(Object node)`
Renvoie true si node doit être affiché comme une feuille.

***javax.swing.tree.DefaultTreeModel* 1.2**

- `void setAsksAllowsChildren(boolean b)`
Si b vaut true, les nœuds sont affichés comme des feuilles lorsque leur méthode `getAllowsChildren` renvoie false. Dans le cas contraire, ils sont affichés comme des feuilles lorsque leur méthode `isLeaf` renvoie true.

***javax.swing.tree.DefaultMutableTreeNode* 1.2**

- `DefaultMutableTreeNode(Object userObject)`
Construit un nœud d'arbre mutable avec l'objet d'utilisateur spécifié.
- `void add(MutableTreeNode child)`
Ajoute un nœud en spécifiant qu'il s'agit du dernier enfant de ce nœud.
- `void setAllowsChildren(boolean b)`
Si b vaut true, des enfants peuvent être ajoutés à ce nœud.

***javax.swing.JComponent* 1.2**

- `void putClientProperty(Object key, Object value)`
Ajoute une paire clé/valeur dans un petit tableau géré par chaque composant. Il s'agit d'un mécanisme de secours dont certains composants Swing se servent pour enregistrer des propriétés spécifiques à leur aspect.

Modifier des arbres et leur structure

Dans le prochain programme d'exemple, vous apprendrez à modifier un arbre. La Figure 5.18 montre une interface utilisateur. Si vous cliquez sur les boutons "Ajouter un frère" ou "Ajouter un enfant", le programme ajoute un nouveau nœud (appelé Nouveau) dans l'arbre. Si vous cliquez sur le bouton "Supprimer", le programme efface le nœud sélectionné.

Pour implémenter ce comportement, vous devrez identifier le nœud sélectionné. La classe `JTree` possède une technique étonnante pour identifier les nœuds d'un arbre. Elle ne gère pas les nœuds de l'arbre, mais les *chemins des objets*, appelés *chemins de l'arbre*. Un chemin d'arbre commence à la racine et correspond à une séquence de nœuds enfant (voir Figure 5.19).

Figure 5.18
Modifier un arbre.

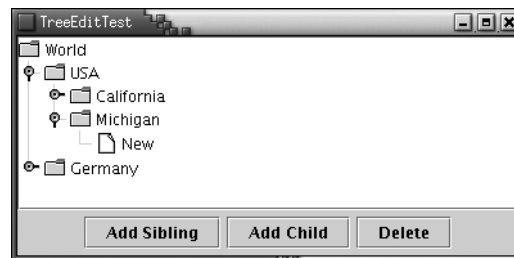


Figure 5.19
Un chemin d'arbre.



Vous vous demandez peut-être pourquoi la classe `JTree` a besoin du chemin complet. Ne peut-elle pas se contenter de récupérer un `TreeNode` et d'appeler en boucle sa méthode `getParent` ? En fait, la classe `JTree` ne connaît pas du tout l'interface `TreeNode`. Cette interface n'est en effet jamais utilisée par l'interface `TreeModel`. Elle ne sert qu'à l'implémentation `DefaultTreeModel`. Vous pouvez posséder d'autres modèles d'arbres dans lesquels les nœuds n'implémentent pas du tout l'interface `TreeNode`. Si vous avez recours à un modèle d'arbre qui gère d'autres types d'objets, ces derniers peuvent ne pas avoir de méthode `getParent` et `getChild`. Ils doivent cependant posséder des connexions entre eux. Cette tâche revient au modèle d'arbre. La classe `JTree` elle-même n'a aucune idée de la nature de leurs connexions. Pour cette raison, la classe `JTree` doit toujours travailler avec des chemins complets.

La classe `TreePath` gère une séquence de références d'`Object` (et pas de `TreeNode`). Un certain nombre de méthodes `JTree` renvoient des objets `TreePath`. Lorsque vous possédez un chemin d'arbre, il vous suffit en général de connaître le nœud final, que vous pouvez récupérer grâce à la méthode `getLastPathComponent`. Par exemple, pour trouver quel nœud est couramment sélectionné dans un arbre, vous pouvez utiliser la méthode `getSelectionPath` de la classe `JTree`. Vous obtiendrez en retour un objet `TreePath`, d'où vous déduirez le nœud sélectionné.

```
TreePath selectionPath = tree.getSelectionPath();
DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode)
    selectionPath.getLastPathComponent();
```

En fait, comme cette requête est très fréquente, il existe une méthode pratique qui vous fournit immédiatement le nœud sélectionné.

```
DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode)
    tree.getLastSelectedPathComponent();
```

Cette méthode n'est pas appelée `getSelectedNode` parce que l'arbre ne sait pas qu'il renferme des nœuds. Seul le modèle d'arbre gère les chemins des objets.

324 Au cœur de Java 2 - Fonctions avancées

INFO

Les chemins d'arbre sont l'une des deux techniques utilisées par la classe `JTree` pour décrire les nœuds. Il existe d'autres méthodes `JTree` qui acceptent ou renvoient un indice entier, une *position de ligne*. Une position de ligne est simplement un numéro de ligne (commençant à 0) correspondant au nœud spécifié dans l'arbre affiché. Seuls les nœuds visibles possèdent un numéro de ligne, et le numéro de ligne d'un nœud change si les nœuds qui le précèdent sont cachés, affichés ou modifiés. C'est pourquoi il vaut mieux éviter de travailler avec des positions de ligne. Toutes les méthodes `JTree` qui se servent de lignes possèdent un équivalent utilisant des chemins d'arbre.

Une fois que vous avez trouvé le nœud sélectionné, vous pouvez le modifier. Cependant, ne vous contentez pas d'ajouter des enfants à un nœud :

```
selectedNode.add(newNode); // NON !
```

Si vous modifiez la structure des nœuds, vous modifiez le modèle, mais l'affichage associé n'est pas mis à jour. Vous pouvez envoyer une notification par vous-même, mais si vous utilisez la méthode `insertNodeInto` de la classe `DefaultTreeModel`, vous refaites le travail de la classe du modèle. Par exemple, l'appel suivant ajoute un nœud et le déclare comme étant le dernier nœud du nœud sélectionné, et met à jour l'affichage de l'arbre.

```
model.insertNodeInto(newNode, selectedNode,  
    selectedNode.getChildCount());
```

L'appel similaire à `removeNodeFromParent` supprime un nœud et met à jour l'affichage.

```
model.removeNodeFromParent(selectedNode);
```

Si vous conservez la structure des nœuds, mais que vous modifiez un objet d'utilisateur, vous devrez appeler la méthode suivante :

```
model.nodeChanged(changedNode);
```

La notification automatique est l'un des avantages majeurs de l'utilisation d'un `DefaultTreeModel`. Si vous fournissez votre propre modèle d'arbre, vous devrez implémenter ce mécanisme à la main. Reportez-vous à l'ouvrage (de langue anglaise) *Core Java Foundation Classes* de Kim Topley pour plus de détails.

ATTENTION

La classe `DefaultTreeModel` possède une méthode `reload` qui recharge le modèle entier. Cependant, évitez d'appeler cette méthode uniquement pour mettre à jour votre arbre lorsque vous y avez apporté des modifications. Lorsqu'un arbre est généré à nouveau, tous les nœuds situés après les enfants de la racine sont cachés. Cela peut être extrêmement déconcertant pour vos utilisateurs, s'ils doivent ouvrir à nouveau leur arbre après chaque modification.

Lorsque l'affichage est mis à jour à cause d'une modification de la structure des nœuds, les enfants ajoutés ne sont pas automatiquement affichés. En particulier, si l'un des utilisateurs de notre programme d'exemple ajoutait un nouvel enfant à un nœud dont les enfants sont actuellement cachés, le nouvel enfant le serait aussi. Cela ne fournit à l'utilisateur aucune information sur le bon fonctionnement de la commande qu'il vient d'effectuer. Dans ce cas, il convient d'ouvrir tous les nœuds parent pour que le nœud qui vient d'être ajouté soit visible. Vous pouvez vous servir de la méthode `makeVisible` de la classe `JTree` dans ce but. La méthode `makeVisible` attend un chemin d'arbre pointant sur le nœud qu'elle doit rendre visible.

Par conséquent, vous serez amené à construire un chemin d'arbre à partir de la racine et allant jusqu'au nouveau nœud. Pour obtenir un chemin d'arbre, il faut commencer par appeler la méthode `getPathToRoot` de la classe `DefaultTreeModel`. Elle renvoie un tableau `TreeNode[]` contenant tous les nœuds situés entre un nœud et la racine. Vous pouvez alors passer ce tableau à un constructeur `TreePath`.

Par exemple, voici comment rendre visible un nœud :

```
TreeNode[] nodes = model.getPathToRoot(newNode);
TreePath path = new TreePath(nodes);
tree.makeVisible(path);
```

INFO

Il est assez étrange que la classe `DefaultTreeModel` fasse semblant d'ignorer la classe `TreePath`, même si son travail est de communiquer avec un `JTree`. La classe `JTree` se sert beaucoup de chemins d'arbre, alors qu'elle n'utilise jamais de tableaux d'objets de nœuds.

Mais supposons maintenant que votre arbre fasse partie d'un panneau d'affichage déroulant. Après l'expansion des nœuds de l'arbre, le nouveau nœud risque une nouvelle fois de ne pas être visible parce qu'il peut se trouver en dehors de la zone visible du panneau. Pour résoudre ce problème, appelez :

```
tree.scrollPathToVisible(path);
```

au lieu d'appeler `makeVisible`. Cet appel ouvre tous les nœuds du chemin et demande au panneau déroulant de se positionner sur le nœud situé à la fin du chemin (voir Figure 5.20).

Figure 5.20

Le panneau déroulant se place sur le nouveau nœud.



Défilement dans l'affichage

Par défaut, les nœuds d'un arbre peuvent être modifiés. Cependant, si vous appelez

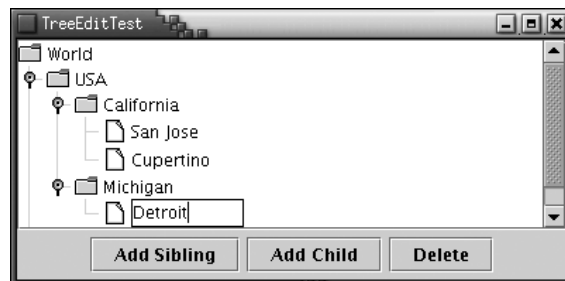
```
tree.setEditable(true);
```

l'utilisateur peut modifier un nœud en double-cliquant simplement dessus, en modifiant la chaîne, puis en appuyant sur la touche Entrée. Cela invoque l'éditeur de cellules par défaut, qui est implémenté par la classe `DefaultCellEditor` (voir Figure 5.21). Il est possible d'installer d'autres éditeurs de cellules, mais nous préférons reporter notre étude sur les éditeurs de cellules à la section concernant les tableaux, avec lesquels les éditeurs de cellules sont plus couramment utilisés.

326 Au cœur de Java 2 - Fonctions avancées

Figure 5.21

*L'éditeur de cellules
par défaut.*



L'Exemple 5.5 fournit le code complet du programme d'édition d'arbres. Exécutez ce programme, ajoutez quelques nœuds, et modifiez-les en double-cliquant dessus. Vous remarquerez la façon dont les nœuds cachés apparaissent pour que l'enfant ajouté soit visible, et la façon dont le panneau déroulant s'organise pour que les nouveaux nœuds soient toujours visibles.

Exemple 5.5 : *TreeEditTest.java*

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;

/**
 * Ce programme montre la modification d'arbre.
 */
public class TreeEditTest
{
    public static void main(String[] args)
    {
        JFrame frame = new TreeEditFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Un bloc avec un arbre et des boutons pour modifier l'arbre.
 */
class TreeEditFrame extends JFrame
{
    public TreeEditFrame()
    {
        setTitle("TreeEditTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // construit l'arbre

        TreeNode root = makeSampleTree();
        model = new DefaultTreeModel(root);
        tree = new JTree(model);
        tree.setEditable(true);

        // ajoute un panneau déroulant contenant un arbre
    }
}
```

```

        JScrollPane scrollPane = new JScrollPane(tree);
        add(scrollPane, BorderLayout.CENTER);

        makeButtons();
    }

    public TreeNode makeSampleTree()
    {
        DefaultMutableTreeNode root
            = new DefaultMutableTreeNode("Monde");
        DefaultMutableTreeNode country
            = new DefaultMutableTreeNode("USA");
        root.add(country);
        DefaultMutableTreeNode state
            = new DefaultMutableTreeNode("Californie");
        country.add(state);
        DefaultMutableTreeNode city
            = new DefaultMutableTreeNode("San José");
        state.add(city);
        city = new DefaultMutableTreeNode("San Diego");
        state.add(city);
        state = new DefaultMutableTreeNode("Michigan");
        country.add(state);
        city = new DefaultMutableTreeNode("Ann Arbor");
        state.add(city);
        country = new DefaultMutableTreeNode("Allemagne");
        root.add(country);
        state = new DefaultMutableTreeNode("Schleswig-Holstein");
        country.add(state);
        city = new DefaultMutableTreeNode("Kiel");
        state.add(city);
        return root;
    }

    /**
     * Crée des boutons pour ajouter un frère, un enfant et
     * effacer un nœud.
     */
    public void makeButtons()
    {
        JPanel panel = new JPanel();
        JButton addSiblingButton = new JButton("Ajouter un frère");
        addSiblingButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    DefaultMutableTreeNode selectedNode
                        = (DefaultMutableTreeNode)
                            tree.getLastSelectedPathComponent();

                    if (selectedNode == null) return;

                    DefaultMutableTreeNode parent
                        = (DefaultMutableTreeNode)
                            selectedNode.getParent();

                    if (parent == null) return;
                }
            }
        );
    }

```

328 *Au cœur de Java 2 - Fonctions avancées*

```

        DefaultMutableTreeNode newNode
            = new DefaultMutableTreeNode("Nouveau");

        int selectedIndex = parent.getIndex(selectedNode);
        model.insertNodeInto(newNode, parent,
            selectedIndex + 1);

        // Affiche maintenant le nouveau nœud

        TreeNode[] nodes = model.getPathToRoot(newNode);
        TreePath path = new TreePath(nodes);
        tree.scrollPathToVisible(path);
    }

    });
    panel.add(addSiblingButton);

    JButton addChildButton = new JButton("Ajouter un enfant");
    addChildButton.addActionListener(new
        ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                DefaultMutableTreeNode selectedNode
                    = (DefaultMutableTreeNode)
                        tree.getLastSelectedPathComponent();

                if (selectedNode == null) return;

                DefaultMutableTreeNode newNode
                    = new DefaultMutableTreeNode("Nouveau");
                model.insertNodeInto(newNode, selectedNode,
                    selectedNode.getChildCount());

                // affiche maintenant le nouveau noeud

                TreeNode[] nodes = model.getPathToRoot(newNode);
                TreePath path = new TreePath(nodes);
                tree.scrollPathToVisible(path);
            }
        });
    panel.add(addChildButton);

    JButton deleteButton = new JButton("Supprimer");
    deleteButton.addActionListener(new
        ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                DefaultMutableTreeNode selectedNode
                    = (DefaultMutableTreeNode)
                        tree.getLastSelectedPathComponent();

                if (selectedNode != null &&
                    selectedNode.getParent() != null)
                    model.removeNodeFromParent(selectedNode);
            }
        });
    });

```

```

        panel.add(deleteButton);
        add(panel, BorderLayout.SOUTH);
    }

    private DefaultTreeModel model;
    private JTree tree;
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 200;
}

```

javax.swing.JTree 1.2

- `TreePath getSelectionPath()`
Renvoie le chemin du nœud sélectionné ou le chemin du premier nœud sélectionné si plusieurs nœuds sont sélectionnés. Renvoie null si aucun nœud n'est sélectionné.
- `Object getLastSelectedPathComponent()`
Renvoie l'objet du nœud qui représente le nœud sélectionné, ou le premier nœud si plusieurs nœuds sont sélectionnés. Renvoie null si aucun nœud n'est sélectionné.
- `void makeVisible(TreePath path)`
Ouvre tous les nœuds du chemin spécifié.
- `void scrollPathToVisible(TreePath path)`
Ouvre tous les nœuds du chemin spécifié. De plus, si l'arbre se trouve dans un panneau déroulant, ce dernier se positionne sur le dernier nœud du chemin pour qu'il soit directement visible.

javax.swing.tree.TreePath 1.2

- `Object getLastPathComponent()`
Renvoie le dernier objet du chemin spécifié, c'est-à-dire l'objet du nœud représenté par le chemin.

javax.swing.tree.TreeNode 1.2

- `TreeNode getParent()`
Renvoie le nœud parent de ce nœud.
- `TreeNode getChildAt(int index)`
Cherche le nœud enfant correspondant à l'indice spécifié. Cet indice doit être compris entre 0 et `getChildCount() - 1`.
- `int getChildCount()`
Renvoie le nombre d'enfants de ce nœud.
- `Enumeration children()`
Renvoie un objet d'énumération qui passe en revue tous les enfants de ce nœud.

javax.swing.tree.DefaultTreeModel 1.2

- `void insertNodeInto(MutableTreeNode newChild, MutableTreeNode parent, int index)`
Insère un nouveau nœud enfant (`newChild`) au nœud parent à l'indice spécifié et avertit les écouteurs de modèle d'arbre.
- `void removeNodeFromParent(MutableTreeNode node)`
Supprime le nœud `node` de ce modèle et avertit les écouteurs de modèle d'arbre.

330 *Au cœur de Java 2 - Fonctions avancées*

- `void nodeChanged(TreeNode node)`
Avertit le processus d'écoute du modèle d'arbre que le nœud `node` a été modifié.
- `void nodesChanged(TreeNode parent, int[] changedChildIndexes)`
Avertit le processus d'écoute du modèle d'arbre que tous les enfants du nœud `parent` aux indices spécifiés ont été modifiés.
- `void reload()`
Recharge tous les nœuds dans le modèle. Il s'agit d'une opération très coûteuse en temps et, par conséquent, il est conseillé de ne l'utiliser que si les nœuds ont complètement changé, à cause d'une influence extérieure.

Enumération de nœuds

Il arrive parfois que vous deviez trouver un nœud dans un arbre, en partant de la racine et en passant en revue tous les enfants jusqu'à ce que vous ayez trouvé le nœud qui vous intéresse. La classe `DefaultMutableTreeNode` possède plusieurs méthodes pratiques pour parcourir les nœuds d'un arbre.

Les méthodes `breadthFirstEnumeration` et `depthFirstEnumeration` renvoient des objets d'énumération dont la méthode `nextElement` parcourt tous les enfants du nœud courant, en utilisant soit une approche horizontale, soit une approche verticale. La Figure 5.22 montre chacune de ces approches pour un arbre donné, dans lequel les étiquettes des nœuds indiquent l'ordre dans lequel les nœuds sont parcourus.

L'approche horizontale est la plus simple à visualiser. L'arbre est parcouru par niveaux, en commençant par la racine, suivie de tous ses enfants, puis de tous ses petits-enfants, etc.

Pour visualiser une énumération verticale, imaginez qu'un rat soit emprisonné dans un labyrinthe en forme d'arbre. Il descend l'arbre jusqu'à ce qu'il trouve une feuille, puis il remonte d'un niveau et parcourt la prochaine branche, etc.

Cette approche est aussi appelée une *traversée postérieure* en informatique, parce que la recherche commence par les enfants avant d'arriver aux parents. La méthode `postOrderTraversal` est donc équivalente à la méthode `depthFirstTraversal`. Pour que la bibliothèque soit complète, il existe aussi une méthode `preOrderTraversal`, une recherche verticale qui passe en revue les parents avant les enfants.

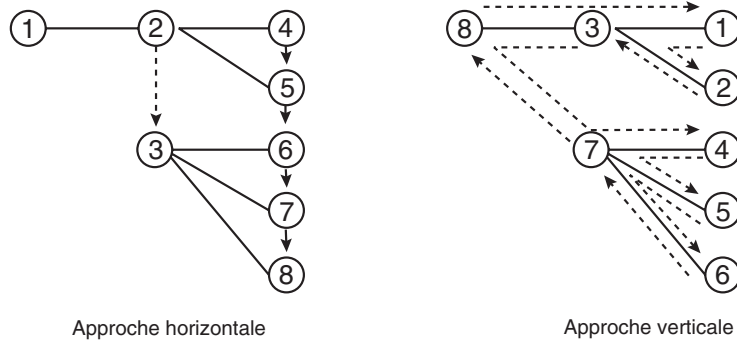
Voici un modèle d'utilisation typique :

```
Enumeration breadthFirst = node.breadthFirstEnumeration();
while (breadthFirst.hasMoreElements())
    utilisation de breadthFirst.nextElement();
```

Pour terminer, il existe une méthode `pathFromAncestorEnumeration` qui trouve un chemin entre un ancêtre et un nœud spécifié, puis parcourt tous les nœuds se trouvant sur ce chemin. Cette méthode est assez simple, en fait elle se contente d'appeler la méthode `getParent` jusqu'à ce que l'ancêtre spécifié soit trouvé, puis elle affiche ensuite en sens inverse le chemin parcouru.

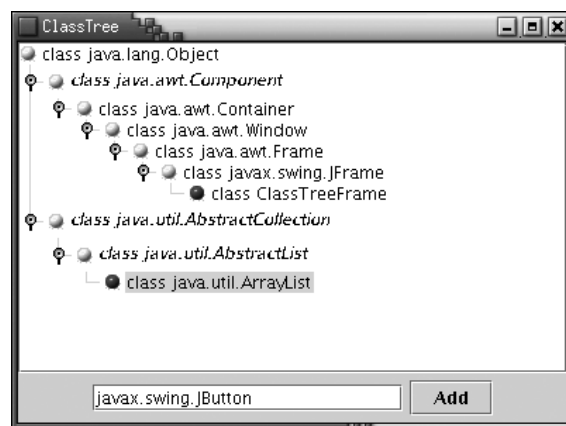
Dans notre prochain programme d'exemple, nous nous servons des énumérations de nœuds. Ce programme affiche les arbres d'héritage des classes. Il suffit de saisir le nom d'une classe dans le

Figure 5.22
Les ordres de traversée
d'un arbre.



champ de texte en bas de la fenêtre. La classe et toutes ses superclasses sont ajoutées dans un arbre (voir Figure 5.23).

Figure 5.23
Un arbre d'héritage.



Dans cet exemple, nous tirons profit du fait que les objets d'utilisateur des nœuds de l'arbre peuvent être des objets de n'importe quel type. Comme nos nœuds décrivent des classes, nous enregistrerons des objets `Class` dans les nœuds.

Naturellement, nous voulons éviter d'ajouter deux fois le même objet (la même classe), donc nous devons vérifier qu'une classe est présente dans l'arbre ou non. La méthode suivante trouve le nœud d'un objet d'utilisateur spécifié si celui-ci existe dans l'arbre :

```
public DefaultMutableTreeNode findUserObject(Object obj)
{
    Enumeration e = root.breadthFirstEnumeration();
    while (e.hasMoreElements())
    {
        DefaultMutableTreeNode node
            = (DefaultMutableTreeNode)e.nextElement();
        if (node.getUserObject().equals(obj))
            return node;
    }
    return null;
}
```

332 Au cœur de Java 2 - Fonctions avancées

Afficher les nœuds

Dans vos applications, vous serez souvent amené à modifier la manière dont un composant d'un arbre représente les nœuds. La modification la plus courante est naturellement la possibilité de choisir plusieurs icônes pour les nœuds et pour les feuilles. Les autres changements peuvent être en rapport avec la police utilisée ou l'affichage d'images sur chaque nœud. Toutes ces modifications sont possibles si vous prenez la peine d'installer un nouvel *afficheur de cellules d'arbre* dans votre arbre. Par défaut, la classe `JTree` se sert d'objets `DefaultTreeCellRenderer` pour afficher chaque nœud. La classe `DefaultTreeCellRenderer` étend la classe `JLabel`. Une étiquette contient l'icône d'un nœud et le nom de ce nœud.

INFO

L'afficheur de cellules n'affiche pas les poignées permettant de savoir si un nœud est ouvert ou fermé. Ces poignées font partie de l'aspect, et il est recommandé de ne pas les changer.

Vous pouvez personnaliser l'affichage de trois manières différentes :

1. Vous pouvez modifier les icônes, la police et la couleur de fond utilisées par un `DefaultTreeCellRenderer`. Ces paramètres sont utilisés pour tous les nœuds d'un arbre.
2. Vous pouvez installer un afficheur qui étend la classe `DefaultTreeCellRenderer` et modifier les icônes, les polices et la couleur de fond de chaque nœud.
3. Vous pouvez installer un afficheur qui implémente l'interface `TreeCellRenderer`, pour afficher une nouvelle image pour chaque nœud.

Examinons maintenant ces possibilités en détail. La personnalisation la plus simple consiste à construire un objet `DefaultTreeCellRenderer`, à modifier les icônes et à l'installer dans l'arbre :

```
DefaultTreeCellRenderer renderer
= new DefaultTreeCellRenderer();
renderer.setLeafIcon(new ImageIcon("blue-ball.gif"));
// utilisé pour les feuilles
renderer.setClosedIcon(new ImageIcon("red-ball.gif"));
// utilisé pour les nœuds cachés
renderer.setOpenIcon(new ImageIcon("yellow-ball.gif"));
// utilisé pour les nœuds ouverts
tree.setCellRenderer(renderer);
```

Vous pouvez en constater l'effet grâce à la Figure 5.23. Nous nous servons simplement d'icônes en forme de balles, mais le concepteur de votre interface utilisateur vous fournira probablement les icônes appropriées pour vos applications.

Nous ne vous recommandons pas de modifier la police ou la couleur de fond d'un arbre entier, parce que cette tâche revient plutôt à l'aspect choisi.

Cependant, il peut être intéressant de modifier la police de certains nœuds, pour les mettre en valeur. Si vous examinez avec attention la Figure 5.23, vous remarquerez que les classes *abstraites* sont en italique.

Pour modifier l'apparence de certains nœuds, vous devez installer un afficheur de cellules d'arbre. Ce type d'afficheur ressemble beaucoup aux afficheurs de cellules de liste que nous avons abordés plus tôt dans ce chapitre. L'interface `TreeCellRenderer` possède une seule méthode :

```
Component getTreeCellRendererComponent(JTree tree,  
    Object value, boolean selected, boolean expanded,  
    boolean leaf, int row, boolean hasFocus)
```

La méthode `getTreeCellRendererComponent` de la classe `DefaultTreeCellRenderer` renvoie `this`, c'est-à-dire une étiquette. La classe `DefaultTreeCellRenderer` étend la classe `JLabel`. Pour personnaliser le composant, il faut étendre la classe `DefaultTreeCellRenderer` et surcharger la méthode `getTreeCellRendererComponent` comme ceci : appelez la méthode de la superclasse, pour préparer les données de l'étiquette. Personnalisez ensuite les propriétés de l'étiquette et renvoyez `this` pour terminer.

```
class MyTreeCellRenderer extends DefaultTreeCellRenderer  
{  
    public Component getTreeCellRendererComponent(JTree tree,  
        Object value, boolean selected, boolean expanded,  
        boolean leaf, int row, boolean hasFocus)  
    {  
        super.getTreeCellRendererComponent(tree, value,  
            selected, expanded, leaf, row, hasFocus);  
        DefaultMutableTreeNode node  
            = (DefaultMutableTreeNode)value;  
        examine node.getUserObject();  
        Font font = police appropriée;  
        setFont(font);  
        return this;  
    }  
};
```

ATTENTION

Le paramètre `value` de la méthode `getTreeCellRendererComponent` est l'objet *nœud*, et non l'objet de l'utilisateur ! Rappelez-vous que l'objet de l'utilisateur est une caractéristique de `DefaultMutableTreeNode`, et qu'un `JTree` peut contenir des nœuds de n'importe quel type. Si votre arbre se sert de nœuds `DefaultMutableTreeNode`, vous devez traiter l'objet de l'utilisateur dans une seconde étape, exactement comme nous l'avons fait dans l'exemple précédent.

ATTENTION

`DefaultTreeCellRenderer` se sert d'un seul objet d'étiquette pour tous les nœuds, et il ne modifie le texte de l'étiquette que d'un seul nœud. Si vous souhaitez modifier la police d'un nœud particulier, vous devez lui redonner sa valeur par défaut lorsque la méthode est appelée à nouveau. Autrement, tous les nœuds suivants seront affichés avec la nouvelle police. Reportez-vous à l'Exemple 5.6 pour voir comment restaurer la valeur par défaut de la police.

Nous ne fournirons pas d'exemple d'afficheur de cellules d'arbre qui affiche des dessins particuliers. Si vous avez besoin de cette fonctionnalité, vous pouvez adapter les afficheurs de cellules de liste de l'Exemple 5.3 : la technique à utiliser est parfaitement identique.

Examinons maintenant le fonctionnement d'un afficheur de cellules d'arbre. L'Exemple 5.6 fournit le code source complet de ce programme, qui affiche la hiérarchie des héritages, et qui personnalise l'affichage pour que les classes abstraites soient en italique. Vous pouvez saisir le nom de n'importe quelle classe dans le champ de texte en bas de l'application. Appuyez sur la touche Entrée ou cliquez

334 Au cœur de Java 2 - Fonctions avancées

sur le bouton "Ajouter" pour ajouter une classe et ses superclasses dans l'arbre. Vous devrez saisir des noms complets de bibliothèque, comme `java.util.ArrayList`.

Ce programme se sert d'une petite astuce : il utilise une sorte de réflexion pour construire l'arbre des classes. Ce travail revient à la méthode `addClass`. Les détails de cette méthode ne sont pas très importants. Il vous suffit de savoir que nous avons choisi un arbre de classes parce que nous n'avons pas besoin de fournir les données à placer dans l'arbre. Si vous affichez des arbres dans vos applications, vous posséderez déjà une source de données hiérarchisées. La méthode `addClass` commence par une recherche horizontale pour trouver si la classe courante existe déjà dans l'arbre, en appelant la méthode `findUserObject` que nous venons d'implémenter dans la section précédente. Si la classe ne se trouve pas déjà dans l'arbre, nous ajoutons les superclasses dans l'arbre, puis nous transformons le nœud de la nouvelle classe en enfant que nous rendons visible.

`ClassNameTreeCellRenderer` définit le nom d'une classe, soit en italique, soit avec la police standard, en fonction du modificateur `ABSTRACT` de l'objet `Class`. Nous n'avons pas besoin de définir une police particulière puisque nous voulons conserver la police standard de l'aspect choisi. Pour cette raison, nous nous servons de la police standard de l'étiquette, et nous *dérivons* une police en italique à partir de cette police. Rappelez-vous qu'il n'existe qu'un seul objet `JLabel` qui est partagé et renvoyé par tous les appels. Nous devons donc conserver la police originale et la restaurer avec le prochain appel à la méthode `getTreeCellRendererComponent`.

Pour terminer, notez de quelle manière nous modifions les icônes du constructeur `ClassTreeFrame`.

Exemple 5.6 : *ClassTree.java*

```
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;

/**
 * Ce programme montre l'affichage de cellule en présentant
 * un arbre de classes et leurs superclasses.
 */
public class ClassTree
{
    public static void main(String[] args)
    {
        JFrame frame = new ClassTreeFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc affiche l'arbre de classe ainsi qu'un champ de texte et
 * ajoute un bouton pour ajouter des classes à l'arbre.
 */
class ClassTreeFrame extends JFrame
{
    public ClassTreeFrame()
    {
        setTitle("ClassTree");
    }
}
```

```
setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

// la racine de l'arbre est Object
root = new DefaultMutableTreeNode(java.lang.Object.class);
model = new DefaultTreeModel(root);
tree = new JTree(model);

// ajoute cette classe pour placer des données dans l'arbre
addClass(getClass());

// définit les icônes des nœuds
ClassNameTreeCellRenderer renderer
    = new ClassNameTreeCellRenderer();
renderer.setClosedIcon(new ImageIcon("red-ball.gif"));
renderer.setOpenIcon(new ImageIcon("yellow-ball.gif"));
renderer.setLeafIcon(new ImageIcon("blue-ball.gif"));
tree.setCellRenderer(renderer);

add(new JScrollPane(tree), BorderLayout.CENTER);

addTextField();
}

/**
 * Ajoute le champ de texte et le bouton "Ajouter" pour ajouter
 * une nouvelle classe.
 */
public void addTextField()
{
    JPanel panel = new JPanel();

    ActionListener addListener = new
        ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                // ajoute la classe dont le nom se trouve dans le champ de
                // texte
                try
                {
                    String text = textField.getText();
                    addClass(Class.forName(text));
                    // efface le champ de texte pour indiquer le succès de
                    // l'opération
                    textField.setText("");
                }
                catch (ClassNotFoundException e)
                {
                    JOptionPane.showMessageDialog(null,
                        "Classe introuvable");
                }
            }
        };

    // les nouveaux noms de classes sont saisis dans ce champ de texte
    textField = new JTextField(20);
    textField.addActionListener(addListener);
    panel.add(textField);
}
```

336 *Au cœur de Java 2 - Fonctions avancées*

```

        JButton addButton = new JButton("Ajouter");
        addButton.addActionListener(addListener);
        panel.add(addButton);

        add(panel, BorderLayout.SOUTH);
    }

    /**
     * Retrouve un objet dans l'arbre.
     * @param obj l'objet à retrouver
     * @return le noeud contenant l'objet ou null
     *         si l'objet ne figure pas dans l'arbre
     */
    public DefaultMutableTreeNode findUserObject(Object obj)
    {
        // trouve le nœud contenant un objet utilisateur
        Enumeration e = root.breadthFirstEnumeration();
        while (e.hasMoreElements())
        {
            DefaultMutableTreeNode node
                = (DefaultMutableTreeNode)e.nextElement();
            if (node.getUserObject().equals(obj))
                return node;
        }
        return null;
    }

    /**
     * Ajoute une nouvelle classe et toute classe parent qui ne fait pas
     * encore partie de l'arbre.
     * @param c la classe à ajouter
     * @return le noeud nouvellement ajouté
     */
    public DefaultMutableTreeNode addClass(Class c)
    {
        // ajoute une nouvelle classe dans l'arbre

        // saute les types qui ne sont pas des classes
        if (c.isInterface() || c.isPrimitive()) return null;

        // si la classe se trouve déjà dans l'arbre, renvoie ce nœud
        DefaultMutableTreeNode node = findUserObject(c);
        if (node != null) return node;

        // la classe n'est pas présente, ajoute d'abord le parent de la
        // classe de manière récursive

        Class s = c.getSuperclass();

        DefaultMutableTreeNode parent;
        if (s == null)
            parent = root;
        else
            parent = addClass(s);

        // ajoute la classe comme enfant du parent
        DefaultMutableTreeNode newNode
            = new DefaultMutableTreeNode(c);
        model.insertNodeInto(newNode, parent,

```

```

        parent.getChildCount());

        // rend le nœud visible
        TreePath path = new TreePath(model.getPathToRoot(newNode));
        tree.makeVisible(path);

        return newNode;
    }

    private DefaultMutableTreeNode root;
    private DefaultTreeModel model;
    private JTree tree;
    private JTextField textField;
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 300;
}

/**
 * Cette classe affiche un nom de classe en normal ou italique.
 * Les classes abstraites sont en italique.
 */
class ClassNameTreeCellRenderer extends DefaultTreeCellRenderer
{
    public Component getTreeCellRendererComponent(JTree tree,
        Object value, boolean selected, boolean expanded,
        boolean leaf, int row, boolean hasFocus)
    {
        super.getTreeCellRendererComponent(tree, value,
            selected, expanded, leaf, row, hasFocus);
        // récupère l'objet utilisateur
        DefaultMutableTreeNode node
            = (DefaultMutableTreeNode)value;
        Class c = (Class)node.getUserObject();

        // la première fois, dérive une police italique à partir de la
        // police normale
        if (plainFont == null)
        {
            plainFont = getFont();
            // l'afficheur de cellule d'arbre est parfois appelé avec une
            // étiquette sans police
            if (plainFont != null)
                italicFont = plainFont.deriveFont(Font.ITALIC);
        }

        // transforme la police en italique si la classe est abstraite,
        // reste normale dans le cas contraire
        if ((c.getModifiers() & Modifier.ABSTRACT) == 0)
            setFont(plainFont);
        else
            setFont(italicFont);
        return this;
    }

    private Font plainFont = null;
    private Font italicFont = null;
}

```


338 *Au cœur de Java 2 - Fonctions avancées*

javax.swing.tree.DefaultMutableTreeNode 1.2

- Enumeration `breadthFirstEnumeration()`
- Enumeration `depthFirstEnumeration()`
- Enumeration `preOrderEnumeration()`
- Enumeration `postOrderEnumeration()`

Renvoient des objets d'énumération pour parcourir tous les nœuds du modèle d'arbre dans un ordre particulier. Avec un parcours horizontal, les enfants qui se trouvent près de la racine sont examinés avant ceux qui se trouvent loin de la racine. Avec un parcours vertical, tous les enfants d'un nœud sont parcourus avant que ses frères ne soient examinés. La méthode `postOrderEnumeration` est équivalente à `depthFirstEnumeration`, et la méthode `preOrderTraversal` correspond à la méthode `postOrderTraversal` dans laquelle les parents sont examinés avant leurs enfants.

javax.swing.tree.TreeCellRenderer 1.2

- Component `getTreeCellRendererComponent(JTree tree, Object value, boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus)`

Renvoie un composant dont la méthode `paint` est invoquée pour afficher une cellule d'arbre.

<i>Paramètres :</i>	<code>tree</code>	l'arbre contenant le nœud à afficher
	<code>value</code>	le nœud à afficher
	<code>selected</code>	true si le nœud est sélectionné
	<code>expanded</code>	true si les enfants du nœud sont visibles
	<code>leaf</code>	true si le nœud doit être affiché comme un arbre
	<code>row</code>	la ligne d'affichage contenant le nœud
	<code>hasFocus</code>	true si le nœud courant possède le focus

javax.swing.tree.DefaultTreeCellRenderer 1.2

- void `setLeafIcon(Icon icon)`
- void `setOpenIcon(Icon icon)`
- void `setClosedIcon(Icon icon)`

Définissent l'icône à utiliser pour une feuille, un nœud ouvert et un nœud fermé.

Ecouter les événements des arbres

Le plus fréquemment, un composant d'arbre est couplé avec un autre composant. Lorsque l'utilisateur sélectionne des nœuds de l'arbre, certaines informations s'affichent dans une autre fenêtre. La Figure 5.24 en montre un exemple. Lorsque l'utilisateur sélectionne une classe, les variables d'instance et les variables statiques de cette classe sont affichées dans la zone de texte à droite.

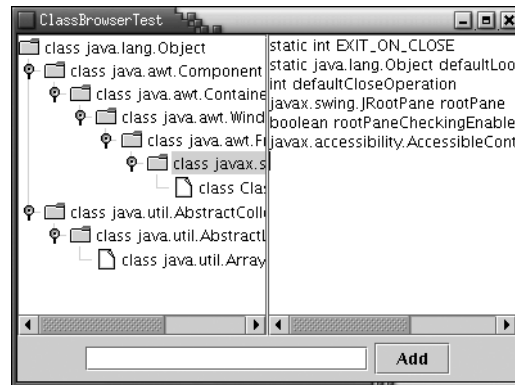
Pour obtenir ce comportement, il suffit d'installer un *écouteur de sélection* de l'arbre. Cet écouteur doit implémenter l'interface `TreeSelectionListener`, une interface possédant une seule méthode :

```
void valueChanged(TreeSelectionEvent event)
```

Cette méthode est appelée lorsque l'utilisateur sélectionne ou désélectionne des nœuds de l'arbre.

Figure 5.24

Un navigateur de classes.



L'écouteur est ajouté à l'arbre de manière classique :

```
tree.addTreeSelectionListener(listener);
```

Vous pouvez autoriser l'utilisateur à sélectionner un seul nœud, une zone continue de nœuds ou un ensemble arbitraire et potentiellement discontinu de nœuds. La classe `JTree` se sert d'un `TreeSelectionModel` pour gérer la sélection des nœuds. Vous devrez retrouver le modèle pour définir l'état de sélection avec l'une des valeurs suivantes : `SINGLE_TREE_SELECTION`, `CONTIGUOUS_TREE_SELECTION` ou `DISCONTIGUOUS_TREE_SELECTION`. Le mode de sélection discontinu est le mode par défaut. Par exemple, dans notre navigateur de classes, nous permettons uniquement à l'utilisateur de sélectionner une seule classe :

```
int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
tree.getSelectionModel().setSelectionMode(mode);
```

Lorsque vous aurez défini un mode de sélection, vous n'aurez plus à vous préoccuper du modèle de sélection de l'arbre.

INFO

La manière dont l'utilisateur peut sélectionner plusieurs éléments dépend de l'aspect choisi. Avec l'aspect Metal, maintenez la touche `Ctrl` enfoncée et cliquez sur les éléments à ajouter à la sélection ou à désélectionner s'ils sont déjà sélectionnés. Servez-vous de la touche `Maj` pour sélectionner une zone d'éléments allant de l'élément sélectionné à l'élément sur lequel vous cliquez.

Pour récupérer la sélection courante, il faut interroger l'arbre avec la méthode `getSelectionPaths` :

```
TreePath[] selectedPaths = tree.getSelectionPaths();
```

Si vous limitez l'utilisateur à une seule sélection, vous pouvez avoir recours à la méthode pratique `getSelectionPath`, qui renvoie le premier chemin sélectionné, ou `null` si aucun chemin n'a été sélectionné.

ATTENTION

La classe `TreeSelectionEvent` possède une méthode `getPaths` qui renvoie un tableau d'objets `TreePath`, mais ce tableau décrit les *modifications de la sélection*, et non la sélection courante.

340 *Au cœur de Java 2 - Fonctions avancées*

L'Exemple 5.7 met en évidence le fonctionnement des arbres. Ce programme est fondé sur l'Exemple 5.6. Cependant, pour réduire la taille de ce programme, nous ne nous sommes pas servis d'un afficheur de cellules d'arbre personnalisé. Dans le constructeur du cadre de l'application, nous limitons l'utilisateur à des sélections simples et nous installons un écouteur de sélection. Lorsque la méthode `valueChanged` est appelée, nous ignorons simplement son paramètre d'événement et nous nous contentons de demander à l'arbre le chemin de la sélection courante. Comme toujours, nous devons chercher le dernier nœud du chemin et récupérer son objet d'utilisateur. Nous appelons ensuite la méthode `getFieldDescription`, qui se sert d'une réflexion pour générer une chaîne contenant tous les champs de la classe sélectionnée. Pour terminer, cette chaîne est affichée dans la zone de texte.

Exemple 5.7 : *ClassBrowserTree.java*

```
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;

/**
 * Ce programme présente des événements de sélection d'arbres.
 */
public class ClassBrowserTest
{
    public static void main(String[] args)
    {
        JFrame frame = new ClassBrowserTestFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Un bloc avec un arbre de classe, une zone de texte présentant les
 * propriétés de la classe sélectionnée et un champ de texte pour ajouter
 * de nouvelles classes.
 */
class ClassBrowserTestFrame extends JFrame
{
    public ClassBrowserTestFrame()
    {
        setTitle("ClassBrowserTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // la racine de l'arbre est Object
        root = new DefaultMutableTreeNode(java.lang.Object.class);
        model = new DefaultTreeModel(root);
        tree = new JTree(model);

        // ajoute cette classe pour remplir l'arbre avec quelques données
        addClass(getClass());

        // définit le mode de sélection
        tree.addTreeSelectionListener(new
```

```

TreeSelectionListener()
{
    public void valueChanged(TreeSelectionEvent event)
    {
        // l'utilisateur a sélectionné un noeud différent
        // --mise à jour de la description
        TreePath path = tree.getSelectionPath();
        if (path == null) return;
        DefaultMutableTreeNode selectedNode
            = (DefaultMutableTreeNode)
                path.getLastPathComponent();
        Class c = (Class)selectedNode.getUserObject();
        String description = getFieldDescription(c);
        textArea.setText(description);
    }
});
int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
tree.getSelectionModel().setSelectionMode(mode);

// cette zone de texte contient la description de la classe
textArea = new JTextArea();

// ajoute l'arbre et la zone de texte au panneau
JPanel panel = new JPanel();
panel.setLayout(new GridLayout(1, 2));
panel.add(new JScrollPane(tree));
panel.add(new JScrollPane(textArea));

add(panel, BorderLayout.CENTER);

addTextField();
}

/**
 * Ajoute le champ de texte et le bouton "Ajouter" pour ajouter une
 * nouvelle classe.
 */
public void addTextField()
{
    JPanel panel = new JPanel();

    ActionListener addListener = new
        ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                // ajoute la classe dont le nom se trouve dans le champ de
                // texte
                try
                {
                    String text = textField.getText();
                    addClass(Class.forName(text));
                    // efface le champ de texte pour indiquer le succès de
                    // l'opération
                    textField.setText("");
                }
            }
        }
}

```

342 *Au cœur de Java 2 - Fonctions avancées*

```

        catch (ClassNotFoundException e)
        {
            JOptionPane.showMessageDialog(null,
                "Classe introuvable");
        }
    }
};

// les noms des nouvelles classes sont saisis dans ce champ de
// texte
textField = new JTextField(20);
textField.addActionListener(addListener);
panel.add(textField);

JButton addButton = new JButton("Ajouter");
addButton.addActionListener(addListener);
panel.add(addButton);

add(panel, BorderLayout.SOUTH);
}

/**
 * Retrouve un objet dans l'arbre.
 * @param obj l'objet à trouver
 * @return le noeud contenant l'objet ou null
 *         si l'objet ne figure pas dans l'arbre
 */
public DefaultMutableTreeNode findUserObject(Object obj)
{
    // trouve le nœud contenant un objet utilisateur
    Enumeration e = root.breadthFirstEnumeration();
    while (e.hasMoreElements())
    {
        DefaultMutableTreeNode node
            = (DefaultMutableTreeNode)e.nextElement();
        if (node.getUserObject().equals(obj))
            return node;
    }
    return null;
}

/**
 * Ajoute une nouvelle classe et toute classe parent qui ne fait pas
 * encore partie de l'arbre.
 * @param c la classe à ajouter
 * @return le noeud nouvellement ajouté
 */
public DefaultMutableTreeNode addClass(Class c)
{
    // ajoute une nouvelle classe dans l'arbre

    // saute les types qui ne sont pas des classes
    if (c.isInterface() || c.isPrimitive()) return null;

    // Si la classe est déjà dans l'arbre, renvoie ce nœud
    DefaultMutableTreeNode node = findUserObject(c);
    if (node != null) return node;
}

```

```
// la classe n'est pas présente, commence par ajouter le parent de
// la classe récursivement

Class s = c.getSuperclass();

DefaultMutableTreeNode parent;
if (s == null)
    parent = root;
else
    parent = addClass(s);

// ajoute la classe comme enfant du parent
DefaultMutableTreeNode newNode
    = new DefaultMutableTreeNode(c);
model.insertNodeInto(newNode, parent,
    parent.getChildCount());

// rend le nœud visible
TreePath path = new TreePath(model.getPathToRoot(newNode));
tree.makeVisible(path);

return newNode;
}

/**
 * Renvoie une description des champs d'une classe.
 * @param la classe à décrire
 * @return une chaîne contenant tous les types et noms de champs
 */
public static String getFieldDescription(Class c)
{
    // utilise une réflexion pour trouver les types et les noms des
    // champs
    StringBuilder r = new StringBuilder();
    Field[] fields = c.getDeclaredFields();
    for (int i = 0; i < fields.length; i++)
    {
        Field f = fields[i];
        if ((f.getModifiers() & Modifier.STATIC) != 0)
            r.append("static ");
        r.append(f.getType().getName());
        r.append(" ");
        r.append(f.getName());
        r.append("\n");
    }
    return r.toString();
}

private DefaultMutableTreeNode root;
private DefaultTreeModel model;
private JTree tree;
private JTextField textField;
private JTextArea textArea;
private static final int DEFAULT_WIDTH = 400;
private static final int DEFAULT_HEIGHT = 300;
}
```

344 Au cœur de Java 2 - Fonctions avancées

javax.swing.JTree 1.2

- `TreePath getSelectionPath()`
- `TreePath[] getSelectionPaths()`

Renvoient le premier chemin sélectionné ou un tableau de chemins correspondant à tous les nœuds sélectionnés. Si aucun chemin n'est sélectionné, les deux méthodes renvoient `null`.

javax.swing.event.TreeSelectionListener 1.2

- `void valueChanged(TreeSelectionEvent event)`

Cette méthode est appelée lorsque des nœuds sont sélectionnés ou désélectionnés.

javax.swing.event.TreeSelectionEvent 1.2

- `TreePath getPath()`
- `TreePath[] getPaths()`

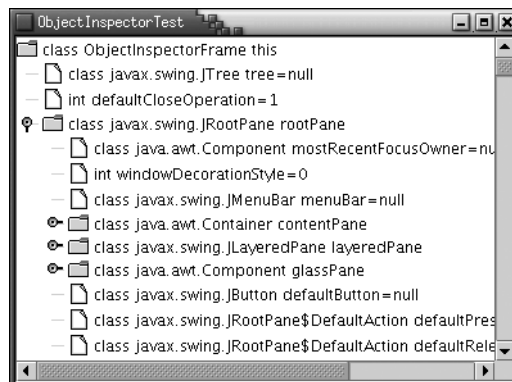
Renvoient le premier chemin ou tous les chemins qui ont été modifiés dans cet événement de sélection. Si vous souhaitez connaître la sélection courante, et non les modifications de sélection, vous devez appeler `JTree.getSelectionPaths()`.

Modèles d'arbre personnalisés

Dans notre dernier exemple, nous avons implémenté un programme qui examine le contenu d'une variable, comme le ferait un débogueur (voir Figure 5.25).

Figure 5.25

Un arbre d'inspection d'objet.



Avant d'aller plus loin, compilez et exécutez le programme d'exemple. Chaque nœud correspond à une variable d'instance. Si la variable est un objet, ouvrez-le pour voir ses variables d'instance. Ce programme examine le contenu de la fenêtre du cadre. Si vous examinez quelques-unes des variables d'instance, vous devriez pouvoir reconnaître quelques classes familières. Vous aurez aussi un meilleur aperçu de la complexité sous-jacente de l'interface utilisateur Swing.

Ce programme présente un aspect remarquable : l'arbre ne se sert pas du modèle par défaut `DefaultTreeModel`. Si vous possédez déjà des données organisées hiérarchiquement, vous n'aurez sûrement pas envie de construire une nouvelle structure sous la forme d'un arbre et de synchroniser en permanence ces deux structures. Nous nous sommes trouvés exactement dans la même situation.

Les objets examinés sont déjà reliés ensemble par des références d'objets, donc il n'y a aucune raison de dupliquer la structure de ces données.

L'interface `TreeModel` ne possède que quelques méthodes. Le premier groupe de méthodes permet à `JTree` de trouver les nœuds en parcourant d'abord la racine, puis ses enfants. La classe `JTree` n'appelle ces méthodes que lorsque l'utilisateur ouvre réellement un nœud.

```
Object getRoot()  
int getChildCount(Object parent)  
Object getChild(Object parent, int index)
```

Notez que l'interface `TreeModel`, comme la classe `JTree`, n'a aucune notion des nœuds ! La racine et ses enfants peuvent correspondre à n'importe quel type d'objet. Le modèle `TreeModel` doit donc indiquer à la classe `JTree` comment ils sont connectés.

La prochaine méthode de l'interface `TreeModel` est l'inverse de `getChild` :

```
int getIndexOfChild(Object parent, Object child)
```

En fait, cette méthode peut être implémentée de trois manières différentes (voir le code de l'Exemple 5.8).

Le modèle d'arbre indique à `JTree` quels nœuds doivent être affichés comme des feuilles.

```
boolean isLeaf(Object node)
```

Si votre code modifie le modèle d'arbre, l'arbre devra en être averti pour qu'il puisse s'afficher à nouveau. L'arbre s'ajoute lui-même au modèle, sous la forme d'un `TreeModelListener`. Par conséquent, le modèle doit supporter les méthodes traditionnelles de gestion des écouteurs :

```
void addTreeModelListener(TreeModelListener l)  
void removeTreeModelListener(TreeModelListener l)
```

Vous trouverez des implémentations de ces méthodes dans l'Exemple 5.8.

Lorsque le modèle modifie le contenu de l'arbre, il appelle l'une des quatre méthodes de l'interface `TreeModelListener` :

```
void treeNodesChanged(TreeModelEvent e)  
void treeNodesInserted(TreeModelEvent e)  
void treeNodesRemoved(TreeModelEvent e)  
void treeStructureChanged(TreeModelEvent e)
```

L'objet `TreeModelEvent` décrit l'emplacement de la modification. Les détails de la génération d'un événement de modèle d'arbre qui décrit une insertion ou une suppression d'un événement sont assez techniques. Il vous suffit de traiter ces événements si les nœuds de votre arbre peuvent réellement être ajoutés ou supprimés. Dans l'Exemple 5.8, nous vous montrerons comment traiter un événement : remplacer la racine par un nouvel objet.

ASTUCE

Pour simplifier le code de génération des événements, nous utilisons la classe `javax.swing.EventListenerList`, qui réunit les écouteurs. Reportez-vous au Chapitre 8 du Volume 1 pour obtenir de plus amples informations sur cette classe.

346 *Au cœur de Java 2 - Fonctions avancées*

Finalement, si l'utilisateur modifie un nœud, votre modèle sera appelé avec la modification apportée :

```
void valueForPathChanged(TreePath path, Object newValue)
```

Si vous ne permettez pas à l'utilisateur de modifier les nœuds, cette méthode ne sera jamais appelée.

Si vous n'avez pas besoin de gérer les modifications, il est très simple de construire un modèle d'arbre. Implémentez les trois méthodes suivantes :

```
Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)
```

Ces méthodes décrivent la structure de l'arbre. Vous devez ensuite fournir l'implémentation des routines des cinq autres méthodes, comme dans l'Exemple 5.8. Vous êtes alors prêt à afficher votre arbre.

Passons maintenant à l'implémentation de notre programme d'exemple. Notre arbre contiendra des objets de type `Variable`.

INFO

Si nous avons utilisé le modèle `DefaultTreeModel`, nos nœuds auraient été des objets de type `DefaultMutableTreeNode`, et les objets de l'utilisateur auraient été de type `Variable`.

Par exemple, supposons que vous examiniez la variable suivante :

```
Employee joe;
```

Cette variable possède un *type*, `Employee.class`, un *nom*, "joe", et une *valeur*, qui est la valeur de la référence d'objet, `joe`. Nous définissons une classe `Variable` qui décrit une variable dans un programme :

```
Variable v = new Variable(Employee.class, "joe", joe);
```

Si le type de la variable est un type primitif, vous devez passer par un emballer d'objet pour sa valeur :

```
new Variable(double.class, "salaire", new Double(salary));
```

Si le type de la variable est une classe, la variable possède des *champs*. En utilisant une réflexion, nous pouvons énumérer tous les champs et les rassembler dans une `ArrayList`. Comme la méthode `getFields` de la classe `Class` ne renvoie pas les champs de la superclasse, nous devons également appeler la méthode `getFields` sur toutes les superclasses. Vous trouverez le code correspondant dans le constructeur `Variable`. La méthode `getFields` de notre classe `Variable` renvoie le tableau des champs. Pour terminer, la méthode `toString` de la classe `Variable` formate les étiquettes des nœuds. Une étiquette contient toujours le type de la variable et son nom. Si la variable n'est pas une classe, l'étiquette contient aussi une valeur.

INFO

Si le type est un tableau, nous n'affichons pas les éléments du tableau. Ce ne serait pas très difficile à faire, nous vous laisserons donc ce plaisir à titre d'entraînement.

Passons maintenant au modèle de l'arbre. Les deux premières méthodes sont simples.

```
public Object getRoot()
{
    return root;
}

public int getChildCount(Object parent)
{
    return ((Variable)parent).getFields().size();
}
```

La méthode `getChild` renvoie un nouvel objet `Variable` qui décrit le champ ayant l'indice donné. Les méthodes `getType` et `getName` de la classe `Field` renvoient le type et le nom du champ. En utilisant une réflexion, vous pouvez lire la valeur du champ : `f.get(parentValue)`. Cette méthode peut déclencher une `IllegalAccessException`. Cependant, tous les champs sont accessibles dans le constructeur `Variable`, donc, cela ne devrait pas se produire en pratique.

Voici le code complet de la méthode `getChild` :

```
public Object getChild(Object parent, int index)
{
    ArrayList fields = ((Variable)parent).getFields();
    Field f = (Field)fields.get(index);
    Object parentValue = ((Variable)parent).getValue();
    try
    {
        return new Variable(f.getType(), f.getName(),
            f.get(parentValue));
    }
    catch(IllegalAccessException e)
    {
        return null;
    }
}
```

Ces trois méthodes révèlent la structure de l'arbre d'objets au composant `JTree`. Les autres méthodes sont classiques, vous en trouverez le code dans l'Exemple 5.8.

Il existe un fait remarquable à propos de ce modèle d'arbre : il décrit en fait un arbre *infini*. Vous pouvez le vérifier en suivant l'un des objets `WeakReference`. Cliquez sur la variable appelée `reference`. Elle vous ramènera directement à l'objet original. Vous obtenez alors un sous-arbre identique, et vous pouvez à nouveau ouvrir son objet `WeakReference`, et ainsi de suite à l'infini. Naturellement, il n'est pas possible d'enregistrer un nombre infini de nœuds. Le modèle d'arbre génère simplement les nœuds au fur et à mesure qu'on le lui demande.

Cet exemple termine notre étude portant sur les arbres. Nous allons passer aux tableaux, un autre composant complexe de Swing. A première vue, les arbres et les tableaux n'ont rien à voir, mais vous allez vous rendre compte qu'ils se servent des mêmes concepts pour les modèles de données et pour l'affichage des cellules.

Exemple 5.8 : *ObjectInspectorTest.java*

```
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.*;
```

348 *Au cœur de Java 2 - Fonctions avancées*

```
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;

/**
 * Ce programme montre comment utiliser un modèle d'arbre personnalisé.
 * Il affiche les champs d'un objet.
 */
public class ObjectInspectorTest
{
    public static void main(String[] args)
    {
        JFrame frame = new ObjectInspectorFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc contient l'arbre d'objets.
 */
class ObjectInspectorFrame extends JFrame
{
    public ObjectInspectorFrame()
    {
        setTitle("ObjectInspectorTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // nous examinons ce cadre

        Variable v = new Variable(getClass(), "this", this);
        ObjectTreeModel model = new ObjectTreeModel();
        model.setRoot(v);

        // construit et affiche l'arbre

        tree = new JTree(model);
        add(new JScrollPane(tree),
            BorderLayout.CENTER);
    }

    private JTree tree;
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 300;
}

/**
 * Ce modèle d'arbre décrit la structure d'arbre d'un objet Java.
 * Les enfants sont les objets stockés dans des variables d'instance.
 */
class ObjectTreeModel implements TreeModel
{
    /**
     * Construit un arbre vide.
     */
    public ObjectTreeModel()
    {
        root = null;
    }
}
```

```

/**
 * Définit la racine sur une variable donnée.
 * @param v la variable décrite par cet arbre
 */
public void setRoot(Variable v)
{
    Variable oldRoot = v;
    root = v;
    fireTreeStructureChanged(oldRoot);
}

public Object getRoot()
{
    return root;
}

public int getChildCount(Object parent)
{
    return ((Variable)parent).getFields().size();
}

public Object getChild(Object parent, int index)
{
    ArrayList<Field> fields = ((Variable)parent).getFields();
    Field f = (Field)fields.get(index);
    Object parentValue = ((Variable)parent).getValue();
    try
    {
        return new Variable(f.getType(), f.getName(),
            f.get(parentValue));
    }
    catch(IllegalAccessException e)
    {
        return null;
    }
}

public int getIndexOfChild(Object parent, Object child)
{
    int n = getChildCount(parent);
    for (int i = 0; i < n; i++)
        if (getChild(parent, i).equals(child))
            return i;
    return -1;
}

public boolean isLeaf(Object node)
{
    return getChildCount(node) == 0;
}

public void valueForPathChanged(TreePath path,
    Object newValue)
{}

```

350 *Au cœur de Java 2 - Fonctions avancées*

```

public void addTreeModelListener(TreeModelListener l)
{
    listenerList.add(TreeModelListener.class, l);
}

public void removeTreeModelListener(TreeModelListener l)
{
    listenerList.remove(TreeModelListener.class, l);
}

protected void fireTreeStructureChanged(Object oldRoot)
{
    TreeModelEvent event
        = new TreeModelEvent(this, new Object[] {oldRoot});
    EventListener[] listeners = listenerList.getListeners(
        TreeModelListener.class);
    for (int i = 0; i < listeners.length; i++)
        ((TreeModelListener)listeners[i]).treeStructureChanged(
            event);
}

private Variable root;
private EventListenerList listenerList
    = new EventListenerList();
}

/**
 * Une variable avec un type, un nom et une valeur.
 */
class Variable
{
    /**
     * Construit une variable.
     * @param aType le type
     * @param aName le nom
     * @param aValue la valeur
     */
    public Variable(Class aType, String aName, Object aValue)
    {
        type = aType;
        name = aName;
        value = aValue;
        fields = new ArrayList<Field>();

        // trouve tous les champs si nous avons un type de classe
        // sauf que nous n'ouvrons pas les chaînes ni les valeurs nulles

        if (!type.isPrimitive() && !type.isArray() &&
            !type.equals(String.class) && value != null)
        {
            // obtient des champs de la classe et de toutes les superclasses
            for (Class c = value.getClass(); c != null;
                c = c.getSuperclass())
            {
                Field[] fs = c.getDeclaredFields();
                AccessibleObject.setAccessible(fs, true);

                // obtient tous les champs non-statiques
                for (Field f : fs)

```

```

        if ((f.getModifiers() & Modifier.STATIC) == 0)
            fields.add(f);
    }
}

/**
 * Obtient la valeur de cette variable.
 * @return la valeur
 */
public Object getValue() { return value; }

/**
 * Obtient tous les champs non-statiques de cette variable.
 * @return une liste de tableau des variables décrivant les champs
 */
public ArrayList<Field> getFields() { return fields; }

public String toString()
{
    String r = type + " " + name;
    if (type.isPrimitive())
        r += "=" + value;
    else if (type.equals(String.class))
        r += "=" + value;
    else if (value == null)
        r += "=null";
    return r;
}

private Class type;
private String name;
private Object value;
private ArrayList<Field> fields;
}

```

javax.swing.tree.TreeModel 1.2

- Object getRoot()
Renvoie le nœud de la racine.
 - int getChildCount(Object parent)
Renvoie le nœud d'enfants du nœud parent.
 - Object getChild(Object parent, int index)
Renvoie l'enfant du nœud parent à l'indice indiqué.
 - int getIndexOfChild(Object parent, Object child)
Renvoie l'indice du nœud child dans le nœud parent ou -1 si child n'est pas un enfant de parent dans ce modèle d'arbre.
 - boolean isLeaf(Object node)
Renvoie true si le nœud node est une feuille de l'arbre au plan conceptuel.
 - void addTreeModelListener(TreeModelListener l)
 - void removeTreeModelListener(TreeModelListener l)
- Ajoutent et suppriment les écouteurs qui sont avertis lorsque les informations du modèle de l'arbre sont modifiées.

352 *Au cœur de Java 2 - Fonctions avancées*

- `void valueForPathChanged(TreePath path, Object newValue)`

Cette méthode est appelée lorsqu'un éditeur de cellules a modifié la valeur d'un nœud.

Paramètres :

<code>path</code>	le chemin du nœud qui a été modifié
<code>newValue</code>	la valeur de remplacement renvoyée par l'éditeur

javax.swing.event.TreeModelListener 1.2

- `void treeNodesChanged(TreeModelEvent e)`
- `void treeNodesInserted(TreeModelEvent e)`
- `void treeNodesRemoved(TreeModelEvent e)`
- `void treeStructureChanged(TreeModelEvent e)`

Ces méthodes sont appelées par le modèle d'arbre lorsque l'arbre a été modifié.

javax.swing.event.TreeModelEvent 1.2

- `TreeModelEvent(Object eventSource, TreePath node)`

Construit un événement de modèle d'arbre.

Paramètres :

<code>eventSource</code>	le modèle d'arbre qui a généré cet événement
<code>node</code>	le chemin menant au nœud modifié

Tableaux

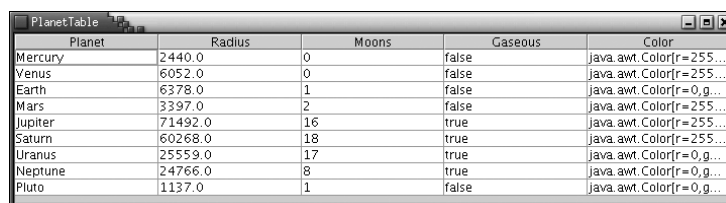
Le composant `JTable` affiche une grille bidimensionnelle d'objets. Naturellement, les tableaux sont très courants dans les interfaces utilisateur. L'équipe de Swing a déployé des efforts considérables pour contrôler et manipuler ces tableaux. De par leur nature, les tableaux sont compliqués, mais, peut-être plus que pour d'autres classes de Swing, le composant `JTable` prend en charge la plus grosse partie de cette complexité. Vous pourrez produire des tableaux parfaitement fonctionnels avec un comportement très riche, en écrivant uniquement quelques lignes de code. Mais vous pouvez bien sûr écrire un code plus complet et personnaliser l'affichage et le comportement de vos applications.

Dans cette section, nous vous expliquons comment réaliser des tableaux simples, comment l'utilisateur interagit avec eux et comment effectuer les réglages les plus courants. Comme pour d'autres outils de contrôle complexes de Swing, il est impossible d'en détailler tous les aspects. Si vous avez besoin d'informations plus poussées, vous pourrez les trouver dans les titres (en langue anglaise) *Core Java Foundation Classes*, de Kim Topley, ou *Graphic Java 2*, de David Geary.

Un tableau simple

Comme pour les composants d'un arbre, un `JTable` n'enregistre pas ses propres données, mais il les obtient à partir d'un *modèle de tableau*. De plus, vous pouvez construire un tableau à partir d'un tableau bidimensionnel d'objets et emballer automatiquement ce tableau dans un modèle par défaut. C'est la stratégie que nous avons retenue pour notre premier exemple. Un peu plus loin dans ce chapitre, nous aborderons les modèles de tableau.

La Figure 5.26 montre un tableau typique, qui décrit les propriétés des planètes du système solaire. Une planète est considérée comme *gazeuse* si elle est composée principalement d'hydrogène et d'hélium. Pour l'instant, ne vous occupez pas de la colonne "Couleur", qui a été ajoutée parce qu'elle nous sera utile dans un prochain exemple de code.

Figure 5.26*Un tableau simple.*


Planet	Radius	Moons	Gaseous	Color
Mercury	2440.0	0	false	java.awt.Color[r=255...
Venus	6052.0	0	false	java.awt.Color[r=255...
Earth	6378.0	1	false	java.awt.Color[r=0,g...
Mars	3397.0	2	false	java.awt.Color[r=255...
Jupiter	71492.0	16	true	java.awt.Color[r=255...
Saturn	60268.0	18	true	java.awt.Color[r=255...
Uranus	25559.0	17	true	java.awt.Color[r=0,g...
Neptune	24766.0	8	true	java.awt.Color[r=0,g...
Pluto	1137.0	1	false	java.awt.Color[r=0,g...

Comme vous pouvez le voir dans le code de l'Exemple 5.9, les données de ce tableau sont enregistrées dans un tableau bidimensionnel de valeurs Object :

```
Object[][] cells =
{
    { "Mercure", 2440.0, 0, false, Color.yellow },
    { "Vénus", 6052.0, 0, false, Color.yellow },
    . . .
}
```

INFO

Nous profitons ici de l'autoboxing. Les entrées des deuxième, troisième et quatrième colonnes sont automatiquement converties en objets du type Double, Integer et Boolean.

Ce tableau se contente d'invoquer la méthode toString de chaque objet pour l'afficher. C'est pourquoi les couleurs sont affichées comme java.awt.Color[r=...,g=...,b=...].

Les noms des colonnes sont fournies dans des chaînes séparées :

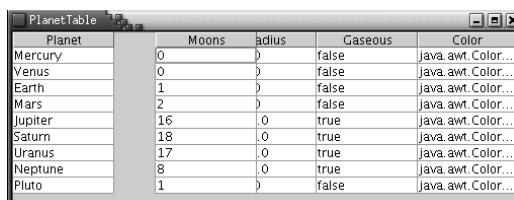
```
String[] columnNames =
{ "Planète", "Rayon", "Lunes", "Gazeuse", "Couleur" };
```

Vous pouvez ensuite construire un tableau à partir des tableaux contenant les noms des colonnes et des cellules. Pour terminer, ajoutez des barres de défilement en emballant le tableau dans un JScrollPane.

```
JTable table = new JTable(cells, columnNames);
JScrollPane pane = new JScrollPane(table);
```

Le tableau résultant possède déjà un comportement étonnamment riche. Recadrez le tableau verticalement jusqu'à ce que l'ascenseur vertical apparaisse et déplacez cet ascenseur. Les noms des colonnes restent constamment visibles !

Cliquez ensuite sur l'un des noms de colonnes et déplacez-le à droite ou à gauche. La colonne entière se détache (voir Figure 5.27). Vous pouvez donc l'amener à un nouvel emplacement. Cela modifie uniquement l'affichage des colonnes. Le modèle des données n'est pas affecté.

Figure 5.27*Déplacer une colonne.*


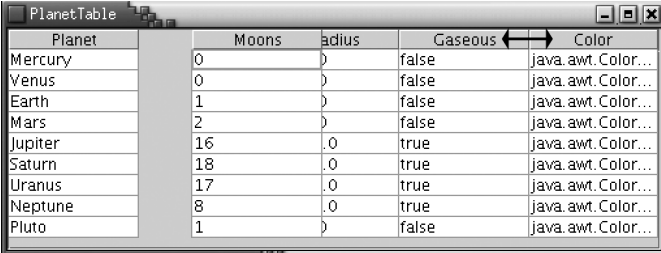
Planet	Moons	Radius	Gaseous	Color
Mercury	0		false	java.awt.Color...
Venus	0		false	java.awt.Color...
Earth	1		false	java.awt.Color...
Mars	2		false	java.awt.Color...
Jupiter	16	.0	true	java.awt.Color...
Saturn	18	.0	true	java.awt.Color...
Uranus	17	.0	true	java.awt.Color...
Neptune	8	.0	true	java.awt.Color...
Pluto	1		false	java.awt.Color...

354 Au cœur de Java 2 - Fonctions avancées

Pour modifier la largeur des colonnes, il suffit de placer le curseur entre deux colonnes jusqu'à ce qu'il se transforme en flèche. Vous pouvez maintenant déplacer le bord de la colonne (voir Figure 5.28).

Figure 5.28

Modifier la largeur des colonnes.



Planet	Moons	Radius	Gaseous	Color
Mercury	0	0	false	java.awt.Color...
Venus	0	0	false	java.awt.Color...
Earth	1	0	false	java.awt.Color...
Mars	2	0	false	java.awt.Color...
Jupiter	16	.0	true	java.awt.Color...
Saturn	18	.0	true	java.awt.Color...
Uranus	17	.0	true	java.awt.Color...
Neptune	8	.0	true	java.awt.Color...
Pluto	1	0	false	java.awt.Color...

Les utilisateurs peuvent à tout moment sélectionner des lignes en cliquant à n'importe quel endroit d'une ligne. Les lignes sélectionnées sont noircies. Vous verrez un peu plus loin comment récupérer les événements de sélection. Les utilisateurs peuvent aussi modifier les données d'un tableau en cliquant sur une cellule et en saisissant de nouvelles données. Cependant, dans cet exemple, les modifications ne sont pas reportées dans le modèle de données. Dans vos programmes, vous devrez soit rendre vos cellules non modifiables, soit gérer les événements de modification des cellules et mettre à jour votre modèle. Nous revenons sur ces points un peu plus loin dans ce chapitre.

Depuis le JDK 5.0, vous pouvez afficher un tableau grâce à la méthode `print` :

```
table.print();
```

Une boîte de dialogue d'impression s'affiche et le tableau est envoyé à l'imprimante.

Exemple 5.9 : *PlanetTable.java*

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;

/**
 * Ce programme montre comment afficher un tableau simple.
 */
public class PlanetTable
{
    public static void main(String[] args)
    {
        JFrame frame = new PlanetTableFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * ce bloc contient un tableau des données sur les planètes.
 */
```

```

class PlanetTableFrame extends JFrame
{
    public PlanetTableFrame()
    {
        setTitle("PlanetTable");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
        final JTable table = new JTable(cells, columnNames);
        add(new JScrollPane(table), BorderLayout.CENTER);
        JButton printButton = new JButton("Imprimer");
        printButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    try
                    {
                        table.print();
                    }
                    catch (java.awt.print.PrinterException e)
                    {
                        e.printStackTrace();
                    }
                }
            });
        JPanel buttonPanel = new JPanel();
        buttonPanel.add(printButton);
        add(buttonPanel, BorderLayout.SOUTH);
    }

    private Object [][] cells =
    {
        {"Mercure", 2440.0, 0, false, Color.yellow },
        {"Venus", 6052.0, 0, false, Color.yellow },
        {"Terre", 6378.0, 1, false, Color.blue },
        {"Mars", 3397.0, 2, false, Color.red },
        {"Jupiter", 71492.0, 16, true, Color.orange },
        {"Saturne", 60268.0, 18, true, Color.orange },
        {"Uranus", 25559.0, 17, true, Color.blue },
        {"Neptune", 24766.0, 8, true, Color.blue },
        {"Pluton", 1137.0, 1, false, Color.black }
    };

    private String[] columnNames =
        { "Planète", "Rayon", "Lunes", "Gazeuse", "Couleur" };

    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 200;
}

```

javax.swing.JTable 1.2

- `JTable(Object[][] entries, Object[] columnNames)`
Construit un tableau à partir du modèle de tableau par défaut.
- `void print() 5.0`
Affiche une boîte de dialogue d'impression et imprime le tableau.

356 Au cœur de Java 2 - Fonctions avancées

Modèles de tableaux

Dans l'exemple précédent, les objets affichés dans le tableau étaient enregistrés dans un tableau bidimensionnel. Cependant, cette stratégie n'est généralement pas conseillée. Il vaut mieux implémenter votre propre modèle de tableau au lieu de placer toutes vos données dans un tableau bidimensionnel pour les afficher sous la forme d'un tableau.

Les modèles de tableaux sont particulièrement simples à implémenter parce que vous pouvez vous servir de la classe `AbstractTableModel` qui implémente la plupart des méthodes nécessaires. Il vous suffira de fournir trois méthodes :

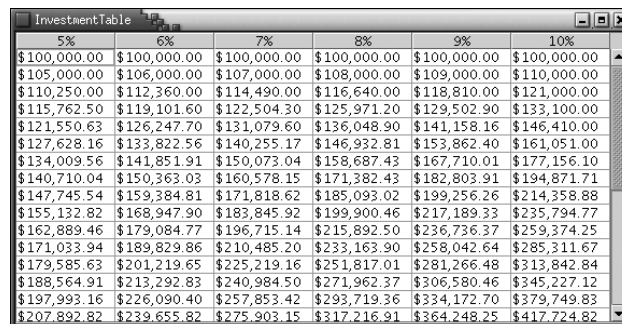
```
public int getRowCount();
public int getColumnCount();
public Object getValueAt(int row, int column);
```

Il existe plusieurs manières d'implémenter la méthode `getValueAt`. Vous pouvez simplement calculer la réponse, ou chercher la valeur dans une base de données, ou encore dans une autre source de données. Prenons maintenant quelques exemples.

Dans le premier exemple, nous construisons un tableau qui affiche simplement quelques valeurs calculées, c'est-à-dire l'évolution d'un placement selon différents taux d'intérêt (voir Figure 5.29).

Figure 5.29

*Evolution
d'un placement.*



	5%	6%	7%	8%	9%	10%
\$100,000.00	\$100,000.00	\$100,000.00	\$100,000.00	\$100,000.00	\$100,000.00	\$100,000.00
\$105,000.00	\$105,000.00	\$107,000.00	\$108,000.00	\$109,000.00	\$110,000.00	\$111,000.00
\$110,250.00	\$112,360.00	\$114,490.00	\$116,640.00	\$118,810.00	\$121,000.00	\$123,210.00
\$115,762.50	\$119,101.60	\$122,504.30	\$125,971.20	\$129,502.90	\$133,100.00	\$136,762.50
\$121,550.63	\$126,247.70	\$131,079.60	\$136,048.90	\$141,158.16	\$146,410.00	\$151,812.50
\$127,628.16	\$133,822.56	\$140,255.17	\$146,932.81	\$153,862.40	\$161,051.00	\$168,393.75
\$134,009.56	\$141,851.91	\$150,073.04	\$158,687.43	\$167,710.01	\$177,156.10	\$186,925.00
\$140,710.04	\$150,363.03	\$160,578.15	\$171,382.43	\$182,803.91	\$194,871.71	\$207,606.25
\$147,745.54	\$159,384.81	\$171,818.62	\$185,093.02	\$199,256.26	\$214,358.88	\$229,175.00
\$155,132.82	\$168,947.90	\$183,845.92	\$199,900.46	\$217,189.33	\$235,794.77	\$253,675.00
\$162,889.46	\$179,084.77	\$196,715.14	\$215,892.50	\$236,736.37	\$259,374.25	\$283,511.67
\$171,033.94	\$189,829.86	\$210,485.20	\$233,163.90	\$258,042.64	\$285,311.67	\$311,667.50
\$179,585.63	\$201,219.65	\$225,219.16	\$251,817.01	\$281,266.48	\$313,842.84	\$345,227.12
\$188,564.91	\$213,292.83	\$240,984.50	\$271,962.37	\$306,580.46	\$345,227.12	\$385,937.50
\$197,993.16	\$226,090.40	\$257,853.42	\$293,719.36	\$334,172.70	\$379,749.83	\$427,937.50
\$207,892.82	\$239,655.82	\$275,903.15	\$317,216.91	\$364,248.25	\$417,724.82	\$474,375.00

La méthode `getValueAt` calcule la valeur appropriée et la formate :

```
public Object getValueAt(int r, int c)
{
    double rate = (c + minRate) / 100.0;
    int nperiods = r;

    double futureBalance = INITIAL_BALANCE
        * Math.pow(1 + rate, nperiods);

    return String.format("%.2f", futureBalance);
}
```

Les méthodes `getRowCount` et `getColumnCount` renvoient simplement le nombre de lignes et de colonnes.

```
public int getRowCount() { return years; }
public int getColumnCount()
{ return maxRate - minRate + 1; }
```

Si vous ne définissez pas de noms de colonnes, la méthode `getColumnName` du modèle `AbstractTableModel` choisit comme noms A, B, C, etc. Pour modifier le nom des colonnes, il suffit de surcharger la méthode `getColumnName`. De manière générale, vous choisirez de surcharger ce comportement par défaut. Dans cet exemple, nous donnons à chaque colonne un nom correspondant au taux d'intérêt.

```
public String getColumnName(int c)
{ return (c + minRate) + "%"; }
```

Vous trouverez le code source complet dans l'Exemple 5.10.

Exemple 5.10 : *InvestmentTable.java*

```
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import javax.swing.*;
import javax.swing.table.*;

/**
 * Ce programme vous montre comment construire un tableau à partir d'un
 * modèle de tableau.
 */
public class InvestmentTable
{
    public static void main(String[] args)
    {
        JFrame frame = new InvestmentTableFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc contient le tableau d'investissement.
 */
class InvestmentTableFrame extends JFrame
{
    public InvestmentTableFrame()
    {
        setTitle("InvestmentTable");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        TableModel model = new InvestmentTableModel(30, 5, 10);
        JTable table = new JTable(model);
        add(new JScrollPane(table));
    }

    private static final int DEFAULT_WIDTH = 600;
    private static final int DEFAULT_HEIGHT = 300;
}

/**
 * Ce modèle de tableau calcule les entrées des cellules
 * chaque fois qu'elles sont demandées. Le contenu du tableau présente
```

358 *Au cœur de Java 2 - Fonctions avancées*

```
    la croissance d'un investissement pendant un certain nombre d'années
    sous différents taux d'intérêt.
*/
class InvestmentTableModel extends AbstractTableModel
{
    /**
     Construit un modèle de tableau d'investissement.
     @param y le nombre d'années
     @param r1 le plus bas taux d'intérêt du tableau
     @param r2 le plus haut taux d'intérêt du tableau
    */
    public InvestmentTableModel(int y, int r1, int r2)
    {
        years = y;
        minRate = r1;
        maxRate = r2;
    }

    public int getRowCount() { return years; }

    public int getColumnCount() { return maxRate - minRate + 1; }

    public Object getValueAt(int r, int c)
    {
        double rate = (c + minRate) / 100.0;
        int nperiods = r;

        double futureBalance = INITIAL_BALANCE
            * Math.pow(1 + rate, nperiods);

        return String.format("%.2f", futureBalance);
    }

    public String getColumnName(int c) { return (c + minRate) + "%"; }

    private int years;
    private int minRate;
    private int maxRate;

    private static double INITIAL_BALANCE = 100000.0;
}
```

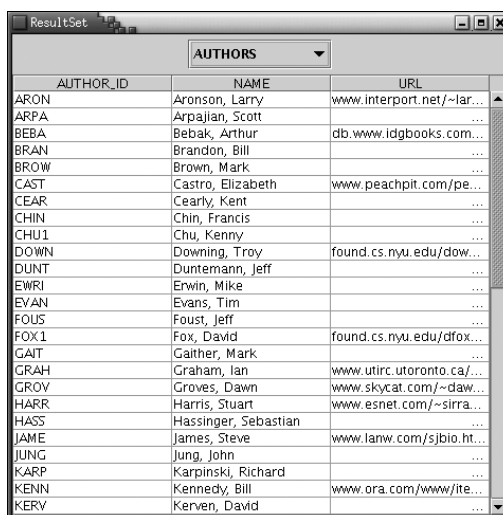
Afficher des enregistrements de base de données

Les informations que l'on retrouve le plus souvent dans un tableau sont des enregistrements issus d'une base de données. Si vous vous servez d'un environnement de développement professionnel, il possède certainement des composants Java (beans) pratiques permettant d'accéder aux informations d'une base de données. Cependant, si vous ne disposez pas de ces beans ou si vous souhaitez comprendre les détails de cette implémentation, vous serez intéressé par le prochain exemple. La Figure 5.30 en montre le résultat, c'est-à-dire le résultat d'une requête sur toutes les lignes d'un tableau d'une base de données.

Dans le programme d'exemple, nous définissons un modèle, `ResultSetTableModel`, qui va chercher des données dans un ensemble de résultats d'une requête de base de données. Consultez le Chapitre 4 pour plus d'informations sur les accès aux bases de données Java et sur les ensembles de résultats.

Figure 5.30

Afficher les résultats d'une requête dans un tableau.



AUTHOR_ID	NAME	URL
ARON	Aronson, Larry	www.interport.net/~lar...
ARPA	Arpajian, Scott	...
BEBE	Bebak, Arthur	db.www.idgbooks.com...
BRAN	Brandon, Bill	...
BROW	Brown, Mark	...
CAST	Castro, Elizabeth	www.peachpit.com/pe...
CEAR	Cearly, Kent	...
CHIN	Chin, Francis	...
CHU1	Chu, Kenny	...
DOWN	Downing, Troy	found.cs.nyu.edu/dow...
DUNT	Duntemann, Jeff	...
EWRI	Erwin, Mike	...
EVAN	Evans, Tim	...
FOUS	Foust, Jeff	...
FOX1	Fox, David	found.cs.nyu.edu/dfox...
GAIT	Gaither, Mark	...
GRAH	Graham, Ian	www.utirc.utoronto.ca/...
GROV	Groves, Dawn	www.skyat.com/~daw...
HARR	Harris, Stuart	www.esnet.com/~sirra...
HASS	Hassinger, Sebastian	...
JAME	James, Steve	www.lanw.com/sjbio.ht...
JUNG	Jung, John	...
KARP	Karpinski, Richard	...
KENN	Kennedy, Bill	www.ora.com/www/ite...
KERV	Kerven, David	...

Vous pouvez obtenir le nombre de colonnes et leurs noms à partir d'un objet `ResultSetMetaData`, c'est-à-dire une métadonnée contenant un ensemble de résultats.

```
public String getColumnName(int c)
{
    try
    {
        return rsmd.getColumnName(c + 1);
    }
    catch(SQLException e)
    {
        . . .
    }
}

public int getColumnCount()
{
    try
    {
        return rsmd.getColumnCount();
    }
    catch(SQLException e)
    {
        . . .
    }
}
```

Si la base de données supporte les curseurs déroulants, il devient particulièrement facile de récupérer la valeur d'une cellule : amenez le curseur sur la ligne choisie et lisez la valeur se trouvant sur la colonne spécifiée.

```
public Object getValueAt(int r, int c)
{
    try
    {
        ResultSet rs = getResultSet();
```

360 *Au cœur de Java 2 - Fonctions avancées*

```

        rs.absolute(r + 1);
        return rs.getObject(c + 1);
    }
    catch (SQLException e)
    {
        e.printStackTrace();
        return null;
    }
}

```

Il est très intéressant d'utiliser ce modèle de données à la place de `DefaultTableModel`. Si vous aviez créé un tableau de valeurs, vous seriez en fait amené à dupliquer le tampon que le pilote de la base de données gère déjà.

Si la base de données ne supporte pas les curseurs déroulants, notre exemple place les données dans un `RowSet`.

Exemple 5.11 : *ResultSetTable.java*

```

import com.sun.rowset.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.sql.*;
import java.util.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.sql.rowset.*;

/**
 * Ce programme montre comment afficher le résultat d'une requête de base
 * de données dans un tableau.
 */
public class ResultSetTable
{
    public static void main(String[] args)
    {
        JFrame frame = new ResultSetFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc contient un menu déroulant pour sélectionner une table de
 * base de données et une table pour afficher les données qui y sont
 * stockées.
 */
class ResultSetFrame extends JFrame
{
    public ResultSetFrame()
    {
        setTitle("ResultSet");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        /* Retrouve toutes les tables dans la base de données et les ajoute
         * à un menu déroulant.
         */
    }
}

```

```

tableNames = new JComboBox();
tableNames.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            try
            {
                if (scrollPane != null) remove(scrollPane);
                String tableName
                    = (String)tableNames.getSelectedItem();
                if (rs != null) rs.close();
                String query = "SELECT * FROM " + tableName;
                rs = stat.executeQuery(query);
                if (scrolling)
                    model = new ResultSetTableModel(rs);
                else
                {
                    CachedRowSet crs =new CachedRowSetImpl();
                    crs.populate(rs);
                    model = new ResultSetTableModel(crs);

                    JTable table = new JTable(model);
                    scrollPane = new JScrollPane(table);
                    add(scrollPane, BorderLayout.CENTER);
                    validate();
                }
            }
            catch (SQLException e)
            {
                e.printStackTrace();
            }
        }
    });
JPanel p = new JPanel();
p.add(tableNames);
add(p, BorderLayout.NORTH);

try
{
    conn = getConnection();
    DatabaseMetaData meta = conn.getMetaData();
    if (meta.supportsResultSetType(
        ResultSet.TYPE_SCROLL_INSENSITIVE))
    {
        scrolling = true;
        stat = conn.createStatement(
            ResultSet.TYPE_SCROLL_INSENSITIVE,
            ResultSet.CONCUR_READ_ONLY);
    }
    else
    {
        stat = conn.createStatement();
        scrolling = false;
    }
    ResultSet tables = meta.getTables(null, null, null,
        new String[] { "TABLE" });
    while (tables.next())
        tableNames.addItem(tables.getString(3));
    tables.close();
}

```


362 *Au cœur de Java 2 - Fonctions avancées*

```

    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
    catch (SQLException e)
    {
        e.printStackTrace();
    }
}

addWindowListener(new
    WindowAdapter()
    {
        public void windowClosing(WindowEvent event)
        {
            try
            {
                if (conn != null) conn.close();
            }
            catch (SQLException e)
            {
                e.printStackTrace();
            }
        }
    });
}

/**
 * Récupère une connexion des propriétés spécifiées dans
 * Le fichier database.properties.
 * @return la connexion à la base de données
 */
public static Connection getConnection()
    throws SQLException, IOException
{
    Properties props = new Properties();
    FileInputStream in
        = new FileInputStream("database.properties");
    props.load(in);
    in.close();

    String drivers = props.getProperty("jdbc.drivers");
    if (drivers != null)
        System.setProperty("jdbc.drivers", drivers);
    String url = props.getProperty("jdbc.url");
    String username = props.getProperty("jdbc.username");
    String password = props.getProperty("jdbc.password");

    return
        DriverManager.getConnection(url, username, password);
}

private JScrollPane scrollPane;
private ResultSetTableModel model;
private JComboBox tableNames;
private ResultSet rs;
private Connection conn;
private Statement stat;
private boolean scrolling;

```

```

        private static final int DEFAULT_WIDTH = 400;
        private static final int DEFAULT_HEIGHT = 300;
    }

    /**
     * Cette classe est la superclasse de défilement et le modèle de table
     * de l'ensemble de résultats en cache. Elle stocke l'ensemble de
     * résultats et ses méta-données.
     */
    class ResultSetTableModel extends AbstractTableModel
    {
        /**
         * Construit le modèle de table.
         * @param aResultSet l'ensemble de résultats à afficher
         */
        public ResultSetTableModel(ResultSet aResultSet)
        {
            rs = aResultSet;
            try
            {
                rsmd = rs.getMetaData();
            }
            catch (SQLException e)
            {
                e.printStackTrace();
            }
        }

        public String getColumnName(int c)
        {
            try
            {
                return rsmd.getColumnName(c + 1);
            }
            catch (SQLException e)
            {
                e.printStackTrace();
                return "";
            }
        }

        public int getColumnCount()
        {
            try
            {
                return rsmd.getColumnCount();
            }
            catch (SQLException e)
            {
                e.printStackTrace();
                return 0;
            }
        }

        public Object getValueAt(int r, int c)
        {
            try
            {
                rs.absolute(r + 1);
            }
        }
    }

```

364 Au cœur de Java 2 - Fonctions avancées

```

        return rs.getObject(c + 1);
    }
    catch(SQLException e)
    {
        e.printStackTrace();
        return null;
    }
}

public int getRowCount()
{
    try
    {
        rs.last();
        return rs.getRow();
    }
    catch(SQLException e)
    {
        e.printStackTrace();
        return 0;
    }
}

private ResultSet rs;
private ResultSetMetaData rsmd;
}

```

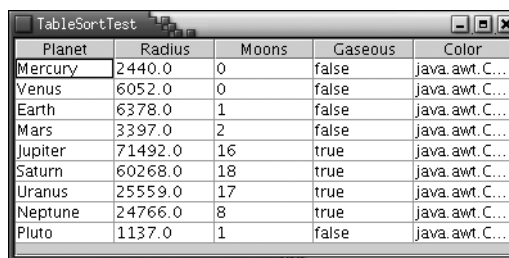
Un filtre de tri

Les deux derniers exemples ont mis en évidence le fait que les tableaux ne stockent pas les données de leurs cellules. Ils les récupèrent à partir d'un modèle. Ce dernier n'a pas non plus besoin de conserver ces données, puisqu'il peut les calculer ou les lire depuis une autre source de données.

Dans cette section, nous introduisons une autre technique utile, un *modèle de filtre*, qui présente les informations d'un autre tableau sous une forme différente. Dans notre exemple, nous allons *trier* les lignes d'un tableau. Exécutez le programme de l'Exemple 5.12 et *double-cliquez* sur le nom d'une colonne. Vous verrez alors comment les lignes sont réorganisées, de sorte que les entrées de cette colonne soient triées (voir Figure 5.31).

Figure 5.31

Trier les lignes d'un tableau.



Planet	Radius	Moons	Gaseous	Color
Mercury	2440.0	0	false	java.awt.C...
Venus	6052.0	0	false	java.awt.C...
Earth	6378.0	1	false	java.awt.C...
Mars	3397.0	2	false	java.awt.C...
Jupiter	71492.0	16	true	java.awt.C...
Saturn	60268.0	18	true	java.awt.C...
Uranus	25559.0	17	true	java.awt.C...
Neptune	24766.0	8	true	java.awt.C...
Pluto	1137.0	1	false	java.awt.C...

Cependant, les lignes ne sont pas réarrangées physiquement dans le modèle de données. Nous nous contentons d'utiliser un *modèle de filtre*, qui permet de conserver un tableau en permutant les indices des lignes.

Le modèle de filtre enregistre une référence au modèle de tableau réel. Lorsque `JTable` a besoin de rechercher une valeur, le modèle de filtre calcule l'indice de ligne réel et récupère la valeur correspondante à partir du modèle. Par exemple,

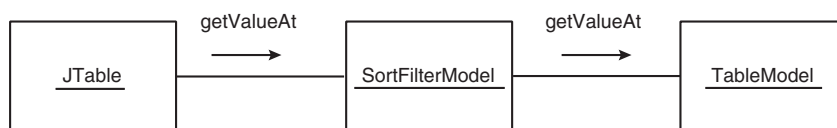
```
public Object getValueAt(int r, int c)
{ return model.getValueAt(indice réel de la ligne, c); }
```

Toutes les autres méthodes sont simplement passées au modèle d'origine.

```
public String getColumnName(int c)
{ return model.getColumnName(c); }
```

La Figure 5.32 montre la position du filtre, entre l'objet `JTable` et le modèle de tableau réel.

Figure 5.32
Un filtre de modèle de tableau.



Il y a deux subtilités pour implémenter ce type de filtre. Tout d'abord, vous devez être averti lorsque l'utilisateur double-clique sur l'un des en-têtes des colonnes. Nous n'entrerons pas trop dans les détails de cette opération. Vous en trouverez le code dans la méthode `addMouseListener` du modèle `SortFilterModel`, dans l'Exemple 5.12. Voici le principe à retenir. En premier lieu, récupérez le composant de l'en-tête du tableau et attachez-y un écouteur de souris. Lorsqu'un double-clic est détecté, vous devrez trouver dans quel en-tête ce double-clic s'est produit. Puis, vous devrez transposer la colonne du tableau en colonne du modèle ; elles peuvent être différentes si l'utilisateur a déplacé les colonnes du tableau. Une fois que vous connaissez la colonne du modèle, vous pouvez commencer à trier les lignes du tableau.

Le tri des lignes d'un tableau peut parfois poser problème. En effet, nous ne voulons pas modifier l'ordre physique des lignes. Nous voulons simplement une séquence d'indices de lignes qui indiqueraient comment réorganiser les lignes entre elles si elles devaient être triées. Cependant, les algorithmes de tri des classes `Arrays` et `Collections` ne nous indiquent pas comment réorganiser les éléments triés. Naturellement, nous pourrions implémenter un nouvel algorithme de tri pour conserver une trace de l'ordre des éléments triés. Mais il existe une méthode bien plus fine, qui consiste à choisir des objets personnalisés et une méthode de comparaison personnalisée, pour que l'algorithme de tri de la bibliothèque puisse être transformé en service.

Nous allons donc trier des objets de type `Row`. Un objet `Row` contient l'indice `r` d'une ligne du modèle. Deux objets de ce type peuvent être comparés comme ceci : trouvez les éléments dans le modèle et comparez-les entre eux. En d'autres termes, la méthode `compareTo` des objets `Row` calcule :

```
model.getValueAt(r1, c).compareTo(model.getValueAt(r2, c))
```

Ici, `r1` et `r2` correspondent à des indices de lignes d'objets `Row`, et `c` est la colonne dont les éléments doivent être triés.

Si les entrées d'une colonne particulière ne sont pas comparables, nous comparons simplement leurs représentations de chaînes. De cette manière, la colonne possédant des valeurs `Color` peuvent être triées. La classe `Color` n'implémente pas l'interface `Comparable`.

366 *Au cœur de Java 2 - Fonctions avancées*

Nous transformons la classe `Row` en classe interne du modèle `SortFilterModel`, parce que la méthode `compareTo` doit pouvoir accéder au modèle courant et à la colonne concernée. En voici le code :

```
class SortFilterModel extends AbstractTableModel
{
    . . .
    private class Row implements Comparable<Row>
    {
        public int index;
        public int compareTo(Row other)
        {
            Object a = model.getValueAt(index, sortColumn);
            Object b = model.getValueAt(other.index, sortColumn);
            if (a instanceof Comparable)
                return ((Comparable)a).compareTo(b);
            else
                return a.toString().compareTo(b.toString());
        }
    }

    private TableModel model;
    private int sortColumn;
    private Row[] rows;
}
```

Dans le constructeur, nous construisons un tableau appelé `rows`, initialisé de telle sorte que `rows[i]` soit égal à `i` :

```
public SortFilterModel(TableModel m)
{
    model = m;
    rows = new Row[model.getRowCount()];
    for (int i = 0; i < rows.length; i++)
    {
        rows[i] = new Row();
        rows[i].index = i;
    }
}
```

Dans la méthode `sort`, nous invoquons l'algorithme `Arrays.sort`, qui trie les objets `Row`. Comme le critère de comparaison examine les éléments du modèle dans la colonne appropriée, les éléments sont arrangés de telle manière que `row[0]` contienne l'indice du plus petit élément de la colonne, `row[1]` l'indice du deuxième plus petit élément, etc.

Lorsque le tableau est trié, nous indiquons à tous les écouteurs du modèle du tableau (en particulier à `JTable`) que le contenu du tableau a changé et qu'il doit être affiché à nouveau.

```
public void sort(int c)
{
    sortColumn = c;
    Arrays.sort(rows);
    fireTableDataChanged();
}
```

Pour terminer, nous pouvons vous montrer le calcul exact de la méthode `getValueAt` de la classe de filtre. Elle se contente de transformer un indice de ligne `r` en indice de ligne du modèle `rows[r].index`.

```
public Object getValueAt(int r, int c)
{ return model.getValueAt(rows[r].index, c); }
```

Cet exemple montre à nouveau la puissance du schéma de modèle-visualisation-contrôleur. Comme les données et l’affichage sont séparés, nous sommes en mesure de modifier les liens entre les deux.

ASTUCE

Vous trouverez une version plus élaborée du tri de tableau à l’adresse <http://java.sun.com/docs/books/tutorial/uiswing/components/table.html#sorting>. Cette implémentation écoute le modèle de tableau et actualise la vue triée en cas de changement du modèle de tableau.

Exemple 5.12 : *TableSortTest.java*

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;

/**
 * Ce programme montre comment trier une colonne de tableau.
 * Double-cliquez sur une colonne de tableau pour la trier.
 */
public class TableSortTest
{
    public static void main(String[] args)
    {
        JFrame frame = new TableSortFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc contient une table des données de planètes.
 */
class TableSortFrame extends JFrame
{
    public TableSortFrame()
    {
        setTitle("TableSortTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // configure un modèle de table et interpose un élément de tri

        DefaultTableModel model
            = new DefaultTableModel(cells, columnNames);
        final SortFilterModel sorter = new SortFilterModel(model);
```

368 *Au cœur de Java 2 - Fonctions avancées*

```

// affiche la table

final JTable table = new JTable(sorter);
add(new JScrollPane(table), BorderLayout.CENTER);

// configure un gestionnaire de double-clic pour les en-têtes de
// colonnes

table.getTableHeader().addMouseListener(new
    MouseAdapter()
    {
        public void mouseClicked(MouseEvent event)
        {
            // vérifie les double-clics
            if (event.getClickCount() < 2) return;

            // trouve la colonne du clic et
            int tableColumn
                = table.columnAtPoint(event.getPoint());

            // la transforme en indice dans le modèle du tableau et
            // effectue le tri
            int modelColumn
                = table.convertColumnIndexToModel(tableColumn);
            sorter.sort(modelColumn);
        }
    });
}

private Object[][] cells =
{
    { "Mercure", 2440.0, 0, false, Color.yellow },
    { "Vénus", 6052.0, 0, false, Color.yellow },
    { "Terre", 6378.0, 1, false, Color.blue },
    { "Mars", 3397.0, 2, false, Color.red },
    { "Jupiter", 71492.0, 16, true, Color.orange },
    { "Saturne", 60268.0, 18, true, Color.orange },
    { "Uranus", 25559.0, 17, true, Color.blue },
    { "Neptune", 24766.0, 8, true, Color.blue },
    { "Pluton", 1137.0, 1, false, Color.black }
};

private String[] columnNames =
{ "Planète", "Rayon", "Lunes", "Gazeuse", "Couleur" };

private static final int DEFAULT_WIDTH = 400;
private static final int DEFAULT_HEIGHT = 200;
}

/**
 * Ce modèle de table prend un modèle de table existant et produit
 * un nouveau modèle qui trie les lignes afin que les entrées d'une
 * colonne particulière sont triées.
 */
class SortFilterModel extends AbstractTableModel
{
    /**
     * Construit un modèle de filtre de tri.
     * @param m le modèle de table à filtrer
     */
}

```

```

    */
    public SortFilterModel(TableModel m)
    {
        model = m;
        rows = new Row[model.getRowCount()];
        for (int i = 0; i < rows.length; i++)
        {
            rows[i] = new Row();
            rows[i].index = i;
        }
    }

    /**
     * Trie les lignes.
     * @param c la colonne à trier
     */
    public void sort(int c)
    {
        sortColumn = c;
        Arrays.sort(rows);
        fireTableDataChanged();
    }

    // Calcule la ligne déplacée pour les trois méthodes qui accèdent
    // aux éléments de modèles

    public Object getValueAt(int r, int c)
    { return model.getValueAt(rows[r].index, c); }

    public boolean isCellEditable(int r, int c)
    { return model.isCellEditable(rows[r].index, c); }

    public void setValueAt(Object aValue, int r, int c)
    {
        model.setValueAt(aValue, rows[r].index, c);
    }

    // délègue toutes les méthodes restantes au modèle

    public int getRowCount() { return model.getRowCount(); }
    public int getColumnCount() { return model.getColumnCount(); }
    public String getColumnName(int c)
    { return model.getColumnName(c); }
    public Class getColumnClass(int c)
    { return model.getColumnClass(c); }

    /**
     * Cette classe intérieure contient l'indice de la ligne de modèle.
     * Les lignes sont comparées en regardant les entrées de la ligne du
     * modèle dans la colonne de tri.
     */
    private class Row implements Comparable<Row>
    {
        public int index;
        public int compareTo(Row other)
        {
            Object a = model.getValueAt(index, sortColumn);
            Object b = model.getValueAt(other.index, sortColumn);
            if (a instanceof Comparable)

```


370 *Au cœur de Java 2 - Fonctions avancées*

```

        return ((Comparable) a).compareTo(b);
    else
        return a.toString().compareTo(b.toString());
    }

    private TableModel model;
    private int sortColumn;
    private Row[] rows;
}

```

javax.swing.table.TableModel 1.2

- `int getRowCount()`
Renvoient le nombre de lignes et de colonnes dans le modèle du tableau.
- `int getColumnCount()`
Renvoie la valeur située à la ligne et à la colonne spécifiées.
- `Object getValueAt(int row, int column)`
Renvoie la valeur située à la ligne et à la colonne spécifiées.
- `void setValueAt(Object newValue, int row, int column)`
Ecrit une nouvelle valeur à la ligne et à la colonne spécifiées.
- `boolean isCellEditable(int row, int column)`
Renvoie true si la cellule située à la ligne et à la colonne spécifiées peut être modifiée.
- `String getColumnName(int column)`
Renvoie le titre de la colonne.

javax.swing.table.AbstractTableModel 1.2

- `void fireTableDataChanged()`
Avertit tous les écouteurs d'un modèle de tableau que les données du tableau ont été modifiées.

javax.swing.JTable 1.2

- `JTableHeader getTableHeader()`
Renvoie le composant de l'en-tête de tableau.
- `int columnAtPoint(Point p)`
Renvoie l'indice de la colonne du tableau correspondant à la position p d'un pixel.
- `int convertColumnIndexToModel(int tableColumn)`
Renvoie l'indice dans le modèle de la colonne dont l'indice est spécifié. Cette valeur peut être différente de la colonne du tableau tableColumn si certaines colonnes du tableau ont été déplacées ou cachées.

Affichage et modification des cellules

Dans le prochain exemple, nous afficherons à nouveau nos données relatives aux planètes, mais cette fois, le tableau doit posséder plus d'informations sur les types des colonnes. Si vous définissez la méthode

```
Class getColumnClass(int columnIndex)
```

de votre modèle de tableau pour qu'elle renvoie la classe qui décrit le type de la colonne, la classe `JTable` choisira un *afficheur* approprié pour cette classe. Le Tableau 5.1 montre la manière dont la classe `JTable` affiche les types par défaut.

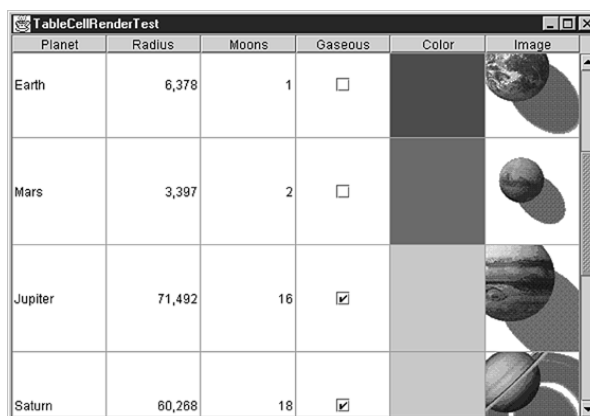
Tableau 5.1 : Les afficheurs par défaut

Type	Affiché comme
Icon	image
Boolean	case à cocher
Object	chaîne

Vous trouverez les cases à cocher et les images dans la Figure 5.33 (merci à Jim Evins, <http://www.snaught.com/JimsCoolIcons/Planets/>, pour les images des planètes).

Figure 5.33

Un tableau avec des afficheurs de cellules.



Pour les autres types, vous pouvez fournir vos propres afficheurs de cellules. Les afficheurs de cellules de tableau sont comparables aux afficheurs de cellules d'arbre, que nous avons déjà abordés. Ils implémentent l'interface `TableCellRenderer`, qui comprend une seule méthode :

```
Component getTableCellRendererComponent(JTable table,
    Object value, boolean isSelected, boolean hasFocus,
    int row, int column)
```

Cette méthode est appelée lorsque l'arbre doit afficher une cellule. Vous renvoyez un composant dont la méthode `paint` est invoquée pour dessiner la cellule.

Pour afficher une cellule de type `Color`, il suffit de renvoyer un panneau dont la couleur de fond correspond à la couleur de l'objet enregistré dans la cellule. La couleur est passée dans le paramètre `value`.

```
class ColorTableCellRenderer extends JPanel implements TableCellRenderer
{
    public Component getTableCellRendererComponent(JTable table,
        Object value, boolean isSelected, boolean hasFocus,
```

372 *Au cœur de Java 2 - Fonctions avancées*

```

        int row, int column)
    {
        setBackground((Color)value);
        if (hasFocus)
            setBorder(UIManager.getBorder(
                "Table.focusCellHighlightBorder"));
        else
            setBorder(null);
    }
}

```

Comme vous le voyez, l’afficheur installe une bordure lorsque la cellule possède le focus (la bordure correcte est demandée à `UIManager` ; pour trouver la clé de recherche, nous avons étudié le code source de la classe `DefaultTableCellRenderer`).

Vous souhaitez aussi peut-être définir la couleur du fond de la cellule, pour signaler sa sélection. Nous allons passer cette étape car cela interférerait avec la couleur affichée. L’exemple `ListRenderingTest` que nous avons déjà vu montre comment signaler le statut de sélection dans un afficheur.

ASTUCE

Si votre afficheur est utilisé pour une simple chaîne de caractères ou pour une icône, vous pouvez étendre la classe `DefaultTableCellRenderer`. La sélection et le focus seront pris en compte automatiquement.

Vous devrez demander au tableau d’utiliser cet afficheur pour tous les objets de type `Color`. La méthode `setDefaultRenderer` de la classe `JTable` permet d’effectuer cette association. Il suffit de fournir un objet `Class` et l’afficheur :

```

table.setDefaultRenderer(Color.class,
    new ColorTableCellRenderer());

```

Cet afficheur est maintenant utilisé pour tous les objets du type spécifié.

Modifier une cellule

Pour qu’une cellule soit modifiable, le modèle du tableau doit indiquer quelles cellules sont modifiables en définissant la méthode `isCellEditable`. La plupart du temps, vous choisirez de rendre certaines colonnes modifiables. Dans notre programme d’exemple, quatre colonnes peuvent être modifiées.

```

public boolean isCellEditable(int r, int c)
{
    return c == PLANET_COLUMN
        || c == MOONS_COLUMN
        || c == GASEOUS_COLUMN
        || c == COLOR_COLUMN;
}

private static final int PLANET_COLUMN = 0;
private static final int MOONS_COLUMN = 2;
private static final int GASEOUS_COLUMN = 3;
private static final int COLOR_COLUMN = 4;

```

INFO

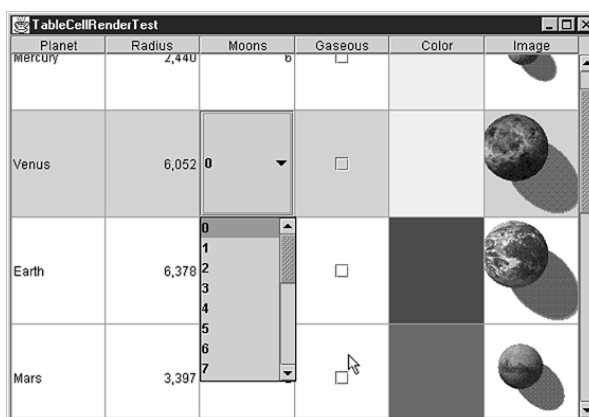
L'AbstractTableModel définit la méthode isCellEditable de sorte qu'elle renvoie toujours false. Le DefaultTableModel surcharge cette méthode pour qu'elle renvoie toujours true.

Si vous exécutez le programme de l'Exemple 5.13, vous verrez que vous pouvez cliquer sur les cases à cocher de la colonne Gazeuse, pour les activer ou les désactiver. Si vous cliquez sur la colonne Lunes, un menu déroulant apparaît (voir Figure 5.34). Nous reviendrons dans un instant sur l'installation de ce type de menu déroulant destiné à modifier le contenu d'une cellule.

Pour terminer, cliquez sur une cellule de la première colonne. Cette cellule prend le focus. Vous pouvez alors saisir de nouvelles informations et modifier le contenu de la cellule.

Figure 5.34

Un éditeur de cellules.



Vous venez de voir les trois variations de la classe DefaultCellEditor en action. Un DefaultCellEditor peut être construit avec un JTextField, un JCheckBox ou un JComboBox. La classe JTable installe automatiquement un éditeur de cases à cocher pour les cellules booléennes et un éditeur de champs de texte pour toutes les cellules qui ne fournissent pas leur propre afficheur. Les champs de texte permettent à l'utilisateur de modifier les chaînes qui ont été créées en appliquant toString à la valeur de retour de la méthode getValueAt du modèle du tableau.

Lorsque la modification est terminée, la valeur modifiée est récupérée par un appel à la méthode getCellEditorValue. Cette méthode doit renvoyer une valeur du type correct (à savoir le type renvoyé par la méthode getColumnTypes du modèle).

Pour obtenir un éditeur de menu déroulant, vous devez définir manuellement l'éditeur de la cellule. En effet, le composant JTable n'a aucune idée des valeurs appropriées pour un type particulier. Pour la colonne Lunes, nous voulions que l'utilisateur puisse choisir n'importe quelle valeur entre 0 et 20. Voici le code d'initialisation de ce menu déroulant :

```
JComboBox moonCombo = new JComboBox();
for (int i = 0; i <= 20; i++)
    moonCombo.addItem(i);
```

Pour construire un DefaultCellEditor, il suffit de fournir le menu déroulant au constructeur :

```
TableCellEditor moonEditor = new DefaultCellEditor(moonCombo);
```

374 *Au cœur de Java 2 - Fonctions avancées*

Ensuite, nous devons installer l'éditeur. Contrairement aux afficheurs de cellules de couleur, cet éditeur ne dépend pas du type de l'objet : nous ne voulons pas forcément l'utiliser pour tous les objets de type `Integer`. Au contraire, nous devons l'installer dans une colonne particulière.

La classe `JTable` enregistre des informations sur les colonnes du tableau dans des objets de type `TableColumn`. Un objet `TableColumnModel` gère les colonnes (la Figure 5.35 montre les relations entre les classes de tableau les plus importantes). Si vous ne voulez pas insérer ou supprimer de colonnes dynamiquement, vous ne vous servirez pas beaucoup du modèle de colonnes du tableau. Cependant, pour obtenir un objet `TableColumn` particulier, vous devrez passer par le modèle de colonnes et lui demander l'objet de colonne correspondant :

```
TableColumnModel columnModel = table.getColumnModel()
TableColumn moonColumn
    = columnModel.getColumn(PlanetTableModel.MOON_COLUMN);
```

Pour terminer, vous pouvez installer l'éditeur de cellules :

```
moonColumn.setCellEditor(moonEditor);
```

Si vos cellules sont plus hautes que les cellules par défaut, vous devrez également définir la hauteur de la ligne :

```
table.setRowHeight(height);
```

Par défaut, toutes les lignes d'un tableau ont la même hauteur. Vous pouvez toutefois définir les hauteurs des lignes individuelles en appelant :

```
table.setRowHeight(row, height);
```

La hauteur de ligne réelle est égale à la hauteur de ligne qui a été définie par ces méthodes, réduite de la marge de la ligne. Le marge de ligne par défaut est de 1, mais vous pouvez également la modifier par l'appel

```
table.setRowMargin(margin);
```

Pour afficher une icône dans l'en-tête, définissez la valeur de l'en-tête :

```
moonColumn.setHeaderValue(new ImageIcon("Moons.gif"));
```

L'en-tête de tableau n'est toutefois pas suffisant pour choisir un afficheur approprié pour la valeur de l'en-tête. Vous devez installer l'afficheur manuellement. Par exemple, pour afficher une icône d'image dans un en-tête de colonne, appelez :

```
moonColumn.setHeaderRenderer(table.getDefaultRenderer(ImageIcon.class));
```

Editeurs personnalisés

Exécutez une nouvelle fois le programme d'exemple et cliquez sur une couleur. Une palette de couleurs apparaît et vous permet de choisir une nouvelle couleur pour une planète. Sélectionnez une couleur et cliquez sur OK. La couleur de la cellule est alors modifiée (voir Figure 5.36).

L'éditeur de couleurs de cellules n'est pas un éditeur de cellules standard, mais une implémentation personnalisée. Pour créer un éditeur de cellules personnalisé, vous devez implémenter l'interface `TableCellEditor`. Cette interface est assez pénible à utiliser, et dans la version JDK 1.3, une classe `AbstractCellEditor` est fournie pour gérer les détails de l'événement.

Figure 5.35

Les relations entre les classes des tableaux.

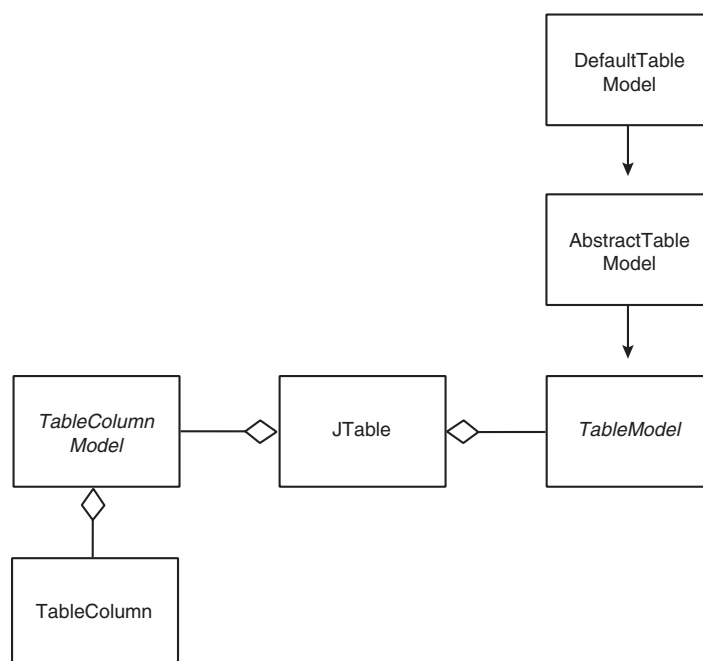
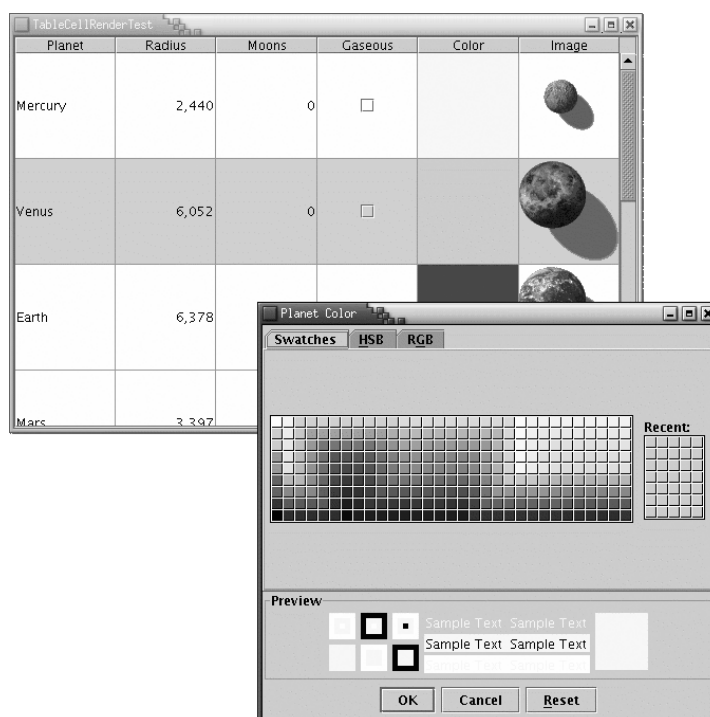


Figure 5.36

Modifier la couleur d'une cellule avec une palette.



376 Au cœur de Java 2 - Fonctions avancées

La méthode `getTableCellEditorComponent` de l'interface `TableCellEditor` a besoin d'un composant pour afficher une cellule. C'est exactement comme pour la méthode `getTableCellRendererComponent` de l'interface `TableCellRenderer`, sauf qu'il n'y a cette fois aucun paramètre `focus`. Comme la cellule doit être modifiée, elle est censée posséder le focus. Dans le cas d'un menu déroulant, le composant de l'éditeur remplace temporairement l'afficheur. Dans notre exemple, nous renvoyons un écran vide, qui n'est pas coloré. Ceci indique à l'utilisateur que la cellule est en cours de modification.

Ensuite, il faut que l'éditeur apparaisse dans une nouvelle fenêtre lorsque l'utilisateur clique sur la cellule.

La classe `JTable` appelle votre éditeur avec un événement (comme un clic de souris) pour déterminer si cet événement est acceptable pour initialiser le processus de la modification. Notre éditeur n'est pas très sélectif et accepte tous les événements.

```
public boolean isCellEditable(EventObject anEvent)
{
    return true;
}
```

Cependant, si cette méthode renvoie `false`, le tableau n'insérera pas le composant de l'éditeur.

Une fois que le composant de l'éditeur est installé, la méthode `shouldSelectCell` est appelée, avec le même événement. Le processus de modification doit commencer dans cette méthode, par exemple en affichant une fenêtre externe.

```
public boolean shouldSelectCell(EventObject anEvent)
{
    colorDialog.setVisible(true);
    return true;
}
```

Si l'utilisateur doit annuler la modification en cours, le tableau appelle la méthode `cancelCellEditing`. Si l'utilisateur a cliqué sur une autre cellule du tableau, il appelle la méthode `stopCellEditing`. Dans les deux cas, il convient alors de cacher la boîte de dialogue. Lorsque votre méthode `stopCellEditing` est appelée, le tableau peut essayer d'utiliser la valeur en cours de modification. C'est pourquoi il ne faut renvoyer `true` que lorsque la valeur courante est valide. Pour notre palette de couleurs, n'importe quelle valeur est valide. Mais si vous modifiez d'autres types de données, vous pouvez ainsi vérifier que seules les valeurs correctes sont renvoyées de l'éditeur.

Vous devrez aussi appeler les méthodes de superclasse pour déclencher des événements, faute de quoi la modification ne fonctionnera pas correctement :

```
public void cancelCellEditing()
{
    colorDialog.setVisible(false);
    super.cancelCellEditing();
}
```

Enfin, vous devez fournir une méthode qui produise la valeur fournie par l'utilisateur au cours de la procédure de modification :

```
public Object getCellEditorValue()
{
    return colorChooser.getColor();
}
```

Pour résumer, votre éditeur personnalisé doit :

1. Étendre la classe `AbstractCellEditor` et implémenter l'interface `TableCellEditor`.
2. Définir la méthode `getTableCellEditorComponent` pour fournir un composant. Il peut s'agir d'un composant dummy (si vous affichez une boîte de dialogue) ou d'un composant pour la modification sur place, comme un menu déroulant ou un champ de texte.
3. Définir les méthodes `shouldSelectCell`, `stopCellEditing` et `cancelCellEditing` pour gérer le début, la réalisation et l'annulation de la procédure de modification. `stopCellEditing` et `cancelCellEditing` doivent appeler les méthodes de superclasse pour s'assurer que les écouteurs sont avertis.
4. Définir la méthode `getCellEditorValue` pour renvoyer la valeur qui résulte de la procédure d'édition.

Enfin, vous devez indiquer le moment où l'utilisateur a terminé la modification en appelant les méthodes `stopCellEditing` et `cancelCellEditing`. Lorsque nous construisons la boîte de dialogue de couleurs, nous installons également des appels d'acceptation et d'annulation qui déclenchent ces événements.

```
colorDialog = JColorChooser.createDialog(null,
    "Planet Color", false, colorChooser,
    new
    ActionListener() // écouteur du bouton OK
    {
        public void actionPerformed(ActionEvent event)
        {
            stopCellEditing();
        }
    },
    new
    ActionListener() // écouteur du bouton Annuler
    {
        public void actionPerformed(ActionEvent event)
        {
            cancelCellEditing();
        }
    }
    ));
```

La modification est terminée lorsque l'utilisateur ferme la boîte de dialogue. Nous obtenons ceci en installant un écouteur de fenêtre.

```
colorDialog.addWindowListener(new
    WindowAdapter()
    {
        public void windowClosing(WindowEvent event)
        {
            cancelCellEditing();
        }
    }
    ));
```

Cela termine l'implémentation de notre éditeur personnalisé.

Vous savez maintenant comment rendre une cellule modifiable et comment installer un éditeur. Mais il existe encore un point délicat : comment mettre à jour le modèle en fonction de la valeur modifiée.

378 *Au cœur de Java 2 - Fonctions avancées*

Lorsque la modification est terminée, la classe `JTable` appelle la méthode suivante du modèle du tableau :

```
void setValueAt(Object value, int r, int c)
```

Vous devrez surcharger cette méthode pour enregistrer la nouvelle valeur. Le paramètre `value` correspond à l'objet renvoyé par l'éditeur de cellules. Si vous avez implémenté l'éditeur de cellule, vous connaissez le type d'objet renvoyé par la méthode `getCellEditorValue`. Dans le cas de `DefaultCellEditor`, il existe trois possibilités pour cette valeur. Il s'agit d'un `Boolean` si la cellule concernée est une case à cocher et d'une chaîne pour un champ de texte. Si la valeur provient d'un menu déroulant, il s'agit alors de l'objet sélectionné par l'utilisateur.

Si l'objet `value` n'est pas du type approprié, vous devrez le convertir. Cela se produit le plus souvent lorsqu'un nombre est modifié dans un champ de texte. Dans notre exemple, nous plaçons des objets `Integer` dans le menu déroulant, de sorte qu'aucune conversion n'est nécessaire.

Exemple 5.13 : *TableCellRenderTest.java*

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.table.*;

/**
 * Ce programme montre l'affichage et la modification de cellules
 * dans un tableau.
 */
public class TableCellRenderTest
{
    public static void main(String[] args)
    {
        JFrame frame = new TableCellRenderFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc contient un tableau des données des planètes.
 */
class TableCellRenderFrame extends JFrame
{
    public TableCellRenderFrame()
    {
        setTitle("TableCellRenderTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        TableModel model = new PlanetTableModel();
        JTable table = new JTable(model);
        table.setRowSelectionAllowed(false);

        // configure les afficheurs et les éditeurs

        table.setDefaultRenderer(Color.class,
            new ColorTableCellRenderer());
    }
}
```

```

        table.setDefaultEditor(Color.class,
            new ColorTableCellEditor());

        JComboBox moonCombo = new JComboBox();
        for (int i = 0; i <= 20; i++)
            moonCombo.addItem(i);

        TableColumnModel columnModel = table.getColumnModel();
        TableColumn moonColumn
            = columnModel.getColumn(PlanetTableModel.MOONS_COLUMN);
        moonColumn.setCellEditor(new DefaultCellEditor(moonCombo));
        moonColumn.setHeaderRenderer(
            table.getDefaultRenderer(ImageIcon.class));
        moonColumn.setHeaderValue(new ImageIcon("Moons.gif"));

        // affiche le tableau

        table.setRowHeight(100);
        add(new JScrollPane(table), BorderLayout.CENTER);
    }

    private static final int DEFAULT_WIDTH = 600;
    private static final int DEFAULT_HEIGHT = 400;
}

/**
 * Le modèle du tableau des planètes spécifie les valeurs et les
 * propriétés d'affichage et de modification pour les données des
 * planètes.
 */
class PlanetTableModel extends AbstractTableModel
{
    public String getColumnName(int c)
    { return columnNames[c]; }
    public Class getColumnClass(int c)
    { return cells[0][c].getClass(); }
    public int getColumnCount()
    { return cells[0].length; }
    public int getRowCount()
    { return cells.length; }
    public Object getValueAt(int r, int c)
    { return cells[r][c]; }
    public void setValueAt(Object obj, int r, int c)
    { cells[r][c] = obj; }
    public boolean isCellEditable(int r, int c)
    {
        return c == PLANET_COLUMN
            || c == MOONS_COLUMN
            || c == GASEOUS_COLUMN
            || c == COLOR_COLUMN;
    }

    public static final int PLANET_COLUMN = 0;
    public static final int MOONS_COLUMN = 2;
    public static final int GASEOUS_COLUMN = 3;
    public static final int COLOR_COLUMN = 4;

    private Object[][] cells =
    {

```

380 *Au cœur de Java 2 - Fonctions avancées*

```
        { "Mercure", 2440.0, 0, false, Color.yellow,
          new ImageIcon("Mercury.gif") },
        { "Vénus", 6052.0, 0, false, Color.yellow,
          new ImageIcon("Venus.gif") },
        { "Terre", 6378.0, 1, false, Color.blue,
          new ImageIcon("Earth.gif") },
        { "Mars", 3397.0, 2, false, Color.red,
          new ImageIcon("Mars.gif") },
        { "Jupiter", 71492.0, 16, true, Color.orange,
          new ImageIcon("Jupiter.gif") },
        { "Saturne", 60268.0, 18, true, Color.orange,
          new ImageIcon("Saturn.gif") },
        { "Uranus", 25559.0, 17, true, Color.blue,
          new ImageIcon("Uranus.gif") },
        { "Neptune", 24766.0, 8, true, Color.blue,
          new ImageIcon("Neptune.gif") },
        { "Pluton", 1137.0, 1, false, Color.black,
          new ImageIcon("Pluto.gif") }
    };

    private String[] columnNames =
        { "Planète", "Rayon", "Lunes", "Gazeuse",
          "Couleur", "Image" };
}

/**
 * Cet afficheur présente une valeur de couleur sous forme de panneau
 * avec la couleur donnée.
 */
class ColorTableCellRenderer extends JPanel implements TableCellRenderer
{
    public Component getTableCellRendererComponent(JTable table,
        Object value, boolean isSelected, boolean hasFocus,
        int row, int column)
    {
        setBackground((Color)value);
        if (hasFocus)
            setBorder(UIManager.getBorder(
                "Table.focusCellHighlightBorder"));
        else
            setBorder(null);
        return this;
    }
}

/**
 * Cet éditeur affiche une boîte de dialogue des couleurs pour modifier
 * la valeur d'une cellule.
 */
class ColorTableCellEditor extends ColorTableCellRenderer
    implements TableCellEditor
{
    public ColorTableCellEditor()
    {
        panel = new JPanel();
        // prépare la boîte de dialogue de couleur
    }
}
```

```

        colorChooser = new JColorChooser();
        colorDialog = JColorChooser.createDialog(null,
            "Couleur de la planète", false, colorChooser,
            new
                ActionListener() // écouteur du bouton OK
            {
                public void actionPerformed(ActionEvent event)
                { stopCellEditing(); }
            },
            new
                ActionListener() // écouteur du bouton Annuler
            {
                public void actionPerformed(ActionEvent event)
                { cancelCellEditing(); }
            });
        colorDialog.addWindowListener(new
            WindowAdapter()
        {
            public void windowClosing(WindowEvent event)
            { cancelCellEditing(); }
        });
    }

    public Component getTableCellEditorComponent(JTable table,
        Object value, boolean isSelected, int row, int column)
    {
        /* c'est ici que nous récupérons la valeur Couleur courante.
        Nous l'enregistrons dans la boîte de dialogue au cas où
        l'utilisateur commencerait des modifications.
        */
        colorChooser.setColor((Color)value);
        return panel;
    }

    public boolean shouldSelectCell(EventObject anEvent)
    {
        // début des modifications
        colorDialog.setVisible(true);

        // indique à l'appelant qu'il peut sélectionner cette cellule
        return true;
    }

    public void cancelCellEditing()
    {
        // les modifications sont annulées, cacher la boîte de dialogue
        colorDialog.setVisible(false);
        super.cancelCellEditing();
    }

    public boolean stopCellEditing()
    {
        // les modifications sont terminées, cacher la boîte de dialogue
        colorDialog.setVisible(false);
        super.stopCellEditing();
    }

```

382 *Au cœur de Java 2 - Fonctions avancées*

```

        // indique à l'appelant qu'il peut utiliser la valeur de couleur
        return true;
    }

    public Object getCellEditorValue()
    {
        return colorChooser.getColor();
    }

    private Color color;
    private JColorChooser colorChooser;
    private JDialog colorDialog;
    private JPanel panel;
}

```

javax.swing.JTable 1.2

- void setRowHeight(int height)
Définit la hauteur de toutes les lignes du tableau, en pixels.
- void setRowHeight(int row, int height)
Définit la hauteur de la ligne donnée du tableau, en pixels.
- void setRowMargin(int margin)
Définit la quantité d'espace vide entre les cellules de lignes adjacentes.
- int getRowHeight()
Définit la hauteur par défaut de toutes les lignes du tableau.
- int getRowHeight(int row)
Récupère la hauteur de la ligne donnée du tableau.
- int getRowMargin()
Récupère la quantité d'espace vide entre les cellules des lignes adjacentes.
- Rectangle getCellRect(int row, int column, boolean includeSpacing)
Renvoie un rectangle entourant la cellule du tableau.
Paramètres : row, column la ligne et la colonne de la cellule
 includeSpacing vaut true si les espaces autour de la cellule doivent être inclus
- Color getSelectionBackground()
• Color getSelectionForeground()
Renvoient les couleurs de fond et d'arrière-plan à utiliser pour les cellules sélectionnées.
- TableCellRenderer getDefaultRenderer(Class<?> type)
Récupère l'afficheur par défaut pour le type donné.
- TableCellEditor getDefaultEditor(Class<?> type)
Récupère l'éditeur par défaut pour le type donné.

javax.swing.table.TableModel 1.2

- Class getColumnClass(int columnIndex)
Renvoie la classe pour les valeurs de cette colonne. Cette information est utilisée par l'afficheur et l'éditeur de cellules.

javax.swing.table.TableCellRenderer 1.2

- Component `getTableCellRendererComponent(JTable table, Object value, boolean selected, boolean hasFocus, int row, int column)`

Renvoie un composant dont la méthode `paint` est invoquée pour afficher une cellule de tableau.

Paramètres :

<code>table</code>	le tableau contenant la cellule à afficher
<code>value</code>	la cellule à afficher
<code>selected</code>	vaut <code>true</code> si la cellule est sélectionnée
<code>hasFocus</code>	vaut <code>true</code> si la cellule possède le focus
<code>row, column</code>	la ligne et la colonne de la cellule

javax.swing.table.TableColumnModel 1.2

- TableColumn `getColumn(int index)`

Renvoie l'objet de colonne qui décrit la colonne située à l'indice spécifié.

javax.swing.table.TableColumn 1.2

- void `setCellEditor(TableCellEditor editor)`
- void `setCellRenderer(TableCellRenderer renderer)`

Définissent l'éditeur ou l'afficheur de cellules pour toutes les cellules de cette colonne.

- void `setHeaderRenderer(TableCellRenderer renderer)`
Définit l'afficheur de cellules pour la cellule d'en-tête dans cette colonne.
- void `setHeaderValue(Object value)`
Définit la valeur à afficher pour l'en-tête de cette colonne.

javax.swing.table.DefaultCellEditor 1.2

- DefaultCellEditor(JComboBox comboBox)

Construit un éditeur de cellules qui présente un menu déroulant pour sélectionner les valeurs de la cellule.

javax.swing.CellEditor 1.2

- boolean `isCellEditable(EventObject event)`

Renvoie `true` si l'événement permet d'initialiser un processus de modification pour cette cellule.

- boolean `shouldSelectCell(EventObject anEvent)`

Démarre le processus de modification. Renvoie `true` si la cellule modifiée doit être *sélectionnée*. Normalement, vous renverrez le plus souvent `true`, mais vous pouvez également renvoyer `false` si vous ne voulez pas que le processus de modification change la sélection de la cellule.

- void `cancelCellEditing()`

Abandonne le processus de modification. Vous pouvez notamment abandonner une modification partielle.

- boolean `stopCellEditing()`

Interrompt le processus de modification, dans l'intention d'en utiliser le résultat. Renvoie `true` si la valeur modifiée est dans un état permettant de l'utiliser par la suite.

384 Au cœur de Java 2 - Fonctions avancées

- `Object getCellEditorValue()`
Renvoie le résultat de l'édition.
- `void addCellEditorListener(CellEditorListener l)`
- `void removeCellEditorListener(CellEditorListener l)`
Ajoutent et suppriment l'écouteur obligatoire de l'éditeur de cellules.

javax.swing.table.TableCellEditor 1.2

- `Component getTableCellEditorComponent(JTable table, Object value, boolean selected, int row, int column)`

Renvoie un composant dont la méthode `paint` affiche une cellule de tableau.

<i>Paramètres :</i>	<code>table</code>	le tableau contenant la cellule à afficher
	<code>value</code>	la cellule à afficher
	<code>selected</code>	vaut <code>true</code> si la cellule est sélectionnée
	<code>row, column</code>	la ligne et la colonne de la cellule

Travailler avec les lignes et les colonnes

Dans cette sous-section, vous allez apprendre à manipuler les lignes et les colonnes d'un tableau. Au cours de la lecture de ce livre, vous devez garder à l'esprit que Swing n'est pas du tout symétrique, c'est-à-dire que les opérations supportées par les lignes d'une part et par les colonnes d'autre part ne sont pas les mêmes. Le composant "tableau" a été optimisé pour afficher des lignes d'informations de même structure, comme le résultat d'une requête de base de données, et non pour une grille bidimensionnelle arbitraire d'objets. Nous reviendrons sur cette asymétrie au cours de cette sous-section.

Modifier la taille des colonnes

La classe `TableColumn` permet de contrôler la taille des colonnes. Vous pouvez ainsi choisir la largeur préférée, minimale ou maximale, avec les méthodes suivantes :

```
void setPreferredWidth(int width)
void setMinWidth(int width)
void setMaxWidth(int width)
```

Cette information est utilisée par le tableau pour la mise en forme des colonnes.

Utilisez la méthode

```
void setResizable(boolean resizable)
```

pour permettre ou non à l'utilisateur de modifier la largeur d'une colonne.

Vous pouvez modifier la largeur d'une colonne avec la méthode suivante :

```
void setWidth(int width)
```

Lorsque la taille d'une colonne est modifiée, le comportement par défaut est de conserver la largeur totale du tableau. Naturellement, dans ce cas, les changements de largeur de la colonne modifiée doivent être reportés sur les autres colonnes. Avec le comportement par défaut, ces changements seront intégralement reportés sur la colonne à droite de la colonne modifiée. Cela permet à l'utilisateur d'ajuster toutes les colonnes de gauche à droite.

Vous pouvez choisir un autre comportement dans le Tableau 5.2 en utilisant la méthode

```
void setAutoResizeMode(int mode)
```

de la classe `JTable`.

Tableau 5.2 : Modes de modification de largeur des colonnes

<i>Mode</i>	<i>comportement</i>
<code>AUTO_RESIZE_OFF</code>	Ne modifie pas les autres colonnes, change la taille du tableau
<code>AUTO_RESIZE_NEXT_COLUMN</code>	Modifie uniquement la taille de la colonne suivante
<code>AUTO_RESIZE_SUBSEQUENT_COLUMNS</code>	Modifie identiquement toutes les colonnes restantes. C'est le comportement par défaut
<code>AUTO_RESIZE_LAST_COLUMN</code>	Modifie uniquement la taille de la dernière colonne
<code>AUTO_RESIZE_ALL_COLUMNS</code>	Modifie toutes les colonnes du tableau. Ce choix est à éviter parce qu'il devient très difficile d'ajuster la largeur de plusieurs colonnes

Sélectionner des lignes, des colonnes et des cellules

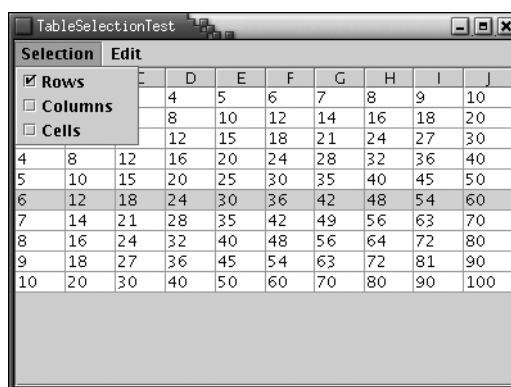
En fonction du mode de sélection, l'utilisateur peut sélectionner des lignes, des colonnes ou des cellules isolées du tableau. Par défaut, la sélection des lignes est autorisée. Une ligne entière est sélectionnée lorsque l'utilisateur clique sur une cellule (voir Figure 5.37). Appelez la méthode

```
table.setRowSelectionAllowed(false)
```

pour supprimer cette sélection par lignes.

Figure 5.37

Sélection d'une ligne.



Lorsque la sélection par lignes est permise, vous pouvez choisir parmi plusieurs modes de sélection : une seule ligne, un ensemble continu de lignes ou n'importe quelles lignes. Vous devrez donc récupérer le *modèle de sélection* et utiliser sa méthode `setSelectionMode` :

```
table.getSelectionModel().setSelectionMode(mode);
```


386 Au cœur de Java 2 - Fonctions avancées

Ici, mode peut prendre l'une des trois valeurs suivantes :

```
ListSelectionMode.SINGLE_SELECTION
ListSelectionMode.SINGLE_INTERVAL_SELECTION
ListSelectionMode.MULTIPLE_INTERVAL_SELECTION
```

La sélection de colonnes est désactivée par défaut. Elle peut être activée avec l'appel suivant :

```
table.setColumnSelectionAllowed(true)
```

L'activation des deux types de sélections (lignes et colonnes) équivaut à activer la sélection de cellules. L'utilisateur sélectionne alors des plages de cellules (voir Figure 5.38). Vous pouvez également activer ce réglage par l'appel

```
table.setCellSelectionEnabled(true)
```

Figure 5.38

Sélectionner une plage de cellules.

	A	B	C	D	E	F	G	H	I	J
1	2	3	4	5	6	7	8	9	10	
2	4	6	8	10	12	14	16	18	20	
3	6	9	12	15	18	21	24	27	30	
4	8	12	16	20	24	28	32	36	40	
5	10	15	20	25	30	35	40	45	50	
6	12	18	24	30	36	42	48	54	60	
7	14	21	28	35	42	49	56	63	70	
8	16	24	32	40	48	56	64	72	80	
9	18	27	36	45	54	63	72	81	90	
10	20	30	40	50	60	70	80	90	100	

INFO

Dans les premières versions de la boîte à outils Swing, lors de la définition de sélection pour les lignes et les colonnes, chaque clic de souris sélectionne une zone en forme de "+" constituée de la ligne et de la colonne où se trouve le curseur.

Vous pouvez identifier les lignes et les colonnes sélectionnées en appelant les méthodes `getSelectedRows` et `getSelectedColumns`. Ces deux méthodes renvoient un tableau d'indices `int[]` correspondant aux éléments sélectionnés.

Pour voir comment fonctionne la sélection par cellules, lancez le programme de l'Exemple 5.14. Vous pouvez activer la sélection par lignes, par colonnes ou par cellules grâce au menu Sélection.

Cacher et afficher des colonnes

La méthode `removeColumn` de la classe `JTable` supprime une colonne de l'affichage d'un tableau. Les données de la colonne ne sont en fait pas réellement supprimées du modèle, elles sont juste cachées. La méthode `removeColumn` prend un argument de type `TableColumn`. Si vous possédez un

numéro de colonne, par exemple à partir d'un appel à `getSelectedColumns`, vous devrez encore demander au modèle de l'arbre la colonne réelle de l'arbre qui correspond à ce numéro :

```
TableColumnModel columnModel = table.getColumnModel();
TableColumn column = columnModel.getColumn(i);
table.removeColumn(column);
```

Si vous avez mémorisé ce numéro de colonne, il est possible de l'afficher à nouveau :

```
table.addColumn(column);
```

Cette méthode ajoute la colonne à la fin du tableau. Si vous préférez qu'elle apparaisse à un autre endroit, vous devez appeler la méthode `moveColumn`.

Vous pouvez également ajouter une nouvelle colonne qui correspond à un indice de colonne du modèle du tableau, en ajoutant un nouvel objet `TableColumn` :

```
table.addColumn(new TableColumn(modelColumnIndex));
```

En fait, plusieurs colonnes du tableau peuvent être affichées à partir d'une même colonne du modèle.

Cependant, il n'existe pas de méthode dans `JTable` pour cacher ou afficher des lignes. Si vous souhaitez cacher une ligne, vous devrez avoir recours à un modèle de filtre semblable au filtre de tri que vous avez vu précédemment.

Ajouter et supprimer des lignes dans le modèle de tableau par défaut

La classe `DefaultTableModel` est une classe concrète qui implémente l'interface `TableModel`. Elle stocke une grille bidimensionnelle d'objets. Si vos données se trouvent déjà sous cette forme, il n'est pas nécessaire de recopier toutes ces données dans un modèle de tableau par défaut. En revanche, si vous possédez peu de données, il peut être pratique de les copier pour obtenir rapidement un tableau. La classe `DefaultTableModel` possède des méthodes permettant d'ajouter des lignes et des colonnes et de supprimer des lignes.

Les méthodes `addRow` et `addColumn` ajoutent une nouvelle ligne ou une nouvelle colonne dans les données. Elles nécessitent un tableau `Object[]` ou un vecteur qui contient les nouvelles données. Avec la méthode `addColumn`, vous devez aussi fournir le nom de la nouvelle colonne. Ces méthodes ajoutent les nouvelles données à la fin de la grille. Pour insérer une ligne au milieu des données existantes, utilisez la méthode `insertRow`. Il n'existe aucune méthode pour insérer une colonne au milieu d'une grille.

Inversement, la méthode `removeRow` supprime une ligne du modèle. Il n'existe aucune méthode pour supprimer une colonne.

Comme l'objet `JTable` s'enregistre lui-même comme un écouteur de modèle de tableau, le modèle avertit le tableau lorsque des données sont insérées ou supprimées. Pour l'instant, le tableau met à jour l'affichage.

Le programme de l'Exemple 5.14 montre comment fonctionnent la sélection et la modification. Un modèle de tableau par défaut contient un simple ensemble de données, comme une table de multiplication. Le menu Edition contient les commandes suivantes :

- Cacher toutes les colonnes sélectionnées ;
- Afficher toutes les colonnes cachées ;

388 Au cœur de Java 2 - Fonctions avancées

- Supprimer du modèle les lignes sélectionnées ;
- Ajouter une ligne de données à la fin du modèle.

Cet exemple termine notre étude portant sur les tableaux Swing. Sur le plan conceptuel, les tableaux sont un peu plus simples à appréhender que les arbres, parce que le modèle de données sous-jacent (une grille d'objets) est simple à afficher. Cependant, dans la réalité de l'implémentation, un tableau est plus complexe qu'un arbre. Cette complexité est en partie due aux en-têtes de colonnes, aux afficheurs et aux éditeurs propres à chaque colonne. Dans cette section, nous nous sommes concentrés sur les sujets que vous aurez le plus de chances de rencontrer en codant : afficher les informations d'une base de données, trier des données, modifier et afficher des cellules personnalisées. Si vous rencontrez d'autres problèmes plus spécialisés, nous vous renvoyons une nouvelle fois aux ouvrages (en langue anglaise) *Core Java Foundation Classes*, de Kim Topley et *Graphic Java 2*, de David Geary.

Exemple 5.14 : *TableSelectionTest.java*

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.text.*;
import javax.swing.*;
import javax.swing.table.*;

/**
 * Ce programme montre la sélection, l'ajout et le retrait de
 * lignes et de colonnes.
 */
public class TableSelectionTest
{
    public static void main(String[] args)
    {
        JFrame frame = new TableSelectionFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc présente une table de multiplication et des menus pour
 * définir le mode de sélection de ligne/colonne/cellule et pour
 * ajouter et supprimer des lignes et des colonnes.
 */
class TableSelectionFrame extends JFrame
{
    public TableSelectionFrame()
    {
        setTitle("TableSelectionTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // définit la table de multiplication
        model = new DefaultTableModel(10, 10);
    }
}
```

```

for (int i = 0; i < model.getRowCount(); i++)
    for (int j = 0; j < model.getColumnCount(); j++)
        model.setValueAt((i + 1) * (j + 1), i, j);

table = new JTable(model);

add(new JScrollPane(table), "Center");

removedColumns = new ArrayList<TableColumn>();

// crée le menu

JMenuBar menuBar = new JMenuBar();
setJMenuBar(menuBar);

JMenu selectionMenu = new JMenu("Sélection");
menuBar.add(selectionMenu);

final JCheckBoxMenuItem rowsItem
    = new JCheckBoxMenuItem("Lignes");
final JCheckBoxMenuItem columnsItem
    = new JCheckBoxMenuItem("Colonnes");
final JCheckBoxMenuItem cellsItem
    = new JCheckBoxMenuItem("Cellules");

rowsItem.setSelected(table.getRowSelectionAllowed());
columnsItem.setSelected(table.getColumnSelectionAllowed());
cellsItem.setSelected(table.getCellSelectionEnabled());

rowsItem.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            table.clearSelection();
            table.setRowSelectionAllowed(
                rowsItem.isSelected());
            cellsItem.setSelected(
                table.getCellSelectionEnabled());
        }
    });
selectionMenu.add(rowsItem);

columnsItem.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            table.clearSelection();
            table.setColumnSelectionAllowed(
                columnsItem.isSelected());
            cellsItem.setSelected(
                table.getCellSelectionEnabled());
        }
    });
selectionMenu.add(columnsItem);

cellsItem.addActionListener(new
    ActionListener()

```

390 Au cœur de Java 2 - Fonctions avancées

```
{
    public void actionPerformed(ActionEvent event)
    {
        table.clearSelection();
        table.setCellSelectionEnabled(
            cellsItem.isSelected());
        rowsItem.setSelected(
            table.getRowSelectionAllowed());
        columnsItem.setSelected(
            table.getColumnSelectionAllowed());
    }
});
selectionMenu.add(cellsItem);

JMenu tableMenu = new JMenu("Edition");
menuBar.add(tableMenu);

JMenuItem hideColumnsItem = new JMenuItem("Cacher les colonnes");
hideColumnsItem.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            int[] selected = table.getSelectedColumns();
            TableColumnModel columnModel
                = table.getColumnModel();

            /* supprime les colonnes de l'affichage, en commençant par le
            dernier indice de sorte que les numéros de colonnes ne sont pas
            affectés.
            */

            for (int i = selected.length - 1; i >= 0; i--)
            {
                TableColumn column
                    = columnModel.getColumn(selected[i]);
                table.removeColumn(column);

                // enregistre les colonnes supprimées pour la commande
                // "Afficher les colonnes"
                removedColumns.add(column);
            }
        }
    });
tableMenu.add(hideColumnsItem);

JMenuItem showColumnsItem = new JMenuItem("Afficher les colonnes");
showColumnsItem.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            // restaure toutes les colonnes supprimées
            for (TableColumn tc : removedColumns)
                table.addColumn(tc);
            removedColumns.clear();
        }
    });
});
```

```

        tableMenu.add(showColumnsItem);

        JMenuItem addRowItem = new JMenuItem("Ajouter une ligne");
        addRowItem.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    // ajoute une nouvelle ligne à la table de multiplication
                    // dans le modèle

                    Integer[] newCells
                        = new Integer[model.getColumnCount()];
                    for (int i = 0; i < newCells.length; i++)
                        newCells[i] = new Integer((i + 1)
                            * (model.getRowCount() + 1));
                    model.addRow(newCells);
                }
            });
        tableMenu.add(addRowItem);

        JMenuItem removeRowsItem = new JMenuItem("Supprime des lignes");
        removeRowsItem.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    int[] selected = table.getSelectedRows();

                    for (int i = selected.length - 1; i >= 0; i--)
                        model.removeRow(selected[i]);
                }
            });
        tableMenu.add(removeRowsItem);

        JMenuItem clearCellsItem = new JMenuItem("Efface les cellules");
        clearCellsItem.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    for (int i = 0; i < table.getRowCount(); i++)
                        for (int j = 0; j < table.getColumnCount(); j++)
                            if (table.isCellSelected(i, j))
                                table.setValueAt(0, i, j);
                }
            });
        tableMenu.add(clearCellsItem);
    }

    private DefaultTableModel model;
    private JTable table;
    private ArrayList<TableColumn> removedColumns;

    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 300;
}

```

392 *Au cœur de Java 2 - Fonctions avancées*

javax.swing.JTable 1.2

- `void setAutoResizeMode(int mode)`
Définit le mode d'ajustement automatique de la taille des colonnes.
Paramètres : `mode` peut prendre n'importe quelle valeur parmi les suivantes :
 `AUTO_RESIZE_OFF`, `AUTO_RESIZE_NEXT_COLUMN`,
 `AUTO_RESIZE_SUBSEQUENT_COLUMNS`,
 `AUTO_RESIZE_LAST_COLUMN`, `AUTO_RESIZE_ALL_COLUMNS`
- `ListSelectionModel getSelectionModel()`
Renvoie le modèle de sélection de liste. Ce modèle est nécessaire pour choisir une sélection de lignes, de colonnes ou de cellules.
- `void setRowSelectionAllowed(boolean b)`
Si `b` vaut `true`, les lignes peuvent être sélectionnées lorsque l'utilisateur clique sur des cellules.
- `void setColumnSelectionAllowed(boolean b)`
Si `b` vaut `true`, les colonnes peuvent être sélectionnées lorsque l'utilisateur clique sur des cellules.
- `void setCellSelectionEnabled(boolean b)`
Si `b` vaut `true`, chaque cellule est sélectionnée. Ceci équivaut à appeler à la fois `setRowSelectionAllowed(b)` et `setColumnSelectionAllowed(b)`.
- `boolean getRowSelectionAllowed()`
Renvoie `true` si la sélection de lignes est autorisée.
- `boolean getColumnSelectionAllowed()`
Renvoie `true` si la sélection de colonnes est autorisée.
- `boolean getCellSelectionEnabled()`
Renvoie `true` si la sélection de lignes et de colonnes est autorisée.
- `void clearSelection()`
Désélectionne toutes les lignes et colonnes sélectionnées.
- `void addColumn(TableColumn column)`
Ajoute une colonne dans l'affichage du tableau.
- `void moveColumn(int from, int to)`
Déplace une colonne du tableau de l'indice `from` à l'indice `to`. Seul l'affichage est affecté.
- `void removeColumn(TableColumn column)`
Supprime la colonne spécifiée de l'affichage.

javax.swing.table.TableColumn 1.2

- `TableColumn(int modelColumnIndex)`
Construit une colonne du tableau pour afficher la colonne du modèle correspondant à l'indice spécifié.
- `void setPreferredWidth(int width)`
- `void setMinWidth(int width)`
- `void setMaxWidth(int width)`
Définissent la largeur préférée, minimale et maximale de cette colonne du tableau sur `width`.

- `void setWidth(int width)`
Choisit `width` comme nouvelle largeur de cette colonne.
- `void setResizable(boolean b)`
Si `b` vaut `true`, la largeur de cette colonne peut être modifiée.

`javax.swing.ListSelectionModel 1.2`

- `void setSelectionMode(int mode)`
Paramètres : `mode` l'une des valeurs suivantes : `SINGLE_SELECTION`,
 `SINGLE_INTERVAL_SELECTION` ou
 `MULTIPLE_INTERVAL_SELECTION`

`javax.swing.DefaultTableModel 1.2`

- `void addRow(Object[] rowData)`
- `void addColumn(Object columnName, Object[] columnData)`
Ajoutent une ligne ou une colonne de données à la fin du modèle du tableau.
- `void insertRow(int row, Object[] rowData)`
Ajoute une ligne de données à l'indice `row`.
- `removeRow(int row)`
Supprime la ligne spécifiée dans le modèle.
- `void moveRow(int start, int end, int to)`
Déplace toutes les lignes dont les indices sont compris entre `start` et `end` à une nouvelle position commençant à `to`.

Composants de texte stylisés

Dans le Volume 1, nous avons abordé les classes de composants de texte essentielles, `JTextField` et `JTextArea`. Naturellement, ces classes sont très pratiques pour obtenir du texte provenant de l'utilisateur. Il existe une autre classe pratique, `JEditorPane`, qui affiche et modifie du texte au format HTML ou RTF. RTF signifie *Rich Text Format* ; il s'agit d'un format utilisé par un certain nombre d'applications Microsoft pour échanger des documents. Ce format est très peu documenté et il ne fonctionne souvent pas très bien, même entre des applications Microsoft. Nous n'aborderons pas les caractéristiques de ce format dans ce livre.

Pour être honnête, le `JEditorPane` est limité pour l'instant. L'afficheur HTML peut afficher des fichiers simples, mais il aura des problèmes pour afficher les pages complexes que vous pourrez trouver sur le Web. L'éditeur HTML est quant à lui assez pauvre et instable.

Nous pensons que la meilleure application de `JEditorPane` est d'afficher l'aide d'un programme au format HTML. Comme vous pouvez choisir les fichiers d'aide que vous fournissez, vous pouvez ignorer les caractéristiques que le `JEditorPane` ne gère pas très bien.

INFO

Pour plus d'informations sur un système d'aide réellement industriel, consultez JavaHelp sur <http://java.sun.com/products/javahelp/index.html>.

394 Au cœur de Java 2 - Fonctions avancées

INFO

La sous-classe `JTextPane` de `JEditorPane` peut gérer du texte stylisé, ainsi que des composants intégrés. Nous n'aborderons pas ce composant dans ce livre. Si vous devez implémenter un composant qui permet à l'utilisateur de saisir du texte stylisé, étudiez l'implémentation de la démonstration `StylePad` incluse dans le JDK.

Le programme de l'Exemple 5.15 contient un panneau d'édition qui montre le contenu d'une page HTML. Saisissez une URL dans le champ de texte. Cette URL doit commencer par "http:" ou "file:". Cliquez ensuite sur le bouton "Charger". La page HTML sélectionnée est alors affichée dans le panneau de l'éditeur (voir Figure 5.39).

Figure 5.39

Le panneau de l'éditeur affiche une page HTML.



Les hyperliens sont actifs : si vous cliquez sur un lien, l'application charge la page correspondante. Le bouton Précédente permet de revenir à la page précédente.

Ce programme est donc en fait un navigateur très simple, qui ne possède aucune des caractéristiques pratiques des navigateurs du commerce, comme la mise en mémoire des pages visitées ou une liste de sites favoris. De plus, le panneau de l'éditeur n'affiche même pas les applets !

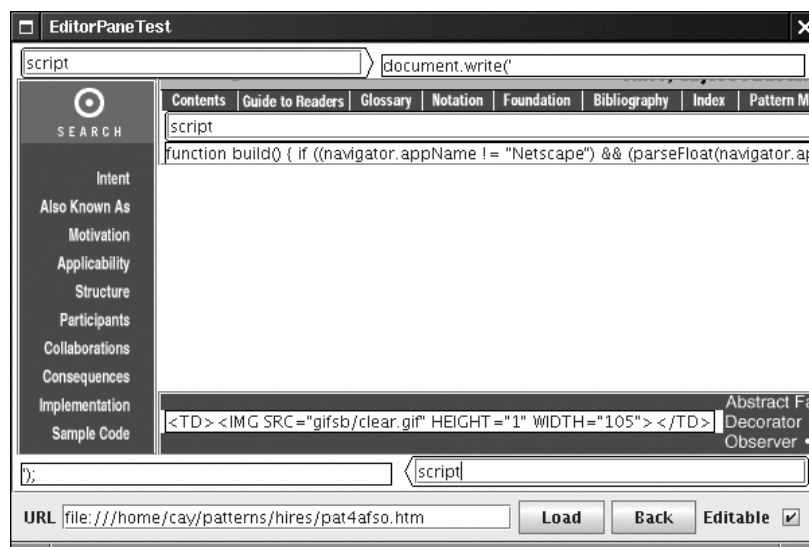
Si vous cliquez sur la case Modifiable, le panneau de l'éditeur devient modifiable : vous pouvez saisir du texte et l'effacer avec la touche de correction arrière. Ce composant comprend également les commandes `Ctrl+X`, `Ctrl+C` et `Ctrl+V` pour couper, copier et coller. Cependant, il faudrait une programmation assez importante pour ajouter la prise en charge des polices et de la mise en forme.

Lorsque le composant peut être modifié, les hyperliens ne sont pas actifs. De plus, vous pourrez voir les commandes JavaScript, les commentaires et d'autres étiquettes des pages Web lorsque le mode d'édition est activé (voir Figure 5.40). Ce programme d'exemple vous permet d'analyser le comportement de l'édition, mais nous vous recommandons de ne pas inclure cette option dans vos programmes.

ASTUCE

Par défaut, le `JEditorPane` est en mode édition. Il faut donc appeler `editorPane.setEditable(false)` pour changer de mode.

Figure 5.40
Le panneau de l'éditeur est en mode édition.



Les caractéristiques du panneau de l'éditeur que vous venez de voir dans le programme d'exemple sont simples à utiliser. La méthode `setPage` sert à charger un nouveau document. Par exemple :

```
JEditorPane editorPane = new JEditorPane();
editorPane.setPage(url);
```

Cette méthode accepte en paramètre soit une chaîne, soit un objet `URL`. La classe `JEditorPane` étend la classe `JTextComponent`. Par conséquent, vous pouvez aussi appeler la méthode `setText`, qui affiche du texte simple.

ASTUCE

La documentation API n'indique pas clairement si `setPage` charge le nouveau document dans un thread séparé (ce qui est généralement souhaitable, `JEditorPane` n'étant pas un démon rapide). Toutefois, vous pouvez forcer le chargement d'un thread séparé avec l'incantation suivante :

```
AbstractDocument doc = (AbstractDocument)editorPane.getDocument();
doc.setAsynchronousLoadPriority(0);
```

Pour écouter les clics sur des hyperliens, vous pouvez ajouter un `HyperlinkListener`. L'interface `HyperlinkListener` possède une seule méthode, `hyperlinkUpdate`, qui est appelée lorsque l'utilisateur déplace la souris au-dessus d'un lien ou qu'il clique dessus. Cette méthode accepte un paramètre de type `HyperlinkEvent`.

Vous devrez appeler la méthode `getEventType` pour déterminer le type de l'événement courant. Il existe trois valeurs possibles en retour :

```
HyperlinkEvent.EventType.ACTIVATED
HyperlinkEvent.EventType.ENTERED
HyperlinkEvent.EventType.EXITED
```

396 *Au cœur de Java 2 - Fonctions avancées*

La première valeur indique que l'utilisateur a cliqué sur un hyperlien. Dans ce cas, vous serez typiquement amené à ouvrir un autre lien. Les deux dernières valeurs peuvent fournir une sorte de retour visuel, comme une indication, lorsque la souris de l'utilisateur s'arrête au-dessus du lien.

INFO

Il est extrêmement bizarre qu'il n'existe pas trois méthodes différentes pour gérer l'activation, l'entrée et la sortie dans l'interface `HyperlinkListener`.

La méthode `getURL` de la classe `HyperlinkEvent` renvoie l'URL d'un hyperlien. Par exemple, voici comment installer un écouteur d'hyperlien qui suit le lien activé par l'utilisateur :

```
editorPane.addHyperlinkListener(new
    HyperlinkListener()
    {
        public void hyperlinkUpdate(HyperlinkEvent event)
        {
            if (event.getEventType()
                == HyperlinkEvent.EventType.ACTIVATED)
            {
                try
                {
                    editorPane.setPage(event.getURL());
                }
                catch (IOException e)
                {
                    editorPane.setText("Exception : " + e);
                }
            }
        }
    });
```

Le gestionnaire d'événements se contente de récupérer l'URL et de mettre à jour le panneau de l'éditeur. La méthode `setPage` peut déclencher une exception `IOException`. Dans ce cas, nous affichons un message d'erreur en texte simple.

Le programme de l'Exemple 5.15 montre toutes les caractéristiques que vous serez amené à réunir pour un système d'aide HTML. Techniquement, le `JEditorPane` est encore plus complexe que les composants d'arbres ou de tableaux. Cependant, si vous n'avez pas besoin d'écrire un éditeur de texte ou un affichage de texte respectant un format particulier, cette complexité sera transparente à vos yeux.

Exemple 5.15 : *EditorPaneTest.java*

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
```

```

/**
 * Ce programme montre comment afficher des documents HTML
 * dans un volet d'édition.
 */
public class EditorPaneTest
{
    public static void main(String[] args)
    {
        JFrame frame = new EditorPaneFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc contient un volet d'édition, un champ de texte et un bouton
 * pour entrer une URL et charger un document et un bouton Précédente
 * pour renvoyer à un document précédemment chargé.
 */
class EditorPaneFrame extends JFrame
{
    public EditorPaneFrame()
    {
        setTitle("EditorPaneTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        final Stack<String> urlStack = new Stack<String>();
        final JEditorPane editorPane = new JEditorPane();
        final JTextField url = new JTextField(30);

        // définit un écouteur de lien hypertexte

        editorPane.setEditable(false);
        editorPane.addHyperlinkListener(new
            HyperlinkListener()
            {
                public void hyperlinkUpdate(HyperlinkEvent event)
                {
                    if (event.getEventType()
                        == HyperlinkEvent.EventType.ACTIVATED)
                    {
                        try
                        {
                            // se souvenir de l'URL pour le bouton Précédente
                            urlStack.push(event.getURL().toString());
                            // afficher l'URL dans le champ de texte
                            url.setText(event.getURL().toString());
                            editorPane.setPage(event.getURL());
                        }
                        catch (IOException e)
                        {
                            editorPane.setText("Exception : " + e);
                        }
                    }
                }
            }
        );

        // configure la case à cocher pour basculer en mode de modification

```

398 *Au cœur de Java 2 - Fonctions avancées*

```

final JCheckBox editable = new JCheckBox();
editable.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            editorPane.setEditable(editable.isSelected());
        }
    });

// configure le bouton de chargement pour charger une URL

ActionListener listener = new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            try
            {
                // se souvenir de l'URL pour le bouton Précédente
                urlStack.push(url.getText());
                editorPane.setPage(url.getText());
            }
            catch (IOException e)
            {
                editorPane.setText("Exception : " + e);
            }
        }
    };

JButton loadButton = new JButton("Charger");
loadButton.addActionListener(listener);
url.addActionListener(listener);

// configure le bouton Précédente et l'action du bouton

JButton backButton = new JButton("Précédente");
backButton.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            if (urlStack.size() <= 1) return;
            try
            {
                // récupère l'URL du bouton Précédente
                urlStack.pop();
                // affiche l'URL dans le champ de texte
                String urlString = urlStack.peek();
                url.setText(urlString);
                editorPane.setPage(urlString);
            }
            catch (IOException e)
            {
                editorPane.setText("Exception : " + e);
            }
        }
    });

```

```

        add(new JScrollPane(editorPane), BorderLayout.CENTER);

        // place tous les composants de contrôle dans un panneau

        JPanel panel = new JPanel();
        panel.add(new JLabel("URL"));
        panel.add(url);
        panel.add(loadButton);
        panel.add(backButton);
        panel.add(new JLabel("Modifiable"));
        panel.add(editable);

        add(panel, BorderLayout.SOUTH);
    }

    private static final int DEFAULT_WIDTH = 600;
    private static final int DEFAULT_HEIGHT = 400;
}

```

javax.swing.JEditorPane 1.2

- void setPage(URL url)
Charge la page à l'url spécifiée et la place dans le panneau de l'éditeur.
- void addHyperlinkListener(HyperlinkListener listener)
Ajoute un écouteur d'hyperlien dans ce panneau d'éditeur.

javax.swing.event.HyperlinkListener 1.2

- void hyperlinkUpdate(HyperlinkEvent event)
Cette méthode est appelée lorsqu'un hyperlien est sélectionné.

javax.swing.HyperlinkEvent 1.2

- URL getURL()
Renvoie l'URL de l'hyperlien sélectionné.

Indicateurs de progression

Nous verrons, dans les prochaines sections, trois classes permettant de signaler la progression d'une activité lente. `JProgressBar` est un composant Swing qui indique une progression. `ProgressMonitor` est une boîte de dialogue qui contient une barre de progression. `ProgressMonitorInputStream` affiche une boîte de dialogue de contrôle de la progression lors de la lecture du flux.

Barres de progression

Une *barre de progression* est un composant simple : un rectangle partiellement rempli de couleur et signalant la progression d'une opération. Par défaut, la progression est signalée par une chaîne "*n*%". La Figure 5.41 en présente un exemple.

Une barre de progression se construit comme un bouton glisseur ; vous indiquez les valeurs minimale et maximale ainsi qu'une orientation optionnelle :

```

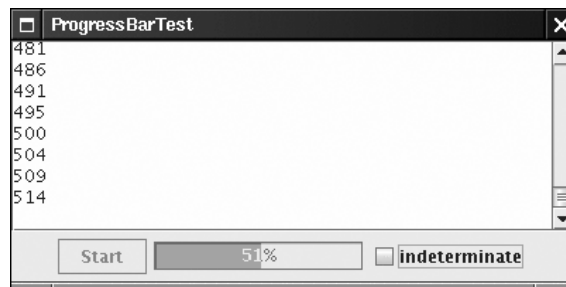
progressBar = new JProgressBar(0, 1000);
progressBar = new JProgressBar(SwingConstants.VERTICAL, 0, 1000);

```

400 Au cœur de Java 2 - Fonctions avancées

Figure 5.41

Une barre
de progression.



Vous pouvez également définir les valeurs minimale et maximale à l'aide des méthodes `setMinimum` et `setMaximum`.

A la différence d'un bouton glisseur, la barre de progression ne peut pas être réglée par l'utilisateur. Votre programme doit appeler `setValue` pour l'actualiser.

Si vous appelez

```
progressBar.setStringPainted(true);
```

la barre de progression calcule le pourcentage réalisé et affiche une chaîne du type "`n%`". Pour afficher une autre chaîne, fournissez-la avec la méthode `setString` :

```
if (progressBar.getValue() > 900)
    progressBar.setString("C'est presque terminé ");
```

Le programme de l'Exemple 5.16 montre la barre de progression d'une activité longue dans le temps.

La classe `SimulatedActivity` implémente un thread qui incrémente une valeur `current` dix fois par seconde. Lorsqu'il atteint une valeur cible, le thread se termine. Pour mettre fin au thread avant qu'il n'atteigne la cible, vous devez l'interrompre.

```
class SimulatedActivity implements Runnable
{
    ...
    public void run()
    {
        try
        {
            while (current < target && !interrupted())
            {
                Thread.sleep(100);
                current++;
            }
        } catch (InterruptedException e)
        {
        }
    }

    int current;
    int target;
}
```

Lorsque vous cliquez sur le bouton "Démarrer", un nouveau thread `SimulatedActivity` est lancé. La mise à jour de la barre de progression semblerait simple, le thread de l'activité simulée appelle la méthode `setValue`. Mais cela n'est pas sans risque : rappelez-vous que vous ne devez appeler les méthodes Swing qu'à partir d'un thread de distribution d'événements. Dans la pratique, l'approche évidente est également irréaliste : en général, un thread travailleur n'est pas averti de l'existence de la barre de progression. Le programme d'exemple montre comment lancer un minuteur qui interroge le thread à intervalles réguliers pour connaître la situation de la progression et actualiser la barre.

ATTENTION

N'oubliez pas qu'un thread travailleur ne peut pas définir directement la valeur de la barre de progression. Il doit utiliser la méthode `SwingUtilities.invokeLater` pour définir cette valeur dans le thread de distribution des événements.

Souvenez-vous qu'un minuteur Swing appelle la méthode `actionPerformed` de ses écouteurs et que ces appels surviennent dans le thread de distribution des événements. Vous pouvez donc actualiser les composants Swing en toute sécurité dans le rappel du minuteur. Voici celui du programme d'exemple. La valeur actuelle de l'activité simulée est affichée dans la zone de texte et dans la barre de progression. Si la fin de la simulation est atteinte, le minuteur s'arrête et le bouton "Démarrer" est réactivé.

```
public void actionPerformed(ActionEvent event)
{
    int current = activity.getCurrent();
    // Afficher la progression

    textArea.append(current + "\n");
    progressBar.setValue(current);

    //Vérifier si la tâche est terminée
    if (current == activity.getTarget())
    {
        activityMonitor.stop();
        startButton.setEnabled(true);
    }
}
```

Le JDK 1.4 prend également en charge une barre de progression *indéterminée* qui affiche une animation de progression, sans en préciser le pourcentage. C'est ce que vous trouvez dans les navigateurs : vous savez que le navigateur attend le serveur mais qu'il n'a aucune idée du temps que cela prendra. Pour afficher cette animation, appelez la méthode `setIndeterminate`.

L'Exemple 5.16 montre la totalité du programme.

Exemple 5.16 : ProgressBarTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.Timer;
```


402 *Au cœur de Java 2 - Fonctions avancées*

```
/**
 * Ce programme montre l'utilisation d'une barre de
 * progression pour contrôler l'évolution d'un thread.
 */
public class ProgressBarTest
{
    public static void main(String [] args)
    {
        JFrame frame = new ProgressBarFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Un cadre qui contient un bouton pour lancer une activité
 * simulée, une barre de progression et une zone de texte pour
 * le résultat.
 */
class ProgressBarFrame extends JFrame
{
    public ProgressBarFrame()
    {
        setTitle("ProgressBarTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // cette zone de texte contient le résultat de l'activité
        textArea = new JTextArea();

        //configure le cadre avec un bouton et une barre de progression

        JPanel panel = new JPanel();
        startButton = new JButton("Démarrer");
        progressBar = new JProgressBar();
        progressBar.setStringPainted(true);
        panel.add(startButton);
        panel.add(progressBar);

        checkBox = new JCheckBox("indéterminé");
        checkBox.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    progressBar.setIndeterminate(checkBox.isSelected());
                }
            });
        panel.add(checkBox);
        add(new JScrollPane(textArea), BorderLayout.CENTER);
        add(panel, BorderLayout.SOUTH);

        // configurer l'action du bouton

        startButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    progressBar.setMaximum(1000);
                }
            });
    }
}
```

```

        activity = new SimulatedActivity(1000);
        new Thread(activity).start();
        activityMonitor.start();
        startButton.setEnabled(false);
    }
});

// configurer l'action du minuteur
activityMonitor = new Timer(500, new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            int current = activity.getCurrent();

            // afficher la progression
            textArea.append(current + "\n");
            progressBar.setStringPainted(!progressBar.isIndeterminate());
            progressBar.setValue(current);

            // vérifier si la tâche est terminée
            if (current == activity.getTarget())
            {
                activityMonitor.stop();
                startButton.setEnabled(true);
            }
        }
    });
}

private Timer activityMonitor;
private JButton startButton;
private JProgressBar progressBar;
private JCheckBox checkBox;
private JTextArea textArea;
private SimulatedActivity activity;

public static final int DEFAULT_WIDTH = 400;
public static final int DEFAULT_HEIGHT = 200;
}

/**
 * Un exécutable d'activité simulée.
 */
class SimulatedActivity implements Runnable
{
    /**
     * Construit l'objet de thread de l'activité simulée. Le
     * thread incrémente un compteur de 0 jusqu'à la cible donnée.
     * @param t la valeur cible du compteur.
     */
    public SimulatedActivity(int t)
    {
        current = 0;
        target = t;
    }
}

```

404 Au cœur de Java 2 - Fonctions avancées

```

public int getTarget()
{
    return target;
}

public int getCurrent()
{
    return current;
}

public void run()
{
    try
    {
        while (current < target)
        {
            Thread.sleep(100);
            current++;
        }
    }
    catch (InterruptedException e)
    {
    }
}

private volatile int current;
private int target;
}

```

Contrôleurs de progression

Une barre de progression est un composant simple que l'on peut insérer dans une fenêtre. En revanche, un `ProgressMonitor` est une boîte de dialogue complète qui contient une barre de progression (voir Figure 5.42). La boîte de dialogue contient un bouton "Annuler". Si vous cliquez dessus, la boîte se ferme. En outre, votre programme peut demander si l'utilisateur a annulé la boîte de dialogue et mettre fin à l'action contrôlée (sachez que le nom de classe ne commence pas par un "J").

Pour construire un contrôleur de progression, fournissez les éléments suivants :

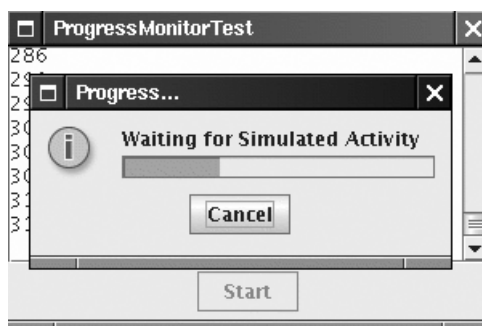
- le composant parent sur lequel doit s'afficher la boîte de dialogue ;
- un objet (une chaîne, une icône ou un composant) qui s'affiche sur la boîte de dialogue ;
- une note optionnelle à afficher sous l'objet ;
- les valeurs minimale et maximale.

Le contrôleur de progression ne peut toutefois pas mesurer la progression ni annuler une activité de lui-même. Il faut toujours définir la valeur de progression à intervalles réguliers en appelant la méthode `setProgress` (l'équivalent de la méthode `setValue` de la classe `JProgressBar`). En actualisant la valeur de la progression, appelez également la méthode `isCanceled` pour voir si l'utilisateur a cliqué sur le bouton "Annuler".

A la fin de l'activité contrôlée, appelez la méthode `close` pour fermer la boîte de dialogue. Vous pouvez réutiliser cette boîte en appelant à nouveau `start`.

Figure 5.42

Une boîte de dialogue pour le contrôle de la progression.



Le programme d'exemple semble très semblable à celui de la section précédente. Il nous faut toujours lancer un minuteur pour surveiller la progression de l'activité simulée et mettre à jour le contrôleur de progression. Voici le rappel du minuteur.

```
public void actionPerformed(ActionEvent event)
{
    int current = activity.getCurrent();

    // afficher la progression
    textArea.append(current + "\n");
    progressDialog.setProgress(current);

    // vérifier si la tâche est terminée ou annulée
    if (current == activity.getTarget() || progressDialog.isCanceled())
    {
        activityMonitor.stop();
        progressDialog.close();
        activity.interrupt();
        startButton.setEnabled(true);
    }
}
```

Il y a deux conditions à la fin de l'activité : elle s'est terminée ou l'utilisateur l'a annulée. Dans les deux cas, nous fermons :

- le minuteur qui a contrôlé l'activité ;
- la boîte de dialogue de progression ;
- l'activité elle-même (par interruption du thread).

Si vous exécutez le programme de l'Exemple 5.17, vous observerez une caractéristique intéressante de la boîte de dialogue. Elle ne s'affiche pas immédiatement, elle attend quelques secondes pour voir si l'activité s'est déjà terminée ou risque de se terminer très rapidement.

Pour contrôler le minutage, procédez comme suit. Utilisez la méthode `setMillisToDecideToPopup` pour définir le nombre de millisecondes à patienter entre la construction de la boîte de dialogue et son affichage potentiel. La valeur par défaut est de 500 millisecondes. `setMillisToPopup` correspond à votre estimation du temps avant l'apparition de la boîte de dialogue. Les concepteurs de Swing l'ont réglée à 2 secondes par défaut. À l'évidence, ils ont tenu compte du fait que les boîtes de dialogue ne s'affichent pas toujours comme nous le souhaitons. Il est déconseillé de modifier cette valeur.

406 *Au cœur de Java 2 - Fonctions avancées*

L'Exemple 5.17 montre le contrôleur de progression, qui mesure une fois de plus la progression d'une activité simulée. Comme vous le voyez, il est plus commode à utiliser et exige simplement que vous interrogiez le thread à surveiller à intervalles réguliers.

Exemple 5.17 : ProgressMonitorTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.Timer;

/**
 * Un programme qui teste un contrôleur de progression.
 */
public class ProgressMonitorTest
{
    public static void main(String [] args)
    {
        JFrame frame = new ProgressMonitorFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Un cadre qui contient un bouton pour lancer une activité
 * simulée et une zone de texte pour le résultat de l'activité.
 */
class ProgressMonitorFrame extends JFrame
{
    public ProgressMonitorFrame()
    {
        setTitle("ProgressMonitorTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // cette zone de texte contient le résultat de l'activité
        textArea = new JTextArea();

        // configurer un cadre de boutons
        JPanel panel = new JPanel();
        startButton = new JButton("Démarrer");
        panel.add(startButton);

        add(new JScrollPane(textArea), BorderLayout.CENTER);
        add(panel, BorderLayout.SOUTH);

        // configurer l'action du bouton
        startButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    // démarrer l'activité
                    activity = new SimulatedActivity(1000);
                    activityThread = new Thread(activity);
                }
            }
        );
    }
}
```

```

        activityThread.start();

        // lancer la boîte de dialogue de progression
        progressDialog = new
            ProgressMonitor(ProgressMonitorFrame.this,
                "En attente de l'activité simulée", null, 0,
                activity.getTarget());

        // démarrer le minuteur
        activityMonitor.start();

        startButton.setEnabled(false);
    }
});

// configurer l'action du minuteur
activityMonitor = new Timer(500, new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            int current = activity.getCurrent();

            // afficher la progression
            textArea.append(current + "\n");
            progressDialog.setProgress(current);

            // vérifier si la tâche est terminée ou annulée
            if (current == activity.getTarget()
                || progressDialog.isCanceled())
            {
                activityMonitor.stop();
                progressDialog.close();
                activityThread.interrupt();
                startButton.setEnabled(true);
            }
        }
    });
}

private Timer activityMonitor;
private JButton startButton;
private ProgressMonitor progressDialog;
private JTextArea textArea;
private Thread activityThread;
private SimulatedActivity activity;

public static final int DEFAULT_WIDTH = 300;
public static final int DEFAULT_HEIGHT = 200;
}

/**
 * Un exécutable d'activité simulée.
 */
class SimulatedActivity implements Runnable
{
    /**

```

408 *Au cœur de Java 2 - Fonctions avancées*

```
        Construit l'objet thread de l'activité simulée. Le
        thread incrémente un compteur de 0 à une cible donnée.
        @param t La valeur cible du compteur.
    */
    public SimulatedActivity(int t)
    {
        current = 0;
        target = t;
    }

    public int getTarget()
    {
        return target;
    }

    public int getCurrent()
    {
        return current;
    }

    public void run()
    {
        try
        {
            while (current < target)
            {
                Thread.sleep(100);
                current++;
            }
        }
        catch (InterruptedException e)
        {
        }
    }

    private volatile int current;
    private int target;
}
```

Surveiller la progression des flux d'entrée

Le package Swing contient un filtre de flux utile, appelé `ProgressMonitorInputStream`, qui affiche automatiquement une boîte de dialogue surveillant la quantité du flux déjà lue. Ce filtre est très facile à utiliser. Vous intégrez le `ProgressMonitorInputStream` dans votre suite usuelle de flux filtrés (voir Chapitre 12 du Volume 1 pour en savoir plus sur les flux).

Supposons par exemple que vous lisiez du texte d'un fichier. Commencez avec un `FileInputStream` :

```
FileInputStream in = new FileInputStream(f);
```

Vous devriez normalement le convertir en `InputStreamReader` :

```
InputStreamReader reader = new InputStreamReader(in);
```

Or, pour surveiller le flux, transformez d'abord le flux d'entrée du fichier en flux avec contrôleur de progression :

```
ProgressMonitorInputStream progressIn = new ProgressMonitorInputStream(parent,
caption, in);
```

Vous fournissez le composant parent, une légende et, bien entendu, le flux à surveiller. La méthode `read` du flux de contrôle de la progression passe les octets et actualise la boîte de dialogue de progression.

Construisez ensuite votre suite de filtres :

```
InputStreamReader reader = new InputStreamReader(progressIn);
```

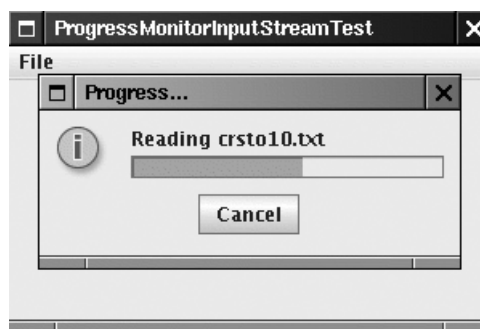
Et voilà ! A la lecture du fichier, le contrôleur de progression s'affiche automatiquement (voir Figure 4.43). C'est une très bonne application du filtrage de flux.

ATTENTION

Le flux de contrôle de la progression utilise la méthode `available` de la classe `InputStream` pour connaître le nombre total d'octets dans le flux. Or la méthode `available` ne signale que le nombre d'octets du flux qui sont disponibles *sans entraîner de blocage*. Le contrôleur de progression fonctionne bien pour les fichiers et les URL HTTP car leur longueur est connue d'avance, mais ce n'est pas le cas avec tous les flux.

Figure 5.43

Un contrôleur de progression pour un flux d'entrée.



Le programme de l'Exemple 5.18 compte le nombre de lignes dans un fichier. Si vous lisez un grand fichier (comme "Le comte de Monte-Christo" sur le CD), la boîte de dialogue de progression s'affiche.

Sachez que le programme ne remplit pas efficacement la zone de texte. Il serait plus rapide de lire le fichier dans un `StringBuffer`, puis de définir le texte de la zone de texte sur le contenu du tampon de chaîne. Toutefois, dans cet exemple de programme, nous apprécions cette approche plutôt lente, elle permet d'admirer la boîte de dialogue.

Pour éviter les tremblements, nous n'affichons pas la zone de texte lorsqu'elle se remplit.

Exemple 5.18 : `ProgressMonitorInputStreamTest.java`

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Un programme pour tester un flux d'entrée de contrôle de la
 * progression.
```


410 *Au cœur de Java 2 - Fonctions avancées*

```

*/
public class ProgressMonitorInputStreamTest
{
    public static void main(String [] args)
    {
        JFrame frame = new TextFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
    Un cadre contenant un menu pour charger un fichier texte et une zone
    de texte pour afficher son contenu. La zone est construite
    au chargement du fichier et définie comme volet conteneur du
    cadre à la fin du chargement. Ceci évite les tremblements
    au cours du chargement.
*/
class TextFrame extends JFrame
{
    public TextFrame()
    {
        setTitle("ProgressMonitorInputStreamTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // configurer le menu

        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);
        JMenu fileMenu = new JMenu("Fichier");
        menuBar.add(fileMenu);
        JMenuItem openItem = new JMenuItem("Ouvrir");
        openItem.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    try
                    {
                        openFile();
                    }
                    catch(IOException exception)
                    {
                        exception.printStackTrace();
                    }
                }
            });

        fileMenu.add(openItem);
        JMenuItem exitItem = new JMenuItem("Quitter");
        exitItem.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    System.exit(0);
                }
            });
        fileMenu.add(exitItem);
    }
}

```


412 Au cœur de Java 2 - Fonctions avancées

```

        public void run()
        {
            setContentPane(new JScrollPane(textArea));
            validate();
        }
    });

    Thread readThread = new Thread(readRunnable);
    readThread.start();
}

private JMenuItem openItem;
private JMenuItem exitItem;

public static final int DEFAULT_WIDTH = 300;
public static final int DEFAULT_HEIGHT = 200;
}

```

javax.swing.JProgressBar 1.2

- JProgressBar()
- JProgressBar(int direction)
- JProgressBar(int min, int max)
- JProgressBar(int direction, int min, int max)

Construisent un bouton glisseur avec la direction donnée, un minimum et un maximum.

Paramètres : direction SwingConstants.HORIZONTAL
ou SwingConstants.VERTICAL. La direction par défaut
est horizontale.

min, max Le minimum et le maximum pour les valeurs de la barre
de progression. Les valeurs par défaut sont 0 et 100.

- int getMinimum()
- int getMaximum()
- void setMinimum(int value)
- void setMaximum(int value)

Récupèrent et définissent les valeurs minimale et maximale.

- int getValue()
- void setValue(int value)

Récupèrent et définissent la valeur courante.

- String getString()
- void setString(String s)

Récupèrent et définissent la chaîne à afficher dans la barre de progression. Si la chaîne est nulle, une chaîne par défaut est affichée "n%".

- `boolean isStringPainted()`
- `void setStringPainted(boolean b)`

Récupèrent et définissent la propriété "StringPainted". Si elle vaut `true`, une chaîne est dessinée au-dessus de la barre de progression. La valeur par défaut est `false` (aucune chaîne n'est dessinée).

- `boolean isIndeterminate()` 1.4
- `void setIndeterminate(boolean b)` 1.4

Récupèrent et définissent une propriété "indeterminate". Si elle vaut `true`, la barre de progression se transforme en bloc qui avance ou recule, signalant une attente de durée indéterminée. La valeur par défaut est `false`.

javax.swing.ProgressMonitor 1.2

- `ProgressMonitor(Component parent, Object message, String note, int min, int max)`

Construit une boîte de dialogue avec un contrôleur de progression.

<i>Paramètres :</i>	<code>parent</code>	Le composant parent sur lequel apparaît la boîte de dialogue.
	<code>message</code>	L'objet de message à afficher dans la boîte de dialogue.
	<code>note</code>	La chaîne optionnelle à afficher sous le message. Si cette valeur vaut <code>null</code> , aucun espace n'est réservé à la note et un appel ultérieur à <code>setNote</code> n'aura aucun effet.
	<code>min, max</code>	Les valeurs minimale et maximale de la barre de progression.

- `void setNote(String note)`

Modifie le texte de la note.

- `void setProgress(int value)`

Définit la valeur de la barre de progression sur la valeur donnée.

- `void close()`

Ferme cette boîte de dialogue.

- `boolean isCanceled()`

Renvoie `true` si l'utilisateur a annulé cette boîte de dialogue.

javax.swing.ProgressMonitorInputStream 1.2

- `ProgressMonitorInputStream(Component parent, Object message, InputStream in)`

Construit un filtre de flux d'entrée associé à une boîte de dialogue de contrôle de la progression.

<i>Paramètres :</i>	<code>parent</code>	Le composant parent sur lequel s'affiche cette boîte de dialogue.
	<code>message</code>	L'objet de message à afficher dans la boîte de dialogue.
	<code>in</code>	Le flux d'entrée objet de la surveillance.

Organisateurs de composants

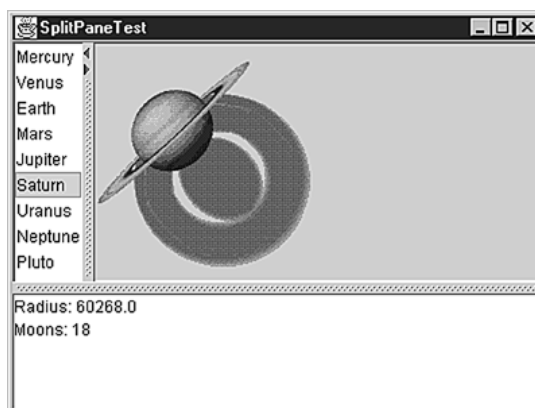
Nous terminerons notre étude des caractéristiques avancées de Swing par une présentation des composants qui aident à organiser d'autres composants. Nous parlerons notamment des *séparateurs*, un mécanisme permettant de séparer une zone en plusieurs parties dont les limites peuvent être ajustées, des *onglets*, qui se servent de tabulations pour basculer l'affichage de plusieurs panneaux, et des *panneaux de bureau*, qui peuvent être utilisés pour implémenter des applications affichant plusieurs *cadres internes*.

Séparateurs

Les séparateurs permettent de découper un composant en deux parties, séparées par une frontière ajustable. La Figure 5.44 montre une fenêtre comprenant deux séparateurs. Le premier panneau est séparé verticalement, avec une zone de texte en bas. La zone supérieure est à nouveau séparée en deux, horizontalement, avec une liste à gauche et une étiquette contenant une image sur la droite.

Figure 5.44

Une fenêtre comprenant deux séparateurs.



Pour construire un séparateur, il faut spécifier une orientation (`JSplitPane.HORIZONTAL_SPLIT` ou `JSplitPane.VERTICAL_SPLIT`), suivie des deux composants. Par exemple :

```
JSplitPane innerPane
    = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
        planetList, planetImage);
```

C'est tout ce qu'il faut faire. Si vous le souhaitez, vous pouvez ajouter des icônes d'agrandissement dans la barre de séparation. Ces icônes sont visibles dans le panneau supérieur de la Figure 5.44. Avec l'aspect Metal, ces icônes prennent la forme de petits triangles. Si vous cliquez sur l'un d'entre eux, le séparateur se déplace dans la direction vers laquelle pointe le triangle, jusqu'à la limite de la fenêtre.

Pour ajouter ces icônes, appelez

```
innerPane.setOneTouchExpandable(true);
```

La caractéristique de "mise en page continue" permet d'afficher en permanence le contenu des deux composants lorsque l'utilisateur ajuste le séparateur. Cette caractéristique peut paraître plus puissante, mais elle est parfois assez lente. Elle est activée par l'appel suivant :

```
innerPane.setContinuousLayout(true);
```

Dans notre programme d'exemple, nous avons conservé en bas le séparateur par défaut (c'est-à-dire sans mise en page continue). Lorsque vous déplacez ce séparateur, vous ne bougez qu'un cadre noir. Lorsque vous avez relâché le bouton de la souris, les composants sont redessinés.

Le programme simple de l'Exemple 5.19 remplit une fenêtre avec des données de planètes. Lorsque l'utilisateur effectue une sélection, l'image de la planète correspondante est affichée à droite et une description est ajoutée dans la zone de texte du bas. Lorsque vous exécuterez ce programme, vous pourrez ajuster les séparateurs et tester les caractéristiques d'agrandissement maximal et de mise en page continue.

Exemple 5.19 : *SplitPaneTest.java*

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Ce programme montre l'organisateur du composant séparateur.
 */
public class SplitPaneTest
{
    public static void main(String[] args)
    {
        JFrame frame = new SplitPaneFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc est constitué de deux volets imbriqués pour montrer
 * les images et les données des planètes.
 */
class SplitPaneFrame extends JFrame
{
    public SplitPaneFrame()
    {
        setTitle("SplitPaneTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // définit les composants pour les noms des planètes, les images et
        // les descriptions

        final JList planetList = new JList(planets);
        final JLabel planetImage = new JLabel();
        final JTextArea planetDescription = new JTextArea();

        planetList.addListSelectionListener(new
            ListSelectionListener()
            {
```

416 Au cœur de Java 2 - Fonctions avancées

```
        public void valueChanged(ListSelectionEvent event)
        {
            Planet value
                = (Planet)planetList.getSelectedValue();

            // met à jour l'image et la description

            planetImage.setIcon(value.getImage());
            planetDescription.setText(value.getDescription());
        }
    });

    // définit les séparateurs

    JSplitPane innerPane
        = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
            planetList, planetImage);

    innerPane.setContinuousLayout(true);
    innerPane.setOneTouchExpandable(true);

    JSplitPane outerPane
        = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
            innerPane, planetDescription);

    add(outerPane, BorderLayout.CENTER);
}

private Planet[] planets =
{
    new Planet("Mercure", 2440, 0),
    new Planet("Vénus", 6052, 0),
    new Planet("Terre", 6378, 1),
    new Planet("Mars", 3397, 2),
    new Planet("Jupiter", 71492, 16),
    new Planet("Saturne", 60268, 18),
    new Planet("Uranus", 25559, 17),
    new Planet("Neptune", 24766, 8),
    new Planet("Pluton", 1137, 1),
};

private static final int DEFAULT_WIDTH = 300;
private static final int DEFAULT_HEIGHT = 300;
}

/**
 * Décrit une planète.
 */
class Planet
{
    /**
     * Construit une planète.
     * @param n le nom de la planète
     * @param r le rayon de la planète
     * @param m le nombre de lunes
     */
    public Planet(String n, double r, int m)
    { name = n;

```

```

        radius = r;
        moons = m;
        image = new ImageIcon(name + ".gif");
    }

    public String toString()
    {
        return name;
    }

    /**
     * Obtient une description de la planète.
     * @return la description
     */
    public String getDescription()
    {
        return "Rayon: " + radius + "\nLunes: " + moons + "\n";
    }

    /**
     * Obtient une image de la planète.
     * @return l'image
     */
    public ImageIcon getImage()
    {
        return image;
    }

    private String name;
    private double radius;
    private int moons;
    private ImageIcon image;
}

```

javax.swing.JSplitPane 1.2

- JSplitPane()
- JSplitPane(int direction)
- JSplitPane(int direction, boolean continuousLayout)
- JSplitPane(int direction, Component first, Component second)
- JSplitPane(int direction, boolean continuousLayout, Component first, Component second)

Construisent un nouveau séparateur.

<i>Paramètres :</i>	direction	HORIZONTAL_SPLIT ou VERTICAL_SPLIT
	continuousLayout	vaut true si les composants sont mis à jour continuellement lorsque le séparateur est déplacé
	first, second	les composants à ajouter

- boolean isOneTouchExpandable()
- void setOneTouchExpandable(boolean b)

Renvoient ou définissent la propriété d'agrandissement maximal. Lorsque cette propriété est définie, le séparateur contient deux icônes pour étendre complètement l'un des deux composants.

- boolean isContinuousLayout()

418 Au cœur de Java 2 - Fonctions avancées

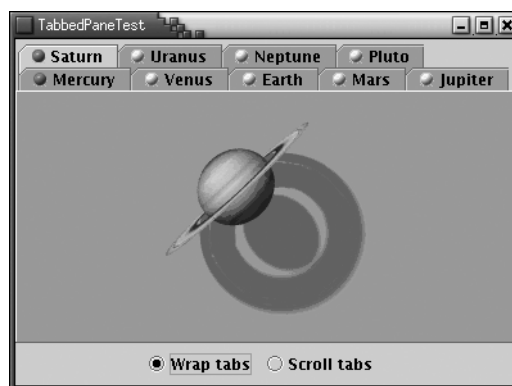
- `void setContinuousLayout(boolean b)`
Renvoient ou définissent la propriété de mise en page continue. Lorsque cette propriété est définie, les composants sont mis à jour en permanence lorsque le séparateur est déplacé.
- `void setLeftComponent(Component c)`
- `void setTopComponent(Component c)`
Ces opérations ont le même effet, elles choisissent `c` comme premier composant d'un panneau renfermant un séparateur.
- `void setRightComponent(Component c)`
- `void setBottomComponent(Component c)`
Ces opérations ont le même effet, elles choisissent `c` comme second composant d'un panneau renfermant un séparateur.

Onglets

Les onglets sont très courants dans les interfaces utilisateur, ils permettent de décomposer une boîte de dialogue complexe en plusieurs sous-ensembles d'options similaires. Ils permettent aussi à un utilisateur de basculer entre plusieurs documents ou images (voir Figure 5.45). C'est l'application que nous avons retenue pour notre programme d'exemple.

Figure 5.45

Un panneau à onglets.



Pour créer un panneau à onglets, il faut commencer par construire un objet `JTabbedPane`, pour y placer des onglets par la suite.

```
JTabbedPane tabbedPane = new JTabbedPane();
tabbedPane.addTab(title, icon, component);
```

Le dernier paramètre de la méthode `addTab` est du type `Component`. Pour ajouter plusieurs composants dans le même onglet, il faut d'abord les réunir dans un conteneur, comme `JPanel`.

L'icône est optionnelle. La méthode `addTab`, par exemple, ne nécessite aucune icône :

```
tabbedPane.addTab(title, component);
```

Vous pouvez aussi ajouter un onglet dans un ensemble d'onglets avec la méthode `insertTab` :

```
tabbedPane.insertTab(title, icon, component, tooltip, index);
```

Pour supprimer un onglet, il suffit d'utiliser :

```
tabPane.removeTabAt(index);
```

Lorsque vous ajoutez un nouvel onglet dans un ensemble d'onglets, le nouvel onglet n'est pas affiché automatiquement. Vous devez le sélectionner avec la méthode `setSelectedIndex`. Par exemple, voici comment afficher un onglet que vous venez d'ajouter :

```
tabbedPane.setSelectedIndex(tabbedPane.getTabCount() - 1);
```

Si vous disposez de nombreux onglets, ils peuvent prendre une assez grande quantité d'espace. Nouveauté dans la version 1.4 du JDK, vous pouvez afficher les onglets en mode de défilement, lorsqu'une seule ligne d'onglets est affichée, avec un ensemble de boutons fléchés permettant à l'utilisateur de faire défiler l'ensemble d'onglets (voir Figure 5.46).

Vous pouvez configurer la disposition des onglets en mode d'emballage ou de défilement, en appelant

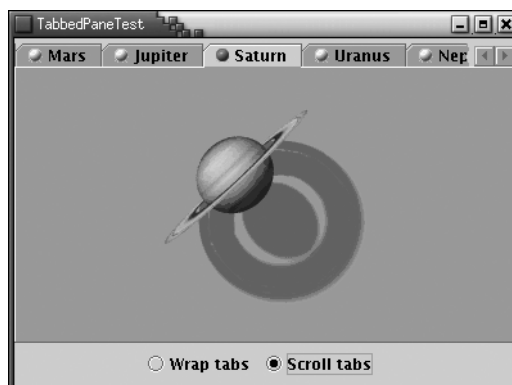
```
tabbedPane.setTabLayoutPolicy(JTabbedPane.WRAP_TAB_LAYOUT);
```

ou

```
tabbedPane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_LAYOUT);
```

Figure 5.46

Un panneau avec des onglets défilants.



Le programme d'exemple montre une technique pratique fondée sur des panneaux contenant des onglets. Il se peut que vous soyez amené à modifier un composant juste avant de l'afficher. Dans notre programme d'exemple, nous ne chargeons l'image d'une planète que lorsque l'utilisateur clique effectivement sur un onglet.

Pour être averti lorsque l'utilisateur clique sur un nouvel onglet, il convient d'installer un écouteur `ChangeListener` dans le panneau contenant les onglets. Cet écouteur doit être installé dans le panneau lui-même, et non dans ses composants.

```
tabbedPane.addChangeListener(listener);
```

Lorsque l'utilisateur sélectionne un onglet, la méthode `stateChanged` de l'écouteur de modifications est appelée. L'onglet est alors considéré comme la source de l'événement. Appelez la méthode `getSelectedIndex` pour déterminer quel panneau est en train d'être affiché.

```
public void stateChanged(ChangeEvent event)
{
    int n = tabbedPane.getSelectedIndex();
```

420 *Au cœur de Java 2 - Fonctions avancées*

```

        loadTab(n);
    }

```

Dans l'Exemple 5.20, nous commençons par mettre tous les composants d'onglets à null. Lorsqu'un nouvel onglet est sélectionné, nous comparons son composant à null. En cas d'égalité, nous le remplaçons par l'image. Cela se produit instantanément lorsque vous cliquez sur l'onglet. Vous ne verrez pas de panneau vide. Pour le plaisir, nous avons aussi transformé l'icône de balle jaune en balle rouge pour indiquer les panneaux qui ont déjà été parcourus.

Exemple 5.20 : *TabbedPaneTest.java*

```

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Ce programme montre l'organisateur du composant à onglets.
 */
public class TabbedPaneTest
{
    public static void main(String[] args)
    {
        JFrame frame = new TabbedPaneFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc montre un onglet et des boutons pour
 * basculer entre la disposition emballée et défilante.
 */
class TabbedPaneFrame extends JFrame
{
    public TabbedPaneFrame()
    {
        setTitle("TabbedPaneTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        tabbedPane = new JTabbedPane();
        /**
         * Nous mettons les composants à null et mettons en attente leur
         * chargement jusqu'à ce que leur onglet soit affiché pour la
         * première fois.
         */

        ImageIcon icon = new ImageIcon("yellow-ball.gif");

        tabbedPane.addTab("Mercure", icon, null);
        tabbedPane.addTab("Vénus", icon, null);
        tabbedPane.addTab("Terre", icon, null);
        tabbedPane.addTab("Mars", icon, null);
        tabbedPane.addTab("Jupiter", icon, null);
        tabbedPane.addTab("Saturne", icon, null);
        tabbedPane.addTab("Uranus", icon, null);
        tabbedPane.addTab("Neptune", icon, null);
    }
}

```

```

tabbedPane.addTab("Pluton", icon, null);
add(tabbedPane, "Center");
tabbedPane.addChangeListener(new
    ChangeListener()
    {
        public void stateChanged(ChangeEvent event)
        {
            // regarde si l'onglet possède encore un composant nul

            if (tabbedPane.getSelectedComponent() == null)
            {
                // règle le composant sur l'icône de l'image

                int n = tabbedPane.getSelectedIndex();
                loadTab(n);
            }
        }
    });
loadTab(0);

JPanel buttonPanel = new JPanel();
ButtonGroup buttonGroup = new ButtonGroup();
JRadioButton wrapButton = new JRadioButton("Emballer les onglets");
wrapButton.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            tabbedPane.setTabLayoutPolicy(
                JTabbedPane.WRAP_TAB_LAYOUT);
        }
    });
buttonPanel.add(wrapButton);
buttonGroup.add(wrapButton);
wrapButton.setSelected(true);
JRadioButton scrollButton
    = new JRadioButton("Fait défiler les onglets");
scrollButton.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            tabbedPane.setTabLayoutPolicy(
                JTabbedPane.SCROLL_TAB_LAYOUT);
        }
    });
buttonPanel.add(scrollButton);
buttonGroup.add(scrollButton);
add(buttonPanel, BorderLayout.SOUTH);
}

/**
 * Charge l'onglet avec l'indice donné.
 * @param n L'indice de l'onglet à charger
 */

```

- JTabbedPane()
- JTabbedPane(int placement)
Construisent un onglet.

Paramètres : placement SwingConstants.TOP, SwingConstants.LEFT,
 SwingConstants.RIGHT ou SwingConstants.BOTTOM

- void addTab(String title, Component c)
- void addTab(String title, Icon icon, Component c)
- void addTab(String title, Icon icon, Component c, String tooltip)

Ajoutent un onglet à la fin d'un panneau.

- void insertTab(String title, Icon icon, Component c, String tooltip, int index)

Insère un onglet dans un panneau à onglets à l'indice spécifié.

- void removeTabAt(int index)

Supprime l'onglet à l'indice spécifié.

- void setSelectedIndex(int index)

Sélectionne un onglet à l'indice spécifié.

- int getSelectedIndex()

Renvoie l'indice d'un onglet spécifié.

- Component getSelectedComponent()

Renvoie le composant de l'onglet sélectionné.

- String getTitleAt(int index)
- void setTitleAt(int index, String title)
- Icon getIconAt(int index)
- void setIconAt(int index, Icon icon)
- Component.getComponentAt(int index)
- void setComponentAt(int index, Component c)

Renvoient ou définissent le titre, l'icône ou le composant à l'indice spécifié.

- int indexOfTab(Icon icon)

- La plupart des applications présentent des informations dans plusieurs fenêtres contenues dans une seule fenêtre principale. Si vous minimisez cette fenêtre principale, toutes ses fenêtres internes seront cachées en même temps. Avec l'environnement de Windows, cette interface utilisateur est parfois appelée une interface à plusieurs documents, ou MDI (*Multiple Document Interface*). La Figure 5.47 montre un exemple d'application typique utilisant cette interface.

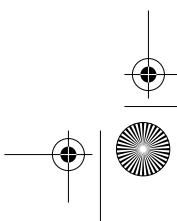


Figure 5.47
Une application possédant une interface à plusieurs documents.

424 Au cœur de Java 2 - Fonctions avancées

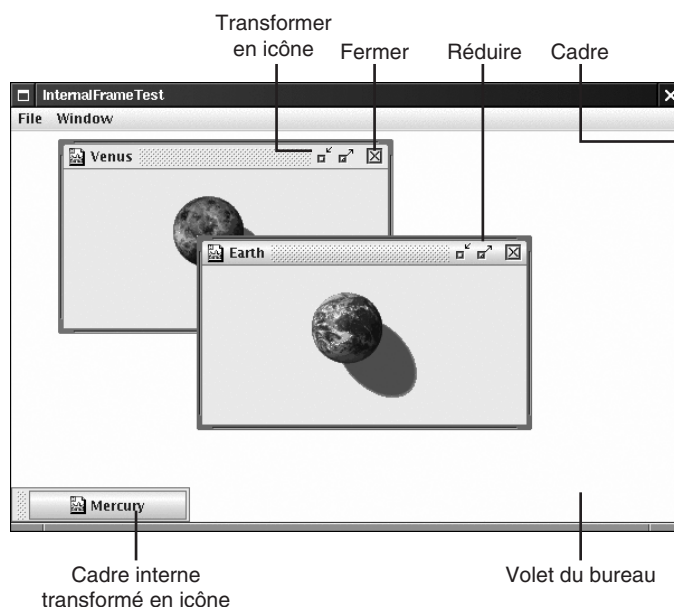
Le style de l'interface utilisateur de cet exemple est populaire, mais il est devenu de moins en moins fréquent au cours de ces dernières années. Aujourd'hui, de nombreuses applications affichent simplement une fenêtre séparée pour chaque document. Laquelle des deux approches est la meilleure ? MDI réduit le désordre existant dans vos fenêtres. En revanche, si vous avez plusieurs fenêtres principales, vous pourrez vous servir des raccourcis clavier pour passer de l'une à l'autre.

Dans l'univers de Java, où vous ne pouvez jamais savoir sur quel système d'exploitation votre application fonctionnera, il est très intéressant que la gestion des fenêtres revienne à votre application elle-même.

La Figure 5.48 montre une application Java possédant trois fenêtres internes. Deux d'entre elles possèdent des icônes pour les réduire et les agrandir. La troisième fenêtre est déjà icônifiée.

Figure 5.48

Une application Java possédant trois fenêtres internes.



Avec l'aspect Metal, les fenêtres internes contiennent une zone spéciale appelée *grabber*, qui permet à l'utilisateur de les déplacer. La taille des fenêtres peut être modifiée en déplaçant ses coins.

Pour obtenir toutes ces caractéristiques, il convient de suivre les étapes suivantes :

1. Placez l'application dans une fenêtre `JFrame`.
2. Ajoutez le `JDesktopPane` au `JFrame`.

```
desktop = new JDesktopPane();
add(desktop, BorderLayout.CENTER);
```

3. Construisez les fenêtres `JInternalFrame`. Vous pouvez spécifier des icônes pour modifier la taille de la fenêtre ou pour la fermer. Normalement, vous choisirez toutes ces icônes.

```
JInternalFrame iframe = new JInternalFrame(title,
    true, // la taille peut être modifiée
    true, // la fenêtre peut être fermée
```

```
true, // la fenêtre peut être agrandie au maximum
true); // la fenêtre peut être icônifiée
```

4. Ajoutez les composants à la fenêtre.

```
iframe.add(c, BorderLayout.CENTER);
```

5. Définissez une icône pour la fenêtre. Cette icône apparaît dans le coin supérieur gauche de la fenêtre.

```
iframe.setFrameIcon(icon);
```

INFO

Dans la version actuelle de l'aspect Metal, les icônes des fenêtres ne sont pas affichées dans les fenêtres icônifiées.

6. Spécifiez la taille de la fenêtre interne. Comme pour les fenêtres normales, les fenêtres internes possèdent au départ une taille de 0 sur 0. Comme vous voudrez sûrement éviter que toutes les fenêtres internes s'affichent les unes sur les autres, il faudra avoir recours à une position variable pour la prochaine fenêtre. Utilisez la méthode `reshape` pour définir à la fois la position et la taille de la fenêtre :

```
iframe.reshape(nextFrameX, nextFrameY, width, height);
```

7. Comme avec `JFrames`, il faudra rendre la fenêtre visible.

```
iframe.setVisible(true);
```

INFO

Dans les versions précédentes de Swing, les fenêtres internes étaient visibles par défaut et cet appel n'était pas nécessaire.

8. Ajoutez la fenêtre au `JDesktopPane` :

```
desktop.add(iframe);
```

9. Vous voudrez probablement que cette nouvelle fenêtre soit la *fenêtre sélectionnée*. Parmi les fenêtres internes du bureau, seules les fenêtres sélectionnées reçoivent le focus du clavier. Avec l'aspect Metal, la fenêtre sélectionnée possède une barre de titre bleue, alors que les autres fenêtres ont une barre de titre grise. La méthode `setSelected` permet de sélectionner une fenêtre. Cependant, la propriété "sélectionnée" peut être refusée lorsque la fenêtre qui est couramment sélectionnée refuse d'abandonner le focus. Dans ce cas, la méthode `setSelected` déclenche une `PropertyVetoException` qu'il vous faudra gérer.

```
try
{
    iframe.setSelected(true);
}
catch (PropertyVetoException e)
{
    // la tentative a été refusée
}
```


426 *Au cœur de Java 2 - Fonctions avancées*

10. De plus, vous voudrez probablement changer la position de la prochaine fenêtre interne, de sorte qu'elle ne recouvre pas la fenêtre courante. Une bonne distance entre deux fenêtres consécutives correspond à la hauteur de la barre de titre, que vous pouvez récupérer de la manière suivante :

```
int frameDistance =
    iframe.getHeight() - iframe.getContentPane().getHeight()
```

11. Utilisez cette distance pour déterminer la position de la prochaine fenêtre interne.

```
nextFrameX += frameDistance;
nextFrameY += frameDistance;
if (nextFrameX + width > desktop.getWidth())
    nextFrameX = 0;
if (nextFrameY + height > desktop.getHeight())
    nextFrameY = 0;
```

Présentation des fenêtres : cascade et juxtaposition

Sous Windows, il existe des commandes standard pour arranger les fenêtres en cascade ou les juxtaposer (voir Figures 5.49 et 5.50). Les classes `JDesktopPane` et `JInternalFrame` ne possèdent aucun support intégré pour ces opérations. Dans l'Exemple 5.21, nous vous montrons comment implémenter ces opérations par vous-même.

Pour mettre toutes les fenêtres en cascade, il faut modifier la taille des fenêtres et leur position. La méthode `getAllFrames` de la classe `JDesktopPane` renvoie un tableau de toutes les fenêtres internes.

```
JInternalFrame[] frames = desktop.getAllFrames();
```

Cependant, il faut faire attention à l'état de chaque fenêtre. Une fenêtre interne peut être dans trois états distincts :

- icônifiée ;
- de taille variable ;
- maximisée.

La méthode `isIcon` permet de déterminer quelles fenêtres internes sont icônifiées et ne doivent par conséquent pas être traitées. Cependant, si une fenêtre est maximisée, il convient de lui donner une taille variable en appelant `setMaximum(false)`. C'est une autre propriété qui peut être refusée. Par conséquent, il faut intercepter l'exception `PropertyVetoException`.

La boucle suivante organise en cascade toutes les fenêtres internes :

```
for (JInternalFrame frame : desktop.getAllFrames())
{
    if (!frame.isIcon())
    {
        try
        {
            // essaye de modifier la taille des fenêtres maximisées.
            // Cela peut être refusé
            frame.setMaximum(false);
            frame.reshape(x, y, width, height);
            x += frameDistance;
            y += frameDistance;
            // emballe au bord du bureau
            if (x + width > desktop.getWidth()) x = 0;
            if (y + height > desktop.getHeight()) y = 0;
        }
        catch (PropertyVetoException e)
        {
            // Fenêtre déjà maximisée ou autre problème
        }
    }
}
```

```

    }
    catch (PropertyVetoException e)
    {}
}

```

Figure 5.49

Des fenêtres internes organisées en cascade.

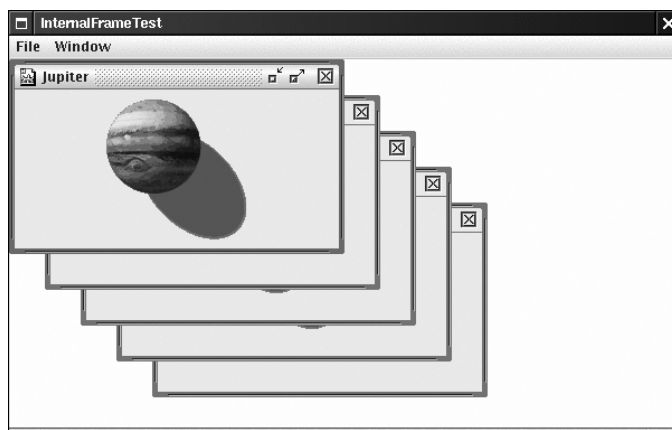
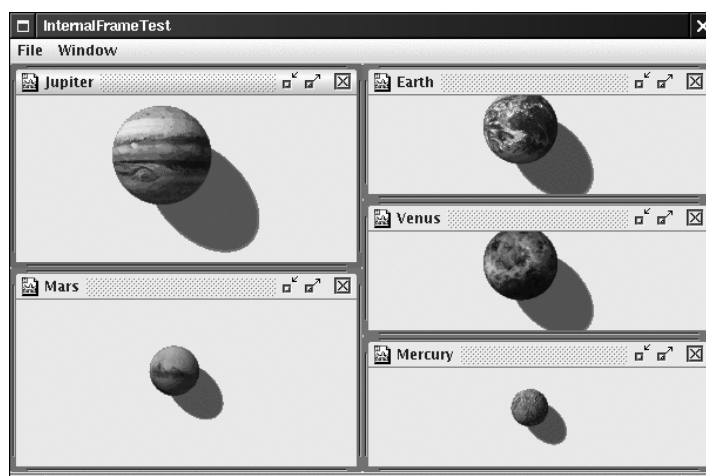


Figure 5.50

Des fenêtres internes juxtaposées.



La juxtaposition des fenêtres internes est plus délicate, particulièrement si le nombre de fenêtres n'est pas un carré parfait. Tout d'abord, comptez le nombre de fenêtres qui ne sont pas icônifiées. Puis, calculez le nombre de lignes, comme ceci :

```
int rows = (int)Math.sqrt(frameCount);
```

Le nombre de colonnes est alors :

```
int cols = frameCount / rows;
```

Sauf que la dernière ligne possède cols+1 colonnes :

```
int extra = frameCount % rows
```

428 Au cœur de Java 2 - Fonctions avancées

Voici la boucle permettant de juxtaposer toutes les fenêtres internes.

```
int width = desktop.getWidth() / cols;
int height = desktop.getHeight() / rows;
int r = 0;
int c = 0;
for (JInternalFrame frame : desktop.getAllFrames())
{
    if (!frame.isIcon())
    {
        try
        {
            frame.setMaximum(false);
            frame.reshape(c * width,
                r * height, width, height);
            r++;
            if (r == rows)
            {
                r = 0;
                c++;
                if (c == cols - extra)
                {
                    // commence à ajouter une ligne supplémentaire
                    rows++;
                    height = desktop.getHeight() / rows;
                }
            }
        }
        catch (PropertyVetoException e)
        {}
    }
}
```

Le programme d'exemple montre une autre opération courante sur les fenêtres : sélectionner la fenêtre suivante non icônifiée. La classe `JDesktopPane` ne possède aucune méthode pour renvoyer la fenêtre sélectionnée. Il faut donc parcourir toutes les fenêtres et appeler `isSelected` jusqu'à ce que la fenêtre sélectionnée soit trouvée. Puis, il faut chercher la fenêtre suivante non icônifiée, et essayer de la sélectionner en appelant

```
frames[next].setSelected(true);
```

Comme auparavant, cette méthode peut déclencher une `PropertyVetoException`, auquel cas il faut continuer à chercher. Si vous revenez à la fenêtre d'origine, cela signifie qu'aucune autre fenêtre ne peut être sélectionnée. Voici le code complet de la boucle :

```
JInternalFrame[] frames == desktop.getAllFrames();
for (int i = 0; i < frames.length; i++)
{
    if (frames[i].isSelected())
    {
        // cherche la prochaine fenêtre non icônifiée qui peut être
        // sélectionnée
        int next = (i + 1) % frames.length;
        while (next != i)
        {
            if (!frames[next].isIcon())
            {
                try
                {

```

```

        // toutes les autres fenêtres sont des icônes
        // ou sélection d'annulation
        frames[next].setSelected(true);
        frames[next].toFront();
        frames[i].toBack();
        return;
    }
    catch (PropertyVetoException e)
    {}
}
next = (next + 1) % frames.length;
}
}
}

```

Refuser des paramètres de propriétés

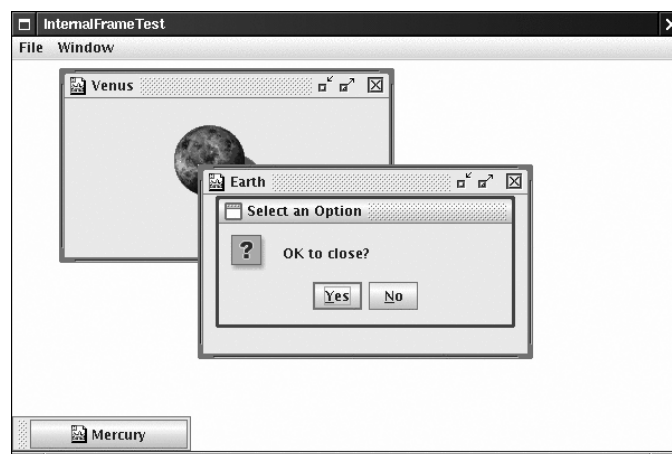
Maintenant que vous connaissez toutes ces exceptions de refus, vous vous demandez peut-être comment vos fenêtres peuvent envoyer un refus. La classe `JInternalFrame` se sert d'un mécanisme général de *JavaBeans* pour examiner les paramètres des propriétés. Nous aborderons ce mécanisme dans tous ses détails au cours du Chapitre 8. Pour l'instant, nous voulons juste vous montrer comment vos fenêtres peuvent refuser des requêtes de modifications de propriétés.

Les fenêtres n'utilisent généralement pas un refus pour protester contre une demande d'icônification ou contre une perte de focus. En revanche, il arrive très couramment que des fenêtres vérifient si elles peuvent être fermées. Une fenêtre peut être fermée par la méthode `setClosed` de la classe `JInternalFrame`. Comme cette méthode peut être refusée, elle appelle tous les *écouteurs de modifications refusables* avant d'effectuer réellement la modification. Cela fournit à chaque écouteur la possibilité de déclencher une `PropertyVetoException`, et par conséquent de terminer l'appel à `setClosed` avant que les paramètres ne soient modifiés.

Dans notre programme d'exemple, nous affichons une boîte de dialogue pour demander à l'utilisateur si la fenêtre peut être fermée (voir Figure 5.51). Si l'utilisateur n'est pas d'accord, la fenêtre reste ouverte.

Figure 5.51

L'utilisateur peut refuser la propriété de fermeture.



430 Au cœur de Java 2 - Fonctions avancées

Voici comment obtenir cet avertissement.

1. Ajoutez un écouteur à chaque fenêtre. Cet écouteur doit appartenir à une classe qui implémente l'interface `VetoableChangeListener`. Il vaut mieux ajouter l'écouteur juste après la construction de la fenêtre. Dans notre exemple, nous nous servons de la classe qui construit toutes les fenêtres internes. Une autre option serait d'utiliser une classe interne anonyme.

```
iframe.addVetoableChangeListener(listener);
```

2. Implémentez la méthode `vetoableChange`, la seule méthode requise par l'interface `VetoableChangeListener`. Cette méthode reçoit un objet `PropertyChangeEvent`. Utilisez la méthode `getName` pour trouver le nom de la propriété qui doit être modifiée (comme "closed" si l'appel de méthode à refuser est `setClosed(true)`). Comme nous le verrons au Chapitre 8, le nom de la propriété est obtenu en supprimant le préfixe "set" du nom de la méthode et en mettant la lettre suivante en minuscule.

Utilisez la méthode `getNewValue` pour obtenir la nouvelle valeur proposée.

```
String name = event.getPropertyName();
Object value = event.getNewValue();
if (name.equals("closed") && value.equals(true))
{
    demande confirmation à l'utilisateur
}
```

3. Il suffit de déclencher une `PropertyVetoException` pour bloquer le changement de la propriété. Revenez normalement si vous ne voulez pas refuser le changement.

```
class DesktopFrame extends JFrame
    implements VetoableChangeListener
{
    . . .
    public void vetoableChange(PropertyChangeEvent event)
        throws PropertyVetoException
    {
        . . .
        if (pas d'accord)
            throw new PropertyVetoException(reason, event);
        // se termine normalement s'il n'y a pas de problème
    }
}
```

Boîtes de dialogue dans des fenêtres internes

Si vous utilisez des fenêtres internes, vous ne devriez pas utiliser la classe `JDialog` pour les boîtes de dialogue. Ces boîtes de dialogue possèdent deux inconvénients :

- Elles sont assez lourdes à gérer, car elles créent une nouvelle fenêtre dans le système de fenêtres.
- Le système de fenêtres ne sait pas comment les positionner par rapport à la fenêtre interne qui les a ouvertes.

Pour des boîtes de dialogue simples, il vaut donc mieux utiliser des méthodes `showInternalXxxDialog` de la classe `JOptionPane`. Elles fonctionnent exactement comme les méthodes `showXxxDialog`, sauf qu'elles placent une fenêtre plus simple par-dessus une fenêtre interne.

Comme pour des boîtes de dialogue plus complexes, vous pouvez les construire avec une `JInternalFrame`. Malheureusement, vous ne bénéficiez plus dans ce cas d'aucun support pour les boîtes de dialogue modales.

Dans notre programme d'exemple, nous utilisons une boîte de dialogue interne pour demander à l'utilisateur s'il est d'accord pour fermer une fenêtre.

```
int result
= JOptionPane.showInternalConfirmDialog(iframe, "Prêt à fermer ?"),
  "Choisissez une option", JOptionPane.YES_NO_OPTION);
```

INFO

Si vous voulez juste être *averti* lorsqu'une fenêtre est fermée, il ne faut pas utiliser le mécanisme de refus. Il vaut mieux installer un `InternalFrameListener`. Un écouteur de fenêtre interne fonctionne exactement comme un `WindowListener`. Lorsque la fenêtre interne est fermée, la méthode `internalFrameClosing` est appelée à la place de la méthode traditionnelle `windowClosing`. Les six autres messages concernant les fenêtres internes (ouverte/fermée, icônifiée/décônifiée, active/inactive) correspondent aussi aux méthodes d'écouteur de fenêtres.

Déplacer le contour d'une fenêtre

L'une des critiques que les développeurs ont exposées à l'encontre des fenêtres internes est leur faible performance. Loin en tête de ces critiques, l'opération la plus lente consiste à déplacer une fenêtre au contenu complexe. Le système demande en permanence à la fenêtre de se redessiner, ce qui prend beaucoup de temps.

A l'heure actuelle, si vous utilisez Windows ou X Windows avec un pilote vidéo peu performant, vous êtes probablement confronté au même problème. Sous Windows, ce problème est résolu directement par la carte vidéo, qui recopie l'intérieur de la fenêtre dans une autre zone mémoire pendant le déplacement.

Pour éviter ces problèmes, vous pouvez choisir de déplacer uniquement le contour de la fenêtre. Lorsque l'utilisateur déplace une fenêtre, seul son cadre est affiché en permanence. L'intérieur de la fenêtre est dessiné uniquement lorsque l'utilisateur lâche le bouton de la souris pour indiquer la position finale de la fenêtre.

Pour activer le déplacement du contour, utilisez :

```
desktop.setDragMode(JDesktopPane.OUTLINE_DRAG_MODE);
```

Ce réglage équivaut à la mise en page continue dans la classe `JSplitPane`.

INFO

Dans les premières versions de Swing, vous deviez utiliser l'incantation magique :

```
desktop.putClientProperty(
  "JDesktopPane.dragMode", "outline");
```

pour activer le glissement du cadre.

Dans notre programme d'exemple, vous pouvez cocher la case Fenêtre > Déplacer le contour pour activer ou désactiver le déplacement du contour.

432 Au cœur de Java 2 - Fonctions avancées**INFO**

Les fenêtres internes du bureau sont gérées par une classe `DesktopManager`. Vous n'avez pas besoin de vous soucier de cette classe dans le cadre d'une programmation classique. Il est possible d'implémenter d'autres comportements pour le bureau en installant un nouveau gestionnaire de bureau, mais nous n'aborderons pas ce point.

L'Exemple 5.21 remplit un bureau avec des fenêtres internes qui montrent des pages HTML. Le menu Fichier > Ouvrir affiche une boîte de dialogue permettant d'afficher un fichier HTML dans une nouvelle fenêtre interne. Si vous cliquez sur un lien, la page correspondante est affichée dans une autre fenêtre interne. Vous pouvez également tester les commandes Fenêtre > Cascade et Fenêtre > Juxtaposition. Cet exemple termine notre étude des caractéristiques avancées de Swing.

Exemple 5.21 : *InternalFrameTest.java*

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Ce programme montre l'utilisation de fenêtres internes.
 */
public class InternalFrameTest
{
    public static void main(String[] args)
    {
        JFrame frame = new DesktopFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ces fenêtres de bureau contiennent des volets d'édition qui présentent
 * des documents HTML.
 */
class DesktopFrame extends JFrame
{
    public DesktopFrame()
    {
        setTitle("InternalFrameTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        desktop = new JDesktopPane();
        add(desktop, BorderLayout.CENTER);

        // définit les menus

        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);
        JMenu fileMenu = new JMenu("Fichier");
        menuBar.add(fileMenu);
        JMenuItem openItem = new JMenuItem("Nouveau");
```

```

openItem.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            createInternalFrame(
                new JLabel(new ImageIcon(planets[counter] + ".gif")),
                planets[counter]);
            counter = (counter + 1) % planets.length;
        }
    });
fileMenu.add(openItem);
JMenuItem exitItem = new JMenuItem("Quitter");
exitItem.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            System.exit(0);
        }
    });
fileMenu.add(exitItem);
JMenu windowMenu = new JMenu("Fenêtre");
menuBar.add(windowMenu);
JMenuItem nextItem = new JMenuItem("Suivante");
nextItem.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            selectNextWindow();
        }
    });
windowMenu.add(nextItem);
JMenuItem cascadeItem = new JMenuItem("Cascade");
cascadeItem.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            cascadeWindows();
        }
    });
windowMenu.add(cascadeItem);
JMenuItem tileItem = new JMenuItem("Juxtaposition");
tileItem.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            tileWindows();
        }
    });
windowMenu.add(tileItem);
final JCheckBoxMenuItem dragOutlineItem
    = new JCheckBoxMenuItem("Déplacer le contour");
dragOutlineItem.addActionListener(new
    ActionListener()
    {

```


434 Au cœur de Java 2 - Fonctions avancées

```

        public void actionPerformed(ActionEvent event)
        {
            desktop.setDragMode(dragOutlineItem.isSelected()
                ? JDesktopPane.OUTLINE_DRAG_MODE
                : JDesktopPane.LIVE_DRAG_MODE);
        }
    });
    windowMenu.add(dragOutlineItem);
}

/**
 * Crée une fenêtre interne sur le bureau.
 * @param c le composant à afficher dans la fenêtre interne
 * @param t le titre de la fenêtre interne
 */
public void createInternalFrame(Component c, String t)
{
    final JInternalFrame iframe = new JInternalFrame(t,
        true, // la taille peut être modifiée
        true, // la fenêtre peut être fermée
        true, // la fenêtre peut être maximisée
        true); // la fenêtre peut être icônifiée

    iframe.add(c, BorderLayout.CENTER);
    desktop.add(iframe);

    iframe.setFrameIcon(new ImageIcon("document.gif"));

    // ajoute un écouteur pour confirmer la fermeture de la fenêtre
    iframe.addVetoableChangeListener(new
        VetoableChangeListener()
    {
        public void vetoableChange(PropertyChangeEvent event)
            throws PropertyVetoException
        {
            String name = event.getPropertyName();
            Object value = event.getNewValue();

            // nous voulons juste vérifier les tentatives pour fermer
            // une fenêtre
            if (name.equals("closed") && value.equals(true))
            {
                // demande à l'utilisateur s'il accepte de fermer la
                // fenêtre
                int result
                    = JOptionPane.showInternalConfirmDialog(
                        iframe, "Prêt à fermer ?",
                        "Choisissez une option",
                        JOptionPane.YES_NO_OPTION);

                // s'il n'est pas d'accord, refus de fermeture
                if (result != JOptionPane.YES_OPTION)
                    throw new PropertyVetoException(
                        "L'utilisateur a annulé la fermeture", event);
            }
        }
    });
}

```

```
// place la fenêtre
int width = desktop.getWidth() / 2;
int height = desktop.getHeight() / 2;
iframe.reshape(nextFrameX, nextFrameY, width, height);

iframe.show();

// sélectionne la fenêtre (peut être refusé)
try
{
    iframe.setSelected(true);
}
catch(PropertyVetoException e)
{}

frameDistance = iframe.getHeight()
    - iframe.getContentPane().getHeight();

// calcule l'emplacement de la prochaine fenêtre

nextFrameX += frameDistance;
nextFrameY += frameDistance;
if (nextFrameX + width > desktop.getWidth())
    nextFrameX = 0;
if (nextFrameY + height > desktop.getHeight())
    nextFrameY = 0;
}

/**
 * Met en cascade les fenêtres internes non-icônifiées du bureau.
 */
public void cascadeWindows()
{
    int x = 0;
    int y = 0;
    int width = desktop.getWidth() / 2;
    int height = desktop.getHeight() / 2;

    for (JInternalFrame frame : desktop.getAllFrames())
    {
        if (!frames[i].isIcon())
        {
            try
            {
                // essaye de modifier la taille des fenêtres maximisées
                // (peut être refusé)
                frame.setMaximum(false);
                frame.reshape(x, y, width, height);

                x += frameDistance;
                y += frameDistance;
                // s'entoure aux bords du bureau
                if (x + width > desktop.getWidth()) x = 0;
                if (y + height > desktop.getHeight()) y = 0;
            }
            catch (PropertyVetoException e)
            {}
        }
    }
}
```

436 Au cœur de Java 2 - Fonctions avancées

```

    }

    /**
     * Juxtapose les fenêtres internes non-icônifiées du bureau.
     */
    public void tileWindows()
    {
        // compte les fenêtres non icônifiées
        int frameCount = 0;
        for (JInternalFrame frame : desktop.getAllFrames())
            if (!frame.isIcon()) frameCount++;
        if (frameCount == 0) return;

        int rows = (int) Math.sqrt(frameCount);
        int cols = frameCount / rows;
        int extra = frameCount % rows;
        // nombre de colonnes avec une ligne supplémentaire

        int width = desktop.getWidth() / cols;
        int height = desktop.getHeight() / rows;
        int r = 0;
        int c = 0;
        for (JInternalFrame frame : desktop.getAllFrames())
        {
            if (!frame.isIcon())
            {
                try
                {
                    frame.setMaximum(false);
                    frame.reshape(c * width,
                                r * height, width, height);
                    r++;
                    if (r == rows)
                    {
                        r = 0;
                        c++;
                        if (c == cols - extra)
                        {
                            // commence à ajouter une autre ligne
                            rows++;
                            height = desktop.getHeight() / rows;
                        }
                    }
                }
                catch (PropertyVetoException e)
                {}
            }
        }
    }

    /**
     * Amène à l'avant la prochaine fenêtre non icônifiée.
     */
    public void selectNextWindow()
    {
        JInternalFrame[] frames = desktop.getAllFrames();
        for (int i = 0; i < frames.length; i++)
        {
            if (frames[i].isSelected())

```

Paramètres :

mode	JDesktopPane.LIVE_DRAG_MODE ou JDesktopPane.OUTLINE_DRAG_MODE

438 Au cœur de Java 2 - Fonctions avancées

javax.swing.JInternalFrame 1.2

- `JInternalFrame()`
- `JInternalFrame(String title)`
- `JInternalFrame(String title, boolean resizable)`
- `JInternalFrame(String title, boolean resizable, boolean closable)`
- `JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable)`
- `JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable, boolean iconifiable)`

Construisent une nouvelle fenêtre interne.

Paramètres :

<code>title</code>	la chaîne à afficher dans la barre de titre
<code>resizable</code>	vaut <code>true</code> si la taille de la fenêtre peut être modifiée
<code>closable</code>	vaut <code>true</code> si la fenêtre peut être fermée
<code>maximizable</code>	vaut <code>true</code> si la fenêtre peut être maximisée
<code>iconifiable</code>	vaut <code>true</code> si la fenêtre peut être icônifiée

- `boolean isResizable()`
- `void setResizable(boolean b)`
- `boolean isClosable()`
- `void setClosable(boolean b)`
- `boolean isMaximizable()`
- `void setMaximizable(boolean b)`
- `boolean isIconifiable()`
- `void setIconifiable(boolean b)`

Renvoient ou définissent les propriétés `resizable`, `closable`, `maximizable` et `iconifiable`. Lorsque la propriété vaut `true`, une icône apparaît dans le titre de la fenêtre pour modifier sa taille, pour la fermer, pour la maximiser ou pour l'icônifier.

- `boolean isIcon()`
- `void setIcon(boolean b)`
- `boolean isMaximum()`
- `void setMaximum(boolean b)`
- `boolean isClosed()`
- `void setClosed(boolean b)`

Renvoient ou définissent la propriété `icon`, `maximum` ou `closed`. Lorsque la propriété vaut `true`, la fenêtre interne est icônifiée, maximisée ou fermée.

- `boolean isSelected()`
- `void setSelected(boolean b)`

Renvoient ou définissent la propriété `selected`. Lorsque la propriété vaut `true`, la fenêtre interne courante devient la fenêtre sélectionnée du bureau.

- `void moveToFront()`
- `void moveToBack()`

Déplacent cette fenêtre interne sur le devant ou sur l'arrière du bureau.

- `void reshape(int x, int y, int width, int height)`

Déplace et modifie la taille de cette fenêtre interne.

Paramètres : `x, y` le coin supérieur gauche de la fenêtre
 `width, height` la largeur et la hauteur de la fenêtre

- `Container getContentPane()`
- `void setContentPane(Container c)`

Renvoient ou définissent le panneau interne de cette fenêtre interne.

- `JDesktopPane getDesktopPane()`

Renvoie le panneau de bureau de cette fenêtre interne.

- `Icon getFrameIcon()`
- `void setFrameIcon(Icon anIcon)`

Renvoient ou définissent l'icône de la fenêtre affichée dans la barre de titre.

- `boolean isVisible()`
- `void setVisible(boolean b)`

Renvoient ou définissent la propriété "visible".

- `void show()`

Rend cette fenêtre interne visible et la place devant toutes les autres.

javax.swing.JComponent 1.2

- `void addVetoableChangeListener(VetoableChangeListener listener)`

Ajoute un écouteur de changement refusable qui est averti lorsqu'une propriété avec contraintes est modifiée.

java.beans.VetoableChangeListener 1.1

- `void vetoableChange(PropertyChangeEvent event)`

Cette méthode est appelée lorsque la méthode `set` d'une propriété avec contrainte avertit un écouteur de modification refusable.

java.beans.PropertyChangeEvent 1.1

- `String getPropertyNames()`

Renvoie le nom de la propriété qui est en train d'être modifiée.

- `Object getNewValue()`

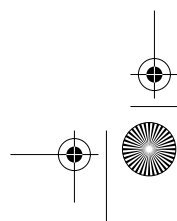
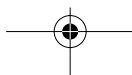
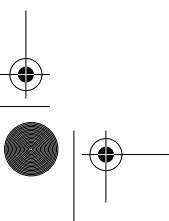
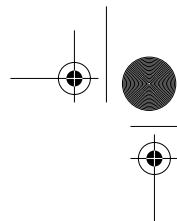
Renvoie la nouvelle valeur proposée pour la propriété.

java.beans.PropertyVetoException 1.1

- `PropertyVetoException(String reason, PropertyChangeEvent event)`

Construit une exception de refus de propriété.

Paramètres : `reason` la raison du refus
 `event` l'événement refusé



6

JavaBeans™

Au sommaire de ce chapitre

- ✓ Pourquoi les beans ?
- ✓ Le processus d'écriture des beans
- ✓ Construire une application à l'aide des beans
- ✓ Les modèles de nom pour les propriétés et événements de bean
- ✓ Les types de propriétés de bean
- ✓ Les classes beanInfo
- ✓ Les éditeurs de propriétés
- ✓ Les customizers
- ✓ La persistance de JavaBeans

La définition officielle d'un bean, donnée dans la spécification JavaBeans, est la suivante :

"Un bean est un composant logiciel réutilisable basé sur la spécification Sun JavaBeans, pouvant être manipulé visuellement dans un outil de génération."

Lorsque vous implémentez un bean, des tiers peuvent les utiliser dans un environnement de génération tel que NetBeans ou JBuilder, ce qui permet de créer des applications de GUI plus efficacement.

Nous ne vous apprendrons pas ici à utiliser ces environnements, vous devez pour cela vous reporter à la documentation du fournisseur. Ce chapitre explique ce que vous devez savoir au sujet des beans pour pouvoir les *implémenter* afin que les autres programmeurs puissent les utiliser facilement.

INFO

Nous aimerions clarifier un point assez confus avant de poursuivre : les JavaBeans que nous verrons dans ce chapitre ont peu de choses en commun avec les "Enterprise JavaBeans" ou EJB. Les Enterprise JavaBeans sont des composants côté serveur qui prennent en charge les transactions, la persistance, la réplication et les fonctions de sécurité. A la base, ce sont aussi des composants que l'on peut manipuler dans les outils de génération. La technologie Enterprise JavaBeans est toutefois un peu plus complexe que celle de "l'Edition Standard".

442 Au cœur de Java 2 - Fonctions avancées

Cela ne signifie pas que les composants JavaBeans standard soient limités à la programmation côté client. Des technologies Web comme le JavaServer Faces (JSF) et le JavaServer Pages (JSP) se reposent lourdement sur le modèle de composants JavaBeans.

Pourquoi les beans ?

Les programmeurs accoutumés à l'environnement Windows (en particulier Visual Basic ou C#) comprendront immédiatement l'importance des beans. Les programmeurs venant d'un environnement où la tradition est de "tout faire soi-même" peuvent ne pas comprendre tout de suite. L'expérience montre que les programmeurs non accoutumés à un environnement Visual Basic croient difficilement que VB représente l'un des exemples les plus réussis de la technologie objet réutilisable. L'une des raisons de la popularité de Visual Basic devient évidente si vous pensez à la façon dont vous construisez une application Visual Basic. Pour ceux qui n'ont jamais travaillé avec Visual Basic, voici, en bref, la façon de procéder :

1. Vous construisez l'interface en déposant des composants (appelés *contrôles* dans Visual Basic) sur une feuille, dans une fenêtre.
2. A l'aide des *feuilles de propriétés*, vous définissez les propriétés des composants comme la hauteur, la couleur et autres attributs.
3. Les feuilles de propriétés recensent également les événements auxquels les composants peuvent réagir. Pour certains de ces événements, vous écrivez de courtes séquences de code pour gérer l'événement.

Par exemple, dans le Chapitre 2 du Volume 1, nous avons écrit un programme affichant une image dans un cadre. Cela a nécessité un peu plus d'une page de code. Voici comment vous procéderiez en Visual Basic pour créer un programme ayant pratiquement la même fonctionnalité.

1. Vous ajoutez deux contrôles dans une fenêtre : un contrôle *Image* pour l'affichage graphique et un contrôle *CommonDialog* (Boîte de dialogue commune) pour la sélection d'un fichier.
2. Vous définissez les propriétés *Filter* du contrôle *CommonDialog* afin que seuls les fichiers pris en charge par le contrôle *Image* s'affichent. Cela est réalisé dans ce que VB appelle la fenêtre de propriétés, montrée à la Figure 6.1.

Il suffit ensuite d'écrire les quatre lignes de code VB qui seront exécutées lors du lancement du projet. Le code suivant fait apparaître la boîte de dialogue Fichier ; seuls les fichiers contenant les extensions prévues seront affichés en raison de la façon dont est définie la propriété *Filter*. Lorsque l'utilisateur choisit un fichier image, le code demande au contrôle *Image* de l'afficher. Le seul code nécessaire à cette séquence est le suivant :

```
Private Sub Form_Load()  
    CommonDialog1.ShowOpen  
    Image1.Picture = LoadPicture(CommonDialog1.FileName)  
End Sub
```

C'est tout. L'activité de présentation, combinée à ces instructions, donne pour l'essentiel la même fonctionnalité qu'une page de code Java. En clair, il est beaucoup plus facile d'apprendre à placer des composants sur une feuille et à définir des propriétés, que d'écrire une page de code.

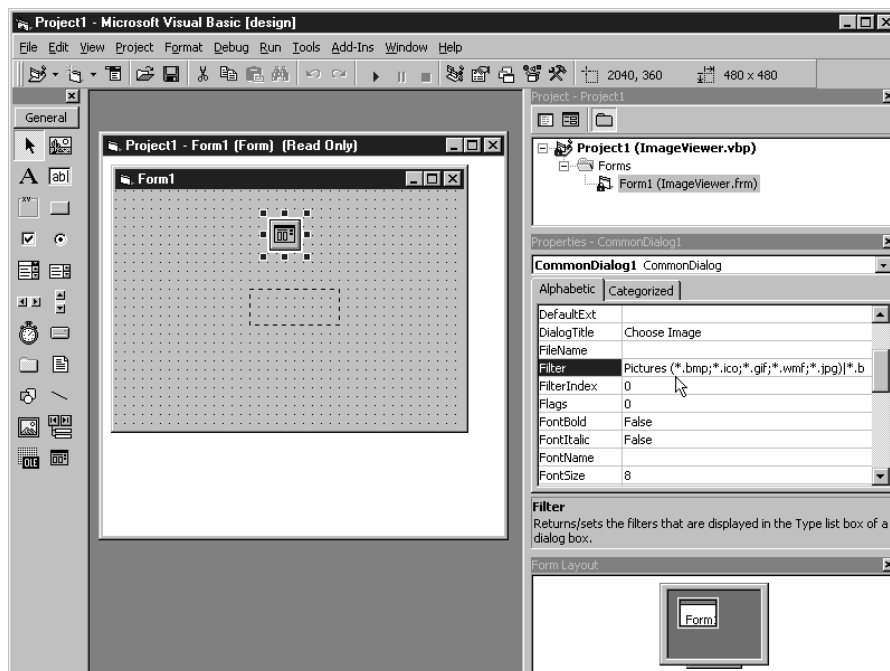


Figure 6.1

La fenêtre de propriétés VB pour une application d’affichage d’image.

Notez que cela ne signifie pas que Visual Basic soit la bonne solution pour tous les problèmes. Il est de toute évidence optimisé pour une forme particulière de solutions : les programmes Windows intensivement axés sur l’interface utilisateur. La technologie JavaBeans a été inventée pour améliorer la compétitivité de Java dans ce domaine. Elle permet aux fournisseurs de créer des environnements dans le style Visual Basic pour le développement Java avec un minimum de programmation.

Le processus d’écriture des beans

L’essentiel du reste de ce chapitre vous enseigne les techniques utilisées pour l’écriture des beans. Avant d’entrer dans les détails, nous allons vous donner un aperçu du processus. Nous voulons tout d’abord insister sur le fait qu’écrire un bean n’est pas techniquement difficile ; il suffit de maîtriser quelques nouvelles classes et interfaces.

En particulier, la forme la plus simple de bean n’est rien de plus qu’une classe Java respectant des conventions d’attribution de nom assez strictes pour ses méthodes.

INFO

Certains auteurs prétendent qu’un bean doit avoir un constructeur par défaut. La spécification JavaBeans n’aborde pas le sujet. En revanche, la plupart des outils de génération exigent un constructeur par défaut pour chaque bean, de sorte qu’ils puissent instancier les beans sans paramètres de construction.

444 Au cœur de Java 2 - Fonctions avancées

L'Exemple 6.1 montre le code pour un bean `ImageViewer` pouvant donner à un environnement de génération Java la même fonctionnalité que le contrôle `Image Visual Basic` mentionné précédemment. En examinant ce code, vous remarquerez que la classe `ImageViewerBean` n'a pas l'air vraiment différente de n'importe quelle autre classe. Par exemple, toutes les méthodes d'accès commencent par `get`, toutes les méthodes d'altération commencent par `set`. Comme vous le verrez bientôt, les outils de génération respectent cette convention standard d'attribution de nom pour découvrir les propriétés. Par exemple, `fileName` possédant les méthodes `get` et `set` est une propriété de ce bean.

Vous remarquerez qu'une propriété n'équivaut pas à une variable d'instance. Dans cet exemple particulier, la propriété `fileName` est calculée à partir de la variable d'instance `file`. Les propriétés sont, de manière conceptuelle, placées à un niveau supérieur aux variables d'instance ; il s'agit en effet de fonctions de l'interface, tandis que les variables d'instance appartiennent à l'implémentation de la classe.

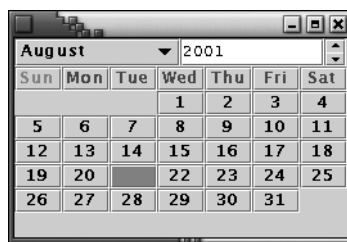
Une chose à garder à l'esprit lors de la lecture des exemples de ce chapitre est que dans la réalité, les beans sont beaucoup plus élaborés et fastidieux à écrire que nos exemples très simples, pour deux raisons.

1. L'utilisation des beans ne doit pas être exclusivement réservée aux experts en programmation. Ces utilisateurs accéderont à l'essentiel des fonctionnalités de vos beans à l'aide d'un outil de conception visuel ne recourant pratiquement pas à la programmation.
2. Le même bean doit être utilisable dans une large *variété de contextes*. Le comportement et l'apparence de votre bean doivent pouvoir être personnalisés. Une fois de plus, cela équivaut à exposer de nombreuses propriétés.

Autre bon exemple de bean affichant un comportement élaboré, `CalendarBean` par Kai Tödter (voir Figure 6.2). Le bean et son code source sont librement disponibles à l'adresse <http://www.toedter.com/en/jcalendar.html>. Ce bean propose aux utilisateurs une méthode commode pour saisir des dates, en les localisant simplement dans un calendrier. Ceci est évidemment assez complexe et vous n'aurez pas envie de le programmer de A à Z. En utilisant des `JavaBeans` comme celui-ci, vous pouvez profiter du travail des autres, en plaçant simplement le bean dans un outil de génération.

Figure 6.2

Un bean de calendrier.



Heureusement, vous n'avez besoin de maîtriser qu'un petit nombre de concepts pour écrire des beans avec une grande variété de comportements. Les exemples de beans dans ce chapitre, bien que ne pouvant être qualifiés de triviaux, sont pourtant suffisamment simples pour illustrer clairement les concepts nécessaires.

Exemple 6.1 : ImageViewerBean.java

```
package com.horstmann.corejava;

import java.awt.*;
import java.io.*;
import javax.imageio.*;
import javax.swing.*;

/**
 * Un bean pour l'affichage d'une image.
 */
public class ImageViewerBean extends JPanel
{
    public ImageViewerBean()
    {
        setBorder(BorderFactory.createEtchedBorder());
    }

    /**
     * Définit la propriété fileName.
     * @param fileName le nom du fichier image
     */
    public void setFileName(String fileName)
    {
        try
        {
            file = new File(fileName);
            setIcon(new ImageIcon(ImageIO.read(file)));
        }
        catch (IOException e)
        {
            file = null;
            setIcon(null);
        }
    }

    /**
     * Récupère la propriété fileName.
     * @return le nom du fichier image
     */
    public String getFileName()
    {
        if (file == null) return null;
        else return file.getPath();
    }

    public Dimension getPreferredSize()
    {
        return new Dimension(XPREFSIZE, YPREFSIZE);
    }

    private File file = null;
    private static final int XPREFSIZE = 200;
    private static final int YPREFSIZE = 200;
}
```

Construire une application à l'aide des beans

Avant d'entrer dans les détails de la rédaction de beans, nous souhaitons vous montrer comment les utiliser ou les tester. `ImageViewerBean` constitue un bean parfaitement utilisable ; toutefois, pris hors d'un environnement de génération, il ne peut pas faire montre de ses fonctions spéciales. En particulier, la seule manière de l'utiliser dans un programme ordinaire dans le langage de programmation Java consisterait à écrire un code qui construirait un objet de la classe bean, placer l'objet dans un conteneur, puis appeler la méthode `setFileName`. Il ne s'agit toutefois pas d'une science exacte, mais c'est un code qu'un programmeur surchargé peut souhaiter écrire. Les environnements de génération ont pour objectif de réduire la somme de travail qu'induit l'écriture de tous les composants dans une application.

Chaque environnement de génération utilise son propre ensemble de stratégies pour faciliter la vie du programmeur. Nous traiterons ici d'un environnement, NetBeans, disponible à l'adresse <http://netbeans.org>. Nous ne voulons toutefois pas proclamer que NetBeans soit meilleur que d'autres produits ; en réalité, comme vous le verrez, il possède sa part d'illogique. Nous l'utilisons simplement parce qu'il s'agit d'un environnement de programme assez typique et parce qu'il est gratuit.

Si vos préférences vont vers un autre environnement de génération, vous pouvez malgré tout suivre les étapes des sections à venir. Les principes de base restent les mêmes pour la plupart des environnements. Bien entendu, les détails diffèrent.

Dans cet exemple, nous utiliserons deux beans, `ImageViewerBean` et `FileNameBean`. Vous connaissez déjà le code de `ImageViewerBean`. Celui de `FileNameBean` est un peu plus compliqué. Nous l'analyserons en détail plus loin dans ce chapitre. Pour l'heure, il vous suffit de savoir que le fait de cliquer sur le bouton contenant "..." affichera une boîte de dialogue Ouvrir standard, dans laquelle vous pourrez sélectionner un fichier.

Avant de vous montrer comment utiliser ces beans, nous devons vous expliquer comment intégrer le bean, afin de l'importer dans un outil de génération.

Intégrer des beans dans des fichiers JAR

Pour rendre un bean utilisable dans un outil de génération, intégrez tous les fichiers de classe utilisés par le code du bean dans un fichier JAR. A la différence des fichiers JAR destinés à une applet que vous avez vus précédemment, un fichier JAR destiné à un bean a besoin d'un fichier manifeste qui spécifie lesquels parmi les fichiers de classe présents dans l'archive constituent des beans et doivent être inclus dans la boîte à outils. Par exemple, voici le fichier `ImageViewerBean.mf` pour `ImageViewerBean` :

```
Manifest-Version: 1.0
```

```
Name: com/horstmann/corejava/ImageViewerBean.class
```

```
Java-Bean: True
```

Vous remarquerez la ligne vierge entre la version du manifeste et le nom du bean.

Si votre classe se trouve dans un package, utilisez des barres obliques pour présenter le pack, comme suit :

```
Name: com/horstmann/beans/ImageViewerBean.class
```

Si votre bean contient plusieurs fichiers de classe, vous ne mentionnerez dans le manifeste que ceux qui constituent des beans et que vous souhaitez afficher dans la boîte à outils. Par exemple, vous pourriez placer un `ImageViewerBean` et un `FileNameBean` dans le même fichier JAR et utiliser le manifeste

```
Manifest-Version: 1.0
```

```
Name: com/horstmann/corejava/ImageViewerBean.class  
Java-Bean: True
```

```
Name: com/horstmann/corejava/FileNameBean.class  
Java-Bean: True
```

ATTENTION

Certains outils de génération sont extrêmement confus sur les manifestes. Assurez-vous qu'il n'y a pas d'espaces après la fin de chaque ligne, qu'il y a des lignes vides après la version et entre les entrées de bean et que la dernière ligne se termine sur une nouvelle ligne.

Pour créer le fichier JAR, procédez comme suit :

1. Modifiez le fichier de manifeste.
2. Rassemblez tous les fichiers de classe nécessaires dans un répertoire.
3. Lancez l'outil `jar` comme suit :

```
jar cfm JarFile ManifestFile ClassFiles
```

Par exemple :

```
jar cfm ImageViewerBean.jar ImageViewerBean.mf  
com/horstmann/corejava/*.class
```

Vous pouvez également ajouter d'autres éléments au fichier JAR, notamment des fichiers GIF pour les icônes. Nous traiterons des icônes des beans plus loin dans ce chapitre.

ATTENTION

Vérifiez que sont inclus tous les fichiers dont vos beans ont besoin dans le fichier JAR. En particulier, prenez garde aux fichiers des classes internes comme `ImageViewBean$1.class`.

Les environnements de génération disposent d'un mécanisme permettant d'ajouter de nouveaux beans, généralement en chargeant des fichiers JAR. Voici ce que vous devez faire pour importer des beans dans NetBeans.

Compilez les classes `ImageViewerBean` et `FileNameBean` et intégrez-les dans des fichiers JAR. Lancez ensuite NetBeans et procédez comme suit :

1. Sélectionnez `Tools > Palette Manager` dans le menu.
2. Cliquez sur le bouton "Add from JAR".

448 *Au cœur de Java 2 - Fonctions avancées*

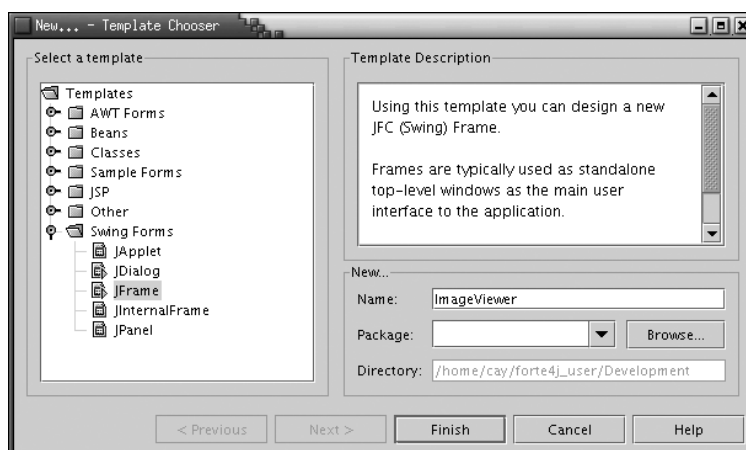
3. Dans la boîte de dialogue, accédez au répertoire `ImageViewerBean` et sélectionnez `ImageViewerBean.jar`.
4. Une boîte de dialogue s'affiche, énumérant tous les beans trouvés dans le fichier JAR. Sélectionnez `ImageViewerBean`.
5. Enfin, vous devez choisir la palette dans laquelle vous souhaitez placer tous les beans. Sélectionnez "Beans" (il existe d'autres palettes pour les composants Swing, les composants AWT, etc.).
6. Etudiez la palette "Beans". Elle contient maintenant une icône représentant le nouveau bean. Toutefois, l'icône n'est qu'une icône par défaut, vous verrez plus tard comment ajouter des icônes à un bean.

Répétez cette étape avec `FileNameBean`. Vous pouvez maintenant composer ces beans dans une application.

Composer des beans dans un environnement de génération

Dans NetBeans, sélectionnez `File > New Project` dans le menu. Une boîte de dialogue s'affiche. Sélectionnez `Standard`, puis `Java Application` (voir Figure 6.3).

Figure 6.3
Construction
d'un nouveau projet.



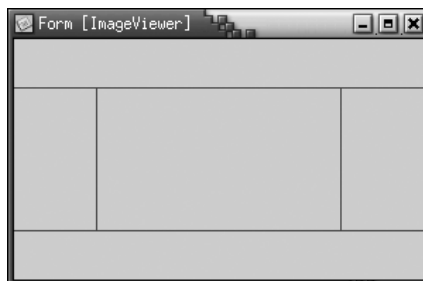
Cliquez sur le bouton "Next". Dans l'écran suivant, définissez le nom de votre application (`ImageViewer`, par exemple) et cliquez sur le bouton "Finish". Vous obtenez maintenant une visionneuse du projet sur la gauche et l'éditeur de code source au milieu.

Cliquez du bouton droit sur le nom du projet dans la visionneuse et choisissez `New -> JFrame Form` dans le menu (voir Figure 6.4).

Une boîte de dialogue s'affiche. Entrez le nom de la classe du cadre (`ImageViewerFrame`, par exemple) et cliquez sur le bouton "Finish". Vous obtenez maintenant un éditeur de formulaire avec un cadre vide. Pour ajouter un bean à la feuille, sélectionnez le bean dans la palette située à droite. Cliquez sur le cadre. Par défaut, le cadre possède une bordure et NetBeans sait ajouter le composant au centre, au nord, au sud, à l'est ou à l'ouest, selon l'endroit où vous cliquez. Vous pouvez toujours modifier la mise en page dans l'éditeur de propriétés.

Figure 6.4

Créer une vue
du formulaire.



INFO

D'autres générateurs possèdent des interfaces utilisateur différentes. Par exemple, vous pouvez avoir besoin de faire glisser le bean à la bonne place sur la feuille.

La Figure 6.5 montre le résultat de l'ajout du bean `ImageViewer` au centre du cadre.

Figure 6.5

Ajout d'un bean.



Si vous étudiez la fenêtre de l'éditeur de code, vous découvrirez que le code source contient désormais les instructions Java permettant d'ajouter un objet bean au cadre (voir Figure 6.6). Le code source est placé entre crochets par des avertissements que vous ne devriez pas modifier. Toute modification serait perdue lorsque l'environnement de génération met à jour le code au moment de modifier le formulaire.

INFO

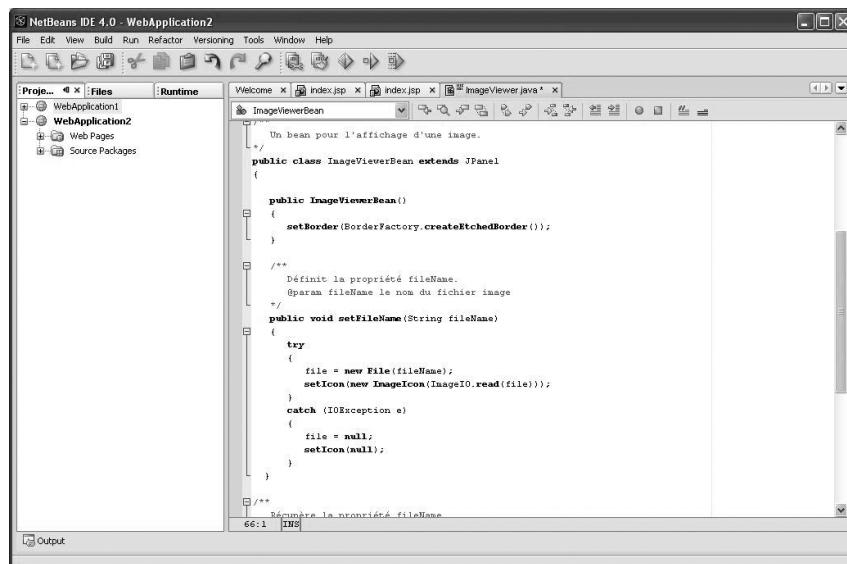
Tous les environnements de génération ne mettent pas à jour le code source lorsque vous construisez une application. Un environnement peut générer un code source lorsque vous avez terminé les modifications, sérialiser les beans que vous avez personnalisés, voire produire une description totalement différente de votre activité de construction.

Le projet expérimental Bean Builder, par exemple (<http://bean-builder.dev.java.net>), vous permet de concevoir des applications de GUI sans même écrire de code source.

Le mécanisme des JavaBeans ne tente pas de forcer la stratégie d'implémentation sur un outil de génération. Il essaie plutôt de fournir des informations sur les beans aux outils de génération capables de choisir s'ils veulent profiter de ces informations d'une manière ou d'une autre. Dans les sections à venir, nous vous montrerons comment programmer ces beans de sorte que les outils découvrent précisément ces informations.

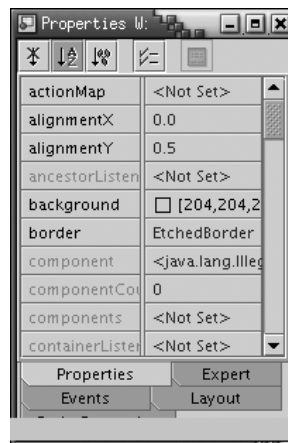
450 Au cœur de Java 2 - Fonctions avancées

Figure 6.6
Code source
de l'ajout
d'un bean.



Revenez maintenant au formulaire et cliquez sur `ImageViewerBean`. Vous trouverez du côté droit une feuille de propriétés qui recense les noms des propriétés des beans, ainsi que leurs valeurs courantes (voir Figure 6.7). Il s'agit d'une partie essentielle des outils de développement basés sur un composant puisque le paramétrage des propriétés au moment de la conception permet de définir l'état initial d'un composant.

Figure 6.7
Une feuille
de propriétés.



Par exemple, vous pouvez modifier la propriété `text` de l'étiquette utilisée pour le bean d'image en tapant simplement un nouveau nom dans la feuille de propriétés. La modification de la propriété `text` demeure simple : il vous suffit de modifier une chaîne dans le champ de texte. Essayez : définissez le texte de l'étiquette sur "Bonjour". La feuille est immédiatement mise à jour en vue de refléter vos modifications.

INFO

Lorsque vous modifiez le réglage d'une propriété, l'environnement de NetBeans met à jour le code source en vue de refléter votre action. Par exemple, si vous définissez le champ text sur Bonjour, l'instruction

```
imageViewBean.setText("Bonjour");
```

est ajoutée à la méthode `initComponents` . Comme nous l'avons déjà indiqué, d'autres outils de génération peuvent disposer de stratégies différentes pour enregistrer les réglages des propriétés.

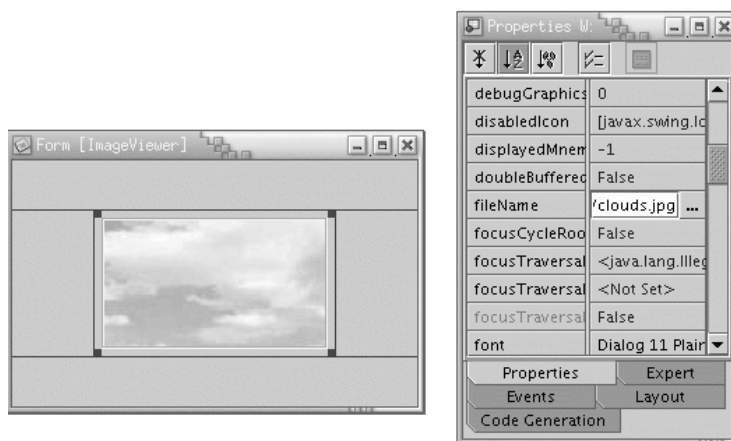
Les propriétés n'ont pas nécessairement à être des chaînes. Il peut s'agir de valeurs de n'importe quel type de plate-forme Java. Pour permettre aux utilisateurs de définir des valeurs pour les propriétés de toutes sortes, les outils de génération accèdent à des *éditeurs de propriétés* spécialisés (les éditeurs de propriétés sont livrés avec le générateur ou fournis avec le développeur de beans. Vous verrez comment écrire vos propres éditeurs de propriétés plus loin dans ce chapitre).

Pour découvrir un éditeur de propriétés simple, étudiez la propriété `foreground` . Son type est `Color` . Vous pouvez voir l'éditeur des couleurs, avec un champ de texte contenant la chaîne `[0,0,0]` , ainsi qu'un bouton intitulé `"... "` qui fait apparaître une boîte de dialogue des couleurs. Modifiez la couleur du premier plan. Vous remarquerez les changements de la valeur de propriété : le texte de l'étiquette change de couleur.

Mieux encore : choisissez un nom de fichier pour un fichier image dans la feuille de propriétés. `ImageViewerBean` affiche automatiquement l'image (voir Figure 6.8).

Figure 6.8

Le bean `ImageViewerBean` en action.



INFO

Si vous examinez la feuille de propriétés de droite à la Figure 6.8, vous verrez quantité de propriétés mystérieuses telles que `focusCycleRoot` et `iconTextGap` . Elles sont l'héritage de la superclasse `JLabel` . Vous verrez, plus loin dans ce chapitre, comment les supprimer de la feuille de propriétés.

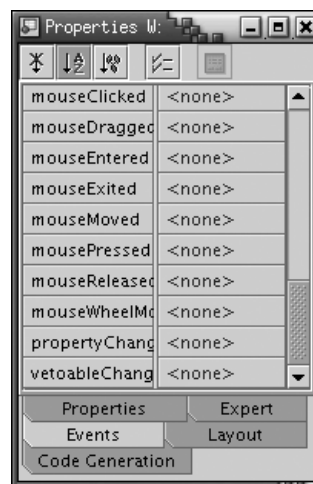
452 Au cœur de Java 2 - Fonctions avancées

Pour terminer notre application, placez un `FileNameBean` à l'extrémité inférieure du cadre. Nous voulons maintenant que l'image se charge lors d'une modification de la propriété `fileName` de `FileNameBean`. Cela survient par le biais de l'événement `PropertyChange`, nous traiterons de ces types d'événements un peu plus loin dans ce chapitre.

Pour réagir à l'événement, sélectionnez `FileNameBean` et l'onglet `Events` de la feuille de propriétés (voir Figure 6.9). Cliquez ensuite sur le bouton "... " près de l'entrée `propertyChange`. Une boîte de dialogue apparaît, indiquant qu'il n'existe actuellement pas de gestionnaires associés à cet événement. Cliquez sur le bouton `Add` de la boîte de dialogue. Un nom de méthode vous est demandé. Tapez `loadImage`.

Figure 6.9

L'onglet `Events` de la feuille de propriétés.



Etudions maintenant la fenêtre d'édition. Elle contient désormais un autre code de gestion d'événements ainsi qu'une nouvelle méthode :

```
private void loadImage(java.beans.PropertyChange evt)
{
    //Ajoutez ici votre code de gestion
}
```

Ajoutez la ligne de code suivante :

```
imageViewBean1.setFileName(fileNameBean1.getFileName());
```

Compilez ensuite l'application et exécutez-la. Vous avez maintenant une visionneuse d'images complète. Cliquez sur le bouton possédant l'étiquette "..." et sélectionnez un fichier image. L'image s'affiche dans l'afficheur.

Cette procédure démontre que vous pouvez créer une application Java à partir des beans Java, en définissant des propriétés et en fournissant une petite partie de code destinée aux gestionnaires d'événements.

Les modèles de nom pour les propriétés et événements de bean

Nous verrons dans cette section les règles de base pour la conception de vos propres beans. Tout d'abord, il est important d'insister sur le fait qu'il n'existe *pas* de classe de bean cosmique que vous puissiez étendre pour construire vos beans. Les beans visuels étendent directement ou indirectement la classe `Component`, mais les beans non visuels n'ont pas besoin d'étendre de superclasse particulière. Souvenez-vous, un bean est simplement *toute* classe pouvant être manipulée dans un outil de conception visuel. L'outil de conception ne consulte pas la superclasse pour déterminer la nature de bean d'une classe, mais il analyse les noms de ses méthodes. Pour permettre cette analyse, les noms des méthodes pour les beans doivent respecter certains modèles.

INFO

Il existe une classe `java.beans.Beans`, mais toutes les méthodes qu'elle contient sont statiques. L'étendre serait donc *a priori* sans intérêt, bien que cela soit fait à l'occasion, en principe pour une plus grande "clarté". En clair, puisqu'un bean ne peut étendre à la fois des classes `Beans` et `Component`, cette approche n'est pas valable pour les beans visuels. En fait, la classe `Beans` contient des méthodes qui sont conçues pour être appelées par les outils du générateur, pour vérifier, par exemple, si l'outil agit au moment de la création ou au moment de l'exécution.

D'autres langages pour les environnements de conception visuels, comme Visual Basic et C#, ont des mots clés spéciaux tels que "Property" et "Event" pour exprimer directement ces concepts. Les concepteurs de Java ont décidé de ne pas ajouter de mots clés au langage pour la prise en charge de la programmation visuelle. Ils avaient par conséquent besoin d'une alternative afin qu'un outil du générateur puisse analyser un bean pour connaître ses propriétés ou ses événements. Il existe en réalité deux mécanismes possibles. Si le créateur du bean utilise les conventions standard d'attribution de nom pour les propriétés et les événements, l'outil du générateur peut utiliser le mécanisme de réflexion pour savoir quels événements et propriétés le bean est censé mettre à disposition. Le créateur du bean peut aussi fournir une classe *d'information de bean* renseignant l'outil du générateur sur les propriétés et événements du bean. Nous allons d'abord utiliser les modèles de nom, car ils sont simples à utiliser. Vous verrez plus loin dans ce chapitre, comment fournir une classe d'information pour le bean.

INFO

Bien que la documentation désigne ces modèles de noms standard comme des "modèles de conception", ce ne sont en réalité que des conventions d'attribution de nom et ils n'ont rien à voir avec les modèles de conception utilisés en programmation orientée objet.

Le modèle de nom pour les propriétés est simple : Toute paire de méthodes

```
public Type getNompropriété()  
public void setNompropriété(Type newValue)
```

correspond à une propriété read/write.

454 *Au cœur de Java 2 - Fonctions avancées*

Par exemple, dans notre `ImageViewerBean`, il n'y a qu'une propriété `read/write` (pour le nom du fichier à visualiser), avec les méthodes suivantes :

```
public String getFileName()
public void setFileName(String newValue)
```

Si vous avez une méthode `get` sans méthode `set` associée, vous définissez une propriété en lecture seule. A l'inverse, une méthode `set` sans méthode `get` associée définit une propriété en écriture seule.

INFO

Les méthodes `get` et `set` que vous créez peuvent faire plus que simplement extraire ou définir un champ de données privé. Comme toute méthode Java, elles peuvent exécuter des actions arbitraires. Par exemple, la méthode `setFileName` de la classe `ImageViewerBean` non seulement définit le champ de données `fileName`, mais elle ouvre aussi le fichier et charge l'image.

INFO

Dans Visual Basic et C#, les propriétés proviennent aussi des méthodes `get` et `set`. Mais, dans ces deux langages, vous pouvez définir explicitement des propriétés plutôt qu'amener les outils de génération à analyser les noms de méthodes pour tenter de deviner les intentions du programmeur. Dans ces langages, les propriétés ont un autre avantage : l'utilisation d'un nom de propriété du côté gauche d'une instruction d'affectation appelle automatiquement la méthode `set`. L'utilisation d'un nom de propriété dans une expression appelle automatiquement la méthode `get`. Par exemple, en VB vous pouvez écrire

```
imageBean.fileName = "corejava.gif"
```

au lieu de

```
imageBean.setFileName("corejava.gif");
```

Cette syntaxe était envisagée pour Java, mais les concepteurs ont considéré qu'il ne valait mieux pas masquer un appel de méthode derrière une syntaxe ressemblant à un accès de champ.

Il y a une exception en ce qui concerne le modèle de nom de `get/set`. Les propriétés ayant des valeurs booléennes doivent utiliser un modèle de nom `is/set`, comme dans les exemples suivants :

```
public boolean isNompropriété()
public void setNompropriété(boolean b)
```

Par exemple, une animation peut avoir une propriété `running`, avec deux méthodes :

```
public boolean isRunning()
public void setRunning(boolean b)
```

La méthode `setRunning` lancera et arrêtera l'animation. La méthode `isRunning` va afficher son statut actuel.

INFO

Vous pouvez utiliser un préfixe `get` pour une méthode d'accès à une propriété booléenne (comme `getRunning`), mais on préférera le préfixe `is`.

Soyez vigilant en ce qui concerne le modèle de capitalisation que vous utilisez pour vos noms de méthodes. Les concepteurs de JavaBeans ont spécifié que le nom de la propriété dans notre exemple serait `fileName`, avec un `f` minuscule, même si les méthodes `get` et `set` contiennent un `F` majuscule (`getFileName`, `setFileName`). L'analyseur de bean exécute un processus appelé *décapitalisation* pour dériver le nom de propriété (c'est-à-dire que le premier caractère après `get` ou `set` est converti en minuscule). Il en résulte des noms de propriétés et de méthodes plus logiques pour les programmeurs.

Toutefois, si les *deux* premières lettres sont en majuscules (comme dans `getURL`), la première lettre de la propriété ne passera pas en minuscules. Après tout, le nom de propriété `url` serait ridicule.

INFO

Que faire si votre classe possède une paire de méthodes `get` et `set` qui ne corresponde pas à une propriété que vos utilisateurs veulent manipuler dans une feuille de propriétés ? Dans vos propres classes, vous pouvez bien sûr éviter cette situation en renommant vos méthodes. Toutefois, si vous étendez une autre classe, vous héritez les noms de méthode de la superclasse. Ceci survient par exemple lorsque votre bean étend `JPanel` ou `JLabel` : un grand nombre de propriétés inintéressantes apparaissent dans la feuille de propriétés. Vous verrez plus loin dans ce chapitre comment surcharger la procédure automatique de découverte de propriétés en fournissant des *informations sur le bean*. Vous pouvez y indiquer précisément les propriétés que votre bean doit exposer.

Pour les événements, le modèle de nom est encore plus simple. Un environnement de génération de bean va impliquer la génération d'événements par le bean, lorsque vous fournirez des méthodes pour l'ajout et la suppression d'écouteurs. Tous les noms de classe d'événement doivent se terminer par `Event`. Les classes doivent étendre la classe `EventObject`.

Supposons que votre bean génère des événements du type `NomEvénementEvent`. L'interface de l'écouteur doit alors s'appeler `NomEvénementListener`, et les méthodes permettant d'ajouter et de supprimer un écouteur doivent s'appeler :

```
public void addNomEvénementListener(NomEvénementListener e)
public void removeNomEvénementListener(NomEvénementListener e)
```

Si vous examinez le code pour `ImageViewerBean`, vous constaterez qu'aucun événement n'est mis à disposition. Toutefois, de nombreux composants Swing génèrent des événements et suivent ce motif. Par exemple, la classe `AbstractButton` génère des objets `ActionEvent` et possède les méthodes suivantes pour gérer des objets `ActionListener` :

```
public void addActionListener(ActionListener e)
public void removeActionListener(ActionListener e)
```

ATTENTION

Si votre classe d'événement n'étend pas `EventObject`, il y a des chances pour que votre code se compile bien car aucune de ses méthodes n'est réellement nécessaire. Toutefois, votre bean échouera mystérieusement : le mécanisme d'introspection ne reconnaît pas les événements.

Les types de propriétés de bean

Un bean élaboré aura quantité de sortes différentes de propriétés pouvant être mises à disposition dans un outil de génération afin que l'utilisateur puisse les définir au moment de la création ou les extraire au moment de l'exécution. Il peut aussi déclencher à la fois des événements standard et des événements personnalisés. Les propriétés peuvent être aussi simples que la propriété `fileName` que nous avons vue dans `ImageViewerBean` et `FileNameBean`, ou être aussi sophistiquées qu'une valeur de couleur ou même un tableau de points de données. Nous allons voir l'un et l'autre cas plus loin dans ce chapitre. Les propriétés peuvent de plus déclencher des événements, comme vous le verrez dans cette section.

Obtenir les propriétés correctes pour vos beans est probablement la partie la plus complexe de la construction d'un bean, car le modèle est très riche. La spécification JavaBeans autorise quatre types de propriétés, que nous allons illustrer à l'aide d'exemples.

Les propriétés simples

Une propriété simple est une propriété qui ne prend qu'une seule valeur comme une chaîne ou un nombre. La propriété `fileName` de `ImageViewer` en est un exemple. Les propriétés simples sont faciles à programmer : il suffit d'utiliser la convention `set/get` décrite précédemment. Si, par exemple, vous examinez le code de l'Exemple 6.1, vous verrez que pour implémenter une propriété chaîne (`String`) simple il suffit d'écrire :

```
public void setFileName(String f)
{
    fileName = f;
    image = . . .
    repaint();
}

public String getFileName()
{
    if (file == null) return null;
    else return file.getPath();
}
```

Remarquez que, en ce qui concerne la spécification JavaBeans, nous avons également une propriété en lecture seule de ce bean puisque nous avons à l'intérieur de la classe une méthode

```
public Dimension getPreferredSize()
```

sans méthode `setPreferredSize` correspondante. Vous ne pouvez normalement pas voir les propriétés en lecture seule, au moment de la création, dans une feuille de propriétés.

Les propriétés indexées

Une propriété indexée est une propriété qui extrait ou définit un tableau. Un bean *Chart* (histogramme) — voir ci-après — utilisera une propriété indexée pour les points de données. Avec une propriété indexée, vous fournissez deux paires de méthodes `get` et `set` : une pour le tableau et une pour les entrées individuelles. Elles doivent être conformes au modèle suivant :

```
Type[] getNomPropriété()
void setNomPropriété(Type[] x)
```

```
Type getNomPropriété(int i)
void setNomPropriété(int i, Type x)
```

Voici un exemple de la propriété indexée que nous utiliserons dans le bean Chart, que nous allons étudier plus loin dans ce chapitre :

```
public double[] getValues() { return values; }
public void setValues(double[] v) { values = v; }
public double getValues(int i)
{
    if (0 <= i && i < values.length) return values[i];
    return 0;
}
public void setValues(int i, double value)
{
    if (0 <= i && i < values.length) values[i] = value;
}
...
private double[] values;
```

La méthode

```
setNomPropriété(int i, Type[] x)
```

ne peut être utilisée pour *agrandir* le tableau. Pour cela, vous devez manuellement construire un nouveau tableau, puis le passer à cette méthode :

```
setNomPropriété(Type[] x)
```

INFO

Lorsque nous écrivons cela, NetBeans ne gère pas les propriétés indexées. Vous verrez plus loin dans ce chapitre comment passer outre cette limitation en fournissant un éditeur de propriétés pour les tableaux.

Les propriétés liées

Les propriétés liées signalent aux écouteurs intéressés que leurs valeurs ont changé. Par exemple, la propriété `fileName` dans `FileNameBean` est une propriété liée. Lorsque le nom de fichier change, `ImageViewerBean` est automatiquement prévenu et il charge le nouveau fichier.

Pour implémenter une propriété liée, vous devez mettre en œuvre deux mécanismes.

1. Chaque fois que la valeur de la propriété change, le bean doit envoyer un événement `PropertyChange` à tous les écouteurs enregistrés. Cette modification peut se produire lorsque la méthode `set` est appelée, ou lorsque l'utilisateur du programme exécute une action, telle que l'édition d'un texte ou la sélection d'un fichier.
2. Pour permettre aux écouteurs intéressés de s'enregistrer, le bean doit implémenter les deux méthodes suivantes :

```
void addPropertyChangeListener(PropertyChangeListener
    listener)
void removePropertyChangeListener(PropertyChangeListener
    listener)
```


458 *Au cœur de Java 2 - Fonctions avancées*

Le package `java.beans` fournit une classe, appelée `PropertyChangeSupport`, conçue pour la gestion des écouteurs. Pour pouvoir utiliser cette classe, votre bean doit avoir un champ de données de cette classe se présentant ainsi :

```
private PropertyChangeSupport changeSupport
    = new PropertyChangeSupport(this);
```

Vous déléguez la tâche d'ajouter et de supprimer les écouteurs de changement de propriété à cet objet.

```
public void addPropertyChangeListener(PropertyChangeListener
    listener)
{
    changeSupport.addPropertyChangeListener(listener);
}

public void removePropertyChangeListener(PropertyChangeListener
    listener)
{
    changeSupport.removePropertyChangeListener(listener);
}
```

Chaque fois que la valeur de la propriété change, utilisez la méthode `firePropertyChange` de l'objet `PropertyChangeSupport` pour envoyer un événement à tous les écouteurs enregistrés. Cette méthode a trois paramètres : le nom de la propriété, l'ancienne valeur et la nouvelle valeur. Par exemple :

```
changeSupport.firePropertyChange("fileName",
    oldValue, newValue);
```

Les valeurs doivent être des objets. Si le type de propriété n'est pas un objet, vous devez utiliser une enveloppe (objet wrapper). Par exemple :

```
changeSupport.firePropertyChange("running", false, true);
```

ASTUCE

Si votre bean étend une quelconque classe Swing qui, en fin de compte, étend la classe `JComponent`, vous n'avez pas besoin d'implémenter les méthodes `addPropertyChangeListener` et `removePropertyChangeListener`. Ces méthodes sont déjà implémentées dans la superclasse `JComponent`.

Pour signaler un changement de valeur de propriété aux écouteurs, appelez simplement la méthode `firePropertyChange` de la superclasse `JComponent` :

```
firePropertyChange("propertyName", oldValue, newValue);
```

Pour vous faciliter la tâche, cette méthode est surchargée pour les types `boolean`, `byte`, `char`, `double`, `float`, `int`, `long` et `short`. Si `oldValue` et `newValue` appartiennent à ces types, vous n'avez pas besoin d'utiliser d'enveloppes (wrappers).

Les autres beans qui veulent être avertis lorsque la valeur de propriété change doivent implémenter l'interface de `PropertyChangeListener`. Cette interface ne contient qu'une méthode :

```
void propertyChange(PropertyChangeEvent event)
```

L'exécution de la méthode `propertyChange` est déclenchée chaque fois que la valeur de la propriété change, en admettant bien sûr que vous avez ajouté le destinataire aux écouteurs de changement de propriété du bean qui génère l'événement. L'objet `PropertyChangeEvent` encapsule l'ancienne et la nouvelle valeur de propriété, récupérables par

```
Object oldValue = event.getOldValue();
Object newValue = event.getNewValue();
```

Si le type de propriété n'est pas un type de classe, les objets renvoyés sont les types enveloppe habituels. Par exemple, si une propriété de type `boolean` est modifiée, un `Boolean` est renvoyé et vous devez extraire la valeur booléenne à l'aide de la méthode `booleanValue`.

Par conséquent, un objet écouteur doit respecter ce modèle :

```
class Listener
{
    public Listener()
    {
        bean.addPropertyChangeListener(new
            PropertyChangeListener()
            {
                void propertyChange(PropertyChangeEvent event)
                {
                    Object newValue = event.getNewValue();
                    . . .
                }
            });
    }
    . . .
}
```

Propriétés contraintes

Une *propriété contrainte* est contrainte par le fait que *n'importe quel* écouteur peut opposer son *veto* aux modifications proposées, l'obligeant ainsi à revenir aux anciens paramètres. La bibliothèque Java ne contient que quelques exemples de propriétés contraintes, dont la propriété `closed` de la classe `JInternalFrame`. Si l'on tente d'appeler `setClosed(true)` sur une fenêtre interne, tous ses `VetoableChangeListener` en sont avertis. Si l'un d'entre eux déclenche une `PropertyVetoException`, la propriété `closed` n'est pas modifiée et la méthode `setClosed` déclenche la même exception. Par exemple, un `VetoableChangeListener` peut refuser de fermer la fenêtre si son contenu n'a pas été enregistré.

Pour construire une propriété contrainte, votre bean doit avoir les deux méthodes suivantes pour gérer les objets `VetoableChangeListener` :

```
public void addVetoableChangeListener(VetoableChangeListener listener);
public void removeVetoableChangeListener(
    VetoableChangeListener listener);
```

Il existe une classe commode pour gérer les écouteurs de changement de propriété, de même qu'il existe une classe appelée `VetoableChangeSupport` qui gère les écouteurs de changements pouvant être refusés. Votre bean doit contenir un objet de cette classe.

```
private VetoableChangeSupport vetoSupport =
    new VetoableChangeSupport(this);
```

460 Au cœur de Java 2 - Fonctions avancées

L'ajout et la suppression d'écouteurs doivent être délégués à cet objet. Par exemple :

```
public void addVetoableChangeListener(VetoableChangeListener listener)
{
    vetoSupport.addVetoableChangeListener(listener);
}
public void removeVetoableChangeListener(VetoableChangeListener listener)
{
    vetoSupport.removeVetoableChangeListener(listener);
}
```

ASTUCE

La classe `JComponent` assure une légère prise en charge des propriétés contraintes, mais elle n'est pas aussi large que pour les propriétés liées. La classe `JComponent` conserve une seule liste d'écouteurs pour les écouteurs de changements annulables, et non une liste pour chaque propriété. De plus, la méthode `fireVetoableChange` n'est pas surchargée pour les types de base. Si votre bean étend `JComponent` et possède une seule propriété contrainte, la prise en charge de l'écouteur de la superclasse `JComponent` convient tout à fait et vous n'avez pas besoin d'un objet `VetoableChangeSupport` séparé.

Pour actualiser une valeur de propriété contrainte, un bean utilise l'approche suivante en trois étapes :

1. Avertir tous les écouteurs de changements annulables de l'intention de modifier la valeur de la propriété (utiliser la méthode `fireVetoableChange` de la classe `VetoableChangeSupport`).
2. Lorsque aucun des écouteurs de changements annulables n'a déclenché de `PropertyVetoException`, actualiser la valeur de la propriété.
3. Avertir tous les écouteurs de changement de propriété pour qu'ils confirment qu'un changement a eu lieu.

Par exemple :

```
public void setValue(Type newValue) throws PropertyVetoException
{
    Type oldValue = getValue();
    vetoSupport.fireVetoableChange("value", oldValue, newValue);
    // a survécu, donc pas d'annulation
    value = newValue;
    changeSupport.firePropertyChange("value", oldValue, newValue);
}
```

Il est important que vous ne modifiiez pas la valeur de la propriété tant que les écouteurs enregistrés n'ont pas accepté le changement proposé. A l'inverse, un écouteur de changements annulable ne doit jamais supposer qu'un changement qu'il accepte survient réellement. La seule manière fiable d'être averti d'un changement passe par un écouteur de changement de propriété.

Nous terminerons notre discussion sur les propriétés des JavaBeans en présentant le code de `FileNameBean` (voir Exemple 6.2). Il possède une propriété `fileName` contrainte. Puisque `FileNameBean` étend la classe `JPanel`, il n'était pas nécessaire d'utiliser explicitement l'objet `PropertyChangeSupport`. Nous tirons parti, dans ce cas, de la capacité de la classe `JPanel` à gérer les écouteurs de changement de propriété.

Exemple 6.2 : FileNameBean.java

```
package com.horstmann.corejava;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.io.*;
import javax.swing.*;

/**
 * Un bean permettant de spécifier des noms de fichiers.
 */
public class FileNameBean extends JPanel
{
    public FileNameBean()
    {
        dialogButton = new JButton("...");
        nameField = new JTextField(30);

        chooser = new JFileChooser();

        chooser.setFileFilter(new
            javax.swing.filechooser.FileFilter()
            {
                public boolean accept(File f)
                {
                    String name = f.getName().toLowerCase();
                    return name.endsWith(".") + defaultExtension)
                        || f.isDirectory();
                }
                public String getDescription()
                {
                    return defaultExtension + " files";
                }
            });

        setLayout(new GridBagLayout());
        GridBagConstraints gbc = new GridBagConstraints();
        gbc.weightx = 100;
        gbc.weighty = 100;
        gbc.anchor = GridBagConstraints.WEST;
        gbc.fill = GridBagConstraints.BOTH;
        gbc.gridwidth = 1;
        gbc.gridheight = 1;
        add(nameField, gbc);

        dialogButton.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    int r = chooser.showOpenDialog(null);
                    if(r == JFileChooser.APPROVE_OPTION)
                    {
                        File f = chooser.getSelectedFile();
                        try
```

462 *Au cœur de Java 2 - Fonctions avancées*

```

        {
            String name = f.getCanonicalPath();
            setFileName(name);
        }
        catch (IOException e)
        {
        }
    }
}
});
nameField.setEditable(false);
gbc.weightx = 0;
gbc.anchor = GridBagConstraints.EAST;
gbc.fill = GridBagConstraints.NONE;
gbc.gridx = 1;
add(dialogButton, gbc);
}

/**
 * Définit la propriété fileName.
 * @param newValue le nouveau nom de fichier
 */
public void setFileName(String newValue)
{
    String oldValue = nameField.getText();
    nameField.setText(newValue);
    firePropertyChange("fileName", oldValue, newValue);
}

/**
 * Récupère la propriété fileName.
 * @return le nom du fichier sélectionné
 */
public String getFileName()
{
    return nameField.getText();
}

/**
 * Définit la propriété defaultExtension.
 * @param s la nouvelle extension par défaut
 */
public void setDefaultExtension(String s)
{
    defaultExtension = s;
}

/**
 * Récupère la propriété defaultExtension.
 * @return l'extension par défaut du sélecteur de fichier
 */
public String getDefaultExtension()
{
    return defaultExtension;
}

```

```

    public Dimension getPreferredSize()
    {
        return new Dimension(XPREFSIZE, YPREFSIZE);
    }

    private static final int XPREFSIZE = 200;
    private static final int YPREFSIZE = 20;
    private JButton dialogButton;
    private JTextField nameField;
    private JFileChooser chooser;
    private String defaultExtension = "gif";
}

```

java.beans.PropertyChangeListener 1.1

- `void propertyChange(PropertyChangeEvent event)`
Est appelé lorsqu'un événement de changement de propriété est déclenché.
Paramètres : `event` l'événement de changement de propriété

java.beans.PropertyChangeSupport 1.1

- `PropertyChangeSupport(Object sourceBean)`
Construit l'objet `PropertyChangeSupport` qui gère les écouteurs pour les changements de propriétés liées du bean donné.
- `void addPropertyChangeListener(PropertyChangeListener listener)`
- `void addPropertyChangeListener(String propertyName, PropertyChangeListener listener)` 1.2
Enregistrent un écouteur intéressé pour les changements de toutes les propriétés liées ou uniquement pour la propriété liée nommée.
- `void removePropertyChangeListener(PropertyChangeListener listener)`
- `void removePropertyChangeListener(String propertyName, PropertyChangeListener listener)` 1.2
Suppriment un écouteur préalablement enregistré de changement de propriété.
- `void firePropertyChange(String propertyName, Object oldValue, Object newValue)`
- `void firePropertyChange(String propertyName, int oldValue, int newValue)` 1.2
- `void firePropertyChange(String propertyName, boolean oldValue, boolean newValue)` 1.2
Envoient un `PropertyChangeEvent` aux écouteurs enregistrés.
- `void fireIndexedPropertyChange(String propertyName, int index, Object oldValue, Object newValue)` 5.0
- `void fireIndexedPropertyChange(String propertyName, int index, int oldValue, int newValue)` 5.0
- `void fireIndexedPropertyChange(String propertyName, int index, boolean oldValue, boolean newValue)` 5.0
Envoient un `IndexedPropertyChangeEvent` aux écouteurs enregistrés.
- `PropertyChangeListener[] getPropertyChangeListeners()` 1.4
- `PropertyChangeListener[] getPropertyChangeListeners(String propertyName)` 1.4
Récupèrent les écouteurs pour les changements dans toutes les propriétés liées ou uniquement la propriété liée nommée.

464 Au cœur de Java 2 - Fonctions avancées**java.beans.PropertyChangeEvent 1.1**

- `PropertyChangeEvent(Object sourceBean, String propertyName, Object oldValue, Object newValue)`
Construit un nouvel objet `PropertyChangeEvent`, indiquant que la propriété donnée est passée de `oldValue` à `newValue`.
- `Object getNewValue()`
Renvoie la nouvelle valeur de la propriété.
- `Object getOldValue()`
Renvoie la précédente valeur de la propriété.
- `String getPropertyName()`
Renvoie le nom de la propriété.

java.beans.IndexedPropertyChangeEvent 5.0

- `IndexedPropertyChangeEvent(Object sourceBean, String propertyName, int index, Object oldValue, Object newValue)`
Construit un nouvel objet `IndexedPropertyChangeEvent`, indiquant que la propriété donnée est passée de `oldValue` à `newValue` à l'indice donné.
- `int getIndex()`
Renvoie l'indice auquel est survenu le changement.

java.beans.VetoableChangeListener 1.1

- `void vetoableChange(PropertyChangeEvent event)`
Est appelé lorsqu'une propriété est sur le point de changer. Doit déclencher une `PropertyVetoException` si le changement n'est pas acceptable.
Paramètres : `event` L'objet événement décrivant le changement de propriété.

java.beans.VetoableChangeSupport 1.1

- `VetoableChangeSupport(Object sourceBean)`
Construit un objet `PropertyChangeSupport` qui gère les écouteurs pour les changements de propriétés contraintes du bean donné.
- `void addVetoableChangeListener(VetoableChangeListener listener)`
- `void addVetoableChangeListener(String propertyName, VetoableChangeListener listener)` 1.2
Enregistrent un écouteur intéressé pour les changements de toutes les propriétés contraintes ou uniquement pour celle nommée.
- `void removeVetoableChangeListener(VetoableChangeListener listener)`
- `void removeVetoableChangeListener(String propertyName, VetoableChangeListener listener)` 1.2
Suppriment un écouteur de changements annulable, précédemment enregistré.
- `void fireVetoableChange(String propertyName, Object oldValue, Object newValue)`
- `void fireVetoableChange(String propertyName, int oldValue, int newValue)` 1.2
- `void fireVetoableChange(String propertyName, boolean oldValue, boolean newValue)` 1.2
Envioient un `VetoableChangeEvent` aux écouteurs enregistrés.

- `VetoableChangeListener[] getVetoableChangeListeners()` 1.4
- `VetoableChangeListener[] getVetoableChangeListeners(String propertyName)` 1.4
Récupèrent les écouteurs pour les changements de toutes les propriétés contraintes ou uniquement pour la propriété liée nommée.

javax.swing.JComponent 1.2

- `void addPropertyChangeListener(PropertyChangeListener listener)`
- `void addPropertyChangeListener(String propertyName, PropertyChangeListener listener)`
Enregistrent un écouteur intéressé pour les changements de toutes les propriétés liées ou uniquement pour celle nommée.
- `void removePropertyChangeListener(PropertyChangeListener listener)`
- `void removePropertyChangeListener(String propertyName, PropertyChangeListener listener)` 1.2
Suppriment un écouteur de changement de propriété préalablement enregistré.
- `void firePropertyChange(String propertyName, Object oldValue, Object newValue)`
Envoie un événement `PropertyChangeEvent` aux écouteurs enregistrés.
- `void addVetoableChangeListener(VetoableChangeListener listener)`
Enregistre un écouteur intéressé pour toutes les propriétés contraintes ou uniquement celle nommée.
- `void removeVetoableChangeListener(VetoableChangeListener listener)`
Supprime un écouteur de changements annulable préalablement enregistré.
- `void fireVetoableChange(String propertyName, Object oldValue, Object newValue)`
Envoie un `PropertyChangeEvent` aux écouteurs enregistrés.

java.beans.PropertyVetoException 1.1

- `PropertyVetoException(String reason, PropertyChangeEvent event)`
Crée une nouvelle `PropertyVetoException`.

<i>Paramètres :</i>	<code>reason</code>	une chaîne décrivant la raison du veto
	<code>event</code>	<code>PropertyChangeEvent</code> pour la propriété contrainte à laquelle vous voulez opposer un veto
- `PropertyChangeEvent getPropertyChangeEvent()`
Renvoie le `PropertyChangeEvent` utilisé pour construire l'exception.

Les classes BeanInfo

Nous avons vu que, si vous utilisez les modèles de noms standard pour les méthodes de votre classe de bean, un outil de génération peut utiliser la réflexion pour déterminer certaines fonctionnalités comme les propriétés et les événements. Cette procédure simplifie la mise en route de la programmation des beans, même si les motifs de nom sont plutôt restrictifs. Lorsque vos beans vont se compliquer, il est possible que certaines de leurs fonctions n'apparaissent pas. De plus, comme indiqué, de nombreux beans possèdent des paires de méthodes `get/set` qui ne doivent *pas* correspondre aux propriétés de bean.

466 *Au cœur de Java 2 - Fonctions avancées*

Par chance, la spécification JavaBeans propose un mécanisme bien plus flexible et puissant pour conserver les informations nécessaires à un outil de génération. Vous pouvez dès lors définir un objet qui implémente l'interface `BeanInfo` pour décrire votre bean. Lorsque vous implémentez cette interface, un outil de génération étudiera ses méthodes pour s'informer sur les fonctions prises en charge.

Bien qu'il soit possible d'utiliser une classe `BeanInfo` pour éviter les modèles de nom, vous devrez tout de même suivre un modèle de nom pour associer l'objet `BeanInfo` au bean : pour former le nom de la classe d'info, ajoutez `BeanInfo` au nom du bean. Par exemple, la classe d'info sur le bean associée à la classe `ImageViewerBean` doit s'appeler `ImageViewerBeanBeanInfo`. Elle doit également faire partie du même package que le bean.

Généralement, vous n'écrirez pas une classe qui implémente toutes les méthodes de l'interface `BeanInfo`. Vous étendrez plutôt la classe `SimpleBeanInfo`, qui possède des implémentations par défaut pour toutes les méthodes de l'interface `BeanInfo`.

La raison la plus habituelle de fournir une classe `BeanInfo` est de vouloir obtenir le contrôle des propriétés de bean. Pour construire un `PropertyDescriptor` pour chaque propriété, fournissez le nom de la propriété et la classe du bean qui la contient.

```
PropertyDescriptor descriptor = new PropertyDescriptor("fileName",
    ImageViewerBean.class);
```

Implémentez ensuite la méthode `getPropertyDescriptors` de votre classe `BeanInfo` pour renvoyer un tableau de tous les descripteurs de propriétés.

Supposons par exemple qu'`ImageViewerBean` veuille masquer toutes les propriétés qu'il hérite de la superclasse `JLabel` et qu'il ne montre que la propriété `fileName`. La classe `BeanInfo` suivante se contente de faire cela :

```
// classe beanInfo pour ImageViewerBean
class ImageViewerBeanBeanInfo extends SimpleBeanInfo
{
    public PropertyDescriptor[] getPropertyDescriptors()
    {
        return new PropertyDescriptor[]
        {
            new PropertyDescriptor("fileName", ImageViewerBean.class);
        };
    }
}
```

D'autres méthodes renvoient également les tableaux `EventSetDescriptor` et `MethodDescriptor`, mais elles sont moins usitées. Si l'une de ces méthodes renvoie `null` (comme pour les méthodes `SimpleBeanInfo`), les modèles de noms standard s'appliqueront. Toutefois, si vous surchargez une méthode pour qu'elle renvoie un tableau non nul, vous devez inclure toutes les propriétés, les événements ou les méthodes dans votre tableau.

INFO

Vous voudrez quelquefois écrire un code générique permettant de découvrir des propriétés ou des événements d'un bean arbitraire. Appelez la méthode statique `getBeanInfo` de la classe `Introspector`. Cette dernière construit une classe `BeanInfo` qui décrit totalement le bean, en prenant en compte les informations des classes compagnon de `BeanInfo`.

L'interface `BeanInfo` contient une autre méthode utile, la méthode `getIcon`, qui permet de donner une icône personnalisée à un bean. Les outils de génération afficheront l'icône dans une palette. En fait, vous pouvez spécifier quatre bitmaps d'icône. L'interface `BeanInfo` possède quatre constantes qui couvrent les formats standard :

```
ICON_COLOR_16x16
ICON_COLOR_32x32
ICON_MONO_16x16
ICON_MONO_32x32
```

Voici un exemple de la manière d'utiliser la méthode `loadImage` dans la classe `SimpleBeanInfo` pour ajouter une icône à une classe :

```
public Image getIcon(int iconType)
{
    String name = "";
    if (iconType == BeanInfo.ICON_COLOR_16x16) name = "COLOR_16x16";
    else if (iconType == BeanInfo.ICON_COLOR_32x32) name = "COLOR_32x32";
    else if (iconType == BeanInfo.ICON_MONO_16x16) name = "MONO_16x16";
    else if (iconType == BeanInfo.ICON_MONO_32x32) name = "MONO_32x32";
    else return null;
    return loadImage("ImageViewerBean_" + name + ".gif");
}
```

Ce code fonctionne à condition que vous nommiez intelligemment les fichiers image :

```
ImageViewerBean_COLOR_16x16.gif
ImageViewerBean_COLOR_32x32.gif
```

etc.

java.beans.Introspector 1.1

- `static BeanInfo getBeanInfo(Class<?> beanClass)`
Récupère les informations de bean de la classe donnée.

java.beans.BeanInfo 1.1

- `EventSetDescriptor[] getEventSetDescriptors()`
- `MethodDescriptor[] getMethodDescriptors()`
- `PropertyDescriptor[] getPropertyDescriptors()`

Renvoient un tableau des objets descripteurs spécifiés. Un retour `null` signale le générateur qui utilise les conventions de noms et la réflexion pour trouver le membre. La méthode `getPropertyDescriptors` renvoie un mélange de descripteurs de propriétés indexés et bruts. Utilisez `instanceof` pour vérifier si un `PropertyDescriptor` spécifique est un `IndexedPropertyDescriptor`.

- `Image getIcon(int iconType)`
Renvoie un objet image pouvant être utilisé pour représenter le bean dans les boîtes à outils, barres d'outils et autres. Il y a quatre constantes, comme indiqué plus haut, pour les types d'icônes standard.
- `int getDefaultEventIndex()`

468 *Au cœur de Java 2 - Fonctions avancées*

- `int getDefaultPropertyIndex()`

Un bean peut avoir une propriété ou un événement par défaut. Ces deux méthodes renvoient l'index du tableau qui spécifie quel élément du tableau de descripteurs doit être utilisé comme membre par défaut ou -1 s'il n'y a pas de valeur par défaut. Un environnement générateur de bean peut signaler visuellement la fonctionnalité par défaut, par exemple, en la plaçant en premier dans une liste des fonctionnalités ou en inscrivant son nom en caractères gras.

- `BeanInfo[] getAdditionalBeanInfo()`

Renvoie un tableau d'objets `BeanInfo` ou la valeur `null`. Utilisez cette méthode si vous voulez que des informations concernant votre bean proviennent des classes `BeanInfo` pour d'autres beans. Vous pouvez par exemple utiliser cette méthode si votre bean a agrégé quantité d'autres beans. C'est la classe `BeanInfo` courante qui prend le pas en cas de conflit.

java.beans.SimpleBeanInfo 1.1

- `Image loadImage(String resourceName)`

Renvoie un fichier d'objet image associé à la ressource. Actuellement, seul le format GIF est pris en charge.

Paramètres : `resourceName` un chemin de fichier (relatif au répertoire contenant la classe courante)

java.beans.FeatureDescriptor 1.1

- `String getName()`

- `void setName(String name)`

Définissent ou récupèrent le nom, du point de vue programmation, pour la fonctionnalité.

- `String getDisplayName()`

- `void setDisplayName(String displayName)`

Renvoient ou récupèrent un nom d'affichage pour la fonctionnalité. La valeur par défaut est celle renvoyée par `getName`. Cependant, il n'existe pas actuellement de prise en charge explicite pour la fourniture de noms de fonctionnalités en fonction de différents paramètres régionaux.

- `String getShortDescription()`

- `void setShortDescription(String text)`

Renvoient ou définissent une chaîne qu'un outil de génération peut utiliser pour fournir une courte description pour cette fonctionnalité. La valeur par défaut est celle renvoyée par `getDisplayName`.

- `Object getValue(String attributeName)`

- `void setValue(String attributeName, Object value)`

Récupèrent ou définissent une valeur nommée associée à cette fonctionnalité.

- `Enumeration attributeNames()`

Renvoie un objet énumération contenant les noms de tous les attributs enregistrés avec `setValue`.

- `boolean isExpert()`

- `void setExpert(boolean b)`
Récupèrent ou définissent un indicateur expert qu'un générateur peut utiliser pour déterminer s'il faut masquer la fonction à un utilisateur naïf (cette fonctionnalité n'est pas prise en charge par tous les générateurs).
- `boolean isHidden()`
- `void setHidden(boolean b)`
Récupèrent ou définissent un indicateur stipulant qu'un générateur ne doit pas voir cette fonctionnalité.

java.beans.PropertyDescriptor 1.1

- `PropertyDescriptor(String propertyName, Class<?> beanClass)`
- `PropertyDescriptor(String propertyName, Class<?> beanClass, String getMethod, String setMethod)`
Construisent un objet `PropertyDescriptor`. Les méthodes déclenchent une `IntrospectionException` si une erreur est survenue lors de l'introspection. Le premier constructeur suppose que vous respectiez la convention standard pour les noms des méthodes `get` et `set`.
- `Class<?> getPropertyType()`
Renvoie un objet `Class` pour le type de propriété.
- `Method getReadMethod()`
Renvoie la méthode `get`.
- `Method getWriteMethod()`
Renvoie la méthode `set`.
- `boolean isBound()`
- `void setBound(boolean b)`
Récupèrent ou définissent un indicateur qui détermine si cette propriété est liée.
- `boolean isConstrained()`
- `void setConstrained(boolean b)`
Récupèrent ou définissent un indicateur qui détermine si cette propriété est contrainte.

java.beans.IndexedPropertyDescriptor 1.1

- `IndexedPropertyDescriptor(String propertyName, Class<?> beanClass)`
- `IndexedPropertyDescriptor(String propertyName, Class<?> beanClass, String getMethod,`
`String setMethod, String indexedGetMethod, String indexedSetMethod)`
Construisent un `IndexedPropertyDescriptor` pour la propriété index. Les méthodes lancent une `IntrospectionException` si une erreur se produit pendant l'introspection. Le premier constructeur suppose que vous respectiez la convention standard pour les noms des méthodes `get` et `set`.
- `Class<?> getIndexedPropertyType()`
Renvoie la classe qui décrit le type des valeurs indexées de la propriété, c'est-à-dire le type de valeur renvoyé par la méthode `get` indexée.

470 *Au cœur de Java 2 - Fonctions avancées*

- Method `getIndexedReadMethod()`
Renvoie la méthode get indexée.
- Method `getIndexedWriteMethod()`
Renvoie la méthode set indexée.

java.beans.EventSetDescriptor 1.1

- EventSetDescriptor(Class<?> sourceClass, String eventSetName, Class<?> listener, String listenerMethod)
Construit un EventSetDescriptor. Ce constructeur suppose que vous respectiez le modèle standard pour les noms de classes d'événement et ceux des méthodes permettant d'ajouter ou de supprimer des écouteurs d'événement. Lance une exception `IntrospectionException` si une erreur se produit pendant l'introspection.
- EventSetDescriptor(Class<?> sourceClass, String eventSetName, Class<?> listener, String[] listenerMethods, String addListenerMethod, String removeListenerMethod)
Construit un EventSetDescriptor avec plusieurs méthodes d'écouteurs et méthodes personnalisées pour ajouter et supprimer des écouteurs. Lance une exception `IntrospectionException` si une erreur se produit pendant l'introspection.
- Method `getAddListenerMethod()`
Renvoie la méthode utilisée pour enregistrer l'écouteur.
- Method `getRemoveListenerMethod()`
Renvoie la méthode utilisée pour supprimer un écouteur enregistré pour l'événement.
- Method[] `getListenerMethods()`
- MethodDescriptor[] `getListenerMethodDescriptors()`
Renvoient un tableau d'objets `Method` ou `MethodDescriptor` pour les méthodes déclenchées dans l'interface de l'écouteur.
- Class<?> `getListenerType()`
Renvoie le type de l'interface de l'écouteur associé à l'événement.
- boolean `isUnicast()`
- void `setUnicast(boolean b)`
Récupèrent ou définissent un indicateur qui vaut `true` si cet événement ne peut être diffusé qu'à un écouteur.

Les éditeurs de propriétés

Si vous ajoutez une propriété de type entier ou chaîne (*integer* ou *string*) à un bean, cette propriété est automatiquement affichée dans la feuille de propriétés du bean. Mais que se passe-t-il si vous ajoutez une propriété dont les valeurs ne peuvent pas être facilement éditées dans un champ texte, une date ou une couleur par exemple ? Vous devez alors fournir un composant séparé que l'utilisateur pourra utiliser pour spécifier la valeur de la propriété. De tels composants sont appelés *éditeurs de propriétés*. Par exemple, un éditeur de propriétés pour un objet date pourrait être un calendrier

permettant à l'utilisateur de naviguer de mois en mois pour choisir une date. Un éditeur de propriétés pour un objet `Color` permettrait à l'utilisateur de sélectionner les composants rouge, vert et bleu de la couleur.

En réalité, NetBeans dispose déjà d'un éditeur de propriétés pour les couleurs. Il existe bien sûr des éditeurs de propriétés pour les types de base comme `String` (un champ texte) et `boolean` (une case à cocher). Ces éditeurs de propriétés sont enregistrés avec le *gestionnaire de l'éditeur de propriétés*.

La procédure permettant de fournir un nouvel éditeur de propriétés est légèrement impliquée. Tout d'abord, vous créez une classe d'informations pour accompagner votre bean. Remplacez la méthode `getPropertyDescriptors`.

Cette méthode renvoie un tableau d'objets `PropertyDescriptor`. Vous créez un objet pour chaque propriété qui doit être affichée dans un éditeur de propriétés, *même celles pour lesquelles vous utilisez simplement l'éditeur par défaut*.

Vous construisez un `PropertyDescriptor` en fournissant le nom de la propriété et la classe du bean qui la contient.

```
PropertyDescriptor descriptor  
    = new PropertyDescriptor("titlePosition", ChartBean.class);
```

Appelez ensuite la méthode `setPropertyEditorClass` de la classe `PropertyDescriptor`.

```
descriptor.setPropertyEditorClass(TitlePositionEditor.class);
```

Vous construisez ensuite un tableau de descripteurs pour les propriétés de votre bean. Par exemple, le bean `Chart` que nous avons vu dans cette section possède cinq propriétés :

- une propriété de type `Color`, `graphColor` ;
- une propriété de type `String`, `title` ;
- une propriété de type `int`, `titlePosition` ;
- une propriété de type `double[]`, `values` ;
- une propriété de type `boolean`, `inverse`.

Le code de l'Exemple 6.3 montre la classe `ChartBeanBeanInfo` spécifiant les éditeurs pour ces propriétés. Elle réalise ceci :

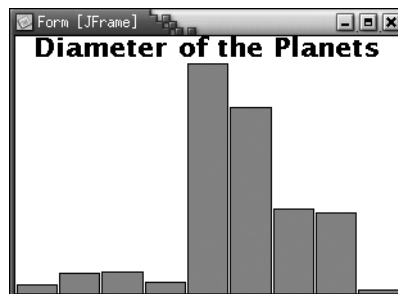
1. La méthode `getPropertyDescriptors` renvoie un descripteur pour chaque propriété. Les propriétés `title` et `graphColor` sont utilisées avec les éditeurs par défaut, à savoir les éditeurs `string` et `color` livrés avec le générateur.
2. Les propriétés `titlePosition`, `values` et `inverse` utilisent des éditeurs particuliers du type `TitlePositionEditor`, `DoubleArrayEditor` et `InverseEditor`, respectivement.

La Figure 6.10 montre le bean `Chart`. Le titre (`title`) apparaît en haut. Sa position peut être définie à `left`, `center` ou `right` (gauche, centre ou droite). La propriété `values` spécifie les valeurs du graphe. Si la propriété `inverse` a la valeur `true`, l'arrière-plan est coloré et les barres du graphe sont blanches. L'Exemple 6.4 donne le code du bean `Chart` ; ce bean est simplement une modification de l'applet histogramme dans le Volume 1, Chapitre 10.

472 Au cœur de Java 2 - Fonctions avancées

Figure 6.10

Le bean Chart.



La méthode statique `registerEditor` de la classe `PropertyEditorManager` définit un éditeur de propriétés pour toutes les propriétés d'un type donné. En voici un exemple :

```
PropertyEditorManager.registerEditor(Date.class,
    CalendarSelector.class);
```

INFO

Vous ne devriez pas appeler la méthode `registerEditor` dans vos beans, l'éditeur par défaut pour un type est un réglage global qui est en fait de la responsabilité de l'environnement de génération.

Vous pouvez utiliser la méthode `findEditor` de la classe `PropertyEditorManager` pour vérifier si un éditeur de propriétés existe pour un type donné dans votre outil de génération. Cette méthode effectue ce qui suit :

1. Elle recherche d'abord les éditeurs de propriétés qui sont déjà enregistrés avec (ce seront les éditeurs fournis par le générateur et par des appels à la méthode `registerEditor`).
2. Puis elle recherche une classe dont le nom est constitué du nom du type plus du mot `Editor`.
3. Si aucune recherche n'aboutit, `findEditor` renvoie `null`.

Par exemple, si une classe `CalendarSelector` est enregistrée pour des objets `java.util.Date`, elle serait utilisée pour modifier une propriété `Date`. Autrement, un `java.util.DateEditor` serait recherché.

Exemple 6.3 : `ChartBeanBeanInfo.java`

```
package com.horstmann.corejava;

import java.beans.*;

/**
 * L'information sur le bean pour le bean chart, indiquant les
 * éditeurs de propriété.
 */
public class ChartBeanBeanInfo extends SimpleBeanInfo
{
    public PropertyDescriptor[] getPropertyDescriptors()
    {
        try
        {
            PropertyDescriptor titlePositionDescriptor
```

```

        = new PropertyDescriptor("titlePosition", ChartBean.class);
        titlePositionDescriptor.setPropertyEditorClass(
            TitlePositionEditor.class);
        PropertyDescriptor inverseDescriptor
            = new PropertyDescriptor("inverse", ChartBean.class);
        inverseDescriptor.setPropertyEditorClass(InverseEditor.class);
        PropertyDescriptor valuesDescriptor
            = new PropertyDescriptor("values", ChartBean.class);
        valuesDescriptor.setPropertyEditorClass(
            DoubleArrayEditor.class);

        return new PropertyDescriptor[]
        {
            new PropertyDescriptor("title", ChartBean.class),
            titlePositionDescriptor,
            valuesDescriptor,
            new PropertyDescriptor("graphColor", ChartBean.class),
            inverseDescriptor
        };
    }
    catch (IntrospectionException e)
    {
        e.printStackTrace();
        return null;
    }
}

```

Exemple 6.4 : ChartBean.java

```

package com.horstmann.corejava;

import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.util.*;
import java.beans.*;
import java.io.*;
import javax.swing.*;

/**
 * Un bean pour dessiner un graphique en barres.
 */
public class ChartBean extends JPanel
{
    public void paint(Graphics g)
    {
        Graphics2D g2 = (Graphics2D)g;

        if (values == null || values.length == 0) return;
        double minValue = 0;
        double maxValue = 0;
        for (int i = 0; i < values.length; i++)
        {
            if (minValue > getValues(i)) minValue = getValues(i);
            if (maxValue < getValues(i)) maxValue = getValues(i);
        }
        if (maxValue == minValue) return;
    }
}

```


474 *Au cœur de Java 2 - Fonctions avancées*

```

Dimension d = getSize();
Rectangle2D bounds = getBounds();
double clientWidth = bounds.getWidth();
double clientHeight = bounds.getHeight();
double barWidth = clientWidth / values.length;

g2.setPaint(inverse ? color : Color.white);
g2.fill(bounds);
g2.setPaint(Color.black);

Font titleFont = new Font("SansSerif", Font.BOLD, 20);
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D titleBounds
    = titleFont.getStringBounds(title, context);

double titleWidth = titleBounds.getWidth();
double y = -titleBounds.getY();
double x;
if (titlePosition == LEFT)
    x = 0;
else if (titlePosition == CENTER)
    x = (clientWidth - titleWidth) / 2;
else
    x = clientWidth - titleWidth;

g2.setFont(titleFont);
g2.drawString(title, (float)x, (float)y);

double top = titleBounds.getHeight();
double scale = (clientHeight - top)
    / (maxValue - minValue);
y = clientHeight;

for (int i = 0; i < values.length; i++)
{
    double x1 = i * barWidth + 1;
    double y1 = top;
    double value = getValues(i);
    double height = value * scale;
    if (Value >= 0)
        y1 += (maxValue - value) * scale;
    else
    {
        y1 += (int)(maxValue * scale);
        height = -height;
    }

    g2.setPaint(inverse ? Color.white : color);
    Rectangle2D bar = new Rectangle2D.Double(x1, y1,
        barWidth - 2, height);
    g2.fill(bar);
    g2.setPaint(Color.black);
    g2.draw(bar);
}
}

```

```
/**
 * Définit la propriété de titre.
 * @param t le nouveau titre du graphique.
 */
public void setTitle(String t) { title = t; }

/**
 * Récupère la propriété du titre.
 * @return le titre du graphique
 */
public String getTitle() { return title; }

/**
 * Définit la propriété des valeurs indexées.
 * @param v les valeurs à afficher dans le graphique
 */
public void setValues(double[] v) { values = v; }

/**
 * Récupère la propriété des valeurs indexées.
 * @return les valeurs à afficher dans le graphique.
 */
public double[] getValues() { return values; }

/**
 * Définit la propriété des valeurs indexées.
 * @param i l'index de la valeur à définir
 * @param value la nouvelle valeur de l'index
 */
public void setValues(int i, double value)
{
    if (0 <= i && i < values.length) values[i] = value;
}

/**
 * Récupère la propriété des valeurs indexées.
 * @param i l'index de la valeur à obtenir
 * @return la valeur de cet index
 */
public double getValues(int i)
{
    if (0 <= i && i < values.length) return values[i];
    return 0;
}

/**
 * Définit la propriété inverse.
 * @param b true si l'affichage est inversé (barres blanches sur fond
 * coloré)
 */
public void setInverse(boolean b) { inverse = b; }

/**
 * Récupère la propriété inverse.
 * @return true si l'affichage est inversé
 */
public boolean isInverse() { return inverse; }
```

476 Au cœur de Java 2 - Fonctions avancées

```

/**
    Définit la propriété titlePosition.
    @param p LEFT, CENTER ou RIGHT
 */
public void setTitlePosition(int p) { titlePosition = p; }

/**
    Récupère la propriété titlePosition.
    @return LEFT, CENTER ou RIGHT
 */
public int getTitlePosition() { return titlePosition; }

/**
    Définit la propriété graphColor.
    @param c la couleur à utiliser pour le graphe
 */
public void setGraphColor(Color c) { color = c; }

/**
    Récupère la propriété graphColor.
    @param c la couleur à utiliser pour le graphe
 */
public Color getGraphColor() { return color; }

public Dimension getPreferredSize()
{
    return new Dimension(XPREFSIZE, YPREFSIZE);
}

private static final int LEFT = 0;
private static final int CENTER = 1;
private static final int RIGHT = 2;

private static final int XPREFSIZE = 300;
private static final int YPREFSIZE = 300;
private double[] values = { 1, 2, 3 };
private String title = "Titre";
private int titlePosition = CENTER;
private boolean inverse;
private Color color = Color.red;
}

```

java.beans.PropertyEditorManager 1.1

- static PropertyEditor findEditor(Class targetType)
Renvoie un éditeur de propriétés pour le type donné, ou null si aucun n'est enregistré.
Paramètres : targetType l'objet Class pour le type à éditer, tel que Class.Color
- static void registerEditor(Class targetType, Class editorClass)
Enregistre un éditeur de classe pour éditer les valeurs du type donné.
Paramètres : targetType l'objet Class pour le type à éditer
 editorClass l'objet Class pour l'éditeur de classe (null annulera l'enregistrement de l'éditeur courant)

java.beans.PropertyDescriptor 1.1

- `PropertyDescriptor(String name, Class beanClass)`

Construit un objet `PropertyDescriptor`.

Paramètres :

<code>name</code>	le nom de la propriété
<code>beanClass</code>	la classe du bean auquel appartient la propriété

- `void setPropertyEditorClass(Class editorClass)`

Définit la classe de l'éditeur de propriétés à utiliser avec cette propriété.

java.beans.BeanInfo 1.1

- `PropertyDescriptor[] getPropertyDescriptors()`

Renvoie un descripteur pour chaque propriété qui doit être affichée dans la feuille de propriétés pour le bean.

Ecrire un éditeur de propriétés

Avant de vous montrer comment écrire un éditeur de propriétés, il est nécessaire d'insister sur le fait que, bien que chaque éditeur de propriétés travaille avec une valeur d'un type spécifique, il peut néanmoins être assez élaboré. Par exemple, un éditeur de propriétés font (qui édite un objet de type `Font`) peut utiliser des échantillons de polices pour permettre à l'utilisateur de choisir une police d'une manière plus agréable.

Ensuite, chaque éditeur de propriétés que vous écrivez doit implémenter l'interface de `PropertyEditor`, une interface comprenant douze méthodes. Tout comme pour l'interface de `BeanInfo`, vous n'allez pas le faire manuellement. Il est beaucoup plus pratique d'étendre la classe `PropertyEditorSupport` fournie avec la bibliothèque standard. Cette classe comprend les méthodes pour ajouter et supprimer les écouteurs de changement de propriétés, et les versions par défaut de toutes les autres méthodes de l'interface de `PropertyEditor`. Par exemple, notre éditeur permettant de modifier la position du titre dans notre bean `Chart` commence ainsi :

```
// Classe PropertyEditor pour la position du titre
class TitlePositionEditor
    extends PropertyEditorSupport
{
    . . .
}
```

Notez que si une classe d'éditeur de propriétés possède un constructeur, elle doit aussi en fournir un par défaut, c'est-à-dire un constructeur ne comprenant pas d'arguments.

Enfin, avant d'entrer dans le mécanisme d'écriture d'un éditeur de propriétés, il convient de noter que c'est l'éditeur qui est sous le contrôle du générateur, et non le bean. Le générateur respecte la procédure suivante pour afficher la valeur actuelle de la propriété :

- Il instancie les éditeurs de propriétés pour chacune des propriétés du bean.
- Il demande au *bean* de lui donner la valeur actuelle de la propriété.
- Il demande ensuite à l'*éditeur de propriétés* d'afficher la valeur.

L'éditeur de propriétés peut utiliser soit des méthodes texte, soit des méthodes graphiques pour afficher la valeur. Nous reviendrons plus loin sur ces méthodes.

478 Au cœur de Java 2 - Fonctions avancées

Les éditeurs simples de propriétés

Les éditeurs simples de propriétés travaillent avec des chaînes de texte. Vous remplacez les méthodes `setAsText` et `getAsText`. Par exemple, notre bean `Chart` a une propriété vous permettant de définir l'emplacement de l'affichage du titre : à gauche, centré ou à droite. Ces choix sont implémentés en tant que constantes entières.

```
private static final int LEFT = 0;
private static final int CENTER = 1;
private static final int RIGHT = 2;
```

Mais bien sûr, nous ne souhaitons pas les voir apparaître sous la forme 0, 1, 2 dans le champ texte, à moins de vouloir concourir pour le titre de "Dinosaure de l'interface utilisateur". Nous allons plutôt définir un éditeur de propriétés dont la méthode `getAsText` va renvoyer la valeur sous forme de chaîne. La méthode appelle la méthode `getValue` de `PropertyEditorSupport` pour extraire la valeur de la propriété. Puisqu'il s'agit d'une méthode générique, la valeur est renvoyée en tant qu'objet. Si le type de la propriété est un type de base, nous devons renvoyer un objet wrapper (enveloppe). Ici, le type de propriété est `int`, et l'appel à `getValue` renvoie un entier (`Integer`).

```
class TitlePositionEditor extends PropertyEditorSupport
{
    public String getAsText()
    {
        int value = (Integer) getValue();
        return options[value];
    }
    .
    .
    private String[] options = { "Gauche", "Centré", "Droite" };
}
```

A présent, le champ de texte affiche l'un de ces champs. Lorsque l'utilisateur édite le champ de texte, cela déclenche un appel à la méthode `setAsText` pour mettre à jour la valeur de la propriété en invoquant la méthode `setValue`. Il s'agit également d'une méthode générique, dont les paramètres sont de type `Object`. Pour définir la valeur d'un type numérique, nous devons passer un objet wrapper (enveloppe).

```
public void setAsText(String s)
{
    for (int i = 0; i < options.length; i++)
    {
        if (options[i].equals(s))
        {
            setValue(i);
            return;
        }
    }
}
```

En réalité, cet éditeur de propriétés n'est pas un bon choix pour la propriété `titlePosition`, à moins bien sûr que nous ne tenions absolument à notre titre de Dinosaure en matière d'interface utilisateur. L'utilisateur peut ne pas savoir quels sont les choix autorisés. Il serait préférable d'afficher toutes les valeurs possibles (voir Figure 6.11). La classe `PropertyEditorSupport` fournit une méthode simple pour afficher les sélections dans un éditeur de propriétés. Nous écrivons simplement une méthode `getTags` qui renvoie un tableau de chaînes.

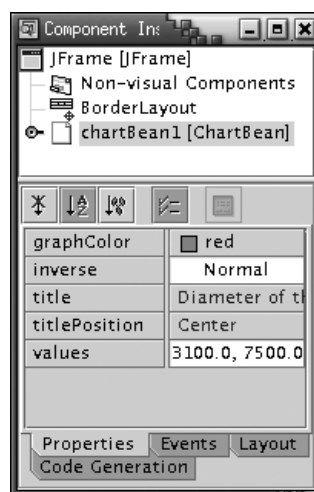
```
public String[] getTags() { return options; }
```

La méthode `getTags` par défaut renvoie `null`. En renvoyant une valeur non `null`, nous indiquons un champ avec plusieurs choix au lieu d'un champ texte.

Nous devons toujours fournir les méthodes `getAsText` et `setAsText`. La méthode `getTags` spécifie simplement les valeurs à afficher dans un menu déroulant. Les méthodes `getAsText`/`setAsText` se chargent de la traduction entre les chaînes et le type de données de la propriété (qui peut être chaîne ou entier, ou de type complètement différent).

Figure 6.11

L'éditeur de position du titre à l'action.



L'Exemple 6.5 donne le code complet de l'éditeur de propriétés.

Exemple 6.5 : TitlePositionEditor.java

```
package com.horstmann.corejava;

import java.beans.*;

/**
 * Un éditeur personnalisé pour la propriété titlePosition du
 * ChartBean. L'éditeur permet à l'utilisateur de choisir entre
 * Left, Center et Right
 */
public class TitlePositionEditor
    extends PropertyEditorSupport
{
    public String[] getTags() { return options; }
    private String[] options = { "Gauche", "Centré", "Droite" };
    public String getJavaInitializationString() { return "" + getValue();

    public String getAsText()
    {
        int value = (Integer) getValue();
        return options[value];
    }
}
```

480 Au cœur de Java 2 - Fonctions avancées

```
public void setAsText(String s)
{
    for (int i = 0; i < options.length; i++)
    {
        if (options[i].equals(s))
        {
            setValue(i);
            return;
        }
    }
}
```

java.beans.PropertyEditorSupport 1.1

- **Object getValue()**
Renvoie la valeur actuelle de la propriété. Les types de base sont enveloppés dans des classes enveloppe.
- **void setValue(Object newValue)**
Affecte une nouvelle valeur à la propriété. Les types de base doivent être enveloppés dans des enveloppes d'objet.

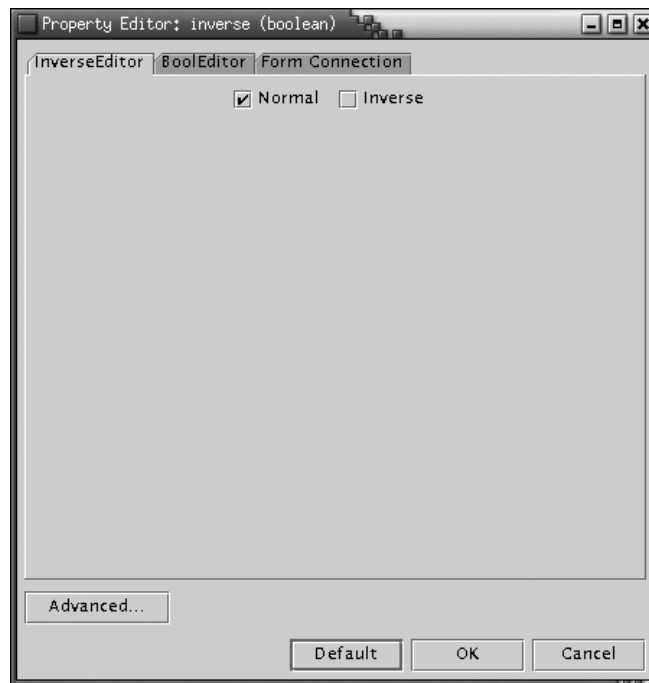
Paramètres : **newValue** la nouvelle valeur de l'objet ; doit être un objet nouvellement créé que la propriété peut posséder
- **String getAsText()**
Remplacez cette méthode afin de renvoyer une représentation sous forme de chaîne de la valeur de la propriété. Renvoie null par défaut, pour indiquer que la propriété ne peut pas être représentée sous forme de chaîne.
- **void setAsText(String text)**
Remplacez cette méthode pour affecter à la propriété une nouvelle valeur obtenue par analyse du texte. Une exception `IllegalArgumentException` peut être envoyée si le texte ne représente pas une valeur légale, ou si cette propriété ne peut pas être représentée sous forme de chaîne.
- **String[] getTags()**
Remplacez cette méthode pour renvoyer un tableau de toutes les représentations de chaîne possibles des valeurs de la propriété, afin qu'elles soient affichées dans une liste de choix. Renvoie null par défaut pour indiquer que le jeu des valeurs de chaîne n'est pas limité.

Editeurs de propriétés à interface graphique

Des types de propriétés plus sophistiqués ne peuvent être édités sous forme de texte. Ils peuvent être représentés de deux façons. La feuille de propriétés contient une petite zone (où habituellement se trouverait une zone de texte ou une liste de choix) où l'éditeur de propriétés va dessiner une représentation graphique de la valeur actuelle. Lorsque l'utilisateur clique sur cette zone, une boîte de dialogue éditeur personnalisé s'affiche (voir Figure 6.12). Cette boîte de dialogue contient un composant fourni par l'éditeur de propriétés et permettant d'éditer les valeurs de la propriété, ainsi que divers boutons, fournis par l'environnement de génération.

Figure 6.12

Une boîte de dialogue éditeur personnalisé.



Pour construire un éditeur de propriétés à interface graphique :

1. Indiquez à l'outil de génération que vous allez dessiner la valeur au lieu d'utiliser une chaîne.
2. "Dessinez" la valeur entrée par l'utilisateur dans l'interface graphique.
3. Indiquez à l'outil de génération que vous allez utiliser un éditeur de propriétés à interface graphique.
4. Construisez l'interface graphique.
5. Ecrivez le code permettant de valider l'entrée de l'utilisateur en tant que valeur.

Pour la première étape, vous remplacez la méthode `getAsText()` dans l'interface de `PropertyEditor` pour qu'elle renvoie `null`, et la méthode `isPaintable()` pour qu'elle renvoie `true`.

```
public String getAsText() { return null; }
public boolean isPaintable() { return true; }
```

Puis vous implémentez la procédure `paintValue`. Elle reçoit un handle `Graphics` et les coordonnées du rectangle dans lequel vous allez dessiner. Notez que ce rectangle est généralement petit, la représentation ne peut donc être très élaborée. Pour représenter graphiquement la propriété inverse, nous dessinons la chaîne "Inverse" en lettres blanches sur fond noir, ou la chaîne "Normal" en noir sur fond blanc (voir Figure 6.11).

```
public void paintValue(Graphics g, Rectangle box)
{
    Graphics2D g2 = (Graphics2D) g;
    boolean isInverse = (Boolean) getValue();
    String s = isInverse ? "Inverse" : "Normal";
```


482 Au cœur de Java 2 - Fonctions avancées

```

        g2.setColor(isInverse ? Color.black : Color.white);
        g2.fill(box);
        g2.setColor(isInverse ? Color.white : Color.black);
        // calcule la position de chaîne pour la centrer
        . . .
        g2.drawString(s, (float) x, (float) y);
    }

```

Bien entendu, cette représentation graphique n'est pas modifiable. L'utilisateur doit cliquer dessus pour faire apparaître un éditeur personnalisé.

Vous indiquez que vous allez avoir un éditeur personnalisé en remplaçant `supportsCustomEditor` dans l'interface de `PropertyEditor` afin de renvoyer `true`.

```
public boolean supportsCustomEditor() { return true; }
```

Vous écrivez ensuite le code qui construit le composant qui contiendra l'éditeur personnalisé. Vous devrez construire une classe éditeur personnalisé séparée pour chaque propriété. Par exemple, associée à notre classe `InverseEditor` nous avons une classe `InverseEditorPanel` (voir Exemple 6.7) qui décrit une interface graphique avec deux boutons radio pour basculer entre les modes normal et inverse. Ce code est simple. Toutefois, les actions de l'interface utilisateur graphique doivent mettre à jour les valeurs de propriété. Cela est réalisé de la façon suivante :

1. Le constructeur de l'éditeur personnalisé reçoit une référence à l'objet éditeur de propriétés et la stocke dans une variable `editor`.
2. Pour lire la valeur de la propriété, l'éditeur personnalisé appelle `editor.getValue()`.
3. Pour définir la valeur de l'objet, l'éditeur personnalisé appelle `editor.setValue(newValue)` suivi de `editor.firePropertyChange()`.

Ensuite, la méthode `getCustomEditor` de l'interface de `PropertyEditor` construit et renvoie un objet de la classe éditeur personnalisé.

```

public Component getCustomEditor()
{
    return new InverseEditorPanel(this);
}

```

Enfin, les éditeurs de propriété devraient implémenter la méthode `getJavaInitializationString`. Elle permet de donner au générateur le code Java pour la définition d'une propriété sur sa valeur actuelle. Le générateur utilise cette chaîne pour produire automatiquement le code. Voici par exemple la méthode pour `InverseEditor` :

```
public String getJavaInitializationString() { return "" + getValue(); }
```

Cette méthode renvoie la chaîne `"false"` ou `"true"`. Testez-la dans NetBeans ; si vous modifiez la propriété inverse, NetBeans insère ce code :

```
chartBean1.setInverse(true);
```

INFO

Si une propriété possède un éditeur personnalisé qui n'implémente pas la méthode `getJavaInitializationString`, NetBeans ne sait pas comment générer le code pour produire un paramètre ???.

L'Exemple 6.6 montre le code complet d'`InverseEditor`, qui affiche la configuration actuelle dans la feuille de propriétés. L'Exemple 6.7 montre le code implémentant le panneau éditeur qui apparaît.

Le code de la classe de l'éditeur de propriétés (voir Exemple 6.8) est presque identique à celui de `InverseEditor`, excepté que nous dessinons simplement une chaîne constituée des quelques premières valeurs du tableau, suivies de `...`, dans la méthode `paintValue`. Et, bien entendu, nous renvoyons un éditeur personnalisé différent dans la méthode `getCustomEditor`. Ces exemples complètent le code du bean `Chart`.

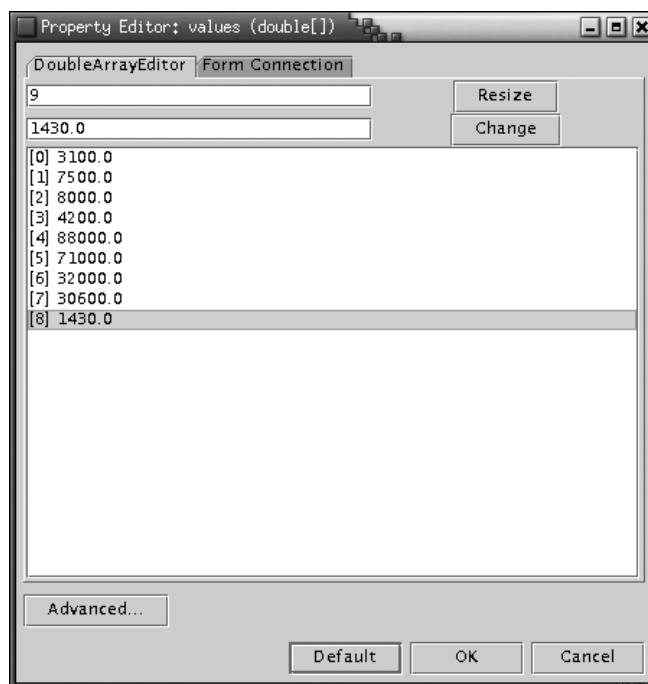
L'autre éditeur personnalisé que nous construisons pour la classe `ChartBean` vous permet d'éditer un tableau `double[]`. NetBeans ne peut éditer de propriétés tableau du tout. Nous avons développé cet éditeur personnalisé pour pallier ce manque évident. La Figure 6.13 montre l'éditeur personnalisé en action. Toutes les valeurs du tableau sont affichées dans la zone de liste, préfixées à l'aide de leur index dans le tableau. Le fait de cliquer sur une valeur du tableau place celle-ci dans le champ texte situé au-dessus, où vous pouvez la modifier. Vous avez également la possibilité de redimensionner le tableau. Le code de la classe `DoubleArrayPanel` qui implémente l'interface graphique se trouve dans l'Exemple 6.9.

INFO

Nous devons malheureusement *dessiner* les valeurs du tableau. Il serait plus pratique de renvoyer une chaîne à l'aide de la méthode `getAsText`. Toutefois, certains environnements de génération se trompent lorsque `getAsText` et `getCustomEditor` renvoient des valeurs non nulles.

Figure 6.13

La boîte de dialogue d'éditeur personnalisé pour un tableau.



484 Au cœur de Java 2 - Fonctions avancées

Exemple 6.6 : *InverseEditor.java*

```
package com.horstmann.corejava;

import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.beans.*;

/**
 * L'éditeur de propriété pour la propriété inverse du ChartBean.
 * La propriété inverse bascule entre des barres du graphique et
 * un fond coloré.
 */
public class InverseEditor extends PropertyEditorSupport
{
    public Component getCustomEditor()
    { return new InverseEditorPanel(this); }
    public boolean supportsCustomEditor() { return true; }
    public boolean isPaintable() { return true; }
    public String getAsText() { return null; }
    public String getJavaInitializationString()
    { return "" + getValue(); }

    public void paintValue(Graphics g, Rectangle box)
    {
        Graphics2D g2 = (Graphics2D)g;
        boolean isInverse = (Boolean) getValue();
        String s = isInverse ? "Inverse" : "Normal";
        g2.setPaint(isInverse ? Color.black : Color.white);
        g2.fill(box);
        g2.setPaint(isInverse ? Color.white : Color.black);
        FontRenderContext context = g2.getFontRenderContext();
        Rectangle2D stringBounds = g2.getFont().getStringBounds(
            s, context);
        double w = stringBounds.getWidth();
        double x = box.x;
        if (w < box.width) x += (box.width - w) / 2;
        double ascent = -stringBounds.getY();
        double y = box.y
            + (box.height - stringBounds.getHeight()) / 2 + ascent;
        g.drawString(s, (float) x, (float) y);
    }
}
```

Exemple 6.7 : *InverseEditorPanel.java*

```
package com.horstmann.corejava;

import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.lang.reflect.*;
import java.beans.*;
import javax.swing.*;

/**
 * Le panneau permettant de définir la propriété inverse. Il contient
 * des boutons permettant de basculer entre des couleurs normales et
 * inversées.
 */
```

```

*/
public class InverseEditorPanel extends JPanel
{
    public InverseEditorPanel(PropertyEditorSupport ed)
    {
        editor = ed;
        ButtonGroup g = new ButtonGroup();
        boolean isInverse = (Boolean) editor.getValue();
        normal = new JRadioButton("Normal", !isInverse);
        inverse = new JRadioButton("Inverse", isInverse);

        g.add(normal);
        g.add(inverse);
        add(normal);
        add(inverse);

        ActionListener buttonListener =
            new ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    editor.setValue(
                        new Boolean(inverse.isSelected()));
                    editor.firePropertyChange();
                }
            };

        normal.addActionListener(buttonListener);
        inverse.addActionListener(buttonListener);
    }

    private JRadioButton normal;
    private JRadioButton inverse;
    private PropertyEditorSupport editor;
}

```

Exemple 6.8 : DoubleArrayEditor.java

```

package com.horstmann.corejava;

import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.beans.*;

/**
    Un éditeur personnalisé pour un tableau composé de chiffres à
    virgule flottante.
*/
public class DoubleArrayEditor extends PropertyEditorSupport
{
    public Component getCustomEditor()
    { return new DoubleArrayEditorPanel(this); }
    public boolean supportsCustomEditor() { return true; }
    public boolean isPaintable() { return true; }
    public String getAsText() { return null; }

    public void paintValue(Graphics g, Rectangle box)
    {

```

486 *Au cœur de Java 2 - Fonctions avancées*

```

Graphics2D g2 = (Graphics2D) g;
double[] values = (double[]) getValue();
StringBuilder s = new StringBuilder();
for (int i = 0; i < 3; i++)
{
    if (values.length > i) s.append(values[i]);
    if (values.length > i + 1) s.append(", ");
}
if (values.length > 3) s.append("...");

g2.setPaint(Color.white);
g2.fill(box);
g2.setPaint(Color.black);
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D stringBounds = g2.getFont().getStringBounds(
    s.toString(), context);
double w = stringBounds.getWidth();
double x = box.x;
if (w < box.width) x += (box.width - w) / 2;
double ascent = -stringBounds.getY();
double y = box.y
    + (box.height - stringBounds.getHeight()) / 2 + ascent;
g2.drawString(s.toString(), (float) x, (float) y);
}

public String getJavaInitializationString()
{
    double[] values = (double[]) getValue();
    StringBuilder s = new StringBuilder();
    s.append("new double[] {");
    for (int i = 0; i < values.length; i++)
    {
        if (i > 0) s.append(", ");
        s.append(values[i]);
    }
    s.append("}");
    return s.toString();
}
}

```

Exemple 6.9 : DoubleArrayEditorPanel.java

```

package com.horstmann.corejava;

import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.lang.reflect.*;
import java.beans.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Le panneau qui se trouve dans DoubleArrayEditor. Il contient une
 * liste des valeurs du tableau, ainsi que des boutons permettant
 * de redimensionner le tableau et de modifier la valeur de liste
 * actuellement sélectionnée.
 */
public class DoubleArrayEditorPanel extends JPanel

```

```
{
    public DoubleArrayEditorPanel(PropertyEditorSupport ed)
    {
        editor = ed;
        setArray((double[])ed.getValue());

        setLayout(new GridBagLayout());

        add(sizeField, new GBC(0, 0, 1, 1).setWeight(100,
            0).setFill(GBC.HORIZONTAL));
        add(valueField, new GBC(0, 1, 1, 1, 1).setWeight(100,
            0).setFill(GBC.HORIZONTAL));
        add(sizeButton, new GBC(1, 0, 1, 1).setWeight(100, 0));
        add(valueButton, new GBC(1, 1, 1, 1).setWeight(100, 0));
        add(new JScrollPane(elementList), new GBC(0, 2, 2,
            1).setWeight(100, 100).setFill(GBC.BOTH));

        sizeButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                { changeSize(); }
            });

        valueButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                { changeValue(); }
            });

        elementList.setSelectionMode(
            ListSelectionMode.SINGLE_SELECTION);

        elementList.addListSelectionListener(new
            ListSelectionListener()
            {
                public void valueChanged(ListSelectionEvent event)
                {
                    int i = elementList.getSelectedIndex();
                    if (i < 0) return;
                    valueField.setText("" + array[i]);
                }
            });

        elementList.setModel(model);
        elementList.setSelectedIndex(0);
    }

    /**
     * Cette méthode est appelée lorsque l'utilisateur souhaite
     * modifier la taille du tableau.
     */
    public void changeSize()
    {
        fmt.setParseIntegerOnly(true);
        int s = 0;
        try
```

488 Au cœur de Java 2 - Fonctions avancées

```

    {
        s = fmt.parse(sizeField.getText()).intValue();
        if (s < 0)
            throw new ParseException("Hors limites", 0);
    }
    catch(ParseException e)
    {
        JOptionPane.showMessageDialog(this, "" + e,
            "Erreur de saisie", JOptionPane.WARNING_MESSAGE);
        sizeField.requestFocus();
        return;
    }
    if (s == array.length) return;
    setArray((double[])arrayGrow(array, s));
    editor.setValue(array);
    editor.firePropertyChange();
}

/**
 * Cette méthode est appelée lorsque l'utilisateur souhaite modifier
 * la valeur de tableau actuellement sélectionnée.
 */
public void changeValue()
{
    double v = 0;
    fmt.setParseIntegerOnly(false);
    try
    {
        v = fmt.parse(valueField.getText()).doubleValue();
    }
    catch (ParseException e)
    {
        JOptionPane.showMessageDialog(this, "" + e,
            "Erreur de saisie", JOptionPane.WARNING_MESSAGE);
        valueField.requestFocus();
        return;
    }
    int currentIndex = elementList.getSelectedIndex();
    setArray(currentIndex, v);
    editor.firePropertyChange();
}

/**
 * Définit la propriété du tableau indexé.
 * @param v le tableau à modifier
 */
public void setArray(double[] v)
{
    if (v == null) array = new double[0];
    else array = v;
    model.setArray(array);
    sizeField.setText("" + array.length);
    if (array.length > 0)
    {
        valueField.setText("" + array[0]);
        elementList.setSelectedIndex(0);
    }
    else
        valueField.setText("");
}

```

```

    }

    /**
     * Récupère la propriété du tableau indexé.
     * @return le tableau modifié
     */
    public double[] getArray()
    {
        return (double[]) array.clone();
    }

    /**
     * Définit la propriété du tableau indexé.
     * @param i l'index dont la valeur est à définir
     * @param value la nouvelle valeur pour l'index donné
     */
    public void setArray(int i, double value)
    {
        if (0 <= i && i < array.length)
        {
            model.setValue(i, value);
            elementList.setSelectedIndex(i);
            valueField.setText("" + value);
        }
    }

    /**
     * Récupère la propriété du tableau indexé.
     * @param i l'index dont la valeur est à récupérer
     * @return la valeur à l'index donné
     */
    private double getArray(int i)
    {
        if (0 <= i && i < array.length) return array[i];
        return 0;
    }

    /**
     * Redimensionne un tableau.
     * @param a le tableau à agrandir
     * @param newLength la nouvelle longueur
     * @return un tableau avec la longueur donnée et les mêmes éléments
     *         que a aux positions habituelles
     */
    private static Object arrayGrow(Object a, int newLength)
    {
        Class cl = a.getClass();
        if (!cl.isArray()) return null;
        Class componentType = a.getClass().getComponentType();
        int length = Array.getLength(a);

        Object newArray = Array.newInstance(componentType,
            newLength);
        System.arraycopy(a, 0, newArray, 0,
            Math.min(length, newLength));
        return newArray;
    }

    private PropertyEditorSupport editor;

```


490 Au cœur de Java 2 - Fonctions avancées

```

private double[] array;
private NumberFormat fmt = NumberFormat.getNumberInstance();
private JTextField sizeField = new JTextField(4);
private JTextField valueField = new JTextField(12);
private JButton sizeButton = new JButton("Redim");
private JButton valueButton = new JButton("Modif");
private JList elementList = new JList();
private DoubleArrayListModel model = new DoubleArrayListModel();
}

/**
 * Le modèle de liste pour la liste d'éléments dans l'éditeur.
 */
class DoubleArrayListModel extends AbstractListModel
{
    public int getSize() { return array.length; }

    public Object getElementAt(int i)
    { return "[" + i + "]" + array[i]; }

    /**
     * Définit un nouveau tableau à afficher dans la liste.
     * @param a le nouveau tableau
     */
    public void setArray(double[] a)
    {
        int oldLength = array == null ? 0 : array.length;
        array = a;
        int newLength = array == null ? 0 : array.length;
        if (oldLength > 0) fireIntervalRemoved(this, 0, oldLength);
        if (newLength > 0) fireIntervalAdded(this, 0, newLength);
    }

    /**
     * Modifie une valeur dans le tableau à afficher dans la liste.
     * @param i l'index dont la valeur doit changer
     * @param value la nouvelle valeur pour l'index donné
     */
    public void setValue(int i, double value)
    {
        array[i] = value;
        fireContentsChanged(this, i, i);
    }

    private double[] array;
}

```

En résumé

Pour chaque éditeur de propriétés que vous écrivez, vous devez choisir l'une des trois manières suivantes d'afficher et d'éditer la valeur de propriété :

- en tant que chaîne de texte (définissez `getAsText` et `setAsText`) ;
- en tant que liste de choix (définissez `getAsText`, `setAsText` et `getTags`) ;
- sous forme graphique, en la dessinant (définissez `isPaintable`, `paintValue`, `supportsCustomEditor` et `getCustomEditor`).

Vous avez des exemples de ces trois méthodes dans le bean Chart.

java.beans.PropertyEditorSupport 1.1

- `boolean isPaintable()`
Doit être remplacée afin de renvoyer `true` si la classe utilise la méthode `paintValue` pour afficher la propriété.
- `void paintValue(Graphics g, Rectangle box)`
Doit être remplacée pour que la valeur soit représentée graphiquement à l'emplacement spécifié, dans le composant utilisé pour la feuille de propriétés.

<i>Paramètres :</i>	<code>g</code>	l'objet graphique sur lequel on dessine
	<code>box</code>	un objet rectangle qui représente où, dans le composant feuille de propriétés, la valeur est dessinée.
- `boolean supportsCustomEditor()`
Doit être remplacée, afin de renvoyer `true` si l'éditeur de propriétés possède un éditeur personnalisé.
- `Component getCustomEditor()`
Doit être remplacée pour renvoyer le composant qui contient une interface graphique personnalisée pour l'édition de la valeur de propriété.
- `String getJavaInitializationString()`
Doit être remplacée pour renvoyer une chaîne de code Java pouvant être utilisée pour générer le code qui initialise la valeur de propriété. Par exemple `"0"`, `"new Color(64, 64, 64)"`.

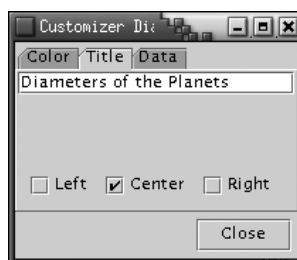
Les "customizers"

Un éditeur de propriétés, quel que soit son niveau de sophistication, ne permet à l'utilisateur de définir qu'une seule propriété à la fois. En particulier, si certaines propriétés d'un bean sont interconnectées, il peut être plus convivial de permettre aux utilisateurs d'éditer plusieurs propriétés à la fois. Pour activer cette fonctionnalité, vous fournissez un *customizer* (qui pourrait se traduire par le néologisme *personnalisateur*) au lieu — ou en plus — de plusieurs éditeurs de propriétés.

Dans l'exemple de programme de cette section, nous développons un customizer pour le bean Chart. Le customizer vous permet de définir plusieurs propriétés du bean Chart en une seule fois, et vous permet de spécifier un fichier dans lequel vous pourrez lire les points de données pour l'histogramme. La Figure 6.14 vous montre un panneau du customizer pour le bean ChartBean.

Figure 6.14

Le customizer pour ChartBean.



492 Au cœur de Java 2 - Fonctions avancées

Pour ajouter un customizer à votre bean, vous *devez* fournir une classe BeanInfo et remplacer la méthode `getBeanDescriptor`, comme dans l'exemple qui suit :

```
public ChartBean2BeanInfo extends SimpleBeanInfo
{
    public BeanDescriptor getBeanDescriptor()
    {
        return new BeanDescriptor(ChartBean2.class,
            ChartBean2Customizer.class);
    }
    ...
}
```

Notez qu'il n'est pas nécessaire de respecter un modèle quelconque de nom pour la classe du customizer. Le générateur peut, pour la localiser :

1. rechercher la classe BeanInfo associée ;
2. invoquer sa méthode `getBeanDescriptor` ;
3. appeler la méthode `getCustomizerClass`.

(L'usage est toutefois de nommer le customizer *NomBeanCustomizer*).

L'Exemple 6.10 contient le code de la classe `ChartBean2BeanInfo` qui référence le customizer `ChartBean2Customizer`. Vous verrez dans la prochaine section comment ce customizer est implémenté.

Exemple 6.10 : ChartBean2BeanInfo.java

```
package com.horstmann.corejava;

import java.awt.*;
import java.beans.*;

/**
 * L'info du bean pour le Chartbean, spécifiant les éditeurs de
 * propriétés.
 */
public class ChartBeanBeanInfo extends SimpleBeanInfo
{
    public PropertyDescriptor[] getPropertyDescriptors()
    {
        try
        {
            PropertyDescriptor titlePositionDescriptor
                = new PropertyDescriptor("titlePosition", ChartBean.class);
            titlePositionDescriptor.setPropertyEditorClass(
                TitlePositionEditor.class);
            PropertyDescriptor inverseDescriptor
                = new PropertyDescriptor("inverse", ChartBean.class);
            inverseDescriptor.setPropertyEditorClass(InverseEditor.class);
            PropertyDescriptor valuesDescriptor
                = new PropertyDescriptor("values", ChartBean.class);
            valuesDescriptor.setPropertyEditorClass(
                DoubleArrayEditor.class);

            return new PropertyDescriptor[]
            {
                new PropertyDescriptor("title", ChartBean.class),
```

```

        titlePositionDescriptor,
        valuesDescriptor,
        new PropertyDescriptor("graphColor", ChartBean.class),
        inverseDescriptor
    };
}
catch (IntrospectionException e)
{
    e.printStackTrace();
    return null;
}
}
}

```

java.beans.BeanInfo 1.1

- `PropertyDescriptor getBeanDescriptor()`
Renvoie un objet `PropertyDescriptor` décrivant les fonctionnalités du bean.

java.beans.BeanDescriptor 1.1

- `BeanDescriptor(Class beanClass, Class customizerClass)`
Construit un objet `BeanDescriptor` pour un bean possédant un customizer.
Paramètres : `beanClass` l'objet `Class` pour le bean
`customizerClass` l'objet `Class` pour le customizer du bean.
- `Class getBeanClass()`
Renvoie l'objet `Class` qui définit le bean.
- `Class getCustomizerClass()`
Renvoie l'objet `Class` qui définit le customizer du bean.

Ecrire une classe Customizer

Toute classe customizer que vous écrivez doit implémenter l'interface de `Customizer`. Il y a seulement trois méthodes dans cette interface :

- La méthode `setObject`, avec un paramètre spécifiant le bean qui est personnalisé.
- Les méthodes `addPropertyChangeListener` et `removePropertyChangeListener`, qui gèrent la collection des écouteurs notifiés lorsqu'une propriété est modifiée dans le customizer.

Il est recommandé de mettre à jour l'apparence visuelle du bean cible en émettant un événement `PropertyChangeEvent` chaque fois que l'utilisateur change l'une quelconque des valeurs de propriétés, et non uniquement lorsque l'utilisateur se trouve à la fin du processus de personnalisation.

Contrairement aux éditeurs de propriétés, les customizers ne sont pas automatiquement affichés. Dans NetBeans, vous devez sélectionner l'option de menu `Customize` pour faire apparaître un customizer. A ce point, le générateur appelle la méthode `setObject` du customizer qui prend le bean à personnaliser pour paramètre. Remarquez que votre customizer est par conséquent créé avant d'être réellement lié à une instance de votre bean. Vous ne pouvez donc présumer d'aucune information en ce qui concerne l'état d'un bean dans le customizer, et vous devez fournir un constructeur par défaut, c'est-à-dire un constructeur sans arguments.

494 *Au cœur de Java 2 - Fonctions avancées*

Il y a trois étapes dans l'écriture d'une classe customizer :

1. la construction de l'interface visuelle ;
2. l'initialisation du customizer dans la méthode `setObject` ;
3. la mise à jour du bean par le déclenchement d'événements de changement de propriété lorsque l'utilisateur modifie des propriétés dans l'interface.

Par définition, une classe customizer est visuelle. Elle doit, par conséquent, étendre `Component` ou une sous-classe de `Component`, telle que `JPanel`. Les customizers proposant généralement plusieurs options à l'utilisateur, il est souvent commode d'utiliser l'interface de panneau à onglets. Nous utilisons cette approche et étendons l'interface de `JTabbedPane` pour le customizer.

Le customizer réunit les informations suivantes dans trois panneaux :

- couleur du graphisme et mode vidéo inverse ;
- titre et position du titre ;
- points de données.

Le développement de ce type d'interface utilisateur peut être assez fastidieux à coder — dans notre exemple, plus de 100 lignes sont consacrées simplement à sa définition dans le constructeur. Cette tâche ne requiert toutefois que les techniques de programmation Swing habituelles, et nous n'allons donc pas entrer ici dans les détails.

Il existe une astuce qu'il est bon de retenir. Vous aurez souvent besoin d'éditer des valeurs de propriétés dans un customizer. Au lieu d'implémenter une nouvelle interface pour définir la valeur de propriété d'une classe particulière, vous pouvez simplement repérer un éditeur de propriétés existant et l'ajouter à votre interface utilisateur ! Par exemple, dans notre customizer `Chart2Bean`, nous devons définir la couleur du graphisme. Puisque nous savons que la `BeanBox` dispose d'un excellent éditeur de propriétés pour les couleurs, nous le recherchons de la façon suivante :

```
PropertyEditor colorEditor  
    = PropertyEditorManager.findEditor(Color.class);
```

Nous appelons ensuite `getCustomEditor` pour récupérer le composant qui contient l'interface utilisateur pour la définition des couleurs.

```
Component colorEditorComponent = colorEditor.getCustomEditor();  
// Ajouter maintenant ce composant à l'interface utilisateur
```

Lorsque nous avons tous nos composants, nous initialisons leurs valeurs dans la méthode `setObject`. La méthode `setObject` est appelée lorsque le customizer est affiché. Son paramètre est le bean en cours de personnalisation. Nous stockons ensuite cette référence de bean ; nous en aurons besoin plus tard pour notifier au bean les changements de propriété. Puis nous initialisons chaque composant de l'interface utilisateur. Voici une partie de la méthode `setObject` du customizer du bean `Chart` qui réalise cette initialisation :

```
public void setObject(Object obj)  
{  
    bean = (ChartBean2)obj;  
    titleField.setText(bean.getTitle());  
    colorEditor.setValue(bean.getGraphColor());  
    . . .  
}
```

Enfin, nous attachons des gestionnaires d'événement pour garder la trace des activités de l'utilisateur. Chaque fois que l'utilisateur change la valeur d'un composant, le composant déclenche un événement que notre customizer doit gérer. Le gestionnaire d'événements doit mettre à jour la valeur de la propriété dans le bean et également déclencher un événement `PropertyChangeEvent` afin que les autres écouteurs (tels que la feuille de propriétés) puissent être mis à jour. Suivons ce processus avec une paire d'éléments d'interface utilisateur dans le customizer du bean `Chart`.

Lorsque l'utilisateur tape un nouveau titre, la propriété `title` doit être mise à jour. Nous attachons un `DocumentListener` au champ de texte dans lequel l'utilisateur saisit le titre.

```
titleField.getDocument().addDocumentListener(new
    DocumentListener()
    {
        public void changedUpdate(DocumentEvent event)
        {
            setTitle(titleField.getText());
        }
        public void insertUpdate(DocumentEvent event)
        {
            setTitle(titleField.getText());
        }
        public void removeUpdate(DocumentEvent event)
        {
            setTitle(titleField.getText());
        }
    });
```

Les trois méthodes de l'écouteur appellent la méthode `setTitle` du customizer. Cette méthode appelle le bean pour mettre à jour la valeur de propriété, puis déclenche un événement de changement de propriété. (Cette mise à jour n'est nécessaire que pour les propriétés qui ne sont pas liées.) Voici le code de la méthode `setTitle` :

```
public void setTitle(String newValue)
{
    if (bean == null) return;
    String oldValue = bean.getTitle();
    bean.setTitle(newValue);
    firePropertyChange("title", oldValue, newValue);
}
```

Lorsque la valeur de couleur change dans l'éditeur de la propriété `Color`, nous voulons mettre à jour la couleur de graphisme du bean. Nous détectons les changements de couleur en attachant un écouteur à l'éditeur de propriétés. Coïncidence peut-être troublante, cet éditeur émet également des événements de changement de propriété.

```
colorEditor.addPropertyChangeListener(new
    PropertyChangeListener()
    {
        public void propertyChange(PropertyChangeEvent
            event)
        {
            setGraphColor((Color) colorEditor.getValue());
        }
    });
```

Chaque fois que la valeur de couleur dans l'éditeur de propriétés `Color` change, nous appelons la méthode `setGraphColor` du customizer. Cette méthode met à jour la propriété `graphColor` du bean

496 Au cœur de Java 2 - Fonctions avancées

et déclenche un événement de changement de propriété différent qui est associé à la propriété `graphColor`.

```
public void setGraphColor(Color newValue)
{
    if (bean == null) return;
    Color oldValue = bean.getGraphColor();
    bean.setGraphColor(newValue);
    firePropertyChange("graphColor", oldValue, newValue);
}
```

L'Exemple 6.11 donne l'intégralité du code du customizer du bean `Chart`.

Ce customizer particulier définit simplement les propriétés du bean. En général, les customizers peuvent appeler n'importe quelle méthode du bean, qu'elles servent ou non à définir des propriétés. C'est-à-dire que les customizers sont plus généraux que les éditeurs de propriétés. (Certains beans peuvent avoir des fonctionnalités qui ne sont pas mises à disposition en tant que propriétés et qui ne peuvent être éditées que par l'intermédiaire du customizer.)

Exemple 6.11 : `ChartBean2Customizer.java`

```
package com.horstmann.corejava;

import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.io.*;
import java.text.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Un customizer destiné au Chartbean et qui permet à l'utilisateur
 * de modifier toutes les propriétés de chart dans une seule
 * boîte de dialogue à onglets.
 */
public class ChartBean2Customizer extends JTabbedPane
    implements Customizer
{
    public ChartBean2Customizer()
    {
        data = new JTextArea();
        JPanel dataPane = new JPanel();
        dataPane.setLayout(new BorderLayout());
        dataPane.add(new JScrollPane(data), BorderLayout.CENTER);
        JButton dataButton = new JButton("Définir données");
        dataButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                { setData(data.getText()); }
            });
        JPanel p = new JPanel();
        p.add(dataButton);
        dataPane.add(p, BorderLayout.SOUTH);

        JPanel colorPane = new JPanel();
```

```

colorPane.setLayout(new BorderLayout());

normal = new JCheckBox("Normal", true);
inverse = new JCheckBox("Inverse", false);
p = new JPanel();
p.add(normal);
p.add(inverse);
ButtonGroup g = new ButtonGroup();
g.add(normal);
g.add(inverse);
normal.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        { setInverse(false); }
    });

inverse.addActionListener(
    new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        { setInverse(true); }
    });

colorEditor
    = PropertyEditorManager.findEditor(Color.class);
colorEditor.addPropertyChangeListener(
    new PropertyChangeListener()
    {
        public void propertyChange(PropertyChangeEvent
            event)
        {
            setGraphColor((Color) colorEditor.getValue());
        }
    });

colorPane.add(p, BorderLayout.NORTH);
colorPane.add(colorEditor.getCustomEditor(),
    BorderLayout.CENTER);

JPanel titlePane = new JPanel();
titlePane.setLayout(new BorderLayout());

g = new ButtonGroup();
position = new JCheckBox[3];
position[0] = new JCheckBox("Gauche", false);
position[1] = new JCheckBox("Centré", true);
position[2] = new JCheckBox("Droite", false);

p = new JPanel();
for (int i = 0; i < position.length; i++)
{
    final int value = i;
    p.add(position[i]);
    g.add(position[i]);
    position[i].addActionListener(new
        ActionListener()
        {
            public void actionPerformed(ActionEvent event)

```


498 *Au cœur de Java 2 - Fonctions avancées*

```

        { setTitlePosition(value); }
    });
}

titleField = new JTextField();
titleField.getDocument().addDocumentListener(
    new DocumentListener()
    {
        public void changedUpdate(DocumentEvent evt)
        { setTitle(titleField.getText()); }
        public void insertUpdate(DocumentEvent evt)
        { setTitle(titleField.getText()); }
        public void removeUpdate(DocumentEvent evt)
        { setTitle(titleField.getText()); }
    });

titlePane.add(titleField, BorderLayout.NORTH);
titlePane.add(p, BorderLayout.SOUTH);
addTab("Couleur", colorPane);
addTab("Titre", titlePane);
addTab("Données", dataPane);

/**
 * Définit les données à afficher dans le graphique.
 * @param s une chaîne contenant les nombres à afficher,
 * séparés par un espace
 */
public void setData(String s)
{
    StringTokenizer tokenizer = new StringTokenizer(s);

    int i = 0;
    double[] values = new double[tokenizer.countTokens()];
    while (tokenizer.hasMoreTokens())
    {
        String token = tokenizer.nextToken();
        try
        {
            values[i] = Double.parseDouble(token);
            i++;
        }
        catch (NumberFormatException e)
        {
        }
    }
    setValues(values);
}

/**
 * Définit le titre du graphique.
 * @param newValue le nouveau titre
 */
public void setTitle(String newValue)
{
    if (bean == null) return;
    String oldValue = bean.getTitle();
    bean.setTitle(newValue);
    firePropertyChange("title", oldValue, newValue);
}

```

```
/**
 * Définit la position de titre du graphique.
 * @param i la nouvelle position de titre (ChartBean2.LEFT,
 * ChartBean2.CENTER ou ChartBean2.RIGHT)
 */
public void setTitlePosition(int i)
{
    if (bean == null) return;
    Integer oldValue = new Integer(bean.getTitlePosition());
    Integer newValue = new Integer(i);
    bean.setTitlePosition(i);
    firePropertyChange("titlePosition", oldValue, newValue);
}

/**
 * Définit le réglage inverse du graphique.
 * @param b true si les couleurs du graphique et du fond sont inversées
 */
public void setInverse(boolean b)
{
    if (bean == null) return;
    Boolean oldValue = new Boolean(bean.isInverse());
    Boolean newValue = new Boolean(b);
    bean.setInverse(b);
    firePropertyChange("inverse", oldValue, newValue);
}

/**
 * Définit les valeurs à afficher dans le graphique.
 * @param newValue le nouveau tableau des valeurs
 */
public void setValues(double[] newValue)
{
    if (bean == null) return;
    double[] oldValue = bean.getValues();
    bean.setValues(newValue);
    firePropertyChange("values", oldValue, newValue);
}

/**
 * Définit la couleur du graphique.
 * @param newValue la nouvelle couleur
 */
public void setGraphColor(Color newValue)
{
    if (bean == null) return;
    Color oldValue = bean.getGraphColor();
    bean.setGraphColor(newValue);
    firePropertyChange("graphColor", oldValue, newValue);
}

public void setObject(Object obj)
{
    bean = (ChartBean2) obj;

    data.setText("");
    for (double value : bean.getValues())
        data.append(value + "\n");
}
```

500 Au cœur de Java 2 - Fonctions avancées

```

        normal.setSelected(!bean.isInverse());
        inverse.setSelected(bean.isInverse());

        titleField.setText(bean.getTitle());

        for (int i = 0; i < position.length; i++)
            position[i].setSelected(i == bean.getTitlePosition());

        colorEditor.setValue(bean.getGraphColor());
    }

    public Dimension getPreferredSize()
    { return new Dimension(XPREFSIZE, YPREFSIZE); }

    private static final int XPREFSIZE = 200;
    private static final int YPREFSIZE = 120;
    private ChartBean2 bean;
    private PropertyEditor colorEditor;

    private JTextArea data;
    private JCheckBox normal;
    private JCheckBox inverse;
    private JCheckBox[] position;
    private JTextField titleField;
}

```

java.beans.Customizer 1.1

- void setObject(Object bean)
Spécifie le bean à personnaliser.

La persistance des JavaBeans

La persistance des JavaBeans fait appel aux propriétés des JavaBeans pour enregistrer des beans dans un flux et pour les lire ultérieurement ou sur une autre machine virtuelle. A cet égard, elle est identique à la sérialisation des objets (voir Chapitre 12 du Volume 1 pour en savoir plus sur la sérialisation). On notera toutefois une importante différence : la persistance des JavaBeans *convient à un stockage à long terme*.

Lorsqu'un objet est sérialisé, ses champs d'instance sont écrits sur un flux. Si l'implémentation d'une classe change, ses champs d'instance peuvent aussi changer. On ne peut plus, dès lors, se contenter de lire les fichiers qui contiennent des objets sérialisés d'anciennes versions. Il est pourtant possible de détecter les différences entre les versions et de les traduire, des anciennes représentations de données vers les nouvelles. La procédure est toutefois extrêmement ennuyeuse et ne doit être employée que dans les situations désespérées. De fait, la sérialisation est inadaptée pour un stockage à long terme. C'est pour cette raison que la documentation de tous les composants Swing indique : "Attention, les objets sérialisés de cette classe ne seront pas compatibles avec les prochaines versions de Swing. La prise en charge actuelle de la sérialisation convient pour un stockage à court terme ou un RMI entre les applications."

Le mécanisme de persistance à long terme a été inventé pour servir de solution à ce problème. Son objectif premier était le glisser-déposer des outils de conception de GUI. L'outil de conception enregistre le résultat des clics de souris (une série de cadres, panneaux, boutons ou autres composants

Swing) dans un fichier, en utilisant le format de la persistance à long terme. Le programme qui s'exécute se contente d'ouvrir ce fichier. Cette approche élimine tout un code ennuyeux visant à disposer et à connecter les composants Swing. Malheureusement, elle n'est pas largement implémentée.

NOTE

BeanBuilder, disponible à l'adresse <http://bean-builder.dev.java.net>, est un générateur de GUI expérimental qui prend en charge la persistance à long terme.

L'idée à l'origine de la persistance des JavaBeans est toute simple. Supposons que vous vouliez enregistrer un objet `JFrame` dans un fichier pour pouvoir le récupérer plus tard. Si vous étudiez le code source de la classe `JFrame` et de ses superclasses, vous verrez des dizaines de champs d'instance. S'ils devaient être sérialisés, il faudrait écrire toutes leurs valeurs. Mais souvenez-vous de la construction d'un cadre :

```
JFrame frame = new JFrame();
frame.setTitle("My Application");
frame.setVisible(true);
```

Le constructeur par défaut initialise tous les champs d'instance ; deux propriétés sont définies. Si vous archivez l'objet cadre, le mécanisme de persistance des JavaBeans enregistre ces instructions au format XML :

```
<object class="javax.swing.JFrame">
  <void property="title">
    <string>My Application</string>
  </void>
  <void property="visible">
    <boolean>true</boolean>
  </void>
</object>
```

A la lecture de l'objet, les instructions sont exécutées : un objet `JFrame` est construit, et son titre et ses propriétés visibles sont définis sur les valeurs par défaut. Peu importe si la représentation interne du `JFrame` a changé entre-temps. Il suffit que vous restauriez l'objet en définissant ses propriétés.

Sachez que seules sont archivées les propriétés qui diffèrent des valeurs par défaut. Le `XMLEncoder` crée un `JFrame` par défaut et compare ses propriétés avec le cadre archivé. Les instructions de définition des propriétés ne sont générées que pour les propriétés qui diffèrent des paramètres par défaut. Cette procédure est appelée *élimination de la redondance*. En conséquence, les archives sont généralement plus petites que le résultat de la sérialisation (lors de la sérialisation de composants Swing, la différence est particulièrement importante, car les objets Swing ont de nombreux états, dont la plupart ne sont jamais modifiés).

Cette approche présente bien entendu quelques tracasseries techniques mineures. Par exemple, l'appel

```
frame.setSize(600, 400);
```

ne définit pas de propriété. Or `XMLEncoder` peut y faire face. Il écrit l'instruction :

```
<void property="bounds">
  <object class="java.awt.Rectangle">
```

502 Au cœur de Java 2 - Fonctions avancées

```
<int>0</int>
<int>0</int>
<int>600</int>
<int>400</int>
</object>
</void>
```

Utilisez `XMLEncoder` pour enregistrer un objet dans un flux :

```
XMLEncoder out = new XMLEncoder(new FileOutputStream(. . .));
out.writeObject(frame);
out.close();
```

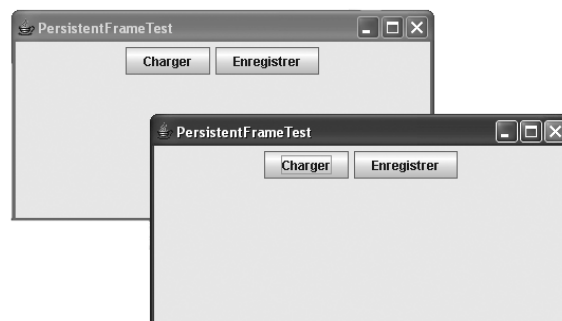
Pour le relire, utilisez `XMLDecoder` :

```
XMLDecoder in = new XMLDecoder(new FileInputStream(. . .));
JFrame newFrame = (JFrame) in.readObject();
in.close();
```

Le programme de l'Exemple 6.12 montre le chargement et l'enregistrement d'un cadre (voir Figure 6.15). Lorsque vous exécutez le programme, cliquez d'abord sur le bouton "Enregistrer", puis enregistrez le cadre dans un fichier. Déplacez ensuite le premier cadre et cliquez sur Charger pour voir apparaître un autre cadre à l'emplacement d'origine. Etudiez le fichier XML produit par le programme.

Figure 6.15

Le programme
PersistentFrameTest.



Etudiez attentivement le résultat XML, vous verrez que `XMLEncoder` réalise un travail important lorsqu'il enregistre le cadre. `XMLEncoder` produit des instructions pour les actions suivantes :

- définir diverses propriétés du cadre, `size`, `layout`, `defaultCloseOperation`, `title`, etc. ;
- ajouter des boutons au cadre ;
- ajouter des écouteurs d'action aux boutons.

Ici, nous avons dû construire les écouteurs d'action à l'aide de la classe `EventHandler`. `XMLEncoder` n'a pas pu archiver de classes internes arbitraires, mais il sait gérer les objets `EventHandler`.

Exemple 6.12 : `PersistentFrameTest.java`

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
```

```
import java.io.*;
import javax.swing.*;

/**
 * Ce programme montre l'utilisation d'un encodeur et d'un décodeur XML
 * pour enregistrer et restaurer un cadre.
 */
public class PersistentFrameTest
{
    public static void main(String[] args)
    {
        chooser = new JFileChooser();
        chooser.setCurrentDirectory(new File("."));
        PersistentFrameTest test = new PersistentFrameTest();
        test.init();
    }

    public void init()
    {
        frame = new JFrame();
        frame.setLayout(new FlowLayout());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setTitle("PersistentFrameTest");
        frame.setSize(400, 200);

        JButton loadButton = new JButton("Charger");
        frame.add(loadButton);
        loadButton.addActionListener(
            EventHandler.create(ActionListener.class, this, "load"));

        JButton saveButton = new JButton("Enregistrer");
        frame.add(saveButton);
        saveButton.addActionListener(
            EventHandler.create(ActionListener.class, this, "save"));

        frame.setVisible(true);
    }

    public void load()
    {
        // afficher la boîte de dialogue de sélection de fichier
        int r = chooser.showOpenDialog(null);

        // si le fichier est sélectionné, ouvrir
        if(r == JFileChooser.APPROVE_OPTION)
        {
            try
            {
                File file = chooser.getSelectedFile();
                XMLDecoder decoder = new XMLDecoder(
                    new FileInputStream(file));
                JFrame newFrame = (JFrame) decoder.readObject();
                decoder.close();
            }
            catch (IOException e)
            {
                JOptionPane.showMessageDialog(null, e);
            }
        }
    }
}
```

504 Au cœur de Java 2 - Fonctions avancées

```

    }

    public void save()
    {
        // afficher la boîte de sélection de fichiers
        int r = chooser.showSaveDialog(null);

        // si le fichier est sélectionné, enregistrer
        if(r == JFileChooser.APPROVE_OPTION)
        {
            try
            {
                File file = chooser.getSelectedFile();
                XMLEncoder encoder = new XMLEncoder(
                    new FileOutputStream(file));
                encoder.writeObject(frame);
                encoder.close();
            }
            catch (IOException e)
            {
                JOptionPane.showMessageDialog(null, e);
            }
        }
    }

    private static JFileChooser chooser;
    private JFrame frame;
}

```

Utiliser la persistance des JavaBeans pour des données arbitraires

La persistance des JavaBeans ne se limite pas au stockage des composants Swing. Vous pouvez utiliser ce mécanisme pour stocker *n'importe quelle* collection d'objets, à condition de suivre quelques règles simples. Dans les sections suivantes, vous allez apprendre à utiliser la persistance des JavaBeans comme format de stockage à long terme pour vos données.

Ecrire un délégué persistant pour construire un objet

La persistance JavaBeans n'aura que peu d'intérêt si l'on peut obtenir le statut de chaque objet en paramétrant des propriétés. Cependant, dans les programmes réels, il y aura toujours des classes qui ne fonctionnent pas de cette manière. Considérez, par exemple, la classe `Employee` rencontrée au Chapitre 4 du Volume 1. `Employee` n'est pas un bean très correct. Il ne possède pas de constructeur par défaut ni de méthode `setName`, `setSalary` ou `setHireDay`. Pour surmonter ce problème, vous installerez un *délégué persistant* dans `XMLWriter` qui saura écrire les objets `Employee` :

```
out.setPersistenceDelegate(Employee.class, delegate);
```

Le délégué persistant de la classe `Employee` surcharge la méthode `instantiate` pour produire une *expression* qui construit un objet.

```

PersistenceDelegate delegate = new
    DefaultPersistenceDelegate()
    {
        protected Expression instantiate(Object oldInstance,
            Encoder out)
        {

```

```

Employee e = (Employee) oldInstance;
GregorianCalendar c = new GregorianCalendar();
c.setTime(e.getHireDay());
return new Expression(oldInstance, Employee.class, "new",
    new Object[]
    {
        e.getName(),
        e.getSalary(),
        c.get(Calendar.YEAR),
        c.get(Calendar.MONTH),
        c.get(Calendar.DATE)
    });
}
};

```

La signification est la suivante : "Pour recréer `oldInstance`, appelez la méthode `new` (à savoir le constructeur) sur l'objet `Employee.class` et fournissez les paramètres donnés." Le nom du paramètre `oldInstance` est un peu trompeur, car il s'agit simplement de l'instance qui est sauvegardée.

Une fois le délégué installé, vous pouvez enregistrer les objets `Employee`. Par exemple, les instructions

```

Object myData = new Employee("Harry Hacker", 50000, 1989, 10, 1);
out.writeObject(myData);

```

génèrent le résultat suivant :

```

<object class="Employee">
  <string>Harry Hacker</string>
  <double>50000.0</double>
  <int>1989</int>
  <int>9</int>
  <int>1</int>
</object>

```

INFO

Il vous suffira de peaufiner le processus d'encodage. Il n'existe pas de méthode particulière pour le décodage. Le décodeur exécute simplement les instructions et expressions qu'il trouve dans ses entrées XML.

Construire un objet à partir des propriétés

Il est souvent possible d'employer ce raccourci : si tous les paramètres du constructeur peuvent être obtenus par un accès aux propriétés d'`oldInstance`, vous n'aurez pas besoin d'écrire la méthode `instantiate` vous-même. Il vous suffira de construire un `DefaultPersistenceDelegate` et de modifier les noms des propriétés.

Par exemple, les instructions suivantes définissent le délégué de persistance pour la classe `Rectangle2D.Double` :

```

out.setPersistenceDelegate(Rectangle2D.Double.class,
    new DefaultPersistenceDelegate(new String[] {
        "x", "y", "width", "height" }));

```


506 *Au cœur de Java 2 - Fonctions avancées*

L'encodeur comprend : "Pour coder un objet `Rectangle2D.Double`, récupérez ses propriétés `x`, `y`, `width` et `height` et appelez le constructeur avec ces quatre valeurs." Par conséquent, le résultat contient un élément ressemblant à ceci :

```
<object class="java.awt.geom.Rectangle2D$Double">
  <double>5.0</double>
  <double>10.0</double>
  <double>20.0</double>
  <double>30.0</double>
</object>
```

Construire un objet avec une méthode factory

Vous devrez parfois sauvegarder des objets obtenus par des méthodes `factory`, et non par des constructeurs. Étudiez, par exemple, comment l'on obtient un objet `InetAddress` :

```
byte[] bytes = new byte[] { 127, 0, 0, 1};
InetAddress address = InetAddress.getByAddress(bytes);
```

La méthode `instantiate` de `PersistenceDelegate` produit un appel à la méthode `factory`.

```
protected Expression instantiate(Object oldInstance, Encoder out)
{
    return new Expression(oldInstance, InetAddress.class, "getByAddress",
        new Object[] { ((InetAddress) oldInstance).getAddress() });
}
```

Voici un exemple du résultat :

```
<object class="java.net.Inet4Address" method="getByAddress">
  <array class="byte" length="4">
    <void index="0">
      <byte>127</byte>
    </void>
    <void index="3">
      <byte>1</byte>
    </void>
  </array>
</object>
```

ATTENTION

Vous devez installer ce délégué avec la sous-classe concrète, comme `Inet4Address`, et non avec la classe abstraite `InetAddress` !

Enumérations

Pour enregistrer une valeur `enum`, il convient de fournir un délégué très simple :

```
enum Mood { SAD, HAPPY };
...
out.setPersistenceDelegate(Mood.class, new EnumDelegate());
```

Vous trouverez la classe `EnumDelegate` dans l'Exemple 6.14. Si vous enregistrez par exemple `Mood.SAD`, le délégué écrira une expression équivalant à `Enum.valueOf(Mood.class, "SAD")` :

```
<object class="java.lang.Enum" method="valueOf">
  <class>Mood</class>
  <string>SAD</string>
</object>
```

Travail de postconstruction

L'état de certaines classes est élaboré en faisant appel à des méthodes qui ne définissent pas de propriétés. Vous pouvez faire face à cela en surchargeant la méthode `initialize` de `DefaultPersistenceDelegate`. La méthode `initialize` est appelée après la méthode `instantiate`. Vous pouvez générer une séquence d'instructions enregistrées dans l'archive.

Etudiez par exemple la classe `BitSet`. Afin de recréer un objet `BitSet`, vous définissez tous les bits qui étaient présents dans l'original. Ci-après, la méthode `initialize` génère les instructions idoines :

```
protected void initialize(Class type, Object oldInstance,
    Object newInstance, Encoder out)
{
    super.initialize(type, oldInstance, newInstance, out);
    BitSet bs = (BitSet) oldInstance;
    for (int i = bs.nextSetBit(0); i >= 0; i = bs.nextSetBit(i + 1))
        out.writeStatement(new Statement(bs, "set",
            new Object[] { i, i + 1, true } ));
}
```

Voici un exemple de résultat :

```
<object class="java.util.BitSet">
  <void method="set">
    <int>1</int>
    <int>2</int>
    <boolean>true</boolean>
  </void>
  <void method="set">
    <int>4</int>
    <int>5</int>
    <boolean>true</boolean>
  </void>
</object>
```

INFO

Cela aurait sans doute plus de sens d'écrire `new Statement(bs, "set", new Object[] { i } )` mais, ainsi, le `XMLWriter` produit une commande disgracieuse qui définit une propriété ayant un nom vide.

Délégués prédéfinis

Il est inutile de fournir vos propres délégués pour toutes les classes. Le `XMLEncoder` en possède déjà de prédéfinis pour les types suivants :

- `null` ;

508 *Au cœur de Java 2 - Fonctions avancées*

- tous les types primitifs et leurs enveloppes ;
- String ;
- les tableaux ;
- les collections et les cartes ;
- les types de réflexions Class, Field, Method et Proxy ;
- les types AWT Color, Cursor, Dimension, Font, Insets, Point, Rectangle et ImageIcon ;
- les composants, bordures, gestionnaires de mise en page et modèles AWT et Swing ;
- les gestionnaires d'événements.

Propriétés transitoires

A l'occasion, une classe peut détenir une propriété avec des éléments de récupération et de définition que XMLDecoder découvrira, mais ces valeurs de propriétés ne doivent pas être incluses dans l'archive. Pour supprimer l'archivage d'une propriété, marquez-la comme *transitoire* dans le descripteur de propriété. L'instruction suivante, par exemple, indique à la classe GregorianCalendar de ne pas archiver la propriété gregorianChange :

```
BeanInfo info = Introspector.getBeanInfo(GregorianCalendar.class);
for (PropertyDescriptor desc : info.getPropertyDescriptors())
    if (desc.getName().equals("gregorianChange"))
        desc.setValue("transient", Boolean.TRUE);
```

La méthode setValue peut stocker des informations arbitraires avec un descripteur de propriété. Le XMLEncoder interroge l'attribut transient avant de générer une instruction de définition de propriété.

Le programme de l'Exemple 6.13 montre divers délégués de persistance. Sachez que ce programme présente le pire des scénarios ; dans les vraies applications, bien des classes peuvent être archivées sans utiliser de délégués.

Exemple 6.13 : PersistenceDelegateTest.java

```
import java.awt.*;
import java.awt.geom.*;
import java.beans.*;
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * Ce programme montre divers délégués de persistance.
 */
public class PersistenceDelegateTest
{
    public enum Mood { SAD, HAPPY };

    public static void main(String[] args) throws Exception
    {
        XMLEncoder out = new XMLEncoder(System.out);
        out.setExceptionListener(new
            ExceptionListener()
            {
                // ...
            }
        );
    }
}
```

```

        public void exceptionThrown(Exception e)
        {
            e.printStackTrace();
        }
    });

    PersistenceDelegate delegate = new
    DefaultPersistenceDelegate()
    {
        protected Expression instantiate(
            Object oldInstance, Encoder out)
        {
            Employee e = (Employee) oldInstance;
            GregorianCalendar c = new GregorianCalendar();
            c.setTime(e.getHireDay());
            return new Expression(oldInstance, Employee.class, "new",
                new Object[]
                {
                    e.getName(),
                    e.getSalary(),
                    c.get(Calendar.YEAR),
                    c.get(Calendar.MONTH),
                    c.get(Calendar.DATE)
                });
        }
    };

    out.setPersistenceDelegate(Employee.class, delegate);

    out.setPersistenceDelegate(Rectangle2D.Double.class,
        new DefaultPersistenceDelegate(new String[] {
            "x", "y", "width", "height" }));

    out.setPersistenceDelegate(InetAddress.class, new
    DefaultPersistenceDelegate()
    {
        protected Expression instantiate(
            Object oldInstance, Encoder out)
        {
            return new Expression(oldInstance,
                InetAddress.class, "getByAddress",
                new Object[] { ((InetAddress)
                    oldInstance).getAddress() });
        }
    });

    out.setPersistenceDelegate(BitSet.class, new
    DefaultPersistenceDelegate()
    {
        protected void initialize(Class type,
            Object oldInstance, Object newInstance,
            Encoder out)
        {
            super.initialize(type, oldInstance, newInstance, out);
            BitSet bs = (BitSet) oldInstance;
            for(int i = bs.nextSetBit(0); i >= 0;
                i = bs.nextSetBit(i + 1))
                out.writeStatement(new Statement(bs, "set",
                    new Object[]{ i, i + 1, true }));
        }
    });

```

510 *Au cœur de Java 2 - Fonctions avancées*

```

    }
  });

  out.setPersistenceDelegate(Mood.class, new EnumDelegate());

  out.writeObject(new Employee("Harry Hacker", 50000, 1989, 10, 1));
  out.writeObject(new java.awt.geom.Rectangle2D.Double(
    5, 10, 20, 30));
  out.writeObject(InetAddress.getLocalHost());
  out.writeObject(Mood.SAD);
  BitSet bs = new BitSet(); bs.set(1, 4); bs.clear(2, 3);
  out.writeObject(bs);
  out.writeObject(Color.PINK);
  out.writeObject(new GregorianCalendar());
  out.close();
}

static
{
  try
  {
    BeanInfo info = Introspector.getBeanInfo(
      GregorianCalendar.class);
    for (PropertyDescriptor desc : info.getPropertyDescriptors())
      if (desc.getName().equals("gregorianChange"))
        desc.setValue("transient", Boolean.TRUE);
  }
  catch (IntrospectionException e)
  {
    e.printStackTrace();
  }
}
}

```

Exemple 6.14 : EnumDelegate.java

```

import java.beans.*;

/**
 * Cette classe peut être utilisée pour sauvegarder tous types enum dans
 * une archive JavaBeans.
 */
public class EnumDelegate extends DefaultPersistenceDelegate
{
  protected Expression instantiate(Object oldInstance, Encoder out)
  {
    return new Expression(Enum.class,
      "valueOf",
      new Object[] { oldInstance.getClass(), ((Enum)
        oldInstance).name() });
  }
}

```

Un exemple complet de persistance des JavaBeans

Nous allons terminer la description de la persistance des JavaBeans par un exemple complet (voir Figure 6.16). Cette application produit un rapport des dommages pour une voiture de location. Le loueur consigne la location, sélectionne le type de voiture, clique aux endroits où la voiture a été

abîmée, puis sauvegarde le rapport. L'application peut aussi charger des rapports de dommages existants. L'Exemple 6.15 contient le code du programme.

L'application utilise la persistance des JavaBeans pour sauvegarder et charger les objets Damage-Report (voir Exemple 6.16). Elle illustre les aspects suivants de la persistance :

- Les propriétés sont automatiquement sauvegardées et récupérées. Il n'y a rien à faire pour les propriétés `rentalRecord` et `carType`.
- Un travail de postconstruction est nécessaire pour récupérer les emplacements des dommages. Le délégué de persistance génère des instructions qui appellent la méthode `click`.
- La classe `Point2D.Double` a besoin d'un `DefaultPersistenceDelegate` qui construise un point à partir de ses coordonnées `x` et `y`.
- Un `EnumDelegate` est requis pour gérer le type énuméré `CarType`.
- La propriété `removeMode` (qui spécifie l'endroit où un clic de souris ajoute ou supprime des marques de dommages) est transitoire car elle n'a pas à être enregistrée dans les rapports.

Voici un exemple de rapport de dommages :

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.5.0" class="java.beans.XMLDecoder">
  <object class="DamageReport">
    <object class="java.lang.Enum" method="valueOf">
      <class>DamageReport$CarType</class>
      <string>SEDAN</string>
    </object>
    <void property="rentalRecord">
      <string>12443-19</string>
    </void>
    <void method="click">
      <object class="java.awt.geom.Point2D$Double">
        <double>181.0</double>
        <double>84.0</double>
      </object>
    </void>
    <void method="click">
      <object class="java.awt.geom.Point2D$Double">
        <double>162.0</double>
        <double>66.0</double>
      </object>
    </void>
  </object>
</java>
```

INFO

L'application d'exemple n'utilise *pas* la persistance des JavaBeans pour sauvegarder la GUI de l'application. Cela peut avoir de l'intérêt pour les créateurs d'outils de développement, mais nous nous concentrerons ici sur la manière d'utiliser le mécanisme de persistance pour stocker les *données de l'application*.

512 Au cœur de Java 2 - Fonctions avancées

ATTENTION

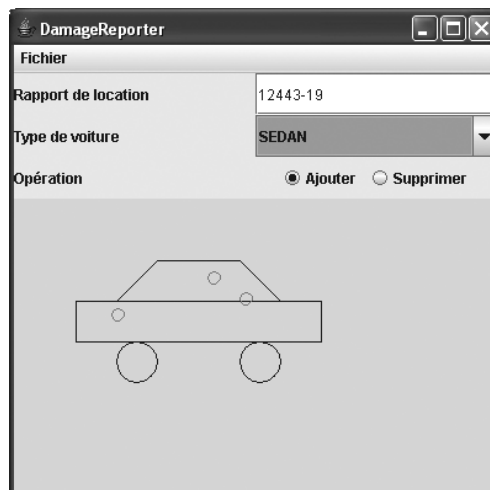
A l'heure où nous écrivons, la persistance des JavaBeans n'est pas compatible avec Java Web Start (voir le bogue 4741757 du "bug parade" à l'adresse <http://bugs.sun.com>).

Cet exemple clôt notre discussion sur la persistance des JavaBeans. En résumé, les caractéristiques de cette persistance sont les suivantes :

- Elle est adaptée à un stockage de longue durée.
- Elle est petite et rapide.
- Elle est facile à créer.
- Elle est modifiable par l'homme.
- Elle est livrée en standard avec Java.

Figure 6.16

L'application de Rapport des dommages.



Exemple 6.15 : DamageReporter.java

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.beans.*;
import java.io.*;
import java.util.*;
import javax.swing.*;

/**
 * Ce programme montre l'utilisation d'un encodeur et d'un décodeur XML.
 * Le code de la GUI et du dessin est collecté dans cette classe. Les
 * seuls éléments intéressants sont les écouteurs d'action pour openItem
 * et saveItem. Vous verrez les personnalisations d'encodeur dans
 * la classe DamageReport.
 */
public class DamageReporter extends JFrame
{
```

```

public static void main(String[] args)
{
    JFrame frame = new DamageReporter();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
}

public DamageReporter()
{
    setTitle("DamageReporter");
    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

    chooser = new JFileChooser();
    chooser.setCurrentDirectory(new File("."));

    report = new DamageReport();
    report.setCarType(DamageReport.CarType.SEDAN);

    // Configurer la barre de menus
    JMenuBar menuBar = new JMenuBar();
    setJMenuBar(menuBar);

    JMenu menu = new JMenu("Fichier");
    menuBar.add(menu);

    JMenuItem openItem = new JMenuItem("Ouvrir");
    menu.add(openItem);
    openItem.addActionListener(new
        ActionListener()
        {
            public void actionPerformed(ActionEvent evt)
            {
                // Afficher la boîte de dialogue de choix du fichier
                int r = chooser.showOpenDialog(null);

                // si le fichier est sélectionné, ouvrir
                if(r == JFileChooser.APPROVE_OPTION)
                {
                    try
                    {
                        File file = chooser.getSelectedFile();
                        XMLDecoder decoder = new XMLDecoder(
                            new FileInputStream(file));
                        report = (DamageReport) decoder.readObject();
                        decoder.close();
                        repaint();
                    }
                    catch (IOException e)
                    {
                        JOptionPane.showMessageDialog(null, e);
                    }
                }
            }
        });

    JMenuItem saveItem = new JMenuItem("Enregistrer");
    menu.add(saveItem);
    saveItem.addActionListener(new
        ActionListener()

```


514 *Au cœur de Java 2 - Fonctions avancées*

```

    {
        public void actionPerformed(ActionEvent evt)
        {
            report.setRentalRecord(rentalRecord.getText());
            chooser.setSelectedFile(new File(
                rentalRecord.getText() + ".xml"));

            // Afficher la boîte de dialogue de choix du fichier
            int r = chooser.showSaveDialog(null);

            // si le fichier est sélectionné, sauvegarder
            if(r == JFileChooser.APPROVE_OPTION)
            {
                try
                {
                    File file = chooser.getSelectedFile();
                    XMLEncoder encoder = new XMLEncoder(
                        new FileOutputStream(file));
                    report.configureEncoder(encoder);
                    encoder.writeObject(report);
                    encoder.close();
                }
                catch (IOException e)
                {
                    JOptionPane.showMessageDialog(null, e);
                }
            }
        }
    });

    JMenuItem exitItem = new JMenuItem("Quitter");
    menu.add(exitItem);
    exitItem.addActionListener(new
        ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                System.exit(0);
            }
        });

    // liste déroulante pour le type de voiture
    rentalRecord = new JTextField();
    carType = new JComboBox();
    carType.addItem(DamageReport.CarType.SEDAN);
    carType.addItem(DamageReport.CarType.WAGON);
    carType.addItem(DamageReport.CarType.SUV);

    carType.addActionListener(new
        ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                DamageReport.CarType item = (DamageReport.CarType)
                    carType.getSelectedItem();
                report.setCarType(item);
                repaint();
            }
        });
    });

```

```
// panneau pour afficher la voiture
carPanel = new
JPanel()
{
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        g2.draw((Shape) shapes.get(report.getCarType()));
        report.drawDamage(g2);
    }
};
carPanel.addMouseListener(new
MouseAdapter()
{
    public void mousePressed(MouseEvent event)
    {
        report.click(new Point2D.Double(event.getX(),
            event.getY()));
        repaint();
    }
});
carPanel.setBackground(new Color(0.9f, 0.9f, 0.45f));

// Boutons radio pour les actions de clic
addButton = new JRadioButton("Ajouter");
removeButton = new JRadioButton("Supprimer");
ButtonGroup group = new ButtonGroup();
JPanel buttonPanel = new JPanel();
group.add(addButton);
buttonPanel.add(addButton);
group.add(removeButton);
buttonPanel.add(removeButton);
addButton.setSelected(!report.getRemoveMode());
removeButton.setSelected(report.getRemoveMode());
addButton.addActionListener(new
ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        report.setRemoveMode(false);
    }
});
removeButton.addActionListener(new
ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        report.setRemoveMode(true);
    }
});

// Composants de mise en page
JPanel gridPanel = new JPanel();
gridPanel.setLayout(new GridLayout(0, 2));
gridPanel.add(new JLabel("Rapport de location"));
gridPanel.add(rentalRecord);
gridPanel.add(new JLabel("Type de voiture"));
gridPanel.add(carType);
```

516 Au cœur de Java 2 - Fonctions avancées

```

        gridPanel.add(new JLabel("Opération"));
        gridPanel.add(buttonPanel);

        add(gridPanel, BorderLayout.NORTH);
        add(carPanel, BorderLayout.CENTER);
    }

    private JTextField rentalRecord;
    private JComboBox carType;
    private JPanel carPanel;
    private JRadioButton addButton;
    private JRadioButton removeButton;
    private DamageReport report;
    private JFileChooser chooser;

    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 400;

    private static Map<DamageReport.CarType, Shape> shapes
        = new EnumMap<DamageReport.CarType,
            Shape>(DamageReport.CarType.class);

    static
    {
        int width = 200;
        int height = 100;
        int x = 50;
        int y = 50;
        Rectangle2D.Double body = new Rectangle2D.Double(
            x, y + width / 6, width - 1, width / 6);
        Ellipse2D.Double frontTire = new Ellipse2D.Double(
            x + width / 6, y + width / 3,
            width / 6, width / 6);
        Ellipse2D.Double rearTire = new Ellipse2D.Double(
            x + width * 2 / 3, y + width / 3,
            width / 6, width / 6);

        Point2D.Double p1 = new Point2D.Double(
            x + width / 6, y + width / 6);
        Point2D.Double p2 = new Point2D.Double(x + width / 3, y);
        Point2D.Double p3 = new Point2D.Double(x + width * 2 / 3, y);
        Point2D.Double p4 = new Point2D.Double(
            x + width * 5 / 6, y + width / 6);

        Line2D.Double frontWindshield = new Line2D.Double(p1, p2);
        Line2D.Double roofTop = new Line2D.Double(p2, p3);
        Line2D.Double rearWindshield = new Line2D.Double(p3, p4);

        GeneralPath sedanPath = new GeneralPath();
        sedanPath.append(frontTire, false);
        sedanPath.append(rearTire, false);
        sedanPath.append(body, false);
        sedanPath.append(frontWindshield, false);
        sedanPath.append(roofTop, false);
        sedanPath.append(rearWindshield, false);
        shapes.put(DamageReport.CarType.SEDAN, sedanPath);

        Point2D.Double p5 = new Point2D.Double(x + width * 11 / 12, y);
        Point2D.Double p6 = new Point2D.Double(x + width, y + width / 6);
    }

```

```

        roofTop = new Line2D.Double(p2, p5);
        rearWindshield = new Line2D.Double(p5, p6);

        GeneralPath wagonPath = new GeneralPath();
        wagonPath.append(frontTire, false);
        wagonPath.append(rearTire, false);
        wagonPath.append(body, false);
        wagonPath.append(frontWindshield, false);
        wagonPath.append(roofTop, false);
        wagonPath.append(rearWindshield, false);
        shapes.put(DamageReport.CarType.WAGON, wagonPath);

        Point2D.Double p7 = new Point2D.Double(
            x + width / 3, y - width / 6);
        Point2D.Double p8 = new Point2D.Double(
            x + width * 11 / 12, y - width / 6);
        frontWindshield = new Line2D.Double(p1, p7);
        roofTop = new Line2D.Double(p7, p8);
        rearWindshield = new Line2D.Double(p8, p6);

        GeneralPath suvPath = new GeneralPath();
        suvPath.append(frontTire, false);
        suvPath.append(rearTire, false);
        suvPath.append(body, false);
        suvPath.append(frontWindshield, false);
        suvPath.append(roofTop, false);
        suvPath.append(rearWindshield, false);
        shapes.put(DamageReport.CarType.SUV, suvPath);
    }
}

```

Exemple 6.16 : DamageReport.java

```

import java.awt.*;
import java.awt.geom.*;
import java.beans.*;
import java.util.*;

/**
 * Cette classe décrit un rapport de dommages à un véhicule qui sera
 * sauvegardé et chargé avec un mécanisme de persistance à long terme.
 */
public class DamageReport
{
    public enum CarType { SEDAN, WAGON, SUV }

    // Cette propriété est sauvegardée automatiquement
    public void setRentalRecord(String newValue)
    {
        rentalRecord = newValue;
    }

    public String getRentalRecord()
    {
        return rentalRecord;
    }

    // Cette propriété est sauvegardée automatiquement
    public void setCarType(CarType newValue)

```

518 *Au cœur de Java 2 - Fonctions avancées*

```

    {
        carType = newValue;
    }

    public CarType getCarType()
    {
        return carType;
    }

    // Cette propriété est définie comme transitoire
    public void setRemoveMode(boolean newValue)
    {
        removeMode = newValue;
    }

    public boolean getRemoveMode()
    {
        return removeMode;
    }

    public void click(Point2D p)
    {
        if (removeMode)
        {
            for (Point2D center : points)
            {
                Ellipse2D circle = new Ellipse2D.Double(
                    center.getX() - MARK_SIZE, center.getY() - MARK_SIZE,
                    2 * MARK_SIZE, 2 * MARK_SIZE);
                if (circle.contains(p))
                {
                    points.remove(center);
                    return;
                }
            }
        }
        else points.add(p);
    }

    public void drawDamage(Graphics2D g2)
    {
        g2.setPaint(Color.RED);
        for (Point2D center : points)
        {
            Ellipse2D circle = new Ellipse2D.Double(
                center.getX() - MARK_SIZE, center.getY() - MARK_SIZE,
                2 * MARK_SIZE, 2 * MARK_SIZE);
            g2.draw(circle);
        }
    }

    public void configureEncoder(XMLEncoder encoder)
    {
        // Cette étape est nécessaire pour sauvegarder les objets
        // Point2D.Double
        encoder.setPersistenceDelegate(
            Point2D.Double.class,
            new DefaultPersistenceDelegate(new String[]{ "x", "y" } ) );
    }

```

```
// Cette étape est nécessaire pour sauvegarder le type énuméré
// CarType
encoder.setPersistenceDelegate(CarType.class, new EnumDelegate());

// Cette étape est nécessaire car la liste des points sous forme de
// tableau n'est pas (et ne doit pas être) exposée
// comme une propriété
encoder.setPersistenceDelegate(
    DamageReport.class, new
    DefaultPersistenceDelegate()
    {
        protected void initialize(Class type,
            Object oldInstance, Object newInstance,
            Encoder out)
        {
            super.initialize(type, oldInstance, newInstance, out);
            DamageReport r = (DamageReport) oldInstance;

            for (Point2D p : r.points)
                out.writeStatement(
                    new Statement(oldInstance, "click",
                        new Object[]{ p }) );
        }
    });

// Cette étape est nécessaire pour rendre transitoire la propriété
// removeMode
static
{
    try
    {
        BeanInfo info = Introspector.getBeanInfo(DamageReport.class);
        for (PropertyDescriptor desc : info.getPropertyDescriptors())
            if (desc.getName().equals("removeMode"))
                desc.setValue("transient", Boolean.TRUE);
    }
    catch (IntrospectionException e)
    {
        e.printStackTrace();
    }
}

private String rentalRecord;
private CarType carType;
private boolean removeMode;
private ArrayList<Point2D> points = new ArrayList<Point2D>();

private static final int MARK_SIZE = 5;
}
```

java.beans.XMLEncoder 1.4

- XMLEncoder(OutputStream out)
Construit un XMLEncoder qui envoie son résultat au flux donné.
- void writeObject(Object obj)
Archive l'objet donné.

520 *Au cœur de Java 2 - Fonctions avancées*

- `void writeStatement(Statement stat)`

Écrit l'instruction donnée dans l'archive. Cette méthode ne doit être appelée qu'à partir d'un délégué de persistance.

java.beans.Encoder 1.4

- `void setPersistenceDelegate(Class<?> type, PersistenceDelegate delegate)`
- `PersistenceDelegate getPersistenceDelegate(Class<?> type)`

Définissent ou récupèrent le délégué pour archiver les objets du type donné.

- `void setExceptionListener(ExceptionListener listener)`
- `ExceptionListener getExceptionListener()`

Définissent ou récupèrent l'écouteur d'exceptions qui est notifié au cas où une exception surviendrait pendant le processus d'encodage.

java.beans.ExceptionListener 1.4

- `void exceptionThrown(Exception e)`

Est appelé quand une exception a été déclenchée pendant le processus d'encodage ou de décodage.

java.beans.XMLDecoder 1.4

- `XMLDecoder(InputStream in)`
Construit un XMLDecoder qui lit une archive du flux d'entrée donné.
- `Object readObject()`
Lit l'objet suivant dans l'archive.

- `void setExceptionListener(ExceptionListener listener)`
- `ExceptionListener getExceptionListener()`
Définissent ou récupèrent l'écouteur d'exceptions qui est notifié si une exception survient pendant le processus d'encodage.

java.beans.PersistenceDelegate 1.4

- `protected abstract Expression instantiate(Object oldInstance, Encoder out)`
Renvoie une expression pour instancier un objet équivalent à oldInstance.
- `protected void initialize(Class<?> type, Object oldInstance, Object newInstance, Encoder out)`
Écrit des instructions pour out qui transforment newInstance en objet équivalent à oldInstance.

java.beans.DefaultPersistenceDelegate 1.4

- `DefaultPersistenceDelegate()`
Construit un délégué de persistance pour une classe dont le constructeur n'a pas de paramètres.
- `DefaultPersistenceDelegate(String[] propertyNames)`
Construit un délégué de persistance pour une classe dont les paramètres de construction sont les valeurs des propriétés données.

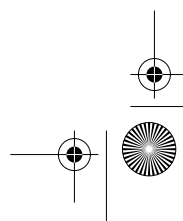
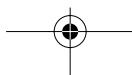
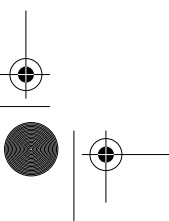
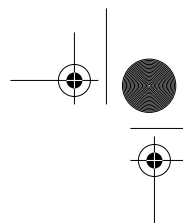
- `protected Expression instantiate(Object oldInstance, Encoder out)`
Renvoie une expression pour appeler le constructeur sans paramètres ou avec les valeurs des propriétés spécifiées dans le constructeur.
- `protected void initialize(Class<?> type, Object oldInstance, Object newInstance, Encoder out)`
Ecrit des instructions pour `out` qui appliquent des éléments de définition de propriétés à `newInstance`, en essayant de le transformer en un objet équivalent à `oldInstance`.

java.beans.Expression 1.4

- `Expression(Object value, Object target, String methodName, Object[] parameters)`
Construit une expression qui appelle la méthode donnée sur `target`, avec les paramètres donnés. Le résultat de cette expression doit être `value`. Pour appeler un constructeur, `target` doit être un objet `Class`, et `methodName` doit être "new".

java.beans.Statement 1.4

- `Statement(Object target, String methodName, Object[] parameters)`
Construit une instruction qui appelle la méthode donnée sur `target`, avec les paramètres donnés.



7

La sécurité

Au sommaire de ce chapitre

- ✓ Les chargeurs de classe
- ✓ La vérification des bytecodes
- ✓ Les gestionnaires de sécurité et les permissions
- ✓ Les signatures numériques
- ✓ La signature de code
- ✓ Le cryptage

Lorsque la technologie Java a fait son apparition, l'intérêt qu'elle a suscité ne concernait pas tant la venue d'un langage de programmation bien conçu que la possibilité de pouvoir exécuter en toute sécurité les applets livrées sur Internet (voir Chapitre 10 du Volume 1 pour plus d'informations sur les applets). De toute évidence, le téléchargement d'applets exécutables n'est pratique que si les destinataires sont certains que le code ne va pas provoquer des ravages sur leurs machines. Pour cette raison, la sécurité a toujours été et reste le souci essentiel à la fois des concepteurs et des utilisateurs de la technologie Java. Cela signifie que, contrairement aux autres langages et systèmes pour lesquels la sécurité a été implémentée après coup ou à la suite de problèmes avérés, les mécanismes de sécurité sont partie intégrante de la technologie Java.

Trois mécanismes permettent d'assurer la sécurité dans la technologie Java :

- Les fonctionnalités de conception du langage (contrôle de limites pour les tableaux, conversions de type légal seulement, pas d'arithmétique de pointeur, etc.).
- Un mécanisme de contrôle d'accès qui limite ce que peut faire le code (accès aux fichiers, accès au réseau, etc.).
- La signature du code, qui permet aux auteurs du code d'utiliser des algorithmes de cryptage standard pour authentifier le code Java. Les utilisateurs du code peuvent ensuite déterminer exactement qui a créé le code et s'il a été altéré après signature.

524 Au cœur de Java 2 - Fonctions avancées

La machine virtuelle Java vérifie la présence de pointeurs non valides, de décalages incorrects dans les tableaux, etc. Les autres étapes impliquent le contrôle de ce qui va vers la machine virtuelle Java.

Lorsque les fichiers de classe sont chargés dans la machine virtuelle, leur intégrité est vérifiée. Nous allons vous montrer en détail comment fonctionne le processus et comment implémenter votre propre *chargeur de classe*.

Pour une sécurité maximale, le mécanisme par défaut de chargement d'une classe et un chargeur de classe personnalisés doivent tous deux fonctionner avec une classe du *gestionnaire de sécurité* qui contrôle les actions que le code peut exécuter. Vous allez apprendre à écrire votre propre classe de gestionnaire de sécurité.

Enfin, vous verrez les algorithmes de cryptographie fournis dans le package `java.security`, qui permettent la signature du code et l'authentification de l'utilisateur.

Comme toujours, nous insistons sur ces sujets qui sont de première importance pour les programmeurs d'application. Pour une étude plus approfondie, vous pouvez vous reporter à l'ouvrage (en langue anglaise) *Inside Java 2 Platform Security* de Li Gong (Addison-Wesley, 1999).

Les chargeurs de classe

Un compilateur Java convertit le code source dans le langage machine d'une machine hypothétique, appelée *machine virtuelle*. Le code de la machine virtuelle est stocké dans un fichier de classe avec une extension `.class`. Chaque fichier de classe contient la définition et le code d'implémentation pour une classe ou une interface. Ces fichiers de classe doivent être interprétés par un programme pouvant traduire le jeu d'instructions de la machine virtuelle dans le langage machine de la machine cible.

Notez que la machine virtuelle ne charge que les fichiers de classe nécessaires à l'exécution d'un programme. Supposons par exemple que l'exécution du programme commence par `MyProgram.class`. Voici les étapes accomplies par la machine virtuelle.

1. La machine virtuelle possède un mécanisme de chargement des fichiers de classe, par exemple, lecture des fichiers sur le disque ou récupération sur le Web ; elle utilise ce mécanisme pour charger le contenu du fichier de la classe `MyProgram`.
2. Si la classe `MyProgram` possède des champs de données ou des superclasses d'un autre type de classe, ces fichiers de classe sont également chargés. (Le processus de chargement de toutes les classes dont dépend une classe donnée est appelé *résolution* de la classe.)
3. La machine virtuelle exécute alors la méthode `main` dans `MyProgram` (qui est statique, aucune instance de classe n'a donc besoin d'être créée).
4. Si la méthode `main`, ou une méthode que `main` appelle, a besoin de classes supplémentaires, elles sont chargées ensuite.

Cependant, le mécanisme de chargement des classes n'utilise pas qu'un seul chargeur de classe. Chaque programme Java possède au moins trois chargeurs de classe :

- le chargeur de classe d'amorce ;
- le chargeur de classe d'extension ;
- le chargeur de classe système (également appelé quelquefois chargeur de classe d'application).

Le chargeur de classe d'amorce charge les classes système (généralement à partir du fichier JAR `rt.jar`). Il s'agit d'une partie intégrante de la machine virtuelle, et elle est généralement implémentée en C. Il n'existe aucun objet `ClassLoader` correspondant au chargeur de classe d'amorce. Par exemple :

```
String.class.getClassLoader()
```

renvoie `null`.

Le chargeur de classe d'extension charge des extensions standard à partir du répertoire `jre/lib/ext`. Vous pouvez déposer les fichiers JAR dans ce répertoire et le chargeur de classe d'extension y trouvera les classes, même sans qu'un chemin de classe soit fourni (certaines personnes recommandent ce mécanisme afin d'éviter un "chemin de la classe illogique", mais vous découvrirez des avertissements plus loin dans cette section).

Le chargeur de classe système charge les classes d'application. Il les localise dans les répertoires et fichiers JAR/ZIP du chemin de la classe, tel que le définit la variable d'environnement `CLASSPATH` ou l'option de ligne de commande `-classpath`.

Dans l'implémentation Java de Sun, l'extension et les chargeurs de classe système sont implémentés en Java. Tous deux sont des instances de la classe `URLClassLoader`.

ATTENTION

Vous risquez d'avoir des problèmes si vous déposez un fichier JAR dans le répertoire `jre/lib/ext` et que l'une de ses classes doit charger une classe qui n'est ni une classe système ni une classe d'extension. Le chargeur de classe d'extension *n'utilise pas le chemin de classe*. Ne l'oubliez pas au moment d'utiliser le répertoire de l'extension pour gérer vos fichiers de classe.

ATTENTION

Il existe un second risque lorsque l'on dépose des fichiers JAR dans le répertoire `jre/lib/ext`. Parfois, les programmeurs oublient les fichiers qu'ils y ont placés quelques mois auparavant. Ils s'interrogent ensuite lorsque le chargeur de classe semble ignorer le chemin de classe, alors qu'il charge en fait d'anciennes classes du répertoire des extensions.

Les chargeurs de classe ont une relation *parent/enfant*. Chaque chargeur de classe, à l'exception du chargeur de classe d'amorce, possède un chargeur de classe parent. Un chargeur de classe est supposé donner à son parent une chance de charger toute classe donnée ; il ne la chargera ensuite que si le parent a échoué. Par exemple, lorsque le chargeur de classe système doit charger une classe système (comme `java.util.ArrayList`), il demande en premier le chargeur de classe d'extension. Celui-ci demande d'abord le chargeur de classe d'amorce. Il trouve et charge la classe dans `rt.jar` et les autres chargeurs de classe ne recherchent pas plus loin.

INFO

Lors de l'implémentation d'un chargeur de classe, délégez toujours le chargement des classes au parent en premier lieu. Vous courrez sinon un risque de sécurité : le chargeur personnalisé peut charger accidentellement une version d'une classe système qui ignore les vérifications de sécurité importantes.

526 *Au cœur de Java 2 - Fonctions avancées*

Des applets, servlets et stubs (éléments de remplacement) RMI sont chargés avec des chargeurs de classe personnalisés. Vous pouvez même écrire votre propre chargeur de classe pour des objectifs particuliers.

Il vous permet d'exécuter des contrôles de sécurité spécialisés avant de passer les bytecodes à la machine virtuelle. Vous pouvez par exemple écrire un chargeur de classe qui refusera de charger une classe n'ayant pas été marquée comme "payée". La section suivante vous montre comment procéder.

Le plus souvent, vous n'aurez pas à vous inquiéter des chargeurs de classes. La plupart des classes sont chargées car elles sont nécessaires à d'autres, mais cette procédure est invisible pour vous.

Si vous chargez une classe par le biais de la programmation, en appelant `Class.forName`, la nouvelle classe est chargée avec le chargeur qui a chargé le code appelant `Class.forName`. Ce comportement fonctionne généralement bien. Il échoue toutefois dans les cas suivants :

1. Vous implémentez une classe de bibliothèque avec une méthode appelant `Class.forName`.
2. Votre méthode est appelée à partir d'une classe d'application chargée avec un autre chargeur que la classe de bibliothèque.
3. La classe chargée n'est pas visible depuis le chargeur qui a chargé la classe de bibliothèque.

Dans ce cas, la classe de bibliothèque doit travailler plus dur et récupérer le chargeur de classe de l'application.

```
Thread t = Thread.currentThread();
ClassLoader loader = t.getContextClassLoader();
Class cl = loader.loadClass(className);
```

Utiliser des chargeurs de classe comme espaces de nom

Tous les programmeurs Java savent que les noms des packages permettent d'éliminer des conflits de noms. Il existe ainsi deux classes appelées `Date` dans la bibliothèque standard mais, bien entendu, leurs noms réels sont `java.util.Date` et `java.sql.Date`. Le nom simple n'est qu'une commodité des programmeurs et nécessite l'inclusion de déclarations `import` appropriées. Dans un programme en cours, tous les noms de classe contiennent leur nom de package.

Cela peut vous surprendre, mais deux classes de la même machine virtuelle peuvent avoir les mêmes noms de classe *et de package*. Une classe se détermine par son nom complet *et* par le chargeur de classe. Cette technique est utile lors du chargement du code de plusieurs sources. Par exemple, un navigateur utilise plusieurs instances du chargeur de la classe d'applet pour chaque page Web. Cela permet à la machine virtuelle de séparer des classes de différentes pages Web, quel que soit leur nom.

INFO

Cette technique possède aussi d'autres utilisations, comme un "déploiement à chaud" des servlets et Enterprise Java Beans. Reportez-vous à l'adresse <http://java.sun.com/developer/TechTips/2000/tt1027.html> pour plus d'informations.

Ecrire votre propre chargeur de classe

Pour écrire votre propre chargeur de classe, étendez simplement la classe `ClassLoader` et surchargez la méthode :

```
findClass(String className)
```

La méthode `loadClass` de la superclasse `ClassLoader` prend soin de la délégation au parent et n'appelle `findClass` que si la classe n'a pas déjà été chargée et si le chargeur de classe parent a été incapable de charger la classe.

INFO

Dans les versions précédentes du JDK, les programmeurs devaient surcharger la méthode `loadClass`. Cette pratique n'est plus recommandée.

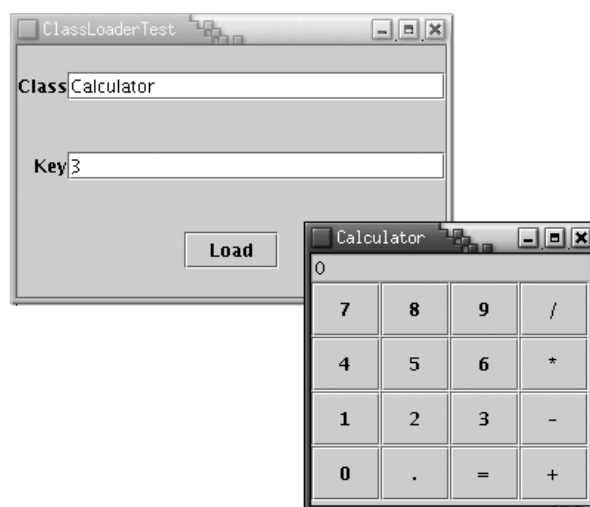
Pour l'implémentation de cette méthode :

1. Vous devez charger les bytecodes pour la classe à partir du système de fichiers local ou d'une autre source.
2. Vous devez appeler la méthode `defineClass` de la superclasse `ClassLoader` pour présenter les bytecodes à la machine virtuelle.

Dans le programme de l'Exemple 7.1, nous implémentons un chargeur de classe qui charge des fichiers de classe cryptés. Le programme demande à l'utilisateur le nom de la première classe à charger (c'est-à-dire la classe contenant `main`) et la clé de décryptage. Il utilise ensuite un chargeur de classe spécial pour charger la classe spécifiée et appelle la méthode `main`. Le chargeur de classe décrypte la classe spécifiée et toutes les classes non système qui sont référencées par elle. Enfin, le programme appelle la méthode `main` de la classe chargée (voir Figure 7.1).

Figure 7.1

Le programme
ClassLoaderTest.



528 Au cœur de Java 2 - Fonctions avancées

Pour simplifier, nous allons ignorer 2 000 ans de progrès dans le domaine de la cryptographie et utiliser la vénérable clé *Caesar* pour crypter les fichiers de classe.

INFO

Le merveilleux livre de David Kahn *The Code Breakers* (Macmillan, NY, 1967, p. 84) fait référence à Suétone comme source historique de la clé Caesar. Le code Caesar consiste à décaler les 24 lettres de l'alphabet romain de trois lettres. A la première rédaction de ce livre, le gouvernement des Etats-Unis restreignait sévèrement l'exportation de méthodes sérieuses de cryptage. C'est pourquoi nous utilisons la méthode Caesar pour notre exemple puisqu'elle ne posait pas de problème au niveau de la législation.

Notre version du code Caesar a pour clé un nombre compris entre 1 et 255. Pour décrypter, ajoutez simplement cette clé à chaque octet et réduisez avec un modulo 256. Le programme `Caesar.java` de l'Exemple 7.2 exécute le cryptage.

Pour ne pas perturber le chargeur de classe normal, nous utilisons une extension différente, `.caesar`, pour les fichiers de la classe cryptée.

Pour décrypter, le chargeur de classe soustrait simplement la clé de chaque octet. Sur le code accompagnant ce livre, vous trouverez quatre fichiers de classe, cryptés avec une valeur de clé de 3 — le choix traditionnel. Pour exécuter le programme crypté, utilisez le chargeur de classe personnalisé défini dans notre programme `ClassLoaderTest`.

Le cryptage de fichiers de classe a quantité d'utilisations pratiques (en supposant, bien entendu, que vous utilisiez une clé plus efficace que le code Caesar). Sans la clé de décryptage, les fichiers de classe sont inutilisables. Ils ne peuvent être ni exécutés par une machine virtuelle standard ni désassemblés facilement.

Cela signifie que vous pouvez utiliser un chargeur de classe personnalisé pour authentifier l'utilisateur de la classe, ou pour vous assurer qu'un programme a bien été payé avant d'autoriser son exécution. Bien entendu, le cryptage n'est qu'une application possible d'un chargeur de classe personnalisé. Vous pouvez utiliser d'autres types de chargeurs de classes pour régler d'autres problèmes, par exemple pour stocker des fichiers de classe dans une base de données.

Exemple 7.1 : `ClassLoaderTest.java`

```
import java.util.*;
import java.io.*;
import java.lang.reflect.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Ce programme montre un chargeur de classe personnalisé
 * qui décrypte les fichiers de classe.
 */
public class ClassLoaderTest
{
    public static void main(String[] args)
    {

```

```

        JFrame frame = new ClassLoaderFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc contient deux champs de texte pour le nom et
 * la classe à charger ainsi que la clé de décryptage.
 */
class ClassLoaderFrame extends JFrame
{
    public ClassLoaderFrame()
    {
        setTitle("ClassLoaderTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
        setLayout(new GridBagLayout());
        add(new JLabel("Classe"), new GBC(0, 0).setAnchor(GBC.EAST));
        add(nameField, new GBC(1, 0).setWeight(
            100, 0).setAnchor(GBC.WEST));
        add(new JLabel("Clé"), new GBC(0, 1).setAnchor(GBC.EAST));
        add(keyField, new GBC(1, 1).setWeight(100, 0).setAnchor(GBC.WEST));
        JButton loadButton = new JButton("Charger");
        add(loadButton, new GBC(0, 2, 2, 1));
        loadButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    runClass(nameField.getText(), keyField.getText());
                }
            });
        pack();
    }

    /**
     * Lance la méthode principale d'une classe donnée.
     * @param name le nom de la classe
     * @param key la clé de décryptage pour les fichiers de classe
     */
    public void runClass(String name, String key)
    {
        try
        {
            ClassLoader loader
                = new CryptoClassLoader(Integer.parseInt(key));
            Class c = loader.loadClass(name);
            String[] args = new String[] {};

            Method m = c.getMethod("main", args.getClass());
            m.invoke(null, (Object) args);
        }
        catch (Throwable e)
        {
            JOptionPane.showMessageDialog(this, e);
        }
    }
}

```


530 *Au cœur de Java 2 - Fonctions avancées*

```

private JTextField keyField = new JTextField("3", 4);
private JTextField nameField = new JTextField(30);
private static final int DEFAULT_WIDTH = 300;
private static final int DEFAULT_HEIGHT = 200;
}

/**
 * Ce chargeur de classe charge des fichiers de classe cryptés.
 */
class CryptoClassLoader extends ClassLoader
{
    /**
     * Construit un chargeur de classe crypté.
     * @param k la clé de décryptage
     */
    public CryptoClassLoader(int k)
    {
        key = k;
    }

    protected Class findClass(String name)
        throws ClassNotFoundException
    {
        byte[] classBytes = null;
        try
        {
            classBytes = loadClassBytes(name);
        }
        catch (IOException e)
        {
            throw new ClassNotFoundException(name);
        }
        Class cl = defineClass(name, classBytes,
            0, classBytes.length);
        if (cl == null)
            throw new ClassNotFoundException(name);
        return cl;
    }

    /**
     * Charge et décrypte les octets du fichier de classe.
     * @param name le nom de classe
     * @return un tableau avec les octets du fichier de classe
     */
    private byte[] loadClassBytes(String name)
        throws IOException
    {
        String cname = name.replace('.', '/') + ".caesar";
        FileInputStream in = null;
        in = new FileInputStream(cname);
        try
        {
            ByteArrayOutputStream buffer
                = new ByteArrayOutputStream();
            int ch;

```

```

        while ((ch = in.read()) != -1)
        {
            byte b = (byte) (ch - key);
            buffer.write(b);
        }
        in.close();
        return buffer.toByteArray();
    }
    finally
    {
        in.close();
    }
}

private int key;
}

```

Exemple 7.2 : Caesar.java

```

import java.io.*;

/**
 * Crypte un fichier à l'aide du code Caesar.
 */
public class Caesar
{
    public static void main(String[] args)
    {
        if (args.length != 3)
        {
            System.out.println("USAGE: clé entrée sortie Caesar java");
            return;
        }

        try
        {
            FileInputStream in = new FileInputStream(args[0]);
            FileOutputStream out = new FileOutputStream(args[1]);
            int key = Integer.parseInt(args[2]);
            int ch;
            while ((ch = in.read()) != -1)
            {
                byte c = (byte)(ch + key);
                out.write(c);
            }
            in.close();
            out.close();
        }
        catch (IOException exception)
        {
            exception.printStackTrace();
        }
    }
}

```

532 *Au cœur de Java 2 - Fonctions avancées*

java.lang.Class 1.0

- `ClassLoader getClassLoader()`
Récupère le chargeur de classe qui a chargé cette classe.

java.lang.ClassLoader 1.0

- `ClassLoader getParent()` 1.2
Renvoie le chargeur de classe parent ou `null` s'il s'agit d'un chargeur de classe d'amorce.
- `static ClassLoader getSystemClassLoader()` 1.2
Récupère le chargeur de classe système, c'est-à-dire celui utilisé pour charger la première classe d'application.
- `protected Class findClass(String name)` 1.2
Devrait être surchargé par un chargeur de classe pour retrouver les bytecodes pour une classe et les présenter à la machine virtuelle en appelant la méthode `defineClass`.

Paramètres : `name` le nom de la classe. Utilisez `.` comme séparateur du nom de package, mais pas de suffixe `.class`.

- `Class defineClass(String name, byte data[], int offset, int length)`
Ajoute une nouvelle classe à la machine virtuelle.

Paramètres : `name` le nom de la classe. Utilisez `.` comme séparateur de nom de package, et n'utilisez pas de suffixe `.class`

`data` un tableau contenant les bytecodes de la classe

`offset` le début des bytecodes dans le tableau

`length` la longueur des bytecodes dans le tableau

java.lang.Thread 1.0

- `ClassLoader getContextClassLoader()` 1.2
Récupère le chargeur de classe que le créateur de ce thread a désigné comme chargeur le plus raisonnable à utiliser lors de l'exécution de ce thread.
- `void setContextClassLoader(ClassLoader loader)` 1.2
Définit un chargeur de classe pour le code de ce thread à récupérer pour charger les classes. Lorsque aucun chargeur de classe contexte n'est défini de manière explicite au moment de démarrer un thread, le chargeur de classe contexte est utilisé.

La vérification des bytecodes

Lorsqu'un chargeur de classe présente à la machine virtuelle les bytecodes d'une classe de plateforme Java nouvellement chargée, ces bytecodes sont d'abord inspectés par un *vérificateur*. Le vérificateur s'assure que les instructions ne pourront pas réaliser d'actions qui seraient de toute évidence dommageables. Toutes les classes, sauf les classes système, sont vérifiées. Vous pouvez toutefois désactiver la vérification à l'aide de l'option `-noverify`.

Par exemple,

```
java -noverify Hello
```

Le vérificateur contrôle entre autres :

- que les variables sont initialisées avant leur utilisation ;
- que les appels de méthode correspondent aux types de références d'objet ;
- que les règles d'accès aux méthodes et données privées ne sont pas violées ;
- que les accès à la variable locale tombent bien dans la pile d'exécution ;
- qu'il n'y a pas dépassement de la pile d'exécution.

Si l'une de ces vérifications échoue, la classe est considérée comme corrompue et elle ne sera pas chargée.

INFO

Si vous connaissez le théorème de Gödel, vous vous demandez peut-être comment le vérificateur peut prouver qu'un fichier de classe est exempt d'incohérences de type, de variables non initialisées ou de dépassement de la pile. Le théorème de Gödel dit qu'il est impossible de concevoir des algorithmes dont les entrées soient des fichiers de programme et les sorties une valeur booléenne qui indique si le programme en entrée a une propriété particulière (comme être exempt de variables non initialisées). Est-ce un conflit entre le service des Relations publiques de Sun Microsystems et les lois de la logique ? Non. En fait, le vérificateur n'est *pas* un algorithme de décision dans l'esprit de Gödel. Si le vérificateur accepte un programme, il est vraiment sécurisé. Il peut toutefois exister de nombreux programmes rejetés par le vérificateur, bien qu'ils soient en réalité sans danger.

Cette vérification stricte est un élément de sécurité important. Des erreurs accidentelles, telles que des variables non initialisées, peuvent facilement faire des dégâts si elles ne sont pas interceptées. Plus important encore, dans le monde sans frontières d'Internet, vous devez être protégé contre les programmeurs malveillants qui provoquent intentionnellement des dommages. Par exemple, en modifiant des valeurs dans la pile d'exécution, ou en écrivant dans les champs de données privées des objets système, un programme peut mettre en péril le système de sécurité d'un navigateur.

Vous pouvez toutefois vous interroger sur la nécessité d'avoir un vérificateur spécial pour contrôler toutes ces fonctionnalités. Après tout, le compilateur ne vous permettra pas de générer un fichier de classe comprenant une variable non initialisée, ou dans lequel un champ de données privées est accessible par une autre classe. Un fichier de classe généré par un compilateur pour le langage Java doit toujours satisfaire à des vérifications. Cependant, le format de bytecode utilisé dans les fichiers de classe est bien documenté, et pour quelqu'un ayant quelque expérience en programmation Assembleur et disposant d'un éditeur hexa, il est relativement facile de produire manuellement un fichier de classe contenant des instructions valides, mais non sécurisées, pour la machine virtuelle Java. Gardez bien à l'esprit que le vérificateur protège en permanence contre des altérations malveillantes des fichiers de classe, et ne se contente pas de vérifier simplement les fichiers de classe produits par un compilateur.

Voici un exemple de la façon de construire un tel fichier de classe altéré. Nous partons du programme `VerifierTest.java` de l'Exemple 7.3. C'est un programme très simple qui appelle une

534 Au cœur de Java 2 - Fonctions avancées

méthode et affiche le résultat de la méthode. Ce programme peut être exécuté à la fois comme programme console ou comme applet. La méthode `fun` elle-même additionne simplement $1 + 2$.

```
static int fun()
{
    int m;
    int n;
    m = 1;
    n = 2;
    int r = m + n;
    return r;
}
```

A titre d'expérience, essayez de compiler la modification suivante de ce programme :

```
static int fun()
{
    int m = 1;
    int n;
    m = 1;
    m = 2;
    int r = m + n;
    return r;
}
```

Ici, `n` n'est pas initialisé. Il peut avoir n'importe quelle valeur aléatoire. Bien entendu, le compilateur détecte ce problème et refuse de compiler le programme. Pour créer un fichier de classe erroné, nous devons procéder autrement. Exécutez tout d'abord le programme `javap` pour trouver comment le compilateur traduit la méthode `fun`. La commande

```
javap -c VerifierTest
```

montre les bytecodes dans le fichier de classe, sous une forme mnémonique.

```
Method int fun()
  0 iconst_1
  1 istore_0
  2 iconst_2
  3 istore_1
  4 iload_0
  5 iload_1
  6 iadd
  7 istore_2
  8 iload_2
  9 ireturn
```

Nous utilisons un éditeur hexa pour modifier l'instruction 3 qui de `istore_1` devient `istore_0`. C'est-à-dire que la variable locale 0 (qui est `m`) est initialisée deux fois, et la variable locale 1 (qui est `n`) n'est pas initialisée du tout. Nous devons connaître les valeurs hexadécimales de ces instructions. Ces valeurs sont facilement disponibles dans l'ouvrage (en langue anglaise) de Tim Lindholm et Frank Yellin *The Java Virtual Machine Specification* (Addison-Wesley, 1999).

```
0 iconst_1 04
1 istore_0 3B
2 iconst_2 05
3 istore_1 3C
4 iload_0 1A
5 iload_1 1B
6 iadd 60
```

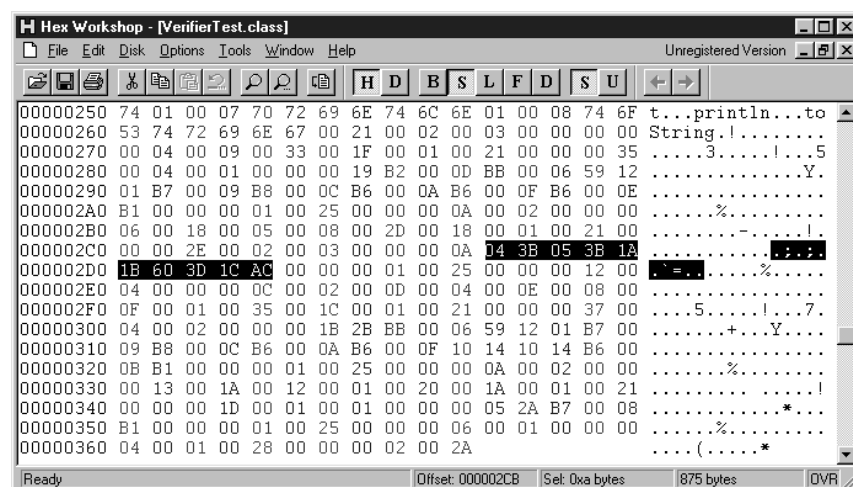
```

7 istore_2 3D
8 iload_2 1C
9 ireturn AC

```

Nous allons utiliser Data Workshop (que vous pouvez télécharger à l'adresse <http://www.dataworkshop.de>) pour réaliser la modification. Vous pouvez également utiliser les emacs en mode hexa1 (hex1-mode). A la Figure 7.2, vous voyez le fichier de classe VerifierTest.class chargé dans Data Workshop, avec les bytecodes de la méthode fun en surbrillance.

Figure 7.2
Modification
des bytecodes
à l'aide d'un
éditeur hexa.



Nous remplaçons simplement 3C par 3B et sauvegardons le fichier de classe. Essayez d'exécuter le programme VerifierTest. Vous obtenez un message d'erreur :

```

Exception in thread "main" java.lang.VerifyError: (class: VerifierTest,
method:fun signature: ()I) Accessing value from uninitialized register 1

```

Ce message signale que la machine virtuelle a détecté notre modification.

Exécutez maintenant le programme avec l'option -noverify (ou -Xverify:none).

```
java -noverify VerifierTest
```

La méthode fun renvoie ce qui semble être une valeur aléatoire. C'est en réalité 2 plus la valeur qui se trouvait stockée dans la variable n, qui n'a jamais été initialisée. Voici un exemple de sortie obtenue.

```
1 + 2 = 15102330
```

Pour voir comment les navigateurs gèrent la vérification, nous avons écrit ce programme pour qu'il tourne soit en tant qu'application, soit en tant qu'applet. Chargez l'applet dans un navigateur, à l'aide d'une URL de fichier comme

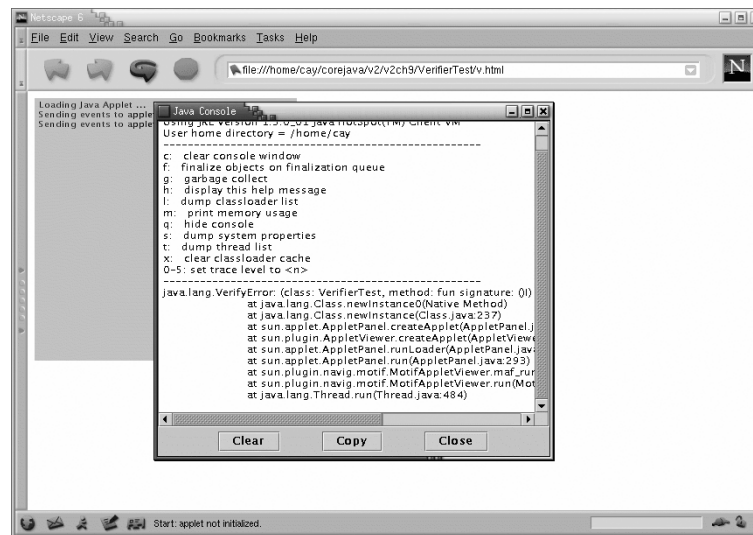
```
file:///C:/CoreJavaBook/v2ch9/VerifierTest/VerifierTest.html
```

Vous verrez apparaître un message d'erreur indiquant que la vérification a échoué (voir Figure 7.3).

536 Au cœur de Java 2 - Fonctions avancées

Figure 7.3

Le chargement d'un fichier de classe corrompu a provoqué une erreur de vérification de la méthode.



INFO

Il existe un manque curieux dans le vérificateur de la machine virtuelle Java connu depuis longtemps mais qui demeure pour des raisons de compatibilité. Supposons que vous ayez deux fichiers de classe A. java et B. java comme ceci :

```
public class A { public int field; }

public class B
{
    public static void main(String[] args)
    {
        System.out.println(new A().field);
    }
}
```

Ce programme se compile et s'exécute, bien entendu. Modifiez maintenant *uniquement* le fichier A. java pour rendre field privé. Recompilez uniquement ce fichier. Le programme qui en résulte ne devrait pas passer la vérification puisque B essaie maintenant d'accéder à un champ privé de A. Toutefois, jusqu'au JDK 5.0, le programme s'exécutera et accèdera au champ privé. Ce n'est que si vous exécutez la machine virtuelle Java avec l'option `-verify` que l'erreur sera détectée.

La raison en est une compatibilité en amont et une optimisation obscure du JDK 1.0 ; reportez-vous à l'adresse <http://bugs.sun.com/developer/bugParade/bugs/4030988.html>. Ce problème n'est pas considéré comme un risque de sécurité car seules les classes chargées à partir d'un système de fichier local sont exemptes de vérification.

Exemple 7.3 : VerifierTest.java

```
import java.awt.*;
import java.applet.*;

/**
 * Cette application montre le vérificateur de bytecode de la
 * machine virtuelle. Si vous utilisez un éditeur hexa pour modifier
```

```
le fichier de classe, la machine virtuelle devrait détecter
la falsification.
*/
public class VerifierTest extends Applet
{
    public static void main(String[] args)
    {
        System.out.println("1 + 2 == " + fun());
    }

    /**
     * Une fonction qui calcule 1 + 2
     * @return 3, si le code n'a pas été corrompu
     */
    public static int fun()
    {
        int m;
        int n;
        m = 1;
        n = 2;
        // éditeur hexa utilisé pour remplacer par "m = 2"
        // dans le fichier de classe
        int r = m + n;
        return r;
    }

    public void paint(Graphics g)
    {
        g.drawString("1 + 2 == " + fun(), 20, 20);
    }
}
```

Les gestionnaires de sécurité et les permissions

Lorsqu'une classe a été chargée dans la machine virtuelle, par un chargeur de classe ou par le mécanisme de chargement par défaut, puis vérifiée par le vérificateur, le troisième mécanisme de sécurité de la plate-forme Java entre en action : le *gestionnaire de sécurité*. Un gestionnaire de sécurité est une classe qui contrôle si une opération spécifique est permise. Le gestionnaire de sécurité vérifie entre autres :

- si le thread courant peut créer un nouveau chargeur de classe ;
- si le thread courant peut arrêter la machine virtuelle ;
- si une classe peut accéder à un membre d'une autre classe ;
- si le thread courant peut accéder à un fichier local ;
- si le thread courant peut ouvrir une connexion socket vers un hôte externe ;
- si une classe peut démarrer un travail d'impression ;
- si une classe peut accéder au Presse-papiers du système ;
- si une classe peut accéder à la file d'attente d'événement AWT ;
- si le thread courant est habilité à afficher une fenêtre visible.

Il existe de nombreuses autres vérifications de ce type dans la bibliothèque Java.

538 Au cœur de Java 2 - Fonctions avancées

Le comportement par défaut lors de l'exécution d'applications Java est qu'*aucun* gestionnaire de sécurité n'est installé, afin que toutes ces opérations soient permises. L'éditeur d'applets, en revanche, installe immédiatement un gestionnaire de sécurité qui est assez restrictif.

Par exemple, les applets ne sont pas autorisées à quitter la machine virtuelle. Si elles essaient d'appeler la méthode `exit`, une exception de sécurité est lancée. En fait, la méthode `exit` de la classe `Runtime` appelle la méthode `checkExit` du gestionnaire de sécurité. Voici la totalité du code de la méthode `exit`.

```
public void exit(int status)
{
    SecurityManager security = System.getSecurityManager();
    if (security != null)
        security.checkExit(status);
    exitInternal(status);
}
```

Le gestionnaire de sécurité vérifie ensuite si la demande de sortie provient du navigateur ou d'une applet individuelle. Si le gestionnaire de sécurité est d'accord avec la demande de sortie, la méthode `checkExit` retourne simplement vers l'appelant et le traitement normal se poursuit. Toutefois, si le gestionnaire de sécurité ne veut pas accepter la demande, la méthode `checkExit` lance une exception `SecurityException`.

La méthode `exit` se poursuit seulement si aucune exception ne se produit. Elle appelle ensuite la méthode *native privée* `exitInternal` qui quitte réellement la machine virtuelle. Il n'y a pas d'autre moyen de quitter la machine virtuelle et, la méthode `exitInternal` étant privée, elle ne peut être appelée par aucune autre classe. Par conséquent, tout code tentant de quitter la machine virtuelle doit passer par la méthode `exit` et satisfaire au contrôle de sécurité de `checkExit` sans déclencher d'exception de sécurité.

En clair, l'intégrité des règles de sécurité dépend d'un codage minutieux. Les fournisseurs de services système dans la bibliothèque standard doivent s'assurer de toujours consulter le gestionnaire de sécurité avant de tenter toute opération délicate.

Lorsque vous exécutez une application Java, aucun gestionnaire de sécurité ne s'exécute par défaut. Votre programme peut installer un gestionnaire de sécurité spécifique par un appel à la méthode statique `setSecurityManager` de la classe `System`. Lorsque votre programme a installé un gestionnaire de sécurité, toute tentative d'installation d'un second gestionnaire de sécurité ne peut aboutir que si le premier gestionnaire accepte d'être remplacé. Cela est essentiel ; sinon une applet non valide pourrait installer son propre gestionnaire de sécurité. Par conséquent, bien qu'il soit possible d'avoir plusieurs chargeurs de classes, un programme Java ne peut être régi que par un seul gestionnaire de sécurité. C'est le responsable de l'implémentation du gestionnaire de sécurité qui décide s'il doit accorder à toutes les classes le même accès, ou s'il doit prendre en compte l'origine des classes avant de décider ce qu'il y a lieu de faire.

Le gestionnaire de sécurité par défaut de la plate-forme Java 2 permet à la fois aux programmeurs et aux administrateurs système un contrôle pointu en ce qui concerne les permissions individuelles de sécurité. Ces fonctionnalités sont décrites dans la section suivante. Nous allons d'abord vous donner un aperçu du modèle de sécurité de la plate-forme Java 2. Puis, nous vous indiquerons comment contrôler les permissions à l'aide des *fichiers de règles (de sécurité)*. Vous verrez enfin comment définir vos propres types de permissions et comment étendre la classe du gestionnaire de sécurité par défaut.

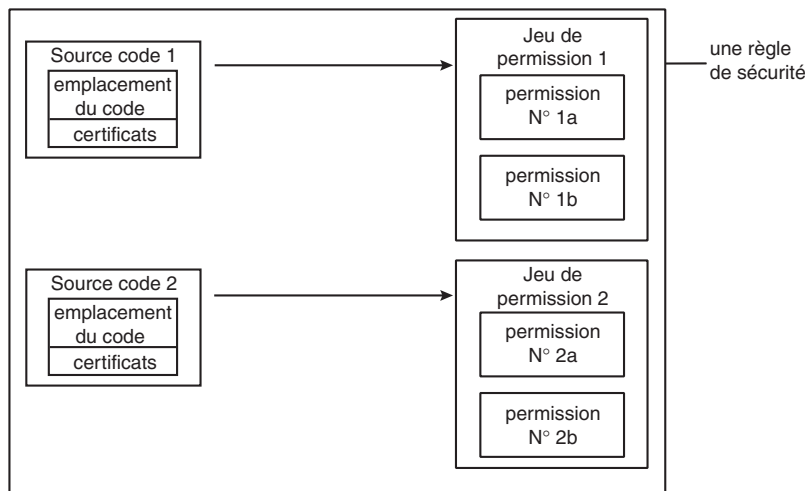
La sécurité de la plate-forme Java 2

Le modèle de sécurité de JDK 1.0 était très simple : les classes locales disposaient de toutes les permissions et les classes distantes étaient confinées au *sandbox* (littéralement "bac à sable"). Sur le modèle de l'enfant qui n'est autorisé à jouer que dans le bac à sable, le code distant ne pouvait dessiner que sur l'écran et interagir avec l'utilisateur. Le gestionnaire de sécurité d'applet refusait tout accès aux ressources locales. JDK 1.1 a implémenté une petite modification : le code distant qui était signé par une entité fiable se voyait attribuer les mêmes permissions que les classes locales. Toutefois, l'approche des deux versions de JDK étaient du type "tout ou rien". Les programmes, soit avaient plein accès, soit devaient se contenter de jouer dans le bac à sable.

La plate-forme Java 2 a un mécanisme beaucoup plus souple. Une *règle de sécurité* (*security policy*) met en correspondance les sources de code avec les jeux de permission (voir Figure 7.4).

Figure 7.4

Une règle de sécurité.



Une *source de code* a deux propriétés : l'emplacement du code (*code location*), par exemple une URL HTTP pour le code distant ou une URL de fichier pour un code local dans un fichier JAR, et les certificats (*certificates*). Vous verrez plus loin dans ce chapitre comment le code peut être certifié par des parties habilitées.

Une *permission* est n'importe quelle propriété vérifiée par un gestionnaire de sécurité. L'implémentation du JDK 1.2 gère un certain nombre de classes de permission, chacune encapsulant les détails d'une permission particulière. Par exemple, l'instance suivante de la classe `FilePermission` précise que n'importe quel fichier peut être lu ou écrit dans le répertoire `/tmp`.

```
FilePermission p = new FilePermission("/tmp/*", "read,write");
```

Plus important encore, l'implémentation par défaut de la classe `Policy` dans JDK 1.2 lit les permissions dans un *fichier de permission*. A l'intérieur d'un fichier de permission, la même autorisation de lecture est exprimée par

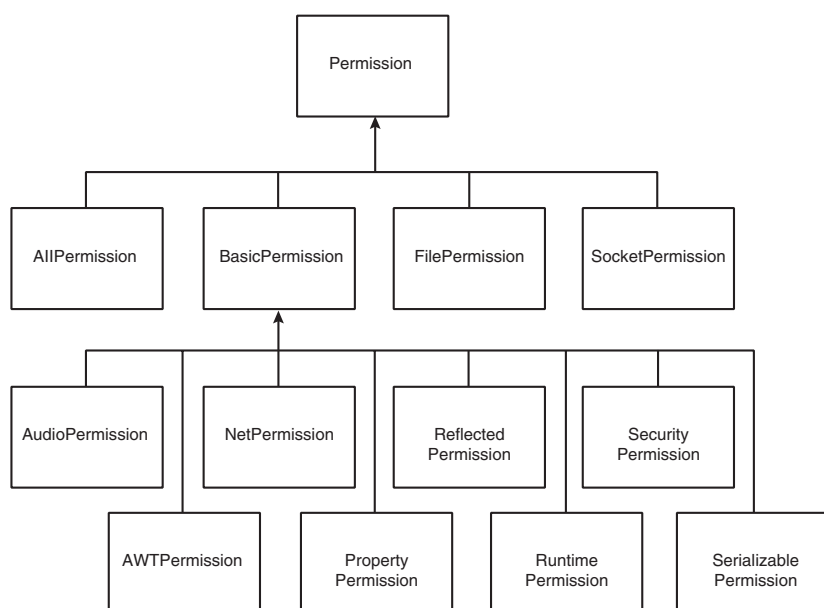
```
permission java.io.FilePermission "/tmp/*", "read,write";
```

Les fichiers de permission sont vus dans la prochaine section.

540 Au cœur de Java 2 - Fonctions avancées

La Figure 7.5 montre la hiérarchie des classes de permission fournies dans le JDK 1.2. Beaucoup d'autres classes de permission ont été ajoutées dans les versions consécutives du JDK.

Figure 7.5
Hiérarchie des classes de permission.



Dans la section précédente, vous avez vu que la classe `SecurityManager` possédait des méthodes de vérification comme `checkExit`. Ces méthodes n'existent que pour faciliter la tâche du programmeur et pour des raisons de compatibilité amont.

Elles correspondent toutes à des vérifications de permission standard. Voici, par exemple, le code source pour la méthode `checkExit`.

```

public void checkExit()
{
    checkPermission(new RuntimePermission("exitVM"));
}
    
```

Chaque gestionnaire de sécurité est libre de fournir sa propre implémentation de la méthode `checkPermission`. JDK fournit cependant un "modèle standard" d'exécution des contrôles de permission. Le reste de cette section décrit ce modèle standard.

Chaque classe a un *domaine de protection*, un objet qui encapsule à la fois la source de code et la collection de permissions de la classe.

Lorsque le gestionnaire de sécurité doit vérifier une permission, il examine les classes de toutes les méthodes actuellement dans la pile d'appel. Il extrait ensuite les domaines de protection de toutes les classes et demande à chacun des domaines si sa collection de permission autorise l'opération actuellement en cours de vérification. Si tous les domaines sont d'accord, le contrôle est positif. Sinon une exception `SecurityException` est lancée.

Pourquoi toutes les méthodes dans la pile d'appel doivent-elles autoriser une opération particulière ? Prenons un exemple. Supposons que la méthode `init` d'une applet veuille ouvrir un fichier. Elle pourrait appeler

```
Reader in = new FileReader(name);
```

Le constructeur de `FileReader` appelle le constructeur de `FileInputStream`, qui appelle la méthode `checkRead` du gestionnaire de sécurité, qui en fin de compte appelle `checkPermission` avec un objet `FilePermission(name, "read")`.

Le Tableau 7.1 montre la pile d'appel.

Tableau 7.1 : La pile d'appel pendant le contrôle de permission

<i>Classe</i>	<i>Méthode</i>	<i>Source de code</i>	<i>Permissions</i>
SecurityManager	SecurityManager	null	AllPermission
SecurityManager	checkRead	null	AllPermission
FileInputStream	constructeur	null	AllPermission
FileReader	constructeur	null	AllPermission
applet	init	source du code de l'applet	permissions de l'applet
...			

Les classes `FileInputStream` et `SecurityManager` sont des *classes système* pour lesquelles `CodeSource` a la valeur `null` et dont les permissions consistent en une instance de la classe `AllPermission`, qui permet toutes les opérations. En clair, leurs permissions seules ne peuvent déterminer le résultat de la vérification. Comme vous le voyez, la méthode `checkPermission` doit prendre en compte les permissions restreintes de la classe d'applet. En vérifiant la totalité de la pile d'appel, le mécanisme de sécurité s'assure qu'aucune classe ne peut jamais demander à une autre classe de réaliser une opération critique pour son compte.

INFO

Cet aperçu du contrôle des permissions vous montre les concepts de base. Il y a toutefois un certain nombre de détails techniques que nous n'avons pas abordés ici. En ce qui concerne la sécurité, le risque se situe au niveau des détails, et nous vous encourageons à lire le livre de Li Gong pour plus d'informations. Pour une approche plus critique du modèle de sécurité de la plate-forme Java, reportez-vous à l'ouvrage (en langue anglaise) de Gary McGraw et Ed Felten, *Securing Java* (John Wiley & Sons, 1999). Vous pouvez trouver une version en ligne de ce livre à l'adresse Web <http://www.securingjava.com>.

java.lang.SecurityManager 1.0

- `void checkPermission(Permission p) 1.2`

Vérifie si ce gestionnaire de sécurité accorde la permission donnée. La méthode déclenche une `SecurityException` si la permission n'est pas accordée.

542 *Au cœur de Java 2 - Fonctions avancées*

java.lang.Class 1.0

- `ProtectionDomain getProtectionDomain()` 1.2
Récupère le domaine de protection pour cette classe ou `null` si cette classe a été chargée sans domaine de protection.

java.security.ProtectionDomain 1.2

- `ProtectionDomain(CodeSource source, PermissionCollection permissions)`
Construit un domaine de protection avec la source de code et les permissions données.
- `CodeSource getCodeSource()`
Récupère la source de code de ce domaine de protection.
- `boolean implies(Permission p)`
Renvoie `true` si la permission donnée est autorisée par ce domaine de protection.

java.security.CodeSource 1.2

- `Certificate[] getCertificates()`
Récupère les certificats pour la signature du fichier de classe associé à cette source de code.
- `URL getLocation()`
Récupère l'emplacement des fichiers de classe associés à cette source de code.

Les fichiers de règles de sécurité

Dans la section précédente, vous avez vu comment le chargeur `SecureClassLoader` affecte des permissions lors du chargement de classes, en demandant à un objet `Policy` de vérifier les permissions pour la source de code de chaque classe. Vous pouvez en principe installer votre propre classe `Policy` pour réaliser la mise en correspondance des sources de code avec les permissions. Toutefois, dans cette section, vous allez étudier la classe `Policy` standard que la machine virtuelle Java 1.2 utilise.

INFO

La classe `Policy` est définie dans le fichier `java.security` dans le sous-répertoire `jre/lib/security` du répertoire personnel (*home directory*) JDK. Par défaut, ce fichier contient la ligne

```
policy.provider=sun.security.provider.PolicyFile
```

Vous pouvez fournir votre propre classe `Policy` et l'installer en modifiant ce fichier.

La règle standard lit les fichiers *policy* (*fichiers de règles de sécurité*) qui contiennent les instructions pour la mise en correspondance des sources de code avec les permissions. Vous avez vu ces fichiers dans le Chapitre 5, où ils permettaient aux programmes utilisant `RMISecurityManager` d'accéder au réseau. Voici un fichier de règles de sécurité typique.

```
grant codeBase "http://www.horstmann.com/classes"
{
    permission java.io.FilePermission "/tmp/*", "read,write";
}
```

Ce fichier accorde les permissions de lecture et d'écriture des fichiers dans le répertoire `/tmp` à tout code téléchargé à partir de <http://www.horstmann.com/classes>.

Vous pouvez installer ces fichiers de règles aux emplacements standard. Par défaut, il existe deux emplacements :

- le fichier `java.policy` dans le répertoire *home* de la plate-forme Java ;
- le fichier `.java.policy` (remarquez le point au début du nom de fichier) dans le répertoire personnel de l'utilisateur.

INFO

Vous pouvez modifier les emplacements de ces fichiers dans le fichier de configuration `java.security`. Les valeurs par défaut sont :

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

Un administrateur système peut modifier le fichier `java.security` et spécifier les URL des règles qui résident sur un autre serveur et qui ne peuvent être modifiées par les utilisateurs. Il peut y avoir n'importe quel nombre d'URL de règles (avec des numéros consécutifs) dans le fichier des règles. Les permissions de tous les fichiers sont combinées.

Pendant la phase de test, nous ne souhaitons pas modifier constamment ces fichiers standard. C'est pourquoi nous allons nommer explicitement le fichier de règles qui est requis pour chaque application. Placez simplement les permissions dans un fichier séparé, par exemple `MyApp.policy`, et lancez la machine virtuelle de la façon suivante :

```
java -Djava.security.policy=MyApp.policy MyApp
```

Pour les applets, vous utiliserez

```
appletviewer -J-Djava.security.policy=MyApplet.policy MyApplet.html
```

(Vous pouvez utiliser l'option `-J` de `appletviewer` pour passer tout argument de ligne de commande à la machine virtuelle.)

Dans ces exemples, le fichier `MyApp.policy` est ajouté aux autres règles effectives. Si vous ajoutez un second signe égal, comme dans

```
java -Djava.security.policy==MyApp.policy MyApp
```

votre application n'utilisera *que* le fichier de règles spécifié et les fichiers standard seront ignorés.

ATTENTION

Il est fréquent, en phase de test, de laisser accidentellement un fichier `.java.policy` qui accorde quantité de permissions, peut-être même `AllPermission`, dans le répertoire courant. Si vous remarquez que votre application ne semble pas tenir compte des restrictions de votre fichier de règles, vérifiez la présence d'un fichier `.java.policy` resté par accident dans votre répertoire courant. Si vous travaillez sous Unix, cette erreur est particulièrement courante, car les fichiers dont le nom commence par un point ne sont pas affichés par défaut.

Comme vous l'avez vu précédemment, les applications Java n'installent pas par défaut de gestionnaire de sécurité. Vous ne verrez donc pas les effets des fichiers de règles tant que vous n'en aurez pas installé un. Vous pouvez, bien sûr, ajouter une ligne

```
System.setSecurityManager(new SecurityManager());
```

544 *Au cœur de Java 2 - Fonctions avancées*

dans votre méthode `main`. Ou ajouter l'option de ligne de commande `-Djava.security.manager` lorsque vous lancez la machine virtuelle.

```
java -Djava.security.manager
    -Djava.security.policy=MyApp.policy MyApp
```

Dans le reste de cette section, vous verrez en détail comment décrire les permissions dans le fichier de règles. Nous allons décrire la totalité du format de fichier de règles, excepté en ce qui concerne les certificats pour le code, que nous verrons plus loin dans ce chapitre.

Un fichier de règles contient une séquence d'entrées `grant`. Chaque entrée a la forme suivante :

```
grant sourcecode
{
    permission1;
    permission2;
    . . .
};
```

sourcecode contient une origine de code (qui peut être omise si l'entrée s'applique au code provenant de toutes sources) et les noms des signataires de certificat et d'éléments fiables (qui peuvent être omis si les signatures ne sont pas requises pour cette entrée).

L'origine du code est spécifiée de la façon suivante :

```
codeBase "url"
```

Si l'URL se termine par un `/`, elle fait référence à un répertoire. Sinon, elle est considérée comme étant le nom d'un fichier JAR. Par exemple :

```
grant codeBase "www.horstmann.com/classes/" { . . . };
grant codeBase "www.horstmann.com/classes/MyApp.jar" { . . . };
```

L'origine du code est une URL et doit toujours contenir des signes slash normaux (`/`) comme séparateurs de fichier, même pour les URL de fichiers Windows. Par exemple :

```
grant codeBase "file:C:/myapps/classes/" { ... };
```

INFO

Tout le monde sait que les URL `http` commencent par deux signes slash (`http://`). C'est moins clair en ce qui concerne les URL de `file` puisque le lecteur de fichier de règles accepte deux formes d'URL de fichier, c'est-à-dire, `file://fichierlocal` et `file:fichierlocal`. De plus, le slash avant une lettre d'unité Windows est facultatif. En résumé, tout ce qui suit est acceptable :

```
file:C:/rép/nomfichier.ext
file:/C:/rép/nomfichier.ext
file://C:/rép/nomfichier.ext
file:///C:/rép/nomfichier.ext
```

En fait, dans nos tests, `file:///C:/rép/nomfichier.ext` était également accepté, mais nous n'avons aucune explication à donner à ce sujet. Dans le système UNIX/Linux, utilisez la forme :

```
file:/rép/nomfichier.ext
```

La structure des permissions est la suivante :

```
permission NomClasse NomCible, ListeActions;
```

NomClasse est le nom totalement qualifié de la classe de permission (par exemple `java.io.FilePermission`). *NomCible* est une valeur spécifique de permission, par exemple un nom de répertoire ou de fichier pour une permission de fichier, ou un hôte et numéro de port pour une permission de socket. *ListeActions* est également spécifique à la permission. C'est une liste d'actions, comme `read` ou `connect`, séparées par des virgules. Certaines classes de permission n'ont pas besoin de noms cibles ni de listes d'actions. Le Tableau 7.2 récapitule les classes de permissions souvent utilisées et leurs actions.

Tableau 7.2 : Les permissions avec leurs cibles et actions associées

<i>Permission</i>	<i>Cible</i>	<i>Action</i>
<code>java.io.FilePermission</code>	cible fichier (voir texte)	<code>read</code> , <code>write</code> , <code>execute</code> , <code>delete</code>
<code>java.net.SocketPermission</code>	cible socket (voir texte)	<code>accept</code> , <code>connect</code> , <code>listen</code> , <code>resolve</code>
<code>java.util.PropertyPermission</code>	cible propriété (voir texte)	<code>read</code> , <code>write</code>
<code>java.lang.RuntimePermission</code>	<code>createClassLoader</code> <code>getClassLoader</code> <code>setContextClassLoader</code> <code>enableContextClassLoaderOverride</code> <code>createSecurityManager</code> <code>setSecurityManager</code> <code>exitVM</code> <code>getenv.<i>nomVariable</i></code> <code>shutdownHooks</code> <code>setFactory</code> <code>setIO</code> <code>modifyThread</code> <code>stopThread</code> <code>modifyThreadGroup</code> <code>getProtectionDomain</code> <code>readFileDescriptor</code> <code>writeFileDescriptor</code> <code>loadLibrary.<i>NomBibliothèque</i></code> <code>accessClassInPackage.<i>NomPackage</i></code> <code>defineClassInPackage.<i>NomPackage</i></code> <code>accessDeclaredMembers.<i>NomClasse</i></code> <code>queuePrintJob</code> <code>getStackTrace</code> <code>setDefaultUncaughtExceptionHandler</code> <code>preferences</code>	

546 Au cœur de Java 2 - Fonctions avancées

Tableau 7.2 : Les permissions avec leurs cibles et actions associées (suite)

<i>Permission</i>	<i>Cible</i>	<i>Action</i>
java.awt.AWTPermission	showWindowWithoutWarningBanner accessClipboard accessEventQueue createRobot fullScreenExclusive listenToAllAWTEvents readDisplayPixels replaceKeyboardFocusManager watchMousePointer setWindowAlwaysOnTop setAppletStub	
java.net.NetPermission	setDefaultAuthenticator specifyStreamHandler requestPasswordAuthentication setProxySelector getProxySelector setCookieHandler getCookieHandler setResponseCache getResponseCache	
java.lang.reflect.Reflect- Permission	suppressAccessChecks	
java.io.Serializable- Permission	enableSubclassImplementation enableSubstitution	
java.security.Security- Permission	createAccessControlContext getDomainCombiner getPolicy setPolicy getProperty.clé setProperty.clé insertProvider.NomFournisseur removeProvider.NomFournisseur setSystemScope setIdentityPublicKey setIdentityInfo addIdentityCertificate removeIdentityCertificate printIdentity clearProviderProperties.Nom- Fournisseur putProviderProperty.NomFournisseur removeProviderProperty.NomFournisseur getSignerPrivateKey setSignerKeyPair	

Tableau 7.2 : Les permissions avec leurs cibles et actions associées (suite)

<i>Permission</i>	<i>Cible</i>	<i>Action</i>
<code>java.security.AllPermission</code>		
<code>javax.audio.AudioPermission</code>	play record	
<code>javax.security.auth.Auth- Permission</code>	doAs doAsPrivileged getSubject getSubjectFromDomainCombiner setReadOnly modifyPrincipals modifyPublicCredentials modifyPrivateCredentials refreshCredential destroyCredential createLoginContext. <i>nomContexte</i> getLoginConfiguration setLoginConfiguration refreshLoginConfiguration	
<code>java.util.logging.Logging- Permission</code>	control	
<code>java.sql.SQLPermission</code>	setLog	

Comme vous le voyez dans le Tableau 7.2, la plupart des permissions autorisent simplement une opération particulière. Vous pouvez imaginer l'opération comme la cible avec une action "permit" implicite. Ces classes de permission étendent toutes la classe `BasicPermission` (reportez-vous à la Figure 7.5 plus haut). Toutefois, les cibles pour les permissions de fichier, de socket et de propriétés, sont plus complexes et doivent être vues plus en détail.

Les cibles de permission de fichier peuvent avoir la forme suivante :

- *fichier* : un fichier ;
- *répertoire/* : un répertoire ;
- *répertoire/** : tous les fichiers du répertoire ;
- *** : tous les fichiers du répertoire courant ;
- *répertoire/-* : tous les fichiers du répertoire ou de l'un de ses sous-répertoires ;
- *-* : tous les fichiers du répertoire courant ou de l'un de ses sous-répertoires ;
- *<<ALL FILES>>* : tous les fichiers du système de fichiers ;

548 *Au cœur de Java 2 - Fonctions avancées*

Par exemple, l'entrée de permission suivante donne accès à tous les fichiers du répertoire /myapp et de chacun de ses sous-répertoires.

```
permission java.io.FilePermission "/myapp/-",
    "read,write,delete";
```

Vous devez utiliser la séquence d'échappement `\\` pour indiquer un slash inverse dans un nom de fichier Windows.

```
permission java.io.FilePermission "c:\\myapp\\-",
    "read,write,delete";
```

Les cibles de permission de socket consistent en un hôte et un numéro de port. Les spécifications d'hôte ont la forme suivante :

- *NomHôte* ou *AdresseIP* : un hôte unique
- `localhost` ou chaîne vide : l'hôte local
- **.SuffixeDomaine* : tout hôte dont le domaine se termine par le suffixe donné
- `*` : tous les hôtes

Les numéros de port sont facultatifs et ont la forme suivante :

- `:n` : un port unique
- `:n-` : tous les ports numérotés *n* et supérieur
- `:-n` : tous les ports numérotés *n* et inférieur
- `:n1-n2` : tous les ports dans la plage indiquée

Voici un exemple :

```
permission java.net.SocketPermission
    "*.horstmann.com:8000-8999", "connect";
```

Enfin, les cibles de permission de propriété peuvent avoir deux formes :

- *propriété* : une propriété spécifique
- *PréfixePropriété.** : toutes les propriétés avec le préfixe indiqué

Par exemple : `"java.home"` et `"java.vm.*"`.

L'entrée de permission suivante autorise un programme à lire toutes les propriétés qui commencent par `java.vm`.

```
permission java.util.PropertyPermission "java.vm.*", "read";
```

Vous pouvez utiliser les propriétés système dans les fichiers de règles. Le terme `${propriété}` est remplacé par la valeur de propriété. Par exemple, `${user.home}` est remplacé par le répertoire personnel (*home*) de l'utilisateur (*user*). Voici un exemple typique d'utilisation de cette propriété système dans une entrée de permission.

```
permission java.io.FilePermission "${user.home}" "read,write";
```

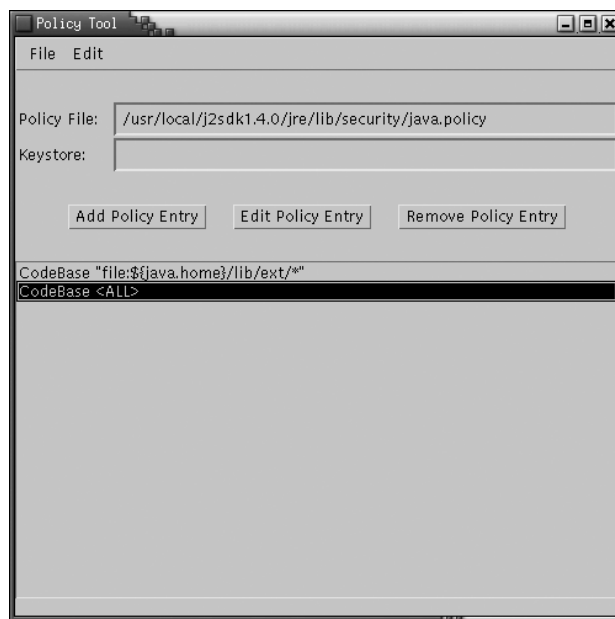
Pour créer des fichiers de règles indépendants de la plate-forme, il est judicieux d'utiliser la propriété `file.separator` au lieu des séparateurs explicites `/` ou `\\`. Plus simplement, la notation `${/}` est un raccourci pour `${file.separator}`. Par exemple,

```
permission java.io.FilePermission "${user.home}${/}-"  
    "read,write";
```

est une entrée portable pour accorder la permission de lecture et d'écriture dans le répertoire personnel de l'utilisateur et n'importe lequel de ses sous-répertoires.

Figure 7.6

L'outil PolicyTool.



INFO

Le JDK est livré avec un outil rudimentaire, appelé `policytool`, que vous pouvez utiliser pour modifier les fichiers de règles (voir Figure 7.6). Bien entendu, cet outil n'est pas destiné à un utilisateur final qui risquerait d'être complètement désorienté par la plupart des valeurs. Il doit être considéré comme un outil d'administration pouvant être utilisé par les administrateurs système qui préfèrent la souris à la syntaxe. Il manque néanmoins un jeu compréhensible de catégories (tel que faible, moyenne ou haute sécurité) qui serait accessible aux non-experts. Plus généralement, nous pensons que la plate-forme Java 2 contient certainement tous les éléments d'un modèle de sécurité très pointu, mais qu'elle pourrait bénéficier de quelques améliorations en ce qui concerne la transmission des informations à l'utilisateur final ou à l'administrateur système.

Les permissions personnalisées

Dans cette section, vous allez voir comment vous pouvez fournir aux utilisateurs votre propre classe de permission, afin qu'ils puissent y faire référence dans leurs fichiers de règles.

550 *Au cœur de Java 2 - Fonctions avancées*

Pour implémenter votre classe de permission, vous étendez la classe `Permission` et fournissez les méthodes suivantes :

un constructeur avec deux paramètres de type `String`, pour la cible et la liste d'actions

```
String getActions()
boolean equals()
int hashCode()
boolean implies(Permission other)
```

La dernière méthode est la plus importante. Les permissions ont un *ordonnement*, dans lequel les permissions les plus générales *en impliquent* d'autres plus spécifiques. Considérez la permission de fichier

```
p1 = new FilePermission("/tmp/-", "read, write");
```

Cette permission autorise la lecture et l'écriture de tout fichier dans le répertoire `/tmp` et l'un quelconque de ses sous-répertoires.

Voici quelques permissions plus spécifiques impliquées par cette permission :

```
p2 = new FilePermission("/tmp/-", "read");
p3 = new FilePermission("/tmp/aFile", "read, write");
p4 = new FilePermission("/tmp/aDirectory/-", "write");
```

En d'autres termes, une permission de fichier `p1` implique une autre permission de fichier `p2` si

1. le jeu de fichier cible de `p1` contient le jeu de fichier cible de `p2` ;
2. le jeu d'actions de `p1` contient le jeu d'actions de `p2`.

Envisagez cet exemple. Lorsque le constructeur de `FileInputStream` veut ouvrir un fichier en lecture, il vérifie s'il possède la permission correspondante. Pour réaliser cette vérification, un objet permission de fichier *spécifique* est passé à la méthode `checkPermission` :

```
checkPermission(new FilePermission(fileName, "read"));
```

Le gestionnaire de sécurité demande ensuite à toutes les permissions applicables si elles impliquent cette permission. Si l'une quelconque d'entre elles l'implique, le contrôle est concluant.

En particulier, `AllPermission` implique toutes les autres permissions.

Si vous définissez vos propres classes de permission, vous devez définir une notion d'implication adéquate pour vos objets `Permission`. Supposons, par exemple, que vous définissiez une permission `TVPermission` pour un programmeur utilisant la technologie Java. Une permission

```
new TVPermission("Tommy:2-12:1900-2200", "watch,record")
```

permettrait à Tommy de regarder et d'enregistrer les chaînes 2 à 12 entre 19:00 et 22:00. Vous devriez implémenter la méthode `implies` afin que cette permission en implique une plus spécifique, telle que

```
new TVPermission("Tommy:4:2000-2100", "watch")
```

Implémenter une classe de permissions

Dans l'exemple de programme suivant, nous implémentons une nouvelle permission pour le contrôle de l'insertion de texte dans une zone de texte. Le programme vérifie que vous ne pouvez pas ajouter de "mots prohibés" comme *sexe*, *drogues* et *C++* dans une zone de texte. Nous utilisons une

classe de permission personnalisée afin que la liste des mots interdits soit fournie dans un fichier de règles.

La sous-classe suivante de `JTextArea` demande au gestionnaire de sécurité s'il est d'accord pour accepter du nouveau texte.

```
class WordCheckTextArea extends JTextArea
{
    public void append(String text)
    {
        WordCheckPermission p
            = new WordCheckPermission(text, "insert");
        SecurityManager manager = System.getSecurityManager();
        if (manager != null) manager.checkPermission(p);
        super.append(text);
    }
}
```

Si le gestionnaire de sécurité accorde la permission `WordCheckPermission`, le texte est ajouté. Dans le cas contraire, la méthode `checkPermission` lance une exception.

Les permissions de vérification de texte ont deux actions possibles : `insert` (la permission d'insérer un texte spécifique) et `avoid` (la permission d'ajouter tout texte ne comprenant pas certains mots interdits). Ce programme sera exécuté avec le fichier de règles suivant :

```
grant
{
    permission WordCheckPermission "sexe,drogues,C++", "avoid";
};
```

Ce fichier accorde la permission d'insérer tout texte ne comprenant pas les mots *sexe*, *drogues* et *C++*.

Lors de la conception de la classe `WordCheckPermission`, nous devons porter une attention particulière à la méthode `implies`. Voici les règles qui contrôlent si la permission `p1` implique la permission `p2`.

1. Si `p1` possède l'action `avoid` et `p2` l'action `insert`, la cible de `p2` doit éviter tous les mots dans `p1`. Par exemple, la permission

```
WordCheckPermission "sexe,drogues,C++", "avoid"
```

implique la permission

```
WordCheckPermission "Mary a un petit chien", "insert"
```

2. Si `p1` et `p2` possèdent tous deux l'action `avoid`, le jeu de mots de `p2` doit contenir tous les mots du jeu de mots de `p1`. Par exemple, la permission

```
WordCheckPermission "sexe,drogues", "avoid"
```

implique la permission

```
WordCheckPermission "sexe,drogues,C++", "avoid"
```

3. Si `p1` et `p2` possèdent tous deux l'action `insert`, le texte de `p1` doit contenir le texte de `p2`. Par exemple, la permission

```
WordCheckPermission "Mary a un petit chien", "insert"
```

552 *Au cœur de Java 2 - Fonctions avancées*

implique la permission

```
WordCheckPermission "un petit chien", "insert"
```

L'implémentation de cette classe se trouve dans l'Exemple 7.5.

Notez que vous extrayez la cible de la permission à l'aide de la méthode abusivement nommée `getName` de la classe `Permission`.

Les permissions étant décrites par une paire de chaînes dans les fichiers de règles, les classes de permission doivent être préparées à analyser ces chaînes. En particulier, nous utilisons la méthode suivante pour transformer la liste des mots interdits séparés par des virgules d'une permission `avoid` en un véritable `Set`.

```
public Set<String> badWordSet()  
{  
    Set<String> set = new HashSet<String>();  
    set.addAll(Arrays.asList(getName().split(", ")));  
    return set;  
}
```

Ce code utilise les méthodes `equals` et `containsAll` pour comparer les jeux. Comme vous l'avez vu dans le Chapitre 2, la méthode `equals` d'une classe `Set` considère deux jeux comme étant égaux s'ils contiennent les mêmes éléments, quel qu'en soit l'ordre. Par exemple, les jeux résultant de "sexe, drogues, C++" et "C++, drogues, sexe" sont égaux.

ATTENTION

Assurez-vous que votre classe de permission est publique. Le chargeur de fichier de règles ne peut pas charger des classes dont la visibilité de package se trouve en dehors du chemin de la classe amorce, et il ignore tout simplement les classes qu'il ne peut trouver.

Le programme de l'Exemple 7.4 montre comment fonctionne la classe `WordCheckPermission`. Tapez un texte dans la zone de texte, puis cliquez sur le bouton "Insérer". Si le contrôle de sécurité est concluant, le texte est ajouté à la zone de texte. Dans le cas contraire, un message d'erreur apparaît (voir Figure 7.7).

Vérifiez que vous lancez bien le programme avec le fichier de règles approprié.

```
java -Djava.security.policy=PermissionTest.policy  
    PermissionTest
```

Sinon, toute tentative d'insertion de texte échouera.

ATTENTION

Si vous examinez attentivement la Figure 7.7, vous verrez que le cadre de la fenêtre porte bizarrement l'intitulé "Java Applet Window". Le libellé de la fenêtre est déterminé par la cible `showWindowWithoutWarningBanner` de `java.awt.AWTPermission`. Si vous le voulez, vous pouvez éditer le fichier de règles pour accorder cette permission.

Figure 7.7
Le programme
PermissionTest.



Exemple 7.4 : *PermissionTest.java*

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.security.*;
import java.util.*;
import javax.swing.*;

/**
 * Cette classe montre la permission personnalisée WordCheckPermission.
 */
public class PermissionTest
{
    public static void main(String[] args)
    {
        System.setSecurityManager(new SecurityManager());
        JFrame f = new PermissionTestFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc contient un champ de texte pour insérer des mots
 * dans une zone de texte protégée des "mots interdits".
 */
class PermissionTestFrame extends JFrame
{
    public PermissionTestFrame()
    {
        setTitle("PermissionTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        textField = new JTextField(20);
        JPanel panel = new JPanel();
        panel.add(textField);
        JButton openButton = new JButton("Insérer");
        panel.add(openButton);
        openButton.addActionListener(new
```


554 Au cœur de Java 2 - Fonctions avancées

```
        ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                insertWords(textField.getText());
            }
        });

        add(panel, BorderLayout.NORTH);

        textArea = new WordCheckTextArea();
        add(new JScrollPane(textArea), BorderLayout.CENTER);
    }

    /**
     * Essaie d'insérer des mots dans la zone de texte.
     * Affiche une boîte de dialogue si la tentative a échoué.
     * @param words les mots à insérer
     */
    public void insertWords(String words)
    {
        try
        {
            textArea.append(words + "\n");
        }
        catch (SecurityException e)
        {
            JOptionPane.showMessageDialog(this,
                "Désolé, c'est interdit.");
        }
    }

    private JTextField textField;
    private WordCheckTextArea textArea;
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 300;
}

/**
 * Une zone de texte dont la méthode append crée une
 * vérification de sécurité afin qu'aucun mot interdit ne
 * soit ajouté.
 */
class WordCheckTextArea extends JTextArea
{
    public void append(String text)
    {
        WordCheckPermission p
            = new WordCheckPermission(text, "insert");
        SecurityManager manager = System.getSecurityManager();
        if (manager != null) manager.checkPermission(p);
        super.append(text);
    }
}
```

Exemple 7.5 : WordCheckPermission.java

```
import java.security.*;
import java.util.*;

/**
 * Une permission qui recherche les mots interdits.
 */
public class WordCheckPermission extends Permission
{
    /**
     * Construit une permission de vérification des mots.
     * @param target une liste de mots séparés par des virgules
     * @param anAction "insérer" ou "éviter"
     */
    public WordCheckPermission(String target, String anAction)
    {
        super(target);
        action = anAction;
    }

    public String getActions() { return action; }

    public boolean equals(Object other)
    {
        if (other == null) return false;
        if (!getClass().equals(other.getClass())) return false;
        WordCheckPermission b = (WordCheckPermission) other;
        if (!action.equals(b.action)) return false;
        if (action.equals("insérer"))
            return getName().equals(b.getName());
        else if (action.equals("éviter"))
            return badWordSet().equals(b.badWordSet());
        else return false;
    }

    public int hashCode()
    {
        return getName().hashCode() + action.hashCode();
    }

    public boolean implies(Permission other)
    {
        if (!(other instanceof WordCheckPermission)) return false;
        WordCheckPermission b = (WordCheckPermission) other;
        if (action.equals("insert"))
        {
            return b.action.equals("insérer") &&
                getName().indexOf(b.getName()) >= 0;
        }
        else if (action.equals("éviter"))
        {
            if (b.action.equals("éviter"))
                return b.badWordSet().containsAll(badWordSet());
            else if (b.action.equals("insérer"))
            {
                for (String badWord : badWordSet())
                    if (b.getName().indexOf(badWord) >= 0)
                        return false;
            }
        }
    }
}
```

556 Au cœur de Java 2 - Fonctions avancées

```

        return true;
    }
    else return false;
}
else return false;
}

/**
 * Obtient les mots interdits décrits par cette règle de permission.
 * @return un ensemble de mots interdits
 */
public Set<String> badWordSet()
{
    Set<String> set = new HashSet<String>();
    set.addAll(Arrays.asList(getName().split(", ")));
    return set;
}

private String action;
}

```

java.security.Permission 1.2

- `Permission(String name)`
Construit une permission avec le nom cible donné.
- `String getName()`
Renvoie le nom cible de cette permission.
- `boolean implies(Permission other)`
Vérifie si cette permission implique l'autre permission. C'est le cas si l'autre permission décrit une condition plus spécifique qui est une conséquence de la condition décrite par cette permission.

Un gestionnaire de sécurité personnalisé

Dans cette section, nous vous apprenons à construire un gestionnaire de sécurité simple. Dans vos programmes, il vaut mieux utiliser des permissions personnalisées qu'implémenter un gestionnaire de sécurité complet, mais nous avons pensé que vous seriez intéressé par leur fonctionnement. Celui-ci surveille tous les accès fichier et s'assure que vous ne pourrez pas ouvrir un fichier texte s'il contient des mots interdits comme *sexe*, *drogues*, et *C++*.

La surveillance de l'accès fichier est faite par surcharge de la méthode `checkPermission` de la classe du gestionnaire de sécurité standard. Si la permission n'est pas *file read permission* (permission de lecture de fichier), nous appelons simplement `super.checkPermission`. Pour vérifier qu'un fichier peut être lu, il est ouvert et son contenu est scanné. L'accès n'est accordé au fichier que si celui-ci ne contient aucun des mots interdits. (Seuls les fichiers dont l'extension est `.txt` sont surveillés, puisque nous ne voulons pas bloquer l'accès aux fichiers système et de propriétés).

```

public class WordCheckSecurityManager extends SecurityManager
{
    public void checkPermission(Permission p)
    {
        if (p instanceof FilePermission
            && p.getActions().equals("read"))

```

```

        {
            String fileName = p.getName();
            if (containsBadWords(fileName))
                throw new SecurityException("Mots interdits dans "
                    + fileName);
            }
        else super.checkPermission(p);
    }
    . . .
}

```

INFO

Une autre façon d'être averti d'une demande de lecture de fichier consiste à surcharger la méthode `checkRead`. La classe `SecurityManager` implémente cette méthode pour appeler la méthode `checkPermission` avec un objet `FilePermission`. Il existe près d'une trentaine de méthodes similaires dans `SecurityManager`, pour des raisons historiques. Le système de permission a été introduit pour la plate-forme Java 2. Nous vous recommandons de ne pas surcharger ces méthodes, mais plutôt d'exécuter tous les contrôles de permission dans la méthode `checkPermission`.

Il y a juste un piège dans notre scénario de contrôle de fichier. Examinez le flux possible d'événements suivant.

- Une méthode d'une classe quelconque ouvre un fichier.
- Puis le gestionnaire de sécurité passe à l'action et utilise sa méthode `checkPermission`.
- La méthode `checkPermission` appelle la méthode `containsBadWords`.

Mais la méthode `containsBadWords` doit elle-même lire le fichier pour vérifier son contenu, ce qui appelle à nouveau le gestionnaire de sécurité ! Cela entraînerait une régression infinie à moins que le gestionnaire de sécurité n'ait un moyen de déterminer dans quel contexte il a été appelé. La méthode `getClassContext` permet de trouver comment la méthode a été appelée. Cette méthode renvoie un tableau d'objets `Class` qui indique toutes les classes dont les appels sont actuellement en cours. Par exemple, lorsque le gestionnaire de sécurité est appelé pour la première fois, le tableau contient

```

class WordCheckSecurityManager
class SecurityManager
class java.io.FileInputStream
class java.io.FileReader
class SecurityManagerFrame
. . .
class java.awt.EventDispatchThread

```

La classe d'index 0 donne l'appel actuellement en cours d'exécution. Malheureusement, vous n'obtenez finalement que les classes, pas les noms des méthodes en cours. Lorsque le gestionnaire de sécurité tente lui-même d'ouvrir le fichier, il est à nouveau appelé et la méthode `getClassContext` renvoie le tableau suivant :

```

class WordCheckSecurityManager
class SecurityManager
class java.io.FileInputStream
class java.io.FileReader
class WordCheckSecurityManager
class WordCheckSecurityManager

```

558 Au cœur de Java 2 - Fonctions avancées

```

class SecurityManager
class java.io.FileInputStream
class java.io.FileReader
class SecurityManagerFrame
. . .
class java.awt.EventDispatchThread

```

Dans ce cas, le gestionnaire de sécurité devrait permettre l'accès au fichier. Comment pouvons-nous faire ? Nous pourrions tester

```
getClassContext()[0] == getClassContext()[4]
```

mais cette approche est délicate. Voici un cas évident où cela peut poser des problèmes : envisagez le cas où l'implémentation changerait, par exemple, pour que le constructeur de `FileReader` appelle directement le gestionnaire de sécurité ; le test serait alors sans signification, car les positions ne seraient plus les mêmes dans le tableau. Il est de loin préférable de tester si, parmi les appels en cours, il *en* existe provenant du même gestionnaire de sécurité.

Voici une méthode qui réalise ce test. Puisque cette méthode est appelée à partir de `checkPermission`, il y a au moins deux copies de la classe de gestionnaire de sécurité dans la pile d'appel. Nous les passons donc, puis nous recherchons une autre instance du même gestionnaire de sécurité.

```

boolean inSameManager()
{
    Class[] cc = getClassContext();

    // sauter le jeu d'appels courant à ce gestionnaire
    int i = 0;
    while (i < cc.length && cc[0] == cc[i])
        i++;

    // vérifier s'il y a un autre appel à ce même gestionnaire
    while (i < cc.length)
    {
        if (cc[0] == cc[i]) return true;
        i++;
    }
    return false;
}

```

Nous appelons cette méthode dans la méthode `checkPermission`. Si nous constatons que le gestionnaire de sécurité est invoqué de manière réursive, nous n'appelons pas à nouveau la méthode `containsBadWords`.

```

if (p instanceof FilePermission
    && p.getActions().equals("read"))
{
    if (inSameManager())
        return;
    String fileName = p.getName();
    if (containsBadWords(fileName))
        throw new SecurityException("Mots interdits dans "
            + fileName);
}

```

L'Exemple 7.6 montre un programme qui met en action ce gestionnaire de sécurité. Le gestionnaire de sécurité est installé dans la méthode `main`. Lorsque vous exécutez le programme, vous pouvez spécifier un fichier. Le programme chargera son contenu dans la zone de texte. Toutefois, si le fichier

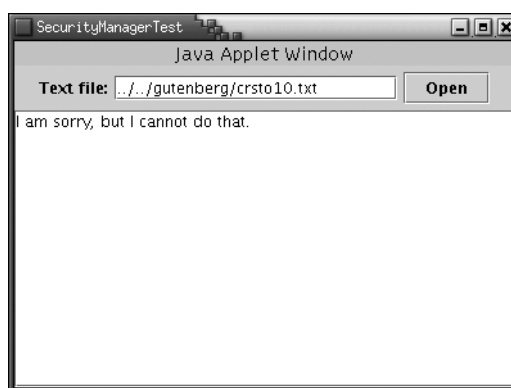
ne passe pas le test de sécurité, le programme va intercepter l'exception de sécurité et afficher un message à la place (voir Figure 7.8). Par exemple, vous pouvez afficher "Alice au pays des merveilles", mais le programme refusera de charger "Le Comte de Monte-Cristo".

INFO

Vous vous demandez peut-être pourquoi nous utilisons un champ de texte au lieu du `JFileChooser` pour sélectionner un nom de fichier. La classe `JFileChooser` appelle la méthode `isHidden` de la classe `File` pour chaque fichier du répertoire courant et essaie d'obtenir une autorisation de lecture auprès du gestionnaire de sécurité.

Figure 7.8

Le programme
`SecurityManagerTest`.



Vous devez faire attention à la façon dont vous invoquez ce programme. La classe `WordCheckSecurityManager` elle-même doit se voir attribuer `AllPermission`, et ce pour une raison subtile. La classe `WordCheckSecurityManager` appelle la superclasse `SecurityManager` pour toutes les permissions autres que les permissions de lecture de fichier. Lorsque la classe `SecurityManager` évalue une permission, elle regarde si *toutes les méthodes dans la pile d'appel* doivent se voir attribuer cette permission particulière. La classe `WordCheckSecurityManager` est l'une de ces classes. Mais il ne s'agit pas d'une classe système, vous devez donc explicitement lui accorder toutes les permissions, sans pour cela accorder toutes les permissions aux autres classes du programme.

Pour séparer les fichiers de la classe `WordCheckSecurityManager` des autres fichiers de classe, créez un fichier JAR contenant seulement ce fichier de classe.

```
jar cvf WordCheck.jar WordCheckSecurityManager.class
```

Puis supprimez le fichier `WordCheckSecurityManager.class`.

Créez ensuite un fichier de règles, `WordCheck.policy`, contenant

```
grant codeBase "file:WordCheck.jar"
{
    permission java.security.AllPermission;
};
```

Cette règle accorde toutes les permissions aux classes dans le fichier `WordCheck.jar`. Lancez enfin l'application de la façon suivante :

```
java -classpath .:WordCheck.jar -Djava.security.policy=WordCheck.policy
    SecurityManagerTest
```

560 *Au cœur de Java 2 - Fonctions avancées*

Sous Windows, utilisez un point-virgule pour séparer les éléments de chemin de la classe :

```
java -classpath .;WordCheck.jar ...
```

ASTUCE

Si vous avez l'idée de changer le gestionnaire de sécurité dans vos propres programmes, vous devez tout d'abord voir si vous ne pouvez pas, à la place, utiliser le gestionnaire de sécurité standard et une permission personnalisée, comme décrit dans la section précédente. L'écriture d'un gestionnaire de sécurité est une source potentielle d'erreur et peut avoir pour conséquence de subtiles failles au niveau de la sécurité. Il est de loin préférable d'utiliser le gestionnaire de sécurité standard et d'augmenter plutôt le système de permission.

Exemple 7.6 : *SecurityManagerTest.java*

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;

/**
 * Cette classe montre l'utilisation d'un gestionnaire de sécurité
 * personnalisé qui interdit la lecture de fichiers de texte contenant
 * les mots interdits.
 */
public class SecurityManagerTest
{
    public static void main(String[] args)
    {
        System.setSecurityManager(new WordCheckSecurityManager());
        JFrame frame = new SecurityManagerFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc contient un champ de texte permettant d'entrer le nom d'un
 * fichier et une zone de texte pour montrer le contenu du fichier
 * chargé.
 */
class SecurityManagerFrame extends JFrame
{
    public SecurityManagerFrame()
    {
        setTitle("SecurityManagerTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        fileNameField = new JTextField(20);
        JPanel panel = new JPanel();
        panel.add(new JLabel("Fichier texte :"));
        panel.add(fileNameField);
        JButton openButton = new JButton("Ouvrir");
        panel.add(openButton);
        openButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    // ...
                }
            }
        );
    }
}
```

```

        {
            public void actionPerformed(ActionEvent event)
            {
                loadFile(fileNameField.getText());
            }
        });

        add(panel, "North");

        fileText = new JTextArea();
        add(new JScrollPane(fileText), "Center");
    }

    /**
     * Tente de charger un fichier dans la zone de texte. En cas de
     * déclenchement d'une exception de sécurité, un message est inséré
     * dans la zone de texte.
     * @param filename le nom de fichier
     */
    public void loadFile(String filename)
    {
        try
        {
            fileText.setText("");
            Scanner in = new Scanner(new FileReader(filename));
            while (in.hasNextLine()) fileText.append(in.nextLine() + "\n");
            fileText.append(s + "\n");
            in.close();
        }
        catch (IOException e)
        {
            fileText.append(e + "\n");
        }
        catch (SecurityException e)
        {
            fileText.append("Désolé, c'est interdit.\n");
            fileText.append(e + "\n");
        }
    }

    private JTextField fileNameField;
    private JTextArea fileText;
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 300;
}

```

Exemple 7.7 : WordCheckSecurityManager.java

```

import java.io.*;
import java.security.*;
import java.util.*;

/**
 * Ce gestionnaire de sécurité vérifie si les mots interdits
 * sont rencontrés à la lecture d'un fichier.
 */
public class WordCheckSecurityManager extends SecurityManager
{
    public void checkPermission(Permission p)

```


562 Au cœur de Java 2 - Fonctions avancées

```
{
    if (p instanceof FilePermission
        && p.getActions().equals("read"))
    {
        if (inSameManager())
            return;
        String fileName = p.getName();
        if (containsBadWords(fileName))
            throw new SecurityException("Mots interdits dans "
                + fileName);
    }
    else super.checkPermission(p);
}

/**
 * Renvoie true si ce gestionnaire est appelé alors qu'un autre appel
 * est en attente vers lui-même.
 * @return true s'il y a plusieurs appels à ce gestionnaire
 */
public boolean inSameManager()
{
    Class[] cc = getClassContext();

    // sauter le jeu courant d'appel à ce gestionnaire
    int i = 0;
    while (i < cc.length && cc[0] == cc[i])
        i++;

    // vérifier si d'autres appels à ce gestionnaire
    while (i < cc.length)
    {
        if (cc[0] == cc[i]) return true;
        i++;
    }
    return false;
}

/**
 * Vérifie si un fichier contient des mots interdits.
 * @param fileName le nom du fichier
 * @return true si le nom de fichier se termine par .txt et
 * qu'il contient au moins un mot interdit.
 */
boolean containsBadWords(String fileName)
{
    if (!fileName.toLowerCase().endsWith(".txt")) return false;
    // ne vérifier que les fichiers texte
    Scanner in;
    try
    {
        in = new Scanner(new FileReader(fileName));
    }
    catch (IOException e)
    {
        return false;
    }
}
```

```

        while (in.hasNext())
            if (badWords.contains(in.next().toLowerCase()))
            {
                in.close();
                System.out.println(fileName);
                return true;
            }
        return false;
    }

    private List badWords = Arrays.asList(new String[] {
        "sexe", "drogues", "c++" });
}

```

java.lang.System 1.0

- `void setSecurityManager(SecurityManager s)`
Définit le gestionnaire de sécurité pour le reste de cette application. Si `s` vaut `null`, aucune action n'est entreprise. Cette méthode lance une exception de sécurité si le gestionnaire de sécurité courant n'autorise pas l'installation d'un nouveau gestionnaire de sécurité.
- `SecurityManager getSecurityManager()`
Récupère le gestionnaire de sécurité système ; renvoie `null` si aucun n'est installé.

java.lang.SecurityManager 1.0

- `Class[] getClassContext() 1.1`
Renvoie un tableau des classes pour les méthodes actuellement en cours d'exécution. L'élément à la position 0 est la classe de la méthode actuellement en cours d'exécution, l'élément à la position 1 est la classe de l'appelant de la méthode courante, ainsi de suite. Seuls les noms de classes, et non les noms de méthodes, sont fournis.

Authentifier les utilisateurs

Depuis le JDK 1.4, le JAAS (*Java Authentication and Authorization Service*) est inclus dans J2SE. La partie "authentification" permet de s'assurer de l'identité de l'utilisateur du programme. La partie "autorisation" associe un utilisateur et une permission.

JAAS est une API "enfichable" qui isole les applications Java de la technologie utilisée pour implémenter l'authentification. Elle accepte, entre autres, les logins UNIX, les logins NT, l'authentification Kerberos et l'authentification fondée sur les certificats.

Lorsqu'un utilisateur est authentifié, il est possible de lui attacher un ensemble de permissions. Par exemple, ici, Harry possède un jeu de permissions particulières que n'auront pas d'autres utilisateurs. La syntaxe est la suivante :

```

grant principal com.sun.security.auth.UnixPrincipal "Harry"
{
    permission java.util.PropertyPermission "user.*", "read";
    ...
};

```

La classe `com.sun.security.auth.UnixPrincipal` vérifie le nom de l'utilisateur UNIX qui exécute ce programme. Sa méthode `getName` renvoie le login UNIX et nous vérifions si son nom est égal à "harry".

564 Au cœur de Java 2 - Fonctions avancées

Vous utilisez un `LoginContext` pour autoriser le gestionnaire de sécurité à vérifier cette déclaration. Voici la base du code de connexion :

```
try
{
    System.setSecurityManager(new SecurityManager());
    LoginContext context = new LoginContext("Login1");
    // défini dans le fichier de configuration JAAS
    context.login();
    // récupère le sujet authentifié
    Subject subject = context.getSubject();
    . . .
    context.logout();
}
catch (LoginException exception)
// déclenché si le login n'a pas réussi
{
    exception.printStackTrace();
}
```

Désormais, le `subject` dénote la personne qui a été authentifiée.

Le paramètre de chaîne "Login1" du constructeur `LoginContext` fait référence à une entrée avec le même nom du fichier de configuration JAAS. Voici un exemple de fichier de configuration :

```
Login1
{
    com.sun.security.auth.module.UnixLoginModule required;
    com.whizzbang.auth.module.RetinaScanModule sufficient;
};

Login2
{
    ...
}
```

Bien entendu, il n'existe aucun module de connexion biométrique dans le JDK. Le JDK contient les modules suivants dans le package `com.sun.security.auth.module` :

```
UnixLoginModule
NTLoginModule
Krb5LoginModule
JndiLoginModule
KeyStoreLoginModule
```

Une règle d'identification est constituée d'une suite de modules d'identification étiquetés *required*, *sufficient*, *requisite* ou *optional*. Les modules sont exécutés tour à tour, jusqu'à ce qu'un module *sufficient* réussisse, qu'un module *requisite* échoue ou que la fin de la liste des modules soit atteinte. L'authentification réussit si tous les modules *required* et *requisite* réussissent ou, lorsque aucun d'eux n'a été exécuté, si au moins un module *sufficient* ou *optional* réussit.

Un identifiant authentifie un *subject* (sujet), lui-même pouvant avoir plusieurs *principals* (éléments principaux). Un *principal* décrit certaines propriétés du sujet, comme le nom d'utilisateur, l'ID de groupe ou le rôle. Comme vous l'avez vu dans l'instruction `grant`, les *principals* régissent les permissions. La classe `com.sun.security.auth.UnixPrincipal` décrit le nom de l'identifiant UNIX et la classe `UnixNumericGroupPrincipal` peut être utilisée pour tester l'adhésion à un groupe UNIX.

Une clause `grant` peut tester un *principal*, avec la syntaxe

```
grant classePrincipale "nomPrincipal"
```

Par exemple :

```
grant com.sun.security.auth.UnixPrincipal "harry"
```

Lorsqu'un utilisateur se connecte, vous devez exécuter le code qui exige la vérification des *principals* dans un contexte de contrôle d'accès séparé. Utilisez la méthode statique `doAs` ou `doAsPrivileged` pour lancer une nouvelle `PrivilegedAction` dont la méthode `run` exécute le code.

Ces deux méthodes exécutent une action en appelant la méthode `run` d'un objet qui implémente l'interface `PrivilegedAction`, à l'aide des permissions des *principals* de l'objet :

```
PrivilegedAction action = new
    PrivilegedAction()
{
    public Object run()
    {
        // exécuter avec les permissions des principals des
        // subject
        . . .
    }
};
Subject.doAsPrivileged(subject, action, null);
// ou doAs(subject, action)
```

Si les actions peuvent déclencher des exceptions vérifiées, vous implémentez plutôt l'interface `PrivilegedExceptionAction`.

La différence qui existe entre les méthodes `doAs` et `doAsPrivileged` est assez subtile. La méthode `doAs` démarre par le contexte actuel du contrôle d'accès, tandis que la méthode `doAsPrivileged` démarre avec un nouveau contexte. Cette dernière vous permet de séparer les permissions pour le code d'identification et la "logique d'activité". Dans notre exemple, le code d'identification possède les permissions

```
permission javax.security.auth.AuthPermission
    "createLoginContext.Login1";
permission javax.security.auth.AuthPermission "doAsPrivileged";
```

L'utilisateur authentifié possède la permission

```
permission java.util.PropertyPermission "user.*", "read";
```

Si nous avons utilisé `doAs` au lieu de `doAsPrivileged`, le code d'identification aurait aussi nécessité cette permission !

Les programmes de l'Exemple 7.8 et de l'Exemple 7.9 montrent comment restreindre les permissions de certains utilisateurs. Le programme `AuthTest` authentifie un utilisateur, puis exécute une action simple qui récupère une propriété de système.

Pour que cet exemple fonctionne, vous devez mettre le code de l'identification et d'action en package dans deux fichiers JAR séparés :

```
javac AuthTest.java
jar cvf login.jar AuthTest*.class
javac SysPropAction.java
jar cvf action.jar SysPropAction.class
```

566 Au cœur de Java 2 - Fonctions avancées

Si vous regardez le fichier de règles de sécurité de l'Exemple 7.10, vous verrez que l'utilisateur UNIX prénommé Harry est autorisé à lire tous les fichiers. Exécutez ensuite la commande

```
java -classpath login.jar:action.jar
-Djava.security.policy=AuthTest.policy
-Djava.security.auth.login.config=jaas.config
AuthTest
```

L'Exemple 7.11 montre la configuration de l'identification.

Sous Windows, changez Unix par NT dans `AuthTest.policy` et `jaas.config`, puis utilisez un point-virgule pour séparer les fichiers JAR :

```
java -classpath login.jar:action.jar
-Djava.security.policy=AuthTest.policy
-Djava.security.auth.login.config=jaas.config
AuthTest
```

Le programme `AuthTest` devrait maintenant afficher la valeur de la propriété `user.home`. Toutefois, si vous changez le nom de connexion dans le fichier `AuthTest.policy`, une exception de sécurité se déclenche, car vous ne bénéficierez plus de la permission exigée.

ATTENTION

Prenez garde de suivre *exactement* ces instructions. Il est très facile de se tromper de configuration en effectuant des modifications inopinées.

Exemple 7.8 : *AuthTest.java*

```
import java.security.*;
import javax.security.auth.*;
import javax.security.auth.login.*;

/**
 * Ce programme authentifie un utilisateur via un identifiant
 * personnalisé et exécute SysPropAction
 * avec les privilèges de l'utilisateur.
 */
public class AuthTest
{
    public static void main(final String[] args)
    {
        try
        {
            System.setSecurityManager(new SecurityManager());

            LoginContext context = new LoginContext("Login1");
            context.login();
            System.out.println("Authentification réussie.");
            Subject subject = context.getSubject();
            System.out.println("subject=" + subject);
            PrivilegedAction action = new SysPropAction("user.home");
            Object result = Subject.doAsPrivileged(subject, action, null);
            System.out.println(result);
            context.logout();
        }
        catch (LoginException e)
        {
        }
    }
}
```

```
        e.printStackTrace();
    }
}
```

Exemple 7.9 : SysPropAction.java

```
import java.security.*;

/**
 * Cette action recherche une propriété système.
 */
public class SysPropAction implements PrivilegeAction
{
    /**
     * Construit une action pour rechercher une propriété donnée.
     * @param propertyName Le nom de la propriété (comme "user.home")
     */
    public SysPropAction(String propertyName)
    { this.propertyName = propertyName; }

    public Object run()
    {
        return System.getProperty(propertyName);
    }

    private String propertyName;
}
```

Exemple 7.10 : AuthTest.policy

```
grant codebase "file:login.jar"
{
    permission javax.security.auth.AuthPermission
        "createLoginContext.Login1";
    permission javax.security.auth.AuthPermission "doAsPrivileged";
};

grant principal com.sun.security.auth.UnixPrincipal "harry"
{
    permission java.util.PropertyPermission "user.*", "read";
};
```

javax.security.auth.login.LoginContext 1.4

- LoginContext(String name)
Construit un contexte d'identification. name correspond au descripteur d'identifiant du fichier de configuration JAAS.
- void login()
Etablit un identifiant ou déclenche une LoginException si la connexion a échoué. Invoque la méthode login sur les gestionnaires dans le fichier de configuration JAAS.
- void logout()
Déconnecte le sujet. Invoque la méthode logout sur les gestionnaires du fichier de configuration JAAS.
- Subject getSubject()
Renvoie le sujet authentifié.

568 Au cœur de Java 2 - Fonctions avancées

javax.security.auth.Subject 1.4

- `Set<Principal> getPrincipals()`
Obtient les *principals* du sujet.
- `static Object doAs(Subject subject, PrivilegedAction action)`
- `static Object doAs(Subject subject, PrivilegedExceptionAction action)`
- `static Object doAsPrivileged(Subject subject, PrivilegedAction action, AccessControlContext context)`
- `static Object doAsPrivileged(Subject subject, PrivilegedExceptionAction action, AccessControlContext context)`
Exécutent l'action privilégiée au nom du sujet. Renvoient la valeur de retour de la méthode `run`. Les méthodes `doAsPrivileged` exécutent l'action dans le contexte du contrôle d'accès donné. Vous pouvez fournir un "instantané du contexte" que vous aurez préalablement obtenu en appelant la méthode statique `AccessController.getContext()` ou fournir `null` pour exécuter le code dans un nouveau contexte.

java.security.PrivilegedAction 1.4

- `Object run()`
Vous devez définir cette méthode pour exécuter le code exigé au nom d'un sujet.

java.security.PrivilegedExceptionAction 1.4

- `Object run()`
Vous devez définir cette méthode pour exécuter le code que vous souhaitez voir exécuter au nom d'un sujet. Cette méthode peut déclencher n'importe quelle exception vérifiée.

java.security.Principal 1.1

- `String getName()`
Renvoie le nom d'identifiant de ce *principal*.

Modules d'identification JAAS

Dans cette section, nous étudions l'exemple JAAS, qui vous montre :

- comment implémenter votre propre module d'identification ;
- comment implémenter l'authentification basée sur les rôles.

Il est utile de fournir votre propre module d'identification si vous stockez les informations d'identification dans une base de données. Même si le module par défaut vous convient, étudier un module personnalisé vous aidera à comprendre les options du fichier de configuration JAAS.

L'authentification fondée sur les rôles est essentielle si vous gérez un grand nombre d'utilisateurs. Il ne serait pas pratique de placer les noms de tous les utilisateurs légitimes dans un fichier de règles. Au lieu de cela, le modèle d'identification doit assigner les utilisateurs aux rôles, comme "admin" ou "HR", et les permissions doivent être basées sur ces rôles.

L'une des tâches du module d'identification est de remplir le jeu de *principals* du *subject* qui fait l'objet de l'authentification. Si un module d'identification prend en charge les rôles, il ajoute les objets `Principal` qui décrivent les rôles. Le JDK ne fournit pas de classe dans ce but, nous avons

donc écrit la nôtre (voir Exemple 7.12). La classe stocke simplement une paire description/valeur, comme `role=admin`. Sa méthode `getName` renvoie cette paire, nous pouvons donc ajouter des permissions basées sur les rôles dans un fichier de règles :

```
grant principal SimplePrincipal "role=admin" { . . . }
```

Notre module d'identification (voir Exemple 7.13) recherche les utilisateurs, les mots de passe et les rôles dans un fichier texte qui contient des lignes comme celles-ci :

```
harry|wombat|admin  
carl|mockturtle|HR
```

Bien entendu, dans un module d'identification réaliste, vous stockeriez cette information dans une base de données ou un répertoire.

Vous trouverez le code de `SimpleLoginModule` dans l'Exemple 7.13. La méthode `checkLogin` vérifie si le nom d'utilisateur et le mot de passe correspondent à un dossier utilisateur dans le fichier des mots de passe. Si c'est le cas, nous ajoutons deux objets `SimplePrincipal` au jeu de *principals* du *subject* :

```
Set<Principal> principals = subject.getPrincipals();  
principals.add(new SimplePrincipal("username", username));  
principals.add(new SimplePrincipal("role", role));
```

Le reste de `SimpleLoginModule` n'est que de la plomberie ! La méthode `initialize` reçoit :

- le `Subject` qui est authentifié ;
- un gestionnaire pour récupérer les informations d'identification ;
- une carte `sharedState` pouvant être utilisée pour la communication entre les modules d'identification ;
- une carte `options` qui contient des paires nom/valeur définies dans la configuration d'identification.

Nous pouvons, par exemple, configurer notre module comme suit :

```
SimpleLoginModule required pwfile="password.txt";
```

Le module d'identification récupère les paramètres `pwfile` de la carte `options`.

Le module d'identification ne collecte pas le nom d'utilisateur et le mot de passe ; c'est le travail d'un autre gestionnaire. Cette séparation vous permet d'utiliser le même module d'identification, sans avoir à vous inquiéter de la provenance des informations (boîte de dialogue de GUI, invite de console prompt ou fichier de configuration).

Le gestionnaire est spécifié lorsque vous construisez le `LoginContext`, par exemple :

```
LoginContext context = new LoginContext("Login1",  
    new com.sun.security.auth.callback.DialogCallbackHandler());
```

`DialogCallbackHandler` affiche une boîte de dialogue simple de GUI destinée à récupérer le nom d'utilisateur et le mot de passe. `com.sun.security.auth.callback.TextCallbackHandler` récupère les informations auprès de la console.

Toutefois, dans notre application, nous disposons de notre propre GUI pour collecter le nom de l'utilisateur et le mot de passe (voir Figure 7.9). Nous produisons un gestionnaire simple qui se contente de stocker et de renvoyer ces informations (voir Exemple 7.14).

570 *Au cœur de Java 2 - Fonctions avancées*

Le gestionnaire possède une seule méthode, `handle`, qui traite un tableau d'objets `Callback`. Plusieurs classes prédéfinies, comme `NameCallback` et `PasswordCallback`, implémentent l'interface `Callback`. Vous pourriez aussi ajouter vos propres classes, comme `RetinaScanCallback`. Le code du gestionnaire est un peu disgracieux car il doit analyser les types des objets de rappel :

```
public void handle(Callback[] callbacks)
{
    for (Callback callback : callbacks)
    {
        if (callback instanceof NameCallback) . . .
        else if (callback instanceof PasswordCallback) . . .
        else . . .
    }
}
```

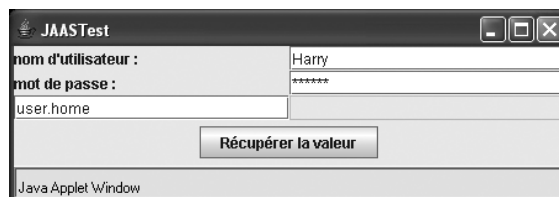
Le module d'identification prépare un tableau des rappels dont il a besoin pour l'authentification :

```
NameCallback nameCall = new NameCallback("nom d'utilisateur : ");
PasswordCallback passCall = new PasswordCallback(
    "mot de passe : ", false);
callbackHandler.handle(new Callback[] { nameCall, passCall });
```

Il récupère ensuite les informations des rappels.

Figure 7.9

Un module d'identification personnalisé.



Le programme de l'Exemple 7.11 affiche un formulaire permettant de saisir les informations d'identification et le nom d'une propriété système. Si l'utilisateur est authentifié, la valeur de la propriété est récupérée dans un `PrivilegedAction`. Comme vous le voyez dans le fichier de règles de l'Exemple 7.15, seuls les utilisateurs ayant des droits administratifs ont une autorisation de lecture.

Comme dans la section précédente, vous devez séparer l'identifiant et l'action. Créez deux fichiers JAR :

```
javac *.java
jar cvf login.jar JAAS*.class Simple*.class
jar cvf action.jar SysPropAction.class
```

Exécutez ensuite le programme sous la forme

```
java -classpath login.jar:action.jar
    -Djava.security.policy=JAASTest.policy
    -Djava.security.auth.login.config=jaas.config
    JAASTest
```

L'Exemple 7.16 montre la configuration de l'identification.

Il est possible de prendre en charge un protocole plus complexe en deux phases, permettant *d'engager* un identifiant si tous les modules de la configuration d'identification ont réussi. Pour en savoir plus, voir le guide du développeur du module à l'adresse <http://java.sun.com/j2se/5.0/docs/guide/security/jaas/JAASLMDevGuide.html>.

Exemple 7.11 : JAASTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.security.auth.*;
import javax.security.auth.login.*;
import javax.swing.*;

/**
 * Ce programme authentifie un utilisateur via un identifiant
 * personnalisé, puis exécute SysPropAction
 * avec les droits de l'utilisateur.
 */
public class JAASTest
{
    public static void main(final String[] args)
    {
        System.setSecurityManager(new SecurityManager());
        JFrame frame = new JAASFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce cadre possède des champs de texte réservés au nom d'utilisateur et
 * au mot de passe ainsi qu'un champ pour le nom de la propriété système
 * demandée, enfin un champ pour montrer la valeur de la propriété.
 */
class JAASFrame extends JFrame
{
    public JAASFrame()
    {
        setTitle("JAASTest");

        username = new JTextField(20);
        password = new JPasswordField(20);
        propertyName = new JTextField(20);
        propertyValue = new JTextField(20);
        propertyValue.setEditable(false);

        JPanel panel = new JPanel();
        panel.setLayout(new GridLayout(0, 2));
        panel.add(new JLabel("nom d'utilisateur :"));
        panel.add(username);
        panel.add(new JLabel("mot de passe :"));
        panel.add(password);
        panel.add(propertyName);
        panel.add(propertyValue);
        add(panel, BorderLayout.CENTER);

        JButton getValueButton = new JButton("Récupérer la valeur");
        getValueButton.addActionListener(new
```

572 Au cœur de Java 2 - Fonctions avancées

```

        ActionListener()
        {
            public void actionPerformed(ActionEvent event) {
                getValue(); }
        });
        JPanel buttonPanel = new JPanel();
        buttonPanel.add(getValueButton);
        add(buttonPanel, BorderLayout.SOUTH);
        pack();
    }

    public void getValue()
    {
        try
        {
            LoginContext context = new LoginContext("Login1",
                new SimpleCallbackHandler(username.getText(),
                    password.getPassword()));
            context.login();
            Subject subject = context.getSubject();
            propertyValue.setText(
                "" + Subject.doAsPrivileged(subject,
                    new SysPropAction(propertyName.getText(), null));
            context.logout();
        }
        catch (LoginException e )
        {
            JOptionPane.showMessageDialog(this, e);
        }
    }

    private JTextField username;
    private JPasswordField password;
    private JTextField propertyName;
    private JTextField propertyValue;
}

```

Exemple 7.12 : SimplePrincipal.java

```

import java.security.*;

/**
 * Un principal avec une valeur nommée (comme "role=HR" ou
 * "username=harry").
 */
public class SimplePrincipal implements Principal
{
    /**
     * Construit un SimplePrincipal pour contenir une description
     * et une valeur.
     * @param roleName Le nom du rôle
     */
    public SimplePrincipal(String descr, String value) {
        this.descr = descr; this.value = value; }

    /**
     * Renvoie le nom du rôle de ce principal
     * @return Le nom du rôle
     */
}

```

```

    */
    public String getName() { return descr + "=" + value; }

    public boolean equals(Object otherObject)
    {
        if (this == otherObject) return true;
        if (otherObject == null) return false;
        if (getClass() != otherObject.getClass()) return false;
        SimplePrincipal other = (SimplePrincipal) otherObject;
        return getName().equals(other.getName());
    }
    public int hashCode() { return getName().hashCode(); }

    private String descr;
    private String value;
}

```

Exemple 7.13 : SimpleLoginModule.java

```

import java.io.*;
import java.lang.reflect.*;
import java.security.*;
import java.util.*;
import javax.security.auth.*;
import javax.security.auth.login.*;
import javax.security.auth.callback.*;
import javax.security.auth.spi.*;

import javax.swing.*;

/**
 * Ce module d'identification authentifie les utilisateurs en lisant
 * les noms d'utilisateur, les mots de passe et les rôles
 * à partir d'un fichier texte.
 */
public class SimpleLoginModule implements LoginModule
{
    public void initialize(Subject subject,
        CallbackHandler callbackHandler,
        Map<String, ?> sharedState, Map<String, ?> options)
    {
        this.subject = subject;
        this.callbackHandler = callbackHandler;
        this.sharedState = sharedState;
        this.options = options;
    }

    public boolean login() throws LoginException
    {
        if (callbackHandler == null)
            throw new LoginException("pas de gestionnaire");

        NameCallback nameCall = new NameCallback("nom d'utilisateur : ");
        PasswordCallback passCall = new PasswordCallback(
            "mot de passe : ", false);
        try
        {
            callbackHandler.handle(new Callback[] { nameCall, passCall });
        }
    }
}

```

574 Au cœur de Java 2 - Fonctions avancées

```
        catch (UnsupportedCallbackException e)
        {
            LoginException e2 = new LoginException("Rappel non accepté");
            e2.initCause(e);
            throw e2;
        }
        catch (IOException e)
        {
            LoginException e2 = new LoginException("
            Exception d'E/S dans le rappel");
            e2.initCause(e);
            throw e2;
        }

        return checkLogin(nameCall.getName(), passCall.getPassword());
    }

    /**
     * Vérifie si les informations d'authentification sont valides.
     * Si oui, le subject acquiert les principals pour le
     * nom d'utilisateur et le rôle.
     * @param username Le nom d'utilisateur
     * @param password Un tableau de caractères contenant le mot de passe
     * @return true si les informations d'authentification sont valides
     */
    private boolean checkLogin(String username, char[] password)
        throws LoginException
    {
        try
        {
            Scanner in = new Scanner(new FileReader("'" +
                options.get("pwfile")));
            while (in.hasNextLine())
            {
                String[] inputs = in.nextLine().split("\\|");
                if (inputs[0].equals(username) &&
                    Arrays.equals(inputs[1].toCharArray(), password))
                {
                    String role = inputs[2];
                    Set<Principal> principals = subject.getPrincipals();
                    principals.add(new SimplePrincipal("username", username));
                    principals.add(new SimplePrincipal("role", role));
                    return true;
                }
            }
            in.close();
            return false;
        }
        catch (IOException e)
        {
            LoginException e2 = new LoginException(
                "Impossible d'ouvrir le fichier de mot de passe");
            e2.initCause(e);
            throw e2;
        }
    }
}
```

```

    public boolean logout() { return true; }
    public boolean abort() { return true; }
    public boolean commit() { return true; }

    private Subject subject;
    private CallbackHandler callbackHandler;
    private Map<String, ?> sharedState;
    private Map<String, ?> options;
}

```

Exemple 7.14 : SimpleCallbackHandler.java

```

import javax.security.auth.callback.*;

/**
 * Ce gestionnaire de rappel simple présente le nom de l'utilisateur
 * et son mot de passe.
 */
public class SimpleCallbackHandler implements CallbackHandler
{
    /**
     * Construit le gestionnaire de rappel.
     * @param username Le nom de l'utilisateur
     * @param password Un tableau de caractères contenant le mot de passe
     */
    public SimpleCallbackHandler(String username, char[] password)
    {
        this.username = username;
        this.password = password;
    }

    public void handle(Callback[] callbacks)
    {
        for (Callback callback : callbacks)
        {
            if (callback instanceof NameCallback)
            {
                ((NameCallback) callback).setName(username);
            }
            else if (callback instanceof PasswordCallback)
            {
                ((PasswordCallback) callback).setPassword(password);
            }
        }
    }

    private String username;
    private char[] password;
}

```

Exemple 7.15 : JAASTest.policy

```

grant codebase "file:login.jar"
{
    permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
    permission javax.security.auth.AuthPermission
        "createLoginContext.Login1";
}

```

576 *Au cœur de Java 2 - Fonctions avancées*

```

    permission javax.security.auth.AuthPermission "doAsPrivileged";
    permission javax.security.auth.AuthPermission "modifyPrincipals";
    permission java.io.FilePermission "password.txt", "read";
};

grant principal SimplePrincipal "role=admin"
{
    permission java.util.PropertyPermission "*", "read";
};

```

Exemple 7.16 : jaas.config

```

Login1
{
    SimpleLoginModule required pwfile="password.txt";
};

```

javax.security.auth.callback.CallbackHandler 1.4

- void handle(Callback[] callbacks)
Gère les rappels donnés, en interagissant avec l'utilisateur si besoin est, et stocke les informations de sécurité dans les objets de rappel.

javax.security.auth.callback.NameCallback 1.4

- NameCallback(String prompt)
- NameCallback(String prompt, String defaultName)
Construisent un NameCallback avec l'invite donnée et un nom par défaut.
- void setName(String name)
- String getName()
Définissent ou récupèrent le nom collecté par ce rappel.
- String getPrompt()
Récupère l'invite à utiliser pour demander ce nom.
- String getDefaultName()
Récupère le nom par défaut à utiliser pour demander ce nom.

javax.security.auth.callback.PasswordCallback 1.4

- PasswordCallback(String prompt, boolean echoOn)
Construit un PasswordCallback avec l'invite donnée et un indicateur d'écho.
- void setPassword(char[] password)
- char[] getPassword()
Définissent ou récupèrent le mot de passe retourné par ce rappel.
- String getPrompt()
Récupère l'invite à utiliser pour demander ce mot de passe.
- boolean isEchoOn()
Récupère l'indicateur d'écho à utiliser pour demander ce mot de passe.

javax.security.auth.spi.LoginModule 1.4

- `void initialize(Subject subject, CallbackHandler handler, Map<String,?> sharedState, Map<String,?> options)`
Initialise ce `LoginModule` pour authentifier le *subject* donné. Lors du traitement de l'identification, utilisez le gestionnaire donné pour rassembler les informations d'identification. Utilisez la carte `sharedState` pour communiquer avec les autres modules d'identification. La carte des options contient des paires nom/valeur spécifiées dans la configuration de l'identification pour cette instance de module.
- `boolean login()`
Réalise la procédure d'authentification et remplit le *principals* du *subject*. Renvoie `true` si l'identification a réussi.
- `boolean commit()`
Est appelé lorsque tous les modules d'identification ont réussi, pour des scénarios d'identification exigeant un engagement en deux phases. Renvoie `true` si l'opération a réussi.
- `boolean abort()`
Est appelé si l'échec d'un autre module d'identification a fait arrêter le processus. Renvoie `true` si l'opération a réussi.
- `boolean logout()`
Se déconnecte de ce *subject*. Renvoie `true` si l'opération a réussi.

Signatures numériques

Comme nous l'avons dit précédemment, les applets ont été le point de départ de l'engouement pour la plate-forme Java. Dans la pratique, les gens ont découvert que, bien qu'ils puissent écrire des applets animées, quantité de choses utiles n'étaient pas réalisables dans le modèle de sécurité JDK 1.0. Par exemple, le fait que les applets sous JDK 1.0 soient si étroitement surveillées ne les rend pas très adaptées à un intranet d'entreprise, même si peu de risques sont inhérents à l'exécution d'une applet à partir de l'environnement relativement sûr de votre intranet d'entreprise. Sun a vite compris que, pour que les applets deviennent réellement efficaces rapidement, il était important que les utilisateurs puissent affecter *différents* niveaux de sécurité, en fonction de l'origine de l'applet. Si une applet provient d'un fournisseur fiable et qu'elle n'a pas été falsifiée, l'utilisateur de cette applet peut alors décider de lui accorder plus de privilèges.

Ce contrôle supplémentaire est maintenant possible grâce au mécanisme de signature d'applet de Java 1.1. Pour assurer plus de fiabilité à une applet, nous devons savoir deux choses :

1. Quelle est la provenance de l'applet ?
2. Le code a-t-il été modifié entre-temps ?

Ces cinquante dernières années, les mathématiciens et les informaticiens ont développé des algorithmes sophistiqués pour assurer l'intégrité des données et mettre au point des signatures électroniques. Le package `java.security` contient les implémentations de nombre de ces algorithmes. Fort heureusement, il n'est pas nécessaire de comprendre la logique sous-jacente pour utiliser les algorithmes du package `java.security`. Dans les sections suivantes, vous verrez comment les condensés de message (*message digests*) peuvent détecter les modifications dans les fichiers de données, et comment les signatures numériques peuvent authentifier l'identité du signataire.

Les condensés de message

Un condensé de message est une empreinte numérique d'un bloc de données. Par exemple, l'algorithme SHA1 (algorithme sûr de hachage n°1) condense n'importe quel bloc de données, quelle que soit sa longueur, en une séquence de 160 bits (20 octets). Tout comme pour les empreintes digitales, on espère que deux messages ne peuvent avoir la même empreinte SHA1. Bien sûr, cela ne peut être vrai — il n'existe que 2^{160} empreintes SHA1, il doit donc bien se trouver quelques messages ayant des empreintes identiques. Mais 2^{160} est suffisamment confortable pour que la probabilité de duplication soit négligeable. Négligeable dans quelle proportion ? Selon James Walsh, dans *True Odds — How Risks Affect Your Everyday Life* (Merritt Publishing, 1996), vos chances de mourir frappé par un éclair sont d'environ une sur 30 000. Pensez à présent à 9 autres personnes, par exemple 9 parmi vos collègues les moins sympathiques. Les chances que vous et *chacun d'entre eux* meurent foudroyés par un éclair sont plus importantes que le risque d'avoir un message contrefait avec la même empreinte SHA1 que l'original. (Bien entendu, plus de 10 personnes, que vous ne connaîtrez probablement jamais, mourront foudroyées. Mais nous parlons de l'éventualité beaucoup plus mince que les gens *précisément choisis par vous* puissent être frappés.)

Un condensé de message a deux propriétés essentielles :

1. Si un ou plusieurs bits de données sont modifiés, le condensé de message change lui aussi.
2. Un pirate en possession d'un message donné ne peut construire un faux message ayant le même condensé de message que l'original.

La seconde propriété est ici encore une question de probabilité, bien sûr. Considérez le message suivant du père du milliardaire :

"A ma mort, mes biens seront répartis également entre mes enfants ;
toutefois mon fils George ne recevra rien."

L'empreinte SHA1 de ce message est :

2D 8B 35 F3 BF 49 CD B1 94 04 E0 66 21 2B 5E 57 70 49 E1 7E

Le père méfiant a déposé le message chez un notaire et l'empreinte chez un autre. Supposons maintenant que George puisse soudoyer le notaire détenant le message et modifier le message afin que ce soit Bill qui n'ait rien. Bien entendu, cela modifie l'empreinte, qui devient totalement différente :

2A 33 0B 4B B3 FE CC 1C 9D 5C 01 A7 09 51 0B 49 AC 8F 98 92

Est-ce que George peut trouver une autre formulation qui corresponde à l'empreinte ? S'il avait été l'heureux possesseur d'un milliard d'ordinateurs depuis la nuit des temps, chacun d'eux calculant un million de messages par seconde, il n'aurait pas encore trouvé à l'heure actuelle.

Un certain nombre d'algorithmes ont été conçus pour le calcul de ces condensés de message. Les plus connus sont SHA1, l'algorithme sûr de hachage développé par le NIST (*National Institute of Standards and Technology*), et MD5, un algorithme mis au point par Ronald Rivest du *Massachusetts Institut of Technology*. Ces deux algorithmes cryptent les bits d'un message de manière ingénieuse. Pour plus de détails sur ces algorithmes, consultez par exemple l'ouvrage (en langue anglaise) *Cryptography and Network Security*, de William Stallings (Prentice-Hall, 1998). Il est à noter que, récemment, des régularités subtiles ont été détectées dans MD5. La plupart des cryptographes recommandent pour cette raison d'utiliser plutôt SHA1. (Ces deux algorithmes sont faciles à mettre en œuvre.)

Le langage Java implémente à la fois SHA1 et MD5. La classe `MessageDigest` est un *factory* (référence) pour la création d'objets qui encapsulent les algorithmes d'empreinte. Elle possède une méthode statique, appelée `getInstance`, qui renvoie un objet d'une classe qui étend la classe `MessageDigest`. Cela signifie que la classe `MessageDigest` a deux rôles :

- celui de classe *factory* ;
- celui de superclasse pour tous les algorithmes de condensés de message.

Voici, par exemple, comment vous obtenez un objet pouvant calculer des empreintes SHA :

```
MessageDigest alg = MessageDigest.getInstance("SHA-1");
```

(Pour obtenir un objet pouvant calculer une empreinte MD5, utilisez la chaîne "MD5" comme argument de `getInstance`.)

Après avoir obtenu un objet `MessageDigest`, vous l'alimentez avec tous les octets du message en appelant itérativement la méthode `update`. Par exemple, le code suivant passe tous les octets d'un fichier à un objet `alg` créé plus haut pour réaliser l'empreinte :

```
InputStream in = ...
int ch;
while ((ch = in.read()) != -1)
    alg.update((byte) ch);
```

Vous pouvez également, si les octets sont dans un tableau, mettre à jour la totalité du tableau en une seule opération :

```
byte[] bytes = . . . ;
alg.update(bytes);
```

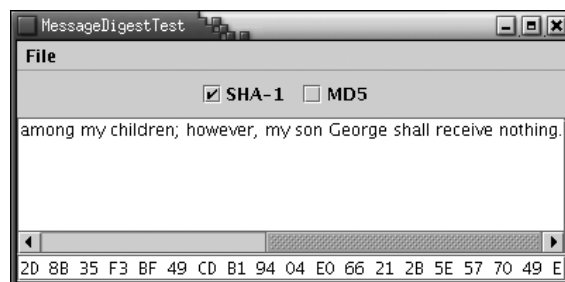
Lorsque c'est fait, appelez la méthode `digest`. Cette méthode complète l'entrée — comme requis par l'algorithme d'empreinte — fait le calcul et renvoie le condensé sous la forme d'un tableau d'octets.

```
byte[] hash = alg.digest();
```

Le programme de l'Exemple 7.17 calcule un condensé de message en utilisant soit SHA soit MD5. Vous pouvez charger les données à condenser à partir d'un fichier, ou taper un message dans la zone de texte. La Figure 7.10 montre l'application.

Figure 7.10

Calcul d'un condensé de message.



Exemple 7.17 : `MessageDigestTest.java`

```
import java.io.*;
import java.security.*;
import java.awt.*;
```

580 *Au cœur de Java 2 - Fonctions avancées*

```
import java.awt.event.*;
import javax.swing.*;

/**
 * Ce programme calcule le condensé du message d'un fichier
 * ou le contenu d'une zone de texte.
 */
public class MessageDigestTest
{
    public static void main(String[] args)
    {
        JFrame f = new MessageDigestFrame();
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
    }
}

/**
 * Ce bloc contient un menu permettant de calculer le condensé
 * du message d'un fichier ou d'une zone de texte, de boutons
 * pour basculer entre SHA-1 et MD5, une zone de texte et un
 * champ de texte affichant le condensé du message.
 */
class MessageDigestFrame extends JFrame
{
    public MessageDigestFrame()
    {
        setTitle("MessageDigestTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        JPanel panel = new JPanel();
        ButtonGroup group = new ButtonGroup();
        addRadioButton(panel, "SHA-1", group);
        addRadioButton(panel, "MD5", group);

        add(panel, BorderLayout.NORTH);
        add(new JScrollPane(message), BorderLayout.CENTER);
        add(digest, BorderLayout.SOUTH);
        digest.setFont(new Font("Monospaced", Font.PLAIN, 12));

        setAlgorithm("SHA-1");

        JMenuBar menuBar = new JMenuBar();
        JMenu menu = new JMenu("Fichier");
        JMenuItem fileDigestItem = new JMenuItem("Condensé fichier");
        fileDigestItem.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                { loadFile(); }
            });
        menu.add(fileDigestItem);
        JMenuItem textDigestItem
            = new JMenuItem("Condensé zone texte");
        textDigestItem.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {

```

```

        String m = message.getText();
        computeDigest(m.getBytes());
    }
});
menu.add(textDigestItem);
menuBar.add(menu);
setJMenuBar(menuBar);
}

/**
 * Ajoute un bouton pour sélectionner un algorithme.
 * @param c le conteneur dans lequel placer le bouton
 * @param name le nom de l'algorithme
 * @param g le groupe de boutons
 */
public void addRadioButton(Container c, final String name,
    ButtonGroup g)
{
    ActionListener listener = new
        ActionListener()
        { public void actionPerformed(ActionEvent event)
            { setAlgorithm(name); }
        };
    JRadioButton b
        = new JRadioButton(name, g.getButtonCount() == 0);
    c.add(b);
    g.add(b);
    b.addActionListener(listener);
}

/**
 * Définit l'algorithme utilisé pour calculer le condensé.
 * @param alg le nom de l'algorithme
 */
public void setAlgorithm(String alg)
{
    try
    {
        currentAlgorithm = MessageDigest.getInstance(alg);
        digest.setText("");
    }
    catch (NoSuchAlgorithmException e)
    {
        digest.setText("" + e);
    }
}

/**
 * Charge un fichier et calcule son condensé de message.
 */
public void loadFile()
{
    JFileChooser chooser = new JFileChooser();
    chooser.setCurrentDirectory(new File("."));

    int r = chooser.showOpenDialog(this);
    if(r == JFileChooser.APPROVE_OPTION)
    {
        try

```

582 *Au cœur de Java 2 - Fonctions avancées*

```

        {
            String name
                = chooser.getSelectedFile().getAbsolutePath();
            computeDigest(loadBytes(name));
        }
        catch (IOException e)
        {
            JOptionPane.showMessageDialog(null, e);
        }
    }
}

/**
 * Charge les octets dans un fichier.
 * @param name le nom du fichier
 * @return un tableau avec les octets du fichier
 */
public byte[] loadBytes(String name) throws IOException
{
    FileInputStream in = null;

    in = new FileInputStream(name);
    try
    {
        ByteArrayOutputStream buffer
            = new ByteArrayOutputStream();
        int ch;
        while ((ch = in.read()) != -1)
            buffer.write(ch);
        return buffer.toByteArray();
    }
    finally
    {
        in.close();
    }
}

/**
 * Calcule le condensé du message d'un tableau d'octets et
 * l'affiche dans le champ de texte.
 * @param b les octets pour lesquels le condensé du message
 * doit être calculé
 */
public void computeDigest(byte[] b)
{
    currentAlgorithm.reset();
    currentAlgorithm.update(b);
    byte[] hash = currentAlgorithm.digest();
    String d = "";
    for (int i = 0; i < hash.length; i++)
    {
        int v = hash[i] & 0xFF;
        if (v < 16) d += "0";
        d += Integer.toString(v, 16).toUpperCase() + " ";
    }
    digest.setText(d);
}

private JTextArea message = new JTextArea();

```

```

private JTextField digest = new JTextField();
private MessageDigest currentAlgorithm;
private static final int DEFAULT_WIDTH = 400;
private static final int DEFAULT_HEIGHT = 300;
}

```

java.security.MessageDigest 1.1

- `static MessageDigest getInstance(String algorithm)`
Renvoie un objet `MessageDigest` qui implémente l'algorithme spécifié. Lance une exception `NoSuchAlgorithmException` si l'algorithme n'est pas fourni.
Paramètres : `algorithm` le nom de l'algorithme, tel que "SHA-1" ou "MD5"
- `void update(byte input)`
- `void update(byte[] input)`
- `void update(byte[] input, int offset, int len)`
Mettent à jour le condensé, à l'aide des octets spécifiés.
- `byte[] digest()`
Exécute le calcul de hachage, renvoie le condensé calculé et réinitialise l'objet `algorithm`.
- `void reset()`
Réinitialise le condensé.

Les signatures numériques

Dans la dernière section, vous avez vu comment calculer un condensé de message, une empreinte pour le message original. Si le message est modifié, l'empreinte du message altéré ne correspondra pas à l'empreinte de l'original. Si le message et son empreinte sont transmis séparément, le destinataire peut vérifier si le message a été falsifié. Si cependant, à la fois le message et l'empreinte ont été interceptés, il est facile de modifier le message, puis de recalculer l'empreinte. Après tout, les algorithmes de condensé de message sont accessibles à tout le monde et ne nécessitent pas de clé d'accès. Dans ce cas, le destinataire du message trafiqué et de l'empreinte recalculée ne saura jamais que le message a été altéré. Dans cette section, vous verrez comment les signatures numériques peuvent authentifier un message. Lorsqu'un message est authentifié, vous *savez* :

- que le message n'a pas été altéré ;
- que le message provient de l'expéditeur déclaré.

Pour comprendre comment fonctionnent les signatures numériques, il est nécessaire d'expliquer quelques-uns des concepts de ce que l'on appelle *cryptographie de clé publique*. La cryptographie de clé publique est fondée sur la notion de clé publique et clé privée. L'idée est de communiquer à tout le monde votre clé publique. Vous êtes toutefois le seul à détenir la clé privée et il est primordial de la conserver secrète et de ne la divulguer sous aucun prétexte. Les clés sont mises en correspondance par relation mathématique, mais il est reconnu comme pratiquement impossible de calculer l'une à partir de l'autre. C'est-à-dire que même si tout le monde connaît votre clé publique, personne ne peut calculer votre clé privée, quelles que soient les ressources informatiques disponibles.

Cela peut sembler difficile à croire, mais personne n'a jamais trouvé d'algorithme capable de le réaliser à partir des algorithmes de cryptage actuellement utilisés. Si les clés sont suffisamment longues, la technique de force brutale — essayer simplement toutes les clés possibles — nécessiterait

584 Au cœur de Java 2 - Fonctions avancées

plus d'ordinateurs qu'on ne peut en construire à partir de tous les atomes du système solaire, en calculant pendant des milliers d'années. Il est bien sûr possible que quelqu'un découvre des algorithmes capables de calculer des clés par un autre moyen plus élaboré. Par exemple, l'algorithme de cryptage RSA (inventé par Rivest, Shamir et Adleman) repose sur la difficulté de calculer de grands nombres. Ces 20 dernières années, les meilleurs mathématiciens ont essayé de trouver des algorithmes, sans succès jusqu'à présent. Pour cette raison, la plupart des cryptographes pensent que les clés avec un "modulo" de 2 000 bits ou plus sont actuellement totalement protégées contre toute intrusion.

Il existe deux sortes de paires de clés publiques/privées : pour le *cryptage* et pour l'*authentification*. Si quelqu'un vous envoie un message qui a été crypté avec votre clé de cryptage publique, vous pouvez le décrypter à l'aide de votre clé de décryptage privée, mais personne d'autre ne le peut. Inversement, si vous signez un message avec votre clé d'authentification privée, tout le monde peut vérifier la signature à l'aide de votre clé publique. Le contrôle n'est positif que pour les messages que vous avez signés et il échoue si quelqu'un d'autre a utilisé sa clé pour signer le message. (Kahn fait remarquer dans la nouvelle édition de son livre, *The Code Breakers*, que c'était la première idée *nouvelle* en matière de cryptographie depuis des centaines d'années.)

De nombreux algorithmes de cryptage, comme RSA et DSA (l'algorithme de signature numérique), utilisent cette idée. La structure exacte des clés et ce que signifie leur correspondance mutuelle dépendent de l'algorithme. Voici par exemple une paire de clés publique/privée DSA.

Clé publique :

```
p: fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae16
17ae01f35b91a47e6df63413c5e12ed0899bcd132acd50d99151bdc43ee7375
92e17
q: 962edddcc369cba8ebb260ee6b6a126d9346e38c5
g: 678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d1427
1b9e35030b71fd73da179069b32e2935630e1c2062354d0da20a6c416e50be794ca4
y: c0b6e67b4ac098eb1a32c5f8c4c1f0e7e6fb9d832532e27d0bdab9ca2d2a
8123ce5a8018b8161a760480fadd040b927281ddb22cb9bc4df596d7de4d1b977d50
```

Clé privée :

```
p: fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae16
17ae01f35b91a47e6df63413c5e12ed0899bcd132acd50d99151bdc43ee737592e17
q: 962edddcc369cba8ebb260ee6b6a126d9346e38c5
g: 678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d1427
1b9e35030b71fd73da179069b32e2935630e1c2062354d0da20a6c416e50be794ca4
x: 146c09f881656cc6c51f27ea6c3a91b85ed1d70a
```

Il y a une relation mathématique entre ces clés, mais la nature exacte de cette relation est sans intérêt en ce qui concerne la programmation (si vous êtes intéressé, vous pouvez consulter l'ouvrage *The Handbook of Applied Cryptography* à l'adresse <http://www.cacr.math.uwaterloo.ca/hac/>).

La question évidente qui se pose est : comment générer la paire de clés ? On le fait habituellement en fournissant le résultat d'un quelconque processus aléatoire à une procédure déterministe qui se charge de renvoyer la paire de clés pour vous. Heureusement, la question de savoir comment obtenir une paire de clés aléatoire pour la cryptographie de clé publique ne concerne que les cryptographes et les mathématiciens.

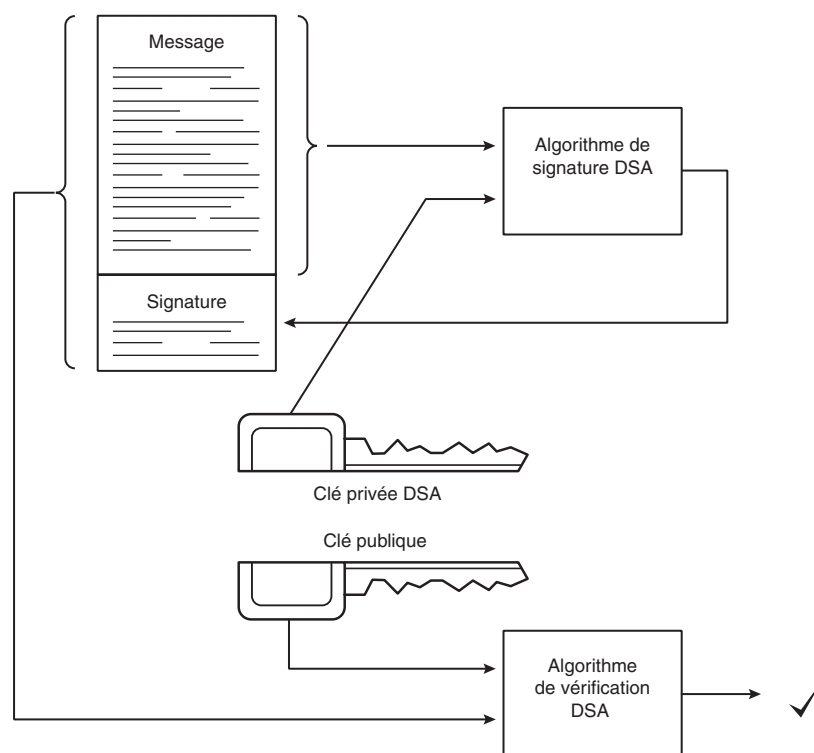
Voici comment cela fonctionne dans la pratique. Supposons qu'Alice veuille envoyer un message à Bob et que Bob veuille être sûr que ce message provient d'Alice et non d'un imposteur. Alice écrit le

message puis *signe* le condensé de message à l'aide de sa clé privée. Bob possède une copie de sa clé publique. Bob applique la clé publique pour *vérifier* la signature. Si la vérification est concluante, Bob peut être certain de deux choses :

1. Le message d'origine n'a pas été altéré.
2. Le message a été signé par Alice, qui détient la clé privée correspondant à la clé publique que Bob a utilisée pour sa vérification.

Voir la Figure 7.11.

Figure 7.11
*Echange de signature
par clé publique
à l'aide de DSA.*



Vous comprenez ainsi pourquoi la sécurité en ce qui concerne les clés privées est si importante. Si quelqu'un dérobe la clé privée d'Alice ou si un gouvernement peut exiger qu'elle la livre, elle aura des problèmes. Le voleur ou un agent gouvernemental peuvent se faire passer pour elle en envoyant des messages, des instructions de transfert de fonds, etc., que les autres penseront provenir d'Alice.

Faisons fonctionner l'algorithme DSA. Il y a en réalité trois algorithmes :

1. pour la génération d'une paire de clés ;
2. pour la signature d'un message ;
3. pour la vérification de la signature.

Bien entendu, vous générez une seule fois une paire de clés, que vous utilisez ensuite pour signer et vérifier de nombreux messages. Pour générer une nouvelle paire de clés aléatoires, vérifiez que vous

586 Au cœur de Java 2 - Fonctions avancées

utilisez des nombres *aléatoires vrais*. Par exemple, le générateur de nombres aléatoires normal de la classe `Random`, initialisé à l'aide des valeurs de date et d'heure actuelles, n'est pas suffisant. (En jargon, on dit que le générateur de nombres aléatoires de base dans `java.util` n'est pas "cryptographiquement sûr".) Supposons par exemple que l'horloge de l'ordinateur ait une précision au 1/10^e de seconde ; il existera au plus 864 000 valeurs de départ possibles par jour. Si un pirate sait quel jour une clé a été créée (facilement déductible de la date d'expiration), il lui est très facile de générer toutes les valeurs initiales possibles pour ce jour précis.

La classe `SecureRandom` génère des nombres aléatoires qui sont de loin beaucoup plus sûrs que ceux de la classe `Random`. Vous devez toujours fournir une valeur initiale pour démarrer la séquence à un point donné. La meilleure méthode est d'obtenir une entrée aléatoire d'un élément matériel comme un générateur de bruit blanc. Une autre source acceptable consiste à demander à l'utilisateur de taper au hasard sur le clavier. Mais chaque frappe ne doit participer que pour un ou deux bits à la valeur de départ. Lorsque vous avez réuni ces bits aléatoires dans un tableau d'octets, vous le passez à la méthode `setSeed`.

```
SecureRandom secrand = new SecureRandom();
byte[] b = new byte[20];
// remplir avec des bits aléatoires vrais
secrand.setSeed(b);
```

Si vous n'initialisez pas le générateur de nombre aléatoire, il calculera sa propre valeur de départ sur 20 octets en lançant des threads, en les mettant en sommeil, et en mesurant leur temps exact de réveil.

INFO

Cela est un algorithme innovant qui, actuellement, n'est *pas* reconnu comme sûr. Et dans le passé, les algorithmes reposant sur la mesure des autres composants de l'ordinateur, comme le temps d'accès au disque dur, se sont révélés ultérieurement comme insuffisamment aléatoires.

Lorsque le générateur est initialisé, vous pouvez extraire des octets aléatoires à l'aide de la méthode `nextBytes`.

```
byte[] randomBytes = new byte[64];
secrand.nextBytes(randomBytes);
```

En réalité, pour calculer une nouvelle clé DSA, vous ne calculez pas les nombres aléatoires vous-même. Vous passez simplement l'objet du générateur de nombre aléatoire à l'algorithme de génération de clé DSA.

Pour créer une nouvelle paire de clés, vous avez besoin d'un objet `KeyPairGenerator`. Tout comme avec la classe `MessageDigest` de la section précédente, la classe `KeyPairGenerator` est à la fois une classe factory et la superclasse pour les algorithmes de génération réelle de la paire de clés. Pour obtenir un générateur de paire de clés DSA, vous appelez la méthode `getInstance` avec la chaîne "DSA".

```
KeyPairGenerator keygen = KeyPairGenerator.getInstance("DSA");
```

L'objet renvoyé est en réalité un objet de la classe `sun.security.provider.DSAKeyPairGenerator`, qui est une sous-classe de `KeyPairGenerator`.

Pour générer des clés, vous devez initialiser l'objet algorithme de génération de clé avec la *densité* (strength) de la clé et un générateur de nombre aléatoire sécurisé. Notez que la densité de clé n'est pas la longueur des clés générées, mais la taille de l'un des blocs de construction de la clé. Dans le cas de DSA, c'est le nombre de bits dans le modulo, l'un des éléments mathématiques qui constituent les clés publiques et privées. Supposons que vous vouliez générer une clé avec un modulo de 512 bits :

```
SecureRandom secrand = new SecureRandom();
secrand.setSeed(...);
keygen.initialize(512, secrand);
```

Vous êtes maintenant prêt à générer des paires de clés.

```
KeyPair keys = keygen.generateKeyPair();
KeyPair morekeys = keygen.generateKeyPair();
```

Chaque paire possède une clé publique et privée.

```
PublicKey pubkey = keys.getPublic();
PrivateKey privkey = keys.getPrivate();
```

Pour signer un message, vous avez besoin d'un objet algorithme de signature. Vous utilisez la classe factory `Signature` :

```
Signature signalg = Signature.getInstance("DSA");
```

Les objets algorithme de signature peuvent être utilisés à la fois pour signer et pour vérifier un message. Pour préparer l'objet pour la signature du message, utilisez la méthode `initSign` et passez la clé privée à l'algorithme de signature.

```
signalg.initSign(privkey);
```

Vous utilisez ensuite la méthode `update` pour ajouter des octets aux objets algorithme, de la même façon que pour l'algorithme de condensé de message.

```
byte[] bytes = ...;
while (...) signalg.update(bytes);
```

Enfin, calculez la signature à l'aide de la méthode `sign`. La signature est renvoyée sous la forme d'un tableau d'octets.

```
byte[] signature = signalg.sign();
```

Le destinataire du message doit obtenir un objet d'algorithme de signature DSA et le préparer pour la vérification de signature en appelant la méthode `initVerify` avec la clé publique comme paramètre.

```
Signature verifyalg = Signature.getInstance("DSA");
verifyalg.initVerify(pubkey);
```

Puis envoyez le message à l'objet algorithme.

```
byte[] bytes = ...;
while (...) verifyalg.update(bytes);
```

Enfin, vérifiez la signature.

```
boolean check = verifyalg.verify(signature);
```

588 Au cœur de Java 2 - Fonctions avancées

Si la méthode `verify` renvoie `true`, la signature du message signé à l'aide de la clé privée correspondante était valide. C'est-à-dire qu'à la fois l'expéditeur et le contenu du message ont été authentifiés.

L'Exemple 7.18 montre les processus de génération, de signature et de vérification de clé. Exécutez le programme comme suit :

```
java SignatureTest -genkey public.key private.key
java SignatureTest -sign sample.txt sample.txt.signed private.key
java SignatureTest -verify sample.txt.signed public.key
```

Le programme doit afficher le message "vérifié". Utilisez ensuite un éditeur hexadécimal pour apporter un petit changement à `sample.txt.signed` et exécutez à nouveau la dernière commande. Vous devriez alors recevoir le message "non vérifié".

Exemple 7.18 : SignatureTest.java

```
import java.io.*;
import java.security.*;

/**
 * Ce programme montre comment signer un message avec une clé
 * DSA privée et le vérifier avec la clé publique correspondante.
 * Utilisation :
 * java SignatureTest -genkey public private
 * java SignatureTest -sign message signed private
 * java SignatureTest -verify signed public
 */
public class SignatureTest
{
    public static void main(String[] args)
    {
        try
        {
            if (args[0].equals("-genkey"))
            {
                KeyPairGenerator pairgen
                    = KeyPairGenerator.getInstance("DSA");
                SecureRandom random = new SecureRandom();
                pairgen.initialize(KEYSIZE, random);
                KeyPair keyPair = pairgen.generateKeyPair();
                ObjectOutputStream out = new ObjectOutputStream(
                    new FileOutputStream(args[1]));
                out.writeObject(keyPair.getPublic());
                out.close();
                out = new ObjectOutputStream(new FileOutputStream(args[2]));
                out.writeObject(keyPair.getPrivate());
                out.close();
            }
            else if (args[0].equals("-sign"))
            {
                ObjectInputStream keyIn = new ObjectInputStream(
                    new FileInputStream(args[3]));
                PrivateKey privkey = (PrivateKey) keyIn.readObject();
                keyIn.close();
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

```

        Signature signalg = Signature.getInstance("DSA");
        signalg.initSign(privkey);

        File infile = new File(args[1]);
        InputStream in = new FileInputStream(infile);
        int length = (int) infile.length();
        byte[] message = new byte[length];
        in.read(message, 0, length);
        in.close();

        signalg.update(message);
        byte[] signature = signalg.sign();

        DataOutputStream out = new DataOutputStream(
            new FileOutputStream(args[2]));
        int signlength = signature.length;
        out.writeInt(signlength);
        out.write(signature, 0, signlength);
        out.write(message, 0, length);
        out.close();
    }
    else if (args[0].equals("-verify"))
    {
        ObjectInputStream keyIn = new ObjectInputStream(
            new FileInputStream(args[2]));
        PublicKey pubkey = (PublicKey) keyIn.readObject();
        keyIn.close();

        Signature verifyalg = Signature.getInstance("DSA");
        verifyalg.initVerify(pubkey);

        File infile = new File(args[1]);
        DataInputStream in = new DataInputStream(
            new FileInputStream(infile));
        int signlength = in.readInt();
        byte[] signature = new byte[signlength];
        in.read(signature, 0, signlength);

        int length = (int) infile.length() - signlength - 4;
        byte[] message = new byte[length];
        in.read(message, 0, length);
        in.close();

        verifyalg.update(message);
        if (!verifyalg.verify(signature))
            System.out.print("non ");
        System.out.println("vérifié");
    }
}
catch (Exception e)
{
    e.printStackTrace();
}
}

private static final int KEYSIZE = 512;
}

```

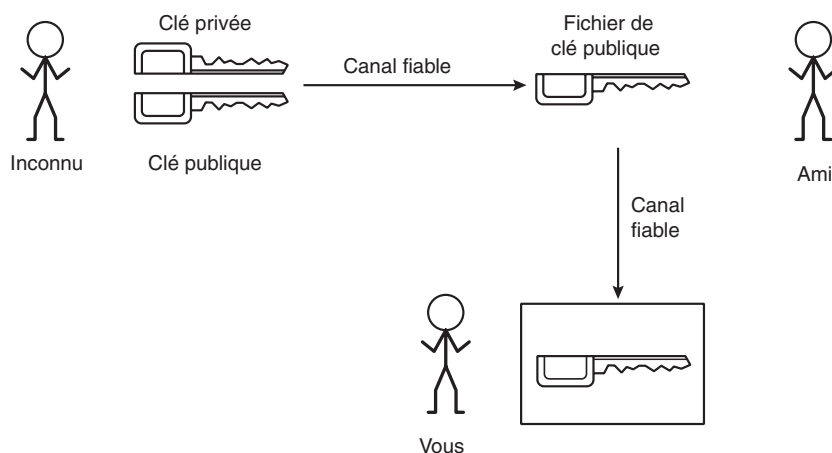

L'authentification des messages

Supposons que vous receviez un message d'un ami, signé par cet ami avec sa clé privée, à l'aide de la méthode que nous venons de présenter. Vous pouvez déjà posséder sa clé publique, ou l'obtenir facilement en lui en demandant une copie ou par l'intermédiaire de sa page Web. Vous pouvez ensuite vérifier que le message a bien été émis par votre ami et n'a pas été falsifié. Supposons maintenant que vous receviez un message d'un inconnu se déclarant représentant d'une célèbre société de logiciels, et vous pressant d'exécuter le programme attaché au message. Ce correspondant vous envoie même une copie de sa clé publique pour vous permettre de vérifier l'origine du message. Vous vérifiez que la signature est valide. Cela prouve que le message a été signé à l'aide de la clé privée correspondante et qu'il n'a pas été modifié.

Mais attention : *vous ne savez toujours pas qui a écrit le message*. N'importe qui peut avoir généré une paire clés publique/privée, signé le message à l'aide de la clé privée, puis vous avoir envoyé le message avec la clé publique. La question de déterminer l'identité de l'expéditeur est appelée *problème d'authentification*.

La manière habituelle de résoudre le problème d'authentification est simple. Supposons que l'expéditeur inconnu et vous ayez une relation commune en qui tous deux avez confiance. Supposons que cet inconnu rencontre votre relation commune en personne et qu'il lui remette un disque contenant sa clé publique. Votre ami vous rencontre plus tard, vous confirme qu'il a rencontré ce correspondant inconnu de vous, que celui-ci travaille bien pour cette fameuse société de logiciels, et il vous donne le disque (voir Figure 7.12). Votre ami se porte ainsi garant de l'authenticité de l'inconnu.

Figure 7.12
Authentification
par un intermédiaire
de confiance.



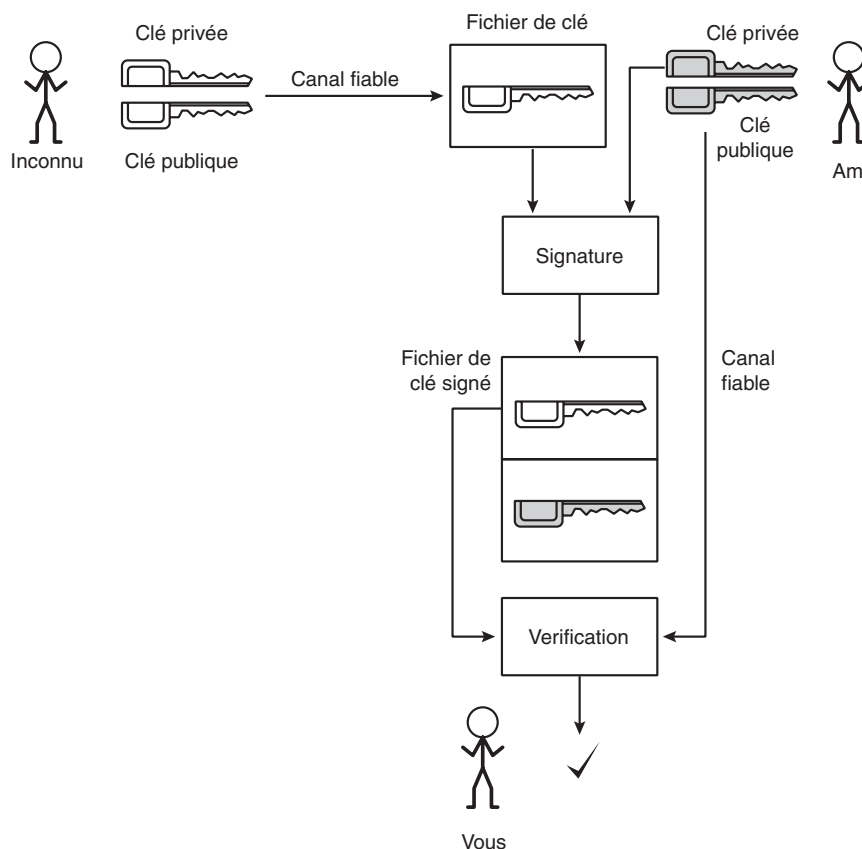
En fait, votre ami n'a pas réellement besoin de vous rencontrer. Il peut simplement appliquer sa signature privée au fichier de clé publique de l'inconnu (voir Figure 7.13).

Lorsque vous obtenez le fichier de clé publique, vous vérifiez la signature de votre ami, et puisque vous lui faites confiance, vous savez qu'il a réellement vérifié l'identité de ce correspondant inconnu avant d'appliquer sa signature.

592 Au cœur de Java 2 - Fonctions avancées

Figure 7.13

Authentification par la signature d'un intermédiaire de confiance.



Vous pouvez toutefois ne pas avoir de relation commune. Certains modèles supposent qu'il existe toujours une "chaîne de confiance" — un réseau de connaissances mutuelles — qui vous permette de faire confiance à chacun des éléments de cette chaîne. Dans la pratique, bien sûr, ce n'est pas toujours vrai. Vous pouvez avoir confiance en votre amie, Alice, et savoir qu'Alice a confiance en Bob, mais vous ne connaissez pas Bob et vous n'êtes pas sûr de pouvoir lui faire confiance. D'autres modèles admettent l'existence d'une sorte de "grand frère" bénévole en qui tout le monde a confiance (la plus connue étant Verisign, Inc. (<http://www.verisign.com>)).

Vous rencontrerez souvent des signatures numériques réalisées par une ou plusieurs entités qui se portent garantes de leur authenticité, et vous devrez évaluer dans quelle mesure vous pouvez faire confiance aux authenticateurs. Vous pouvez peut-être accorder une grande confiance à Verisign, parce que, par exemple, vous avez vu son logo sur de nombreuses pages Web ou que vous avez entendu dire qu'il exigeait la présence dans une chambre forte de plusieurs personnes munies d'attachés-cases noirs, chaque fois que de nouvelles clés doivent être forgées.

Vous devez néanmoins avoir des attentes réalistes en ce qui concerne ce qui est réellement authentifié. Le PDG de Verisign ne rencontre pas personnellement chacun des représentants de sociétés lors de l'authentification d'une clé publique. Vous pouvez obtenir un ID de "classe 1" en complétant un formulaire Web et en payant une petite somme. La clé vous est envoyée à l'adresse e-mail incluse

dans le certificat. Vous pouvez donc raisonnablement penser que l'adresse électronique est authentique, mais le demandeur peut avoir fourni *n'importe quelles* informations de nom et de société. Avec un ID de "classe 1" chez Verisign, ces informations ne sont pas vérifiées. Il existe des classes d'ID plus rigoureuses. Par exemple, pour la "classe 3", Verisign demandera à un souscripteur individuel de se présenter devant un notaire, et il vérifiera la santé financière s'il s'agit d'une entreprise. D'autres authenticateurs appliqueront des procédures différentes. Par conséquent, lorsque vous recevez un message authentifié, il est important de comprendre ce qui est réellement authentifié.

Le format de certification X.509

L'un des formats les plus courants pour les certifications de signature est le format X.509. Il est largement utilisé par Verisign, Microsoft, Netscape et de nombreuses autres sociétés, pour la signature de messages électroniques, l'authentification de code de programme et la certification de nombreuses autres sortes de données. La norme X.509 fait partie de la série X.500 des recommandations pour le service d'annuaire du CCITT (Comité consultatif international télégraphique et téléphonique). Dans sa forme la plus simple, une certification X.509 contient les données suivantes :

- la version du format de certificat ;
- le numéro de série du certificat ;
- l'identifiant de l'algorithme de signature (ID algorithme + paramètres de l'algorithme utilisés pour signer le certificat) ;
- le nom du signataire du certificat ;
- la période de validité (date début/fin) ;
- le nom de l'identité certifiée ;
- la clé publique de l'identité certifiée (ID algorithme + paramètres de l'algorithme + valeur de clé publique) ;
- la signature (code de hachage de tous les champs précédents, codés avec la clé privée du signataire).

Le signataire garantit par conséquent qu'une certaine identité possède une clé publique particulière.

Les extensions du format X.509 de base permettent aux certificats de contenir des informations supplémentaires.

La structure précise des certificats X.509 est décrite dans une notation formelle, appelée "*abstract syntax notation One*" ou ASN.1. La Figure 7.14 donne la définition ASN.1 de la version 3 du format X.509. La syntaxe exacte n'est pas importante pour nous, mais comme vous pouvez le constater, ASN.1 donne une définition précise de la structure d'un fichier de certification. Les *règles d'encodage de base*, ou BER (*Basic encoding rules*), décrivent précisément comment sauvegarder cette structure dans un fichier binaire. C'est-à-dire que les règles BER décrivent comment encoder des entiers, des chaînes de caractères, des chaînes binaires, et construire les éléments SEQUENCE, CHOICE et OPTIONAL.

```
Certificate ::= SEQUENCE {
    tbsCertificate      TBSCertificate,
```


594 Au cœur de Java 2 - Fonctions avancées

```

signatureAlgorithm  AlgorithmIdentifier,
signature           BIT STRING }

TBSCertificate ::= SEQUENCE {
    version          [0] EXPLICIT Version DEFAULT v1,
    serialNumber      CertificateSerialNumber,
    signature         AlgorithmIdentifier,
    issuer            Name,
    validity          Validity,
    subject           Name,
    subjectPublicKeyInfo SubjectPublicKeyInfo,
    issuerUniqueID    [1] IMPLICIT UniqueIdentifier OPTIONAL,
                      -- Si présente, la version doit être v2
                      ou v3
    subjectUniqueID   [2] IMPLICIT UniqueIdentifier OPTIONAL,
                      -- Si présente, la version doit être v2
                      ou v3
    extensions        [3] EXPLICIT Extensions OPTIONAL
                      -- Si présente, la version doit être v3
}

Version ::= INTEGER { v1(0), v2(1), v3(2) }

CertificateSerialNumber ::= INTEGER

Validity ::= SEQUENCE {
    notBefore        CertificateValidityDate,
    notAfter         CertificateValidityDate }

CertificateValidityDate ::= CHOICE {
    utcTime          UTCTime,
    generalTime      GeneralizedTime }

UniqueIdentifier ::= BIT STRING

SubjectPublicKeyInfo ::= SEQUENCE {
    algorithm         AlgorithmIdentifier,
    subjectPublicKey  BIT STRING }

Extensions ::= SEQUENCE OF Extension

Extension ::= SEQUENCE {
    extnID           OBJECT IDENTIFIER,
    critical         BOOLEAN DEFAULT FALSE,
    extnValue        OCTET STRING }

```

INFO

Vous trouverez de plus amples informations sur ASN.1 dans *ASN.1 – Communication Between Heterogeneous Systems*, par Olivier Dubuisson [Academic Press, 2000] (<http://www.oss.com/asn1/dubuisson.html>) et *ASN.1 Complete*, de John Larmouth [Morgan Kaufmann Publishers, 1999] (<http://www.nokalva.com/asn1/larmouth.html>).

```

Certificate ::= SEQUENCE {
    tbsCertificate      TBSCertificate,
    signatureAlgorithm  AlgorithmIdentifier,
    signature            BIT STRING }

TBSCertificate ::= SEQUENCE {
    version             [0] EXPLICIT Version DEFAULT v1,
    serialNumber        CertificateSerialNumber,
    signature            AlgorithmIdentifier,
    issuer              Name,
    validity             Validity,
    subject             Name,
    subjectPublicKeyInfo SubjectPublicKeyInfo,
    issuerUniqueID      [1] IMPLICIT UniqueIdentifier OPTIONAL,
                        -- Si présente, la version doit être v2
                        ou v3
    subjectUniqueID     [2] IMPLICIT UniqueIdentifier OPTIONAL,
                        -- Si présente, la version doit être v2
                        ou v3
    extensions          [3] EXPLICIT Extensions OPTIONAL
                        -- Si présente, la version doit être v3
}

Version ::= INTEGER { v1(0), v2(1), v3(2) }

CertificateSerialNumber ::= INTEGER

Validity ::= SEQUENCE {
    notBefore      CertificateValidityDate,
    notAfter       CertificateValidityDate }

CertificateValidityDate ::= CHOICE {
    utcTime        UTCTime,
    generalTime    GeneralizedTime }

UniqueIdentifier ::= BIT STRING

SubjectPublicKeyInfo ::= SEQUENCE {
    algorithm        AlgorithmIdentifier,
    subjectPublicKey  BIT STRING }

Extensions ::= SEQUENCE OF Extension

Extension ::= SEQUENCE {
    extnID          OBJECT IDENTIFIER,
    critical         BOOLEAN DEFAULT FALSE,
    extnValue        OCTET STRING }
    
```

Figure 7.14

Définition ASN.1 de X.509v3.

596 Au cœur de Java 2 - Fonctions avancées

En réalité, les règles BER ne sont pas uniques ; il y a plusieurs façons de spécifier certains éléments. Les règles DER (*Distinguished encoding rules*) ou règles "*distinguées*" d'encodage suppriment ces ambiguïtés. Si vous souhaitez une description compréhensible du format d'encodage BER, vous pouvez consulter *A Layman's Guide to a Subset of ASN.1, BER, and DER*, de Burton S. Kaliski, Jr., disponible à l'adresse <ftp://ftp.rsa.com/pub/pkcs/ps/layman.ps>. Vous trouverez le code source d'un programme utile pour la recopie (*dumping*) de fichiers codés BER à l'adresse <http://www.cs.auckland.ac.nz/~pgut001/dumpasn1.c>.

INFO

Le site Web de Pete Gutmann (<http://www.cs.auckland.ac.nz/~pgut001/x509guide.txt>) contient une description amusante et informative des nombreuses différences du format X.509, tel qu'implémenté par les différents fournisseurs.

Générer des certificats

JDK est livré avec le programme `keytool`, qui est un outil ligne de commande permettant de générer et de gérer un jeu de certificats. Il est à souhaiter qu'à l'avenir, les fonctionnalités de cet outil soient intégrées dans d'autres programmes plus conviviaux. Mais pour l'instant, nous allons utiliser `keytool` pour montrer comment Alice peut signer un document et l'envoyer à Bob, et comment Bob peut vérifier que le document a bien été signé par Alice et non par un imposteur. Nous n'allons pas aborder toutes les fonctionnalités de `keytool` – reportez-vous à la documentation JDK pour plus d'informations.

Le programme `keytool` gère des magasins de clés (*key stores*), bases de données de certificats et de clés privées. Chaque entrée dans le magasin de clés possède un *alias*. Voici comment Alice crée un magasin de clés `alice.store` et génère une paire de clés avec pour alias `alice`.

```
keytool -genkey -keystore alice.store -alias alice
```

Lorsque vous créez ou ouvrez un magasin de clés, vous êtes invité à fournir un mot de passe pour le magasin de clés. Ici, nous allons simplement utiliser le mot de passe `password`. Si vous deviez utiliser le magasin de clés généré par `keytool` dans la réalité, il conviendrait de choisir un vrai mot de passe et de protéger ce fichier, car il contient les clés de signature privées.

Lorsque vous générez une clé, vous êtes invité à fournir les informations suivantes :

```
Enter keystore password: password
What is your first and last name?
[Unknown]: Alice Lee
What is the name of your organizational unit?
[Unknown]: Engineering Department
What is the name of your organization?
[Unknown]: ACME Software
What is the name of your City or Locality?
[Unknown]: Cupertino
What is the name of your State or Province?
[Unknown]: CA
What is the two-letter country code for this unit?
[Unknown]: US
Is <CN=Alice Lee, OU=Engineering Department, O=ACME Software, L=Cupertino,
ST=CA, C=US> correct?
[no]: Y
```

L'outil `keytool` utilise les noms "distingués" X.500, avec les composants :

Common Name (CN)	prénom et nom
Organizational Unit (OU)	service
Organization (O)	société
Location (L)	ville
State (ST)	état
Country (C)	code pays

pour identifier les propriétaires de clé et les émetteurs de certificat.

Vous devez enfin fournir un mot de passe pour la clé, ou appuyer sur `ENTREE` pour utiliser le mot de passe du magasin de clés comme mot de passe pour la clé.

Supposons qu'Alice veuille donner sa clé publique à Bob. Elle doit exporter un fichier de certificat :

```
keytool -export -keystore alice.store -alias alice -file alice.cert
```

Ensuite, Alice peut envoyer le certificat à Bob. Lorsque Bob reçoit le certificat, il peut l'imprimer :

```
keytool -printcert -file alice.cert
```

L'impression a l'aspect suivant :

```
Owner: CN=Alice Lee, OU=Engineering Department, O=ACME Software, L=Cupertino,
ST=CA, C=US
```

```
Issuer: CN=Alice Lee, OU=Engineering Department, O=ACME Software, L=Cupertino,
ST=CA, C=US
```

```
Serial number: 4145f46b
```

```
Valid from: Mon Sep 13 12:26:35 PDT 2004 until: Sun Dec 12 11:26:35 PST 2004
```

```
Certificate fingerprints:
```

```
MD5:      B6:12:34:88:7C:5E:63:28:21:0D:DC:C9:F1:C1:9C:94
```

```
SHA1:05:AF:B6:0B:8A:D6:86:0A:0D:06:97:FA:9A:63:F4:9A:63:F4:09:0B:64:68:10
```

Ce certificat est *autosigné*. Par conséquent, Bob ne peut pas utiliser un autre certificat garanti pour vérifier que ce certificat est valide. Il peut, à la place, appeler Alice et lui demander de lire l'empreinte du certificat au téléphone.

INFO

Certains émetteurs de certificat publient les empreintes de certificat sur leurs sites Web. Par exemple, JRE (l'environnement d'exécution Java) inclut un magasin de clés `cacerts` dans le répertoire `jre/lib/security`. Il contient les certificats émanant de Thawte et Verisign. Pour répertorier le contenu d'un magasin de clés, utilisez l'option `-list` :

```
keytool -list -v -keystore jre/lib/security/cacerts
```

598 Au cœur de Java 2 - Fonctions avancées

Le mot de passe pour ce magasin de clés est *changeit* (qui signifie *changez-le*). L'un des certificats dans ce magasin de clés est

```
Owner: OU=VeriSign Trust Network, OU="(c) 1998 VeriSign, Inc. - For authorized
use only", OU=Class 1 Public Primary Certification Authority - G2, O="VeriSign,
Inc.", C=US
Issuer: OU=VeriSign Trust Network, OU="(c) 1998 VeriSign, Inc. - For authorized
use only", OU=Class 1 Public Primary Certification Authority - G2, O="VeriSign,
Inc.", C=US
Serial number: 4cc7eaaa983e71d39310f83d3a899192
Valid from: Sun May 17 17:00:00 PDT 1998 until: Tue Aug 01 16:59:59 PDT 2028
Certificate fingerprints:
    MD5:  DB:23:3D:F9:69:FA:4B:B9:95:80:44:73:5E:7D:41:83
    SHA1: 27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
```

Vous pouvez vérifier que vos certificats sont valides en visitant le site Thawte sur le Web à l'adresse <http://www.verisign.com/repository/root.html>.

Lorsque Bob a toute confiance en son certificat, il peut l'importer dans son magasin de clés.

```
keytool -import -keystore bob.store -alias alice -file alice.cert
```

ATTENTION

N'importez jamais dans un magasin de clés un certificat dont vous n'êtes pas absolument sûr. Une fois qu'un certificat est ajouté au magasin de clés, n'importe quel programme utilisant le magasin de clés suppose que le certificat peut être utilisé pour vérifier des signatures.

A présent, Alice peut commencer à envoyer des documents signés à Bob. L'outil `jarsigner` signe et vérifie les fichiers JAR. Alice ajoute simplement le document à signer dans un fichier JAR.

```
jar cvf document.jar document.txt
```

Elle utilise ensuite l'outil `jarsigner` pour ajouter la signature au fichier. Elle doit spécifier le magasin de clés, le fichier JAR et l'alias de la clé à utiliser.

```
jarsigner -keystore alice.store document.jar alice
```

Lorsque Bob reçoit le fichier, il utilise l'option `-verify` du programme `jarsigner`.

```
jarsigner -verify -keystore bob.store document.jar
```

Bob n'a pas besoin de spécifier l'alias de clé. Le programme `jarsigner` trouve le nom X.500 du propriétaire de la clé dans la signature numérique et recherche les certificats correspondants dans le magasin de clés.

Si le fichier JAR n'est pas corrompu et que la signature corresponde, le programme `jarsigner` imprime le texte suivant :

```
jar verified.
```

Dans le cas contraire, le programme affiche un message d'erreur.

Signer des certificats

Dans la précédente section, vous avez vu comment utiliser un certificat autosigné pour communiquer une clé publique à un tiers. Toutefois, le destinataire du certificat devait s'assurer que le certificat était valide en vérifiant l'empreinte auprès de l'émetteur. Le plus souvent, le certificat est signé par un intermédiaire de confiance.

Le kit JDK ne contient pas d'outils pour la signature de certificat. Dans cette section, vous verrez comment écrire un programme qui signe un certificat à l'aide d'une clé privée provenant d'un magasin de clés. Ce programme vous montre comment écrire des programmes accédant au contenu de certificats et de magasins de clés.

Avant d'examiner le code du programme, voyons comment l'utiliser. Supposons qu'Alice veuille envoyer à sa collègue Cindy un message signé. Toutefois, Cindy n'a pas envie d'appeler toutes les personnes qui lui envoient un fichier de signature pour en vérifier l'empreinte. Il est donc nécessaire d'avoir une entité en laquelle Cindy aura confiance pour vérifier les signatures. Dans cet exemple, nous allons supposer que Cindy fasse pour cela appel au service des Ressources d'information de ACME Software. Pour simuler ce processus, nous devons créer un magasin de clés supplémentaire `acmesoft.store`. Générez une clé et exportez le certificat autosigné.

```
keytool -genkey -keystore acmesoft.store -alias acmeroot
keytool -export -alias acmeroot -keystore acmesoft.store
-file acmeroot.cert
```

Puis ajoutez-les au magasin de clés de Cindy.

```
keytool -import -alias acmeroot -keystore cindy.store
-file acmeroot.cert
```

Cindy doit toujours vérifier l'empreinte de ce *certificat racine*, mais à partir de maintenant, elle pourra simplement accepter tous les certificats qui seront signés par lui.

Pour qu'Alice puisse envoyer des messages à Cindy et à toute autre personne de ACME Software, elle doit apporter son certificat au service des Ressources d'information et le faire signer. Toutefois, le programme `keytool` dans JDK ne possède pas cette fonctionnalité. C'est ici que le programme signataire de certificat de l'Exemple 7.19 entre en jeu. Le programme lit un fichier de certificat et le signe avec la clé privée d'un magasin de clés. Un membre habilité de ACME Software va vérifier l'identité d'Alice et générer un certificat signé de la façon suivante :

```
java CertificateSigner -keystore acmesoft.store -alias acmeroot
-infile alice.cert -outfile alice_signedby_acmeroot.cert
```

Le programme signataire de certificat doit avoir accès au magasin de clés de ACME Software, et le membre du personnel doit connaître le mot de passe du magasin de clés. Cela constitue de toute évidence une opération sensible.

Alice donne le fichier `alice_signedby_acmeroot.cert` à Cindy et aux autres membres de ACME Software. Le fichier peut aussi être simplement stocké dans un répertoire d'entreprise de ACME Software. Souvenez-vous que ce fichier contient la clé publique d'Alice et une assertion par ACME Software selon laquelle cette clé appartient bien à Alice.

600 Au cœur de Java 2 - Fonctions avancées

INFO

Le programme `keytool` gère un mécanisme différent pour la signature de clé. L'option `-certreq` produit une demande de certificat sous un format standard qui peut être traité par des autorités de certification comme Verisign ou par un logiciel de traitement local d'autorisation comme le serveur de certificat de Netscape.

Lorsque Cindy importe le certificat signé dans son magasin de clés, le magasin vérifie que la clé a été signée par une clé racine fiable qui se trouve déjà dans le magasin de clés, et elle n'a pas besoin de vérifier l'empreinte du certificat.

Lorsque Cindy a ajouté le certificat racine et les certificats des personnes qui lui envoient régulièrement des documents, elle n'a plus à se préoccuper du magasin de clés.

ATTENTION

Ce scénario n'a qu'une valeur de démonstration. L'outil `keytool` n'est en réalité pas adapté pour un utilisateur final. L'outil `keytool` accepte discrètement les certificats signés par un tiers déjà habilité et demande à l'utilisateur de confirmer seulement ceux qu'il ne peut pas vérifier. Il ne rejette jamais un certificat. Il est beaucoup trop facile pour un utilisateur non averti d'accepter un certificat non valide.

Examinons maintenant le code source de l'Exemple 7.19. Nous chargeons tout d'abord le magasin de clés. La méthode `factory getInstance` de la classe `KeyStore` crée une instance de magasin de clés du type approprié. Le magasin de clés généré par `keytool` est du type "JKS". Le fournisseur de ce type de magasin de clés est "SUN".

```
KeyStore store = KeyStore.getInstance("JKS", "SUN");
```

Nous chargeons ensuite les données du magasin de clés. La méthode `load` requiert une chaîne en entrée et un mot de passe. Notez que le mot de passe est de type `char[]` (un tableau de caractères) et non une chaîne. JVM peut conserver assez longtemps les chaînes avant de récupérer leur emplacement dans le *garbage collector* (ramasse-miettes). Des pirates sont potentiellement capables de découvrir ces chaînes, par exemple en examinant le contenu des fichiers d'échange. Mais les tableaux de caractères peuvent être effacés immédiatement après utilisation.

Voici le code de chargement du magasin de clés. Notez que le mot de passe est remplacé par des espaces aussitôt après utilisation.

```
InputStream in = . . . ;
char[] password = . . . ;
store.load(in, password);
Arrays.fill(password, ' ');
in.close();
```

Nous utilisons ensuite la méthode `getKey` pour extraire la clé privée destinée à la signature. La méthode `getKey` requiert l'alias et le mot de passe de la clé. Sa valeur de retour est du type `Key`, et elle est convertie en `PrivateKey` puisque nous savons que la clé extraite est une clé privée.

```
char[] keyPassword = . . . ;
PrivateKey issuerPrivateKey
    = (PrivateKey) store.getKey(alias, keyPassword);
Arrays.fill(keyPassword, ' ');
```

Nous sommes maintenant prêts à lire le certificat qui doit être signé. La classe `CertificateFactory` peut lire des certificats à partir d'une chaîne en entrée. Vous devez d'abord obtenir un *factory* du type approprié :

```
CertificateFactory factory
    = CertificateFactory.getInstance("X.509");
```

Puis appeler la méthode `generateCertificate` avec une chaîne en entrée :

```
in = new FileInputStream(inname);
X509Certificate inCert
    = (X509Certificate) factory.generateCertificate(in);
in.close();
```

Le type renvoyé par la méthode `generateCertificate` est la classe abstraite `Certificate` qui est la superclasse des classes concrètes telles que `X509Certificate`. Puisque nous savons que le fichier en entrée contient réellement un `X509Certificate`, nous procédons à la conversion.

ATTENTION

Il existe deux types appelés `Certificate` : une interface obsolète du package `java.security`, et une classe abstraite du package `java.security.cert`, qui est la classe que nous voulons utiliser. Si vous importez à la fois les packages `java.security` et `java.security.cert` dans votre programme, vous devez lever l'ambiguïté et faire explicitement référence à la classe `Certificate java.security.cert`.

Le but de ce programme est de signer les octets dans le certificat. Vous extrayez les octets à l'aide de la méthode `getTBSCertificate` :

```
byte[] inCertBytes = inCert.getTBSCertificate();
```

Nous avons ensuite besoin du nom "*distingué*" de l'émetteur, que nous devons insérer dans le certificat signé. Le nom est stocké dans le certificat de l'émetteur dans le magasin de clés. Vous extrayez les certificats du magasin de clés à l'aide de la méthode `getCertificate`. Les certificats étant des informations publiques, vous ne fournissez que l'alias, pas le mot de passe.

```
X509Certificate issuerCert
    = (X509Certificate) store.getCertificate(alias);
```

La valeur renvoyée par la méthode `getCertificate` est du type `Certificate`, mais ici encore, nous savons qu'il s'agit en réalité d'une valeur de type `X509Certificate`. Nous obtenons l'identité de l'émetteur à partir du certificat en appelant la méthode `getSubjectDN`. Cette méthode renvoie un objet d'un type quelconque qui implémente l'interface de `Principal`. En matière de concept, un *principal* est une entité du monde réel, comme une personne, une organisation ou une entreprise.

```
Principal issuer = issuerCert.getSubjectDN();
```

Nous extrayons également le nom de l'algorithme de signature.

```
String issuerSigAlg = issuerCert.getSigAlgName();
```

Nous devons maintenant quitter le domaine de la bibliothèque standard de sécurité, car elle ne contient pas de méthodes permettant de générer de nouveaux certificats. Des bibliothèques pour la génération de certificats sont disponibles chez des fournisseurs tiers comme RSA Security Inc. et Legion of Bouncy Castle (<http://www.bouncycastle.org>). Nous allons cependant utiliser les classes du package `sun.security.x509`. Les précautions d'usage s'appliquent. Ce package peut ne pas être

602 *Au cœur de Java 2 - Fonctions avancées*

fourni par des fournisseurs tiers de la technologie Java, et Sun Microsystems peut en modifier le comportement à tout moment.

Le segment de code qui suit exécute les étapes suivantes :

- Il génère un objet informations de certificat à partir des octets du certificat à signer.
- Il définit le nom de l'émetteur.
- Il crée le certificat et le signe à l'aide de la clé privée de l'émetteur.
- Il sauvegarde le certificat signé dans un fichier.

```
X509CertInfo info = new X509CertInfo(inCertBytes);
info.set(X509CertInfo.ISSUER,
    new CertificateIssuerName((X500Name) issuer));
X509CertImpl outCert = new X509CertImpl(info);
outCert.sign(issuerPrivateKey, issuerSigAlg);
outCert.derEncode(out);
```

Nous n'allons pas voir ces classes en détail puisque nous ne recommandons pas l'utilisation des bibliothèques Sun pour un code en production. Une future version de JDK contiendra peut-être des classes permettant la génération de certificat. En attendant, vous pouvez utiliser des bibliothèques de fournisseurs tiers ou simplement un logiciel existant de génération de certificat.

Exemple 7.19 : *CertificateSigner.java*

```
import java.io.*;
import java.security.*;
import java.security.cert.*;
import java.util.*;

import sun.security.x509.X509CertInfo;
import sun.security.x509.X509CertImpl;
import sun.security.x509.X500Name;
import sun.security.x509.CertificateIssuerName;

/**
 * Ce programme signe un certificat, à l'aide de la clé privée
 * d'un autre certificat d'un magasin de clés.
 */
public class CertificateSigner
{
    public static void main(String[] args)
    {
        String ksname = null; // nom du magasin de clés
        String alias = null; // alias de clé privée
        String inname = null; // nom du fichier en entrée
        String outname = null; // nom du fichier en sortie
        for (int i = 0; i < args.length; i += 2)
        {
            if (args[i].equals("-keystore"))
                ksname = args[i + 1];
            else if (args[i].equals("-alias"))
                alias = args[i + 1];
            else if (args[i].equals("-infile"))
                inname = args[i + 1];
        }
    }
}
```

```
        else if (args[i].equals("-outfile"))
            outname = args[i + 1];
        else usage();
    }

    if (ksname == null || alias == null ||
        inname == null || outname == null) usage();

    try
    {
        PushbackReader console = new PushbackReader(new
            InputStreamReader(System.in));

        KeyStore store = KeyStore.getInstance("JKS", "SUN");
        InputStream in = new FileInputStream(ksname);
        System.out.print("Mot de passe du magasin de clés : ");
        System.out.flush();
        char[] password = readPassword(console);
        store.load(in, password);
        Arrays.fill(password, ' ');
        in.close();

        System.out.print("Mot de passe de la clé pour " + alias);
        System.out.flush();
        char[] keyPassword = readPassword(console);
        PrivateKey issuerPrivateKey
            = (PrivateKey) store.getKey(alias, keyPassword);
        Arrays.fill(keyPassword, ' ');

        if (issuerPrivateKey == null)
            error("Clé privée introuvable");

        in = new FileInputStream(inname);

        CertificateFactory factory
            = CertificateFactory.getInstance("X.509");

        X509Certificate inCert
            = (X509Certificate) factory.generateCertificate(in);
        in.close();
        byte[] inCertBytes = inCert.getTBSCertificate();

        X509Certificate issuerCert
            = (X509Certificate) store.getCertificate(alias);
        Principal issuer = issuerCert.getSubjectDN();
        String issuerSigAlg = issuerCert.getSigAlgName();

        FileOutputStream out = new FileOutputStream(outname);

        X509CertInfo info = new X509CertInfo(inCertBytes);
        info.set(X509CertInfo.ISSUER,
            new CertificateIssuerName((X500Name) issuer));

        X509CertImpl outCert = new X509CertImpl(info);
        outCert.sign(issuerPrivateKey, issuerSigAlg);
        outCert.derEncode(out);

        out.close();
    }
}
```

604 *Au cœur de Java 2 - Fonctions avancées*

```

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }

    /**
     * Lit un mot de passe.
     * @param in le lecteur à partir duquel lire le mot de passe
     * @return un tableau de caractères contenant le mot de passe
     */
    public static char[] readPassword(PushbackReader in)
        throws IOException
    {
        final int MAX_PASSWORD_LENGTH = 100;
        int length = 0;
        char[] buffer = new char[MAX_PASSWORD_LENGTH];

        while (true)
        {
            int ch = in.read();
            if (ch == '\r' || ch == '\n' || ch == -1
                || length == MAX_PASSWORD_LENGTH)
            {
                if (ch == '\r') // fin de ligne DOS "\r\n"
                {
                    ch = in.read();
                    if (ch != '\n' && ch != -1) in.unread(ch);
                }
                char[] password = new char[length];
                System.arraycopy(buffer, 0, password, 0, length);
                Arrays.fill(buffer, ' ');
                return password;
            }
            else
            {
                buffer[length] = (char) ch;
                length++;
            }
        }
    }

    /**
     * Imprime un message d'erreur et se ferme.
     * @param message
     */
    public static void error(String message)
    {
        System.out.println(message);
        System.exit(1);
    }

    /**
     * Imprime un message d'utilisation et se ferme.
     */
    public static void usage()
    {

```

```

        System.out.println("Usage: java CertificateSigner"
            + " -keystore keyStore -alias issuerKeyAlias"
            + " -infile inputFile -outfile outputFile");
        System.exit(1);
    }
}

```

java.security.KeyStore 1.2

- static getInstance(String type)
- static getInstance(String type, String provider)
Construisent un objet magasin de clés du type donné. Si aucun fournisseur n'est spécifié, le fournisseur par défaut est utilisé. Pour travailler avec les magasins de clés générés par l'outil keytool, spécifiez "JKS" pour le type et "SUN" pour le fournisseur.
- void load(InputStream in, char[] password)
Charge un magasin de clés à partir d'un flot de données. Le mot de passe est conservé dans un tableau de caractères afin de ne pas faire partie du pool de chaîne de JVM.
- Key getKey(String alias, char[] password)
Renvoie une clé privée avec l'alias donné, stockée dans ce magasin de clés.
- Certificate getCertificate(String alias)
Renvoie un certificat pour une clé privée avec l'alias donné, stocké dans ce magasin de clés.

java.security.cert.CertificateFactory 1.2

- CertificateFactory getInstance(String type)
Crée un *factory* de certificat pour le type donné. Le type est un type de certificat tel que "X509".
- Certificate generateCertificate(InputStream in)
Charge un certificat à partir d'un flot d'entrée.

java.security.cert.Certificate 1.2

- PublicKey getPublicKey()
Renvoie la clé publique qui est garantie par ce certificat.
- byte[] getEncoded()
Récupère la forme codée de ce certificat.
- String getType()
Renvoie le type de ce certificat, tel que "X509".

java.security.cert.X509Certificate 1.2

- Principal getSubjectDN()
- Principal getIssuerDN()
Récupèrent les noms *distingués* du propriétaire (ou sujet) et de l'émetteur dans le certificat.
- Date getNotBefore()
- Date getNotAfter()
Récupèrent les dates de début et de fin de la période de validité du certificat.

606 Au cœur de Java 2 - Fonctions avancées

- `BigInteger getSerialNumber()`
Récupère la valeur de numéro de série dans le certificat.
- `String getSigAlgName()`
- `byte[] getSignature()`
Récupèrent le nom de l'algorithme de signature et la signature du propriétaire dans le certificat.
- `byte[] getTBSCertificate()`
Récupère les informations codées DER du certificat qui doivent être signées par l'émetteur du certificat.

La signature de code

L'une des plus importantes utilisations de la technologie d'authentification est la signature de programmes exécutables. En cas de téléchargement de programme, vous êtes bien entendu au courant des dommages que peut provoquer un exécutable. Il peut par exemple être infecté par un virus. Si vous connaissez la provenance du code *et* savez qu'il n'a pas été altéré depuis son origine, votre degré de confiance sera considérablement plus élevé que si vous n'avez pas ces assurances. En fait, si le programme a aussi été écrit en langage Java, vous pouvez tirer parti de cette information pour prendre une décision rationnelle concernant les privilèges que vous accorderez à ce programme. Vous pouvez souhaiter l'utiliser comme une applet normale cantonnée au *bac à sable*, ou bien vouloir lui affecter une série de droits et restrictions. Si par exemple, vous téléchargez un programme de traitement de texte, vous pouvez vouloir lui accorder l'accès à l'imprimante et aux fichiers d'un certain répertoire. Mais en revanche, vous pouvez ne pas souhaiter lui accorder le droit de faire des connexions réseau, afin que le programme ne puisse pas essayer de transmettre vos fichiers à un tiers sans que vous en soyez averti.

Vous savez maintenant comment implémenter ce processus sophistiqué.

- Tout d'abord vous utilisez l'authentification pour vérifier la provenance du code.
- Puis vous exécutez le code avec une règle de sécurité mettant en œuvre les permissions que vous voulez accorder au programme, en fonction de son origine.

Signer des fichiers JAR

Dans cette section, nous vous indiquons comment signer des applets et des applications Web Start à utiliser avec Java Plug-in. Deux scénarios sont possibles :

1. livraison sur un intranet ;
2. livraison sur l'Internet public.

Dans le premier scénario, un administrateur système installe les fichiers de certificats et de règles sur les machines locales. A chaque chargement d'un code signé par l'outil Java Plug-in, celui-ci consulte le magasin de clés pour les signatures, et le fichier de règles pour les permissions. L'installation des certificats et des règles est assez simple et peut être réalisée une fois pour chaque poste. Un utilisateur final peut alors exécuter en dehors du bac à sable des codes d'entreprise signés. Chaque fois qu'un nouveau code est créé ou qu'un code existant est mis à jour, il doit être signé et déployé sur le serveur Web. Toutefois, aucun poste n'a besoin d'être touché en cas d'évolution des programmes.

Ce scénario nous paraît une alternative tentante par rapport au déploiement des applications d'entreprise sur chacun des postes.

Dans le second scénario, les fournisseurs de logiciel obtiennent des certificats qui sont signés par des autorités de certification comme Verisign. Lorsqu'un utilisateur final visite un site Web qui contient un applet signée, une boîte de dialogue s'affiche qui identifie le fournisseur du logiciel et offre deux choix à l'utilisateur : exécuter l'applet avec tous les privilèges, ou la confiner au bac à sable. Nous verrons ce scénario en détail ultérieurement.

Dans le reste de cette section, nous vous indiquons comment construire des fichiers de règles accordant des permissions spécifiques au code provenant de sources connues. Ce processus de construction et de déploiement de ces fichiers de règles n'est pas destiné à un utilisateur final occasionnel. Toutefois, les administrateurs système peuvent réaliser ces tâches en vue de la distribution de programmes d'intranet.

Supposons qu'ACME Software veuille que ses utilisateurs exécutent certains programmes qui nécessitent un accès fichier local et qu'il souhaite déployer ce programme dans un navigateur, sous la forme d'applets ou d'applications Web Start. Puisque ces programmes ne peuvent s'exécuter dans le bac à sable, ACME Software doit installer des fichiers de règles sur les machines de ses employés. Comme vous l'avez vu précédemment dans ce chapitre, ACME peut identifier les applets par leur origine de code. Mais cela signifie qu'ACME devra mettre à jour les fichiers de règles chaque fois que le code de l'applet sera déplacé sur un serveur Web différent. ACME va plutôt décider de *signer* les fichiers JAR qui contiennent le code de l'applet.

Pour créer un fichier JAR signé, ajoutez d'abord vos fichiers de classe à un fichier JAR. Par exemple :

```
javac FileReadApplet.java
jar cvf FileReaderApplet.jar *.class
```

Exécutez ensuite l'outil `jarsigner` et spécifiez le fichier JAR et l'alias de la clé privée :

```
keytool -genkey -keystore acmesoft.store -alias acmeroot
jarsigner -keystore acmesoft.store FileReaderApplet.jar acmeroot
```

Dans cet exemple, le fichier JAR est signé à l'aide de la clé racine autosignée. ACME pourrait émettre des clés pour ses programmeurs et les signer avec la clé racine.

Bien entendu, le magasin de clés contenant la clé racine privée doit être conservé dans un endroit sûr. Nous allons créer un second magasin de clés `certs.store` pour les certificats et lui ajouter le certificat `acmeroot`.

```
keytool -export -keystore acmesoft.store -alias acmeroot -file
acmeroot.cert
keytool -import -keystore certs.store -alias acmeroot -file
acmeroot.cert
```

Créez ensuite un fichier de règles qui accorde la permission à toutes les applets qui sont signées à l'aide des signatures de ce magasin de clés.

Incluez l'emplacement de votre magasin de clés dans le fichier de règles. Ajoutez une ligne

```
keystore "URLmagasinclés", "Typemagasinclés";
```

en haut du fichier de règles. Le type est `JKS` si le magasin de clés a été généré par `keytool`.

```
keystore "file:certs.store", "JKS";
```

608 Au cœur de Java 2 - Fonctions avancées

Ajoutez ensuite `signedBy "alias"` à une ou plusieurs clauses `grant` dans le fichier de règles. Par exemple :

```
grant signedBy "acmeroot"
{
    . . .
};
```

Tout code signé qui peut être vérifié par la clé publique associée à l'alias se voit accorder ces permissions dans la clause `grant`.

INFO

La plate-forme Java associant l'authentification et le contrôle dans les permissions du code, elle assure une sécurité bien supérieure à la technologie ActiveX de Microsoft. Le code ActiveX est authentifié mais, lorsqu'il obtient la permission de s'exécuter, il ne peut pas être contrôlé. Microsoft a intégré un modèle de sécurité plus complet dans la plate-forme .NET.

Vous pouvez tester le processus de signature de code à l'aide de l'applet de l'Exemple 7.20. L'applet essaie de lire dans un fichier local. La règle de sécurité par défaut permet seulement à l'applet de lire des fichiers à partir de sa base de code et de n'importe quel sous-répertoire. Utilisez `appletviewer` pour exécuter l'applet et vérifier que vous pouvez lire des fichiers à partir du répertoire d'origine du code, mais pas à partir des autres répertoires.

Placez maintenant l'applet dans un fichier JAR et signez-le à l'aide de la clé `acmeroot`. Créez ensuite un fichier de règles `applets.policy` contenant

```
keystore "file:///home/mydir/certs.store", "JKS";
grant signedBy "acmeroot"
{
    permission java.io.FilePermission "<<ALL FILES>>", "read";
};
```

Enfin, dites à l'éditeur d'applets d'utiliser le fichier de règles :

```
appletviewer -J-Djava.security.policy=applet.policy
FileReadApplet.html
```

L'applet peut à présent lire tous les fichiers, ce qui montre que le mécanisme de signature fonctionne.

En guise de test final, vous pouvez exécuter votre applet dans votre navigateur (voir Figure 7.15). Modifiez le fichier `deployment.properties`. Si vous exécutez UNIX ou Linux, le fichier est contenu dans le sous-répertoire `.java/deployment` de votre répertoire racine. Sous Windows, le fichier est situé dans `C:\Documents and Settings\ votreNomLogin\Application Data\Sun\Java\Deployment`.

Ajoutez une ligne avec une URL de fichier pour le fichier de règles, par exemple :

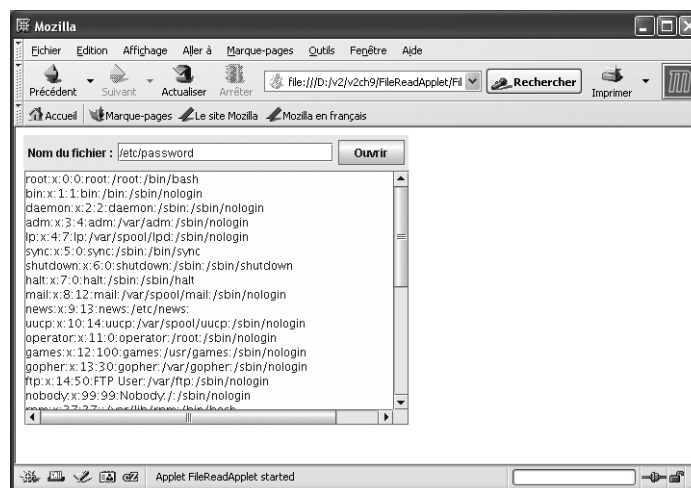
```
deployment.user.security.policy=file:///home/mydir/applet.policy
```

Chargez le fichier `FileReadApplet.html` dans le navigateur. N'acceptez *pas* la signature du fichier JAR (ceci donnerait automatiquement toutes les permissions à l'applet). Néanmoins, vous devriez pouvoir charger tout fichier local, ce qui prouve que le gestionnaire de sécurité a donné à l'applet des droits supplémentaires.

Lorsque vous avez terminé, n'oubliez pas de nettoyer votre fichier `deployment.properties` et supprimez les paramètres du fichier de règles. Si vous chargez à nouveau l'applet après le nettoyage, vous ne devriez plus pouvoir lire les fichiers à partir du système de fichiers local.

Figure 7.15

Une applet signée peut lire des fichiers locaux.



ATTENTION

Si vous testez votre applet à l'aide du navigateur au lieu de l'éditeur d'applets, vous devez savoir que le plug-in ne lit le fichier `deployment.properties` qu'une seule fois, lorsqu'il est chargé pour la première fois. Si vous avez fait une erreur, fermez le navigateur, corrigez votre erreur, puis redémarrez le navigateur. Le fait de simplement recharger la page Web contenant l'applet ne suffit pas.

ATTENTION

Si vous décidez de déployer des applets signées ou des applications Web Start sur un intranet, configurez les fichiers `deployment.properties` des machines de vos utilisateurs. Les sections "Deployment Configuration File and Properties" et "Java Control Panel" du guide de déploiement, à l'adresse <http://java.sun.com/j2se/1.5.0/docs/guide/deployment/deployment-guide/contents.html>, contiennent des instructions détaillées.

Exemple 7.20 : *FileReadApplet.java*

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;

/**
 * Cette applet peut s'exécuter "en dehors du bac à sable" et
 * lire des fichiers locaux lorsqu'elle reçoit les bonnes permissions.
 */
public class FileReadApplet extends JApplet
{
    public FileReadApplet()
```


610 Au cœur de Java 2 - Fonctions avancées

```

    {
        fileNameField = new JTextField(20);
        JPanel panel = new JPanel();
        panel.add(new JLabel("Nom fichier :"));
        panel.add(fileNameField);
        JButton openButton = new JButton("Ouvrir");
        panel.add(openButton);
        openButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    loadFile(fileNameField.getText());
                }
            });

        add(panel, "North");

        fileText = new JTextArea();
        add(new JScrollPane(fileText), "Center");
    }

    /**
     * Charge le contenu d'un fichier dans la zone de texte.
     * @param filename le nom du fichier
     */
    public void loadFile(String filename)
    {
        try
        {
            fileText.setText("");
            Scanner in
                = new Scanner(new FileReader(filename));
            while (in.hasNextLine())
                fileText.append(in.nextLine() + "\n");
            in.close();
        }
        catch (IOException e)
        {
            fileText.append(e + "\n");
        }
        catch (SecurityException e)
        {
            fileText.append("Désolé, c'est interdit.\n");
            fileText.append(e + "\n");
        }
    }

    private JTextField fileNameField;
    private JTextArea fileText;
}

```

Les certificats de développeur de logiciel

Jusqu'à présent, nous avons vu des scénarios où les programmes sont délivrés sur un réseau intranet et où un administrateur système configure des règles de sécurité contrôlant les privilèges des programmes. Toutefois, cette stratégie ne fonctionne qu'avec les programmes dont la source est connue.

Supposons que vous surfiez sur Internet et que vous rencontriez un site Web qui vous propose d'exécuter une applet ou une application Web Start provenant d'un fournisseur non connu, à condition que vous lui accordiez la permission de le faire (voir Figure 7.16). Un tel programme est signé à l'aide du certificat d'un développeur émis par une autorité de certification. La boîte de dialogue qui s'affiche identifie le développeur du logiciel et l'émetteur du certificat. Vous avez deux possibilités :

- exécuter le programme avec tous les privilèges, ou
- confiner le programme au bac à sable.

Figure 7.16

*Lancement
d'une applet signée.*



Quels éléments avez-vous à votre disposition pour vous aider à prendre une décision ? Vous savez que :

1. Thawte a vendu un certificat au développeur du logiciel.
2. Le programme a réellement été signé à l'aide de ce certificat et n'a pas été modifié entre-temps.
3. Le certificat a réellement été signé par Thawte — il a été vérifié à l'aide de la clé publique dans le fichier local cacerts.

Cela signifie-t-il que le code peut être exécuté en toute sécurité ? Allez-vous faire confiance au vendeur si tout ce que vous savez est son nom et le fait que Thawte lui a vendu un certificat de développeur de logiciel ? Il est probable que Thawte a fait certains efforts pour s'assurer que ChemAxon Kft. n'est pas un pirate avéré. Toutefois, aucun émetteur de certificat ne fait une vérification exhaustive concernant l'honnêteté et la compétence des vendeurs de logiciel.

Dans le cas d'un vendeur inconnu, un utilisateur final est assez démuni pour décider en toute connaissance de cause s'il doit laisser ce programme s'exécuter en dehors du bac à sable, avec toutes les permissions d'une application locale. Si le vendeur est une société réputée, l'utilisateur peut au moins tenir compte des antécédents de la société.

INFO

Vous pouvez utiliser des certificats très faibles pour signer du code (voir l'adresse <http://www.dallaway.com/acad/webstart>, qui en présente un très bon exemple). Certains développeurs demandent même à leurs utilisateurs d'ajouter des certificats non confirmés dans leur banque de certificats, comme http://www.agrsrhichome.bnl.gov/Controls/doc/javaws/javaws_howto.html. Du point de vue de la sécurité, cela semble très mauvais.

612 Au cœur de Java 2 - Fonctions avancées

Les situations où un programme demande "*accorde-moi tous les droits, ou je ne ferai rien du tout*" sont réellement inconfortables. Les utilisateurs non avertis sont trop souvent tentés d'accorder des droits d'accès qui peuvent les mettre en danger. Serait-il préférable que le programme explique de quels droits elle a besoin et demande une permission spécifique pour ces droits ? Malheureusement, comme vous l'avez vu, cela peut devenir rapidement très technique. Il ne paraît pas raisonnable, pour un utilisateur final, d'avoir à décider si un programme peut réellement avoir le droit d'inspecter la file d'attente des événements AWT. Nous ne sommes pas très enthousiastes en ce qui concerne les programmes signés à l'aide de certificats de développeurs et dont l'exécution est approuvée par les utilisateurs finaux. Nous vous suggérons de limiter la signature du code aux applications d'entreprise dans lesquelles les certificats sont configurés par un administrateur système.

Le cryptage

Dans ce chapitre, nous avons étudié une importante technique de cryptographie implémentée à l'aide de l'API Java platform security, c'est-à-dire l'authentification par le biais de signatures numériques. Un autre aspect important de la sécurité est le *cryptage*. Lorsque les informations sont authentifiées, les informations elles-mêmes sont totalement visibles. La signature numérique vérifie en fait que les informations n'ont pas été modifiées. En revanche, lorsque les informations sont cryptées, elles ne sont pas visibles. Elles ne peuvent être décryptées qu'à l'aide de la clé correspondante.

L'authentification est suffisante pour la signature de code — il n'est pas nécessaire de masquer le code. Mais le cryptage est nécessaire lorsque les applets ou les applications véhiculent des informations confidentielles, comme des numéros de carte de crédit ou autres données personnelles.

Jusque récemment, des brevets et contrôles d'exportation ont empêché de nombreuses sociétés, notamment Sun, d'offrir un cryptage fort. Heureusement, les contrôles d'exportation sont maintenant bien moins stricts et le brevet d'un important algorithme a finalement expiré. Quant au JDK 1.4, une bonne prise en charge du cryptage fait partie de la bibliothèque standard. La prise en charge du cryptage est également disponible sous forme d'extension séparée (appelée JCE) pour les plus anciennes versions du JDK.

Chiffres symétriques

Les extensions cryptographiques Java contiennent une classe `Cipher`, à savoir une superclasse pour tous les algorithmes de cryptage. Vous obtenez un objet de chiffre en appelant la méthode `getInstance` :

```
Cipher cipher = Cipher.getInstance(algorithmName);  
ou
```

```
Cipher cipher = Cipher.getInstance(algorithmName, providerName);
```

Le JDK est livré avec des chiffres du fournisseur nommés "SunJCE". Il s'agit du fournisseur par défaut si vous ne spécifiez pas d'autre nom de fournisseur. Vous pourrez également avoir besoin d'un autre fournisseur si vous utilisez des algorithmes spécialisés que Sun ne prend pas en charge.

Le nom de l'algorithme est une chaîne du type "AES" ou "DES/CBC/PKCS5Padding".

DES, la norme de cryptage des données (*Data Encryption Standard*), est un très bon chiffre de bloc, contenant une clé longue de 56 bits. De nos jours, l'algorithme DES est considéré comme obsolète car il est devenu possible de le "craquer" par force (voir, par exemple, l'adresse <http://www.eff.org/>

Privacy/Crypto/Crypto_misc/DESCracker/). Son successeur propose une solution bien meilleure, l'AES, la norme de cryptage avancée (*Advanced Encryption Standard*). Reportez-vous au site <http://csrc.nist.gov/encryption/aes/> pour obtenir de plus amples informations sur AES. Nous utilisons AES dans notre exemple.

Dès que vous obtenez un objet de chiffre, vous l'initialisez en définissant le mode et la clé.

```
int mode = . . . ;
Key key = . . . ;
cipher.init(mode, key);
```

Le mode est à choisir parmi

```
Cipher.ENCRYPT_MODE
Cipher.DECRYPT_MODE
Cipher.WRAP_MODE
Cipher.UNWRAP_MODE
```

Les modes d'emballage et de déballage permettent de crypter une clé avec une autre ; reportez-vous à la prochaine section pour en obtenir un exemple.

Vous pouvez maintenant appeler à plusieurs reprises la méthode `update` pour crypter des blocs de données :

```
int blockSize = cipher.getBlockSize();
byte[] inBytes = new byte[blockSize];
. . . // lire inBytes
int outputSize = cipher.outputSize(inLength);
byte[] outBytes = new byte[outputSize];
int outLength = cipher.getOutputSize(inBytes, 0, outputSize, outBytes);
. . . // écrire outBytes
```

Lorsque vous avez terminé, vous devez appeler la méthode `doFinal` une fois. S'il existe un bloc de données d'entrée disponible (avec moins que la valeur `blockSize` en octets), appelez

```
outBytes = cipher.doFinal(inBytes, 0, inLength);
```

Si toutes les données d'entrée ont été cryptées, appelez plutôt

```
outBytes = cipher.doFinal();
```

L'appel à `doFinal` est nécessaire pour réaliser le *remplissage* du bloc final. Observez le chiffre DES. Sa taille de bloc est de 8 octets. Supposons que le dernier bloc des données d'entrée ait moins de 8 octets. Bien entendu, nous pouvons remplir les octets restants avec 0, pour obtenir un bloc final de 8 octets et le crypter. Mais, au moment du décryptage des blocs, le résultat aura plusieurs 0 octets collés à la suite et il sera donc légèrement différent du fichier d'entrée d'origine. Cela peut poser un problème et il est nécessaire d'utiliser un *schéma de remplissage* pour l'éviter. Un schéma de remplissage communément utilisé est celui décrit dans la PKCS (*Public Key Cryptography Standard* ou *Norme de cryptographie de clé publique*) n°5 de RSA Security Inc (<ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf>). Dans ce schéma, le dernier bloc n'est pas rempli avec une valeur de zéro, mais avec une valeur égale au nombre d'octets du remplissage. En d'autres termes, si `L` est le dernier bloc (incomplet), il est rempli comme suit :

```
L 01          if length(L) = 7
L 02 02      if length(L) = 6
L 03 03 03   if length(L) = 5
. . .
L 07 07 07 07 07 07 07 if length(L) = 1
```

614 *Au cœur de Java 2 - Fonctions avancées*

Enfin, si la longueur de l'entrée est véritablement divisible par 8, un bloc

```
08 08 08 08 08 08 08 08
```

est ajouté à l'entrée et crypté. Au moment du décryptage, le tout dernier octet du texte brut correspond à un comptage des caractères de remplissage à rejeter.

Enfin, nous devons expliquer comment obtenir une clé. Vous pouvez générer une clé totalement aléatoire en suivant ces étapes.

1. Obtenez un `KeyGenerator` pour votre algorithme.
2. Initialisez le générateur avec une source aléatoire. Si la longueur du bloc du chiffre est variable, vous devez aussi spécifier la longueur de bloc souhaitée.
3. Appelez la méthode `generateKey`.

Par exemple, voici comment générer une clé AES.

```
KeyGenerator keygen = KeyGenerator.getInstance("AES");
SecureRandom random = new SecureRandom();
keygen.init(random);
Key key = keygen.generateKey();
```

Vous pouvez également produire une clé à partir d'un ensemble fixe de données brutes (peut-être dérivées d'un mot de passe ou du temps de pression sur les touches). Utilisez ensuite un `SecretKeyFactory` comme ceci :

```
SecretKeyFactory keyFactory
    = SecretKeyFactory.getInstance("AES");
byte[] keyData = . . .; // 16 octets pour AES
SecretKeySpec keySpec = new SecretKeySpec(keyData, "AES");
Key key = keyFactory.generateSecret(keySpec);
```

Le programme d'exemple figurant à la fin de cette section fait travailler le chiffre AES (voir Exemple 7.21). Pour utiliser le programme, vous devez tout d'abord générer une clé secrète. Exécutez

```
java AESTest -genkey secret.key
```

La clé secrète est enregistrée dans le fichier `secret.key`.

Vous pouvez maintenant effectuer le cryptage avec la commande

```
java AESTest -encrypt plaintextFile encryptedFile secret.key
```

Vous procéderez au décryptage avec la commande

```
java AESTest -decrypt encryptedFile decryptedFile secret.key
```

Le programme est simple. L'option `-genkey` produit une nouvelle clé secrète et la sérialise dans le fichier donné. Cette opération est longue car l'initialisation du générateur aléatoire sécurisé est elle-même longue. Les options `-encrypt` et `-decrypt` appellent toutes deux la même méthode `crypt` qui appelle les méthodes `update` et `doFinal` du chiffre. Vous remarquerez la manière dont la méthode `update` est appelée tant que les blocs d'entrée ont la longueur complète et que la méthode `doFinal` est soit appelée avec un bloc d'entrée partiel (qui est alors rempli) soit sans données supplémentaires (pour générer un bloc de remplissage).

Exemple 7.21 : `AESTest.java`

```
import java.io.*;
import java.security.*;
```

```
import javax.crypto.*;
import javax.crypto.spec.*;

/**
 * Ce programme teste le chiffre AES. Usage:
 * java AESTest -genkey fichier de clés
 * java AESTest -encrypt fichier de clés crypté en texte brut
 * java AESTest -decrypt fichier de clés crypté et décrypté
 */
public class AESTest
{
    public static void main(String[] args)
    {
        try
        {
            if (args[0].equals("-genkey"))
            {
                KeyGenerator keygen
                    = KeyGenerator.getInstance("AES");
                SecureRandom random = new SecureRandom();
                keygen.init(random);
                SecretKey key = keygen.generateKey();
                ObjectOutputStream out = new ObjectOutputStream(
                    new FileOutputStream(args[1]));
                out.writeObject(key);
                out.close();
            }
            else
            {
                int mode;
                if (args[0].equals("-encrypt"))
                    mode = Cipher.ENCRYPT_MODE;
                else
                    mode = Cipher.DECRYPT_MODE;

                ObjectInputStream keyIn = new ObjectInputStream(
                    new FileInputStream(args[3]));
                Key key = (Key) keyIn.readObject();
                keyIn.close();

                InputStream in = new FileInputStream(args[1]);
                OutputStream out = new FileOutputStream(args[2]);
                Cipher cipher = Cipher.getInstance("AES");
                cipher.init(mode, key);

                crypt(in, out, cipher);
                in.close();
                out.close();
            }
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
        catch (GeneralSecurityException e)
        {
            e.printStackTrace();
        }
        catch (ClassNotFoundException e)
        {
        }
    }
}
```

616 Au cœur de Java 2 - Fonctions avancées

```

        {
            e.printStackTrace();
        }
    }

    /**
     * Utilise un chiffre pour transformer les octets en flux d'entrée
     * et envoie les octets transformés à un flux de sortie.
     * @param in le flux d'entrée
     * @param out le flux de sortie
     * @param cipher le chiffre qui transforme les octets
     */
    public static void crypt(InputStream in, OutputStream out,
        Cipher cipher) throws IOException, GeneralSecurityException
    {
        int blockSize = cipher.getBlockSize();
        int outputSize = cipher.getOutputSize(blockSize);
        byte[] inBytes = new byte[blockSize];
        byte[] outBytes = new byte[outputSize];

        int inLength = 0;
        boolean more = true;
        while (more)
        {
            inLength = in.read(inBytes);
            if (inLength == blockSize)
            {
                int outLength
                    = cipher.update(inBytes, 0, blockSize, outBytes);
                out.write(outBytes, 0, outLength);
            }
            else more = false;
        }
        if (inLength > 0)
            outBytes = cipher.doFinal(inBytes, 0, inLength);
        else
            outBytes = cipher.doFinal();
        out.write(outBytes);
    }
}

```

java.security.Cipher 1.4

- static Cipher getInstance(String algorithm)
- static Cipher getInstance(String algorithm, String provider)

Renvoient un objet Cipher qui implémente l'algorithme spécifié. Déclenchent une NoSuchAlgorithmException si l'algorithme n'est pas fourni.

Paramètres :

algorithm	le nom de l'algorithme, comme "DES" ou "DES/CBC/PKCS5Padding". Contactez votre fournisseur pour de plus amples informations sur les noms d'algorithmes valides.
provider	le nom du fournisseur, comme "SunJCE" ou "BC"

- int getBlockSize()
 - int getOutputSize(int inputLength)
- Renvoie la taille (en octets) d'un bloc de chiffre ou 0 si le chiffrement n'est pas un chiffre de bloc.

Renvoie la taille d'un tampon de sortie nécessaire si l'entrée suivante possède le nombre donné d'octets. Cette méthode prend en compte tout octet mis en tampon dans l'objet de chiffre.

- `void init(int mode, Key key)`

Initialise l'objet d'algorithme de chiffre.

Paramètres :

<code>mode</code>	ENCRYPT_MODE, DECRYPT_MODE, WRAP_MODE ou UNWRAP_MODE
<code>key</code>	la clé à utiliser pour la transformation

- `byte[] update(byte[] in)`
- `byte[] update(byte[] in, int offset, int length)`
- `int update(byte[] in, int offset, int length, byte[] out)`

Transforment un bloc de données d'entrée. Les deux premières méthodes renvoient le résultat. La troisième méthode renvoie le nombre d'octets placés dans out.

Paramètres :

<code>in</code>	les nouveaux octets d'entrée à traiter
<code>offset</code>	l'indice de départ des octets d'entrée dans le tableau in
<code>length</code>	le nombre d'octets d'entrée
<code>out</code>	le tableau dans lequel placer les octets de sortie

- `byte[] doFinal()`
- `byte[] doFinal(byte[] in)`
- `byte[] doFinal(byte[] in, int offset, int length)`
- `int doFinal(byte[] in, int offset, int length, byte[] out)`

Transforment le dernier bloc des données d'entrée et nettoient le tampon de cet objet d'algorithme. Les trois premières méthodes renvoient le résultat. La quatrième méthode renvoie le nombre d'octets placés dans out.

Paramètres :

<code>in</code>	les nouveaux octets d'entrée à traiter
<code>offset</code>	l'indice de départ des octets d'entrée dans le tableau in
<code>length</code>	le nombre d'octets d'entrée
<code>out</code>	le tableau dans lequel placer les octets de sortie

java.security.KeyGenerator 1.4

- `static KeyGenerator getInstance(String algorithm)`

Renvoie un objet KeyGenerator qui implémente l'algorithme spécifié. Déclenche une NoSuchAlgorithmException si l'algorithme n'est pas fourni.

Paramètres :

<code>algorithm</code>	le nom de l'algorithme, comme "DES"

- `void init(SecureRandom random)`
- `void init(int keySize, SecureRandom random)`

Initialisent le générateur de clé.

Paramètres :

<code>keySize</code>	la taille de clé souhaitée
<code>random</code>	la source de hasard pour la génération des clés

- `SecretKey generateKey()`
- Génère une nouvelle clé.

618 Au cœur de Java 2 - Fonctions avancées

javax.crypto.SecretKeyFactory 1.4

- static SecretKeyFactory getInstance(String algorithm)
 - static SecretKeyFactory getInstance(String algorithm, String provider)
- Renvoient un objet SecretKeyFactory pour l'algorithme spécifié.

Paramètres :

algorithm	le nom de l'algorithme, comme "DES"
provider	le nom du fournisseur, comme "SunJCE" ou "BC"

- SecretKey generateSecret(KeySpec spec)
- Génère une nouvelle clé secrète à partir de la spécification donnée.

javax.crypto.spec.SecretKeySpec 1.4

- SecretKeySpec(byte[] key, String algorithm)
- Construit une spécification de clé.

Paramètres :

key	les octets à utiliser pour fabriquer la clé
algorithm	le nom de l'algorithme, comme "DES"

Flux de chiffres

La bibliothèque JCE contient un jeu cohérent de classes de flux qui cryptent ou décryptent automatiquement les données de flux. Voici, par exemple, comment crypter des données dans un fichier :

```
Cipher cipher = . . .;
cipher.init(Cipher.ENCRYPT_MODE, key);
CipherOutputStream out = new CipherOutputStream(
    new FileOutputStream(outputFileName), cipher);
byte[] bytes = new byte[BLOCKSIZE];
int inLength = getData(bytes); // récupérer les données de la source
while (inLength != -1)
{
    out.write(bytes, 0, inLength);
    inLength = getData(bytes); // récupérer d'autres données de la source
}
out.flush();
```

De même, vous pouvez utiliser un CipherInputStream pour lire et décrypter les données d'un fichier :

```
Cipher cipher = . . .;
cipher.init(Cipher.DECRYPT_MODE, key);
CipherInputStream in = new CipherInputStream(new
    FileInputStream(inputFileName), cipher);
byte[] bytes = new byte[BLOCKSIZE];
int inLength = in.read(bytes);
while (inLength != -1)
{
    putData(bytes, inLength); // placer les données dans la destination
    inLength = in.read(bytes);
}
```

Les classes de flux du chiffre gèrent, de manière transparente, les appels à update et à doFinal, ce qui est très pratique.

javax.crypto.CipherInputStream 1.4

■ CipherInputStream(InputStream in, Cipher cipher)

Construit un flux d'entrée qui lit des données depuis *in* et les décrypte ou les crypte en utilisant le chiffre donné.

■ int read()

■ int read(byte[] b, int off, int len)

Lisent les données depuis le flux d'entrée, qui est automatiquement décrypté ou crypté.

javax.crypto.CipherOutputStream 1.4

■ CipherOutputStream(OutputStream out, Cipher cipher)

Construit un flux de sortie qui écrit les données sur *out* et les crypte ou les décrypte à l'aide du chiffre donné.

■ void write(int ch)

■ void write(byte[] b, int off, int len)

Ecrivent des données sur le flux de sortie, qui est automatiquement crypté ou décrypté.

■ void flush()

Vide le tampon du chiffre et réalise un remplissage si nécessaire.

Chiffres de clés publiques

Le chiffre AES que vous avez vu dans la section précédente est un chiffre *symétrique*. La même clé est utilisée pour le cryptage et le décryptage. Le talon d'Achille de ces chiffres symétriques réside dans la distribution des clés. Si Alice envoie à Bob une méthode cryptée, Bob a besoin de la clé utilisée par Alice. Si Alice modifie la clé, elle doit envoyer à Bob à la fois le message et, par le biais d'un canal sécurisé, la nouvelle clé. Mais peut-être ne possède-t-elle pas de canal sécurisé pour accéder à Bob, et c'est la raison pour laquelle elle crypte les messages qu'elle lui envoie.

La cryptographie de clé publique permet de résoudre ce problème. Dans un chiffre de clé publique, Bob possède une paire de clés, constituée d'une clé publique et d'une clé privée correspondante. Il peut publier la clé publique n'importe où, mais doit surveiller de près la clé privée. Alice utilise simplement la clé publique pour crypter ses messages envoyés à Bob.

En fait, cela n'est pas aussi simple. Tous les algorithmes de clés publiques connus sont *bien plus* lents que les algorithmes de clé symétriques comme DES ou AES. Il ne serait ainsi pas pratique d'utiliser un algorithme de clé publique pour crypter de grandes quantités d'informations. Mais ce problème peut être facilement surmonté en combinant un chiffre de clé publique avec un chiffre symétrique rapide, comme ceci :

1. Alice génère une clé de cryptage symétrique aléatoire. Elle l'utilise pour crypter son texte brut.
2. Alice crypte la clé symétrique avec la clé publique de Bob.
3. Alice envoie à Bob la clé symétrique cryptée et le texte brut crypté.
4. Bob utilise sa clé privée pour décrypter la clé symétrique.
5. Bob utilise la clé symétrique décryptée pour décrypter le message.

620 Au cœur de Java 2 - Fonctions avancées

Personne, à l'exception de Bob, ne peut décrypter la clé symétrique car il est le seul à posséder la clé privée pour le décryptage. Ainsi, le cryptage onéreux de la clé publique n'est appliqué qu'à une petite quantité de données, celles de la clé.

L'algorithme de clé publique utilisé le plus souvent est l'algorithme RSA inventé par Rivest, Shamir et Adleman. Jusqu'au mois d'octobre 2000, l'algorithme était protégé par un brevet attribué à RSA Security Inc. Les licences n'étaient pas données, elles comprenaient généralement des royalties de 3 %, avec un paiement minimum de 50 000 \$ par an. Désormais, l'algorithme se trouve dans le domaine public. L'algorithme RSA est pris en charge par le JDK 1.5 et plus.

INFO

Si vous utilisez une ancienne version du JDK, consultez Legion of Bouncy Castle (<http://www.bouncycastle.org>). Il contient un fournisseur de cryptographie qui prend en charge RSA ainsi que plusieurs autres fonctions manquant dans le fournisseur SunJCE. Legion of Bouncy Castle a été signé par Sun Microsystems afin que vous puissiez l'associer avec le JDK.

Pour utiliser l'algorithme RSA, vous avez besoin d'une paire de clés publique/privée. Utilisez `KeyPairGenerator` comme suit :

```
KeyPairGenerator pairgen = KeyPairGenerator.getInstance("RSA");
SecureRandom random = new SecureRandom();
pairgen.initialize(KEYSIZE, random);
KeyPair keyPair = pairgen.generateKeyPair();
Key publicKey = keyPair.getPublic();
Key privateKey = keyPair.getPrivate();
```

Le programme de l'Exemple 7.22 possède trois options. L'option `-genkey` produit une paire de clés. L'option `-encrypt` génère une clé AES et l'enveloppe avec la clé publique.

```
Key key = . . . ; // une clé AES
Key publicKey = . . . ; // une clé publique RSA
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.WRAP_MODE, publicKey);
byte[] wrappedKey = cipher.wrap(key);
```

Il produit alors un fichier qui contient :

- la longueur de la clé emballée ;
- les octets de clé emballée ;
- le texte brut crypté avec la clé AES.

L'option `-decrypt` décrypte un fichier de ce type. Pour tester le programme, générez en premier les clés RSA :

```
java RSATest -genkey public.key private.key
```

Puis cryptez un fichier

```
java RSATest -encrypt plaintextFile encryptedFile public.key
```

Enfin, décryptez-le et vérifiez que le fichier décrypté correspond au texte brut.

```
java RSATest -decrypt encryptedFile decryptedFile private.key
```

Cet exemple nous amène à la fin de notre discussion sur la sécurité Java. Vous avez vu que la machine virtuelle et le gestionnaire de sécurité assurent une bonne sécurité pour les programmes, et vous avez vu comment utiliser la bibliothèque Java pour l'authentification et le décryptage. Plusieurs sujets avancés ou spécialisés n'auront pas été traités ici, notamment :

- GSS-API pour les "services de sécurité générique", qui assure la prise en charge du protocole Kerberos (et, en principe, d'autres protocoles pour l'échange sécurisé des messages). Le JDK propose un didacticiel à l'adresse <http://java.sun.com/j2se/5.0/docs/guide/security/jgss/tutorials/index.html>.
- La prise en charge de SASL (*Simple Authentication and Security Layer*), qui est utilisé par les protocoles LDAP et IMAP. Si vous devez implémenter SASL dans votre application, consultez l'adresse <http://java.sun.com/j2se/5.0/docs/guide/security/sasl/sasl-refguide.html>.
- La prise en charge de SSL (*Secure Sockets Layer*). SSL s'utilise en toute transparence sur HTTP pour les programmeurs d'applications. Utilisez simplement les URL qui commencent par http. Si vous souhaitez ajouter SSL à votre application, consultez la référence du JSEE (*Java Secure Socket Extension*) à l'adresse <http://java.sun.com/j2se/5.0/docs/guide/security/jsse/JSE-RefGuide.html>.

Exemple 7.22 : RSATest.java

```
import java.io.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

/**
 * Ce programme teste le chiffre RSA. Usage:
 * java RSATest -genkey public privé
 * java RSATest -encrypt public crypté en texte brut
 * java RSATest -decrypt privé crypté et décrypté
 */
public class RSATest
{
    public static void main(String[] args)
    {
        try
        {
            if (args[0].equals("-genkey"))
            {
                KeyPairGenerator pairgen
                    = KeyPairGenerator.getInstance("RSA");
                SecureRandom random = new SecureRandom();
                pairgen.initialize(KEYSIZE, random);
                KeyPair keyPair = pairgen.generateKeyPair();
                ObjectOutputStream out = new ObjectOutputStream(
                    new FileOutputStream(args[1]));
                out.writeObject(keyPair.getPublic());
                out.close();
                out = new ObjectOutputStream(
                    new FileOutputStream(args[2]));
                out.writeObject(keyPair.getPrivate());
                out.close();
            }
            else if (args[0].equals("-encrypt"))
            {
```

622 *Au cœur de Java 2 - Fonctions avancées*

```

{
    KeyGenerator keygen
        = KeyGenerator.getInstance("AES");
    SecureRandom random = new SecureRandom();
    keygen.init(random);
    SecretKey key = keygen.generateKey();

    // emballe avec la clé publique RSA
    ObjectInputStream keyIn = new ObjectInputStream(
        new FileInputStream(args[3]));
    Key publicKey = (Key) keyIn.readObject();
    keyIn.close();

    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.WRAP_MODE, publicKey);
    byte[] wrappedKey = cipher.wrap(key);
    DataOutputStream out = new DataOutputStream(new
        FileOutputStream(args[2]));
    out.writeInt(wrappedKey.length);
    out.write(wrappedKey);

    InputStream in = new FileInputStream(args[1]);
    cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.ENCRYPT_MODE, key);
    crypt(in, out, cipher);
    in.close();
    out.close();
}
else
{
    DataInputStream in = new DataInputStream(new
        FileInputStream(args[1]));
    int length = in.readInt();
    byte[] wrappedKey = new byte[length];
    in.read(wrappedKey, 0, length);

    // déballe avec la clé privée RSA
    ObjectInputStream keyIn = new ObjectInputStream(
        new FileInputStream(args[3]));
    Key privateKey = (Key) keyIn.readObject();
    keyIn.close();

    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.UNWRAP_MODE, privateKey);
    Key key = cipher.unwrap(wrappedKey, "AES",
        Cipher.SECRET_KEY);

    OutputStream out = new FileOutputStream(args[2]);
    cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.DECRYPT_MODE, key);

    crypt(in, out, cipher);
    in.close();
    out.close();
}
}

```

```

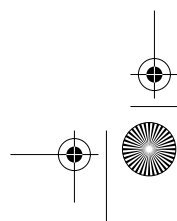
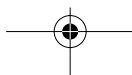
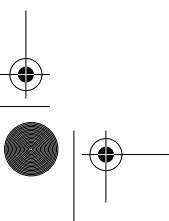
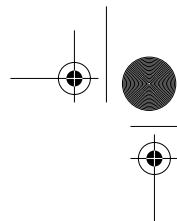
        catch (IOException e)
        {
            e.printStackTrace();
        }
        catch (GeneralSecurityException e)
        {
            e.printStackTrace();
        }
        catch (ClassNotFoundException e)
        {
            e.printStackTrace();
        }
    }

    /**
     * Utilise un chiffre pour transformer les octets en un flux
     * d'entrée et envoie les octets transformés à un flux de sortie.
     * @param in le flux d'entrée
     * @param out le flux de sortie
     * @param cipher le chiffre qui transforme les octets
     */
    public static void crypt(InputStream in, OutputStream out,
        Cipher cipher) throws IOException, GeneralSecurityException
    {
        int blockSize = cipher.getBlockSize();
        int outputSize = cipher.getOutputSize(blockSize);
        byte[] inBytes = new byte[blockSize];
        byte[] outBytes = new byte[outputSize];

        int inLength = 0;;
        boolean more = true;
        while (more)
        {
            inLength = in.read(inBytes);
            if (inLength == blockSize)
            {
                int outLength
                    = cipher.update(inBytes, 0, blockSize, outBytes);
                out.write(outBytes, 0, outLength);
            }
            else more = false;
        }
        if (inLength > 0)
            outBytes = cipher.doFinal(inBytes, 0, inLength);
        else
            outBytes = cipher.doFinal();
        out.write(outBytes);
    }

    private static final int KEYSIZE = 512;
}

```



8

Internationalisation

Au sommaire de ce chapitre

- ✓ Les paramètres régionaux
- ✓ Les formats des nombres
- ✓ La date et l'heure
- ✓ Classement
- ✓ La mise en forme des messages
- ✓ Les fichiers texte et les jeux de caractères
- ✓ Les groupes de ressources
- ✓ Un exemple complet

Ce monde est immense ; il faut espérer que quantité de ses habitants seront intéressés par votre logiciel. Internet, après tout, abat sans effort les barrières entre les pays. En revanche, si vous ne prenez pas garde à écrire votre programme pour un public international, vous établissez une barrière.

Le langage de programmation Java a été le premier langage conçu dès le départ pour la prise en charge de l'internationalisation. Dès le début, il possédait *LA* fonctionnalité essentielle nécessaire à une réelle internationalisation : il utilisait Unicode pour toutes les chaînes. Le système Unicode rend facile l'écriture en langage Java de programmes manipulant des chaînes dans n'importe laquelle des langues disponibles.

De nombreux programmeurs pensent que, pour rendre internationale leur application, il leur suffit de prendre en charge le système Unicode et de traduire les messages de l'interface utilisateur. Toutefois, comme vous le verrez dans ce chapitre, le support Unicode est loin d'être suffisant pour internationaliser les programmes. Les dates, les heures, les valeurs monétaires — et même les nombres — ont un format différent dans les diverses parties du monde. Vous devez pouvoir configurer aisément les noms de menus et de boutons, les chaînes des messages, ainsi que les raccourcis clavier pour les différentes langues.

626 *Au cœur de Java 2 - Fonctions avancées*

Dans ce chapitre, vous apprendrez à écrire des applications et des applets Java internationales. Vous verrez comment localiser la date et l'heure, les nombres et les textes, ainsi que les interfaces graphiques utilisateur, et vous étudierez les outils proposés par le JDK pour l'écriture de programmes internationaux. Nous terminerons ce chapitre par un exemple complet, une applet de calcul de la retraite qui affichera ses résultats différemment en fonction du pays où se trouvera la machine qui la téléchargera.

INFO

Pour en savoir plus sur l'internationalisation, consultez le site Web <http://www.joconner.com/javai18n> ainsi que le site Web officiel de Sun, <http://java.sun.com/j2se/corejava.intl>.

Les paramètres régionaux

Si vous examinez une application adaptée pour le marché international, la différence la plus évidente à vos yeux est la langue. Cette simple observation ne reflète pas vraiment une réelle internationalisation : certains pays peuvent avoir une langue en commun, mais vous aurez des efforts supplémentaires à faire pour satisfaire pleinement les utilisateurs d'ordinateurs des deux pays.¹

Dans tous les cas, les menus, les intitulés de boutons et les messages devront être traduits dans la langue locale ; ils peuvent également devoir être transcrits dans un alphabet différent. Il existe bien d'autres différences plus subtiles ; les nombres, par exemple, ont un format différent en anglais et en allemand. Le nombre (en anglais)

123,456.78

s'écrit

123.456,78

en allemand. En fait, les rôles du point et de la virgule décimale sont inversés ! Des variations similaires existent dans l'affichage des dates. Aux Etats-Unis, les dates sont assez bizarrement affichées sous la forme mois/jour/année. La France, par exemple, utilise l'ordre plus logique de jour/mois/année, alors qu'en Chine on écrit année/mois/jour. Par conséquent, la date (format américain)

3/22/61

se présentera sous la forme

22/03/1961

pour un Français. Bien entendu, si les noms de mois sont écrits explicitement, la différence entre les deux langues devient évidente. L'anglais

March 22, 1961

se présentera sous la forme

22 mars 1961

1. "Nous avons réellement tout en commun avec l'Amérique de nos jours, excepté bien sûr, la langue." (Oscar Wilde).

en français, ou

1961年3月22日

en chinois.

Il existe plusieurs classes de mise en forme qui prennent en compte ces différences. Pour contrôler la mise en forme, vous pouvez utiliser la classe `Locale`. Un paramètre régional (*locale*) décrit :

- une langue ;
- éventuellement, un emplacement géographique (*location*) ;
- facultativement, une variante.

Par exemple, aux Etats-Unis, vous utilisez un paramètre régional avec

`language=English, location=United States.`

En Allemagne, vous utiliserez

`language=German, location=Germany.`

La Suisse a quatre langues officielles (l'allemand, le français, l'italien et le romanche). Un germanophone en Suisse utilisera

`language=German, location=Switzerland`

Ce paramètre régional entraînera un formatage semblable à celui des paramètres régionaux pour l'Allemagne ; toutefois, les valeurs monétaires seront exprimées en francs suisses et non en euros.

Si vous ne précisez que la langue, comme :

`language=German`

le paramètre régional ne pourra pas être utilisé pour la devise, par exemple.

Les variantes sont heureusement rares et ne sont nécessaires que dans des situations exceptionnelles ou dépendantes du système. Par exemple, les Norvégiens ont bien du mal à se mettre d'accord sur l'orthographe de leur langue (un dérivé du danois). Ils utilisent deux jeux différents de règles orthographiques : l'un traditionnel, appelé Bokmål, et l'autre nouveau, appelé Nynorsk. L'orthographe traditionnelle s'exprimerait sous la forme d'une variante

`language=Norwegian, location=Norway, variant=Bokmål`

Pour exprimer la langue et l'emplacement géographique d'une manière concise et standardisée, le langage Java utilise des codes définis par la norme ISO (International Standards Organization). La langue locale est exprimée sous la forme d'un code de deux lettres minuscules, conforme à la norme ISO-639 et le code pays est exprimé sous la forme d'un code de deux lettres majuscules, conforme à la norme ISO-3166. Les Tableaux 8.1 et 8.2 montrent certains de ces codes les plus courants.

628 Au cœur de Java 2 - Fonctions avancées**INFO**

Pour obtenir la liste complète des codes ISO-639, consultez par exemple, le site <http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt>. La liste complète des codes ISO-3166 se trouve notamment sur le site <http://www.niso.org/3166.html>.

Tableau 8.1 : Codes ISO-639 les plus courants pour la langue

<i>Langue</i>	<i>Code</i>
Allemand	de
Anglais	en
Chinois	zh
Coréen	ko
Danois	da
Espagnol	sp
Français	fr
Finnois	fi
Grec	el
Hollandais	nl
Italien	it
Japonais	ja
Norvégien	no
Portugais	pt
Suédois	sv
Turc	tr

Tableau 8.2 : Codes ISO-3166 les plus courants pour le pays

<i>Pays</i>	<i>Code</i>
Allemagne	DE
Autriche	AT
Belgique	BE
Canada	CA
Chine	CN
Corée	KR

Tableau 8.2 : Codes ISO-3166 les plus courants pour le pays (*suite*)

<i>Pays</i>	<i>Code</i>
Danemark	DK
Espagne	ES
Etats-Unis	US
Finlande	FI
Grande-Bretagne	GB
Grèce	GR
Irlande	IE
Italie	IT
Japon	JP
Norvège	NO
Pays-Bas	NL
Portugal	PT
Suède	SE
Suisse	CH
Taiwan	TW
Turquie	TR

Ces codes peuvent sembler un peu aléatoires, surtout du fait que certains d'entre eux sont dérivés de langues locales (allemand = Deutsch = de, chinois = zhongwen = zh), mais, au moins, ils sont standardisés.

Pour décrire un paramétrage régional, vous concaténez la langue, le code pays et la variante éventuelle et passez cette chaîne au constructeur de la classe `Locale`.

```
Locale allemand = new Locale("de");
Locale allemandAllemagne = new Locale("de", "DE");
Locale allemandSuisse = new Locale("de", "CH");
Locale norvegienNorvegeBokmål = new Locale("no", "NO", "B");
```

Pour vous faciliter la tâche, le JDK prédéfinit un certain nombre de paramètres régionaux de pays :

```
Locale.CANADA
Locale.CANADA_FRENCH
Locale.CHINA
Locale.FRANCE
Locale.GERMANY
Locale.ITALY
Locale.JAPAN
```

630 Au cœur de Java 2 - Fonctions avancées

```
Locale.KOREA  
Locale.PRC  
Locale.TAIWAN  
Locale.UK  
Locale.US
```

Le JDK prédéfinit également un certain nombre de paramètres régionaux concernant la langue, sans emplacement géographique.

```
Locale.CHINESE  
Locale.ENGLISH  
Locale.FRENCH  
Locale.GERMAN  
Locale.ITALIAN  
Locale.JAPANESE  
Locale.KOREAN  
Locale.SIMPLIFIED_CHINESE  
Locale.TRADITIONAL_CHINESE
```

En plus de la possibilité de construire un paramétrage régional ou d'en utiliser un prédéfini, vous disposez de deux autres méthodes pour obtenir un objet `Locale`.

La méthode statique `getDefault` de la classe `Locale` extrait initialement les paramètres régionaux par défaut stockés dans le système d'exploitation. Vous pouvez modifier le paramètre régional de Java en appelant `setDefault` ; toutefois, ce changement n'affecte que votre programme, et non le système d'exploitation. De même, dans une applet, la méthode `getLocale` renvoie les paramètres régionaux de l'utilisateur qui affiche l'applet. Enfin, toutes les classes utilitaires dépendant des paramètres régionaux peuvent renvoyer un tableau des paramètres régionaux qu'elles gèrent. Par exemple,

```
Locale[] supportedLocales = DateFormat.getAvailableLocales();
```

renvoie toutes les classes que la classe `DateFormat` peut gérer.

ASTUCE

A des fins de test, vous pourrez basculer le paramètre régional par défaut de votre programme. Indiquez les propriétés de langue et de région lorsque vous lancez le programme. Ici, par exemple, nous définissons le paramètre par défaut sur Allemand (Suisse) :

```
java -Duser.language=de -Duser.region=CH programme
```

Lorsque vous avez vos paramètres régionaux, que pouvez-vous faire avec ? En réalité, pas grand-chose. Les seules méthodes utiles dans la classe `Locale` sont celles permettant d'identifier les codes de langue et de pays. La plus importante est la méthode `getDisplayName`. Elle renvoie une chaîne décrivant les paramètres régionaux. Cette chaîne ne contient pas les codes ISO sur deux lettres, mais elle se présente sous une forme compréhensible par l'utilisateur, comme par exemple :

```
German (Switzerland)
```

En réalité, un problème se pose. Le nom d'affichage est édité en fonction des paramètres régionaux par défaut (ci-dessus anglais (Etats-Unis)). Cela peut ne pas être approprié. Si votre utilisateur a déjà sélectionné l'allemand comme préférence de langue, vous voudrez probablement présenter la chaîne en langue allemande. Cela se fait en passant les paramètres régionaux allemands en paramètre.

Le code

```
Locale loc = new Locale("de", "CH");
System.out.println(loc.getDisplayName(Locale.GERMAN));
```

imprimera

```
Deutsch (Schweiz)
```

Cet exemple vous montre l'utilité des objets `Locale`. Vous le fournissez aux méthodes averties des paramètres régionaux qui produisent du texte pour les utilisateurs de diverses régions. Vous en verrez de nombreux exemples dans les sections à venir.

java.util.Locale 1.1

- `Locale(String language)`
- `Locale(String language, String country)`
- `Locale(String language, String country, String variant)`
Construisent un paramètre régional avec la langue donnée, le pays et la variante.
- `static Locale getDefault()`
Renvoie les paramètres régionaux par défaut.
- `static void setDefault(Locale loc)`
Définit les paramètres régionaux par défaut.
- `String getDisplayName()`
Renvoie un nom décrivant les paramètres régionaux, exprimé dans la langue locale courante.
- `String getDisplayName(Locale loc)`
Renvoie un nom décrivant les paramètres régionaux, exprimé dans la langue locale donnée.
- `String getLanguage()`
Renvoie le code langue, sous la forme du code ISO-639 en minuscules sur deux lettres.
- `String getDisplayLanguage()`
Renvoie le nom de la langue, exprimé dans la langue locale courante.
- `String getDisplayLanguage(Locale loc)`
Renvoie le nom de la langue, exprimé dans la langue locale donnée.
- `String getCountry()`
Renvoie le code pays, sous la forme du code ISO-3166 en majuscules sur deux lettres.
- `String getDisplayCountry()`
Renvoie le nom du pays, exprimé dans la langue locale courante.
- `String getDisplayCountry(Locale loc)`
Renvoie le nom du pays, exprimé dans la langue locale donnée.
- `String getVariant()`
Renvoie la chaîne variante.
- `String getDisplayVariant()`
Renvoie le nom de la variante, exprimé dans la langue locale courante.
- `String getDisplayVariant(Locale loc)`
Renvoie le nom de la variante, exprimé dans la langue locale donnée.

632 Au cœur de Java 2 - Fonctions avancées

- `String toString()`
Renvoie une description des paramètres régionaux, avec la langue, le pays et la variante, séparés par des traits de soulignement (par ex. "de_CH").
- `Locale getLocale() [1.1]`
Récupère le paramètre régional pour cette applet.

Le format des nombres

Nous avons déjà mentionné comment le formatage des nombres et des monnaies dépendait fortement des paramètres régionaux. Le langage Java fournit une collection d'objets `formatter` qui peuvent analyser et mettre en forme des valeurs numériques dans la classe `java.text`. Vous réalisez les étapes suivantes pour formater un nombre en fonction d'un paramétrage régional particulier :

1. Récupération de l'objet paramètres régionaux (`Locale`), comme décrit dans la section précédente.
2. Utilisation d'une "méthode *factory*" pour obtenir un objet *formateur*.
3. Utilisation de l'objet formateur pour l'analyse et la mise en forme.

Les méthodes *factory* sont des méthodes statiques de la classe `NumberFormat` qui acceptent un argument `Locale`. Il y a trois méthodes *factory* : `getNumberInstance`, `getCurrencyInstance` et `getPercentInstance`. Ces méthodes renvoient des objets qui peuvent formater et analyser des nombres, des valeurs monétaires et des pourcentages, respectivement. Voici, par exemple, comment vous pouvez formater une valeur monétaire en allemand.

```
Locale loc = new Locale("de", "DE");
NumberFormat currFmt = NumberFormat.getCurrencyInstance(loc);
double amt = 123456.78;
System.out.println(currFmt.format(amt));
```

Ce code imprime :

```
123.456,78 DM
```

Le symbole monétaire DM est placé à la fin de la chaîne. Remarquez aussi que l'utilisation des points et virgules décimales est inversée par rapport au format américain.

Inversement, pour lire un nombre qui a été entré ou stocké avec les conventions d'un certain paramétrage régional, utilisez la méthode `parse`. Par exemple, le code suivant analyse la valeur entrée par l'utilisateur dans un champ de texte. La méthode `parse` peut traiter les points et virgules décimales, ainsi que les chiffres dans d'autres langues.

```
TextField inputField;
...
NumberFormat fmt = NumberFormat.getNumberInstance();
// obtenir le formateur de nombre pour les
// paramètres régionaux par défaut
Number input = fmt.parse(inputField.getText().trim());
double x = input.doubleValue();
```

Le type renvoyé par `parse` est le type abstrait `Number`. L'objet renvoyé est un objet *wrapper* (enveloppe) `Double` ou `Long`, selon que le nombre analysé était ou non en virgule flottante. Si la distinction

vous importe peu, vous pouvez utiliser simplement la méthode `doubleValue` de la classe `Number` pour extraire le nombre enveloppé.

ATTENTION

Les objets du type `Number` ne sont pas automatiquement unboxed : vous ne pouvez pas simplement affecter un objet `Number` à un type primitif. Utilisez plutôt la méthode `doubleValue` ou `intValue`.

Si le texte du nombre n'a pas la forme correcte, la méthode lance une `ParseException`. Par exemple, les espaces en début de chaîne ne sont pas autorisés. (Appelez `trim` pour les supprimer). En revanche, tous les caractères suivant le nombre sont simplement ignorés et ne déclenchent pas d'exception.

Notez que les classes renvoyées par les méthodes factory `getXxxInstance` ne sont pas en réalité du type `NumberFormat`. Le type `NumberFormat` est une classe abstraite et les formateurs réels appartiennent à l'une de ses sous-classes. Les méthodes factory savent simplement comment localiser l'objet qui appartient à un paramétrage régional particulier.

Vous pouvez obtenir une liste des paramétrages régionaux pris en charge à l'aide de la méthode statique `getAvailableLocales`. Cette méthode renvoie un tableau des paramètres régionaux pour lesquels des objets formateurs de nombres peuvent être obtenus.

L'exemple de programme de cette section vous permet d'expérimenter des formateurs de nombres (voir Figure 8.1). La zone de liste modifiable en haut contient tous les paramétrages régionaux disposant de formateurs de nombres. Vous pouvez choisir entre des formateurs de nombre, de monnaie et de pourcentage. A chaque nouveau choix, le nombre dans le champ de texte est reformaté. Si vous expérimentez plusieurs paramétrages régionaux, vous aurez une bonne idée du nombre de formatages différents possibles pour les nombres et les valeurs monétaires. Vous pouvez aussi entrer un nombre différent et cliquer sur le bouton `Analyse` pour appeler la méthode `parse`, qui tente d'analyser ce que vous entrez. Si votre entrée est analysée avec succès, elle est passée à format et le résultat est affiché. Si l'analyse échoue, un message "Erreur d'analyse" est affiché dans le champ de texte.

Figure 8.1

Le programme
`NumberFormatTest`.



Le code, dans l'Exemple 8.1, est assez simple. Dans le constructeur, nous appelons `NumberFormat.getAvailableLocales`. Pour chaque paramétrage régional, nous appelons la méthode `getDisplayName` et remplissons une zone de liste modifiable avec les chaînes renvoyées par cette méthode (les chaînes ne sont pas triées ; nous résoudrons ce problème avec une liste personnalisée plus loin dans ce chapitre). Chaque fois que l'utilisateur sélectionne un nouveau paramétrage régional ou qu'il clique sur l'un des boutons radio, nous créons un nouvel objet `formatter` et mettons à jour le champ de texte. Lorsque l'utilisateur clique sur le bouton `Analyse`, la méthode `parse` qui réalise l'analyse est appelée, en fonction du paramétrage régional choisi.

634 *Au cœur de Java 2 - Fonctions avancées*

Exemple 8.1 : *NumberFormatTest.java*

```
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;
import javax.swing.*;

/**
 * Ce programme montre la mise en forme des nombres
 * en fonction de divers paramétrages régionaux.
 */
public class NumberFormatTest
{
    public static void main(String[] args)
    {
        JFrame frame = new NumberFormatFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc contient des boutons permettant de
 * sélectionner un format de nombre, une zone de liste
 * modifiable pour choisir un paramètre régional, un
 * champ de texte pour afficher un numéro mis en forme
 * ainsi qu'un bouton pour analyser le contenu du champ
 * de texte.
 */
class NumberFormatFrame extends JFrame
{
    public NumberFormatFrame()
    {
        setTitle("NumberFormatTest");
        setLayout(new GridBagLayout());

        ActionListener listener = new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                { updateDisplay(); }
            };

        JPanel p = new JPanel();
        addRadioButton(p, numberRadioButton, rbGroup, listener);
        addRadioButton(p, currencyRadioButton, rbGroup, listener);
        addRadioButton(p, percentRadioButton, rbGroup, listener);

        add(new JLabel("Param. Rég."), new GBC(0, 0).setAnchor(GBC.EAST));
        add(p, new GBC(1, 1));
        add(parseButton, new GBC(0, 2).setInsets(2));
        add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
        add(numberText, new GBC(1, 2).setFill(GBC.HORIZONTAL));
    }
}
```

```

        locales = NumberFormat.getAvailableLocales();
        for (Locale loc : locales)
            localeCombo.addItem(loc.getDisplayName());
        localeCombo.setSelectedItem(
            Locale.getDefault().getDisplayName());
        currentNumber = 123456.78;
        updateDisplay();

        localeCombo.addActionListener(listener);

        parseButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    String s = numberText.getText().trim();
                    try
                    {
                        Number n = currentNumberFormat.parse(s);
                        if (n != null)
                        {
                            currentNumber = n.doubleValue();
                            updateDisplay();
                        }
                        else
                        {
                            numberText.setText("Erreur analyse : " + s);
                        }
                    }
                    catch (ParseException e)
                    {
                        numberText.setText("Erreur analyse : " + s);
                    }
                }
            });
        pack();
    }

    /**
     * Ajoute un bouton à un conteneur.
     * @param p le conteneur dans lequel placer le bouton
     * @param b le bouton
     * @param g le groupe de boutons
     * @param listener l'écouteur de boutons
     */
    public void addRadioButton(Container p, JRadioButton b,
        ButtonGroup g, ActionListener listener)
    {
        b.setSelected(g.getButtonCount() == 0);
        b.addActionListener(listener);
        g.add(b);
        p.add(b);
    }

    /**
     * Actualise l'affichage et formate le nombre
     * en fonction des réglages de l'utilisateur.
     */
    public void updateDisplay()

```

636 Au cœur de Java 2 - Fonctions avancées

```

    {
        Locale currentLocale = locales[
            localeCombo.getSelectedIndex()];
        currentNumberFormat = null;
        if (numberRadioButton.isSelected())
            currentNumberFormat
                = NumberFormat.getNumberInstance(currentLocale);
        else if (currencyRadioButton.isSelected())
            currentNumberFormat
                = NumberFormat.getCurrencyInstance(currentLocale);
        else if (percentRadioButton.isSelected())
            currentNumberFormat
                = NumberFormat.getPercentInstance(currentLocale);
        String n = currentNumberFormat.format(currentNumber);
        numberText.setText(n);
    }

    private Locale[] locales;

    private double currentNumber;

    private JComboBox localeCombo = new JComboBox();
    private JButton parseButton = new JButton("Analyse");
    private JTextField numberText = new JTextField(30);
    private JRadioButton numberRadioButton = new JRadioButton("Nombre");
    private JRadioButton currencyRadioButton = new
        JRadioButton("Monnaie");
    private JRadioButton percentRadioButton = new
        JRadioButton("Pourcentage");
    private ButtonGroup rbGroup = new ButtonGroup();
    private NumberFormat currentNumberFormat;
}

```

java.text.NumberFormat 1.1

- `static Locale[] getAvailableLocales()`
Renvoie un tableau d'objets `Locale` pour lesquels des formateurs `NumberFormat` sont disponibles.
- `static NumberFormat getNumberInstance()`
- `static NumberFormat getNumberInstance(Locale l)`
- `static NumberFormat getCurrencyInstance()`
- `static NumberFormat getCurrencyInstance(Locale l)`
- `static NumberFormat getPercentInstance()`
- `static NumberFormat getPercentInstance(Locale l)`
Renvoient un formateur pour les nombres, les valeurs monétaires ou de pourcentage, pour le paramétrage régional actuel ou le paramétrage donné.
- `String format(double x)`
- `String format(long x)`
Renvoient la chaîne résultant du formatage du nombre entier ou en virgule flottante.
- `Number parse(String s)`
Analyse la chaîne donnée et renvoie la valeur numérique, sous la forme d'un `Double` si la chaîne en entrée décrivait un nombre en virgule flottante et sous la forme `Long` sinon. Le début de la

chaîne doit contenir un nombre, sans espaces avant. Le nombre peut être suivi d'autres caractères qui seront ignorés. Une `ParseException` est lancée en cas d'échec de l'analyse.

- `void setParseIntegerOnly(boolean b)`
- `boolean isParseIntegerOnly()`

Définissent ou extraient (set/get) un indicateur signalant si ce formateur ne doit analyser que des valeurs entières.

- `void setGroupingUsed(boolean b)`
- `boolean isGroupingUsed()`

Définissent ou extraient un indicateur signalant si ce formateur émet et reconnaît les séparateurs (par ex. `100,000`).

- `void setMinimumIntegerDigits(int n)`
- `int getMinimumIntegerDigits()`
- `void setMaximumIntegerDigits(int n)`
- `int getMaximumIntegerDigits()`
- `void setMinimumFractionDigits(int n)`
- `int getMinimumFractionDigits()`
- `void setMaximumFractionDigits(int n)`
- `int getMaximumFractionDigits()`

Définissent ou extraient le nombre maximal ou minimal de chiffres autorisés dans la partie entière ou fractionnaire d'un nombre.

Devises

Pour mettre en forme une valeur de devise, vous pouvez utiliser la méthode `NumberFormat.getCurrencyInstance`. Elle n'est toutefois pas très flexible, car elle renvoie un élément de mise en forme pour une seule devise. Or supposons que vous prépariez une facture pour un client américain dans laquelle certaines sommes sont en dollars et d'autres, en euros. Vous ne pouvez pas utiliser les deux mises en forme :

```
NumberFormat dollarFormatter =  
    NumberFormat.getCurrencyInstance(Locale.US);  
NumberFormat euroFormatter =  
    NumberFormat.getCurrencyInstance(Locale.GERMANY);
```

Votre facture aurait un air étrange, certaines valeurs étant présentées sous la forme \$100,000 et d'autres, 100.000 €.

Utilisez plutôt la classe `Currency` pour contrôler la devise utilisée par les formateurs. Pour ce faire, passez un identifiant de devise à la méthode statique `Currency.getInstance`. Appelez ensuite la méthode `setCurrency` pour chaque formateur. Voici comment vous pourriez configurer le formateur Euro pour votre client américain :

```
NumberFormat euroFormatter = NumberFormat.getCurrencyInstance(Locale.US);  
euroFormatter.setCurrency(Currency.getInstance("EUR"));
```

638 Au cœur de Java 2 - Fonctions avancées

Les identifiants de devises sont définis par la norme ISO 4217 (voir l'adresse http://en.wikipedia.org/wiki/ISO_4217 pour en connaître la liste). Le Tableau 8.3 en propose une liste partielle.

Tableau 8.3 : Identifiants des principales devises

Dollar américain	USD
Euro	EUR
Livre britannique	GBP
Yen japonais	JPY
Yuan chinois	CNY
Roupie indienne	INR
Rouble russe	RUB

java.util.Currency 1.4

- `static Currency getInstance(String currencyCode)`
- `static Currency getInstance(Locale locale)`
Renvoient l'instance `Currency` pour le code de devise ISO 4217 donné ou le pays du paramètre donné.
- `String toString()`
- `String getCurrencyCode()`
Récupèrent le code de devise ISO 4217 de cette devise.
- `String getSymbol()`
- `String getSymbol(Locale locale)`
Récupèrent le symbole de mise en forme de cette devise pour le paramètre par défaut ou le paramètre régional donné. Par exemple, le symbole pour USD pourrait être "\$" ou "US\$", selon le paramètre régional.
- `int getDefaultFractionDigits()`
Récupère le nombre par défaut de chiffres après la virgule pour cette devise.

La date et l'heure

Lorsque vous formatez une date ou une heure, vous devez tenir compte de quatre éléments dépendant des paramètres régionaux :

- Les noms des mois et des jours doivent être écrits dans la langue locale.
- Il existe des préférences locales en ce qui concerne l'ordre année/mois/jour.
- Le calendrier grégorien peut ne pas être celui pris en compte localement pour exprimer les dates.
- Le fuseau horaire du lieu géographique doit être pris en compte.

La classe Java `DateFormat` gère ces éléments. Elle est facile d'utilisation et assez semblable à la classe `NumberFormat`. Tout d'abord, vous récupérez les paramètres régionaux. Vous pouvez utiliser

le paramétrage régional par défaut ou appeler la méthode statique `getAvailableLocales` pour obtenir un tableau des paramètres régionaux qui prennent en charge le formatage de date. Vous appelez ensuite l'une des trois méthodes factory :

```
fmt = DateFormat.getDateInstance(dateStyle, loc);
fmt = DateFormat.getTimeInstance(timeStyle, loc);
fmt = DateFormat.getDateTimeInstance(dateStyle, timeStyle, loc);
```

Pour spécifier le style désiré, ces méthodes factory ont un paramètre qui est l'une des constantes suivantes :

- `DateFormat.DEFAULT`
- `DateFormat.FULL` (par exemple Wednesday, September 15, 2004 8:51:03 P.M. PDT pour le paramètre régional U.S.)
- `DateFormat.LONG` (par exemple September 15, 2004 8:51:03 P.M. PDT pour le paramétrage régional U.S.)
- `DateFormat.MEDIUM` (par exemple Sep 15, 2004 8:51:03 P.M. pour le paramétrage régional U.S.)
- `DateFormat.SHORT` (par exemple 9/15/04 8:51 P.M. pour le paramétrage régional U.S.)

La méthode factory renvoie un objet format que vous pouvez utiliser pour les dates.

```
Date now = new Date();
String s = fmt.format(now);
```

Tout comme pour la classe `NumberFormat`, vous pouvez utiliser la méthode `parse` pour analyser une date entrée par l'utilisateur. Par exemple, le code suivant analyse la valeur entrée par l'utilisateur dans un champ de texte, en utilisant les paramètres régionaux par défaut.

```
TextField inputField;
...
DateFormat fmt = DateFormat.getDateInstance(DateFormat.MEDIUM);
Date input = fmt.parse(inputField.getText().trim());
```

Si le nombre n'a pas été entré correctement, ce code lance une `ParseException`. Ici non plus, les espaces en début de chaîne ne sont pas autorisés. Vous pouvez les supprimer en appelant `trim`. Tout caractère suivant le nombre dans la chaîne sera ignoré. Malheureusement, l'utilisateur doit entrer la date exactement dans le format attendu. Par exemple, si le format est défini à `MEDIUM` pour le paramétrage régional U.S., les dates doivent se présenter ainsi :

Sep 18, 1997

Si l'utilisateur tape

Sep 18 1997

(sans la virgule), ou le format court,

9/18/97

il en résultera une erreur d'analyse.

Un indicateur indulgent (`lenient`) interprète les dates avec clémence. Par exemple, le 30 février 1999 sera automatiquement converti en 2 mars 1999. Cela peut paraître dangereux, mais c'est le comportement par défaut. Il est préférable de désactiver cette fonctionnalité. L'objet `calendar`

640 Au cœur de Java 2 - Fonctions avancées

(calendar), qui interprète la date analysée, lancera une `IllegalArgumentException` si l'utilisateur entre une combinaison de jour/mois/année erronée.

L'Exemple 8.2 montre la classe `DateFormat` en action. Vous pouvez sélectionner un paramétrage régional et voir comment la date et l'heure sont formatées en différents endroits du monde. Si vous obtenez des caractères point d'interrogation en sortie, cela signifie que les polices correspondant à la langue du paramétrage régional ne sont pas installées. Par exemple, si vous choisissez un paramétrage régional chinois, la date peut s'afficher sous la forme

1961年3月22日

La Figure 8.2 montre le programme (après installation des polices chinoises). Comme vous le voyez, la sortie s'affiche correctement.

Vous pouvez aussi expérimenter l'analyse. Entrez une date ou une heure, cochez la case *Analyse indulgente* si désiré, et cliquez sur le bouton *Analyse date* ou *Analyse heure*.

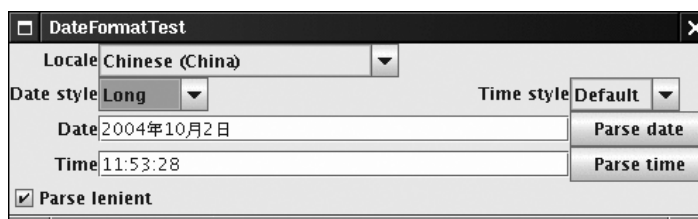
Nous utilisons une classe d'assistance, `EnumCombo`, pour résoudre un problème technique (voir Exemple 8.3). Nous voulions remplir une zone de liste modifiable avec des valeurs comme `Short`, `Medium` et `Long`, puis convertir automatiquement la sélection de l'utilisateur en valeurs entières `DateFormat.SHORT`, `DateFormat.MEDIUM` et `DateFormat.LONG`. Au lieu d'écrire un code répétitif, nous utilisons la réflexion : nous avons converti le choix de l'utilisateur en majuscules, remplacé tous les espaces par des caractères de soulignement (`_`), puis trouvé la valeur du champ statique avec ce nom. (Reportez-vous au Chapitre 5 du Volume 1 pour plus d'informations sur la réflexion.)

ASTUCE

Pour calculer les heures dans différents fuseaux horaires, utilisez la classe `TimeZone`. Consultez l'adresse [http://java.sun.com/ developer/JDCTechTips/2003/tt1104.html#2](http://java.sun.com/developer/JDCTechTips/2003/tt1104.html#2) pour obtenir un petit didacticiel.

Figure 8.2

Le programme
DateFormatTest.



Exemple 8.2 : *DateFormatTest.java*

```
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;
import javax.swing.*;
```

```

/**
 * Ce programme montre la mise en forme des dates
 * selon divers paramètres régionaux.
 */
public class DateFormatTest
{
    public static void main(String[] args)
    {
        JFrame frame = new DateFormatFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc contient les listes modifiables pour choisir
 * un paramétrage local, une date et des formats
 * d'heure, des champs de texte pour afficher la date et
 * l'heure mises en forme pour analyser le contenu du
 * champ de texte ainsi qu'une case à cocher indulgente.
 */
class DateFormatFrame extends JFrame
{
    public DateFormatFrame()
    {
        setTitle("DateFormatTest");

        setLayout(new GridBagLayout());
        add(new JLabel("Param. rég."), new GBC(0, 0).setAnchor(GBC.EAST));
        add(new JLabel("Style date"), new GBC(0, 1).setAnchor(GBC.EAST));
        add(new JLabel("Style heure"), new GBC(2, 1).setAnchor(GBC.EAST));
        add(new JLabel("Date"), new GBC(0, 2).setAnchor(GBC.EAST));
        add(new JLabel("Heure"), new GBC(0, 3).setAnchor(GBC.EAST));
        add(localeCombo, new GBC(1, 0, 2, 1).setAnchor(GBC.WEST));
        add(dateStyleCombo, new GBC(1, 1).setAnchor(GBC.WEST));
        add(timeStyleCombo, new GBC(3, 1).setAnchor(GBC.WEST));
        add(dateParseButton, new GBC(3, 2).setAnchor(GBC.WEST));
        add(timeParseButton, new GBC(3, 3).setAnchor(GBC.WEST));
        add(lenientCheckbox, new GBC(0, 4, 2, 1).setAnchor(GBC.WEST));
        add(dateText, new GBC(1, 2, 2, 1).setFill(GBC.HORIZONTAL));
        add(timeText, new GBC(1, 3, 2, 1).setFill(GBC.HORIZONTAL));

        locales = DateFormat.getAvailableLocales();
        for (Locale loc : locales)
            localeCombo.addItem(loc.getDisplayName());
        localeCombo.setSelectedItem(
            Locale.getDefault().getDisplayName());
        currentDate = new Date();
        currentTime = new Date();
        updateDisplay();

        ActionListener listener = new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    updateDisplay();
                }
            };
    }
}

```


642 *Au cœur de Java 2 - Fonctions avancées*

```

localeCombo.addActionListener(listener);
dateStyleCombo.addActionListener(listener);
timeStyleCombo.addActionListener(listener);

dateParseButton.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            String d = dateText.getText().trim();
            try
            {
                currentDateFormat.setLenient
                    (lenientCheckbox.isSelected());
                Date date = currentDateFormat.parse(d);
                currentDate = date;
                updateDisplay();
            }
            catch (ParseException e)
            {
                dateText.setText("Erreur analyse : " + d);
            }
            catch (IllegalArgumentException e)
            {
                dateText.setText("Erreur argument : " + d);
            }
        }
    });

timeParseButton.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            String t = timeText.getText().trim();
            try
            {
                currentDateFormat.setLenient
                    (lenientCheckbox.isSelected());
                Date date = currentTimeFormat.parse(t);
                currentTime = date;
                updateDisplay();
            }
            catch (ParseException e)
            {
                timeText.setText("Erreur analyse : " + t);
            }
            catch (IllegalArgumentException e)
            {
                timeText.setText("Erreur argument : " + t);
            }
        }
    });
pack();
}

/**
Met à jour l'affichage et formate la date selon
les réglages de l'utilisateur.

```

```

*/
public void updateDisplay()
{
    Locale currentLocale = locales[
        localeCombo.getSelectedIndex()];
    int dateStyle = dateStyleCombo.getValue();
    currentDateFormat
        = DateFormat.getDateInstance(dateStyle,
            currentLocale);
    String d = currentDateFormat.format(currentDate);
    dateText.setText(d);
    int timeStyle = timeStyleCombo.getValue();
    currentTimeFormat
        = DateFormat.getTimeInstance(timeStyle,
            currentLocale);
    String t = currentTimeFormat.format(currentTime);
    timeText.setText(t);
}

private Locale[] locales;

private Date currentDate;
private Date currentTime;
private DateFormat currentDateFormat;
private DateFormat currentTimeFormat;

private JComboBox localeCombo = new JComboBox();
private EnumCombo dateStyleCombo
    = new EnumCombo(DateFormat.class,
        new String[] { "Default", "Full", "Long",
            "Medium", "Short" });
private EnumCombo timeStyleCombo
    = new EnumCombo(DateFormat.class,
        new String[] { "Default", "Full", "Long",
            "Medium", "Short" });
private JButton dateParseButton = new JButton("Analyse date");
private JButton timeParseButton = new JButton("Analyse heure");
private JTextField dateText = new JTextField(30);
private JTextField timeText = new JTextField(30);
private JTextField parseText = new JTextField(30);
private JCheckBox lenientCheckbox
    = new JCheckBox("Analyse indulgente", true);
}

```

Exemple 8.3 : EnumCombo.java

```

import java.util.*;
import javax.swing.*;

/**
 * Une liste modifiable permettant aux utilisateurs de
 * choisir parmi plusieurs valeurs de champ statiques
 * dont les noms sont fournis dans le constructeur.
 */
public class EnumCombo extends JComboBox
{
    /**
     * Construit un EnumCombo.
     * @param cl une classe
     */
}

```

644 Au cœur de Java 2 - Fonctions avancées

```

        @param labels un tableau de noms de champs
        statiques de cl
    */
    public EnumCombo(Class cl, String[] labels)
    {
        for (int i = 0; i < labels.length; i++)
        {
            String label = labels[i];
            String name = label.toUpperCase().replace(' ', '_');
            int value = 0;
            try
            {
                java.lang.reflect.Field f = cl.getField(name);
                value = f.getInt(cl);
            }
            catch (Exception e)
            {
                label = "(" + label + ")";
            }
            table.put(label, value);
            addItem(label);
        }
        setSelectedItem(labels[0]);
    }

    /**
     Renvoie la valeur du champ sélectionné par
     l'utilisateur.
     @return la valeur de champ statique
    */
    public int getValue()
    {
        return table.get(getSelectedItem());
    }

    private Map<String, Integer> table = new TreeMap<String, Integer>();
}

```

java.text.DateFormat 1.1

- static Locale[] getAvailableLocales()
Renvoie un tableau d'objets Locale pour lesquels des formateurs DateFormat sont disponibles.
 - static DateFormat getDateInstance(int dateStyle)
 - static DateFormat getDateInstance(int dateStyle, Locale l)
 - static DateFormat getTimeInstance(int timeStyle)
 - static DateFormat getTimeInstance(int timeStyle, Locale l)
 - static DateFormat getDateTimeInstance(int dateStyle, int timeStyle)
 - static DateFormat getDateTimeInstance(int dateStyle, int timeStyle, Locale l)
Renvoient un formateur pour la date, l'heure, ou la date et l'heure, pour le paramétrage régional par défaut ou celui donné.
- Paramètres :* dateStyle, timeStyle l'un des styles DEFAULT, FULL, LONG, MEDIUM, SHORT
- String format(Date d)
Renvoie la chaîne résultant du formatage de la date/l'heure donnée.

- `Date parse(String s)`
Analyse la chaîne donnée et renvoie la date/l'heure qu'elle décrit. La chaîne doit contenir une date ou une heure, sans espaces avant. La date peut être suivie d'autres caractères, qui seront ignorés. Une `ParseException` est lancée en cas d'échec de l'analyse.
- `void setLenient(boolean b)`
- `boolean isLenient()`
Définissent ou extraient un indicateur signalant si l'analyse doit être rigoureuse ou indulgente. En mode lenient (indulgent), des dates comme 30 février 1999 seront automatiquement converties en 2 mars 1999. Par défaut, l'analyse est indulgente.
- `void setCalendar(Calendar cal)`
- `Calendar getCalendar()`
Définissent ou extraient l'objet `Calendar` utilisé pour l'extraction des informations année, mois, jour, heure, minute et seconde de l'objet `Date`. Utilisez cette méthode si vous ne voulez pas utiliser le calendrier par défaut pour les paramètres régionaux (habituellement le calendrier grégorien).
- `void setTimeZone(TimeZone tz)`
- `TimeZone getTimeZone()`
Définissent ou extraient l'objet `TimeZone` (*fuseau horaire*) utilisé pour le formatage de l'heure. Utilisez cette méthode si vous ne voulez pas utiliser le fuseau horaire par défaut pour les paramètres régionaux. Le fuseau horaire par défaut est celui du paramétrage régional par défaut, obtenu à partir du système d'exploitation. Pour les autres paramètres régionaux, il s'agit du fuseau horaire correspondant à l'emplacement géographique (*location*).
- `void setNumberFormat(NumberFormat f)`
- `NumberFormat getNumberFormat()`
Définissent ou extraient le format utilisé pour le formatage des nombres, pour la représentation des informations année, mois, jour, heure, minute et seconde.
- `static String[] getAvailableIDs()`
Récupère tous les ID de fuseaux horaires pris en charge.
- `static TimeZone getDefault()`
Récupère le `TimeZone` par défaut pour cet ordinateur.
- `static TimeZone getTimeZone(String timeZoneId)`
Récupère le `TimeZone` pour l'ID donné.
- `String getID()`
Récupère l'ID de ce fuseau horaire.
- `String getDisplayName()`
- `String getDisplayName(Locale locale)`
- `String getDisplayName(boolean daylight, int style)`
- `String getDisplayName(boolean daylight, int style, Locale locale)`
Récupèrent le nom d'affichage de ce fuseau dans le paramètre régional par défaut ou dans le paramètre donné. Si le paramètre `daylight` vaut `true`, renvoie le nom de l'heure d'été. Le paramètre `style` peut être `SHORT` ou `LONG`.

646 Au cœur de Java 2 - Fonctions avancées

- `boolean useDaylightTime()`
Renvoie `true` si ce `TimeZone` utilise l'heure d'été.
- `boolean inDaylightTime(Date date)`
Renvoie `true` si la date donnée est en heure d'été dans ce `TimeZone`.

Classement

Le tri de chaînes par ordre alphabétique est simple lorsque les chaînes ne sont constituées que de caractères ASCII anglais. Vous comparez simplement les chaînes à l'aide de la méthode `compareTo` de la classe `String`. La valeur résultant de

```
a.compareTo(b)
```

est un nombre négatif si `a` est lexicographiquement inférieur à `b`, 0 s'ils sont identiques et positif sinon.

Malheureusement, sauf si tous vos mots sont en caractères ASCII anglais majuscules, cette méthode est sans intérêt. Le problème est que la méthode `compareTo` dans le langage Java utilise les valeurs de caractères Unicode pour déterminer l'ordre de tri. Par exemple, les caractères minuscules ont une valeur Unicode supérieure à celle des caractères majuscules et les caractères accentués ont une valeur plus élevée encore. Cela peut donner des résultats absurdes ; par exemple, les cinq chaînes suivantes sont classées à l'aide de la méthode `compareTo` :

```
America  
Zulu  
ant  
zebra  
Ångström
```

Pour l'ordre du dictionnaire, nous voulons que les caractères majuscules et minuscules soient considérés comme équivalents. Pour un anglophone, cette liste de mots serait classée ainsi :

```
America  
Ångström  
ant  
zebra  
Zulu
```

Cet ordre ne serait toutefois pas acceptable pour un utilisateur suédois. En suédois, la lettre Å est différente de la lettre A et elle se place *après* la lettre Z ! C'est-à-dire qu'un utilisateur suédois voudra que les mots soient classés de la façon suivante :

```
America  
ant  
zebra  
Zulu  
Ångström
```

Heureusement, lorsque vous êtes conscient du problème, le classement est assez facile. Comme toujours, vous commencez par extraire un objet `Locale`. Puis vous appelez la méthode `factory getInstance` pour obtenir un objet `Collator`. Enfin, vous utilisez la méthode `compare` de

collator, au lieu de la méthode `compareTo` de la classe `String`, chaque fois que vous voulez trier des chaînes :

```
Locale loc = . . . ;
Collator coll = Collator.getInstance(loc);
if (coll.compare(a, b) < 0) // a est avant b . . . ;
```

Plus important encore, la classe `Collator` implémente l'interface de `Comparator`. Vous pouvez, par conséquent, passer un objet `collator` à la méthode `Collections.sort` afin de trier une liste de chaînes :

```
Collections.sort(strings, coll);
```

Vous pouvez définir une "force" (*strength*) de tri, pour déterminer combien elle doit être sélective. Les différences entre les caractères sont classifiées comme primaires, secondaires et tertiaires. Par exemple, en anglais la différence entre "A" et "Z" est considérée comme primaire, la différence entre "A" et "Å" est secondaire, et entre "A" et "a" tertiaire.

En définissant la force du collator (*trieur*) à `Collator.PRIMARY`, vous lui demandez de tenir compte seulement des différences primaires. Si vous définissez cette force à `Collator.SECONDARY`, les différences secondaires seront prises en compte. C'est-à-dire que deux chaînes auront plus de probabilité d'être considérées comme différentes si la force est secondaire.

Le Tableau 8.4 montre un exemple de jeu de chaînes trié avec les trois forces de classement. Remarquez que la force indique seulement si deux chaînes sont considérées comme identiques.

Tableau 8.4 : Classement avec différentes forces

<i>PRIMARY</i>	<i>SECONDARY</i>	<i>TERTIARY</i>
Angstrom = Ångström,	Angstrom,	Angstrom,
Ant = ant,	Ångström,	Ångström,
	Ant = ant	Ant,
		ant

Enfin, il existe un autre paramètre technique : le "mode décomposition". Le mode par défaut, "*canonical decomposition*" (décomposition canonique), est approprié dans la majorité des cas. Si vous choisissez "*no decomposition*" (pas de décomposition), les caractères accentués ne sont pas décomposés dans leur forme de base + accent. Cette option est plus rapide, mais elle ne donne des résultats corrects que si l'entrée ne contient pas de caractères accentués. Enfin, "*full decomposition*" (décomposition complète) analyse les variantes Unicode, c'est-à-dire les caractères Unicode qui doivent être considérés comme identiques. Par exemple, les affichages japonais ont deux façons de montrer les caractères anglais, katakana et hiragana, appelées *half-width* et *full-width*. Les caractères "half-width" (demi-largeur) ont un espacement normal, alors que les caractères "full-width" (pleine largeur) sont espacés sur la même grille que les idéogrammes. (On peut rétorquer qu'il s'agit là d'un problème de présentation et que cela ne devrait pas engendrer des caractères Unicode différents, mais les règles sont ainsi faites.) Avec la décomposition complète, les variantes half-width et full-width de la même lettre sont reconnues comme identiques.

648 Au cœur de Java 2 - Fonctions avancées

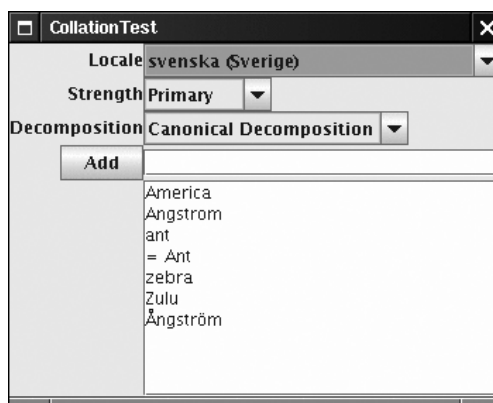
Il est inutile que le trieur décompose une chaîne plusieurs fois. Si une chaîne est comparée plusieurs fois à d'autres chaînes, vous pouvez sauvegarder la décomposition dans un objet `CollationKey`. La méthode `getCollationKey` renvoie un objet `CollationKey` que vous pouvez utiliser ultérieurement pour accélérer les comparaisons. Voici un exemple :

```
String a = . . . ;
CollationKey aKey = coll.getCollationKey(a);
if(aKey.compareTo(coll.getCollationKey(b)) == 0)
    // comparaison rapide
. . .
```

Le programme dans l'Exemple 8.4 vous permet d'expérimenter l'ordre de classement. Entrez un mot dans le champ de texte et cliquez sur Ajouter pour l'ajouter à la liste de mots. Chaque fois que vous ajouterez un autre mot, ou que vous changerez les paramètres régionaux, la force ou le mode de décomposition, la liste des mots sera triée à nouveau. Un signe = indique les mots qui sont considérés comme identiques (voir Figure 8.3).

Figure 8.3

Le programme
CollationTest.



Dans ce programme, nous fournissons une liste améliorée pour choisir le paramètre régional (voir Exemple 8.5). La liste résout deux problèmes :

- Les chaînes des paramètres régionaux sont stockées selon l'ordre de tri du paramètre actuel. Nous installons un modèle personnalisé qui contient les chaînes triées et les remplace lorsque le paramètre régional d'un composant est modifié (sachez que les noms du paramètre régional changent lorsque vous modifiez le paramètre).
- La liste déroulante contient les objets `Locale`, et non des chaînes. Nous installons un système de rendu personnalisé de sorte que la méthode `getDisplay` du paramètre du composant soit utilisée pour le rendu.

Exemple 8.4 : *CollationTest.java*

```
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;
import java.util.List;
import javax.swing.*;
```

```

/**
 * Ce programme montre le tri des chaînes en fonction
 * de divers paramètres régionaux.
 */
public class CollationTest
{
    public static void main(String[] args)
    {
        JFrame frame = new CollationFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc contient des zones de liste modifiables
 * permettant de choisir un paramètre régional, une
 * force de tri et des règles de décomposition, un champ
 * de texte ainsi qu'un bouton pour ajouter de nouvelles
 * chaînes et une zone de texte pour recenser les
 * chaînes triées.
 */
class CollationFrame extends JFrame
{
    public CollationFrame()
    {
        setTitle("CollationTest");

        setLayout(new GridBagLayout());
        add(new JLabel("Param. Rég."), new GBC(0, 0).setAnchor(GBC.EAST));
        add(new JLabel("Force"), new GBC(0, 1).setAnchor(GBC.EAST));
        add(new JLabel("Décomposition"), new GBC(
            0, 2).setAnchor(GBC.EAST));
        add(addButton, new GBC(0, 3).setAnchor(GBC.EAST));
        add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
        add(strengthCombo, new GBC(1, 1).setAnchor(GBC.WEST));
        add(decompositionCombo, new GBC(1, 2).setAnchor(GBC.WEST));
        add(newWord, new GBC(1, 3).setFill(GBC.HORIZONTAL));
        add(new JScrollPane(sortedWords), new GBC(1, 4).setFill(GBC.BOTH));

        strings.add("America");
        strings.add("ant");
        strings.add("Zulu");
        strings.add("zebra");
        strings.add("\u00C5ngstr\u00F6m");
        strings.add("Angstrom");
        strings.add("Ant");
        updateDisplay();

        addButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    strings.add(newWord.getText());
                    updateDisplay();
                }
            });
    }
}

```


650 Au cœur de Java 2 - Fonctions avancées

```
ActionListener listener = new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        { updateDisplay(); }
    };

localeCombo.addActionListener(listener);
strengthCombo.addActionListener(listener);
decompositionCombo.addActionListener(listener);
pack();
}

/**
 Actualise l'affichage et trie les chaînes en
 fonction des chaînes de l'utilisateur.
 */
public void updateDisplay()
{
    Locale currentLocale = (Locale)
        localeCombo.getSelectedIndex();
    localeCombo.setLocale(currentLocale);

    currentCollator
        = Collator.getInstance(currentLocale);
    currentCollator.setStrength(strengthCombo.getValue());
    currentCollator.setDecomposition(
        decompositionCombo.getValue());

    Collections.sort(strings, currentCollator);

    sortedWords.setText("");
    for (int i = 0; i < strings.size(); i++)
    {
        String s = strings.get(i);
        if (i > 0
            && currentCollator.compare(s, strings.get(i - 1)) == 0)
            sortedWords.append("= ");
        sortedWords.append(s + "\n");
    }
    pack();
}

private Locale[] locales;
private List<String> strings = new ArrayList<String>();
private Collator currentCollator;
private JComboBox localeCombo =
    new LocaleCombo(Collator.getAvailableLocales());

private EnumCombo strengthCombo
    = new EnumCombo(Collator.class,
        new String[] { "Primaire", "Secondaire", "Tertiaire" });
private EnumCombo decompositionCombo
    = new EnumCombo(Collator.class,
        new String[] { "Décomposition Canonique ",
```

```

        "Décomposition complète", "Pas de décomposition" });
private JTextField newWord = new JTextField(20);
private JTextArea sortedWords = new JTextArea(10, 20);
private JButton addButton = new JButton("Ajouter");
}

```

Exemple 8.5 : LocaleCombo.java

```

import java.awt.*;
import java.text.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Cette liste permet de choisir un paramètre régional. Les paramètres
 * sont affichés dans le paramètre de la liste et triés selon le tri
 * des paramètres régionaux affichés.
 */
public class LocaleCombo extends JComboBox
{
    /**
     * Construit une liste de paramètres qui affiche une collection
     * inaltérable de paramètres régionaux.
     * @param locales Les paramètres à afficher dans cette liste
     */
    public LocaleCombo(Locale[] locales)
    {
        this.locales = (Locale[]) locales.clone();
        sort();
        setSelectedItem(getLocale());
    }

    public void setLocale(Locale newValue)
    {
        super.setLocale(newValue);
        sort();
    }

    private void sort()
    {
        Object selected = getSelectedItem();
        final Locale loc = getLocale();
        final Collator collator = Collator.getInstance(loc);
        final Comparator<Locale> comp = new
            Comparator<Locale>()
        {
            public int compare(Locale a, Locale b)
            {
                return collator.compare(a.getDisplayName(loc),
                    b.getDisplayName(loc));
            }
        };
        Arrays.sort(locales, comp);
        setModel(new
            ComboBoxModel()
        {
            public Object getElementAt(int i) { return locales[i]; }
            public int getSize() { return locales.length; }
        });
    }
}

```

652 *Au cœur de Java 2 - Fonctions avancées*

```

        public void addListDataListener(ListDataListener l) {}
        public void removeListDataListener(ListDataListener l) {}
        public Object getSelectedItem() {
            return selected >= 0 ? locales[selected] : null; }
        public void setSelectedItem(Object anItem)
        {
            if (anItem == null) selected = -1;
            else selected = Arrays.binarySearch(
                locales, (Locale) anItem, comp);
        }

        private int selected;
    });
    setSelectedItem(selected);
}

public ListCellRenderer getRenderer()
{
    if (renderer == null)
    {
        final ListCellRenderer originalRenderer = super.getRenderer();
        if (originalRenderer == null) return null;
        renderer = new
            ListCellRenderer()
        {
            public Component getListCellRendererComponent(JList list,
                Object value, int index, boolean isSelected,
                boolean cellHasFocus)
            {
                String renderedValue = ((Locale)
                    value).getDisplayName(getLocale());
                return originalRenderer.getListCellRendererComponent(
                    list, renderedValue, index, isSelected,
                    cellHasFocus);
            }
        };
    }
    return renderer;
}

public void setRenderer(ListCellRenderer newValue)
{
    renderer = null;
    super.setRenderer(newValue);
}

private Locale[] locales;
private ListCellRenderer renderer;
}

```

java.text.Collator 1.1

- `static Locale[] getAvailableLocales()`
Renvoie un tableau d'objets `Locale` pour lesquels des objets `Collator` sont disponibles.
- `static Collator getInstance()`
- `static Collator getInstance(Locale l)`
Renvoient un collator (*trieur*) pour le paramétrage régional par défaut ou celui donné.

- `int compare(String a, String b)`
Renvoie une valeur négative si la chaîne *a* vient avant *b*, 0 si elles sont considérées comme identiques, une valeur positive sinon.
- `boolean equals(String a, String b)`
Renvoie `true` si elles sont considérées comme identiques, `false` sinon.
- `void setStrength(int strength)`
- `int getStrength()`
Définissent ou extraient la force (*strength*) du trieur. Une force plus élevée différencie plus les mots. Les valeurs pour *Strength* sont `Collator.PRIMARY`, `Collator.SECONDARY` et `Collator.TERTIARY`.
- `void setDecomposition(int decomp)`
- `int getDecompositon()`
Définissent ou extraient le mode de décomposition du trieur. Plus un trieur décompose une chaîne, plus il sera strict pour décider que deux chaînes sont considérées comme identiques. Les valeurs de décomposition sont :

`Collator.NO_DECOMPOSITION`, `Collator.CANONICAL_DECOMPOSITION` et `Collator.FULL_DECOMPOSITION`.
- `CollationKey getCollationKey(String a)`
Renvoie une clé de classement (*collation key*) qui contient une décomposition des caractères sous une forme permettant une comparaison rapide avec une autre clé de classement.

java.text.CollationKey 1.1

- `int compareTo(CollationKey b)`
Renvoie une valeur négative si cette clé vient avant *b*, 0 si elles sont considérées comme identiques, une valeur positive sinon.

Formatage de message

La bibliothèque Java possède une classe `MessageFormat` permettant le formatage de texte avec des parties variables, comme ceci :

```
"Le {2}, un {0} a détruit {1} maisons et provoqué {3} de dégâts."
```

Les nombres entre accolades représentent des emplacements réservés pour des valeurs et des noms réels. La méthode statique `MessageFormat` vous permet de remplacer les variables par les valeurs. Depuis le JDK 5.0, on trouve une méthode `"varargs"`, et vous pouvez donc simplement fournir les paramètres comme suit :

```
String msg = MessageFormat.format("Le {2}, un {0} a détruit {1} maisons  
et entraîné {3} de dégâts.",  
    "ouragan", 99, new GregorianCalendar(1999, 0, 1).getTime(), 10.0E7);
```

Avant le JDK 5.0, vous deviez placer les valeurs des emplacements dans un tableau `Object[]`.

Dans cet exemple, l'emplacement `{0}` est remplacé par `"ouragan"`, `{1}` est remplacé par `99`, etc.

654 Au cœur de Java 2 - Fonctions avancées

ATTENTION

La méthode statique `format` utilise le paramètre régional actuel pour mettre les valeurs en forme. Pour utiliser la classe `MessageFormat` avec un paramètre régional arbitraire, il faudra fournir d'autres efforts :

```
MessageFormat mf = new MessageFormat(pattern, loc);
String msg = mf.format(new Object[] { valeurs });
```

Ici, nous appelons la méthode `format` de la superclasse `Format`. Malheureusement, la classe `MessageFormat` ne fournit pas de méthode "varargs" équivalente.

Le résultat de notre exemple est la chaîne

```
Le 1/1/99 12:00 AM, un ouragan a détruit 99 maisons et causé 100,000,000
de dégâts.
```

C'est un début, mais ce n'est pas parfait. Nous ne voulons pas simplement afficher l'heure "12:00 AM" et voulons que la somme des dommages soit affichée sous forme de devise. Pour ce faire, nous fournissons un format optionnel pour certains emplacements :

```
"Le {2,date,long}, un {0} a détruit {1} maisons et causé
{3,number,currency} de dégâts."
```

Cet exemple affiche :

```
Le 1 janvier 1999, un ouragan a détruit 99 maisons
et causé 100 000 000,00 USD de dégâts.
```

En général, l'indice de l'emplacement peut être suivi par un *type* et un *style*. Séparez l'indice, le type et le style par des virgules. Le type peut être l'un de ceux-ci :

```
number
time
date
choice
```

Si le type est `number`, le style peut être

```
integer
currency
percent
```

ou un masque de formatage de nombre comme `$,##0` (consultez la documentation de la classe `DecimalFormat` pour en savoir plus sur les formats disponibles).

Si le type est `time` ou `date`, le style peut être

```
short
medium
long
full
```

ou un masque de formatage de date, comme `aaaa-MM-jj`. (Reportez-vous à la documentation de la classe `SimpleDateFormat` pour plus d'informations concernant les formats possibles.)

Les formats "de choix" sont plus complexes, nous les verrons dans la section suivante.

java.text.MessageFormat 1.1

- `MessageFormat(String pattern)`
- `MessageFormat(String pattern, Locale loc)`
Construisent un objet `MessageFormat` avec le modèle spécifié et un paramètre régional.
- `void applyPattern(String pattern)`
Définit un objet `MessageFormat` selon le modèle spécifié.
- `void setLocale(Locale loc)`
- `Locale getLocale()`
Définissent ou extraient les paramètres régionaux à utiliser pour les emplacements réservés du message. Les paramètres régionaux sont utilisés seulement pour les modèles que vous définissez par la suite en appelant la méthode `applyPattern`.
- `static String format(String pattern, Object... args)`
Met en forme la chaîne du masque en utilisant `args[i]` sous forme d'entrée pour un emplacement `{i}`.
- `StringBuffer format(Object args, StringBuffer result, FieldPosition pos)`
Met en forme le masque de ce `MessageFormat`. Le paramètre `args` doit être un tableau d'objets. La chaîne mise en forme est annexée au résultat et le résultat est renvoyé. Si `pos` est égal à `new FieldPosition(MessageFormat.Field.ARGUMENT)`, ses propriétés `beginIndex` et `endIndex` sont définies sur l'endroit du texte qui remplace l'emplacement `{1}`. Fournissez `null` si vous n'êtes pas intéressé par les informations de position.

java.text.Format 1.1

- `String format(Object obj)`
Met en forme l'objet donné, selon les règles du formateur. Cette méthode appelle `format(obj, new StringBuffer(), new FieldPosition(1)).toString()`.

Formats de choix

Examinons de plus près le modèle de la section précédente :

"Le {2}, un {0} a détruit {1} maisons et provoqué {3} de dégâts."

Si nous remplaçons l'emplacement réservé du désastre {0} par "tempête", la phrase n'est plus grammaticalement correcte en français.

Le 1 janvier 1999, un tempête a détruit ...

Cela signifie que nous devons intégrer l'article dans l'emplacement réservé :

"Le {2}, {0} a détruit {1} maisons et provoqué {3} de dégâts."

Puis, {0} sera remplacé par "une tempête" ou "un ouragan". Cela est particulièrement approprié si ce message doit être traduit dans une langue dans laquelle le genre d'un mot affecte l'article. C'est vrai aussi en allemand, le masque serait :

"{0} zerstörte am {2} {1} Häuser und richtete einen Schaden von {3} an."

L'emplacement réservé serait alors remplacé par une combinaison grammaticalement correcte de l'article et du nom, comme "Ein Wirbelsturm", "Eine Naturkatastrophe".

656 Au cœur de Java 2 - Fonctions avancées

Voyons maintenant le paramètre {1}. Si le désastre n'est pas aussi catastrophique, {1} pourrait alors être remplacé par le nombre 1, et le message serait :

```
Le 1 janvier 1999, un glissement de terrain a détruit 1 maisons et ...
```

Il serait préférable que le message varie en fonction de la valeur de l'emplacement réservé, afin de pouvoir exprimer

```
aucune maison
une maison
deux maisons
. . .
```

selon la valeur de l'emplacement réservé. L'option de mise en forme *choice* a été conçue pour cela.

Un format de choix est une suite de paires contenant chacune :

- une limite inférieure ;
- une chaîne de formatage.

La limite inférieure et la chaîne de formatage sont séparées par un caractère # et les paires sont séparées par des caractères |.

Par exemple :

```
{1,choice,0#aucune maison|1#une maison|2#{1} maisons}
```

Le Tableau 8.5 montre l'effet de cette chaîne de formatage pour diverses valeurs de {1}.

Pourquoi utiliser {1} deux fois dans la chaîne de formatage ? Lorsque le format du message applique le format de choix à l'emplacement {1} et que la valeur est supérieure ou égale à 2, le format de choix renvoie "{1} maisons". Cette chaîne est alors à nouveau mise en forme par le format du message et la réponse est répartie dans le résultat.

Tableau 8.5 : Chaîne mise en forme par *ChoiceFormat*

{1}	Résultat
0	"aucune maison"
1	"une maisons"
3	"3 maisons"
-1	"aucune maison"

INFO

Cet exemple montre que le concepteur de *ChoiceFormat* était un peu compliqué. Si vous avez trois chaînes de formatage, vous avez besoin de deux limites pour les séparer. En général, vous avez besoin d'une limite de moins que vous n'avez de chaînes de formatage. Comme vous l'avez vu au Tableau 8.4, la classe *MessageFormat* ignore la première limite.

La syntaxe aurait été un peu plus claire si le concepteur de cette classe avait réalisé que les limites se placent entre les choix, comme

```
aucune maison|1|une maison|2|{1} maisons // ce n'est pas le vrai format
```

Vous pouvez utiliser le symbole $<$ pour montrer qu'un choix doit être sélectionné si la limite inférieure est strictement inférieure à la valeur.

Vous pouvez également utiliser le symbole $\leq$ (exprimé sous la forme du code de caractère Unicode `\u2264`) comme synonyme de $\#$. Si vous le souhaitez, vous pouvez même spécifier une limite inférieure de $-\infty$ comme `-\u221E` pour la première valeur

Par exemple,

```
-\u221Eaucune maison|0<une maison|2\leq{1} maisons
```

ou, à l'aide des caractères d'échappement Unicode,

```
-\u221E<no houses|0<one house|2\u2264{1} houses
```

Revenons maintenant à notre histoire de catastrophe naturelle. Si nous mettons la chaîne de choix à l'intérieur de la chaîne de message d'origine, nous obtenons l'instruction suivante :

```
String pattern =
"Le {2,date,long}, {0} a détruit {1,choice,0#aucune
maison|1#une maison|2#{1} maisons}" +
"et provoqué {3,number,currency} de dégâts.";
```

Ou, en allemand :

```
String pattern =
"{0} zerstörte am {2,date,long} {1,choice,0#kein Haus|1#ein
Haus|2#{1} Häuser}" +
"und richtete einen Schaden von {3,number,currency} an.";
```

Vous remarquerez que l'ordre des mots est différent en allemand, mais le tableau d'objets que vous passez à la méthode `format` est le même. L'ordre des emplacements réservés dans la chaîne de formatage tient compte de la modification dans l'ordre des mots.

Fichiers texte et jeux de caractères

Comme vous le savez, le langage Java est totalement fondé sur Unicode. Toutefois, les systèmes d'exploitation possèdent généralement leur propre codage de caractères, comme ISO-8859-1 (un code sur 8 bits quelquefois appelé code "ANSI") aux Etats-Unis ou Big5 à Taïwan.

Lorsque vous enregistrez des données dans un fichier texte, respectez le codage local pour que les utilisateurs de votre programme puissent ouvrir le fichier texte avec leurs applications. Indiquez ensuite le codage des caractères dans le constructeur `FileWriter` :

```
out = new FileWriter(filename, "ISO-8859-1");
```

Vous trouverez une liste complète des codages pris en charge dans le Chapitre 12 du Volume 1.

Malheureusement, il n'y a généralement aucune connexion entre les paramètres régionaux et les codages caractères. Par exemple, si votre utilisateur a sélectionné le paramétrage régional taïwanais `zh_TW`, aucune méthode du langage Java ne vous dira que le code caractère BIG5 est le plus approprié.

Codage de caractères des fichiers source

Il est bon de garder à l'esprit que vous, programmeur, avez besoin de communiquer avec le compilateur Java. Et *vous le faites à l'aide d'outils de votre système local*. Par exemple, vous pouvez utiliser la version chinoise du Bloc-notes pour écrire vos fichiers code source Java. Les fichiers code source résultants ne seront *pas portables*, car ils utilisent le codage de caractère local (GB or BIG5, selon le système d'exploitation chinois utilisé). Seuls les fichiers de classe compilés sont portables ; ils utiliseront automatiquement le codage "UTF-8 modifié" pour les identificateurs et les chaînes. Cela signifie que même si un programme est compilé et exécuté, trois codages de caractères sont impliqués :

- Fichiers source : codage local.
- Fichiers de classe : UTF-8 modifié.
- Machine virtuelle : UTF-16.

Consultez le Chapitre 12 du Volume 1 pour obtenir une définition des formats UTF-8 modifié et UTF-16.

ASTUCE

Vous pouvez spécifier le codage des caractères de vos fichiers source grâce à l'indicateur `-encoding`, par exemple :

```
java -encoding Big5 Myfile.java
```

Pour rendre portables vos fichiers source, limitez-vous au codage ASCII pur. C'est-à-dire que vous devez remplacer tous les caractères non ASCII par leur équivalent Unicode. Par exemple, au lieu d'utiliser la chaîne "Häuser", utilisez "H\u0084user". Le JDK contient un utilitaire, `native2ascii`, que vous pouvez utiliser pour convertir le codage caractère natif en ASCII pur. Cet utilitaire remplace simplement chaque caractère non ASCII dans l'entrée par une séquence `\u` suivie des quatre chiffres hexa de la valeur Unicode. Pour utiliser le programme `native2ascii`, fournissez simplement les noms des fichiers en entrée et en sortie.

```
native2ascii Myfile.java Myfile.temp
```

Vous pouvez convertir dans l'autre sens à l'aide de l'option `-reverse` :

```
native2ascii -reverse Myfile.temp Myfile.java
```

Vous pouvez aussi spécifier un autre codage à l'aide de l'option `-encoding`. Le nom de codage doit être l'un de ceux répertoriés dans le tableau de codages du Chapitre 12 du Volume 1.

```
native2ascii -encoding Big5 Myfile.java Myfile.temp
```

ASTUCE

Il est recommandé de vous limiter aux noms de classes en ASCII pur. Puisque le nom de la classe compose le nom du fichier de classe, vous êtes à la merci du système de fichier local pour la gestion des noms codés en caractères non ASCII. Voici un exemple malheureux. Windows 95 utilise, pour ses noms de fichiers, le codage de caractères nommé *Page de code 437* ou *original PC*. Si vous construisez une classe Bär et que vous essayiez de l'exécuter sous Windows 95, vous obtenez un message d'erreur indiquant que la classe "BΣr" est introuvable.

Les groupes de ressources

Lorsque vous localiserez une application, vous aurez probablement un nombre impressionnant de chaînes de message, de libellés de boutons, etc., qui devront être traduits. Pour faciliter cette tâche, vous définirez les chaînes de message dans un emplacement externe, appelé habituellement *ressource*. La personne réalisant la traduction pourra simplement modifier les fichiers ressources sans avoir à toucher au code source du programme.

En Java, on utilise les fichiers de propriété pour spécifier des ressources de chaînes et implémenter des classes pour les ressources d'autres types.

INFO

Les ressources de la technologie Java ne sont pas les mêmes que les ressources Windows ou Macintosh. Un exécutable Windows ou Macintosh stocke les ressources telles que les menus, boîtes de dialogue, icônes et messages dans une section séparée du code du programme. Un éditeur de ressources peut inspecter et mettre à jour ces ressources sans affecter le code du programme.

INFO

Le Chapitre 10 du Volume 1 décrit un concept de fichiers ressources JAR, où les fichiers de données, sons et images peuvent être placés dans un fichier JAR. La méthode `getResource` de la classe `Class` recherche le fichier, l'ouvre et renvoie une URL à la ressource. En plaçant les fichiers dans le fichier JAR, la tâche de recherche des fichiers est laissée au chargeur de classe, qui sait déjà comment trouver les éléments d'un fichier JAR. Toutefois, ce mécanisme ne prend pas en charge les paramètres régionaux.

Trouver l'emplacement des groupes de ressources

Lorsque vous localisez une application, vous produisez un jeu de *groupes de ressources*. Chaque groupe est un fichier de propriétés ou une classe qui décrit les éléments spécifiques aux paramètres régionaux (comme les messages, libellés, etc.). Pour chaque groupe, vous fournissez les versions de tous les paramètres régionaux que vous voulez prendre en charge.

Vous devez utiliser une convention spécifique pour l'attribution des noms pour ces groupes. Par exemple, des ressources spécifiques à l'Allemagne iront dans un fichier *NomGroupe_de_DE*, alors que celles partagées par tous les pays germanophones iront dans la classe *NomGroupe_de*. En règle générale, utilisez

NomGroupe_langue_pays

pour toutes les ressources spécifiques à un pays, et utilisez

NomGroupe_langue

pour toutes les ressources spécifiques à une langue. Enfin, en dernier ressort, vous pouvez mettre les paramètres par défaut dans un fichier sans aucun suffixe.

Vous chargez un groupe de ressources à l'aide de la commande

```
ResourceBundle currentResources =  
    ResourceBundle.getBundle(NomGroupe, currentLocale);
```

660 Au cœur de Java 2 - Fonctions avancées

La méthode `getBundle` tente de charger le groupe correspondant aux paramètres régionaux actuels, par langue, par pays et par variante. Si le chargement échoue, la variante, le pays et la langue sont successivement abandonnés. La même recherche est appliquée au paramètre régional par défaut, puis le fichier de groupe par défaut est finalement consulté. Si même cette tentative échoue, la méthode lance une `MissingResourceException`.

C'est-à-dire que la méthode `getBundle` essaie de charger successivement l'un des groupes suivants, jusqu'à ce que le résultat soit positif :

```
NomGroupe_LangueParamRegActuel_PaysParamRegActuel_
    VarianteParamRegActuel
NomGroupe_LangueParamRegActuel_PaysParamRegActuel
NomGroupe_LangueParamRegActuel
NomGroupe_LangueParamRegDéfaut_PaysParamRegDéfaut_
    VarianteParamRegDéfaut
NomGroupe_LangueParamRegDéfaut_PaysParamRegDéfaut
NomGroupe_LangueParamRegDéfaut
NomGroupe
```

Une fois que la méthode `getBundle` aura trouvé l'emplacement d'un groupe, par exemple `NomGroupe_de_DE`, elle continuera pourtant à rechercher `NomGroupe_de` et `NomGroupe`. Si ces groupes existent, ils deviendront parents du groupe `NomGroupe_de_DE` dans la hiérarchie des ressources. Plus tard, lors de la recherche d'une ressource, les parents seront recherchés si la recherche a échoué dans le groupe courant. C'est-à-dire que, si une ressource particulière n'a pas été trouvée dans `NomGroupe_de_DE`, les classes `NomGroupe_de` et `NomGroupe` seront également recherchées.

Cela constitue un service de toute évidence très utile, qu'il serait fastidieux de programmer à la main. Le mécanisme de groupe de ressources du langage Java recherche automatiquement l'emplacement des éléments qui correspond le mieux pour un paramètre régional donné. Il est facile d'ajouter des localisations supplémentaires à un programme existant : il suffit d'ajouter des groupes de ressource supplémentaires.

ASTUCE

Vous n'avez pas besoin de placer toutes les ressources pour votre application dans un même groupe. Vous pouvez avoir un groupe pour les libellés de boutons, un pour les messages d'erreur, etc.

Fichiers de propriétés

Les chaînes d'internationalisation sont assez simples. Vous les placez toutes dans un fichier de propriétés comme `MyProgramStrings.properties`. Il s'agit d'un simple fichier texte contenant une paire de clés/valeurs par ligne. Un fichier ordinaire ressemblerait à ceci :

```
computeButton=Rechnen
colorName=black
defaultPaperSize=210x297
```

Vous nommez ensuite vos fichiers de propriétés comme indiqué dans la section précédente, par exemple :

```
MyProgramStrings.properties
MyProgramStrings_en.properties
MyProgramStrings_de_DE.properties
```

Puis vous chargez le groupe comme suit :

```
ResourceBundle bundle = ResourceBundle.getBundle(  
    "MyProgramStrings", locale);
```

Pour rechercher une chaîne spécifique, appelez :

```
String computeButtonLabel = bundle.getString("computeButton");
```

ATTENTION

Les fichiers pour le stockage de propriétés sont toujours des fichiers ASCII. Si vous devez placer des caractères Unicode dans un fichier de propriété, vous devez les coder à l'aide du codage \uxxxx. Par exemple, pour spécifier "colorName=Grün", utilisez

```
colorName=Gr\u00FCn
```

Vous pouvez utiliser l'outil `native2ascii` pour générer ces fichiers.

Classes de groupes

Pour fournir des ressources qui ne soient pas des chaînes, vous définissez des classes qui étendent la classe `ResourceBundle`. Utilisez la convention de nom standard pour nommer vos classes, par exemple :

```
MyProgramResources.java  
MyProgramResources_en.java  
MyProgramResources_de_DE.java
```

Vous chargez la classe avec la méthode `getBundle` utilisée pour charger un fichier de propriétés :

```
ResourceBundle bundle = ResourceBundle.getBundle(  
    "MyProgramResources", locale);
```

ATTENTION

Lorsque vous recherchez des groupes, un groupe d'une classe obtient la priorité par rapport à un fichier de propriétés lorsque les deux groupes ont le même nom de base.

Chaque classe de groupe de ressources implémente une table de recherche. Fournissez une chaîne de clés pour chaque paramètre à localiser et utilisez cette chaîne pour récupérer le paramètre. Par exemple :

```
Color backgroundColor = (Color) bundle.getObject("backgroundColor");  
double[] paperSize = (double[]) bundle.getObject("defaultPaperSize");
```

La manière la plus simple d'implémenter des classes de groupe de ressources consiste à étendre la classe `ListResourceBundle`. La classe `ListResourceBundle` vous permet de placer toutes vos ressources dans un tableau d'objets et réalise la correspondance pour vous. Vous devez fournir le squelette suivant :

```
public class NomGroupe_langue_pays  
    extends ListResourceBundle  
{  
    public Object[][] getContents() { return contents; }  
}
```

662 *Au cœur de Java 2 - Fonctions avancées*

```
static final Object[][] contents =
{
    { key1, valeur2 }
    { key2, valeur2 }
    . . .
}
```

Par exemple :

```
public class ProgramResources_de
    extends ListResourceBundle
{
    public Object[][] getContents() { return contents; }
    private static final Object[][] contents =
    {
        { "backgroundColor", Color.black },
        { "defaultPaperSize", new double[] { 210, 297 } }
    }
}

public class ProgramResources_en_US
    extends ListResourceBundle
{
    public Object[][] getContents() { return contents; }
    private static final Object[][] contents =
    {
        { "backgroundColor", Color.blue },
        { "defaultPaperSize", new double[] { 216, 279 } }
    }
}
```

INFO

Tout le monde sur cette planète, à l'exception des Etats-Unis et du Canada, utilise les formats de papier ISO 216. Pour plus d'informations, consultez le site <http://www.cl.cam.ac.uk/~mgk25/iso-paper.html>. Selon U.S. Metric Association (<http://lamar.colostate.edu/~hillger>), seulement trois pays dans le monde n'ont pas encore officiellement adopté le système métrique ; il s'agit du Liberia, de Myanmar (Birmanie) et des Etats-Unis d'Amérique. Les entreprises américaines qui souhaitent étendre leurs activités au niveau international doivent passer au système métrique. Reportez-vous à l'adresse http://ts.nist.gov/ts/htdocs/200/202/mpo_reso.htm pour obtenir un ensemble de liens utiles pointant vers des informations sur le système métrique.

Vos classes de groupe de ressources peuvent aussi étendre la classe `ResourceBundle`. Vous devez alors implémenter deux méthodes pour énumérer toutes les clés et rechercher la valeur d'une clé donnée :

```
Enumeration<String> getKeys()
Object handleGetObject(String key)
```

La méthode `getObject` de la classe `ResourceBundle` appelle la méthode `handleGetObject` que vous fournissez.

java.util.ResourceBundle 1.1

- `static ResourceBundle getBundle(String baseName, Locale loc)`
- `static ResourceBundle getBundle(String baseName)`

Chargent la classe `ResourceBundle` avec le nom donné, pour les paramètres régionaux donnés ou par défaut, et ses classes parent. Si les classes `ResourceBundle` se trouvent dans un package, le nom de base doit contenir le nom complet du package, comme `"intl.ProgramResources"`. Les classes `ResourceBundle` doivent être publiques, afin que la méthode `getBundle` puisse y accéder.

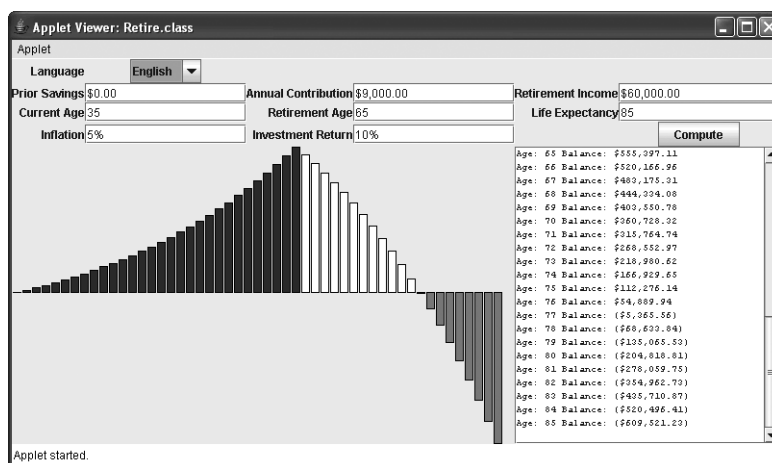
- `Object getObject(String name)`
Recherche un objet dans le groupe de ressources ou ses parents.
- `String getString(String name)`
Recherche un objet dans le groupe de ressources ou ses parents et le convertit en chaîne.
- `String[] getStringArray(String name)`
Recherche un objet dans le groupe de ressources ou ses parents et le convertit en tableau de chaînes.
- `Enumeration<String> getKeys()`
Renvoie un objet enumeration pour énumérer les clés de ce groupe de ressources. Les clés sont énumérées dans les groupes parent également.
- `Object handleGetObject(String key)`
Doit être écrasé pour rechercher la valeur de la ressource associée à la clé donnée, si vous définissez votre propre mécanisme de recherche de ressources.

Un exemple complet

Nous allons appliquer ici les conseils donnés dans ce chapitre, pour la localisation d'un applet de calcul de la retraite. L'applet calcule si vous épargnez ou non assez d'argent pour votre retraite. Vous entrez votre âge, la somme que vous épargnez chaque mois, etc. (voir Figure 8.4).

Figure 8.4

L'applet de calcul de la retraite en anglais.



La zone de texte et le graphique montrent le solde du compte pour chaque année. Si les nombres deviennent négatifs dans la dernière partie de votre vie et que les barres se retrouvent au-dessous de

664 Au cœur de Java 2 - Fonctions avancées

l'axe des x, il vous faudra trouver une solution, par exemple, économiser plus d'argent, différer votre retraite, mourir plus tôt ou être plus jeune.

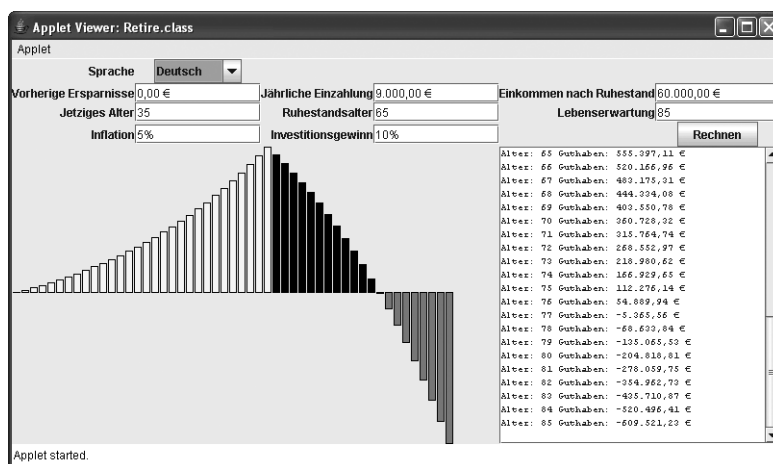
La calculatrice fonctionne maintenant avec trois paramétrages régionaux (anglais, français et chinois). Voici quelques-uns des points importants de l'internationalisation.

- Les libellés, boutons et messages sont traduits en français et chinois. Vous pouvez les trouver dans les classes `RetireResources_fr` et `RetireResources_zh`. L'anglais est utilisé par défaut avec le fichier `RetireResources`. Pour générer les messages chinois, nous avons d'abord saisi le fichier, à l'aide du Bloc-notes sous Windows version chinoise, puis nous avons utilisé l'utilitaire `native2ascii` pour convertir les caractères en Unicode.
- Chaque fois que les paramètres régionaux changent, nous réinitialisons les libellés et reformatons le contenu des champs texte.
- Les champs texte gèrent des nombres, des valeurs monétaires et des pourcentages dans le format local.
- Le champ de calcul utilise un `MessageFormat`. La chaîne de formatage est stockée dans le groupe de ressources de chaque langue.
- Juste pour montrer que c'est possible, nous avons utilisé des couleurs différentes pour le graphique, en fonction de la langue choisie par l'utilisateur.

Les Exemples 8.6 à 8.9 montrent le code. Les Exemples 8.10 à 8.12 sont les fichiers de propriété pour les chaînes localisées. Les Figures 8.5 et 8.6 montrent les sorties obtenues en français et chinois. Pour pouvoir afficher les caractères chinois, installez manuellement les polices chinoises. Dans le cas contraire, tous les caractères chinois s'afficheront sous la forme "caractère manquant".

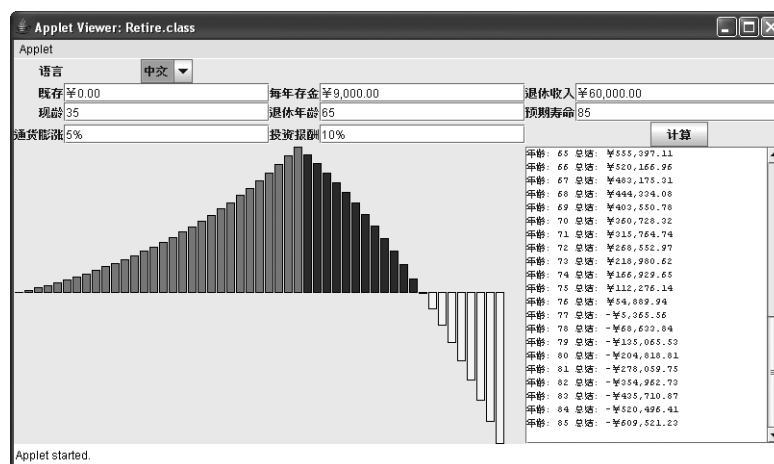
Figure 8.5

L'applet de calcul de la retraite en français.



En somme, bien que le mécanisme de localisation de Java présente toujours quelques inconvénients, il affiche une vertu majeure. Dès que vous avez organisé votre application en vue de la localisation, l'ajout d'autres versions localisées se fait en toute simplicité. Il vous suffit de fournir d'autres fichiers de ressources, ils seront automatiquement chargés lorsqu'un utilisateur souhaite les utiliser.

Figure 8.6
L'applet de calcul
de la retraite en
chinois.



INFO

Cette applet a été plus difficile à écrire qu'une application localisée typique, car l'utilisateur peut changer les paramètres régionaux à la volée. L'applet doit, par conséquent, être prête à se redessiner elle-même chaque fois que l'utilisateur sélectionne un nouveau paramétrage régional. Si vous voulez simplement afficher votre applet dans le paramétrage régional par défaut de l'utilisateur, ce sera beaucoup plus facile. Il suffira d'appeler `getLocale()` pour trouver le paramétrage régional du système de l'utilisateur, puis de l'utiliser pendant toute la durée de l'application.

Exemple 8.6 : *Retire.java*

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.applet.*;
import java.util.*;
import java.text.*;
import java.io.*;
import javax.swing.*;

/**
 * Cette applet affiche une calculatrice pour la
 * retraite. L'interface utilisateur est affichée en
 * anglais, français et chinois.
 */
public class Retire extends JApplet
{
    public void init()
    {
        setLayout(new GridBagLayout());
        add(languageLabel, new GBC(0, 0).setAnchor(GBC.EAST));
        add(savingsLabel, new GBC(0, 1).setAnchor(GBC.EAST));
        add(contribLabel, new GBC(2, 1).setAnchor(GBC.EAST));
        add(incomeLabel, new GBC(4, 1).setAnchor(GBC.EAST));
        add(currentAgeLabel, new GBC(0, 2).setAnchor(GBC.EAST));
        add(retireAgeLabel, new GBC(2, 2).setAnchor(GBC.EAST));
        add(deathAgeLabel, new GBC(4, 2).setAnchor(GBC.EAST));
        add(inflationPercentLabel, new GBC(0, 3).setAnchor(GBC.EAST));
    }
}
```


666 Au cœur de Java 2 - Fonctions avancées

```

add(investPercentLabel, new GBC(2, 3).setAnchor(GBC.EAST));
add(localeCombo, new GBC(1, 0));
add(savingsField, new GBC(1, 1).setWeight(
    100, 0).setFill(GBC.HORIZONTAL));
add(contribField, new GBC(3, 1).setWeight(
    100, 0).setFill(GBC.HORIZONTAL));
add(incomeField, new GBC(5, 1).setWeight(
    100, 0).setFill(GBC.HORIZONTAL));
add(currentAgeField, new GBC(1, 2).setWeight(
    100, 0).setFill(GBC.HORIZONTAL));
add(retireAgeField, new GBC(3, 2).setWeight(
    100, 0).setFill(GBC.HORIZONTAL));
add(deathAgeField, new GBC(5, 2).setWeight(
    100, 0).setFill(GBC.HORIZONTAL));
add(inflationPercentField, new GBC(1, 3).setWeight(
    100, 0).setFill(GBC.HORIZONTAL));
add(investPercentField, new GBC(3, 3).setWeight(
    100, 0).setFill(GBC.HORIZONTAL));
add(retireCanvas, new GBC(0, 4, 4, 1).setWeight(
    0, 100).setFill(GBC.BOTH));
add(new JScrollPane(retireText), new GBC(4, 4, 2, 1).setWeight(
    0, 100).setFill(GBC.BOTH));

computeButton.setName("computeButton");
computeButton.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            getInfo();
            updateData();
            updateGraph();
        }
    });
add(computeButton, new GBC(5, 3));

retireText.setEditable(false);
retireText.setFont(new Font("Monospaced", Font.PLAIN, 10));

info.setSavings(0);
info.setContrib(9000);
info.setIncome(60000);
info.setCurrentAge(35);
info.setRetireAge(65);
info.setDeathAge(85);
info.setInvestPercent(0.1);
info.setInflationPercent(0.05);

int localeIndex = 0; // Paramètres régional US par défaut
for (int i = 0; i < locales.length; i++) // si le paramètre actuel
    // est l'un des choix, le sélectionner
    if (getLocale().equals(locales[i])) localeIndex = i;
setCurrentLocale(locales[localeIndex]);

localeCombo.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {

```

```

        setCurrentLocale((Locale)
            localeCombo.getSelectedIndex());
    });
}

/**
 * Définit le paramétrage régional actuel.
 * @param locale le paramétrage régional souhaité
 */
public void setCurrentLocale(Locale locale)
{
    currentLocale = locale;
    localeCombo.setSelectedItem(currentLocale);
    localeCombo.setLocale(currentLocale);

    res = ResourceBundle.getBundle("RetireResources",
        currentLocale);
    resStrings = ResourceBundle.getBundle("RetireStrings",
        currentLocale);
    currencyFmt
        = NumberFormat.getCurrencyInstance(currentLocale);
    numberFmt
        = NumberFormat.getNumberInstance(currentLocale);
    percentFmt
        = NumberFormat.getPercentInstance(currentLocale);

    updateDisplay();
    updateInfo();
    updateData();
    updateGraph();
}

/**
 * Met à jour toutes les étiquettes dans l'affichage.
 */
public void updateDisplay()
{
    languageLabel.setText(res.getString("language"));
    savingsLabel.setText(res.getString("savings"));
    contribLabel.setText(res.getString("contrib"));
    incomeLabel.setText(res.getString("income"));
    currentAgeLabel.setText(res.getString("currentAge"));
    retireAgeLabel.setText(res.getString("retireAge"));
    deathAgeLabel.setText(res.getString("deathAge"));
    inflationPercentLabel.setText
        (res.getString("inflationPercent"));
    investPercentLabel.setText
        (res.getString("investPercent"));
    computeButton.setText(res.getString("computeButton"));

    validate();
}

/**
 * Met à jour les informations des champs de texte.
 */
public void updateInfo()
{

```

668 *Au cœur de Java 2 - Fonctions avancées*

```
savingsField.setText(currencyFmt.format(info.getSavings()));
contribField.setText(currencyFmt.format(info.getContrib()));
incomeField.setText(currencyFmt.format(info.getIncome()));
currentAgeField.setText(numberFmt.format(info.getCurrentAge()));
retireAgeField.setText(numberFmt.format(info.getRetireAge()));
deathAgeField.setText(numberFmt.format(info.getDeathAge()));
investPercentField.setText(percentFmt.format
    (info.getInvestPercent()));
inflationPercentField.setText(percentFmt.format
    (info.getInflationPercent()));
}

/**
 * Met à jour les données affichées dans la zone de
 * texte.
 */
public void updateData()
{
    retireText.setText("");
    MessageFormat retireMsg = new MessageFormat("");
    retireMsg.setLocale(currentLocale);
    retireMsg.applyPattern(resStrings.getString("retire"));

    for (int i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
    {
        Object[] args = { i, info.getBalance(i) };
        retireText.append(retireMsg.format(args) + "\n");
    }
}

/**
 * Met le graphe à jour.
 */
public void updateGraph()
{
    retireCanvas.setColorPre((Color) res.getObject("colorPre"));
    retireCanvas.setColorGain((Color) res.getObject("colorGain"));
    retireCanvas.setColorLoss((Color) res.getObject("colorLoss"));
    retireCanvas.setInfo(info);
    repaint();
}

/**
 * Lit l'entrée de l'utilisateur à partir des champs
 * de texte.
 */
public void getInfo()
{
    try
    {
        info.setSavings(currencyFmt.parse
            (savingsField.getText()).doubleValue());
        info.setContrib(currencyFmt.parse
            (contribField.getText()).doubleValue());
        info.setIncome(currencyFmt.parse
            (incomeField.getText()).doubleValue());
        info.setCurrentAge(numberFmt.parse
            (currentAgeField.getText()).intValue());
        info.setRetireAge(numberFmt.parse
```

```

        (retireAgeField.getText()).intValue());
    info.setDeathAge(numberFmt.parse
        (deathAgeField.getText()).intValue());
    info.setInvestPercent(percentFmt.parse
        (investPercentField.getText()).doubleValue());
    info.setInflationPercent(percentFmt.parse
        (inflationPercentField.getText()).doubleValue());
    }
    catch (ParseException e)
    {
    }
}

private JTextField savingsField = new JTextField(10);
private JTextField contribField = new JTextField(10);
private JTextField incomeField = new JTextField(10);
private JTextField currentAgeField = new JTextField(4);
private JTextField retireAgeField = new JTextField(4);
private JTextField deathAgeField = new JTextField(4);
private JTextField inflationPercentField = new JTextField(6);
private JTextField investPercentField = new JTextField(6);
private JTextArea retireText = new JTextArea(10, 25);
private RetireCanvas retireCanvas = new RetireCanvas();
private JButton computeButton = new JButton();
private JLabel languageLabel = new JLabel();
private JLabel savingsLabel = new JLabel();
private JLabel contribLabel = new JLabel();
private JLabel incomeLabel = new JLabel();
private JLabel currentAgeLabel = new JLabel();
private JLabel retireAgeLabel = new JLabel();
private JLabel deathAgeLabel = new JLabel();
private JLabel inflationPercentLabel = new JLabel();
private JLabel investPercentLabel = new JLabel();

private RetireInfo info = new RetireInfo();

private Locale[] locales = {
    Locale.US, Locale.CHINA, Locale.GERMANY };
private Locale currentLocale;
private JComboBox localeCombo = new JComboBox(locales);
private ResourceBundle res;
private ResourceBundle resStrings;
private NumberFormat currencyFmt;
private NumberFormat numberFmt;
private NumberFormat percentFmt;
}

/**
 * Les informations requises pour calculer les revenus
 * de la retraite.
 */
class RetireInfo
{
    /**
     * Extrait le solde disponible pour une année donnée.
     * @param year l'année pour laquelle calculer le
     * solde
     * @return la somme d'argent disponible (ou exigée)
     * pour cette année
     */
}

```

670 *Au cœur de Java 2 - Fonctions avancées*

```

*/
public double getBalance(int year)
{
    if (year < currentAge) return 0;
    else if (year == currentAge)
    {
        age = year;
        balance = savings;
        return balance;
    }
    else if (year == age)
        return balance;
    if (year != age + 1)
        getBalance(year - 1);
    age = year;
    if (age < retireAge)
        balance += contrib;
    else
        balance -= income;
    balance = balance
        * (1 + (investPercent - inflationPercent));
    return balance;
}

/**
 * Extrait le montant des économies précédentes.
 * @return le montant des économies
 */
public double getSavings() { return savings; }

/**
 * Définit le montant des économies précédentes.
 * @param newValue la somme des économies
 */
public void setSavings(double newValue) { savings = newValue; }

/**
 * Extrait la contribution annuelle du compte de
 * retraite.
 * @return la somme de la contribution
 */
public double getContrib() { return contrib; }

/**
 * Définit la contribution annuelle au compte de
 * retraite.
 * @param newValue la somme de la contribution
 */
public void setContrib(double newValue) { contrib = newValue; }

/**
 * Extrait le revenu annuel.
 * @return le montant des revenus
 */
public double getIncome() { return income; }

/**
 * Définit les revenus annuels.
 * @param newValue le montant des revenus

```

```

    */
    public void setIncome(double newValue) { income = newValue; }

    /**
     * Extrait l'âge actuel.
     * @return l'âge
     */
    public int getCurrentAge() { return currentAge; }

    /**
     * Définit l'âge actuel.
     * @param newValue l'âge
     */
    public void setCurrentAge(int newValue) { currentAge = newValue; }

    /**
     * Extrait l'âge souhaité pour la retraite.
     * @return l'âge
     */
    public int getRetireAge() { return retireAge; }

    /**
     * Définit l'âge souhaité pour la retraite.
     * @param newValue l'âge
     */
    public void setRetireAge(int newValue) { retireAge = newValue; }

    /**
     * Extrait l'âge de décès attendu.
     * @return l'âge
     */
    public int getDeathAge() { return deathAge; }

    /**
     * Définit l'âge attendu de décès.
     * @param newValue l'âge
     */
    public void setDeathAge(int newValue) { deathAge = newValue; }

    /**
     * Extrait le pourcentage d'inflation estimé.
     * @return le pourcentage
     */
    public double getInflationPercent() { return inflationPercent; }

    /**
     * Définit le pourcentage d'inflation estimé.
     * @param newValue le pourcentage
     */
    public void setInflationPercent(double newValue) {
        inflationPercent = newValue; }

    /**
     * Extrait la production estimée de l'investissement.
     * @return le pourcentage
     */
    public double getInvestPercent() { return investPercent; }

    /**

```

672 *Au cœur de Java 2 - Fonctions avancées*

```

        Définit la production estimée de l'investissement.
        @param newValue le pourcentage
    */
    public void setInvestPercent(double newValue) {
        investPercent = newValue; }

    private double savings;
    private double contrib;
    private double income;
    private int currentAge;
    private int retireAge;
    private int deathAge;
    private double inflationPercent;
    private double investPercent;

    private int age;
    private double balance;
}

/**
    Ce panneau dessine un graphe du résultat de
    l'investissement.
*/
class RetireCanvas extends JPanel
{
    public RetireCanvas()
    {
        setSize(PANEL_WIDTH, PANEL_HEIGHT);
    }

    /**
        Définit l'information de retraite à dessiner.
        @param newInfo les nouvelles infos de retraite
    */
    public void setInfo(RetireInfo newInfo)
    {
        info = newInfo;
        repaint();
    }

    public void paint(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
        if (info == null) return;

        double minValue = 0;
        double maxValue = 0;
        int i;
        for (i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
        {
            double v = info.getBalance(i);
            if (minValue > v) minValue = v;
            if (maxValue < v) maxValue = v;
        }
        if (maxValue == minValue) return;

        int barWidth = getWidth() / (info.getDeathAge()
            - info.getCurrentAge() + 1);
        double scale = getHeight() / (maxValue - minValue);
    }
}

```

```

    for (i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
    {
        int x1 = (i - info.getCurrentAge()) * barWidth + 1;
        int y1;
        double v = info.getBalance(i);
        int height;
        int yOrigin = (int) (maxValue * scale);

        if (v >= 0)
        {
            y1 = (int) ((maxValue - v) * scale);
            height = yOrigin - y1;
        }
        else
        {
            y1 = yOrigin;
            height = (int) (-v * scale);
        }

        if (i < info.getRetireAge())
            g2.setPaint(colorPre);
        else if (v >= 0)
            g2.setPaint(colorGain);
        else
            g2.setPaint(colorLoss);
        Rectangle2D bar = new Rectangle2D.Double(x1, y1,
            barWidth - 2, height);
        g2.fill(bar);
        g2.setPaint(Color.black);
        g2.draw(bar);
    }
}

/**
 * Définit la couleur à utiliser avant la retraite.
 * @param color la couleur souhaitée
 */
public void setColorPre(Color color)
{
    colorPre = color;
    repaint();
}

/**
 * Définit la couleur à utiliser après la retraite si
 * le solde du compte est positif.
 * @param color la couleur souhaitée
 */
public void setColorGain(Color color)
{
    colorGain = color;
    repaint();
}

/**
 * Définit la couleur à utiliser après la retraite si
 * le solde du compte est négatif.
 * @param color la couleur souhaitée

```


674 *Au cœur de Java 2 - Fonctions avancées*

```

    */
    public void setColorLoss(Color color)
    {
        colorLoss = color;
        repaint();
    }

    private RetireInfo info = null;

    private Color colorPre;
    private Color colorGain;
    private Color colorLoss;
    private static final int PANEL_WIDTH = 400;
    private static final int PANEL_HEIGHT = 200;
}

```

Exemple 8.7 : RetireResources.java

```

import java.util.*;
import java.awt.*;

/**
 * Voici les ressources anglaises différentes des
 * chaînes pour le calcul de la retraite.
 */
public class RetireResources
    extends java.util.ListResourceBundle
{
    public Object[][] getContents() { return contents; }
    static final Object[][] contents =
    {
        // DEBUT LOCALISATION
        { "colorPre", Color.blue },
        { "colorGain", Color.white },
        { "colorLoss", Color.red }
        // FIN LOCALISATION
    };
}

```

Exemple 8.8 : RetireResources_fr.java

```

import java.util.*;
import java.awt.*;

/**
 * Voici les ressources françaises différentes des
 * chaînes pour le calcul de la retraite.
 */
public class RetireResources_fr
    extends java.util.ListResourceBundle
{
    public Object[][] getContents() { return contents; }
    static final Object[][] contents =
    {
        // DEBUT LOCALISATION
        { "colorPre", Color.yellow },

```

```
        { "colorGain", Color.black },
        { "colorLoss", Color.red }
        // FIN LOCALISATION
    };
}
```

Exemple 8.9 : RetireResources_zh.java

```
import java.util.*;
import java.awt.*;

/**
 * Voici les ressources chinoises différentes des
 * chaînes pour le calcul de la retraite.
 */
public class RetireResources_zh
    extends java.util.ListResourceBundle
{
    public Object[][] getContents() { return contents; }
    static final Object[][] contents =
    {
        // DEBUT LOCALISATION
        { "colorPre", Color.red },
        { "colorGain", Color.blue },
        { "colorLoss", Color.yellow }
        // FIN LOCALISATION
    };
}
```

Exemple 8.10 : RetireStrings.properties

```
language=Language
computeButton=Compute
savings=Prior Savings
contrib=Annual Contribution
income=Retirement Income
currentAge=Current Age
retireAge=Retirement Age
deathAge=Life Expectancy
inflationPercent=Inflation
investPercent=Investment Return
retire=Age: {0,number} Balance: {1,number,currency}
```

Exemple 8.11 : RetireStrings_fr.properties

```
language=français
computeButton=Calculer
savings=Epargne précéd.
contrib=Contrib. annuelle
income=Revenus retraite
currentAge=Age actuel
retireAge=Age retraite
deathAge=Espérance vie
inflationPercent=Inflation
investPercent=Retour invest.
retire=Age : {0,number} Solde : {1,number,currency}
```

676 *Au cœur de Java 2 - Fonctions avancées*

Exemple 8.12 : RetireStrings_zh.properties

```
language=\u8bed\u8a00
computeButton=\u8ba1\u7b97
savings=\u65e2\u5b58
contrib=\u6bcf\u5e74\u5b58\u91d1
income=\u9000\u4f11\u6536\u5165
currentAge=\u73b0\u9f84
retireAge=\u9000\u4f11\u5e74\u9f84
deathAge=\u9884\u671f\u5bff\u547d
inflationPercent=\u901a\u8d27\u81a8\u6da8
investPercent=\u6295\u8d44\u62a5\u916c
retire=\u5e74\u9f84: {0,number} \u603b\u7ed3: {1,number,currency}
```

java.awt.Applet 1.0

- `Locale getLocale()` 1.1

Extrait le paramétrage régional actuel de l'applet. Le paramétrage régional actuel est déterminé par la configuration de l'ordinateur de l'utilisateur exécutant l'applet.

9

Méthodes natives

Au sommaire de ce chapitre

- ✓ Appeler une fonction C à partir du langage de programmation Java
- ✓ Les paramètres numériques et les valeurs renvoyées
- ✓ Les paramètres chaîne
- ✓ Accéder aux champs
- ✓ Coder les signatures
- ✓ Appeler les méthodes Java
- ✓ Accéder aux éléments de tableau
- ✓ La gestion des erreurs
- ✓ L'API d'invocation
- ✓ Un exemple exhaustif : accès à la base de registres de Windows

Vous êtes certainement convaincu que le code écrit en langage Java a un certain nombre d'avantages sur le code écrit dans des langages comme C ou C++, même en ce qui concerne les applications spécifiques à une plate-forme. Il n'est bien sûr pas question ici de portabilité, mais plutôt de caractéristiques comme :

- La probabilité plus grande de produire un code exempt de bogues avec le langage Java qu'avec C ou C++.
- Le multithreading est certainement plus facile à coder en langage Java que dans la plupart des autres langages.
- Le code réseau est facile comme tout.

La portabilité est simplement un *plus*, dont vous pouvez ou non vouloir tirer parti accessoirement.

Tandis qu'une solution "100 % pure Java" est en principe sympathique, pour être réaliste, en ce qui concerne une application, il existe des situations où vous voudrez écrire (ou utiliser) du code écrit

678 Au cœur de Java 2 - Fonctions avancées

dans un autre langage. (Un tel code est généralement appelé *code natif*.) Il y a trois raisons évidentes pour lesquelles ce peut être le bon choix :

1. Vous disposez d'une quantité substantielle de code testé et débogué, disponible dans cet autre langage. Le portage du code en langage Java prendrait beaucoup de temps et le code résultant devrait être de nouveau testé et débogué.
2. Votre application nécessite l'accès à des unités ou des fonctionnalités système et l'utilisation de la technologie Java serait au mieux fastidieuse, et au pire impossible.
3. La vitesse d'exécution du code est essentielle. La tâche peut par exemple être critique au niveau temps, ou il peut s'agir d'un code utilisé si souvent que son optimisation peut réellement se révéler rentable. C'est en fait la raison la moins plausible. Avec la compilation JIT (*juste à temps*), les calculs intensifs codés en langage Java ne sont pas tellement plus lents que le code compilé C.

Si vous êtes dans l'une de ces trois situations, il peut être judicieux d'appeler le code natif à partir de programmes écrits en langage Java. Bien entendu, avec le gestionnaire de sécurité habituel en place, une fois que vous avez commencé à utiliser le code natif, vous êtes limité aux applications au lieu des applets. En particulier, la bibliothèque de code natif que vous appelez doit exister sur la machine cliente, et elle doit fonctionner avec l'architecture de la machine cliente.

Pour rendre possibles les appels aux méthodes natives, la technologie Java prévoit des points d'insertion pour le travail avec les bibliothèques système, et le JDK dispose de quelques outils permettant de soulager d'une partie de la corvée de programmation (mais d'une partie seulement).

INFO

Le langage utilisé pour votre code natif ne doit pas nécessairement être C ou C++ ; vous pouvez utiliser du code FORTRAN, si vous avez accès à une liaison entre le code Java et FORTRAN.

Gardez toutefois ceci à l'esprit : si vous utilisez des méthodes natives, vous perdez la portabilité. Même lorsque vous distribuez votre programme en tant qu'application, vous devez fournir une bibliothèque de méthodes natives séparée pour chacune des plates-formes que vous désirez prendre en charge. Cela signifie que vous devrez aussi indiquer à vos utilisateurs comment installer ces bibliothèques ! De plus, même si les utilisateurs peuvent faire confiance aux applets, pour ne pas endommager les données ni intercepter des informations confidentielles, ils peuvent ne pas souhaiter accorder la même confiance à un code utilisant des bibliothèques de méthodes natives. Pour cette raison, nombre d'utilisateurs potentiels peuvent être réticents à utiliser des programmes en langage Java nécessitant du code natif. En plus du problème de sécurité, les bibliothèques natives ne sont pas potentiellement aussi sûres que du code Java, surtout dans le cas d'un langage tel que C ou C++ qui n'offre aucune protection contre le remplacement en mémoire de données par le biais de l'utilisation de pointeur non valide. Il est facile d'écrire des méthodes natives qui vont corrompre la machine virtuelle de Java, compromettre sa sécurité ou perturber le système d'exploitation.

Il est par conséquent conseillé de n'utiliser du code natif qu'en dernier ressort. Si vous avez besoin d'avoir accès à une unité, comme un port série, dans un programme, il peut être nécessaire d'utiliser un code natif. Si vous devez accéder à une portion de code existant, pourquoi ne pas envisager les méthodes natives comme une mesure provisoire pour en fin de compte réaliser le portage du code en

langage Java ? Si vous recherchez l'efficacité, faites un essai d'implémentation de plate-forme Java. Dans la plupart des cas, la vitesse d'exécution avec un compilateur JTI sera suffisante. Un exposé lors de la conférence JavaOne de 1996 l'a clairement démontré. Les concepteurs de la bibliothèque cryptographique de Sun Microsystems ont indiqué qu'une implémentation "plate-forme Java pure" de leurs fonctions cryptographiques était plus que satisfaisante. Il est vrai que le code n'était pas aussi rapide qu'avec une implémentation C, mais cela s'est révélé sans importance. L'implémentation sur plate-forme Java était beaucoup plus rapide que les E/S réseau. Et en fait, c'est là que se trouvait le réel goulot d'étranglement.

En bref, sacrifier la portabilité pour une hypothétique amélioration de la rapidité n'a pas de sens ; n'ayez pas recours au "natif" à moins d'être certain de ne pas avoir d'autre choix.

INFO

Dans ce chapitre, nous décrivons la liaison appelée Java Native Interface (JNI). Une liaison précédente (parfois appelée *interface native brute*) était utilisée avec Java 1.0, et une variante de cette liaison précédente était utilisée par la machine virtuelle Microsoft. Sun Microsystems a assuré les développeurs que la liaison JNI décrite ici ferait partie en permanence de la plate-forme Java, et qu'elle serait prise en charge par toutes les machines virtuelles Java.

Enfin, nous utilisons C comme langage pour les méthodes natives dans ce chapitre, car le langage C est probablement le plus utilisé pour les méthodes natives. En particulier, vous verrez comment faire la correspondance entre les types de données, les noms de fonctionnalités et les appels de fonction Java et ceux de C. Cette correspondance est généralement appelée "C binding" (*liaison C*).

INFO C++

Vous pouvez aussi utiliser C++ au lieu de C pour écrire les méthodes natives. Il y a peu d'avantages : les contrôles de type sont légèrement plus stricts, et l'accès aux fonctions JNI est un peu plus facile. Cependant, JNI ne gère pas de correspondance directe entre les classes de la plate-forme Java et celles de C++.

Appeler une fonction C à partir du langage Java

Supposons que vous ayez une fonction C qui réalise une tâche que vous appréciez, et que pour une raison ou une autre, vous vouliez vous dispenser de la réimplémenter en langage Java. Pour les besoins de la démonstration, nous allons admettre qu'il s'agit de la vénérable et incontournable fonction `printf`. Vous voulez pouvoir appeler `printf` à partir de vos programmes. Le langage Java utilise le mot clé `native` pour une méthode native, et il est évident que vous devrez encapsuler la fonction `printf` dans une classe. Vous pouvez parfaitement écrire quelque chose comme :

```
public class Printf
{
    public native String printf(String s);
}
```

et vous pourrez en fait compiler cette classe, mais quand vous voudrez l'utiliser dans un programme, la machine virtuelle va vous dire qu'elle ne peut pas trouver `printf`, et elle va émettre une erreur

680 *Au cœur de Java 2 - Fonctions avancées*

UnsatisfiedLinkError. L'astuce consiste donc à fournir les informations d'exécution nécessaires pour qu'elle puisse lier cette classe. Comme vous le verrez bientôt, avec le JDK cela implique un processus en trois étapes :

1. Générer un pseudo-code C pour une fonction qui fait la translation entre l'appel Java et la fonction C réelle. Le pseudo-code réalise cette translation en prenant les informations de paramètres de la pile de la machine virtuelle, pour les passer à la fonction C compilée.
2. Créer une bibliothèque partagée spéciale et exporter le pseudo-code à partir de celle-ci.
3. Utiliser une méthode spéciale, appelée `System.loadLibrary` pour demander à l'environnement d'exécution Java de charger la bibliothèque à partir de l'étape 2.

Nous allons maintenant vous montrer comment exécuter ces étapes pour différents exemples, en partant d'un cas spécial d'utilisation triviale de `printf` pour finir avec un exemple réaliste impliquant les fonctions de base de registres pour des fonctions dépendantes de la plate-forme Windows, et qui ne sont de toute évidence pas disponibles directement à partir de la plate-forme Java.

Travailler avec la fonction `printf`

Nous allons commencer par la situation virtuellement la plus simple possible d'utilisation de `printf` : l'appel d'une méthode native qui imprime le message "Hello, Native World". En fait, nous n'allons même pas exploiter les fonctionnalités de formatage de `printf` ! C'est toutefois une excellente façon pour vous de vérifier que votre compilateur C fonctionne comme prévu avant de tenter l'implémentation de méthodes natives plus élaborées.

Vous déclarez d'abord la méthode native dans une classe. Le mot clé `native` avertit le compilateur que la méthode sera définie en externe. Bien entendu, les méthodes natives ne contiennent aucun code en langage Java, et l'en-tête de la méthode est immédiatement suivi du point-virgule de fin. Cela signifie, comme vous l'avez vu dans l'exemple plus haut, que les déclarations de méthodes natives ont un aspect analogue à celui des déclarations de méthodes abstraites.

```
class HelloNative
{
    public native static void greeting();
    . . .
}
```

Dans cet exemple particulier, remarquez que la méthode native est aussi déclarée comme `static`. Les méthodes natives peuvent être statiques et non statiques. Nous commençons par une méthode statique car nous ne voulons pas pour l'instant nous préoccuper du passage de paramètres.

Ecrivez ensuite une fonction C correspondante. Vous devez nommer cette fonction exactement comme l'attend l'environnement d'exécution de Java. C'est-à-dire :

1. Utilisez le nom complet de méthode Java, comme `HelloNative.greeting`.
Si la classe se trouve dans un package, faites précéder du nom du package, comme par exemple `com.horstmann>HelloNative.greeting`.
2. Remplacez chaque point par un caractère de soulignement, et faites précéder du préfixe `Java_`. Par exemple, `Java_HelloNative_greeting` ou `Java_com_horstmann_HelloNative_greeting`.

3. Si le nom de la classe contient des caractères qui ne sont ni des chiffres ni des lettres ASCII, par exemple, `'_'`, `'$'`, ou des caractères Unicode supérieurs à `'\u007F'`, remplacez-les par `_0xxxx`, où `xxxx` est la séquence de quatre chiffres hexadécimaux de la valeur Unicode du caractère.

INFO

Si vous surchargez des méthodes natives, c'est-à-dire si vous fournissez plusieurs méthodes natives avec le même nom, vous devez ajouter un double caractère de soulignement suivi des types d'arguments codés. Le codage des types d'arguments est décrit plus loin dans ce chapitre. Par exemple, si vous avez une méthode native, `greeting`, et une autre méthode native `greeting(int repeat)`, la première s'appellera `Java_HelloNative_greeting__`, et la seconde `Java_HelloNative_greeting__I`.

En réalité, personne ne fait cela manuellement ; exécutez plutôt l'utilitaire `javah`, qui génère automatiquement les noms de fonctions. Pour utiliser `javah`, compilez le fichier source (donné dans l'Exemple 9.3).

```
javac HelloNative.java
```

Appelez ensuite l'utilitaire `javah` pour produire un fichier d'en-tête C. L'exécutable `javah` se trouve dans le répertoire `\jdk\bin`.

```
javah HelloNative
```

L'utilisation de `javah` crée un fichier d'en-tête, `HelloNative.h`, comme dans l'Exemple 9.1.

Exemple 9.1 : *HelloNative.h*

```
/* NE PAS EDITER CE FICHER - il est généré par la machine */
#include <jni.h>
/* En-tête pour la classe HelloNative */

#ifdef _Included_HelloNative
#define _Included_HelloNative
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:      HelloNative
 * Method:     greeting
 * Signature:  ()V
 */
JNIEXPORT void JNICALL Java_HelloNative_greeting
    (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

Comme vous le voyez, ce fichier contient la déclaration d'une fonction `Java_HelloNative_greeting`. (Les chaînes `JNIEXPORT` et `JNICALL` sont définies dans le fichier d'en-tête `jni.h`.)

682 *Au cœur de Java 2 - Fonctions avancées*

Elles indiquent des spécificateurs dépendants du compilateur pour les fonctions exportées qui proviennent d'une bibliothèque chargée dynamiquement.)

Il ne vous reste plus maintenant qu'à copier le prototype de fonction du fichier d'en-tête vers le fichier source, et à fournir le code d'implémentation pour la fonction, comme le montre l'Exemple 9.2.

Exemple 9.2 : HelloNative.c

```
#include "HelloNative.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_HelloNative_greeting
(JNIEnv* env, jclass cl)
{
    printf("Hello, Native World!\n");
}
```

Dans cette simple fonction, ignorez les arguments `env` et `cl`. Vous verrez plus loin leur utilisation.

INFO C++

Vous pouvez utiliser C++ pour implémenter des méthodes natives. Toutefois, vous devez alors déclarer les fonctions qui implémentent les méthodes natives en tant que fonctions "C" externes. Ceci empêche le compilateur C++ de générer du code spécifique au C++. Par exemple :

```
#include "HelloNative.h"
#include <stdio.h>

extern "C"
JNIEXPORT void JNICALL Java_HelloNative_greeting
(JNIEnv* env, jclass cl)
{
    printf("Hello, Native World!\n");
}
```

Compilez ensuite le code C natif dans une bibliothèque chargée dynamiquement. Les détails dépendent de votre compilateur.

Par exemple, avec le compilateur C de Gnu sous Linux, utilisez ces commandes :

```
gcc -fPIC -I jdk/include/ -I jdk/include/linux
-shared -o libHelloNative.so HelloNative.c
```

Avec le compilateur Sun sous environnement Solaris, la commande est :

```
cc -G -I jdk/include -I jdk/include/solaris -o
libHelloNative.so HelloNative.c
```

Avec le compilateur Microsoft C++ sous Windows, la commande utilisée est la suivante :

```
cl -I jdk\include -I jdk\include\win32
-LD HelloNative.c -FeHelloNative.dll
```

Ici, *jdk* désigne le répertoire qui contient le JDK.

ASTUCE

Si vous utilisez le compilateur Microsoft C++ pour compiler les DLL depuis une shell de commande, exécutez d'abord le fichier de lot `vcvars32.net` ou `vsvars32.bat`. Ce fichier configure correctement le compilateur de ligne de commande en définissant le chemin et les variables d'environnement nécessaires au compilateur. Vous le trouverez dans le répertoire `c:\Program Files\Microsoft Visual Studio .NET 2003\Common7\tools` (ou autre monstruosité similaire !).

Vous pouvez également utiliser gratuitement l'environnement de programmation Cygwin, disponible à l'adresse <http://www.cygwin.com>. Il contient le compilateur C de Gnu et des bibliothèques pour la programmation de type UNIX sous Windows. Avec Cygwin, utilisez les commandes

```
gcc --mno-cygwin -D__int64="long long"
-I jdk/include/ -I jdk/include/win32
-shared -Wl, --add-stdcall-alias -o HelloNative.dll HelloNative.c
```

Tapez la commande sur une seule ligne.

INFO

La version Windows du fichier d'en-tête `jni_md.h` contient la déclaration de type

```
typedef __int64 jlong;
```

spécifique au compilateur Microsoft. Si vous utilisez le compilateur Gnu, vous voudrez peut-être modifier ce fichier, par exemple :

```
#ifdef __GNUC__
typedef long long jlong;
#else
typedef __int64 jlong;
#endif
```

Vous pouvez également compiler avec `-D__int64="long long"`, comme le montre l'invocation du compilateur d'exemple.

Enfin, ajoutez un appel à la méthode `System.loadLibrary` dans la classe Java qui définit la méthode native. Ceci permet de s'assurer que la machine virtuelle va charger la bibliothèque avant la première utilisation de la classe. Le moyen le plus facile pour cela consiste à utiliser un bloc d'initialisation statique, comme dans l'Exemple 9.3.

Exemple 9.3 : HelloNative.java

```
class HelloNative
{
    public static native void greeting();

    static
    {
        System.loadLibrary("HelloNative");
    }
}
```

En admettant que vous ayez respecté toutes les étapes ci-dessus, vous êtes maintenant prêt à exécuter l'application `HelloNativeTest` de l'Exemple 9.4.

684 *Au cœur de Java 2 - Fonctions avancées*

Exemple 9.4 : HelloNativeTest.java

```
class HelloNativeTest
{
    public static void main(String[] args)
    {
        HelloNative.greeting();
    }
}
```

Si vous compilez et exécutez ce programme, le message "Hello, Native World!" va s'afficher dans une fenêtre terminal.

INFO

Si vous travaillez sous Linux, ajoutez le répertoire courant au chemin de la bibliothèque. Définissez la variable d'environnement LD_LIBRARY_PATH,

```
export LD_LIBRARY_PATH=.:LD_LIBRARY_PATH
```

ou définissez la propriété système java.library.path system :

```
java -Djava.library.path=. HelloNativeTest
```

Ce n'est évidemment pas très impressionnant en soi. Toutefois, si vous gardez à l'esprit que ce message est généré par la commande C printf et non par un quelconque code en langage Java, vous admettrez que nous avons fait les premiers pas vers un comblement du fossé entre les deux langages !

java.lang.System 1.0

- void loadLibrary(String libname)

Charge la bibliothèque avec le nom donné. La bibliothèque est localisée dans le chemin de recherche de la bibliothèque. La méthode exacte pour la localisation de la bibliothèque dépend du système d'exploitation. Sous Windows, cette méthode recherche d'abord dans le répertoire courant, puis dans les répertoires listés dans la variable d'environnement PATH.

- void load(String filename)

Charge la bibliothèque avec le nom de fichier donné. Si la bibliothèque est introuvable, une erreur UnsatisfiedLinkError est émise.

INFO

Certaines bibliothèques partagées pour code natif doivent exécuter un code d'initialisation. Vous pouvez placer le code d'initialisation dans une méthode JNI_OnLoad. De même, lors d'un arrêt de la machine virtuelle, celle-ci appellera la méthode JNI_OnUnload si vous la fournissez. Les prototypes sont :

```
jint JNI_OnLoad(JavaVM* vm, void* reserved);
void JNI_OnUnload(JavaVM* vm, void* reserved);
```

La méthode JNI_OnLoad doit renvoyer la version minimale de la machine virtuelle qu'elle requiert, par exemple JNI_VERSION_1_1.

Les paramètres numériques et les valeurs renvoyées

Pour le passage de nombres entre C et Java, vous devez savoir quels types correspondent à quels autres. Par exemple, si C possède des types de données appelés `int` et `long`, leur implémentation est dépendante de la plate-forme. Sur certaines plates-formes, les types entiers (`int`) font 16 bits, alors que sur d'autres ils font 32 bits. Sur la plate-forme Java, bien entendu, un type `int` est toujours un entier 32 bits. Pour cette raison, la JNI définit les types `jint`, `jlong`, etc.

Le Tableau 9.1 montre la correspondance entre les types Java et les types C.

Tableau 9.1 : Les types Java et les types C

<i>Langage Java</i>	<i>Langage C</i>	<i>Octets</i>
<code>boolean</code>	<code>jboolean</code>	1
<code>byte</code>	<code>jbyte</code>	1
<code>char</code>	<code>jchar</code>	2
<code>short</code>	<code>jshort</code>	2
<code>int</code>	<code>jint</code>	4
<code>long</code>	<code>jlong</code>	8
<code>float</code>	<code>jfloat</code>	4
<code>double</code>	<code>jdouble</code>	8

Dans le fichier d'en-tête `jni.h`, ces types sont déclarés à l'aide d'instructions `typedef` comme les types équivalents sur la plate-forme cible. Ce fichier d'en-tête définit aussi les constantes `JNI_FALSE` = 0 et `JNI_TRUE` = 1.

Utiliser `printf` pour le formatage de nombres

Jusqu'au JDK 5.0, Java n'avait pas d'analogie directe pour la fonction `printf` du C. Supposons que vous possédiez une ancienne version du JDK et que vous décidiez d'implémenter cette même fonctionnalité en appelant la fonction `printf` de C dans une méthode native.

L'Exemple 9.5 montre une classe appelée `Printf1` qui utilise une méthode native pour imprimer un nombre en virgule flottante avec une largeur de champ et une précision données.

Exemple 9.5 : `Printf1.java`

```
class Printf1
{
    public static native int print(int width, int
        precision, double x);
}
```

686 Au cœur de Java 2 - Fonctions avancées

```

static
{
    System.loadLibrary("Printf1");
}

```

Remarquez que, lors de l'implémentation de la méthode en C, tous les paramètres `int` et `double` deviennent `jint` et `jdouble`, comme le montre l'Exemple 9.6.

Exemple 9.6 : Printf1.c

```

#include "Printf1.h"
#include <stdio.h>

JNIEXPORT jint JNICALL Java_Printf1_print
(JNIEnv* env, jclass cl, jint width, jint precision, jdouble x)
{
    char fmt[30];
    jint ret;
    sprintf(fmt, "%%d.%%df", width, precision);
    ret = printf(fmt, x);
    fflush(stdout);
    return ret;
}

```

La fonction assemble simplement une chaîne de formatage `"%w.pf"` dans la variable `fmt`, puis elle appelle `printf`. Elle renvoie ensuite le nombre de caractères imprimés.

L'Exemple 9.7 montre le programme de test qui fait la démonstration de la classe `Printf1`.

Exemple 9.7 : Printf1Test.java

```

class Printf1Test
{
    public static void main(String[] args)
    {
        int count = Printf1.print(8, 4, 3.14);
        count += Printf1.print(8, 4, count);
        System.out.println();
        for (int i = 0; i < count; i++)
            System.out.print("-");
        System.out.println();
    }
}

```

Les paramètres chaîne

Nous allons ensuite voir comment transférer des chaînes, de et vers des méthodes natives. Comme vous le savez, les chaînes dans le langage Java sont des séquences de points de code UTF-16 tandis que les chaînes C sont des séquences de caractères sur 8 octets terminées par un `null`. La notion de chaîne est donc assez différente dans les deux langages. La JNI a deux jeux de fonctions pour manipuler les chaînes, l'un qui convertit les chaînes Java en séquence d'octets "UTF-8 modifié" et l'autre qui les convertit en tableaux de valeurs UTF-16, c'est-à-dire en tableaux de `jchar`. Les formats UTF-8, "UTF-8 modifié" et UTF-16 ont été vus au Chapitre 12 du Volume 1. Souvenez-vous que le codage "UTF-8 modifié" laisse les caractères ASCII "tels quels", mais tous les autres caractères Unicode sont codés sous la forme de séquences de plusieurs octets.

INFO

Le codage UTF-8 standard et le codage "UTF-8 modifié" ne diffèrent que pour les caractères "supplémentaires" dont le code est supérieur à 0xFFFF. Dans le codage UTF-8 standard, ces caractères sont codés sous forme de séquence de 4 octets. Toutefois, dans le codage "modifié", le caractère est d'abord codé comme une paire de "substituts" dans le codage UTF-16, puis chaque substitut est codé avec UTF-8, ce qui produit un total de 6 octets. C'est assez confus, mais cela existe pour des raisons historiques : la spécification JVM était écrite lorsque Unicode était limité à 16 bits.

Si votre code C utilise déjà le système Unicode, vous utiliserez le second jeu de fonctions de conversion. En revanche, si toutes vos chaînes sont limitées aux caractères ASCII, vous pouvez utiliser les fonctions de conversion "UTF-8 modifié".

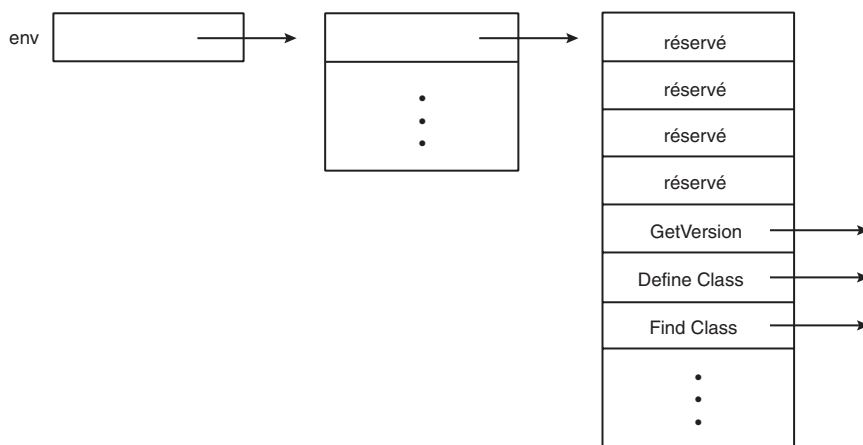
Une méthode native avec un paramètre String reçoit en réalité une valeur d'un type obscur appelé `jstring`. Une méthode native avec une valeur de retour de type String doit renvoyer une valeur de type `jstring`. Les fonctions JNI lisent et construisent ces objets `jstring`. Par exemple, la fonction `NewStringUTF` crée un nouvel objet `jstring` à partir d'un tableau de caractères (`char`) qui contient des caractères codés ASCII ou, plus généralement, des séquences codées en "UTF-8 modifié".

Les fonctions JNI ont une convention d'appel quelque peu curieuse. Voici un appel à la fonction `NewStringUTF`.

```
JNIEXPORT jstring JNICALL Java_HelloNative_getGreeting
(JNIEnv* env, jclass cl)
{
    jstring jstr;
    char greeting[] = "Hello, Native World\n";
    jstr = (*env)->NewStringUTF(env, greeting);
    return jstr;
}
```

Tous les appels aux fonctions JNI utilisent le pointeur `env` qui est le premier argument de toute méthode native. Le pointeur `env` est un pointeur sur une table de pointeurs de fonction (voir Figure 9.1). Vous devez par conséquent prefixer chaque appel à JNI avec `(*env)->` pour réellement déréférencer le pointeur de fonction. De plus, `env` est le premier paramètre de chaque fonction JNI.

Figure 9.1
Le pointeur `env`.



688 *Au cœur de Java 2 - Fonctions avancées*

INFO C++

Il est plus simple d'accéder aux fonctions JNI dans C++. La version C++ de la classe `JNIEnv` possède des fonctions membres en ligne qui se chargent pour vous de la correspondance du pointeur de fonction. Vous pouvez par exemple appeler la fonction `NewStringUTF` de la façon suivante :

```
jstr = env->NewStringUTF(greeting);
```

Notez l'omission du pointeur `JNIEnv` à partir de la liste de paramètres de l'appel.

La fonction `NewStringUTF` vous permet de construire un nouvel objet `jstring`. Pour lire le contenu d'un objet `jstring` existant, utilisez la fonction `GetStringUTFChars`. Cette fonction renvoie un pointeur `const jbyte*` sur les caractères "UTF-8 modifiés" qui décrivent la chaîne de caractères. Remarquez qu'une machine virtuelle spécifique est libre de choisir ce codage de caractères pour sa représentation de chaîne interne ; vous pouvez donc obtenir un pointeur de caractère dans la chaîne Java réelle. Les chaînes Java étant destinées à être immuables, il est très important de considérer `const` sérieusement et de ne pas essayer d'écrire dans ce tableau de caractères. En revanche, si la machine virtuelle utilise des caractères UTF-16 ou UTF-32 pour sa représentation de chaîne interne, cet appel de fonction allouera alors un nouveau bloc de mémoire qui sera rempli avec les équivalents "UTF-8 modifiés".

La machine virtuelle doit savoir quand vous avez fini d'utiliser la chaîne, afin qu'elle puisse récupérer son emplacement. (Le *garbage collector* ou ramasse-miettes s'exécute dans un thread séparé, et il peut interrompre l'exécution de méthodes natives). Vous devez, pour cette raison, appeler la fonction `ReleaseStringUTFChars`.

Vous pouvez aussi fournir votre propre tampon pour contenir les caractères de la chaîne en appelant les méthodes `GetStringRegion` ou `GetStringUTFRegion`.

Enfin, la fonction `GetStringUTFLength` renvoie le nombre de caractères nécessaires au codage "UTF-8 modifié" de la chaîne.

INFO

Vous trouverez l'API JNI à l'adresse <http://java.sun.com/j2se/5.0/docs/guide/jni/spec/functions.html>.

Accès aux chaînes Java à partir du code C

- `jstring NewStringUTF(JNIEnv* env, const char bytes[])`

Renvoie un nouvel objet chaîne Java à partir d'une séquence d'octets "UTF-8 modifié" ou `NULL` si la chaîne ne peut être construite.

Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>bytes</code>	la chaîne UTF terminée par un caractère null

- `jsize GetStringUTFLength(JNIEnv* env, jstring string)`

Renvoie le nombre d'octets requis pour le codage "UTF-8 modifié".

Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>string</code>	un objet chaîne Java

- `const jbyte* GetStringUTFChars(JNIEnv* env, jstring string, jboolean* isCopy)`
Renvoie un pointeur sur le codage "UTF-8 modifié" d'une chaîne, ou NULL si le tableau de caractères ne peut être construit. Le pointeur est valide jusqu'à ce que `ReleaseStringUTFChars` soit appelé.

Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>string</code>	un objet chaîne Java
<code>isCopy</code>	pointe sur un <code>jboolean</code> qui contient <code>JNI_TRUE</code> si une copie est réalisée ; <code>JNI_FALSE</code> sinon

- `void ReleaseStringUTFChars(JNIEnv* env, jstring string, const jbyte bytes[])`
Informe la machine virtuelle que le code natif n'a plus besoin d'accéder à la chaîne Java par l'intermédiaire de bytes.

Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>string</code>	un objet chaîne Java
<code>bytes</code>	un pointeur renvoyé par <code>GetStringUTFChars</code>

- `void GetStringRegion(JNIEnv *env, jstring string, jsize start, jsize length, jchar *buffer)`
Copie une séquence d'octets "UTF-8 modifié" d'une chaîne vers un tampon fourni par l'utilisateur.

Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>string</code>	un objet chaîne Java
<code>start</code>	l'index de départ
<code>length</code>	le nombre de caractères à copier
<code>buffer</code>	le tampon fourni par l'utilisateur

- `void GetStringUTFRegion(JNIEnv *env, jstring string, jsize start, jsize length, jbyte *buffer)`
Copie une séquence de caractères UTF8 d'une chaîne vers un tampon fourni par l'utilisateur. Le tampon doit être suffisamment long pour contenir les octets. Dans le pire des cas, $3 \times \text{length}$ octets sont copiés.

- `jstring NewString(JNIEnv* env, const jchar chars[], jsize length)`
Renvoie un nouvel objet chaîne Java à partir d'une chaîne Unicode, ou NULL si la chaîne ne peut être construite.

Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>chars</code>	la chaîne UTF terminée par un caractère null
<code>length</code>	le nombre de caractères dans la chaîne

- `jsize GetStringLength(JNIEnv* env, jstring string)`
Renvoie le nombre de caractères dans la chaîne

Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>string</code>	un objet chaîne Java

690 Au cœur de Java 2 - Fonctions avancées

- `const jchar* GetStringChars(JNIEnv* env, jstring string, jboolean* isCopy)`
Renvoie un pointeur sur le codage Unicode d'une chaîne, ou NULL si le tableau de caractères ne peut être construit. Le pointeur est valide jusqu'à ce que `ReleaseStringUTFChars` soit appelé.

Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>string</code>	un objet chaîne Java
<code>isCopy</code>	est soit NULL, soit pointe sur un <code>jboolean</code> qui contient JNI_TRUE si une copie est réalisée ; ou JNI_FALSE sinon.

- `void ReleaseStringChars(JNIEnv* env, jstring string, const jchar chars[])`
Informe la machine virtuelle que le code natif n'a plus besoin d'accéder à la chaîne Java par l'intermédiaire de `chars`.

Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>string</code>	un objet chaîne Java
<code>chars</code>	un pointeur renvoyé par <code>GetStringChars</code>

Appeler `sprintf` dans une méthode native

Nous allons utiliser ces fonctions que nous venons de décrire, et écrire une classe qui appelle la fonction C `sprintf`. L'appel de la fonction est montré dans l'Exemple 9.8.

Exemple 9.8 : `Printf2Test.java`

```
class Printf2Test
{
    public static void main(String[] args)
    {
        double price = 44.95;
        double tax = 7.75;
        double amountDue = price * (1 + tax / 100);

        String s = Printf2.sprintf("Montant dû = %8.2f",
                                   amountDue);
        System.out.println(s);
    }
}
```

L'Exemple 9.9 montre la classe avec la méthode native `sprintf`.

Exemple 9.9 : `Printf2.java`

```
class Printf2
{
    public static native String sprintf(String format, double x);

    static
    {
        System.loadLibrary("Printf2");
    }
}
```

La fonction C qui met en forme un nombre en virgule flottante a le prototype :

```
JNIEXPORT jstring JNICALL Java_Printf2_sprint
(JNIEnv* env, jclass cl, jstring format, jdouble x)
```

L'Exemple 9.10 montre le code pour l'implémentation en C. Remarquez les appels : à `GetStringUTFChars` pour lire l'argument format, à `NewStringUTF` pour générer la valeur renvoyée et à `ReleaseStringUTFChars` pour informer la machine virtuelle que l'accès à la chaîne n'est plus requis.

Exemple 9.10 : Printf2.c

```
#include "Printf2.h"
#include <string.h>
#include <stdlib.h>
#include <float.h>

/**
 * @param formate une chaîne contenant un spécificateur de format printf
 * (comme "%8.2f"). Les sous-chaînes "%" sont sautées.
 * @return renvoie un pointeur sur le spécificateur de format (sautant "%")
 * ou NULL s'il n'y avait pas un spécificateur unique de format
 */
char* find_format(const char format[])
{
    char* p;
    char* q;

    p = strchr(format, '%');
    while (p != NULL && *(p + 1) == '%') /* sauter %% */
        p = strchr(p + 2, '%');
    if (p == NULL) return NULL;
    /* vérifier maintenant que % est unique */
    p++;
    q = strchr(p, '%');
    while (q != NULL && *(q + 1) == '%') /* sauter %% */
        q = strchr(q + 2, '%');
    if (q != NULL) return NULL; /* % non unique */
    q = p + strspn(p, "-0+#"); /* sauter les indicateurs */
    q += strspn(q, "0123456789"); /* sauter la largeur de champ */
    if (*q == '.') { q++; q += strspn(q, "0123456789"); }
    /* sauter la précision */
    if (strchr("eEfFgG", *q) == NULL) return NULL;
    /* n'est pas un format en virgule flottante */
    return p;
}

JNIEXPORT jstring JNICALL Java_Printf2_sprint
(JNIEnv* env, jclass cl, jstring format, jdouble x)
{
    const char* cformat;
    char* fmt;
    jstring ret;

    cformat = (*env)->GetStringUTFChars(env, format, NULL);
    fmt = find_format(cformat);
    if (fmt == NULL)
        ret = format;
    else
```

692 *Au cœur de Java 2 - Fonctions avancées*

```

    {
        char* cret;
        int width = atoi(fmt);
        if (width == 0) width = DBL_DIG + 10;
        cret = (char*)malloc(strlen(cformat) + width);
        sprintf(cret, cformat, x);
        ret = (*env)->NewStringUTF(env, cret);
        free(cret);
    }
    (*env)->ReleaseStringUTFChars(env, format, cformat);
    return ret;
}

```

Dans cette fonction, nous avons choisi une gestion d'erreurs simple. Si le code de formatage pour imprimer un nombre en virgule flottante n'a pas la forme `%w.pc`, où `c` est l'un des caractères `e`, `E`, `f`, `g` ou `G`, le nombre n'est tout simplement pas mis en forme. Vous verrez plus tard comment une méthode native peut lancer une exception.

Accéder aux champs

Toutes les méthodes natives que vous avez vues jusqu'à présent étaient statiques, avec des paramètres nombre ou chaîne. Nous allons maintenant voir les méthodes natives qui opèrent sur les objets. En guise d'exercice, nous implémentons une méthode de la classe `Employee` que nous avons vue au Chapitre 4 du Volume 1, en utilisant une méthode native. Là encore, ce n'est pas quelque chose que vous feriez habituellement, mais cela illustre la façon d'accéder aux champs à partir d'une méthode native, en cas de besoin.

Accéder aux champs d'instance

Pour voir comment accéder aux champs d'instance à partir d'une méthode native, nous réimplémenterons la méthode `raiseSalary`. En Java, le code était simple :

```

public void raiseSalary(double byPercent)
{
    salary *= 1 + byPercent / 100;
}

```

Nous allons la réécrire en tant que méthode native. Contrairement aux exemples précédents de méthodes natives, il ne s'agit pas d'une méthode statique. L'exécution de `javah` donne le prototype suivant :

```

JNIEXPORT void JNICALL Java_Employee_raiseSalary
    (JNIEnv *, jobject, jdouble);

```

Remarquez le deuxième argument. Il n'est plus du type `jclass`, mais du type `jobject`. C'est en fait l'équivalent de la référence à `this`. Les méthodes statiques obtiennent une référence à la classe, alors que les méthodes non statiques obtiennent une référence à l'objet argument `this` implicite.

Nous accédons maintenant au champ `salary` de l'argument implicite. Dans la liaison "brute" Java-à-C de Java 1.0, c'était simple : un programmeur pouvait accéder directement aux champs de données Objet. Toutefois, l'accès direct requiert que toutes les machines virtuelles exposent leur représentation interne de données. Pour cette raison, l'interface JNI nécessite que les programmeurs extraient et définissent les valeurs des champs de données en appelant des fonctions JNI spéciales.

En ce qui nous concerne, nous devons utiliser les fonctions `GetDoubleField` et `SetDoubleField`, car `salary` est du type `double`. Il existe d'autres fonctions : `GetIntField/SetIntField`, `GetObjectField/SetObjectField`, etc., pour les autres types de champs. La syntaxe générale est la suivante :

```
x = (*env)->GetXxxField(env, this_obj, fieldID);
(*env)->SetXxxField(env, this_obj, fieldID, x);
```

Ici, `class` est une valeur qui représente un objet Java du type `Class`, et `fieldID` est une valeur d'un type spécial, `jfieldID`, qui identifie un champ dans une structure ; `Xxx` représente un type de données Java (`Object`, `Boolean`, `Byte`, etc.). Il y a deux façons d'obtenir l'objet `class`. La fonction `GetObjectClass` renvoie la classe de n'importe quel objet. Par exemple :

```
jclass class_Employee = (*env)->GetObjectClass(env, this_obj);
```

La fonction `FindClass` vous permet de spécifier le nom de classe comme une chaîne (curieusement, avec / au lieu de points comme séparateurs de nom de package).

```
jclass class_String
= (*env)->FindClass(env, "java/lang/String");
```

Utilisez la fonction `GetFieldID` pour extraire `fieldID`. Vous devez fournir le nom du champ et sa signature, un codage de son type. Voici par exemple le code pour obtenir l'ID de champ (`fieldID`) du champ `salary`.

```
jfieldID id_salary
= (*env)->GetFieldID(env, class_Employee, "salary", "D");
```

"D" signifie un type `double`. Vous étudierez les règles complètes de codage des signatures dans la prochaine section.

L'accès à un champ de données vous paraît peut-être bien compliqué. Comme les concepteurs de JNI ne voulaient pas exposer directement les champs de données, ils ont dû prévoir des fonctions permettant d'extraire et de définir les valeurs de champ. Pour réduire le coût de ces fonctions, le calcul de l'ID de champ à partir du nom de champ — qui constitue l'étape la plus coûteuse — a été regroupé dans une étape séparée. C'est-à-dire que si vous répétez l'extraction et la définition de la valeur d'un champ particulier plusieurs fois, vous ne subissez qu'une seule fois le coût du calcul de l'identifiant de champ.

Récapitulons. Le code suivant réimplémente la méthode `raiseSalary` comme méthode native.

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary
(JNIEnv* env, jobject this_obj, jdouble byPercent)
{
    /* extraire la classe */
    jclass class_Employee = (*env)->GetObjectClass(env,
        this_obj);

    /* extraire l'ID de champ */
    jfieldID id_salary = (*env)->GetFieldID(env, class_Employee,
        "salary", "D");
```

694 *Au cœur de Java 2 - Fonctions avancées*

```

/* extraire la valeur de champ */
jdouble salary = (*env)->GetDoubleField(env, this_obj,
    id_salary);

salary *= 1 + byPercent / 100;

/* définir la valeur de champ */
(*env)->SetDoubleField(env, this_obj, id_salary, salary);
}

```

ATTENTION

Les références de classe ne sont valides que jusqu'au retour de la méthode native. Vous ne pouvez donc pas mettre en cache les valeurs de retour de `GetObjectClass` dans votre code. Ne stockez pas en réserve une référence de classe pour réutilisation dans un futur appel de méthode. Vous devez appeler `GetObjectClass` chaque fois que la méthode native s'exécute. Si cela n'est pas envisageable, vous pouvez verrouiller la référence avec un appel à `NewGlobalRef` :

```

static jclass class_X = 0;
static jfieldID id_a;
. . .
if (class_X == 0)
{
    jclass cx = (*env)->GetObjectClass(env, obj);
    class_X = (*env)->NewGlobalRef(env, cx);
    id_a = (*env)->GetFieldID(env, cls, "a", ". . .");
}

```

Vous pouvez alors utiliser la référence de classe et les ID de champ dans des appels ultérieurs. Lorsque vous avez fini d'utiliser la classe, n'oubliez pas d'appeler

```
(*env)->DeleteGlobalRef(env, class_X);
```

Les Exemples 9.11 et 9.12 montrent le code Java pour un programme de test et la classe `Employee`. L'Exemple 9.13 contient le code C pour la méthode native `raiseSalary`.

Exemple 9.11 : *EmployeeTest.java*

```

public class EmployeeTest
{
    public static void main(String[] args)
    {
        Employee[] staff = new Employee[3];

        staff[0] = new Employee("Harry Hacker", 35000);
        staff[1] = new Employee("Carl Cracker", 75000);
        staff[2] = new Employee("Tony Tester", 38000);

        int i;
        for (Employee e : staff) e.raiseSalary(5);
        for (Employee e : staff) e.print();
    }
}

```

Exemple 9.12 : *Employee.java*

```
public class Employee
{
    public Employee(String n, double s)
    {
        name = n;
        salary = s;
    }

    public native void raiseSalary(double byPercent);

    public void print()
    {
        System.out.println(name + " " + salary);
    }

    private String name;
    private double salary;

    static
    {
        System.loadLibrary("Employee");
    }
}
```

Exemple 9.13 : *Employee.c*

```
#include "Employee.h"

#include <stdio.h>

JNIEXPORT void JNICALL Java_Employee_raiseSalary
(JNIEnv* env, jobject this_obj, jdouble byPercent)
{
    /* extraire la classe */
    jclass class_Employee = (*env)->GetObjectClass(env,
        this_obj);

    /* extraire l'ID de champ */
    jfieldID id_salary = (*env)->GetFieldID(env,
        class_Employee, "salary", "D");

    /* extraire la valeur de champ */
    jdouble salary = (*env)->GetDoubleField(env, this_obj,
        id_salary);

    salary *= 1 + byPercent / 100;

    /* définir la valeur de champ */
    (*env)->SetDoubleField(env, this_obj, id_salary, salary);
}
```

696 Au cœur de Java 2 - Fonctions avancées

Accès aux champs Objet

- `jfieldID GetFieldID(JNIEnv *env, jclass cl, const char name[], const char sig[])`
Renvoie l'identifiant d'un champ dans une classe.
Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>cl</code>	l'objet classe
<code>name</code>	le nom du champ
<code>sig</code>	la signature du champ codée
- `Xxx GetXxxField(JNIEnv *env, jobject obj, jfieldID id)`
Renvoie la valeur d'un champ. Le type de champ `Xxx` est l'un parmi `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` ou `Double`.
Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>obj</code>	l'objet dont le champ est renvoyé
<code>id</code>	l'identifiant du champ
- `void SetXxxField(JNIEnv *env, jobject obj, jfieldID id, Xxx value)`
Affecte une nouvelle valeur au champ. Le type de champ `Xxx` est l'un parmi `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` ou `Double`.
Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>obj</code>	l'objet dont le champ est défini
<code>id</code>	l'identifiant du champ
<code>value</code>	la nouvelle valeur du champ
- `jfieldID GetStaticFieldID(JNIEnv *env, jclass cl, const char name[], const char sig[])`
Renvoie l'identifiant d'un champ statique dans une classe.
Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>cl</code>	l'objet classe
<code>name</code>	le nom du champ
<code>sig</code>	la signature du champ codée
- `Xxx GetStaticXxxField(JNIEnv *env, jclass cl, jfieldID id)`
Renvoie la valeur d'un champ statique. Le type de champ `Xxx` est l'un parmi `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` ou `Double`.
Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>cl</code>	l'objet classe dont le champ statique est défini
<code>id</code>	l'identifiant du champ
- `void SetStaticXxxField(JNIEnv *env, jclass cl, jfieldID id, Xxx value)`
Affecte une nouvelle valeur à un champ statique. Le type de champ `Xxx` est l'un parmi `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` ou `Double`.
Paramètres :

<code>env</code>	le pointeur de l'interface JNI

cl	l'objet classe dont le champ statique est défini
id	l'identifiant du champ
value	la nouvelle valeur du champ

Accéder aux champs statiques

L'accès aux champs statiques est analogue à l'accès aux champs non statiques. Vous utilisez les fonctions `GetStaticFieldID` et `GetStaticXxxField/SetStaticXxxField`. Elles fonctionnent presque comme leur contrepartie non statique. Il n'y a que deux différences.

- Puisque vous n'avez pas d'objet, vous devez utiliser `FindClass` au lieu de `GetObjectClass` pour obtenir la référence de classe.
- Vous fournissez la classe et non l'objet d'instance lorsque vous accédez au champ.

Voici par exemple comment vous pouvez obtenir une référence à `System.out` :

```
/* extraire la classe */
jclass class_System = (*env)->FindClass(env,
    "java/lang/System");

/* extraire l'ID de champ */
jfieldID id_out = (*env)->GetStaticFieldID(env,
    class_System, "out", "Ljava/io/PrintStream;");

/* extraire la valeur de champ */
jobject obj_out = (*env)->GetStaticObjectField(env,
    class_System, id_out);
```

Coder les signatures

Pour accéder aux champs d'instance et appeler les méthodes définies en langage Java, vous devez apprendre les règles de "*mutilation*" des noms des types de données et des signatures des méthodes. (Une signature de méthode décrit les paramètres de la méthode et le type renvoyé.) Le schéma de codage est le suivant :

B	byte (octet)
C	char (caractère)
D	double
F	float (flottant)
I	int (entier)
J	long
L <i>nomclasse</i> ;	un type de classe
S	short (entier court)
V	void (vide)
Z	boolean (booléen)

698 *Au cœur de Java 2 - Fonctions avancées*

Notez que le point-virgule à la fin de l'expression L est le caractère de fin de l'expression de type, et non un séparateur entre paramètres. Par exemple, le constructeur :

```
Employee(java.lang.String, double, java.util.Date)
```

a une signature :

```
"(Ljava/lang/String;DLjava/util/Date;)V"
```

Remarquez qu'il n'y a pas de séparateur entre le D et Ljava/util/Date;.

Remarquez aussi que, dans ce schéma de codage, vous devez utiliser / au lieu de . pour séparer les noms de package et de classe.

Pour décrire un type tableau, utilisez un caractère [. Par exemple, un tableau de chaîne s'écrit

```
[Ljava/lang/String;
```

Un float[][] est abrégé en

```
[[F
```

Pour la signature complète d'une méthode, vous répertoriez les types de paramètres à l'intérieur d'une paire de parenthèses, puis vous indiquez le type renvoyé. Par exemple, une méthode recevant deux entiers et renvoyant un entier est codée de la façon suivante :

```
(II)I
```

La méthode d'impression que nous avons utilisée dans l'exemple précédent a une signature mutilée de :

```
(Ljava/lang/String;)V
```

C'est-à-dire que la méthode reçoit une chaîne et renvoie void.

ASTUCE

Vous pouvez utiliser la commande javap avec l'option -s pour générer les signatures de méthode à partir des fichiers de classe. Si vous exécutez par exemple :

```
javap -s -private Employee
```

vous obtenez la sortie suivante, affichant les signatures de tous les champs et méthodes.

```
Compiled from "Employee.java"
public class Employee extends java.lang.Object
{
    private java.lang.String name;
        Signature: Ljava/lang/String;
    private double salary;
        Signature: D
    public Employee(java.lang.String, double);
        Signature: (Ljava/lang/String;D)V
    public native void raiseSalary(double);
        Signature: (D)V
    public void print();
        Signature: ()V
    static {};
        Signature: ()V
}
```

INFO

Il n'y a pas de raison rationnelle à l'obligation pour les programmeurs d'utiliser ce schéma mutilant pour la description des signatures. Les concepteurs de l'appel natif auraient pu tout aussi facilement écrire une fonction lisant les signatures dans le style du langage Java, comme `void(int, java.lang.String)`, et les coder en n'importe quelle représentation interne de leur choix. Disons que l'utilisation des signatures *mutilées* vous permet de prendre part à la magie de la programmation proche de la machine virtuelle.

Appeler les méthodes Java

Bien entendu, des fonctions en langage Java peuvent appeler des fonctions C ; c'est précisément le but des méthodes natives. Est-ce que l'inverse est possible ? Quel en serait l'intérêt ? La réponse est qu'il arrive souvent qu'une méthode native ait besoin de demander un service à un objet qui lui a été passé. Nous allons tout d'abord vous montrer comment procéder pour les méthodes non statiques, puis nous verrons les méthodes statiques.

Les méthodes non statiques

Comme exemple d'appel d'une méthode Java à partir de code natif, nous allons améliorer la classe `Printf` et ajouter une méthode qui fonctionne de la même façon que la fonction C `fprintf`. C'est-à-dire qu'elle doit pouvoir imprimer une chaîne sur un objet `PrintWriter` arbitraire.

```
class Printf3
{
    public native static void fprintf(PrintWriter out,
        String s, double x);
    . . .
}
```

Nous assemblons tout d'abord la chaîne à imprimer dans un objet `String` `str`, comme dans la méthode `sprint` que nous avons déjà implémentée. Puis nous appelons la méthode `print` de la classe `PrintWriter` à partir de la fonction C qui implémente la méthode native.

Vous pouvez appeler n'importe quelle méthode Java à partir de C en utilisant l'appel de fonction :

```
(*env)->CallXxxMethod(env, implicit parameter, methodID,
    explicit parameters)
```

Remplacez `Xxx` par `Void`, `Int`, `Object`, etc., en fonction du type renvoyé par la méthode. Tout comme vous avez besoin d'un ID de champ pour accéder au champ d'un objet, vous avez besoin d'un ID de méthode pour appeler une méthode. Vous obtenez un ID de méthode en appelant la fonction `JNI GetMethodID` et en fournissant la classe, le nom de la méthode et la signature de la méthode.

Dans notre exemple, nous voulons obtenir l'ID de la méthode `print` de la classe `PrintWriter`. Comme vous l'avez vu au Chapitre 12 du Volume 1, la classe `PrintWriter` possède plusieurs méthodes surchargées, appelées `print`. Pour cette raison, vous devez aussi fournir une chaîne décrivant les paramètres et les valeurs renvoyées pour la fonction spécifique que vous voulez utiliser. Par exemple, nous voulons utiliser `void print(java.lang.String)`. Comme décrit dans la section précédente, nous devons maintenant "mutiler" la signature pour obtenir la chaîne `"(Ljava/lang/string;)V"`.

700 *Au cœur de Java 2 - Fonctions avancées*

Voici le code complet pour réaliser l'appel à la méthode :

1. en obtenant la classe du paramètre implicite ;
2. en obtenant l'ID de méthode ;
3. en faisant l'appel.

```
/* extraire la classe */
class_PrintWriter = (*env)->GetObjectClass(env, out);

/* extraire l'ID de méthode */
id_print = (*env)->GetMethodID(env, class_PrintWriter,
    "print", "(Ljava/lang/String;)V");

/* appeler la méthode */
(*env)->CallVoidMethod(env, out, id_print, str);
```

Les Exemples 9.14 et 9.15 montrent le code Java pour un programme de test et la classe `Printf3`. L'Exemple 9.16 contient le code C de la méthode native `fprint`.

INFO

Les ID numériques de méthodes et de champs sont semblables, sur le plan conceptuel, aux objets `Method` et `Field` dans l'API de réflexion. Vous pouvez les convertir mutuellement à l'aide des fonctions suivantes :

```
jobject ToReflectedMethod(JNIEnv* env, jclass class,
    jmethodID methodID); // renvoie l'objet Method
methodID FromReflectedMethod(JNIEnv* env, jobject method);
jobject ToReflectedField(JNIEnv* env, jclass class,
    jfieldID fieldID); // renvoie l'objet Field
fieldID FromReflectedField(JNIEnv* env, jobject field);
```

Les méthodes statiques

L'appel de méthodes statiques à partir de méthodes natives est analogue à l'appel de méthodes non statiques. Il existe deux différences.

- Vous utilisez les fonctions `GetStaticMethodID` et `CallStaticXxxMethod`.
- Vous fournissez un objet classe, et non un objet paramètre implicite, lors de l'invocation de la méthode.

A titre d'exemple, nous allons réaliser l'appel à la méthode statique

```
System.getProperty("java.class.path")
```

à partir d'une méthode native. La valeur renvoyée par cet appel est une chaîne qui donne le chemin de la classe courante.

Nous devons tout d'abord rechercher la classe à utiliser. Puisque nous n'avons pas d'objet de la classe `System` facilement disponible, nous utilisons `FindClass` plutôt que `GetObjectClass`.

```
jclass class_System = (*env)->FindClass(env, "java/lang/System");
```

Nous avons ensuite besoin d'un ID de la méthode statique `getProperty`. La signature codée de cette méthode est :

```
"(Ljava/lang/String;)Ljava/lang/String;"
```

puisque à la fois le paramètre et la valeur renvoyée sont une chaîne. Nous obtenons donc l'ID de la méthode de la façon suivante :

```
jmethodID id_getProperty = (*env)->GetStaticMethodID(env,
    class_System, "getProperty",
    "(Ljava/lang/String;)Ljava/lang/String;");
```

Enfin, nous pouvons réaliser l'appel. Remarquez que l'objet classe est passé à la fonction `CallStaticObjectMethod`.

```
jobject obj_ret = (*env)->CallStaticObjectMethod(env,
    class_System, id_getProperty,
    (*env)->NewStringUTF(env, "java.class.path"));
```

La valeur renvoyée par cette méthode est du type `jobject`. Si nous voulons la manipuler comme une chaîne, nous devons la convertir en `jstring`:

```
jstring str_ret = (jstring)obj_ret;
```

INFO C++

En C, les types `jstring` et `jclass`, ainsi que les types tableau qui seront vus plus loin, sont tous des types équivalents à `jobject`. Le transtypage dans l'exemple précédent n'est donc pas strictement nécessaire en C. Mais en C++, ces types sont définis comme pointeurs sur des "classes fictives" qui ont la hiérarchie d'héritage correcte. Par exemple, l'affectation d'un `jstring` à un `jobject` est légal sans transtypage en C++, mais l'inverse nécessite une conversion de type.

Les constructeurs

Une méthode native peut créer un nouvel objet Java en invoquant son constructeur. Vous invoquez le constructeur en appelant la fonction `NewObject`.

```
jobject obj_new = (*env)->NewObject(env, class, methodID,
    construction parameters);
```

Vous obtenez l'ID de méthode nécessaire à cet appel à l'aide de la fonction `GetMethodID` en spécifiant le nom de méthode comme `"<init>"` et la signature codée du constructeur (avec une valeur renvoyée de type `void`). Voici par exemple comment une méthode native peut créer un objet `FileOutputStream`.

```
const char[] fileName = ". . .";
jstring str_fileName = (*env)->NewStringUTF(env, fileName);
jclass class_FileOutputStream = (*env)->FindClass(env,
    "java/io/FileOutputStream");
jmethodID id_FileOutputStream = (*env)->GetMethodID(env,
    class_FileOutputStream, "<init>", "(Ljava/lang/String;)V");
jobject obj_stream = (*env)->NewObject(env,
    class_FileOutputStream, id_FileOutputStream, str_fileName);
```

Remarquez que la signature du constructeur prend un paramètre de type `java.lang.String` et que la valeur renvoyée est de type `void`.

Autres possibilités d'invocation de méthodes

Plusieurs variantes des fonctions JNI appellent une méthode Java à partir de code natif. Elles ne sont pas aussi importantes que les fonctions que nous avons déjà vues, mais elles sont occasionnellement utiles.

Les fonctions `CallNonvirtualXxxMethod` reçoivent un argument implicite, un ID de méthode, un objet classe (qui doit correspondre à une superclasse de l'argument implicite), et des arguments explicites. La fonction appelle la version de la méthode dans la classe spécifiée, outrepassant le mécanisme normal de répartition dynamique.

Toutes les fonctions d'appel ont des versions avec les suffixes "A" et "V" qui reçoivent les paramètres explicites dans un tableau ou une `va_list` (comme défini dans le fichier d'en-tête C `stdarg.h`).

Exemple 9.14 : `Printf3Test.java`

```
import java.io.*;

class Printf3Test
{
    public static void main(String[] args)
    {
        double price = 44.95;
        double tax = 7.75;
        double amountDue = price * (1 + tax / 100);
        PrintWriter out = new PrintWriter(System.out);
        Printf3.fprint(out,
            "Montant dû = %8.2f\n", amountDue);
        out.flush();
    }
}
```

Exemple 9.15 : `Printf3.java`

```
import java.io.*;

class Printf3
{
    public static native void fprint(PrintWriter out,
        String format, double x);

    static
    {
        System.loadLibrary("Printf3");
    }
}
```

Exemple 9.16 : `Printf3.c`

```
#include "Printf3.h"
#include <string.h>
#include <stdlib.h>
#include <float.h>

/**
 * @param format une chaîne contenant un spécificateur de format printf
 * (comme "%8.2f"). Les sous-chaînes "%" sont sautées.
```

```

    @return un pointeur sur le spécificateur de format (sautant "%")
    ou NULL s'il n'y avait pas un spécificateur unique de format
    */
char* find_format(const char format[])
{
    char* p;
    char* q;

    p = strchr(format, '%');
    while (p != NULL && *(p + 1) == '%') /* sauter %% */
        p = strchr(p + 2, '%');
    if (p == NULL) return NULL;
    /* vérifier maintenant que % est unique */
    p++;
    q = strchr(p, '%');
    while (q != NULL && *(q + 1) == '%') /* sauter %% */
        q = strchr(q + 2, '%');
    if (q != NULL) return NULL; /* % non unique */
    q = p + strspn(p, "-0+#"); /* sauter les indicateurs */
    q += strspn(q, "0123456789"); /* sauter la largeur de champ */
    if (*q == '.') { q++; q += strspn(q, "0123456789"); }
    /* sauter la précision */
    if (strchr("eEfFgG", *q) == NULL) return NULL;
    /* n'est pas un format virgule flottante */
    return p;
}

JNIEXPORT void JNICALL Java_Printf3_fprint
(JNIEnv* env, jclass cl, jobject out, jstring format,
 jdouble x)
{
    const char* cformat;
    char* fmt;
    jstring str;
    jclass class_PrintWriter;
    jmethodID id_print;

    cformat = (*env)->GetStringUTFChars(env, format, NULL);
    fmt = find_format(cformat);
    if (fmt == NULL)
        str = format;
    else
    {
        char* cstr;
        int width = atoi(fmt);
        if (width == 0) width = DBL_DIG + 10;
        cstr = (char*)malloc(strlen(cformat) + width);
        sprintf(cstr, cformat, x);
        str = (*env)->NewStringUTF(env, cstr);
        free(cstr);
    }
    (*env)->ReleaseStringUTFChars(env, format, cformat);

    /* appeler maintenant ps.print(str) */
}

```

704 Au cœur de Java 2 - Fonctions avancées

```

/* extraire la classe */
class_PrintWriter = (*env)->GetObjectClass(env, out);

/* extraire l'ID de méthode */
id_print = (*env)->GetMethodID(env, class_PrintWriter,
    "print", "(Ljava/lang/String;)V");

/* appeler la méthode */
(*env)->CallVoidMethod(env, out, id_print, str);
}

```

Exécution des méthodes Java à partir du code C

- `jmethodID GetMethodID(JNIEnv *env, jclass cl, const char name[], const char sig[])`

Renvoie l'identifiant d'une méthode dans une classe.

<i>Paramètres :</i>	<code>env</code>	le pointeur de l'interface JNI
	<code>cl</code>	l'objet classe
	<code>name</code>	le nom de la méthode
	<code>sig</code>	la signature de méthode codée

- `Xxx CallXxxMethod(JNIEnv *env, jobject obj, jmethodID id, args)`
- `Xxx CallXxxMethodA(JNIEnv *env, jobject obj, jmethodID id, jvalue args[])`
- `Xxx CallXxxMethodV(JNIEnv *env, jobject obj, jmethodID id, va_list args)`

Appellent une méthode. Le type de retour `Xxx` peut être `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` ou `Double`. La première fonction a un nombre variable d'arguments ; ajoutez simplement les paramètres de la méthode après l'ID de la méthode. La deuxième fonction reçoit les arguments de la méthode dans un tableau de `jvalue`, où `jvalue` est une union définie de la façon suivante :

```

typedef union jvalue
{
    jboolean z;
    jbyte b;
    jchar c;
    jshort s;
    jint i;
    jlong j;
    jfloat f;
    jdouble d;
    jobject l;
} jvalue;

```

La troisième fonction reçoit les paramètres de la méthode dans une `va_list`, comme défini dans le fichier d'en-tête C `stdarg.h`.

<i>Paramètres :</i>	<code>env</code>	le pointeur de l'interface JNI
	<code>obj</code>	l'argument implicite de la méthode

id l'identifiant de la méthode
args les arguments de la méthode

- `Xxx CallNonvirtualXxxMethod(JNIEnv *env, jobject obj, jclass cl, jmethodID id, args)`
- `Xxx CallNonvirtualXxxMethodA(JNIEnv *env, jobject obj, jclass cl, jmethodID id, jvalue args[])`
- `Xxx CallNonvirtualXxxMethodV(JNIEnv *env, jobject obj, jclass cl, jmethodID id, va_list args)`

Appellent une méthode, outrepassant la répartition dynamique. Le type de retour `Xxx` peut être `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` ou `Double`. La première fonction a un nombre variable d'arguments ; ajoutez simplement les paramètres de la méthode après l'ID de la méthode. La deuxième fonction reçoit les arguments de la méthode dans un tableau de `jvalue`. La troisième fonction reçoit les paramètres de la méthode dans une `va_list`, comme défini dans le fichier d'en-tête `C stdarg.h`.

Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>obj</code>	l'argument implicite de la méthode
<code>cl</code>	la classe dont l'implémentation de la méthode doit être appelée
<code>id</code>	l'identifiant de la méthode
<code>args</code>	les arguments de la méthode

- `jmethodID GetStaticMethodID(JNIEnv *env, jclass cl, const char name[], const char sig[])`

Renvoie l'identifiant d'une méthode statique dans une classe.

Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>cl</code>	l'objet classe
<code>name</code>	le nom de la méthode
<code>sig</code>	la signature de méthode codée

- `Xxx CallStaticXxxMethod(JNIEnv *env, jclass cl, jmethodID id, args)`
- `Xxx CallStaticXxxMethodA(JNIEnv *env, jclass cl, jmethodID id, jvalue args[])`
- `Xxx CallStaticXxxMethodV(JNIEnv *env, jclass cl, jmethodID id, va_list args)`

Appellent une méthode statique. Le type de retour `Xxx` peut être `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` ou `Double`. La première fonction a un nombre variable d'arguments ; ajoutez simplement les paramètres de la méthode après l'ID de la méthode. La deuxième fonction reçoit les arguments de la méthode dans un tableau de `jvalue`. La troisième fonction reçoit les paramètres de la méthode dans une `va_list`, comme défini dans le fichier d'en-tête `C stdarg.h`.

Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>cl</code>	la classe de la méthode statique
<code>id</code>	l'identifiant de la méthode
<code>args</code>	les arguments de la méthode

706 Au cœur de Java 2 - Fonctions avancées

- `jobject NewObject(JNIEnv *env, jclass cl, jmethodID id, args)`
- `jobject NewObjectA(JNIEnv *env, jclass cl, jmethodID id, jvalue args[])`
- `jobject NewObjectV(JNIEnv *env, jclass cl, jmethodID id, va_list args)`

Appellent un constructeur. L'ID de méthode est obtenu à partir de `GetMethodID` avec un nom de méthode "<init>" et un type renvoyé `void`. La première fonction a un nombre variable d'arguments ; ajoutez simplement les paramètres de la méthode après l'ID de la méthode. La deuxième fonction reçoit les arguments de la méthode dans un tableau de `jvalue`. La troisième fonction reçoit les paramètres de la méthode dans une `va_list`, comme défini dans le fichier d'en-tête `C stdarg.h`.

<i>Paramètres :</i>	<code>env</code>	le pointeur de l'interface JNI
	<code>cl</code>	la classe à instancier
	<code>id</code>	l'identifiant de la méthode du constructeur
	<code>args</code>	les arguments du constructeur

Accéder aux éléments de tableaux

Tous les types de tableaux du langage Java ont des types C correspondants, comme le montre le Tableau 9.2.

Tableau 9.2 : Correspondance entre les types de tableaux dans Java et dans C

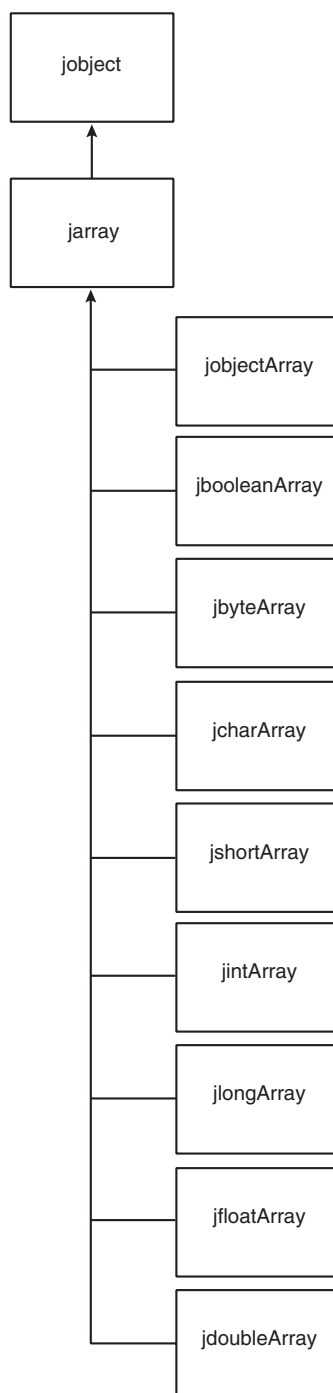
Type Java	Type C
<code>boolean[]</code>	<code>jbooleanArray</code>
<code>byte[]</code>	<code>jbyteArray</code>
<code>char[]</code>	<code>jcharArray</code>
<code>int[]</code>	<code>jintArray</code>
<code>short[]</code>	<code>jshortArray</code>
<code>long[]</code>	<code>jlongArray</code>
<code>float[]</code>	<code>jfloatArray</code>
<code>double[]</code>	<code>jdoubleArray</code>
<code>Object[]</code>	<code>jobjectArray</code>

Le type `jarray` indique un tableau générique.

INFO C++

En C, tous ces types de tableaux sont en réalité des types synonymes de `jobject`. En C++, toutefois, ils sont organisés dans la hiérarchie d'héritage de la Figure 9.2.

Figure 9.2
Hiérarchie d'héritage
des types de tableaux.



708 *Au cœur de Java 2 - Fonctions avancées*

La fonction `GetArrayLength` renvoie la longueur d'un tableau.

```
jarray array = ...;
jsize length = (*env)->GetArrayLength(env, array);
```

La façon dont vous accédez aux éléments dans le tableau varie selon que le tableau stocke des objets ou un type primitif (`bool`, `char` ou un type numérique). Vous accédez aux éléments dans un tableau d'objets à l'aide des méthodes `GetObjectArrayElement` et `SetObjectArrayElement`.

```
jobjectArray array = ...;
int i, j;
jobject x = (*env)->GetObjectArrayElement(env, array, i);
(*env)->SetObjectArrayElement(env, array, j, x);
```

Bien que simple, cette approche est manifestement inefficace ; vous devez pouvoir accéder directement aux éléments du tableau, surtout si vous faites des calculs de vecteur et de matrice.

La fonction `GetXxxArrayElements` renvoie un pointeur C sur l'élément de début du tableau. Comme avec les chaînes ordinaires, vous ne devez pas oublier d'appeler la fonction `ReleaseXxxArrayElements` correspondante pour indiquer à la machine virtuelle que vous n'avez plus besoin de ce pointeur. Ici, le type `Xxx` doit être un type primitif, c'est-à-dire pas un type `Object`. Vous pouvez alors lire et écrire directement dans les éléments du tableau. Toutefois, puisque le pointeur peut pointer sur une copie, toutes les modifications que vous pouvez apporter ne sont censées être réfléchies dans le tableau original que lorsque vous appelez la fonction `ReleaseXxxArrayElements` correspondante !

INFO

Vous pouvez savoir si un tableau est une copie en passant un pointeur sur une variable `jboolean` comme troisième paramètre d'une méthode `GetXxxArrayElements`. La variable contient `JNI_TRUE` si le tableau est une copie. Si cette information est sans intérêt pour vous, passez simplement un pointeur `NULL`.

Voici un exemple de code qui multiplie tous les éléments dans un tableau de valeurs double par une constante. Nous obtenons un pointeur C `a` dans le tableau Java, puis accédons aux éléments individuels par `a[i]`.

```
jdoubleArray array_a = ...;
double scaleFactor = ...;
double* a = (*env)->GetDoubleArrayElements(env, array_a, NULL);
for (i = 0; i < (*env)->GetArrayLength(env, array_a); i++)
    a[i] = a[i] * scaleFactor;
(*env)->ReleaseDoubleArrayElements(env, array_a, a, 0);
```

Le fait que la machine virtuelle copie ou non réellement le tableau dépend de la façon dont elle alloue les tableaux et récupère les emplacements libérés. Certains ramasse-miettes "de copie" déplacent les objets de façon routinière et mettent à jour les références d'objet. Cette stratégie n'est pas compatible avec "l'épinglage" d'un tableau à un emplacement particulier, car le ramasse-miettes ne peut pas mettre à jour les valeurs de pointeur en code natif.

INFO

Dans l'implémentation Sun JVM (*machine virtuelle Java Sun*), les tableaux booléens sont représentés comme des tableaux compactés de mots de 32 bits. La méthode `GetBooleanArrayElements` les copie dans des tableaux de valeurs `jboolean` non compactés.

Pour accéder à quelques éléments seulement d'un tableau volumineux, utilisez les méthodes `GetXxxArrayRegion` et `SetXxxArrayRegion` qui copient une plage d'éléments à partir du tableau Java vers un Tableau C et inversement.

Vous pouvez créer de nouveaux tableaux Java dans des méthodes natives à l'aide de la fonction `NewXxxArray`. Pour créer un nouveau tableau d'objets, vous spécifiez la longueur, le type des éléments du tableau et une valeur initiale pour toutes les entrées (généralement, `NULL`). Voici un exemple :

```
jclass class_Employee = (*env)->FindClass(env, "Employee");
jobjectArray array_e = (*env)->NewObjectArray(env, 100,
    class_Employee, NULL);
```

Les tableaux de types primitifs sont plus simples. Vous fournissez simplement la longueur du tableau.

```
jdoubleArray array_d = (*env)->NewDoubleArray(env, 100);
```

Le tableau est alors complété avec des zéros.

INFO

Le JDK 1.4 ajoute trois méthodes dans l'API JNI :

```
jobject NewDirectByteBuffer(JNIEnv* env, void* address, jlong
    capacity)
void* GetDirectBufferAddress(JNIEnv* env, jobject buf)
jlong GetDirectBufferCapacity(JNIEnv* env, jobject buf)
```

Des tampons directs sont utilisés dans le package `java.nio` pour supporter plus efficacement des opérations d'entrée/sortie et réduire la copie des données entre des tableaux natifs et des tableaux Java.

Manipulation des tableaux Java dans le code C

- `jsize GetArrayLength(JNIEnv *env, jobject array)`

Renvoie le nombre d'éléments dans le tableau.

Paramètres : `env` le pointeur de l'interface JNI
 `array` l'objet tableau

- `jobject GetObjectArrayElement(JNIEnv *env, jobjectArray array, jsize index)`

Renvoie la valeur d'un élément du tableau.

Paramètres : `env` le pointeur de l'interface JNI
 `array` l'objet tableau
 `index` le décalage du tableau

- `void SetObjectArrayElement(JNIEnv *env, jobjectArray array, jsize index, jobject value)`

Affecte une nouvelle valeur à un élément du tableau.

Paramètres : `env` le pointeur de l'interface JNI
 `array` l'objet tableau

710 Au cœur de Java 2 - Fonctions avancées

index le décalage du tableau
value la nouvelle valeur

- `Xxx* GetXxxArrayElements(JNIEnv *env, jarray array, jboolean* isCopy)`
Produit un pointeur C sur les éléments d'un tableau Java. Le type de champ `Xxx` peut être `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` ou `Double`. Le pointeur doit être passé à `ReleaseXxxArrayElements` lorsqu'il n'est plus utilisé.

Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>array</code>	l'objet tableau
<code>isCopy</code>	est soit <code>NULL</code> , soit pointe sur un <code>jboolean</code> qui contient <code>JNI_TRUE</code> si une copie est réalisée ; ou <code>JNI_FALSE</code> sinon.

- `void ReleaseXxxArrayElements(JNIEnv *env, jarray array, Xxx elems[], jint mode)`
Signale à la machine virtuelle qu'un pointeur obtenu par `GetXxxArrayElements` n'est plus nécessaire.

Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>array</code>	l'objet tableau
<code>elems</code>	le pointeur sur les éléments de tableau, qui n'est plus nécessaire
<code>mode</code>	0 = libère le tampon <code>elems</code> après mise à jour des éléments du tableau <code>JNI_COMMIT</code> = ne libère pas le tampon <code>elems</code> après mise à jour des éléments du tableau <code>JNI_ABORT</code> = libère le tampon <code>elems</code> sans mise à jour des éléments du tableau

- `void GetXxxArrayRegion(JNIEnv *env, jarray array, jint start, jint length, Xxx elems[])`
Copie des éléments d'un tableau Java vers un Tableau C. Le type de champ `Xxx` peut être `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` ou `Double`.

Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>array</code>	l'objet tableau
<code>start</code>	l'index de départ
<code>length</code>	le nombre des éléments à copier
<code>elems</code>	le Tableau C qui contient les éléments

- `void SetXxxArrayRegion(JNIEnv *env, jarray array, jint start, jint length, Xxx elems[])`
Copie des éléments d'un Tableau C vers un tableau Java. Le type de champ `Xxx` peut être `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` ou `Double`.

Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>array</code>	l'objet tableau
<code>start</code>	l'index de départ
<code>length</code>	le nombre des éléments à copier
<code>elems</code>	le Tableau C qui contient les éléments

La gestion des erreurs

Les méthodes natives constituent un risque significatif au niveau de la sécurité des programmes en langage Java. Le système d'exécution C n'a aucune protection contre les erreurs de limites de tableau, l'indirection *via* des pointeurs erronés, etc. Il est particulièrement important que les programmeurs de méthodes natives gèrent toutes les conditions d'erreur afin de préserver l'intégrité de la plate-forme Java. En particulier, lorsque votre méthode native diagnostique un problème qu'elle ne peut gérer, elle doit signaler ce problème à la machine virtuelle de Java. Dans un tel cas, vous lanceriez alors naturellement une exception. Cependant, il n'y a pas d'exceptions en C. A la place, vous devez appeler la fonction `Throw` ou `ThrowNew` pour créer un nouvel objet exception. Lors de la sortie de la méthode native, la machine virtuelle de Java lance cette exception.

Pour utiliser la fonction `Throw`, appelez `NewObject` pour créer un objet d'un sous-type de `Throwable`. Par exemple, ici, nous allouons un objet `EOFException` et nous le lançons.

```
jclass class_EOFException = (*env)->FindClass(env,
    "java/io/EOFException");
jmethodID id_EOFException = (*env)->GetMethodID(env,
    class_EOFException,
    "<init>", "()V"); /* ID du constructeur par défaut */
jthrowable obj_exc = (*env)->NewObject(env, class_EOFException,
    id_EOFException);
(*env)->Throw(env, obj_exc);
```

Il est généralement plus pratique d'appeler `ThrowNew`, qui construit un objet exception, à partir d'une classe et d'une séquence d'octets "UTF-8 modifié".

```
(*env)->ThrowNew(env, (*env)->FindClass(env,
    "java/io/EOFException"),
    "Fin inattendue de fichier");
```

Les fonctions `Throw` et `ThrowNew` l'une comme l'autre expédient simplement l'exception ; elles n'interrompent pas le flux de la méthode native. Ce n'est que lors du retour de la méthode que la machine virtuelle lance réellement l'exception. Par conséquent, chaque appel à `Throw` et `ThrowNew` doit toujours être suivi immédiatement d'une instruction `return`.

INFO C++

Si vous implémentez des méthodes natives en C++, vous ne pouvez pas lancer un objet exception Java dans votre code C++. Dans une liaison C++, il serait possible d'implémenter une translation entre les exceptions dans les langages C++ et Java ; toutefois, cela n'est pas actuellement implémenté. Utilisez `Throw` ou `ThrowNew` pour lancer une exception Java dans une méthode native C++, et assurez-vous que vos méthodes natives ne lancent pas d'exceptions C++.

Normalement, un code natif n'a pas à se préoccuper de l'interception d'exceptions Java. Toutefois, lorsqu'une méthode native appelle une méthode Java, cette méthode peut très bien lancer une exception. De plus, un certain nombre des fonctions JNI lancent également des exceptions. Par exemple, `SetObjectArrayElement` lance une exception `ArrayIndexOutOfBoundsException` si l'index est hors limites, et une exception `ArrayStoreException` si la classe de l'objet stocké n'est pas une

712 *Au cœur de Java 2 - Fonctions avancées*

sous-classe de la classe de l'élément du tableau. Dans de telles situations, une méthode native doit appeler la méthode `ExceptionOccurred` pour déterminer si une exception a été lancée. L'appel :

```
jthrowable obj_exc = (*env)->ExceptionOccurred(env);
```

renvoie NULL si aucune exception n'est en attente, ou il renvoie une référence à l'objet exception courant. Si vous voulez simplement vérifier qu'une exception a été lancée, sans obtenir de référence à l'objet exception, utilisez :

```
jboolean occurred = (*env)->ExceptionCheck(env);
```

Normalement, une méthode native doit simplement retourner lorsqu'une exception s'est produite, afin que la machine virtuelle puisse la propager au code Java. Cependant, une méthode native peut analyser l'objet exception pour déterminer si elle peut gérer l'exception. Dans l'affirmative, la fonction

```
(*env)->ExceptionClear(env);
```

doit être appelée pour désactiver l'exception.

Dans notre prochain exemple, nous implémentons la méthode native `fprint` avec la paranoïa justifiée dans le cas d'une méthode native. Voici les exceptions que nous lançons :

- une exception `NullPointerException` si la chaîne de format est NULL ;
- une exception `IllegalArgumentException` si la chaîne de format ne contient pas de spécificateur `%` dans le cas d'impression d'un type double ;
- une exception `OutOfMemoryError` si l'appel à `malloc` échoue.

Enfin, pour démontrer la façon de vérifier la présence d'une exception lors de l'appel d'une méthode Java à partir d'une méthode native, nous envoyons la chaîne au flux, un caractère à la fois, et appelons `ExceptionOccurred` après chaque appel. L'Exemple 9.17 montre le code pour la méthode native, et l'Exemple 9.18 contient la définition de la classe contenant la méthode native. Remarquez que la méthode native ne se termine pas immédiatement lorsqu'une exception se produit dans l'appel à `PrintWriter.print` ; elle libère d'abord le tampon `cstr`. Lors du retour de la méthode native, la machine virtuelle émet à nouveau l'exception. Le programme de test de l'Exemple 9.19 montre comment la méthode native lance une exception lorsque la chaîne de formatage n'est pas valide.

Exemple 9.17 : `Printf4.c`

```
#include "Printf4.h"
#include <string.h>
#include <stdlib.h>
#include <float.h>

/**
 * @param format une chaîne contenant un spécificateur de format printf
 * (comme "%8.2f"). Les sous-chaînes "%" sont sautées.
 * @return renvoie un pointeur sur le spécificateur de format (sautant
 * "%") ou NULL s'il n'y avait pas un spécificateur unique de format
 */
char* find_format(const char format[])
{
    char* p;
    char* q;

    p = strchr(format, '%');
```

```

while (p != NULL && *(p + 1) == '%') /* sauter %% */
    p = strchr(p + 2, '%');
if (p == NULL) return NULL;
/* vérifier maintenant que % est unique */
p++;
q = strchr(p, '%');
while (q != NULL && *(q + 1) == '%') /* sauter %% */
    q = strchr(q + 2, '%');
if (q != NULL) return NULL; /* % non unique */
q = p + strspn(p, " -0+#"); /* sauter les indicateurs */
q += strspn(q, "0123456789"); /* sauter la largeur de champ */
if (*q == '.') { q++; q += strspn(q, "0123456789"); }
/* sauter la précision */
if (strchr("eEfFgG", *q) == NULL) return NULL;
/* n'est pas un format virgule flottante */
return p;
}

JNIEXPORT void JNICALL Java_Printf4_fprint
(JNIEnv* env, jclass cl, jobject out, jstring format,
 jdouble x)
{
    const char* cformat;
    char* fmt;
    jclass class_PrintWriter;
    jmethodID id_print;
    char* cstr;
    int width;
    int i;

    if (format == NULL)
    {
        (*env)->ThrowNew(env,
            (*env)->FindClass(env,
                "java/lang/NullPointerException"),
            "Printf4.fprint: format null");
        return;
    }

    cformat = (*env)->GetStringUTFChars(env, format, NULL);
    fmt = find_format(cformat);

    if (fmt == NULL)
    {
        (*env)->ThrowNew(env,
            (*env)->FindClass(env,
                "java/lang/IllegalArgumentException"),
            "Printf4.fprint: format non valide");
        return;
    }

    width = atoi(fmt);
    if (width == 0) width = DBL_DIG + 10;
    cstr = (char*)malloc(strlen(cformat) + width);

    if (cstr == NULL)
    {
        (*env)->ThrowNew(env,
            (*env)->FindClass(env, "java/lang/OutOfMemoryError"),

```


714 *Au cœur de Java 2 - Fonctions avancées*

```

        "Printf4.fprint: échec malloc");
    return;
}

sprintf(cstr, cformat, x);

(*env)->ReleaseStringUTFChars(env, format, cformat);

/* appeler maintenant ps.print(str) */

/* extraire la classe */
class_PrintWriter = (*env)->GetObjectClass(env, out);

/* extraire l'ID de méthode */
id_print = (*env)->GetMethodID(env, class_PrintWriter,
    "print", "(C)V");

/* appeler la méthode */
for (i = 0; cstr[i] != 0 && !(*env)->ExceptionOccurred(env);
    i++)
    (*env)->CallVoidMethod(env, out, id_print, cstr[i]);

free(cstr);
}

```

Exemple 9.18 : Printf4.java

```

import java.io.*;

class Printf4
{
    public static native void fprint(PrintWriter ps,
        String format, double x);

    static
    {
        System.loadLibrary("Printf4");
    }
}

```

Exemple 9.19 : Printf4Test.java

```

import java.io.*;

class Printf4Test
{
    public static void main(String[] args)
    {
        double price = 44.95;
        double tax = 7.75;
        double amountDue = price * (1 + tax / 100);
        PrintWriter out = new PrintWriter(System.out);
        /* Cet appel déclenchera une exception - notez les %% */
        Printf4.fprint(out, "Montant dû = %%8.2f\n", amountDue);
        out.flush();
    }
}

```

Gestion des erreurs dans le code C

- `jint Throw(JNIEnv *env, jthrowable obj)`
Prépare une exception qui sera lancée lors de la sortie du code natif. Renvoie 0 en cas de succès, une valeur négative en cas d'échec.

Paramètres : `env` le pointeur de l'interface JNI
 `obj` l'objet exception à lancer
- `jint ThrowNew(JNIEnv *env, jclass clazz, const char msg[])`
Prépare une exception qui sera lancée lors de la sortie du code natif. Renvoie 0 en cas de succès, une valeur négative en cas d'échec.

Paramètres : `env` le pointeur de l'interface JNI
 `cl` la classe de l'objet exception à lancer
 `msg` une chaîne UTF indiquant l'argument de construction de String de l'objet exception
- `jthrowable ExceptionOccurred(JNIEnv *env)`
Renvoie l'objet exception si une exception est en attente, NULL sinon.

Paramètres : `env` le pointeur de l'interface JNI
- `jboolean ExceptionCheck(JNIEnv *env)`
Renvoie true si une exception est en attente.

Paramètres : `env` le pointeur de l'interface JNI
- `void ExceptionClear(JNIEnv *env)`
Efface toutes les exceptions en attente.

Paramètres : `env` le pointeur de l'interface JNI

L'API d'invocation

Jusqu'à présent, nous avons vu des programmes en langage Java réalisant quelques appels en C, *a priori* du fait que C était plus rapide ou permettait d'accéder à des fonctionnalités inaccessibles à partir de la plate-forme Java. Imaginons que vous soyez dans la situation inverse. Vous avez un programme C ou C++ et vous voudriez faire quelques appels en code Java, du fait peut-être que le code Java est plus facile à programmer. Vous savez, bien entendu, comment appeler les méthodes Java. Mais vous devez néanmoins toujours ajouter la machine virtuelle Java à votre programme, afin que le code Java puisse être interprété. L'API appelée *invocation* est utilisée pour incorporer la machine virtuelle Java dans un programme C ou C++. Voici le code minimal nécessaire pour l'initialisation d'une machine virtuelle.

```
JavaVMOption options[1];
JavaVMInitArgs vm_args;
JavaVM *jvm;
JNIEnv *env;

options[0].optionString = "-Djava.class.path=.";

memset(&vm_args, 0, sizeof(vm_args));
vm_args.version = JNI_VERSION_1_2;
```

716 Au cœur de Java 2 - Fonctions avancées

```
vm_args.nOptions = 1;
vm_args.options = options;

JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);
```

L'appel à `JNI_CreateJavaVM` crée la machine virtuelle et initialise un pointeur `jvm` sur la machine virtuelle et un pointeur `env` sur l'environnement d'exécution.

Vous pouvez fournir n'importe quel nombre d'options à la machine virtuelle. Augmentez simplement la taille du tableau d'options et la valeur de `vm_args.nOptions`. Par exemple,

```
options[i].optionString = "-Djava.compiler=NONE";
```

désactive le compilateur JIT (*juste à temps*).

ASTUCE

Lorsque vous avez des problèmes et que votre programme plante, refuse d'initialiser la JVM ou ne peut charger vos classes, activez le mode débogage de JNI. Définissez une option à

```
options[i].optionString = "-verbose:jni";
```

Vous verrez apparaître une rafale de messages indiquant la progression de l'initialisation de la JVM. Si vous ne voyez pas vos classes se charger, vérifiez vos paramètres à la fois dans `PATH` et `CLASSPATH`.

Lorsque vous avez initialisé la machine virtuelle, vous pouvez appeler les méthodes Java de la façon décrite dans les sections précédentes : utilisez simplement le pointeur `env` de la manière habituelle. Vous n'avez besoin du pointeur `jvm` que pour appeler d'autres fonctions dans l'API d'invocation. Il n'existe actuellement que quatre de ces fonctions. La plus importante est la fonction permettant de supprimer la machine virtuelle :

```
(*jvm)->DestroyJavaVM(jvm);
```

Le programme C dans l'Exemple 9.20 initialise une machine virtuelle puis appelle la méthode `main` de la classe `Welcome`, que nous avons vue au Chapitre 2 du Volume 1. N'oubliez pas de compiler le fichier `Welcome.java` avant de démarrer le programme de test de l'invocation.

Exemple 9.20 : *InvocationTest.c*

```
#include <jni.h>
#include <stdlib.h>

int main()
{
    JavaVMOption options[2];
    JavaVMInitArgs vm_args;
    JavaVM *jvm;
    JNIEnv *env;
    long status;

    jclass class_Welcome;
    jclass class_String;
    jobjectArray args;
    jmethodID id_main;

    options[0].optionString = "-Djava.class.path=.";
```

```
memset(&vm_args, 0, sizeof(vm_args));
vm_args.version = JNI_VERSION_1_2;
vm_args.nOptions = 1;
vm_args.options = options;

status = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);
if (status == JNI_ERR)
{
    printf("Erreur lors de la création de la VM\n");
    return 1;
}

class_Welcome = (*env)->FindClass(env, "Welcome");
id_main = (*env)->GetStaticMethodID(env, class_Welcome,
    "main", "(Ljava/lang/String;)V");

class_String = (*env)->FindClass(env, "java/lang/String");
args = (*env)->NewObjectArray(env, 0, class_String, NULL);
(*env)->CallStaticVoidMethod(env, class_Welcome,
    id_main, args);

(*jvm)->DestroyJavaVM(jvm);

return 0;
}
```

Pour compiler ce programme sous Linux, utilisez :

```
gcc -I jdk/include -I jdk/include/linux
-o InvocationTest -L jdk/jre/lib/i386/client
-ljvm InvocationTest.c
```

Sous Solaris, utilisez :

```
cc -I jdk/include -I jdk/include/solaris
-o InvocationTest -L jdk/jre/lib/sparc
-ljvm InvocationTest.c
```

Pour compiler sous Windows à l'aide du compilateur Microsoft C, utilisez la ligne de commande :

```
cl -I jdk\include -I jdk\include\win32
InvocationTest.c jdk\lib\jvm.lib
```

Avec Cygwin, d'autres opérations sont nécessaires. Créez tout d'abord un fichier `jvm.def` contenant l'instruction :

```
EXPORTS
    JNI_CreateJavaVM@12
```

Exécutez la commande :

```
dlltool -k --input-def jvm.def
--dll jdk\jre\bin\client\jvm.dll
--output-lib jvm.a
```

ATTENTION

Si vous émettez cette commande à partir du shell bash, insérez quatre barres inversées `\\` pour chaque séparateur de fichier.

718 *Au cœur de Java 2 - Fonctions avancées*

Puis compilez avec :

```
gcc -I jdk\include -I jdk\include\win32
-D__int64="long long" -o InvocationTest
InvocationTest.c jvm.a
```

Avant de lancer le programme sous Linux/UNIX, vérifiez que LD_LIBRARY_PATH contient les répertoires destinés aux bibliothèques partagées. Par exemple, si vous utilisez le shell bash sous Linux, utilisez la commande suivante :

```
export LD_LIBRARY_PATH= jdk/jre/lib/i386/client:$LD_LIBRARY_PATH
```

Sous Windows, vérifiez que le répertoire

```
jdk\jre\bin\client
```

se trouve dans PATH.

ATTENTION

Les emplacements exacts des fichiers de bibliothèque varient en fonction de la version du JDK que vous possédez. Vous devrez peut-être rechercher les fichiers intitulés libjvm.so, jvm.dll ou (dans les anciennes versions du JDK) libjava.so, dans les répertoires jdk/bin, jdk/lib et jdk/jre/lib, puis ajuster les instructions en fonction de ces informations.

ASTUCE

Si vous déployez une application qui invoque la machine virtuelle à l'aide d'un lanceur Windows, vous pouvez ne pas faire confiance à vos utilisateurs pour la configuration du chemin d'accès à la bibliothèque. Vous pouvez les aider et charger manuellement la bibliothèque partagée ou le fichier DLL. Pour un exemple de code, voir le fichier launcher\java_md.c dans le fichier src.jar faisant partie du JDK.

Fonctions de l'API d'invocation

- `jint JNI_CreateJavaVM(JavaVM** p_jvm, void** p_env, JavaVMInitArgs* vm_args)`
Initialise la machine virtuelle Java. La fonction renvoie 0 en cas de succès, JNI_ERR en cas d'échec.

<i>Paramètres :</i>	<code>p_jvm</code>	contient un pointeur sur la table de fonction de l'API d'invocation
	<code>p_env</code>	contient un pointeur sur la table de fonction de la JNI
	<code>vm_args</code>	Les arguments de la machine virtuelle

- `jint DestroyJavaVM(JavaVM* jvm)`
Détruit la machine virtuelle. Renvoie 0 en cas de succès, une valeur négative en cas d'échec. Cette fonction doit être appelée par l'intermédiaire d'un pointeur sur la machine virtuelle, c'est-à-dire `(*jvm)->DestroyJavaVM(jvm)`.

<i>Paramètres :</i>	<code>jvm</code>	le pointeur sur la machine virtuelle

Un exemple exhaustif : accès à la base de registres de Windows

Dans cette section, nous allons décrire un exemple complet, opérationnel, traitant tous les éléments décrits dans ce chapitre : utilisation de méthodes natives avec des chaînes, des tableaux, des objets, des appels au constructeur et la gestion d'erreur. Nous allons vous montrer comment placer une enveloppe (*wrapper*) de plate-forme Java autour d'un sous-ensemble de l'API ordinaire à base de C, utilisé pour travailler avec la base de registres de Windows. Bien entendu, puisqu'il s'agit d'une fonctionnalité spécifique à Windows, un programme utilisant la base de registres de Windows est forcément non portable. Pour cette raison, la bibliothèque Java standard n'a pas de support pour la base de registres, et il est donc logique d'utiliser des méthodes natives pour y accéder.

Aperçu de la base de registres de Windows

La base de registres de Windows est un référentiel de données contenant des informations de configuration pour les systèmes d'exploitation Windows et les programmes d'application. Elle constitue un lieu central pour la gestion et la sauvegarde du système et des préférences applicatives.

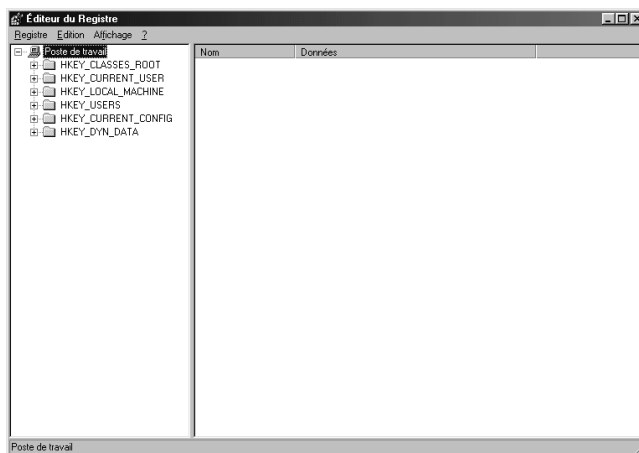
Il faut aussi admettre que la base de registres constitue à elle seule une source de panne potentielle ; si vous la désorganisez, votre ordinateur peut faire état de dysfonctionnements, ou même être incapable de démarrer ! L'exemple de programme que nous présentons dans cette section est sûr, mais si vous envisagez de le modifier, vous devez savoir comment sauvegarder votre base de registres avant de poursuivre.

Nous ne vous suggérons pas d'utiliser la base de registre pour stocker des paramètres de configuration des programmes Java. L'API des préférences Java constitue une meilleure solution (reportez-vous au Chapitre 10 du Volume 1 pour obtenir de plus amples informations). Nous utilisons simplement la base de registre pour montrer comment envelopper une API native non-triviale dans une classe Java.

L'outil principal de consultation de la base de registres est l'Editeur de base de registres. Du fait du risque élevé d'erreurs potentielles de son utilisation par des utilisateurs novices, mais parfois enthousiastes, il n'existe pas d'icône pour le lancement de cet outil. Vous devez démarrer un shell DOS (ou ouvrir la boîte de dialogue Démarrer/Exécuter) et taper `regedit`. La Figure 9.3 montre l'Editeur de base de registres à l'action.

Figure 9.3

L'Editeur de la base de registres.



720 *Au cœur de Java 2 - Fonctions avancées*

Le volet gauche affiche les clés, sous la forme d'une arborescence. Remarquez que chaque clé commence par l'un des nœuds HKEY comme

```
HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
. . .
```

Le volet droit affiche la paire nom/valeur associée à une clé particulière. Par exemple, la clé

```
HKEY_LOCAL_MACHINE\Software\JavaSoft\Java Development Kit\1.5
```

a deux paires nom/valeur, c'est-à-dire

```
JavaHome="c:\jdk1.5.0"
MicroVersion="0"
```

Ici, les valeurs sont des chaînes. Elles peuvent aussi être des entiers ou des tableaux d'octets.

Une interface de plate-forme Java pour accéder à la base de registres

Nous allons implémenter une interface simple permettant d'accéder à la base de registres à partir de code Java, puis implémenter cette interface avec du code natif. Notre interface permet seulement quelques opérations sur la base de registres ; pour limiter la taille du code, nous avons omis quelques opérations importantes comme l'ajout, la suppression et l'énumération de clés. (Il serait facile d'ajouter les fonctions API de base de registres restantes.)

Même avec le sous-ensemble limité que nous fournissons, vous pouvez :

- énumérer tous les noms stockés dans une clé ;
- lire la valeur stockée avec un nom ;
- définir la valeur stockée avec un nom.

Voici la classe de plate-forme Java qui encapsule une clé de la base de registres.

```
public class Win32RegKey
{
    public Win32RegKey(int theRoot, String thePath) { . . . }
    public Enumeration names() { . . . }
    public native Object getValue(String name);
    public native void setValue(String name, Object value);

    public static final int HKEY_CLASSES_ROOT = 0x80000000;
    public static final int HKEY_CURRENT_USER = 0x80000001;
    public static final int HKEY_LOCAL_MACHINE = 0x80000002;
    . . .
}
```

La méthode `names` renvoie une énumération qui contient tous les noms stockés avec la clé. Vous pouvez y accéder à l'aide des méthodes habituelles `hasMoreElements`/`nextElement`. La méthode `getValue` renvoie un objet qui peut être une chaîne, un entier ou un tableau d'octets. Le paramètre `value` de la méthode `setValue` doit aussi être de l'un de ces trois types.

Voici une fonction simple qui répertorie les chaînes qui sont stockées avec la clé :

```
HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\Java Development Kit\1.5
```

(Il est recommandé de modifier la clé en fonction de la version du JDK installée sur votre système.)

```
public static void main(String[] args)
{
    Win32RegKey key = new Win32RegKey(
        Win32RegKey.HKEY_LOCAL_MACHINE,
        "Software\\JavaSoft\\Java Development Kit\\1.5");

    Enumeration<String> e = key.names();

    while (e.hasMoreElements())
    {
        String name = e.nextElement();
        System.out.println(name + " = " + key.getValue(name));
    }
}
```

Typiquement, la sortie de ce programme pourrait être :

```
JavaHome=c:\\jdk5.0
MicroVersion=0
```

Implémenter les fonctions d'accès à la base de registres avec un code natif

Nous devons implémenter trois actions :

- extraire la valeur d'une clé ;
- définir la valeur d'une clé ;
- itérer parmi les noms d'une clé.

Vous avez heureusement déjà vu l'essentiel des outils nécessaires, par exemple pour la conversion entre les chaînes et tableaux dans Java et ceux dans C. Et vous savez comment lancer une exception Java en cas de problème.

Deux difficultés rendent ces méthodes natives plus complexes que les exemples précédents. Les méthodes `getValue` et `setValue` travaillent avec le type `Object`, qui peut être de type `String`, `Integer` ou `byte[]`. L'objet énumération stocke l'état entre les appels successifs à `hasMoreElements` et `nextElement`.

Voyons d'abord la méthode `getValue`. Le code (montré dans l'Exemple 9.22) exécute les étapes suivantes.

1. Il ouvre la clé de base de registres. Pour lire leurs valeurs, l'API de base de registres requiert que les clés soient ouvertes.
2. Il demande le type et la taille de la valeur associée au nom.
3. Il lit les données dans un tampon.
4. Si le type est `REG_SZ` (une chaîne), il appelle `NewStringUTF` pour créer une nouvelle chaîne avec les valeurs.
5. Si le type est `REG_DWORD` (un entier 32 bits), il invoque le constructeur de `Integer`.
6. Si le type est `REG_BINARY`, il appelle `NewByteArray` pour créer un nouveau tableau d'octets, et appelez `SetByteArrayRegion` pour copier la valeur dans le tableau d'octets.

722 Au cœur de Java 2 - Fonctions avancées

7. Si le type n'est aucun de ceux mentionnés ou si une erreur s'est produite lors de l'appel d'une fonction API, il lance une exception et libère soigneusement toutes les ressources qui ont été acquises jusqu'à présent.
8. Il ferme la clé et renvoie l'objet (`String`, `Integer` ou `byte[]`) qui a été créé.

Comme vous le voyez, cet exemple illustre bien comment générer des objets Java de différents types.

Dans cette méthode native, il n'était pas difficile de se débrouiller avec le type renvoyé générique. La référence à `jstring`, `jobject` ou `jarray` a été simplement renvoyée comme `jobject`. Toutefois, la méthode `setValue` reçoit une référence à un `Object`, et elle doit déterminer le type exact de l'objet afin de pouvoir le sauvegarder comme chaîne, entier ou tableau d'octets. Nous pouvons le déterminer en demandant la classe de l'objet `value`, en trouvant les références de classe pour `java.lang.String`, `java.lang.Integer` et `byte[]`, et en les comparant à l'aide de la fonction `IsAssignableFrom`.

Si `class1` et `class2` sont deux références de classe, alors l'appel

```
(*env)->IsAssignableFrom(env, class1, class2)
```

renvoie `JNI_TRUE` lorsque `class1` et `class2` sont de la même classe ou lorsque `class1` est une sous-classe de `class2`. Dans l'un ou l'autre cas, les références aux objets de `class1` peuvent être converties en `class2`. Par exemple, lorsque

```
(*env)->IsAssignableFrom(env,
    (*env)->GetObjectClass(env, value),
    (*env)->FindClass(env, "[B"))
```

donne `true`, nous savons que `value` est un tableau d'octets.

Voici un aperçu de ce que réalise le code de la méthode `setValue` :

1. Il ouvre la clé de base de registres en écriture.
2. Il trouve le type de la valeur à écrire.
3. Si le type est `String`, il appelle `GetStringUTFChars` pour obtenir un pointeur sur les caractères. Il obtient également la longueur de la chaîne.
4. Si le type est `Integer`, il appelle la méthode `intValue` pour obtenir l'entier stocké dans l'objet wrapper.
5. Si le type est `byte[]`, il appelle `GetByteArrayElements` pour obtenir un pointeur sur les octets. Il obtient également la longueur de la chaîne.
6. Il passe les données et la longueur à la base de registres.
7. Il ferme la clé. Si le type est `String` ou `byte[]`, il libère également le pointeur sur les caractères ou les octets.

Voyons maintenant les méthodes natives qui énumèrent les clés. Ce sont des méthodes de la classe `Win32RegKeyNameEnumeration` (voir Exemple 9.21). Lorsque le processus d'énumération démarre, nous devons ouvrir la clé. Pour la durée de l'énumération, nous devons conserver le handle de clé. C'est-à-dire que le handle de clé doit être stocké avec l'objet énumération. Le handle de clé est du type `DWORD`, 32 bits et peut par conséquent être stocké dans un entier Java. Il est stocké dans le champ `hkey` de la classe `enumeration`. Lorsque l'énumération démarre, le champ est initialisé à l'aide de `SetIntField`. Les appels suivants lisent la valeur à l'aide de `GetIntField`.

Dans cet exemple, nous stockons trois autres éléments de données avec l'objet énumération. Lors du premier démarrage de l'énumération, nous pouvons interroger la base de registres pour connaître le compte de paires nom/valeur et la longueur du nom le plus long, ce qui nous permettra d'allouer les tableaux de caractères C devant contenir les noms. Ces valeurs sont stockées dans les champs `count` et `maxsize` de l'objet énumération. Enfin, le champ `index` est initialisé à `-1` pour indiquer le début de l'énumération, puis défini à `0` lorsque les autres champs d'instance sont initialisés, et incrémenté après chaque étape de l'énumération.

Voici le détail des méthodes natives qui gèrent l'énumération. La méthode `hasMoreElements` est simple.

1. Elle extrait les champs `index` et `count`.
2. Si `index` vaut `-1`, elle appelle la fonction `startNameEnumeration`, qui ouvre la clé, demande le compte et la longueur maximale, et initialise les champs `hkey`, `count`, `maxsize` et `index`.
3. Elle renvoie `JNI_TRUE` si `index` est inférieur à `count` ; `JNI_FALSE` sinon.

La méthode `nextElement` est un peu plus complexe.

1. Elle extrait les champs `index` et `count`.
2. Si `index` vaut `-1`, elle appelle la fonction `startNameEnumeration`, qui ouvre la clé, demande le compte et la longueur maximale, et initialise les champs `hkey`, `count`, `maxsize` et `index`.
3. Si `index` est égal à `count`, elle lance une exception `NoSuchElementException`.
4. Elle lit le nom suivant dans la base de registres.
5. Elle incrémente l'index.
6. Si `index` est égal à `count`, elle ferme la clé.

Avant la compilation, n'oubliez pas d'exécuter `javah` à la fois sur `Win32RegKey` et `Win32RegKeyNameEnumeration`. La ligne de commande complète pour Microsoft C++ est la suivante :

```
cl -I jdk\include -I jdk\include\win32 -LD
Win32RegKey.c advapi32.lib -FeWin32RegKey.dll
```

Avec Cygwin, utilisez :

```
gcc -mno-cygwin -D __int64="long long" -I jdk\include
-I jdk\include\win32 -I c:\cygwin\usr\include\w32api
-shared -Wl,--add-stdcall-alias -o Win32RegKey.dll Win32RegKey.c
```

L'Exemple 9.23 montre un programme qui teste nos nouvelles fonctions base de registres. Nous ajoutons trois paires nom/valeur, une chaîne, un entier et un tableau d'octets à la clé.

```
HKEY_LOCAL_MACHINE\Software\JavaSoft\Java Development Kit\1.5
```

Vous devrez modifier le numéro de version afin qu'il corresponde à votre installation du JDK (ou utilisez simplement d'autres clés de registre existantes).

Nous énumérons ensuite tous les noms de cette clé et extrayons leurs valeurs. Le programme imprimera :

```
JavaHome=c:\jdk1.5.0
MicroVersion=0
Utilisateur par défaut = Bozo le clown
Chiffre porte-bonheur = 13
Nombres premiers = 2 3 5 7 11 13
```

724 *Au cœur de Java 2 - Fonctions avancées*

Bien que l'ajout de ces paires nom/valeur à cette clé ne puisse pas causer de dommages, il est préférable d'utiliser l'Editeur de la base de registres pour les supprimer après exécution de ce programme.

Exemple 9.21 : Win32RegKey.java

```
import java.util.*;

/**
 * Un objet Win32RegKey peut être utilisé pour extraire et définir les
 * valeurs d'une clé de registre dans la base de registre Windows.
 */
public class Win32RegKey
{
    /**
     * Construit un objet de clé de registre.
     * @param theRoot, parmi HKEY_CLASSES_ROOT, HKEY_CURRENT_USER,
     * HKEY_LOCAL_MACHINE, HKEY_USERS, HKEY_CURRENT_CONFIG,
     * HKEY_DYN_DATA
     * @param thePath le chemin de la clé de registre
     */
    public Win32RegKey(int theRoot, String thePath)
    {
        root = theRoot;
        path = thePath;
    }

    /**
     * Enumère tous les noms des entrées de registre sous le chemin décrit
     * par cet objet.
     * @return une énumération recensant tous les noms d'entrée
     */
    public Enumeration<String> names()
    {
        return new Win32RegKeyNameEnumeration(root, path);
    }

    /**
     * Extrait la valeur d'une entrée de registre.
     * @param name le nom de l'entrée
     * @return la valeur associée
     */
    public native Object getValue(String name);

    /**
     * Définit la valeur d'une entrée de registre.
     * @param name le nom de l'entrée
     * @param value la nouvelle valeur
     */
    public native void setValue(String name, Object value);

    public static final int HKEY_CLASSES_ROOT = 0x80000000;
    public static final int HKEY_CURRENT_USER = 0x80000001;
    public static final int HKEY_LOCAL_MACHINE = 0x80000002;
    public static final int HKEY_USERS = 0x80000003;
    public static final int HKEY_CURRENT_CONFIG = 0x80000005;
    public static final int HKEY_DYN_DATA = 0x80000006;

    private int root;
    private String path;
}
```

```

        static
        {
            System.loadLibrary("Win32RegKey");
        }
    }

    class Win32RegKeyNameEnumeration implements Enumeration<String>
    {
        Win32RegKeyNameEnumeration(int theRoot, String thePath)
        {
            root = theRoot;
            path = thePath;
        }

        public native String nextElement();
        public native boolean hasMoreElements();

        private int root;
        private String path;
        private int index = -1;
        private int hkey = 0;
        private int maxsize;
        private int count;
    }

    class Win32RegKeyException extends RuntimeException
    {
        public Win32RegKeyException() {}
        public Win32RegKeyException(String why)
        {
            super(why);
        }
    }
}

```

Exemple 9.22 : Win32RegKey.c

```

#include "Win32RegKey.h"
#include "Win32RegKeyNameEnumeration.h"
#include <string.h>
#include <stdlib.h>
#include <windows.h>

JNIEXPORT jobject JNICALL Java_Win32RegKey_getValue
(JNIEnv* env, jobject this_obj, jobject name)
{
    const char* cname;
    jstring path;
    const char* cpath;
    HKEY hkey;
    DWORD type;
    DWORD size;
    jclass this_class;
    jfieldID id_root;
    jfieldID id_path;
    HKEY root;
    jobject ret;
    char* cret;
}

```

726 Au cœur de Java 2 - Fonctions avancées

```
/* extraire la classe */
this_class = (*env)->GetObjectClass(env, this_obj);

/* extraire les ID de champ */
id_root = (*env)->GetFieldID(env, this_class, "root", "I");
id_path = (*env)->GetFieldID(env, this_class, "path",
    "Ljava/lang/String;");

/* extraire les champs */
root = (HKEY) (*env)->GetIntField(env, this_obj, id_root);
path = (jstring) (*env)->GetObjectField(env, this_obj,
    id_path);
cpath = (*env)->GetStringUTFChars(env, path, NULL);

/* ouvrir la clé de base de registres */
if (RegOpenKeyEx(root, cpath, 0, KEY_READ, &hkey)
    != ERROR_SUCCESS)
{
    (*env)->ThrowNew(env,
        (*env)->FindClass(env, "Win32RegKeyException"),
        "Echec de l'ouverture de clé");
    (*env)->ReleaseStringUTFChars(env, path, cpath);
    return NULL;
}

(*env)->ReleaseStringUTFChars(env, path, cpath);
cname = (*env)->GetStringUTFChars(env, name, NULL);

/* trouver le type et la taille de la valeur */
if (RegQueryValueEx(hkey, cname, NULL, &type, NULL, &size) !=
    ERROR_SUCCESS)
{
    (*env)->ThrowNew(env,
        (*env)->FindClass(env, "Win32RegKeyException"),
        "Echec de la recherche de valeur de clé");
    RegCloseKey(hkey);
    (*env)->ReleaseStringUTFChars(env, name, cname);
    return NULL;
}

/* allouer la mémoire pour contenir la valeur */
cret = (char*)malloc(size);

/* lire la valeur */
if (RegQueryValueEx(hkey, cname, NULL, &type, cret, &size) !=
    ERROR_SUCCESS)
{
    (*env)->ThrowNew(env,
        (*env)->FindClass(env, "Win32RegKeyException"),
        "Echec de la recherche de valeur de clé");
    free(cret);
    RegCloseKey(hkey);
    (*env)->ReleaseStringUTFChars(env, name, cname);
    return NULL;
}

/* selon le type, stocker la valeur dans
une chaîne, un entier ou un tableau d'octets */
if (type == REG_SZ)
```

```

    {
        ret = (*env)->NewStringUTF(env, cret);
    }
    else if (type == REG_DWORD)
    {
        jclass class_Integer = (*env)->FindClass(env,
            "java/lang/Integer");
        /* extraire l'ID de méthode du constructeur */
        jmethodID id_Integer = (*env)->GetMethodID(env,
            class_Integer, "<init>", "(I)V");
        int value = *(int*)cret;
        /* invoquer le constructeur */
        ret = (*env)->NewObject(env, class_Integer, id_Integer,
            value);
    }
    else if (type == REG_BINARY)
    {
        ret = (*env)->NewByteArray(env, size);
        (*env)->SetByteArrayRegion(env, (jarray)ret, 0, size,
            cret);
    }
    else
    {
        (*env)->ThrowNew(env,
            (*env)->FindClass(env, "Win32RegKeyException"),
            "Type de valeur non géré");
        ret = NULL;
    }

    free(cret);
    RegCloseKey(hkey);
    (*env)->ReleaseStringUTFChars(env, name, cname);

    return ret;
}

JNIEXPORT void JNICALL Java_Win32RegKey_setValue
(JNIEnv* env, jobject this_obj, jstring name, jobject value)
{
    const char* cname;
    jstring path;
    const char* cpath;
    HKEY hkey;
    DWORD type;
    DWORD size;
    jclass this_class;
    jclass class_value;
    jclass class_Integer;
    jfieldID id_root;
    jfieldID id_path;
    HKEY root;
    const char* cvalue;
    int ivalue;

    /* extraire la classe */
    this_class = (*env)->GetObjectClass(env, this_obj);

    /* extraire les ID de champ */
    id_root = (*env)->GetFieldID(env, this_class, "root", "I");

```

728 Au cœur de Java 2 - Fonctions avancées

```

id_path = (*env)->GetFieldID(env, this_class, "path",
    "Ljava/lang/String;");

/* extraire les champs */
root = (HKEY)(*env)->GetIntField(env, this_obj, id_root);
path = (jstring)(*env)->GetObjectField(env, this_obj,
    id_path);
cpath = (*env)->GetStringUTFChars(env, path, NULL);

/* ouvrir la clé de base de registres */
if (RegOpenKeyEx(root, cpath, 0, KEY_WRITE, &hkey)
    != ERROR_SUCCESS)
{
    (*env)->ThrowNew(env,
        (*env)->FindClass(env, "Win32RegKeyException"),
        "Echec de l'ouverture de clé");
    (*env)->ReleaseStringUTFChars(env, path, cpath);
    return;
}

(*env)->ReleaseStringUTFChars(env, path, cpath);
cname = (*env)->GetStringUTFChars(env, name, NULL);

class_value = (*env)->GetObjectClass(env, value);
class_Integer = (*env)->FindClass(env, "java/lang/Integer");
/* déterminer le type de l'objet value */
if ((*env)->IsAssignableFrom(env, class_value,
    (*env)->FindClass(env, "java/lang/String")))
{
    /* c'est une chaîne--obtenir un pointeur sur les caractères */
    cvalue = (*env)->GetStringUTFChars(env, (jstring) value,
        NULL);
    type = REG_SZ;
    size = (*env)->GetStringLength(env, (jstring) value) + 1;
}
else if ((*env)->IsAssignableFrom(env, class_value,
    class_Integer))
{
    /* c'est un entier--appeler intValue pour obtenir la valeur */
    jmethodID id_intValue = (*env)->GetMethodID(env,
        class_Integer, "intValue", "()I");
    ivalue = (*env)->CallIntMethod(env, value, id_intValue);
    type = REG_DWORD;
    cvalue = (char*)&ivalue;
    size = 4;
}
else if ((*env)->IsAssignableFrom(env, class_value,
    (*env)->FindClass(env, "[B]")))
{
    /* c'est un tableau d'octets--obtenir un pointeur sur les octets */
    type = REG_BINARY;
    cvalue = (char*)(*env)->GetByteArrayElements(env,
        (jarray) value, NULL);
    size = (*env)->GetArrayLength(env, (jarray) value);
}
else
{
    /* nous ne savons pas comment gérer ce type */
    (*env)->ThrowNew(env,

```

```

        (*env)->FindClass(env, "Win32RegKeyException"),
        "Type de valeur non géré");
    RegCloseKey(hkey);
    (*env)->ReleaseStringUTFChars(env, name, cname);
    return;
}

/* définir la valeur */
if (RegSetValueEx(hkey, cname, 0, type, cvalue, size)
    != ERROR_SUCCESS)
{
    (*env)->ThrowNew(env,
        (*env)->FindClass(env, "Win32RegKeyException"),
        "Echec de la valeur du jeu");
}

RegCloseKey(hkey);
(*env)->ReleaseStringUTFChars(env, name, cname);

/* si la valeur était une chaîne ou un tableau d'octets,
   libérer le pointeur */
if (type == REG_SZ)
{
    (*env)->ReleaseStringUTFChars(env, (jstring) value,
        cvalue);
}
else if (type == REG_BINARY)
{
    (*env)->ReleaseByteArrayElements(env, (jarray) value,
        (jbyte*) cvalue, 0);
}
}

/* fonction d'aide au démarrage de l'énumération des noms */
static int startNameEnumeration(JNIEnv* env, jobject this_obj,
    jclass this_class)
{
    jfieldID id_index;
    jfieldID id_count;
    jfieldID id_root;
    jfieldID id_path;
    jfieldID id_hkey;
    jfieldID id_maxsize;

    HKEY root;
    jstring path;
    const char* cpath;
    HKEY hkey;
    DWORD maxsize = 0;
    DWORD count = 0;

    /* extraire les ID de champ */
    id_root = (*env)->GetFieldID(env, this_class, "root", "I");
    id_path = (*env)->GetFieldID(env, this_class, "path",
        "Ljava/lang/String;");
    id_hkey = (*env)->GetFieldID(env, this_class, "hkey", "I");
    id_maxsize = (*env)->GetFieldID(env, this_class, "maxsize",
        "I");
    id_index = (*env)->GetFieldID(env, this_class, "index",
        "I");

```


730 Au cœur de Java 2 - Fonctions avancées

```

    id_count = (*env)->GetFieldID(env, this_class, "count",
        "I");

    /* extraire les valeurs de champs */
    root = (HKEY)(*env)->GetIntField(env, this_obj, id_root);
    path = (jstring)(*env)->GetObjectField(env, this_obj,
        id_path);
    cpath = (*env)->GetStringUTFChars(env, path, NULL);

    /* ouvrir la clé de base de registres */
    if (RegOpenKeyEx(root, cpath, 0, KEY_READ, &hkey)
        != ERROR_SUCCESS)
    {
        (*env)->ThrowNew(env,
            (*env)->FindClass(env, "Win32RegKeyException"),
            "Echec de l'ouverture de clé");
        (*env)->ReleaseStringUTFChars(env, path, cpath);
        return -1;
    }
    (*env)->ReleaseStringUTFChars(env, path, cpath);

    /* demander compte et longueur max des noms */
    if (RegQueryInfoKey(hkey, NULL, NULL, NULL, NULL,
        NULL, NULL, &count, &maxsize, NULL, NULL, NULL)
        != ERROR_SUCCESS)
    {
        (*env)->ThrowNew(env,
            (*env)->FindClass(env, "Win32RegKeyException"),
            "Echec de la recherche d'infos de clé");
        RegCloseKey(hkey);
        return -1;
    }

    /* définir les valeurs de champ */
    (*env)->SetIntField(env, this_obj, id_hkey, (DWORD)hkey);
    (*env)->SetIntField(env, this_obj, id_maxsize, maxsize + 1);
    (*env)->SetIntField(env, this_obj, id_index, 0);
    (*env)->SetIntField(env, this_obj, id_count, count);
    return count;
}

JNIEXPORT jboolean JNICALL
Java_Win32RegKeyNameEnumeration_hasMoreElements
(JNIEnv* env, jobject this_obj)
{
    jclass this_class;
    jfieldID id_index;
    jfieldID id_count;
    int index;
    int count;
    /* extraire la classe */
    this_class = (*env)->GetObjectClass(env, this_obj);

    /* extraire les ID de champ */
    id_index = (*env)->GetFieldID(env, this_class, "index",
        "I");
    id_count = (*env)->GetFieldID(env, this_class, "count",
        "I");

    index = (*env)->GetIntField(env, this_obj, id_index);

```

```

    if (index == -1) /* première fois */
    {
        count = startNameEnumeration(env, this_obj, this_class);
        index = 0;
    }
    else
        count = (*env)->GetIntField(env, this_obj, id_count);
    return index < count;
}

JNIEXPORT jobject JNICALL
Java_Win32RegKeyNameEnumeration_nextElement
(JNIEnv* env, jobject this_obj)
{
    jclass this_class;
    jfieldID id_index;
    jfieldID id_hkey;
    jfieldID id_count;
    jfieldID id_maxsize;

    HKEY hkey;
    int index;
    int count;
    DWORD maxsize;

    char* cret;
    jstring ret;

    /* extraire la classe */
    this_class = (*env)->GetObjectClass(env, this_obj);

    /* extraire les ID de champ */
    id_index = (*env)->GetFieldID(env, this_class, "index",
        "I");
    id_count = (*env)->GetFieldID(env, this_class, "count",
        "I");
    id_hkey = (*env)->GetFieldID(env, this_class, "hkey", "I");
    id_maxsize = (*env)->GetFieldID(env, this_class, "maxsize",
        "I");

    index = (*env)->GetIntField(env, this_obj, id_index);
    if (index == -1) /* première fois */
    {
        count = startNameEnumeration(env, this_obj, this_class);
        index = 0;
    }
    else
        count = (*env)->GetIntField(env, this_obj, id_count);

    if (index >= count) /* déjà la fin */
    {
        (*env)->ThrowNew(env,
            (*env)->FindClass(env,
                "java/util/NoSuchElementException"),
            "Passé la fin de l'énumération");
        return NULL;
    }
}

```

732 *Au cœur de Java 2 - Fonctions avancées*

```

maxsize = (*env)->GetIntField(env, this_obj, id_maxsize);
hkey = (HKEY)(*env)->GetIntField(env, this_obj, id_hkey);
cret = (char*)malloc(maxsize);

/* rechercher le nom suivant */
if (RegEnumValue(hkey, index, cret, &maxsize, NULL, NULL,
    NULL, NULL) != ERROR_SUCCESS)
{
    (*env)->ThrowNew(env,
        (*env)->FindClass(env, "Win32RegKeyException"),
        "Echec énum valeur");
    free(cret);
    RegCloseKey(hkey);
    (*env)->SetIntField(env, this_obj, id_index, count);
    return NULL;
}

ret = (*env)->NewStringUTF(env, cret);
free(cret);

/* incrémenter index */
index++;
(*env)->SetIntField(env, this_obj, id_index, index);

if (index == count) /* à la fin */
{
    RegCloseKey(hkey);
}

return ret;
}

```

Exemple 9.23 : Win32RegKeyTest.java

```

import java.util.*;

public class Win32RegKeyTest
{
    public static void main(String[] args)
    {
        Win32RegKey key = new Win32RegKey(
            Win32RegKey.HKEY_LOCAL_MACHINE,
            "Software\\JavSoft\\Java Development Kit\\1.5");

        key.setValue("Utilisateur par défaut", "Bozo le clown");
        key.setValue("Chiffre porte-bonheur", new Integer(13));
        key.setValue("Nombres premiers", new byte[]
            { 2, 3, 5, 7, 11 });

        Enumeration<String> e = key.names();

        while (e.hasMoreElements())
        {
            String name = e.nextElement();
            System.out.print(name + " = ");

```

```
Object value = key.getValue(name);

if (value instanceof byte[])
    for (byte b : (byte[])value)
        System.out.print((b & 0xFF) + " ");
else
    System.out.print(value);

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

Fonctions de demande du type

- `jboolean IsAssignableFrom(JNIEnv *env, jclass c1, jclass c2)`
Renvoie `JNI_TRUE` si les objets de la première classe peuvent être affectés aux objets de la seconde classe ; `JNI_FALSE` sinon. C'est le cas si les classes sont identiques, si `c1` est une sous-classe de `c2`, ou si `c2` représente une interface qui est implémentée par `c1` ou l'une de ses superclasses.

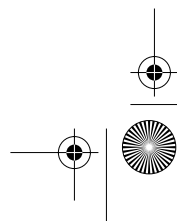
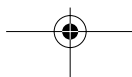
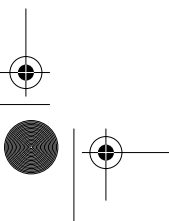
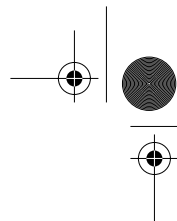
Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>c1, c2</code>	références de classe

- `jclass GetSuperClass(JNIEnv *env, jclass c1)`
Renvoie la superclasse d'une classe. Si `c1` représente la classe `Object` ou une interface, renvoie `NULL`.

Paramètres :

<code>env</code>	le pointeur de l'interface JNI
<code>c1</code>	une référence de classe



10

XML

Au sommaire de ce chapitre

- ✓ Une introduction au XML
- ✓ Analyse d'un document XML
- ✓ Validation des documents XML
- ✓ Localisation des informations avec XPATH
- ✓ Espaces de noms
- ✓ L'analyseur SAX
- ✓ Génération des documents XML
- ✓ Transformations XSL

La préface de l'ouvrage *Essential XML* par Don Box *et al.* (Addison-Wesley, 2000) précise, en plaisantant à peine : "Le langage XML (*Extensible Markup Language*) a remplacé Java, les modèles de conception et la technologie Objet et se présente comme la solution du marché du logiciel dont l'on connaît la gourmandise internationale". En effet, comme vous le verrez dans ce chapitre, XML constitue une technologie fort utile pour décrire des informations structurées. Les outils XML facilitent le traitement et la transformation de ces informations. Toutefois, il n'a pas pour autant réponse à tout. Vous avez besoin de standards spécifiques à un domaine et de bibliothèques de code pour l'utiliser efficacement. De plus, XML ne rend pas Java obsolète : XML fonctionne très bien avec Java. Pour l'essentiel, toutes les bibliothèques XML importantes ont été implémentées en premier lieu en Java, et nombre d'entre elles ne sont pas disponibles dans un autre langage de programmation. Depuis la fin des années 90, IBM, Apache et d'autres ont travaillé à produire des bibliothèques Java de grande qualité pour le traitement XML. Nouveauté dans JDK 1.4, Sun a intégré les bibliothèques les plus importantes dans la plate-forme Java 2, Edition Standard.

INFO

Vous pouvez télécharger l'API Java pour la bibliothèque de traitement XML (JAXP) à l'adresse <http://java.com/xml> pour ajouter les capacités de traitement XML à d'autres installations Java plus anciennes.

736 Au cœur de Java 2 - Fonctions avancées

Ce chapitre présente XML et traite des fonctions XML de la bibliothèque Java. Comme à chaque fois, nous soulignons au fur et à mesure les aspects qui justifient le battage qui entoure le XML et les moments où vous devrez résoudre les problèmes comme au bon vieux temps, grâce à un code bien conçu.

Une introduction à XML

Vous avez déjà vu plusieurs exemples de l'utilisation de *fichiers de propriété* (par exemple aux Chapitres 3 et 8) pour décrire la configuration d'un programme. Un fichier de propriétés contient un ensemble de paires nom/valeur, comme :

```
fontname=Times Roman
fontsize=12
windowsize=400 200
color=0 50 100
```

Vous pouvez utiliser la classe `Properties` pour lire un fichier de ce type à l'aide d'un seul appel de méthode. Cette fonction est agréable, mais elle n'est pas véritablement suffisante. Dans de nombreux cas, les informations à décrire possèdent plus de structures que le format du fichier de propriétés ne peut gérer en tout confort. Considérez les entrées `fontname/fontsize` dans l'exemple. Une seule entrée du type suivant permettrait une approche plus orientée objet :

```
font=Times Roman 12
```

Toutefois, l'analyse de la description de la police devient désagréable, vous devez deviner maintenant le moment où se termine le nom de police et où commence son corps.

Les fichiers de propriété disposent d'une seule hiérarchie uniforme. Vous verrez souvent les programmeurs surmonter cette difficulté grâce à des noms de clés tels que :

```
title.fontname=Helvetica
title.fontsize=36
body.fontname=Times Roman
body.fontsize=12
```

Autre inconvénient du format des fichiers de propriété : les clés doivent être uniques. Pour stocker une suite de valeurs, vous devrez utiliser un autre compromis, comme :

```
menu.item.1=Times Roman
menu.item.2=Helvetica
menu.item.3=Goudy Old Style
```

Le format XML résout ces problèmes, puisqu'il peut exprimer des structures hiérarchiques, une situation plus flexible que la structure de tableau uniforme d'un fichier de propriété.

Un fichier XML permettant de décrire une configuration de programme pourrait être le suivant :

```
<configuration>
  <title>
    <font>
      <name>Helvetica</name>
      <size>36</size>
    </font>
```

```
</title>
<body>
  <font>
    <name>Times Roman</name>
    <size>12</size>
  </font>
</body>
<window>
  <width>400</width>
  <height>200</height>
</window>
<color>
  <red>0</red>
  <green>50</green>
  <blue>100</blue>
</color>
<menu>
  <item>Times Roman</item>
  <item>Helvetica</item>
  <item>Goudy Old Style</item>
</menu>
</configuration>
```

Le format XML vous permet d'exprimer la structure de manière hiérarchique, ainsi que des éléments apparaissant à plusieurs reprises, sans grandes difficultés.

Comme vous pouvez le voir, le format d'un fichier XML est simple. Il ressemble d'ailleurs à un fichier HTML. Il existe une bonne raison à cela, les formats XML et HTML sont des descendants du grand SGML (*Standard Generalized Markup Language*).

SGML existe depuis les années 1970 et permet de décrire la structure de documents complexes. Il est utilisé avec un certain succès sur quelques marchés nécessitant de conserver en continu de grandes masses de documents, particulièrement sur le marché de l'aéronautique. Mais le SGML est assez complexe et n'a donc jamais été amené au premier plan. Une grande partie de cette complexité naît du fait que le SGML affiche deux objectifs contradictoires. Il veut s'assurer que les documents sont constitués conformément aux règles de leur type de document. Et il veut également faciliter la saisie de données, en autorisant des raccourcis. XML a en fait été conçu comme une version simplifiée du SGML, à utiliser sur Internet. Comme souvent, le plus simple est le mieux et le XML a immédiatement reçu un accueil enthousiaste et a depuis longtemps supplanté le SGML.

INFO

Vous trouverez une version très agréable de la norme XML, accompagnée d'annotations de Tim Bray, à l'adresse <http://www.xml.com/axml/axml.html>.

Même si XML et HTML ont des racines communes, il existe des différences importantes entre les deux.

- A la différence du HTML, XML est sensible à la casse. Par exemple <H1> et <h1> constituent des balises XML différentes.

738 Au cœur de Java 2 - Fonctions avancées

- Sous HTML, vous pouvez omettre les balises de fin comme dans `</p>` ou `</li>` si la fin d'un paragraphe ou d'un élément de liste est évidente, en fonction du contexte. En XML, cette situation est impossible.
- En XML, les éléments disposant d'une seule balise, sans balise de fin correspondante, doivent se terminer par une barre oblique (/), comme ``. De cette manière, l'analyseur sait qu'il ne doit pas rechercher de balise `</img>`.
- En XML, les valeurs d'attributs doivent être comprises entre guillemets, lesquels sont optionnels en HTML. Par exemple, `<applet code="MyApplet.class" width=300 height=300>` est autorisé en HTML, mais non en XML. En XML, il vous faudra utiliser des guillemets : `width="300"`.
- En HTML, il est possible d'obtenir des noms d'attributs sans valeurs, comme `<input type="radio" name="language" value="Java" checked>`. En XML, tous les attributs doivent posséder des valeurs comme `checked="true"` ou `checked="checked"`.

INFO

Les recommandations actuelles relatives aux documents Web et réalisées par le World Wide Web Consortium (W3C) concernent la norme XHTML, qui restreint la norme HTML à une compatibilité avec le XML. Vous trouverez une copie de la norme XHTML à l'adresse <http://www.w3.org/TR/xhtml1/>. XHTML est compatible en amont avec les navigateurs actuels, malheureusement de nombreux outils de création HTML actuels ne le supportent pas encore. Lorsque XHTML sera plus répandu, vous pourrez utiliser les outils XML décrits dans ce chapitre pour analyser des documents Web.

La structure d'un document XML

Un document XML doit commencer par un en-tête tel que :

```
<?xml version="1.0"?>
```

ou

```
<?xml version="1.0" encoding="UTF-8"?>
```

A strictement parler, l'en-tête est optionnel mais fortement recommandé.

INFO

Le SGML ayant été créé pour traiter de véritables documents, les fichiers XML sont appelés des *documents*, même si la plupart des fichiers XML décrivent des ensembles de données que l'on n'appellerait pas normalement des documents.

L'en-tête peut être suivi d'une *définition du type de document*, comme :

```
<!DOCTYPE Web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/Web-app_2_2.dtd">
```

Les définitions du type de document constituent un mécanisme important pour s'assurer de l'exactitude d'un document, mais ne sont pas obligatoires. Nous en discutons plus loin dans ce chapitre.

Enfin, le corps du document XML comprend l'*élément racine*, qui peut contenir d'autres éléments. Par exemple :

```
<?xml version="1.0"?>
<!DOCTYPE configuration . . .>
<configuration>
  <title>
    <font>
      <name>Helvetica</name>
      <size>36</size>
    </font>
  </title>
  . . .
</configuration>
```

Un élément peut contenir des *éléments enfant*, du texte, voire les deux. Dans l'exemple qui précède, les éléments font possèdent deux éléments enfant, name et size. L'élément name contient le texte "Helvetica".

ASTUCE

Il est plutôt recommandé de structurer vos documents XML afin qu'un élément contienne soit des éléments enfant, soit du texte. En d'autres termes, il convient d'éviter des situations comme :

```
<font>
  Helvetica
  <size>36</size>
</font>
```

Dans la spécification XML, il s'agit d'un *contenu mixte*. Comme vous le verrez plus loin dans ce chapitre, vous pouvez concevoir des définitions de types de documents plus propres si vous évitez le contenu mixte.

Les éléments XML peuvent contenir des attributs, comme :

```
<size unit="pt">36</size>
```

Les concepteurs XML ne sont pas tous d'accord sur le moment auquel utiliser des éléments ou des attributs. Par exemple, il semblerait plus facile de décrire une police comme ceci :

```
<font name="Helvetica" size="36"/>
```

que comme ceci :

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

Toutefois, les attributs sont bien moins flexibles. Supposons que vous vouliez ajouter des unités à la valeur de taille. Si vous utilisez des attributs, vous devez ajouter l'unité à la valeur de l'attribut :

```
<font name="Helvetica" size="36 pt"/>
```

740 Au cœur de Java 2 - Fonctions avancées

Vous devez maintenant analyser la chaîne "36 pt", l'un des inconvénients que XML était censé éviter. L'ajout d'un attribut à l'élément `size` est bien plus propre :

```
<font>
  <name>Helvetica</name>
  <size unit="pt">36</size>
</font>
```

Une règle souvent utilisée consiste à n'utiliser les attributs que lorsqu'ils modifient l'interprétation d'une valeur, et non pour spécifier des valeurs. Si vous vous retrouvez engagé dans des discussions métaphysiques sur le fait qu'un paramètre particulier soit une modification de l'interprétation d'une valeur ou non, arrêtez tout et refusez les attributs pour utiliser plutôt des éléments tout du long. De nombreuses très bonnes DTD n'utilisent pas du tout d'attributs.

INFO

En HTML, la règle d'utilisation des attributs est simple : s'il n'est pas affiché sur la page Web, c'est qu'il s'agit d'un attribut. Considérez par exemple le lien hypertexte

```
<a href="http://java.sun.com">Java Technology</a>
```

La chaîne `Java Technology` s'affiche sur la page Web, mais l'URL du lien ne fait pas partie de la page affichée. Toutefois, cette règle n'est pas vraiment utile pour la plupart des fichiers XML, puisque ses données n'ont pas normalement pour objectif d'être consultées par des personnes.

Les éléments et le texte sont l'essence des documents XML. Il existe quelques instructions de marquage que vous pouvez rencontrer.

- Les *références de caractères* sont de la forme `&#d` ; ou `&#xh` ;. Ici, *d* est une valeur Unicode décimale et *h* une valeur Unicode hexadécimale. Par exemple,

```
&#233;
&#x2122;
```

affichent les caractères é et TM.

- Les *références d'entités* ont la forme `&name` ;. Les références d'entités

```
&lt;
&gt;
&amp;
&quot;
&apos;
```

ont des significations prédéfinies : moins que (less than), plus que (greater than), & (esperluette), guillemets (quotation mark) et apostrophe. Vous pouvez définir d'autres références d'entité dans une DTD (*Document Type Definition*).

- Les *sections CDATA* sont délimitées par `<![CDATA[` et `]]>`. Il s'agit d'une forme spéciale de données de caractères. Vous pouvez les utiliser pour inclure des chaînes qui contiennent des caractères comme `<` `>` `&` sans les avoir interprétés sous forme de marquage, par exemple `<![CDATA[< & > sont mes délimiteurs préférés]]>`. Les sections CDATA ne peuvent pas contenir la chaîne `]]>`. Employez cette fonction avec précaution. Elle est trop souvent utilisée comme porte dérobée pour contrefaire des données existantes dans des documents XML.

- Les *instructions de traitement* sont délimitées par `<? et ?>`, par exemple :

- `<?xml-stylesheet href="mystyle.css" type="text/css"?>`

Ces instructions profitent à l'application qui traite le document XML. Chaque document XML commence par une instruction de traitement :

```
<?xml version="1.0"?>
```

- Les *commentaires* sont délimités par `<!-- et -->`, par exemple :

```
<!--Voici un commentaire. -->
```

Les commentaires ne doivent pas contenir la chaîne `--`. Il s'agit uniquement d'informations pour les lecteurs humains. Ils ne doivent jamais contenir de commandes cachées. Utilisez des instructions de traitement pour les commandes.

Analyse d'un document XML

Pour traiter un document XML, il convient de *l'analyser*. Un analyseur est un programme qui lit un fichier, confirme que le fichier a le format correct, le décompose en fonction des éléments constitutifs et permet au programmeur d'accéder à ces éléments. La bibliothèque Java fournit deux types d'analyseurs XML :

- l'analyseur DOM (*Document Object Model*), qui lit un document XML dans une structure arborescente ;
- l'analyseur Simple API for XML (SAX), qui génère des événements en lisant un document XML.

L'analyseur DOM est plus simple à utiliser dans la plupart des cas et nous allons le traiter en premier lieu. Il est à envisager si vous travaillez sur de très longs documents dont les structures arborescentes utilisent une grande quantité de mémoire ou si vous êtes simplement intéressé par quelques éléments et que vous ne vous préoccupez pas de son contexte.

L'interface de l'analyseur DOM est standardisée par le consortium World Wide Web (W3C). Le package `org.w3c.dom` contient les définitions des types d'interface comme `Document` et `Element`. Différents fournisseurs, notamment Apache Organization et IBM, ont rédigé des analyseurs DOM dont les classes implémentent ces interfaces. L'API Java de Sun pour la bibliothèque de traitement XML (JAXP) permet en fait de brancher n'importe lequel de ces analyseurs. Toutefois, Sun inclut également son propre analyseur DOM dans le SDK Java. Nous utiliserons donc l'analyseur Sun dans ce chapitre.

Pour lire un document XML, vous aurez besoin d'un objet `DocumentBuilder` que vous obtiendrez à partir d'un `DocumentBuilderFactory`, comme ceci :

```
DocumentBuilderFactory factory
    = DocumentBuilderFactory.newInstance();
DocumentBuilder builder
    = factory.newDocumentBuilder();
```

Vous pouvez maintenant lire un document à partir d'un fichier :

```
File f = . . .
Document doc = builder.parse(f);
```

742 Au cœur de Java 2 - Fonctions avancées

Vous pouvez également utiliser une URL :

```
URL u = . . .
Document doc = builder.parse(u);
```

Vous pouvez même spécifier un flux d'entrée arbitraire :

```
InputStream in = . . .
Document doc = builder.parse(in);
```

INFO

Si vous utilisez un flux comme source d'entrée, l'analyseur ne pourra pas retrouver d'autres fichiers référencés relativement à l'emplacement du document, comme une DTD, dans le même répertoire. Il est possible d'installer un "système de résolution d'entités" pour surmonter ce problème.

L'objet Document constitue une représentation en mémoire de la structure arborescente du document XML. Il est composé d'objets dont les classes implémentent l'interface Node et ses diverses sous-interfaces. La Figure 10.1 présente la hiérarchie d'héritage des sous-interfaces.

Vous commencez par analyser le contenu d'un document en appelant la méthode `getDocumentElement`. Cela renvoie l'élément racine.

```
Element root = doc.getDocumentElement();
```

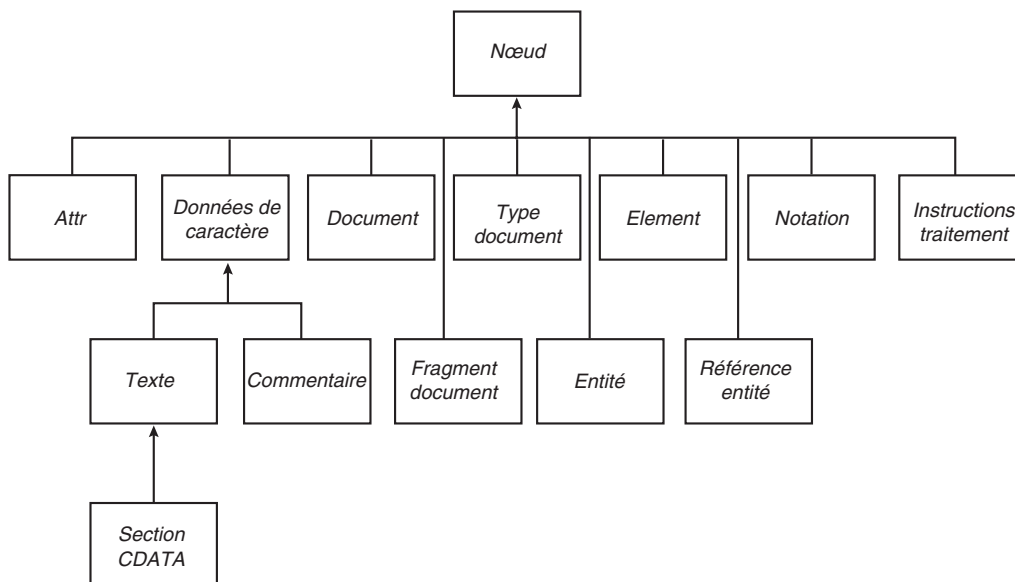


Figure 10.1

Les interfaces Node et leurs sous-interfaces.

Par exemple, si vous traitez un document

```
<?xml version="1.0"?>*
<font>
    . . .
</font>
```

l'appel à `getDocumentElement` renvoie l'élément `font`.

La méthode `getTagName` renvoie le nom de balise d'un élément. Dans l'exemple précédent, `root.getTagName()` renvoie la chaîne `"font"`.

Pour obtenir les enfants de l'élément (qui peuvent être des sous-éléments, du texte, des commentaires ou d'autres nœuds), utilisez la méthode `getChildNodes`. Cette méthode renvoie une collection du type `NodeList`. Ce type a été inventé avant les collections standard Java et possède un protocole d'accès différent. La méthode `item` obtient l'élément avec un index donné et la méthode `getLength` donne le nombre total d'éléments. Ainsi, vous pouvez énumérer tous les enfants comme ceci :

```
NodeList children = root.getChildNodes();
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    . . .
}
```

Prenez garde lorsque vous analysez les enfants. Supposons que vous traitiez le document

```
<font>
    <name>Helvetica</name>
    <size>36</size>
</font>
```

Vous pourriez vous attendre à ce que l'élément `font` ait deux enfants, mais l'analyseur en signale cinq :

- le blanc entre `<font>` et `<name>` ;
- l'élément `name` ;
- le blanc entre `</name>` et `<size>` ;
- l'élément `size` ;
- le blanc entre `</size>` et `</font>`.

La Figure 10.2 présente l'arbre DOM.

Si vous ne vous attendez à trouver que des sous-éléments, vous pouvez ignorer le blanc :

```
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    if (child instanceof Element)
    {
        Element childElement = (Element)child;
        . . .
    }
}
```

Maintenant, vous n'étudiez que deux éléments, avec les noms de balise `name` et `size`.

744 Au cœur de Java 2 - Fonctions avancées

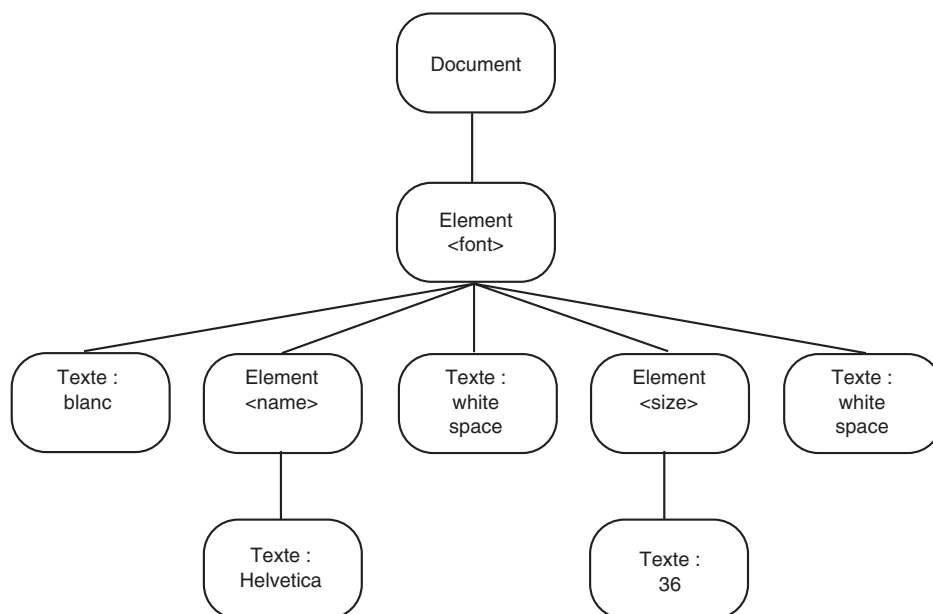


Figure 10.2

Un arbre DOM simple.

Comme vous le verrez dans la section suivante, vous pouvez faire encore mieux si votre document dispose d'une définition du type de document. L'analyseur sait alors lesquels des éléments ne disposent pas de nœuds de texte comme enfants et peut supprimer le blanc pour vous.

Lorsque vous analysez les éléments `name` et `size`, vous voulez récupérer les chaînes de texte qu'ils contiennent. Ces chaînes de texte sont elles-mêmes contenues dans des nœuds enfant de type `Text`. Puisque vous savez que ces nœuds `Text` sont les seuls enfants, vous pouvez utiliser la méthode `getFirstChild` sans avoir à traverser un autre `NodeList`. Utilisez ensuite la méthode `getData` pour récupérer la chaîne stockée dans le nœud `Text`.

```

for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    if (child instanceof Element)
    {
        Element childElement = (Element) child;
        Text textNode = (Text) childElement.getFirstChild();
        String text = textNode.getData().trim();
        if (childElement.getTagName().equals("name"))
            name = text;
        else if (childElement.getTagName().equals("size"))
            size = Integer.parseInt(text);
    }
}
    
```

ASTUCE

Il est conseillé d'appeler `trim` sur la valeur de retour de la méthode `getData`. Si l'auteur d'un fichier XML place la balise de début et de fin sur des lignes séparées, comme

```
<size>
    36
</size>
```

l'analyseur inclut alors tous les sauts de ligne et les espaces dans les données du nœud de texte. L'appel à la méthode `trim` supprime le blanc entourant les données réelles.

Vous pouvez également obtenir le dernier enfant avec la méthode `getLastChild` et le prochain frère d'un nœud avec `getNextSibling`. Ainsi, une autre manière de traverser un ensemble de nœuds enfant est

```
for (Node childNode = element.getFirstChild();
     childNode != null;
     childNode = childNode.getNextSibling())
{
    . . .
}
```

Pour énumérer les attributs d'un nœud, appelez la méthode `getAttributes`. Elle renvoie un objet `NamedNodeMap` qui contient les objets `Node` décrivant les attributs. Vous pouvez traverser les nœuds dans un `NamedNodeMap` de la même manière qu'avec `NodeList`. Appelez ensuite les méthodes `getNodeName` et `getNodeValue` pour obtenir les noms et valeurs d'attributs.

```
NamedNodeMap attributes = element.getAttributes();
for (int i = 0; i < attributes.getLength(); i++)
{
    Node attribute = attributes.item(i);
    String name = attribute.getNodeName();
    String value = attribute.getNodeValue();
    . . .
}
```

Si vous connaissez le nom d'un attribut, vous pouvez également récupérer directement la valeur correspondante :

```
String unit = element.getAttribute("unit");
```

Vous avez maintenant vu comment analyser un arbre DOM. Le programme de l'Exemple 10.1 utilise ces techniques. Vous pouvez recourir à l'option de menu Fichier > Ouvrir pour lire un fichier XML. Un objet `DocumentBuilder` analyse le fichier XML et produit un objet `Document`. Le programme affiche l'objet `Document` sous forme de `JTree` (voir Figure 10.3).

L'affichage en arbre présente clairement la manière dont les éléments enfant sont entourés par du texte contenant un blanc et des commentaires. Pour plus de clarté, le programme affiche une nouvelle ligne et des caractères de retour comme `\n` et `\r` (faute de quoi, ils afficheraient des boîtes creuses, le symbole par défaut pour un caractère que Swing ne peut pas dessiner dans une chaîne).

Si vous avez suivi la section sur les arbres au Chapitre 6, vous reconnaîtrez les techniques utilisées par ce programme pour afficher l'arbre. La classe `DOMTreeModel` implémente l'interface `TreeModel`. La méthode `getRoot` renvoie l'élément racine du document. La méthode `getChild` obtient la liste de nœuds des enfants et renvoie l'élément avec l'index exigé. Toutefois, le modèle d'arbre

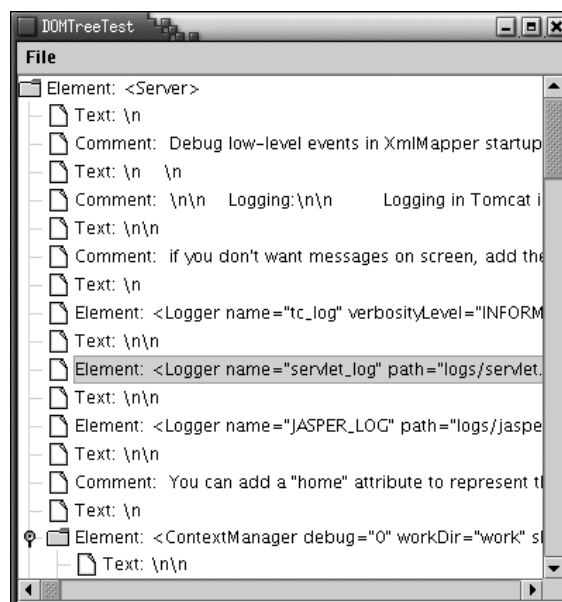
746 Au cœur de Java 2 - Fonctions avancées

renvoie les objets `Node` de DOM dont les méthodes `toString` ne sont pas nécessairement descriptives. Ainsi, le programme installe un afficheur de cellules d'arbre qui étend l'afficheur de cellule par défaut et définit le texte de l'étiquette sur une forme plus descriptive. L'afficheur de cellule affiche ce qui suit :

- pour les éléments, le nom de balise de l'élément et un tableau de tous les attributs ;
- pour les données de caractères, l'interface (`Text`, `Comment` ou `CDATASection`), suivie des données, avec les nouvelles lignes et les caractères de retour remplacés par `\n` et `\r` ;
- pour tous les autres types de nœuds, le nom de classe suivi du résultat de `toString`.

Figure 10.3

Un arbre d'analyse d'un document XML.



Exemple 10.1 : DOMTreeTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;
import javax.swing.tree.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;

/**
 * Ce programme affiche un document XML sous forme d'arborescence.
 */
public class DOMTreeTest
{
    public static void main(String[] args)
```

```

    {
        JFrame frame = new DOMTreeFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce bloc contient une arborescence affichant le contenu d'un document
 * XML.
 */
class DOMTreeFrame extends JFrame
{
    public DOMTreeFrame()
    {
        setTitle("DOMTreeTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        JMenu fileMenu = new JMenu("Fichier");
        JMenuItem openItem = new JMenuItem("Ouvrir");
        openItem.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                { openFile(); }
            });
        fileMenu.add(openItem);

        JMenuItem exitItem = new JMenuItem("Quitter");
        exitItem.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                { System.exit(0); }
            });
        fileMenu.add(exitItem);

        JMenuBar menuBar = new JMenuBar();
        menuBar.add(fileMenu);
        setJMenuBar(menuBar);
    }

    /**
     * Ouvre un fichier et charge le document.
     */
    public void openFile()
    {
        JFileChooser chooser = new JFileChooser();
        chooser.setCurrentDirectory(new File("."));

        chooser.setFileFilter(new
            javax.swing.filechooser.FileFilter()
            {
                public boolean accept(File f)
                {
                    return f.isDirectory()
                        || f.getName().toLowerCase().endsWith(".xml");
                }
                public String getDescription()
            }
        );
    }
}

```

748 *Au cœur de Java 2 - Fonctions avancées*

```

        { return "Fichiers XML"; }
    });
    int r = chooser.showOpenDialog(this);
    if (r != JFileChooser.APPROVE_OPTION) return;
    File f = chooser.getSelectedFile();
    try
    {
        if (builder == null)
        {
            DocumentBuilderFactory factory
                = DocumentBuilderFactory.newInstance();
            builder = factory.newDocumentBuilder();
        }

        Document doc = builder.parse(f);
        JTree tree = new JTree(new DOMTreeModel(doc));
        tree.setCellRenderer(new DOMTreeCellRenderer());

        setContentPane(new JScrollPane(tree));
        validate();
    }
    catch (IOException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }
    catch (ParserConfigurationException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }
    catch (SAXException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }
}

private DocumentBuilder builder;
private static final int DEFAULT_WIDTH = 400;
private static final int DEFAULT_HEIGHT = 400;
}

/**
 * Ce modèle d'arborescence décrit la structure
 * d'un document XML.
 */
class DOMTreeModel implements TreeModel
{
    /**
     * Construit un modèle d'arbre de document.
     * @param doc Le document
     */
    public DOMTreeModel(Document doc)
    { this.doc = doc; }

    public Object getRoot()
    { return doc.getDocumentElement(); }

    public int getChildCount(Object parent)
    {
        Node node = (Node) parent;

```

```

        NodeList list = node.getChildNodes();
        return list.getLength();
    }

    public Object getChild(Object parent, int index)
    {
        Node node = (Node) parent;
        NodeList list = node.getChildNodes();
        return list.item(index);
    }

    public int getIndexOfChild(Object parent, Object child)
    {
        Node node = (Node) parent;
        NodeList list = node.getChildNodes();
        for (int i = 0; i < list.getLength(); i++)
            if (getChild(node, i) == child)
                return i;
        return -1;
    }

    public boolean isLeaf(Object node)
    {
        return getChildCount(node) == 0;
    }

    public void valueForPathChanged(TreePath path,
        Object newValue) {}

    public void addTreeModelListener(TreeModelListener l) {}

    public void removeTreeModelListener(TreeModelListener l) {}

    private Document doc;
}

/**
 * Cette classe renvoie un noeud XML.
 */
class DOMTreeCellRenderer extends DefaultTreeCellRenderer
{
    public Component getTreeCellRendererComponent(JTree tree,
        Object value, boolean selected, boolean expanded,
        boolean leaf, int row, boolean hasFocus)
    {
        Node node = (Node) value;
        if (node instanceof Element) return elementPanel((Element) node);

        super.getTreeCellRendererComponent(tree, value,
            selected, expanded, leaf, row, hasFocus);
        if (node instanceof CharacterData)
            setText(characterString((CharacterData) node));
        else
            setText(node.getClass() + ": " + node.toString());
        return this;
    }
}

```

750 *Au cœur de Java 2 - Fonctions avancées*

```

public static JPanel elementPanel(Element e)
{
    JPanel panel = new JPanel();
    panel.add(new JLabel("Élément : " + e.getTagName()));
    panel.add(new JTable(new AttributeTableModel(e.getAttributes())));
    return panel;
}

public static String characterString(CharacterData node)
{
    StringBuilder builder = new StringBuilder(node.getData());
    for (int i = 0; i < builder.length(); i++)
    {
        if (builder.charAt(i) == '\r')
        {
            builder.replace(i, i + 1, "\\r");
            i++;
        }
        else if (builder.charAt(i) == '\n')
        {
            builder.replace(i, i + 1, "\\n");
            i++;
        }
        else if (builder.charAt(i) == '\t')
        {
            builder.replace(i, i + 1, "\\t");
            i++;
        }
    }
    if (node instanceof CDATASection)
        builder.insert(0, "CDATASection : ");
    else if (node instanceof Text)
        builder.insert(0, "Commentaire : ");

    return builder.toString();
}

/**
 * Ce modèle de tableau décrit les attributs d'un élément XML.
 */
class AttributeTableModel extends AbstractTableModel
{
    /**
     * Construit un modèle de tableau d'attributs.
     * @param map la carte de noeud nommée
     */
    public AttributeTableModel(NamedNodeMap map) { this.map = map; }

    public int getRowCount() { return map.getLength(); }
    public int getColumnCount() { return 2; }
    public Object getValueAt(int r, int c)
    {
        return c == 0 ? map.item(r).getNodeName() :
            map.item(r).getNodeValue();
    }

    private NamedNodeMap map;
}

```

javax.xml.parsers.DocumentBuilderFactory 1.4

- `static DocumentBuilderFactory newInstance()`
Renvoie une instance de la classe `DocumentBuilderFactory`.
- `DocumentBuilder newDocumentBuilder()`
Renvoie une instance de la classe `DocumentBuilder`.

javax.xml.parsers.DocumentBuilder 1.4

- `Document parse(File f)`
- `Document parse(String url)`
- `Document parse(InputStream in)`
Analysent un document XML depuis un fichier donné, une URL ou un flux d'entrée et renvoient le document analysé.

org.w3c.dom.Document 1.4

- `Element getDocumentElement()`
Renvoie l'élément racine du document.

org.w3c.dom.Element 1.4

- `String getTagName()`
Renvoie le nom de l'élément.
- `String getAttribute(String name)`
Renvoie la valeur d'attribut de l'attribut avec le nom donné ou la chaîne vide s'il n'y a pas d'attribut de ce type.

org.w3c.dom.Node 1.4

- `NodeList getChildNodes()`
Renvoie une liste de nœuds contenant tous les enfants de ce nœud.
- `Node getFirstChild()`
- `Node getLastChild()`
Extraient le premier ou le dernier nœud enfant de ce nœud ou `null` si le nœud n'a pas d'enfant.
- `Node getNextSibling()`
- `Node getPreviousSibling()`
Extraient le prochain frère ou le précédent de ce nœud ou `null` si ce nœud n'a pas de frère.
- `Node getParentNode()`
Extrait le parent de ce nœud ou `null` si ce nœud est le nœud du document.
- `NamedNodeMap getAttributes()`
Renvoie une carte de nœuds contenant les nœuds `Attr` qui décrivent tous les attributs de ce nœud.
- `String getNodeName()`
Renvoie le nom de ce nœud. Si le nœud est un nœud `Attr`, le nom est celui de l'attribut.
- `String getNodeValue()`
Renvoie la valeur de ce nœud. Si le nœud est un nœud `Attr`, la valeur correspond à celle de l'attribut.

752 *Au cœur de Java 2 - Fonctions avancées*

org.w3c.dom.CharacterData 1.4

- `String getData()`
Renvoie le texte stocké dans ce nœud.

org.w3c.dom.NodeList 1.4

- `int getLength()`
Renvoie le nombre de nœuds dans cette liste.
- `Node item(int index)`
Renvoie le nœud de l'index donné. L'index se trouve entre 0 et `getLength() - 1`.

org.w3c.NamedNodeMap 1.4

- `int getLength()`
Renvoie le nombre de nœuds de cette carte.
- `Node item(int index)`
Renvoie le nœud avec l'index donné. L'index se trouve entre 0 et `getLength() - 1`.

Valider les documents XML

Dans la section précédente, vous avez vu comment traverser la structure en arbre d'un document DOM. Toutefois, si vous suivez simplement cette approche, vous découvrirez que vous allez devoir réaliser une partie assez ennuyeuse de programmation et de vérification des erreurs. Non seulement vous devrez traiter les blancs entre les éléments, mais vous devrez également vérifier que le document contient les nœuds auxquels vous vous attendez. Par exemple, supposons que vous lisiez un élément du style :

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

Vous obtenez le premier enfant. Attention, c'est un nœud de texte contenant un espace blanc "`\n`". Ignorez les nœuds de texte et retrouvez le premier nœud d'élément. Vous devez ensuite vérifier que son nom de balise est "`name`", puis qu'il a un nœud enfant du type `Text`. Vous passez alors au prochain enfant qui ne soit pas un blanc et vous effectuez la même vérification. Que se passera-t-il si l'auteur du document a mélangé l'ordre des enfants ou ajouté un autre élément enfant ? Il sera alors très ennuyeux de coder toutes les vérifications d'erreurs, et dangereux d'ignorer toutes les vérifications.

Heureusement, l'un des principaux avantages d'un analyseur XML réside dans le fait qu'il est capable de vérifier automatiquement la structure d'un document. L'analyse devient alors beaucoup plus simple. Par exemple, si vous savez que le fragment `font` a été validé, vous pouvez simplement obtenir les deux petits-enfants, les diffuser sous forme de nœuds `Text` et obtenir les données de texte, sans autre vérification.

Pour indiquer la structure du document, vous fournissez une DTD ou une définition XML Schema, contenant les règles qui expliquent la formation d'un document en spécifiant les éléments enfant

autorisés ainsi que les attributs de chaque élément. Par exemple, une DTD pourrait contenir une règle du type :

```
<!ELEMENT font (name,size)>
```

Cette règle exprime qu'un élément `font` doit toujours avoir deux enfants, des éléments `name` et `size`. Le langage XML Schema exprime la même contrainte que

```
<xsd:element name="font">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
  </xsd:sequence>
</xsd:element>
```

XML Schema est en mesure d'exprimer des conditions de validation plus sophistiquées (comme le fait que la taille de l'élément doive contenir un entier) que les DTD. A la différence de la syntaxe des DTD, la syntaxe XML Schema utilise XML, ce qui est un avantage si vous devez traiter des fichiers de schéma.

Le langage XML Schema avait pour objet de remplacer les DTD. Toutefois, à l'heure où nous rédigeons, les DTD sont toujours fort utilisées. XML Schema est un langage complexe, et tous les analystes ne l'acceptent pas. La prise en charge de XML Schema a été ajoutée au JDK 5.0 mais, comme vous le verrez, l'implémentation est un peu difficile.

INFO

Certains utilisateurs du XML sont tellement ennuyés par la complexité de XML Schema qu'ils font appel à d'autres langages de validation. Le plus fréquent s'intitule Relax NG (<http://www.relaxng.org>).

Dans la prochaine section, nous verrons les DTD plus en détail. Nous traiterons ensuite brièvement des bases de la prise en charge de XML Schema. Enfin, nous vous présenterons une application complète, montrant en quoi la validation simplifie la programmation XML.

Définitions du type de document (DTD)

Il existe plusieurs méthodes permettant de fournir une DTD. Vous pouvez l'inclure dans un document XML comme ceci :

```
<?xml version="1.0"?>
<!DOCTYPE configuration [
  <!ELEMENT configuration . . .>
  autres règles
  . . .
]>
<configuration>
  . . .
</configuration>
```

Comme vous pouvez le voir, les règles sont incluses à l'intérieur d'une déclaration `DOCTYPE`, dans un bloc délimité par les signes `[ . . . ]`. Le type de document doit correspondre au nom de l'élément racine, comme `configuration`, dans notre exemple.

754 *Au cœur de Java 2 - Fonctions avancées*

Fournir une DTD à l'intérieur d'un document XML est assez inhabituel car les DTD peuvent être longues. Il est plus logique de les stocker en externe. Vous pouvez utiliser la déclaration SYSTEM à cette fin. Vous spécifiez une URL qui contient la DTD, par exemple :

```
<!DOCTYPE configuration SYSTEM "config.dtd">
```

ou

```
<!DOCTYPE configuration SYSTEM "http://myserver.com/config.dtd">
```

ATTENTION

Si vous utilisez une URL relative pour la DTD (comme "config.dtd"), vous donnez à l'analyseur un objet `File` ou URL, et non un objet `InputStream`. Si vous devez analyser à partir d'un flux d'entrée, fournissez un système de résolution d'entité ; reportez-vous à la note suivante.

Enfin, le mécanisme permettant d'identifier des DTD connues à son origine en SGML. En voici un exemple :

```
<!DOCTYPE Web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/Web-app_2_2.dtd">
```

Si un processeur XML sait comment retrouver la DTD avec l'identificateur public, il n'a pas besoin d'accéder à l'URL.

INFO

Si vous utilisez un analyseur DOM et que vous souhaitez prendre en charge un identificateur PUBLIC, appelez la méthode `setEntityResolver` de la classe `DocumentBuilder` pour installer un objet d'une classe qui implémente l'interface `EntityResolver`. Cette interface dispose d'une méthode unique, `resolveEntity`. Voici les grandes lignes d'une implémentation typique :

```
class MyEntityResolver implements EntityResolver
{
    public InputSource resolveEntity(String publicID,
        String systemID)
    {
        if (publicID.equals(une ID connue))
            return new InputSource(données de la DTD);
        else
            return null; // utiliser le comportement par défaut
    }
}
```

Vous pouvez construire la source d'entrée à partir d'un `InputStream`, d'un `Reader` ou d'une chaîne.

Vous savez désormais comment l'analyseur retrouve la DTD, examinons maintenant les diverses sortes de règles.

La règle `ELEMENT` spécifie ce que les enfants d'un élément peuvent avoir. Vous spécifiez une expression ordinaire, constituée des composants affichés au Tableau 10.1.

Tableau 10.1 : Règles pour un contenu d'élément

<i>Règle</i>	<i>Signification</i>
E^*	0 ou plusieurs occurrences de E
E^+	1 ou plusieurs occurrences de E
$E?$	0 ou 1 occurrence de E
$E_1 E_2 \dots E_n$	Parmi $E_1, E_2, \dots, E_n$
$E_1, E_2, \dots, E_n$	E_1 suivi de $E_2, \dots, E_n$
<code>#PCDATA</code>	Texte
$(\#PCDATA E_1 E_2 \dots E_n)^*$	0 ou plusieurs occurrences du texte et de $E_1, E_2, \dots, E_n$ dans n'importe quel ordre (contenu mixte)
<code>ANY</code>	N'importe quel enfant autorisé
<code>EMPTY</code>	Aucun enfant autorisé

Voici plusieurs exemples simples, mais habituels. La règle suivante indique qu'un élément `menu` contient 0 ou plusieurs éléments `item`.

```
<!ELEMENT menu (item)*>
```

Cet ensemble de règles indique qu'une police est décrite par un nom, suivi d'une taille, chacun contenant du texte.

```
<!ELEMENT font (name,size)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT size (#PCDATA)>
```

L'abréviation `PCDATA` montre des données de caractères analysées. Ces données sont dites "analysées" car l'analyseur interprète la chaîne de texte, à la recherche des caractères `<` qui montrent le début d'une nouvelle balise ou des caractères `&` qui montrent le début d'une entité.

Une spécification d'élément peut contenir des expressions ordinaires imbriquées et complexes. Par exemple, voici une règle décrivant la constitution d'un chapitre de cet ouvrage :

```
<!ELEMENT chapter (intro,(heading,(para|image|table|note)+)+)>
```

Chaque chapitre commence par une introduction, suivie d'une ou plusieurs sections constituées d'un titre et d'un ou plusieurs paragraphes, images, tableaux ou notes.

Toutefois, il existe un cas commun où vous ne pouvez pas définir les règles de manière aussi flexible que vous l'aimeriez. Dès qu'un élément peut contenir du texte, il n'y a que deux cas valides : soit l'élément ne contient que du texte, comme

```
<!ELEMENT name (#PCDATA)>
```

756 Au cœur de Java 2 - Fonctions avancées

soit l'élément contient *n'importe quelle combinaison de texte et de balises dans n'importe quel ordre*, comme :

```
<!ELEMENT para (#PCDATA|em|strong|code)*>
```

Il est interdit de spécifier d'autres types de règles qui contiennent #PCDATA. Par exemple, la règle suivante est interdite :

```
<!ELEMENT captionedImage (image,#PCDATA)>
```

Vous devez récrire cette règle, soit en introduisant un autre élément caption, soit en autorisant n'importe quelle combinaison de balises image et de texte.

Cette restriction simplifie la tâche de l'analyseur XML au moment d'analyser un contenu mixte (un mélange de balises et de texte). Etant donné que vous perdez une partie du contrôle lorsque vous autorisez le contenu mixte, il est plutôt conseillé de concevoir des DTD pour que tous les éléments contiennent soit d'autres éléments, soit uniquement du texte.

INFO

En fait, il n'est pas tout à fait vrai que vous pouvez spécifier des expressions ordinaires arbitraires d'éléments dans une règle DTD. Un analyseur XML peut refuser certains ensembles de règles complexes qui amènent à une analyse "non-déterministe". Supposons une expression ordinaire $((x, y) \mid (x, z))$ qui soit non-déterministe. Lorsque l'analyseur voit x , il ne sait pas laquelle des deux alternatives choisir. Cette expression peut être réécrite sous forme déterministe, comme $(x, (y \mid z))$. Toutefois, il existe quelques expressions qui ne peuvent pas être reformulées, comme $((x, y) * \mid x?)$. L'analyseur Sun n'apporte aucun avertissement lorsqu'il reçoit une DTD ambiguë. Il choisit simplement la première possibilité correspondante lors de l'analyse, ce qui l'amène à refuser certaines entrées correctes. Bien entendu, l'analyseur est tout à fait en droit de le faire, puisque la norme XML permet à un analyseur de supposer que la DTD ne présente aucune ambiguïté.

Dans la pratique, ce problème ne doit pas particulièrement vous inquiéter puisque la plupart des DTD sont tellement simples que vous ne rencontrerez jamais de problèmes d'ambiguïté.

Vous spécifiez également les règles permettant de décrire les attributs autorisés des éléments. La syntaxe générale est la suivante :

```
<!ATTLIST element attribute type default>
```

Le Tableau 10.2 présente les types d'attributs autorisés et le Tableau 10.3, la syntaxe des paramètres par défaut.

Tableau 10.2 : Types d'attributs

Type	Signification
CDATA	Toute chaîne de caractères
$(A_1 A_2 ... A_n)$	Un attribut de chaîne parmi $A_1, A_2, ... A_n$
NMTOKEN, NMTOKENS	Un ou plusieurs tokens de noms
ID	Une ID unique
IDREF, IDREFS	Une ou plusieurs références à une ID unique
ENTITY, ENTITIES	Une ou plusieurs entités non-analysées

Tableau 10.3 : Paramètres par défaut des attributs

<i>Paramètre par défaut</i>	<i>Signification</i>
#REQUIRED	Attribut exigé.
#IMPLIED	Attribut optionnel.
A	Attribut optionnel, l'analyseur le signale comme A s'il n'est pas spécifié.
#FIXED A	L'attribut doit être soit non-spécifié, soit A. Dans l'un ou l'autre cas, l'analyseur le signale comme A.

Voici deux spécifications habituelles d'attributs :

```
<!ATTLIST font style (plain|bold|italic|bold-italic) "plain">
<!ATTLIST size unit CDATA #IMPLIED>
```

La première spécification décrit l'attribut `style` d'un élément `font`. Il existe quatre valeurs d'attributs légaux et la valeur par défaut est `plain`. La deuxième spécification exprime que l'attribut `unit` de l'élément `size` peut contenir n'importe quelle donnée de caractère.

INFO

Nous conseillons généralement d'utiliser les éléments, et non les attributs, pour décrire les données. Suite à cette recommandation, le style de police doit être un élément séparé, comme `<font><style>plain</style>...</font>`. Toutefois, les attributs présentent un avantage indéniable pour les types énumérés, puisque l'analyseur peut vérifier que les valeurs sont autorisées. Par exemple, si le style de police est un attribut, l'analyseur vérifie qu'il a l'une des quatre valeurs autorisées et donne une valeur par défaut si rien n'est fourni.

La gestion d'une valeur d'attribut `CDATA` diffère légèrement du traitement de `#PCDATA` que vous avez vu précédemment, et reste peu lié aux sections `<![CDATA[...]]>`. La valeur d'attribut est d'abord *normalisée*, c'est-à-dire que l'analyseur traite les références de caractère et d'entité (comme `é` ou `<t;`) et remplace les blancs par des espaces.

Un `NMTOKEN` (ou token de nom) est identique à `CDATA`, mais la plupart des caractères alphanumériques et des blancs internes sont interdits et l'analyseur supprime les blancs d'interlignage et de fin. `NMTOKENS` est une liste de tokens de noms séparés par des blancs.

La construction de `ID` est assez utile. Il s'agit d'un token de noms qui doit être unique dans le document, l'analyseur vérifiant d'ailleurs son unicité. Vous découvrirez une application dans le prochain programme d'exemple. Un `IDREF` est une référence à une `ID` qui existe dans le même document, que l'analyseur vérifie également. `IDREFS` constitue une liste de références `ID` séparées par des blancs.

Une valeur d'attribut `ENTITY` fait référence à une "entité externe non-analysée". Il s'agit d'un reste du SGML, rarement utilisé dans la pratique. La spécification XML annotée à l'adresse <http://www.xml.com/axml/axml.html> en fournit un exemple.

Une DTD peut également définir des *entités* ou abréviations qui sont remplacées au cours de l'analyse. Vous trouverez un bon exemple d'utilisation des entités dans les descriptions d'interface

758 *Au cœur de Java 2 - Fonctions avancées*

utilisateur pour le navigateur Mozilla/Netscape 6. Ces descriptions sont mises en forme en XML et contiennent des définitions d'entité, telles que :

```
<!ENTITY back.label "Back">
```

Ailleurs, le texte peut contenir une référence d'entité, comme :

```
<menuitem label="&back.label;" />
```

L'analyseur remplace la référence d'entité avec la chaîne de remplacement. Pour internationaliser l'application, seule la chaîne de la définition de l'entité doit être modifiée. Il existe d'autres utilisations des entités qui sont plus complexes et moins souvent utilisées. Etudiez la spécification XML pour connaître les détails.

Ainsi s'achève notre introduction sur les DTD. Vous savez maintenant comment les utiliser, vous pouvez configurer votre analyseur afin d'en profiter. Indiquez tout d'abord au factory de construction du document d'activer la validation.

```
factory.setValidating(true);
```

Tous les constructeurs produits par ce factory valideront leur entrée par rapport à une DTD. L'avantage le plus utile de la validation consiste à ignorer les blancs dans un contenu d'élément. Par exemple, considérez le fragment XML

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

Un analyseur sans validation signale le blanc présent entre les éléments `font`, `name` et `size` car il ne peut pas savoir si les enfants de `font` sont :

```
(name,size)
(#PCDATA,name,size)*
```

ou peut-être :

```
ANY
```

Lorsque la DTD spécifie que les enfants sont `(name, size)`, l'analyseur sait que le blanc présent entre eux ne constitue pas du texte. Appelez :

```
factory.setIgnoringElementContentWhitespace(true);
```

Le constructeur cessera alors de signaler les blancs dans les nœuds de texte. Cela signifie que vous pouvez maintenant *compter sur* le fait qu'un nœud `font` possède deux enfants. Vous n'avez plus besoin de programmer de boucle compliquée :

```
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    if (child instanceof Element)
    {
        Element childElement = (Element) child;
        if (childElement.getTagName().equals("name")) . . .
        else if (childElement.getTagName().equals("size")) . . .
    }
}
```

Au lieu de cela, vous pouvez simplement accéder aux premier et deuxième enfants :

```
Element nameElement = (Element) children.item(0);  
Element sizeElement = (Element) children.item(1);
```

C'est pourquoi les DTD sont si utiles. Vous ne surchargez pas votre programme avec le code de vérification des règles ; l'analyseur a déjà effectué ce travail au moment où vous obtenez le document.

ASTUCE

De nombreux programmeurs qui commencent à utiliser le XML ne sont pas à l'aise avec les validations et finissent par analyser l'arbre DOM à la volée. Si vous devez convaincre vos collègues des avantages de l'utilisation des documents validés, montrez-leur les deux possibilités de codage.

Lorsque l'analyseur signale une erreur, votre application voudra effectuer une action à ce sujet : la consigner, la montrer à l'utilisateur ou déclencher une exception pour abandonner l'analyse. Ainsi, vous devez installer un gestionnaire d'erreurs dès que vous utilisez la validation. Fournissez un objet qui implémente l'interface `ErrorHandler`. Cette interface possède trois méthodes :

```
void warning(SAXParseException exception)  
void error(SAXParseException exception)  
void fatalError(SAXParseException exception)
```

Vous installez le gestionnaire d'erreurs avec la méthode `setErrorHandler` de la classe `DocumentBuilder` :

```
builder.setErrorHandler(handler);
```

javax.xml.parsers.DocumentBuilder 1.4

- `void setEntityResolver(EntityResolver resolver)`
Définit le système de résolution afin qu'il retrouve des entités référencées dans les documents XML à analyser.
- `void setErrorHandler(ErrorHandler handler)`
Définit le gestionnaire afin qu'il signale les erreurs et les avertissements qui surviennent au cours de l'analyse.

org.xml.sax.EntityResolver 1.4

- `public InputSource resolveEntity(String publicID, String systemID)`
Renvoie une source d'entrée qui contient les données référencées par les ID données ou `null` pour indiquer que ce "resolver" ne sait pas comment résoudre le nom particulier. Le paramètre `publicID` peut être `null` lorsque aucune ID publique n'a été fournie.

org.xml.sax.InputSource 1.4

- `InputSource(InputStream in)`
- `InputSource(Reader in)`
- `InputSource(String systemID)`

Construisent une source d'entrée à partir d'un flux, d'un lecteur ou d'une ID système (généralement une URL relative ou absolue).

760 *Au cœur de Java 2 - Fonctions avancées*

org.xml.sax.ErrorHandler 1.4

- void fatalError(SAXParseException exception)
- void error(SAXParseException exception)
- void warning(SAXParseException exception)

Surcharge ces méthodes afin de fournir des gestionnaires pour les erreurs fatales, non fatales et les avertissements.

org.xml.sax.SAXParseException 1.4

- int getLineNumber()
- int getColumnNumber()

Renvoient le numéro de ligne et de colonne de la fin de l'entrée traitée qui a causé l'exception.

javax.xml.parsers.DocumentBuilderFactory 1.4

- boolean isValidating()
- void setValidating(boolean value)

La propriété "validante" du factory. Si elle est définie sur true, les analyseurs générés par ce factory valident leur entrée.

- boolean isIgnoringElementContentWhitespace()
- void setIgnoringElementContentWhitespace(boolean value)

La propriété "ignoringElementContentWhitespace" du factory. Si elle est définie sur true, les analyseurs générés par ce factory ignorent le texte blanc entre les nœuds d'éléments qui n'ont pas de contenu mixte (par exemple un mélange d'éléments et de #PCDATA).

XML Schema

La syntaxe de XML Schema étant un peu plus complexe que celle de la DTD, nous n'en verrons que les concepts de base. Pour en savoir plus, nous vous conseillons le didacticiel à l'adresse <http://www.w3.org/TR/xmlschema-0>.

Pour référencer un fichier Schema dans un document, ajoutez des attributs à l'élément racine, par exemple :

```
<?xml version="1.0"?>
<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="config.xsd">
    . . .
</configuration>
```

Cette déclaration indique qu'il convient d'utiliser le fichier config.xsd pour valider le document. Si votre document utilise des espaces de noms, la syntaxe est un peu plus complexe (voir le didacticiel de XML Schema pour en savoir plus). Le préfixe xsi est un alias d'espace de nom.

Un schéma définit un type pour chaque élément. Le type peut être un type simple, une chaîne avec des restrictions de mise en forme ou un type complexe. Certains types simples sont intégrés dans XML Schema, notamment :

```
xsd:string
xsd:int
xsd:boolean
```

INFO

Nous utilisons le préfixe `xsd:` pour désigner l'espace de nom XSL Schema Definition. Certains auteurs préfèrent le préfixe `xs:`.

Vous pouvez définir vos propres types simples. Par exemple, voici un type énuméré :

```
<xsd:simpleType name="StyleType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="PLAIN" />
    <xsd:enumeration value="BOLD" />
    <xsd:enumeration value="ITALIC" />
    <xsd:enumeration value="BOLD_ITALIC" />
  </xsd:restriction>
</xsd:simpleType>
```

Lorsque vous définissez un élément, spécifiez son type :

```
<xsd:element name="name" type="xsd:string"/>
<xsd:element name="size" type="xsd:int"/>
<xsd:element name="style" type="StyleType"/>
```

Le type contraint le contenu de l'élément. Par exemple, les éléments

```
<size>10</size>
<style>PLAIN</style>
```

se valideront correctement, mais les éléments

```
<size>default</size>
<style>SLANTED</style>
```

seront refusés par l'analyseur.

Vous pouvez composer des types dans des types complexes, par exemple :

```
<xsd:complexType name="FontType">
  <xsd:sequence>
    <xsd:element ref="name"/>
    <xsd:element ref="size"/>
    <xsd:element ref="style"/>
  </xsd:sequence>
</xsd:complexType>
```

Un `FontType` est une suite d'éléments `name`, `size` et `style`. Dans cette définition de type, nous utilisons l'attribut `ref` et faisons référence aux définitions situées ailleurs dans le schéma. Vous pouvez aussi imbriquer des définitions, comme ceci :

```
<xsd:complexType name="FontType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
    <xsd:element name="style" type="StyleType">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="PLAIN" />
          <xsd:enumeration value="BOLD" />
          <xsd:enumeration value="ITALIC" />
          <xsd:enumeration value="BOLD_ITALIC" />
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```


762 Au cœur de Java 2 - Fonctions avancées

```

        </xsd:simpleType>
    </xsd:element>
</xsd:sequence>
</xsd:complexType>

```

Remarquez la *définition de type anonyme* de l'élément `style`.

La construction `xsd:sequence` équivaut à la notation de concaténation dans les DTD. La construction `xsd:choice` équivaut à l'opérateur `|`. Par exemple :

```

<xsd:complexType name="contactinfo">
  <xsd:choice>
    <xsd:element ref="email"/>
    <xsd:element ref="phone"/>
  </xsd:choice>
</xsd:complexType>

```

C'est l'équivalent du type DTD `email|phone`.

Pour permettre les éléments répétés, utilisez les attributs `minoccurs` et `maxoccurs`. Par exemple, l'équivalent du type DTD `item*` est :

```

<xsd:element name="item" type=". . ." minoccurs="0"
maxoccurs="unbounded">

```

Pour spécifier les attributs, ajoutez l'élément `xsd:attribute` aux définitions `complexType` :

```

<xsd:element name="size">
  <xsd:complexType>
    . . .
    <xsd:attribute name="unit" type="xsd:string" use="optional"
default="cm"/>
  </xsd:complexType>
</xsd:element>

```

C'est l'équivalent de l'instruction de DTD

```
<!ATTLIST size unit CDATA #IMPLIED "cm">
```

Vous intégrez l'élément et les définitions de type de votre schéma dans un élément `xsd:schema` :

```

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  . . .
</xsd:schema>

```

L'analyse d'un fichier XML avec un schéma revient à l'analyser avec une DTD, à trois différences près :

1. Vous devez activer la prise en charge des espaces de noms, même si vous ne les utilisez pas dans vos fichiers XML.

```
factory.setNamespaceAware(true);
```
2. Vous devez préparer le factory pour la gestion des schémas, avec l'incantation magique suivante :

```

final String JAXP_SCHEMA_LANGUAGE =
  "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
final String W3C_XML_SCHEMA = "http://www.w3.org/2001/XMLSchema";
factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);

```

3. Au moins dans le JDK 5.0, l'analyseur ne rejette pas les *espaces blancs du contenu des éléments*. Consultez le code de l'Exemple 10.4 pour obtenir un contournement.

INFO

Ceci est en fait un bogue, même si le rapport n° 4867706 le classe comme une demande d'amélioration.

Un exemple pratique

Dans cette section, nous travaillerons sur un exemple pratique qui montre l'utilisation de XML dans un cadre réaliste. Rappelez-vous ce que nous avons vu au Chapitre 9 du Volume 1 : `GridBagLayout` est le gestionnaire de mise en page le plus utile pour les composants Swing. Toutefois, il est redouté, non seulement pour sa complexité, mais également pour les lourdeurs de sa programmation. Il serait bien plus pratique de placer les instructions de présentation dans un fichier texte, au lieu de produire de grandes quantités de code répétitif. Dans cette section, vous verrez comment utiliser le code XML pour décrire une position de grille et comment analyser les fichiers de présentation.

Une grille est constituée de lignes et de colonnes, très semblables à un tableau HTML. A l'instar d'un tableau HTML, nous le décrirons comme une suite de lignes, chacune contenant des cellules :

```
<gridbag>
  <row>
    <cell>...</cell>
    <cell>...</cell>
    .
    .
    .
  </row>
  <row>
    <cell>...</cell>
    <cell>...</cell>
    .
    .
    .
  </row>
  .
  .
  .
</gridbag>
```

Le `gridbag.dtd` spécifie ces règles :

```
<!ELEMENT gridbag (row)*>
<!ELEMENT row (cell)*>
```

Certaines cellules peuvent s'étendre sur plusieurs lignes et colonnes. Dans la mise en page, cela s'obtient en définissant les contraintes `gridwidth` et `gridheight` sur des valeurs supérieures à 1. Nous utiliserons des attributs du même nom :

```
<cell gridwidth="2" gridheight="2">
```

De même, nous utiliserons des attributs pour les autres contraintes de grille `fill`, `anchor`, `gridx`, `gridy`, `weightx`, `weighty`, `ipadx` et `ipady` (nous ne gérons pas les contraintes `insets` car leur valeur n'est pas un type simple, mais il serait plus simple de les prendre en charge). Par exemple :

```
<cell fill="HORIZONTAL" anchor="NORTH">
```

Pour la plupart de ces attributs, nous fournissons les mêmes paramètres par défaut que le constructeur par défaut `GridBagConstraints` :

```
<!ATTLIST cell gridwidth CDATA "1">
<!ATTLIST cell gridheight CDATA "1">
```

764 *Au cœur de Java 2 - Fonctions avancées*

```
<!ATTLIST cell fill (NONE|BOTH|HORIZONTAL|VERTICAL) "NONE">
<!ATTLIST cell anchor (CENTER|NORTH|NORTHEAST|EAST
|SOUTHEAST|SOUTH|SOUTHWEST|WEST|NORTHWEST) "CENTER">
```

Les valeurs `gridx` et `gridy` obtiennent un traitement spécial car il serait laborieux et quelque peu sujet à erreur de les spécifier à la main. Les fournir est optionnel :

```
<!ATTLIST cell gridx CDATA #IMPLIED>
<!ATTLIST cell gridy CDATA #IMPLIED>
```

S'ils ne sont pas fournis, le programme les détermine en fonction de la règle suivante : en colonne 0, le `gridx` par défaut vaut 0. Autrement, il utilise le `gridx` précédent plus le `gridwidth` précédent. Le `gridy` par défaut équivaut toujours au nombre de lignes. Ainsi, il sera inutile de spécifier `gridx` et `gridy` dans la plupart des cas, lorsqu'un composant s'étend sur plusieurs lignes. Mais, si un composant s'étend sur plusieurs colonnes que vous ne connaissez pas, vous devez spécifier `gridx`.

INFO

Les experts des grilles peuvent se demander pourquoi nous n'utilisons pas ici les mécanismes `RELATIVE` et `REMAIN- DER` pour permettre aux dispositions de grille de déterminer automatiquement les positions `gridx` et `gridy`. Nous avons essayé, mais rien ne peut produire la disposition de la boîte de dialogue de police de l'exemple du Volume 1. A la lecture du code source de `GridBagLayout`, il apparaît que l'algorithme n'effectuera pas les opérations lourdes qui seraient nécessaires pour récupérer les positions absolues.

Le programme analyse les attributs et définit les contraintes de grille. Pour lire, par exemple, la largeur de grille, le programme contient une instruction unique :

```
constraints.gridwidth
= Integer.parseInt(e.getAttribute("gridwidth"));
```

Le programme ne s'inquiète pas d'un attribut manquant, puisque l'analyseur fournit automatiquement la valeur par défaut si aucune autre valeur n'a été spécifiée dans le document.

Pour tester si un attribut `gridx` ou `gridy` a été spécifié, nous appelons la méthode `getAttribute` et vérifions si elle renvoie la chaîne vide :

```
String value = e.getAttribute("gridy");
if (value.length() == 0) // utiliser le paramètre par défaut
    constraints.gridy = r;
else
    constraints.gridx = Integer.parseInt(value);
```

Une cellule peut contenir n'importe quel composant, mais nous avons trouvé commode de permettre à l'utilisateur de spécifier des beans plus généraux. Cela nous permet de spécifier des types non-composants comme des bordures. Un bean se définit par un nom de classe et zéro ou plusieurs propriétés :

```
<!ELEMENT bean (class, property*)>
<!ELEMENT class (#PCDATA)>
```

Comme toujours avec les beans, nous supposons que la classe de bean dispose d'un constructeur par défaut.

Une propriété contient un nom et une valeur :

```
<!ELEMENT property (name, value)>
<!ELEMENT name (#PCDATA)>
```

La valeur est un entier, un booléen, une chaîne ou un autre bean :

```
<!ELEMENT value (int|string|boolean|bean)>
<!ELEMENT int (#PCDATA)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT boolean (#PCDATA)>
```

Voici un exemple habituel, un JLabel dont la propriété text est définie sur la chaîne "œil de caractère : " :

```
<bean>
  <class>javax.swing.JLabel</class>
  <property>
    <name>text</name>
    <value><string>œil de caractère : </string></value>
  </property>
</bean>
```

Il semble ennuyeux d'entourer une chaîne avec la balise <string>. Pourquoi ne pas simplement utiliser #PCDATA pour les chaînes et laisser les balises pour les autres types ? Nous aurions besoin d'utiliser un contenu mixte et d'affaiblir la règle pour l'élément value sur

```
<!ELEMENT value (#PCDATA|int|boolean|bean)*>
```

Toutefois, cette règle permettrait un mélange arbitraire de texte et de balises.

Le programme définit une propriété en utilisant la classe BeanInfo. BeanInfo énumère les descripteurs de propriétés du bean. Nous recherchons la propriété contenant le nom correspondant, puis nous appelons sa méthode setter avec la valeur fournie.

Lorsque notre programme lit une description d'interface utilisateur, il possède suffisamment d'informations pour construire et arranger les composants de l'interface utilisateur. Toutefois, bien entendu, l'interface n'est pas vivante, aucun écouteur d'événements n'a été attaché. Pour ajouter des écouteurs d'événements, il est nécessaire de localiser les composants. Pour cette raison, nous supportons un attribut optionnel du type ID pour chaque bean :

```
<!ATTLIST bean id ID #IMPLIED>
```

Par exemple, voici une zone de liste modifiable avec un ID :

```
<bean id="face">
  <class>javax.swing.JComboBox</class>
</bean>
```

Rappelez-vous que l'analyseur vérifie que les ID sont uniques.

Un programmeur peut joindre des gestionnaires d'événements comme :

```
gridbag = new GridBagPane("fontdialog.xml");
setContentPane(gridbag);
JComboBox face = (JComboBox) gridbag.get("œil de caractère");
face.addListener(listener);
```

766 Au cœur de Java 2 - Fonctions avancées

INFO

Dans cet exemple, nous n'utilisons XML que pour décrire la mise en page des composants et laisser aux programmeurs le soin d'attacher les gestionnaires d'événements dans le code Java. Vous pourriez toutefois ajouter le code à la description XML. L'approche la plus prometteuse consiste à utiliser un langage de script comme JavaScript pour le code. Si vous souhaitez ajouter cette amélioration, consultez l'interpréteur Rhino à l'adresse <http://www.mozilla.org/rhino>.

Le programme de l'Exemple 10.2 montre comment utiliser la classe `GridBagPane` pour effectuer le travail ennuyeux qui consiste à définir la disposition de grille. La disposition est définie dans l'Exemple 10.3. La Figure 10.4 montre le résultat. Le programme n'initialise que les zones de liste modifiable (qui sont trop complexes pour le mécanisme de définition de propriété de bean que le `GridBagPane` prend en charge) et attache les écouteurs d'événements. La classe `GridBagPane` de l'Exemple 10.4 analyse le fichier XML, construit les composants et les dispose. L'Exemple 10.5 affiche la DTD.

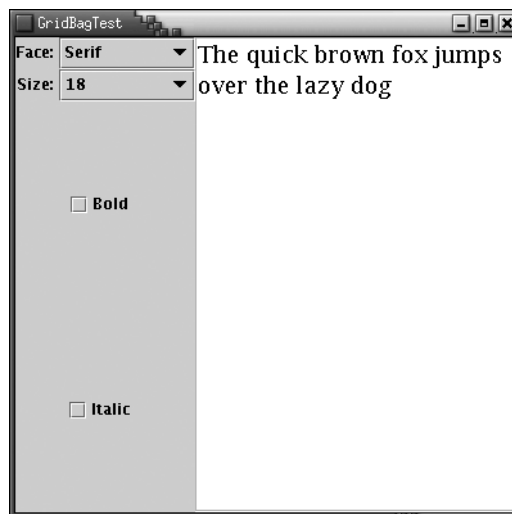
Le programme peut aussi traiter un schéma au lieu d'une DTD si vous le lancez avec

```
java GridBagTest fontdialog-schema.xml
```

L'Exemple 10.6 contient le schéma.

Figure 10.4

Une boîte de dialogue de police définie par une disposition XML.



Cet exemple montre une utilisation habituelle du XML. Le format XML est assez fiable pour exprimer des relations complexes. L'analyseur XML ajoute de la valeur au programme en reprenant le travail de routine qui consiste à vérifier la validité et à fournir les paramètres par défaut.

Exemple 10.2 : `GridBagTest.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
```

```

    Ce programme montre comment utiliser un fichier XML pour décrire
    une disposition de grille.
*/
public class GridBagTest
{
    public static void main(String[] args)
    {
        String filename = args.length == 0 ? "fontdialog.xml" : args[0];
        JFrame frame = new JFrame(filename);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
    Ce bloc contient une boîte de dialogue pour la sélection des polices
    décrites par un fichier XML.
    @param filename Le fichier contenant les composants de l'interface
    utilisateur pour la boîte de dialogue
*/
class FontFrame extends JFrame
{
    public FontFrame(String filename)
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
        setTitle("GridBagTest");

        gridbag = new GridBagPane(filename);
        add(gridbag);

        face = (JComboBox) gridbag.get("Œil de caractère");
        size = (JComboBox) gridbag.get("Taille");
        bold = (JCheckBox) gridbag.get("Gras");
        italic = (JCheckBox) gridbag.get("Italique");

        face.setModel(new DefaultComboBoxModel(new Object[]
        {
            "Serif", "SansSerif", "Monospaced",
            "Dialog", "DialogInput"
        }));

        size.setModel(new DefaultComboBoxModel(new Object[]
        {
            "8", "10", "12", "15", "18", "24", "36", "48"
        }));

        ActionListener listener = new
            ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            { setSample(); }
        };

        face.addActionListener(listener);
        size.addActionListener(listener);
        bold.addActionListener(listener);
        italic.addActionListener(listener);
    }
}

```

768 Au cœur de Java 2 - Fonctions avancées

```

        setSample();
    }

    /**
     * Cette méthode applique la police sélectionnée à l'échantillon de
     * texte.
     */
    public void setSample()
    {
        String fontFace = (String) face.getSelectedItem();
        int fontSize = Integer.parseInt(
            (String) size.getSelectedItem());
        JTextArea sample = (JTextArea) gridbag.get("sample");
        int fontStyle
            = (bold.isSelected() ? Font.BOLD : 0)
            + (italic.isSelected() ? Font.ITALIC : 0);

        sample.setFont(new Font(fontFace, fontStyle, fontSize));
        sample.repaint();
    }

    private GridBagPane gridbag;
    private JComboBox face;
    private JComboBox size;
    private JCheckBox bold;
    private JCheckBox italic;
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 400;
}

```

Exemple 10.3 : fontdialog.xml

```

<?xml version="1.0"?>
<!DOCTYPE gridbag SYSTEM "gridbag.dtd">
<gridbag>
  <row>
    <cell anchor="EAST">
      <bean>
        <class>javax.swing.JLabel</class>
        <property>
          <name>text</name>
          <value><string>Face: </string></value>
        </property>
      </bean>
    </cell>
    <cell fill="HORIZONTAL" weightx="100">
      <bean id="face">
        <class>javax.swing.JComboBox</class>
      </bean>
    </cell>
    <cell gridheight="4" fill="BOTH" weightx="100" weighty="100">
      <bean id="sample">
        <class>javax.swing.JTextArea</class>
        <property>
          <name>text</name>
          <value><string>The quick brown fox jumps over the lazy
dog</string></value>
        </property>
        <property>

```

```

        <name>editable</name>
        <value><boolean>>false</boolean></value>
    </property>
    <property>
        <name>lineWrap</name>
        <value><boolean>>true</boolean></value>
    </property>
    <property>
        <name>border</name>
        <value>
            <bean>
                <class>javax.swing.border.EtchedBorder</class>
            </bean>
        </value>
    </property>
</bean>
</cell>
</row>
<row>
    <cell anchor="EAST">
        <bean>
            <class>javax.swing.JLabel</class>
            <property>
                <name>text</name>
                <value><string>Size: </string></value>
            </property>
        </bean>
    </cell>
    <cell fill="HORIZONTAL" weightx="100">
        <bean id="size">
            <class>javax.swing.JComboBox</class>
        </bean>
    </cell>
</row>
<row>
    <cell gridwidth="2" weighty="100">
        <bean id="bold">
            <class>javax.swing.JCheckBox</class>
            <property>
                <name>text</name>
                <value><string>Bold</string></value>
            </property>
        </bean>
    </cell>
</row>
<row>
    <cell gridwidth="2" weighty="100">
        <bean id="italic">
            <class>javax.swing.JCheckBox</class>
            <property>
                <name>text</name>
                <value><string>Italic</string></value>
            </property>
        </bean>
    </cell>
</row>
</gridbag>

```


770 *Au cœur de Java 2 - Fonctions avancées*

Exemple 10.4 : GridBagPane.java

```
import java.awt.*;
import java.beans.*;
import java.io.*;
import java.lang.reflect.*;
import javax.swing.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;

import org.w3c.dom.ls.*;
import org.w3c.dom.bootstrap.*;

/**
 * Ce panneau utilise un fichier XML pour décrire ses composants et
 * leurs positions dans la disposition de grille.
 */
public class GridBagPane extends JPanel
{
    /**
     * Construit un volet de grille.
     * @param filename le nom du fichier XML qui décrit les
     * composants du volet et leurs positions
     */
    public GridBagPane(String filename)
    {
        setLayout(new GridBagLayout());
        constraints = new GridBagConstraints();

        try
        {
            DocumentBuilderFactory factory
                = DocumentBuilderFactory.newInstance();
            factory.setValidating(true);

            if (filename.contains("-schema"))
            {
                factory.setNamespaceAware(true);
                final String JAXP_SCHEMA_LANGUAGE =
                    "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
                final String W3C_XML_SCHEMA =
                    "http://www.w3.org/2001/XMLSchema";
                factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
            }

            factory.setIgnoringElementContentWhitespace(true);

            DocumentBuilder builder = factory.newDocumentBuilder();
            Document doc = builder.parse(new File(filename));

            if (filename.contains("-schema"))
                // contournement au bogue #4867706
            {
                int count = removeWhitespace(doc.getDocumentElement());
                System.out.println(count + " whitespace nodes removed.");
            }

            parseGridbag(doc.getDocumentElement());
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }

    private int removeWhitespace(Node node)
    {
        int count = 0;
        if (node instanceof Element)
        {
            Element elem = (Element) node;
            NodeList children = elem.getChildNodes();
            for (int i = 0; i < children.getLength(); i++)
            {
                Node child = children.item(i);
                count += removeWhitespace(child);
            }
        }
        return count;
    }
}
```

```

    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}

/**
 * Supprime l'espace blanc du contenu d'élément
 * @param e L'élément racine
 * @return Le nombre de noeuds d'espace blanc supprimés.
 */
private int removeWhitespace(Element e)
{
    NodeList children = e.getChildNodes();
    if (children.getLength() <= 1) return 0;
    int count = 0;
    for (int i = children.getLength() - 1; i >= 0; i--)
    {
        Node child = children.item(i);
        if (child instanceof Text && (
            (Text) child).getData().trim().length() == 0)
        {
            e.removeChild(child);
            count++;
        }
        else if (child instanceof Element)
            count += removeWhitespace((Element) child);
    }
    return count;
}

/**
 * Extrait un composant avec un nom donné.
 * @param name un nom de composant
 * @return le composant avec le nom donné ou null lorsqu'aucun
 * composant dans ce volet de grille ne possède le nom donné
 */
public Component get(String name)
{
    Component[] components = getComponents();
    for (int i = 0; i < components.length; i++)
    {
        if (components[i].getName().equals(name))
            return components[i];
    }
    return null;
}

/**
 * Analyse un élément gridbag.
 * @param e un élément gridbag
 */
private void parseGridbag(Element e)
{
    NodeList rows = e.getChildNodes();
    for (int i = 0; i < rows.getLength(); i++)
    {
        Element row = (Element) rows.item(i);

```

772 Au cœur de Java 2 - Fonctions avancées

```

        NodeList cells = row.getChildNodes();
        for (int j = 0; j < cells.getLength(); j++)
        {
            Element cell = (Element) cells.item(j);
            parseCell(cell, i, j);
        }
    }
}

/**
 * Analyse un élément de cellule.
 * @param e un élément de cellule
 * @param r la ligne de la cellule
 * @param c la colonne de la cellule
 */
private void parseCell(Element e, int r, int c)
{
    // obtenir les attributs

    String value = e.getAttribute("gridx");
    if (value.length() == 0) // utiliser les paramètres par défaut
    {
        if (c == 0) constraints.gridx = 0;
        else constraints.gridx += constraints.gridwidth;
    }
    else
        constraints.gridx = Integer.parseInt(value);

    value = e.getAttribute("gridy");
    if (value.length() == 0) // utiliser les paramètres par défaut
        constraints.gridy = r;
    else
        constraints.gridy = Integer.parseInt(value);

    constraints.gridwidth
        = Integer.parseInt(e.getAttribute("gridwidth"));
    constraints.gridheight
        = Integer.parseInt(e.getAttribute("gridheight"));
    constraints.weightx
        = Integer.parseInt(e.getAttribute("weightx"));
    constraints.weighty
        = Integer.parseInt(e.getAttribute("weighty"));
    constraints.ipadx
        = Integer.parseInt(e.getAttribute("ipadx"));
    constraints.ipady
        = Integer.parseInt(e.getAttribute("ipady"));

    // utiliser une réflexion pour obtenir des valeurs d'entier de
    // champs statiques
    Class cl = GridBagConstraints.class;

    try
    {
        String name = e.getAttribute("fill");
        Field f = cl.getField(name);
        constraints.fill = f.getInt(cl);

        name = e.getAttribute("anchor");
        f = cl.getField(name);
    }
}

```

```

        constraints.anchor = f.getInt(c1);
    }
    catch (Exception ex) // les méthodes de réflexion
                        // peuvent lancer diverses exceptions
    {
        ex.printStackTrace();
    }

    Component comp = (Component) parseBean(
        (Element) e.getFirstChild());
    add(comp, constraints);
}

/**
 * Analyse un élément bean.
 * @param e un élément bean
 */
private Object parseBean(Element e)
{
    try
    {
        NodeList children = e.getChildNodes();
        Element classElement = (Element) children.item(0);
        String className
            = ((Text) classElement.getFirstChild()).getData();

        Class c1 = Class.forName(className);

        Object obj = c1.newInstance();

        if (obj instanceof Component)
            ((Component) obj).setName(e.getAttribute("id"));

        for (int i = 1; i < children.getLength(); i++)
        {
            Node propertyElement = children.item(i);
            Element nameElement
                = (Element) propertyElement.getFirstChild();
            String propertyName
                = ((Text) nameElement.getFirstChild()).getData();

            Element valueElement
                = (Element) propertyElement.getLastChild();
            Object value = parseValue(valueElement);
            BeanInfo beanInfo = Introspector.getBeanInfo(c1);
            PropertyDescriptor[] descriptors
                = beanInfo.getPropertyDescriptors();
            boolean done = false;
            for (int j = 0; !done && j < descriptors.length; j++)
            {
                if (descriptors[j].getName().equals(propertyName))
                {
                    descriptors[j].getWriteMethod().invoke(obj, value);
                    done = true;
                }
            }
        }
    }
    return obj;
}

```

774 Au cœur de Java 2 - Fonctions avancées

```

    }
    catch (Exception ex)
        // les méthodes de réflexion peuvent déclencher diverses
        // exceptions
    {
        ex.printStackTrace();
        return null;
    }
}

/**
 * Analyse un élément de valeur.
 * @param e un élément de valeur
 */
private Object parseValue(Element e)
{
    Element child = (Element) e.getFirstChild();
    if (child.getTagName().equals("bean"))
        return parseBean(child);
    String text = ((Text) child.getFirstChild()).getData();
    if (child.getTagName().equals("int"))
        return new Integer(text);
    else if (child.getTagName().equals("boolean"))
        return new Boolean(text);
    else if (child.getTagName().equals("string"))
        return text;
    else
        return null;
}

private GridBagConstraints constraints;
}

```

Exemple 10.5 : gridbag.dtd

```

<!ELEMENT gridbag (row)*>
<!ELEMENT row (cell)*>
<!ELEMENT cell (bean)>
<!ATTLIST cell gridx CDATA #IMPLIED>
<!ATTLIST cell gridy CDATA #IMPLIED>
<!ATTLIST cell gridwidth CDATA "1">
<!ATTLIST cell gridheight CDATA "1">
<!ATTLIST cell weightx CDATA "0">
<!ATTLIST cell weighty CDATA "0">
<!ATTLIST cell fill (NONE|BOTH|HORIZONTAL|VERTICAL) "NONE">
<!ATTLIST cell anchor (CENTER|NORTH|NORTHEAST|EAST|SOUTH
    EAST|SOUTH|SOUTHWEST|WEST|NORTHWEST) "CENTER">
<!ATTLIST cell ipadx CDATA "0">
<!ATTLIST cell ipady CDATA "0">

<!ELEMENT bean (class, property)*>
<!ATTLIST bean id ID #IMPLIED>
<!ELEMENT class (#PCDATA)>
<!ELEMENT property (name, value)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT value (int|string|boolean|bean)>
<!ELEMENT int (#PCDATA)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT boolean (#PCDATA)>

```

Exemple 10.6 : gridbag.xsd

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="gridbag" type="GridBagType"/>

  <xsd:element name="bean" type="BeanType"/>

  <xsd:complexType name="GridBagType">
    <xsd:sequence>
      <xsd:element name="row" type="RowType" minOccurs="0"
        maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="RowType">
    <xsd:sequence>
      <xsd:element name="cell" type="CellType" minOccurs="0"
        maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="CellType">
    <xsd:sequence>
      <xsd:element ref="bean"/>
    </xsd:sequence>
    <xsd:attribute name="gridx" type="xsd:int" use="optional"/>
    <xsd:attribute name="gridy" type="xsd:int" use="optional"/>
    <xsd:attribute name="gridwidth" type="xsd:int"
      use="optional" default="1" />
    <xsd:attribute name="gridheight" type="xsd:int"
      use="optional" default="1" />
    <xsd:attribute name="weightx" type="xsd:int"
      use="optional" default="0" />
    <xsd:attribute name="weighty" type="xsd:int"
      use="optional" default="0" />
    <xsd:attribute name="fill" use="optional" default="NONE">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="NONE" />
          <xsd:enumeration value="BOTH" />
          <xsd:enumeration value="HORIZONTAL" />
          <xsd:enumeration value="VERTICAL" />
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="anchor" use="optional" default="CENTER">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="CENTER" />
          <xsd:enumeration value="NORTH" />
          <xsd:enumeration value="NORTHEAST" />
          <xsd:enumeration value="EAST" />
          <xsd:enumeration value="SOUTHEAST" />
          <xsd:enumeration value="SOUTH" />
          <xsd:enumeration value="SOUTHWEST" />
          <xsd:enumeration value="WEST" />
          <xsd:enumeration value="NORTHWEST" />
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
  </xsd:complexType>

```

776 Au cœur de Java 2 - Fonctions avancées

```

        </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="ipady" type="xsd:int"
        use="optional" default="0" />
    <xsd:attribute name="ipadx" type="xsd:int"
        use="optional" default="0" />
</xsd:complexType>

<xsd:complexType name="BeanType">
    <xsd:sequence>
        <xsd:element name="class" type="xsd:string"/>
        <xsd:element name="property" type="PropertyType"
            minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID" use="optional" />
</xsd:complexType>

<xsd:complexType name="PropertyType">
    <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="value" type="ValueType"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ValueType">
    <xsd:choice>
        <xsd:element ref="bean"/>
        <xsd:element name="int" type="xsd:int"/>
        <xsd:element name="string" type="xsd:string"/>
        <xsd:element name="boolean" type="xsd:boolean"/>
    </xsd:choice>
</xsd:complexType>
</xsd:schema>

```

Localiser des informations avec XPath

Localiser des informations spécifiques dans un document XML peut se révéler un peu difficile du fait de la navigation dans les nœuds de l'arbre DOM. Le langage XPath simplifie l'accès aux nœuds de l'arbre. Supposons, par exemple, que vous possédiez ce document XML :

```

<configuration>
    .
    .
    <database>
        <username>dbuser</username>
        <password>secret</password>
    .
    .
    </database>
</configuration>

```

Vous pouvez obtenir le nom de l'utilisateur de la base de données en évaluant l'expression XPath

```
/configuration/database/username
```

Ceci est bien plus simple que l'approche DOM :

1. Récupérez le nœud du document.
2. Enumérez ses enfants.

3. Localisez l'élément database.
4. Retrouvez son premier enfant, l'élément username.
5. Récupérez son premier enfant, un nœud Text.
6. Récupérez ses données.

Un XPath peut décrire un jeu de nœuds dans un document XML. Par exemple, le XPath

```
/gridbag/row
```

décrit le jeu de tous les éléments row qui sont des enfants de l'élément racine gridbag. Vous pouvez choisir un élément particulier avec l'opérateur [] :

```
/gridbag/row[1]
```

est le premier row (les valeurs d'indice commencent à 1).

Utilisez l'opérateur @ pour récupérer les valeurs d'attribut. L'expression XPath

```
/gridbag/row[1]/cell[1]/@anchor
```

décrit l'attribut anchor de la première cellule de la première ligne. L'expression XPath

```
/gridbag/row/cell/@anchor
```

décrit tous les nœuds d'attribut anchor des éléments cell contenus dans les éléments row qui sont des enfants du nœud racine gridbag.

Il existe de nombreuses fonctions XPath utiles. Par exemple,

```
count(/gridbag/row)
```

renvoie le nombre d'enfants row de la racine gridbag. Il existe de nombreuses expressions XPath plus élaborées (pour cela, consultez l'adresse <http://www.w3c.org/TR/xpath> ou le didacticiel en ligne à l'adresse <http://www.zvon.org/xxl/XPathTutorial/General/examples.html>).

Le JDK 5.0 contient une nouvelle API pour évaluer les expressions XPath. Vous créez d'abord un objet XPath à partir d'un XPathFactory :

```
XPathFactory xpfactory = XPathFactory.newInstance();
path = xpfactory.newXPath();
```

Vous appelez ensuite la méthode evaluate pour évaluer les expressions XPath :

```
String username = path.evaluate("/configuration/database/username", doc);
```

Vous pouvez utiliser le même objet XPath pour évaluer plusieurs expressions.

Cette forme de la méthode evaluate renvoie un résultat de chaîne. Il convient pour récupérer du texte, comme le texte du nœud username de l'exemple précédent. Si une expression XPath produit un jeu de nœuds, effectuez un appel comme celui-ci :

```
NodeList nodes = (NodeList) path.evaluate("/gridbag/row", doc,
XPathConstants.NODESET);
```

Si le résultat est un nœud unique, utilisez plutôt XPathConstants.NODE :

```
Node node = (Node) path.evaluate("/gridbag/row[1]", doc, XPathConstants.NODE);
```

Si le résultat est un nombre, utilisez XPathConstants.NUMBER :

```
int count = ((Number) path.evaluate("count(/gridbag/row)", doc,
XPathConstants.NUMBER)).intValue();
```


778 Au cœur de Java 2 - Fonctions avancées

Vous n'avez pas à commencer la recherche à la racine du document. Vous pouvez commencer à n'importe quel nœud ou liste de nœuds. Par exemple, si vous possédez un nœud d'une précédente évaluation, vous pouvez appeler

```
result = path.evaluate(expression, node);
```

Le programme de l'Exemple 10.7 montre l'évaluation des expressions XPath. Chargez un fichier XML et tapez une expression. Choisissez le type d'expression et cliquez sur le bouton "Evaluer". Le résultat est affiché au bas du cadre (voir Figure 10.5).

Figure 10.5

Evaluation des expressions XPath.0.



Exemple 10.7 : XPathTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.xml.namespace.*;
import javax.xml.parsers.*;
import javax.xml.xpath.*;
import org.w3c.dom.*;
import org.xml.sax.*;

/**
 * Ce programme évalue les expressions XPath.
 */
public class XPathTest
{
    public static void main(String[] args)
    {
        JFrame frame = new XPathFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Ce cadre présente un document XML, un panneau pour taper une
 * expression XPath et un champ de texte pour afficher le résultat.
 */
class XPathFrame extends JFrame
```

```

{
    public XPathFrame()
    {
        setTitle("XPathTest");

        JMenu fileMenu = new JMenu("Fichier");
        JMenuItem openItem = new JMenuItem("Ouvrir");
        openItem.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                { openFile(); }
            });
        fileMenu.add(openItem);

        JMenuItem exitItem = new JMenuItem("Quitter");
        exitItem.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                { System.exit(0); }
            });
        fileMenu.add(exitItem);

        JMenuBar menuBar = new JMenuBar();
        menuBar.add(fileMenu);
        setJMenuBar(menuBar);

        ActionListener listener = new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                { evaluate(); }
            };
        expression = new JTextField(20);
        expression.addActionListener(listener);
        JButton evaluateButton = new JButton("Evaluer");
        evaluateButton.addActionListener(listener);

        typeCombo = new JComboBox(new Object[]
            { "STRING", "NODE", "NODESET", "NUMBER", "BOOLEAN" });
        typeCombo.setSelectedItem("STRING");

        JPanel panel = new JPanel();
        panel.add(expression);
        panel.add(typeCombo);
        panel.add(evaluateButton);
        docText = new JTextArea(10, 40);
        result = new JTextField();
        result.setBorder(new TitledBorder("Résultat"));

        add(panel, BorderLayout.NORTH);
        add(new JScrollPane(docText), BorderLayout.CENTER);
        add(result, BorderLayout.SOUTH);

        try
        {
            DocumentBuilderFactory factory =
                DocumentBuilderFactory.newInstance();

```

780 *Au cœur de Java 2 - Fonctions avancées*

```

        builder = factory.newDocumentBuilder();
    }
    catch (ParserConfigurationException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }

    XPathFactory xpfactory = XPathFactory.newInstance();
    path = xpfactory.newXPath();
    pack();
}

/**
 * Ouvre un fichier et charge le document.
 */
public void openFile()
{
    JFileChooser chooser = new JFileChooser();
    chooser.setCurrentDirectory(new File("."));

    chooser.setFileFilter(new
        javax.swing.filechooser.FileFilter()
        {
            public boolean accept(File f)
            {
                return f.isDirectory() ||
                    f.getName().toLowerCase().endsWith(".xml");
            }
            public String getDescription() { return "Fichiers XML"; }
        });
    int r = chooser.showOpenDialog(this);
    if (r != JFileChooser.APPROVE_OPTION) return;
    File f = chooser.getSelectedFile();
    try
    {
        byte[] bytes = new byte[(int) f.length()];
        new FileInputStream(f).read(bytes);
        docText.setText(new String(bytes));
        doc = builder.parse(f);
    }
    catch (IOException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }
    catch (SAXException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }
}

public void evaluate()
{
    try
    {
        String typeName = (String) typeCombo.getSelectedItem();
        QName returnType = (QName)
            XPathConstants.class.getField(typeName).get(null);
        Object evalResult = path.evaluate(expression.getText(),

```

```

        doc, returnType);
    if (typeName.equals("NODESET"))
    {
        NodeList list = (NodeList) evalResult;
        StringBuilder builder = new StringBuilder();
        builder.append("{");
        for (int i = 0; i < list.getLength(); i++)
        {
            if (i > 0) builder.append(", ");
            builder.append("'" + list.item(i));
        }
        builder.append("}");
        result.setText("'" + builder);
    }
    else
        result.setText("'" + evalResult);
}
catch (XPathExpressionException e)
{
    result.setText("'" + e);
}
catch (Exception e) // exception de réflexion
{
    e.printStackTrace();
}
}

private DocumentBuilder builder;
private Document doc;
private XPath path;
private JTextField expression;
private JTextField result;
private JTextArea docText;
private JComboBox typeCombo;
}

```

javax.xml.xpath.XPathFactory 5.0

- `static XPathFactory newInstance()`
Renvoie une instance `XPathFactory` pour créer des objets `XPath`.
- `XPath newPath()`
Construit un objet `XPath` pour évaluer les expressions `XPath`.

javax.xml.xpath.XPath 5.0

- `String evaluate(String expression, Object startingPoint)`
Evalue une expression, en commençant par le point de départ donné. Le point de départ peut être un nœud ou un jeu de nœuds. La chaîne renvoyée est alors constituée des données de tous les enfants du nœud de texte.
- `Object evaluate(String expression, Object startingPoint, QName resultType)`
Evalue une expression, en commençant par le point de départ donné. Le point de départ peut être un nœud ou une liste de nœuds. `resultType` est l'une des constantes `STRING`, `NODE`, `NODESET`, `NUMBER` ou `BOOLEAN` dans la classe `XPathConstants`. La valeur renvoyée est `String`, `Node`, `NodeList`, `Number` ou `Boolean`.

Espaces de noms

Java utilise des packages pour éviter les séparations des noms. Les programmeurs peuvent utiliser le même nom pour des classes différentes tant qu'ils ne se trouvent pas dans le même package. XML possède un mécanisme *d'espace de noms* identique pour les noms d'éléments et d'attributs.

Un espace de noms s'identifie par une URI (*Uniform Resource Identifier*), comme :

```
http://www.w3.org/2001/XMLSchema
uuid:1c759aed-b748-475c-ab68-10679700c4f2
urn:com:books-r-us
```

L'URL HTTP est la plus commune. Vous remarquerez que l'URL est simplement utilisée comme chaîne d'identification, et non pour localiser un document. Par exemple, les identifiants d'espaces de noms

```
http://www.horstmann.com/corejava
http://www.horstmann.com/corejava/index.html
```

dénotent des espaces de noms *différents*, même si un serveur Web afficherait le même document pour les deux URL.

Il ne doit pas y avoir nécessairement d'URL d'espace de noms ; l'analyseur XML ne tente pas de trouver quelque chose à cet emplacement. Toutefois, pour aider les programmeurs qui rencontrent un espace de noms potentiellement peu familier, l'habitude consiste à placer un document expliquant l'objectif de l'espace de noms à l'emplacement de l'URL. Par exemple, si vous dirigez votre navigateur vers l'URL de l'espace de noms du XML Schema (<http://www.w3.org/2001/XMLSchema>), vous trouverez un document décrivant le standard XML Schema.

Pourquoi utiliser des URL HTTP pour les identifiants d'espaces de noms ? Il est en effet plus simple de s'assurer qu'ils sont uniques. Si vous choisissez une URL réelle, le caractère unique de la partie hôte sera alors garanti par le système de noms de domaine. Votre entreprise peut alors s'arranger pour obtenir que le reste de l'URL soit unique. C'est ce même raisonnement qui est à l'origine de l'utilisation de noms de domaine inversés dans les noms de package Java.

Bien entendu, même si l'on souhaite obtenir des identifiants d'espaces de noms uniques, il est inutile d'obtenir plus d'identifiants que ce dont vous avez besoin. En Java, vous utilisez le mécanisme `import` pour spécifier des noms de package longs, puis vous utilisez simplement des noms de classe courts. En XML, il existe un mécanisme similaire :

```
<element xmlns="URI espace de noms">
  enfants
</element>
```

Les éléments et leurs enfants font maintenant partie de l'espace de noms donné.

Un enfant peut fournir son propre espace de noms, par exemple :

```
<element xmlns="URI1 espace de noms">
  <child xmlns="URI2 espace de noms">
    petits-enfants
  </child>
  autres enfants
</element>
```

Les enfants et les petits-enfants font maintenant partie du deuxième espace de noms.

Ce mécanisme simple fonctionne si vous n'avez besoin que d'un seul espace de noms ou si les espaces de noms sont naturellement imbriqués. Autrement, vous devrez utiliser un deuxième mécanisme qui ne possède pas d'homologue en Java. Vous pouvez alors disposer d'un *alias*, c'est-à-dire d'un identifiant court que vous choisirez pour un document particulier. En voici un exemple typique :

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="gridbag" type="GridBagType"/>
  . . .
</xsd:schema>
```

L'attribut

```
xmlns:alias="URI espace de noms"
```

définit un espace de noms et un alias. Dans notre exemple, l'alias correspond à la chaîne `xsd`. Ainsi, `xsd:schema` signifie réellement "schema dans l'espace de noms `http://www.w3.org/2001/XMLSchema`".

INFO

Seuls les éléments enfant héritent de l'espace de noms de leur parent. Les attributs qui ne contiennent pas de préfixe alias explicite ne font jamais partie d'un espace de nom. Étudiez cet exemple :

```
<configuration xmlns=http://www.horstmann.com/corejava
  xmlns:si=http://www.bipm.fr/enus/3_SI/si.html>
  <size value="210" si:unit="mm"/>
  . . .
</configuration>
```

Les éléments `configuration` et `size` font partie de l'espace de noms avec l'URI `http://www.horstmann.com/corejava`. L'attribut `si:unit` fait partie de l'espace de noms avec l'URI `http://www.bipm.fr/enus/3_SI/si.html`. Toutefois l'attribut `value` ne fait partie d'aucun espace de noms.

Vous pouvez tout à fait contrôler la manière dont l'analyseur traite les espaces de noms. Par défaut, l'analyseur DOM de Sun ne reconnaît pas les espaces de noms.

Pour activer la gestion des espaces de noms, appelez la méthode `setNamespaceAware` de `DocumentBuilderFactory` :

```
factory.setNamespaceAware(true);
```

Dès lors, tous les constructeurs produits par le `factory` prennent en charge les espaces de noms. Chaque nœud dispose de trois propriétés :

- le nom qualifié, avec un préfixe alias, renvoyé par `getNodeName`, `getTagName`, etc. ;
- le nom local, sans préfixe ni espace de noms, renvoyé par la méthode `getLocalName` ;
- l'URI de l'espace de nom, renvoyée par la méthode `getNamespaceURI`.

En voici un exemple. Supposons que l'analyseur découvre les éléments suivants :

```
<xsd:schema
  xmlns:xsl="http://www.w3.org/2001/XMLSchema">
```

784 Au cœur de Java 2 - Fonctions avancées

Il signale alors ceci :

- `nom qualifié = xsd: schema`
- `nom local = schema`
- `URI de l'espace de noms = http://www.w3.org/2001/XMLSchema`

INFO

Si la reconnaissance des espaces de noms est désactivée, `getLocalName` et `getNamespaceURI` renvoient `null`.

`org.w3c.dom.Node` 1.4

- `String getLocalName()`
Renvoie le nom local (sans préfixe alias) ou `null` si l'analyseur ne reconnaît pas les espaces de noms.
- `String getNamespaceURI()`
Renvoie l'URI de l'espace de noms ou `null` si le nœud ne fait pas partie d'un espace de noms ou si l'analyseur ne les reconnaît pas.

`javax.xml.parsers.DocumentBuilderFactory` 1.4

- `boolean isNamespaceAware()`
- `void setNamespaceAware(boolean value)`
La propriété "namespaceAware" du factory. Si elle est réglée sur `true`, les analyseurs générés par ce factory reconnaissent les espaces de noms.

L'analyseur SAX

L'analyseur DOM lit un document XML dans son intégralité, et ce dans une structure de données en arbre. Dans la plupart des applications pratiques, le système DOM fonctionne bien. Toutefois, il peut être tout à fait inefficace si le document est important et que votre algorithme de traitement est suffisamment simple pour pouvoir analyser les nœuds à la volée, sans avoir à inspecter la totalité de la structure arborescente. Dans ces cas, vous utiliserez plutôt l'analyseur SAX. Il signale les événements lorsqu'il analyse les composants de l'entrée XML. Toutefois, il ne stocke pas le document, il revient en effet au gestionnaire des événements de construire une structure de données s'il le souhaite. En fait, l'analyseur DOM se place au-dessus de l'analyseur SAX. Il construit l'arbre DOM à mesure qu'il reçoit les événements de l'analyseur.

Lorsque vous utilisez un analyseur SAX, vous avez besoin d'un gestionnaire qui définit les actions de l'événement pour les différents événements d'analyse. L'interface `ContentHandler` définit plusieurs méthodes de rappel que l'analyseur exécute au moment de parcourir le document. Voici les plus importantes :

- `startElement` et `endElement` sont appelés lorsqu'une balise de début ou de fin est rencontrée.
- `characters` est appelé lorsque des données de caractères sont rencontrées.
- `startDocument` et `endDocument` sont appelés une fois chacun, au début et à la fin du document.

Par exemple, lorsque l'on analyse le fragment

```
<font>
  <name>Helvetica</name>
  <size units="pt">36</size>
</font>
```

l'analyseur s'assure que les appels suivants sont générés :

1. `startElement`, nom d'élément : `font`
2. `startElement`, nom d'élément : `name`
3. `characters`, contenu : `Helvetica`
4. `endElement`, nom d'élément : `name`
5. `startElement`, nom d'élément : `size`, attributs : `units="pt"`
6. `characters`, contenu : `36`
7. `endElement`, nom d'élément : `size`
8. `endElement`, nom d'élément : `font`

Votre gestionnaire doit surcharger ces méthodes et leur faire réaliser toute action que vous souhaitez lorsque vous analysez le fichier. Le programme qui figure à la fin de cette section imprime tous les liens `<a href="...">` dans un fichier HTML. Il surcharge simplement la méthode `startElement` du gestionnaire pour vérifier les liens contenant le nom `a` et un attribut avec le nom `href`. Ceci peut être utile pour implémenter un "Web crawler", un programme qui suit les pages Web par leurs liens.

INFO

Malheureusement, la plupart des pages HTML sont tellement éloignées d'un code XML correct que le programme d'exemple sera incapable de les analyser. Comme nous l'avons déjà vu, le consortium World Wide Web (W3C) recommande aux concepteurs Web d'utiliser le XHTML, c'est-à-dire un dialecte HTML capable d'être affiché par les navigateurs Web actuels mais qui utilise également du XML correct. Reportez-vous à l'adresse <http://www.w3.org/TR/xhtml1/> pour obtenir de plus amples informations sur le XHTML. Etant donné que le consortium applique ce qu'il recommande, leurs pages Web sont écrites en XHTML. Vous pouvez les utiliser pour tester le programme d'exemple. Par exemple, si vous exécutez

```
java SAXTest http://www.w3c.org/MarkUp
```

vous obtiendrez une liste des URL de tous les liens de cette page.

Ce programme est un bon exemple de l'utilisation de SAX. Nous ne nous préoccupons pas ici du contexte dans lequel l'élément `a` apparaît et il est inutile de stocker une structure en arbre.

Voici maintenant comment obtenir un analyseur SAX :

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
```

Vous pouvez maintenant traiter un document :

```
parser.parse(source, handler);
```


786 Au cœur de Java 2 - Fonctions avancées

Ici, `source` peut être un fichier, une chaîne URL ou un flux d'entrée. Le handler appartient à une sous-classe de `DefaultHandler`. La classe `DefaultHandler` définit des méthodes inactives pour les quatre interfaces :

```
ContentHandler
DTDHandler
EntityResolver
ErrorHandler
```

Le programme d'exemple définit un handler qui surcharge la méthode `startElement` de l'interface `ContentHandler`, afin de rechercher les éléments a disposant d'un attribut `href` :

```
DefaultHandler handler = new
    DefaultHandler()
{
    public void startElement(String namespaceURI,
        String lname, String qname, Attributes attrs)
        throws SAXException
    {
        if (lname.equalsIgnoreCase("a") && attrs != null)
        {
            for (int i = 0; i < attrs.getLength(); i++)
            {
                String aname = attrs.getLocalName(i);
                if (aname.equalsIgnoreCase("href"))
                    System.out.println(attrs.getValue(i));
            }
        }
    }
};
```

La méthode `startElement` possède trois paramètres qui décrivent le nom de l'élément. Le paramètre `qname` signale le nom qualifié de la forme `alias:localname`. Si le traitement de l'espace de noms est activé, les paramètres `namespaceURI` et `lname` décrivent l'espace de noms et le nom local (non qualifié).

Comme avec l'analyseur DOM, le traitement de l'espace de noms est désactivé par défaut. Vous pouvez l'activer en appelant la méthode `setNamespaceAware` de la classe `factory` :

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setNamespaceAware(true);
SAXParser saxParser = factory.newSAXParser();
```

L'exemple 10.8 contient le code du programme de l'aspirateur de site. Plus loin dans ce chapitre, vous découvrirez une autre utilisation intéressante de SAX. Une manière simple de transformer une source de données non-XML en XML consiste à signaler les événements SAX qu'un analyseur SAX signalerait. Reportez-vous à la section sur les transformations XSL pour de plus amples détails.

Exemple 10.8 : *SAXTest.java*

```
import java.io.*;
import java.net.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
```

```
/**
 * Ce programme montre comment utiliser un analyseur SAX. Il imprime tous
 * les liens hypertexte d'une page Web XHTML.
 * Usage: java SAXTest url
 */
public class SAXTest
{
    public static void main(String[] args) throws Exception
    {
        String url;
        if (args.length == 0)
        {
            url = "http://www.w3c.org";
            System.out.println("Using " + url);
        }
        else
            url = args[0];

        DefaultHandler handler = new
            DefaultHandler()
        {
            public void startElement(String namespaceURI,
                String lname, String qname, Attributes attrs)
            {
                if (lname.equalsIgnoreCase("a") && attrs != null)
                {
                    for (int i = 0; i < attrs.getLength(); i++)
                    {
                        String aname = attrs.getLocalName(i);
                        if (aname.equalsIgnoreCase("href"))
                            System.out.println(attrs.getValue(i));
                    }
                }
            }
        };

        SAXParserFactory factory = SAXParserFactory.newInstance();
        factory.setNamespaceAware(true);
        SAXParser saxParser = factory.newSAXParser();
        InputStream in = new URL(url).openStream();
        saxParser.parse(in, handler);
    }
}
```

javax.xml.parsers.SAXParserFactory 1.4

- **static SAXParserFactory newInstance()**
Renvoie une instance de la classe SAXParserFactory.
- **SAXParser newSAXParser()**
Renvoie une instance de la classe SAXParser.
- **boolean isNamespaceAware()**
- **void setNamespaceAware(boolean value)**
La propriété "namespaceAware" du factory. Si elle est définie sur true, les analyseurs générés par ce factory reconnaissent les espaces de noms.

788 *Au cœur de Java 2 - Fonctions avancées*

- `boolean isValidating()`
- `void setValidating(boolean value)`

La propriété "validante" du factory. Si elle est réglée sur `true`, les analyseurs générés par ce factory valident leur entrée.

javax.xml.parsers.SAXParser 1.4

- `void parse(File f, DefaultHandler handler)`
- `void parse(String url, DefaultHandler handler)`
- `void parse(InputStream in, DefaultHandler handler)`

Analysent un document XML à partir du fichier donné, de l'URL ou du flux d'entrée et signalent les événements d'analyse sur le handler donné.

org.xml.sax.ContentHandler 1.4

- `void startDocument()`
- `void endDocument()`

Sont appelées au début et à la fin du document.

- `void startElement(String uri, String lname, String qname, Attributes attr)`
- `void endElement(String uri, String lname, String qname)`

Sont appelées au début et à la fin d'un élément.

<i>Paramètres :</i>	<code>uri</code>	l'URI de l'espace de noms (si l'analyseur reconnaît les espaces de noms)
	<code>lname</code>	le nom local sans préfixe alias (si l'analyseur reconnaît les espaces de noms)
	<code>qname</code>	le nom de l'élément (si l'analyseur ne reconnaît pas les espaces de noms) ou le nom qualifié avec le préfixe alias (si l'analyseur signale des noms qualifiés en plus des noms locaux)

- `void characters(char[] data, int start, int length)`
Est appelée lorsque l'analyseur signale les données de caractères.

<i>Paramètres :</i>	<code>data</code>	un tableau de données de caractères
	<code>start</code>	l'index du premier caractère du tableau de données faisant partie des caractères signalés
	<code>length</code>	la longueur de la chaîne de caractères signalés

org.xml.sax.Attributes 1.4

- `int getLength()`
- `String getLocalName(int index)`

Renvoie le nombre d'attributs stockés dans cette collection d'attributs.

Renvoie le nom local (sans préfixe alias) de l'attribut avec l'index donné ou la chaîne vide si l'analyseur ne reconnaît pas les espaces de noms.

- `String getURI(int index)`
Renvoie l'URI de l'espace de noms de l'attribut avec l'index donné ou la chaîne vide si le nœud ne fait pas partie d'un espace de noms ou si l'analyseur ne reconnaît pas les espaces de noms.
- `String getQName(int index)`
Renvoie le nom qualifié (avec le préfixe alias) de l'attribut avec l'index donné ou une chaîne vide si le nom qualifié n'est pas signalé par l'analyseur.
- `String getValue(int index)`
- `String getValue(String qname)`
- `String getValue(String uri, String lname)`
Renvoient la valeur d'attribut à partir d'un index donné, d'un nom qualifié ou de l'URI de l'espace de noms plus le nom local. Renvoient `null` si la valeur n'existe pas.

Génération de documents XML

Vous savez maintenant comment rédiger des programmes Java capables de lire le XML. Intéressons-nous maintenant au processus opposé, à savoir la production d'une sortie XML. Bien entendu, vous pourriez écrire un fichier XML simplement en réalisant une suite d'appels `print`, en imprimant les éléments, les attributs et le contenu du texte. Mais cela ne serait pas une bonne idée. Le code est plutôt ennuyeux, et les erreurs arrivent rapidement si vous ne prêtez pas attention aux caractères spéciaux (comme " ou <) dans les valeurs d'attribut et le contenu du texte. Une meilleure approche consiste à construire un arbre DOM avec le contenu du document, puis à rédiger le contenu de l'arbre. Pour construire un arbre DOM, commencez par un document vide. Pour cela, appelez la méthode `newDocument` de la classe `DocumentBuilder`.

```
Document doc = builder.newDocument();
```

Utilisez la méthode `createElement` de la classe `Document` pour construire les éléments de votre document.

```
Element rootElement = doc.createElement(rootName);
Element childElement = doc.createElement(childName);
```

Utilisez la méthode `createTextNode` pour construire les nœuds de texte :

```
Text textNode = doc.createTextNode(textContents);
```

Ajoutez les éléments racine au document, puis ajoutez les nœuds enfant à leurs parents :

```
doc.appendChild(rootElement);
rootElement.appendChild(childElement);
childElement.appendChild(textNode);
```

A mesure que vous construirez l'arbre DOM, vous aurez peut-être aussi besoin de définir des attributs d'éléments. Appelez alors simplement la méthode `setAttribute` de la classe `Element` :

```
rootElement.setAttribute(name, value);
```

Assez curieusement, l'API DOM ne prend actuellement pas en charge l'écriture d'un arbre DOM dans un flux de sortie. Pour contourner cette restriction, nous utilisons l'API XSLT (*XML Style Sheet Transformation*). Nous appliquons une transformation inactive au document et nous capturerons sa sortie.

790 Au cœur de Java 2 - Fonctions avancées

Pour inclure un nœud dans la sortie, vous devrez également configurer les identifiants SYSTEM et PUBLIC comme propriétés de sortie.

```
// construire la transformation inactive
Transformer t = TransformerFactory
    .newInstance().newTransformer();
// définir les propriétés de sortie pour obtenir un nœud DOCTYPE
t.setOutputProperty(OutputKeys.DOCTYPE-SYSTEM, systemIdentifiant);
t.setOutputProperty(OutputKeys.DOCTYPE-PUBLIC, publicIdentifiant);
// appliquer la transformation inactive
// et envoyer le résultat dans un fichier
t.transform(new DOMSource(doc),
    new StreamResult(new FileOutputStream(file)));
```

Pour obtenir de plus amples informations sur le XSLT, reportez-vous à la prochaine section. Pour l'heure, considérez ce code comme une incantation magique permettant de produire un résultat XML.

INFO

Le fichier XML qui en résulte ne contient aucun blanc (c'est-à-dire aucun saut de ligne ni aucune indentation). Si vous aimez les blancs, réglez la propriété de sortie "OutputKeys.INDENT" sur "yes".

L'Exemple 10.9 est un programme typique qui produit un résultat XML. Il dessine une peinture moderne, un ensemble aléatoire de rectangles colorés (voir Figure 10.6). Pour enregistrer un chef-d'œuvre, nous utiliserons le format Scalable Vector Graphics (SVG). Il s'agit d'un format XML permettant de décrire des graphiques complexes indépendamment du périphérique. Vous trouverez de plus amples informations sur SVG à l'adresse <http://www.w3c.org/Graphics/SVG>. Pour afficher des fichiers SVG, téléchargez l'afficheur Apache Batik (<http://xml.apache.org/batik>) ou le plug-in du navigateur Adobe (<http://www.adobe.com/svg/main.html>). La Figure 10.7 présente l'afficheur Apache Batik.

Figure 10.6

Générer de l'art moderne.



Nous n'allons pas entrer dans les détails sur SVG. Si ce système vous intéresse, nous vous suggérons de vous reporter au didacticiel présent sur le site Adobe. Pour l'heure, nous devons juste savoir comment exprimer un ensemble de rectangles colorés. Voici un exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20000802//EN"
"http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd">
```

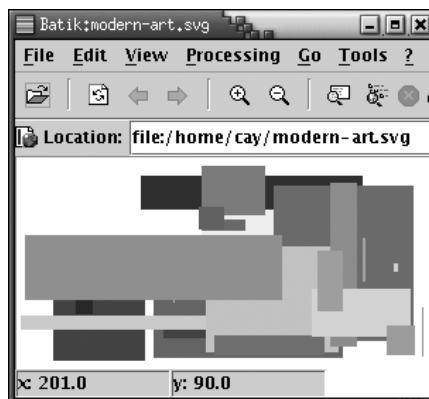
```

<svg width="300" height="150">
<rect x="231" y="61" width="9" height="12" fill="#6e4a13"/>
<rect x="107" y="106" width="56" height="5" fill="#c406be"/>
. . .
</svg>

```

Figure 10.7

L'afficheur
Apache Batik SVG.



Comme vous pouvez le voir, chaque rectangle est décrit à la manière d'un nœud `rect`. La position, la largeur, la hauteur et les couleurs de remplissage sont des attributs. La couleur de remplissage est une valeur RVB exprimée en hexadécimal.

INFO

Comme vous pouvez le voir, SVG utilise de nombreux attributs. En fait, certains sont assez complexes. Par exemple, voici un élément de chemin :

```
<path d="M 100 100 L 300 100 L 200 300 z">
```

Le M montre une commande "moveto", L correspond à "lineto" et Z est "closepath". Apparemment, les concepteurs de ce format de données n'avaient pas vraiment confiance dans l'utilisation du XML pour les données structurées. Dans vos propres formats XML, vous pourrez utiliser des éléments au lieu d'attributs complexes.

Voici le code source du programme. Vous pouvez utiliser la même technique dès que vous aurez besoin de générer une sortie en XML.

Exemple 10.9 : *XMLWriteTest.java*

```

import java.awt.*;
import java.awt.geom.*;
import java.io.*;
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import org.w3c.dom.*;

```

792 *Au cœur de Java 2 - Fonctions avancées*

```
/**
 * Ce programme montre comment écrire un fichier XML. Il enregistre
 * un fichier décrivant un dessin moderne au format SVG.
 */
public class XMLWriteTest
{
    public static void main(String[] args)
    {
        XMLWriteFrame frame = new XMLWriteFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Un bloc contenant un panneau permettant d'afficher un dessin moderne.
 */
class XMLWriteFrame extends JFrame
{
    public XMLWriteFrame()
    {
        setTitle("XMLWriteTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        chooser = new JFileChooser();

        // ajouter le panneau au bloc

        panel = new RectanglePanel();
        add(panel);

        // définir la barre de menu

        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);

        JMenu menu = new JMenu("Fichier");
        menuBar.add(menu);

        JMenuItem newItem = new JMenuItem("Nouveau");
        menu.add(newItem);
        newItem.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                { panel.newDrawing(); }
            });

        JMenuItem saveItem = new JMenuItem("Enregistrer");
        menu.add(saveItem);
        saveItem.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    try
                    {
                        saveDocument();
                    }
                }
            });
    }
}
```

```

        catch (TransformerException e)
        {
            JOptionPane.showMessageDialog(
                XMLWriteFrame.this, e.toString());
        }
        catch (IOException e)
        {
            JOptionPane.showMessageDialog(
                XMLWriteFrame.this, e.toString());
        }
    }
}

JMenuItem exitItem = new JMenuItem("Quitter");
menu.add(exitItem);
exitItem.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        { System.exit(0); }
    });
}

/**
 * Enregistre le dessin au format SVG.
 */
public void saveDocument()
    throws TransformerException, IOException
{
    if (chooser.showSaveDialog(this) != JFileChooser.APPROVE_OPTION)
        return;
    File f = chooser.getSelectedFile();
    Document doc = panel.buildDocument();
    Transformer t = TransformerFactory
        .newInstance().newTransformer();

    t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,
        "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd");
    t.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC,
        "-//W3C//DTD SVG 20000802//EN");

    t.transform(new DOMSource(doc),
        new StreamResult(new FileOutputStream(f)));
}

public static final int DEFAULT_WIDTH = 300;
public static final int DEFAULT_HEIGHT = 200;

private RectanglePanel panel;
private JFileChooser chooser;
}

/**
 * Un panneau qui affiche un ensemble de rectangles colorés.
 */
class RectanglePanel extends JPanel
{
    public RectanglePanel()

```


794 *Au cœur de Java 2 - Fonctions avancées*

```

{
    rects = new ArrayList<Rectangle2D>();
    colors = new ArrayList<Color>();
    generator = new Random();

    DocumentBuilderFactory factory
        = DocumentBuilderFactory.newInstance();
    try
    {
        builder = factory.newDocumentBuilder();
    }
    catch (ParserConfigurationException e)
    {
        e.printStackTrace();
    }
}

/**
 * Crée un nouveau dessin aléatoire.
 */
public void newDrawing()
{
    int n = 10 + generator.nextInt(20);
    rects.clear();
    colors.clear();
    for (int i = 1; i <= n; i++)
    {
        int x = generator.nextInt(getWidth());
        int y = generator.nextInt(getHeight());
        int width = generator.nextInt(getWidth() - x);
        int height = generator.nextInt(getHeight() - y);
        rects.add(new Rectangle(x, y, width, height));
        int r = generator.nextInt(256);
        int g = generator.nextInt(256);
        int b = generator.nextInt(256);
        colors.add(new Color(r, g, b));
    }
    repaint();
}

public void paintComponent(Graphics g)
{
    if (rects.size() == 0) newDrawing();
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;

    // dessine tous les rectangles
    for (int i = 0; i < rects.size(); i++)
    {
        g2.setPaint(colors.get(i));
        g2.fill(rects.get(i));
    }
}

/**
 * Crée un document SVG du dessin en cours.
 * @return l'arbre DOM du document SVG

```

```

    */
    public Document buildDocument()
    {
        Document doc = builder.newDocument();
        Element svgElement = doc.createElement("svg");
        doc.appendChild(svgElement);
        svgElement.setAttribute("width", "" + getWidth());
        svgElement.setAttribute("height", "" + getHeight());

        for (int i = 0; i < rects.size(); i++)
        {
            Color c = colors.get(i);
            Rectangle2D r = rects.get(i);
            Element rectElement = doc.createElement("rect");
            rectElement.setAttribute("x", "" + r.getX());
            rectElement.setAttribute("y", "" + r.getY());
            rectElement.setAttribute("width", "" + r.getWidth());
            rectElement.setAttribute("height", "" + r.getHeight());
            rectElement.setAttribute("fill", colorToString(c));
            svgElement.appendChild(rectElement);
        }

        return doc;
    }

    /**
     * Transforme une couleur en valeur hexa.
     * @param c une couleur
     * @return une chaîne de la forme #rrggbb
     */
    private static String colorToString(Color c)
    {
        StringBuffer buffer = new StringBuffer();
        buffer.append(Integer.toHexString(
            c.getRGB() & 0xFFFFFF));
        while(buffer.length() < 6) buffer.insert(0, '0');
        buffer.insert(0, '#');
        return buffer.toString();
    }

    private ArrayList<Rectangle2D> rects;
    private ArrayList<Color> colors;
    private Random generator;
    private DocumentBuilder builder;
}

```

javax.xml.parsers.DocumentBuilder 1.4

- Document newDocument()
Renvoie un document vide.

org.w3c.dom.Document 1.4

- Element createElement(String name)
Renvoie un élément avec le nom donné.
- Text createTextNode(String data)
Renvoie un nœud de texte avec les données fournies.

796 Au cœur de Java 2 - Fonctions avancées

org.w3c.dom.Node 1.4

- `Node appendChild(Node child)`
Ajoute un nœud à la liste des enfants de ce nœud. Renvoie le nœud annexé.

org.w3c.dom.Element 1.4

- `void setAttribute(String name, String value)`
Définit l'attribut avec le nom donné sur la valeur donnée.
- `void setAttributeNS(String uri, String qname, String value)`
Définit l'attribut avec l'URI de l'espace de noms donnée et le nom qualifié sur la valeur donnée.
Paramètres :

<code>uri</code>	l'URI de l'espace de noms ou null
<code>qname</code>	le nom qualifié. S'il a un préfixe alias, <code>uri</code> ne doit pas être null
<code>value</code>	la valeur de l'attribut

javax.xml.transform.TransformerFactory 1.4

- `static TransformerFactory newInstance()`
Renvoie une instance de la classe `TransformerFactory`.
- `Transformer newTransformer()`
Renvoie une instance de la classe `Transformer` qui réalise une transformation inactive ou une identité.

javax.xml.transform.Transformer 1.4

- `void setOutputProperty(String name, String value)`
Définit une propriété de sortie. Voir l'adresse <http://www.w3.org/TR/xslt#output> pour obtenir la liste des propriétés de sortie standard. Les plus utiles sont :
Paramètres :

<code>doctype-public</code>	l'ID publique à utiliser dans la déclaration DOCTYPE
<code>doctype-system</code>	l'ID système à utiliser dans la déclaration DOCTYPE
<code>indent</code>	yes ou no
- `void transform(Source from, Result to)`
Transforme un document XML.

javax.xml.transform.dom.DOMSource 1.4

- `DOMSource(Node n)`
Construit une source à partir du nœud donné. Généralement, `n` est un nœud de document.

javax.xml.transform.stream.StreamResult 1.4

- `StreamResult(File f)`
- `StreamResult(OutputStream out)`
- `StreamResult(Writer out)`
- `StreamResult(String systemID)`
Construisent un résultat de flux à partir d'un fichier, d'un flux, d'un rédacteur ou d'une ID système (généralement une URL relative ou absolue).

Transformations XSL

Comme vous l'avez déjà vu, le format XML constitue un format d'échange très utile pour les données structurées. Toutefois, la plupart des données XML n'ont pas pour objectif d'être consultées par l'utilisateur final. Dans cette section, vous découvrirez comment transformer des données XML dans des formats de présentation, comme le HTML ou le texte brut.

INFO

Dans cette section, vous verrez comment transformer des données XML en HTML afin de les afficher dans un navigateur. Vous pouvez également afficher les fichiers XML directement dans ce même navigateur. Par exemple, Internet Explorer présente les documents XML sous forme d'arbres. Par ailleurs, les dernières versions de Netscape, Internet Explorer et Opera vous permettent de spécifier des feuilles de style en cascade (CSS) avec un document XML. Une feuille de style vous permet de contrôler la manière dont le navigateur affichera le contenu des éléments XML. Vous trouverez les spécifications des feuilles de style à l'adresse <http://www.w3c.org/TR/REC-CSS2>. L'article intitulé "XML and CSS : Structured Markup with Display Semantics" de Pankaj Kamthan (<http://tech.irt.org/articles/js198>) contient une discussion très intéressante sur les avantages et les inconvénients de l'utilisation des feuilles de style avec XML.

L'objectif d'une feuille de style dotée de modèles de transformation est de décrire la conversion des documents XML dans un autre format (voir Figure 10.8).

Voici un exemple commun. Nous voulons transformer les fichiers XML contenant les enregistrements des employés en documents HTML. Etudiez ce fichier d'entrée :

```
<staff>
  <employee>
    <name>Carl Cracker</name>
    <salary>75000</salary>
    <hiredate year="1987" month="12" day="15" />
  </employee>
  <employee>
    <name>Harry Hacker</name>
    <salary>50000</salary>
    <hiredate year="1989" month="10" day="1" />
  </employee>
  <employee>
    <name>Tony Tester</name>
    <salary>40000</salary>
    <hiredate year="1990" month="3" day="15" />
  </employee>
</staff>
```

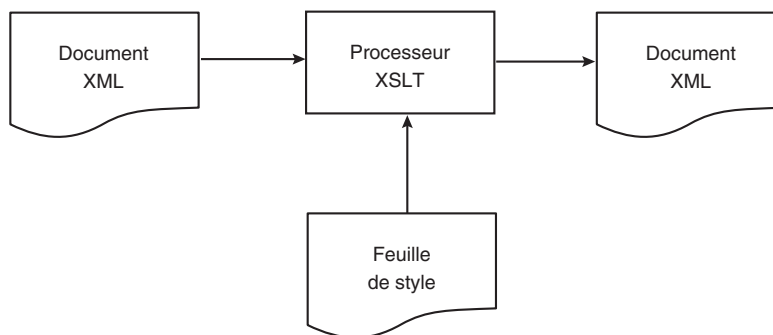
Le résultat souhaité est un tableau HTML :

```
<table border="1">
<tr>
<td>Carl Cracker</td><td>$75000.0</td><td>1987-12-15</td>
</tr>
<tr>
<td>Harry Hacker</td><td>$50000.0</td><td>1989-10-1</td>
</tr>
<tr>
<td>Tony Tester</td><td>$40000.0</td><td>1990-3-15</td>
</tr>
</table>
```

798 Au cœur de Java 2 - Fonctions avancées

Ce tableau peut être incorporé dans un document HTML plus grand, par exemple une page créée avec la technologie JavaServer Pages.

Figure 10.8
Application
des transformations
XSL.



La spécification XSL est assez complexe et des ouvrages entiers lui ont été consacrés. Nous ne pouvons pas traiter ici de toutes les fonctions de XSLT, nous allons donc simplement aborder un exemple représentatif. Vous trouverez de plus amples informations dans l'ouvrage *Essential XML* de Don Bonx, et al. La spécification XSLT est disponible à l'adresse <http://www.w3.org/TR/xslt>.

Une feuille de style dotée de modèles de transformation a la forme suivante :

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:output method="html"/>
  modèle1
  modèle2
  . . .
</xsl:stylesheet>
  
```

Dans notre exemple, les éléments `xsl:output` spécifient la méthode sous la forme HTML. D'autres paramètres de méthode valides sont `xml` et `text`.

Voici un modèle habituel :

```

<xsl:template match="/staff/employee">
  <tr><xsl:apply-templates/></tr>
</xsl:template>
  
```

La valeur de l'attribut `match` est une expression *XPath*. Le modèle indique : lorsque vous voyez un nœud dans l'ensemble *Xpath* `/staff/employee`, procédez comme suit :

- Emettez la chaîne `<tr>`.
- Continuez à appliquer des modèles à mesure que vous traitez ses enfants.
- Emettez la chaîne `</tr>` après avoir terminé avec tous les enfants.

En d'autres termes, ce modèle génère les indicateurs de lignes de tableaux HTML qui entourent chaque enregistrement d'employé.

Le processeur XSLT commence le traitement en examinant l'élément racine. Lorsqu'un nœud correspond à l'un des modèles, il applique ce modèle. Si plusieurs modèles correspondent, il utilise le meilleur ; reportez-vous à la spécification pour connaître les détails. Lorsque aucun modèle ne correspond, le système réalise une action par défaut. Pour les nœuds de texte, l'opération par défaut consiste à inclure le contenu dans le résultat. Pour les éléments, l'action par défaut consiste à ne créer aucun résultat, mais à continuer à traiter les enfants.

Voici un modèle permettant de transformer les nœuds `name` dans un fichier d'employés :

```
<xsl:template match="/staff/employee/name">
  <td><xsl:apply-templates/></td>
</xsl:template>
```

Comme vous pouvez le voir, le modèle produit les délimiteurs `<td>...</td>` et demande au système de visiter régulièrement les enfants de l'élément `name`. Il n'y a qu'un enfant, le nœud de texte. Lorsque le système visite ce nœud, il émet le contenu du texte (à condition bien sûr qu'il n'existe pas d'autre modèle correspondant).

Vous devrez engager d'autres efforts si vous souhaitez copier des valeurs d'attribut dans le résultat. En voici un exemple :

```
<xsl:template match="/staff/employee/hiredate">
  <td><xsl:value-of select="@year"/>-<xsl:value-of
    select="@month"/>-<xsl:value-of select="@day"/></td>
</xsl:template>
```

Lors du traitement d'un nœud `hiredate`, ce modèle émet :

- la chaîne `<td>` ;
- la valeur de l'attribut `year` ;
- un tiret ;
- la valeur de l'attribut `day` ;
- un tiret ;
- la chaîne `</td>`.

L'instruction `xsl:value-of` calcule la valeur de la chaîne d'un ensemble de nœuds. L'ensemble de nœuds est spécifié par la valeur XPath de l'attribut `select`. Dans ce cas, le chemin est relatif au nœud actuellement traité. L'ensemble de nœuds est transformé en chaîne par concaténation des valeurs de chaînes de tous les nœuds. La valeur de chaîne d'un nœud d'attribut correspond à sa valeur. La valeur de chaîne d'un nœud de texte correspond à son contenu. La valeur de chaîne d'un nœud d'élément correspond à la concaténation des valeurs de chaînes de ses nœuds enfant (mais non de ses attributs).

L'Exemple 10.11 contient tous les modèles de transformation permettant de transformer un fichier XML avec les enregistrements des employés en tableau HTML.

L'Exemple 10.12 présente un autre ensemble de transformations. L'entrée est le même fichier XML et le résultat est un texte brut compris dans le format de fichiers de propriété ordinaire :

```
employee.1.name=Carl Cracker
employee.1.salary=75000.0
employee.1.hiredate=1987-12-15
employee.2.name=Harry Hacker
```

800 *Au cœur de Java 2 - Fonctions avancées*

```

employee.2.salary=50000.0
employee.2.hiredate=1989-10-1
employee.3.name=Tony Tester
employee.3.salary=40000.0
employee.3.hiredate=1990-3-15

```

Cet exemple utilise la fonction `position()` qui produit la position du nœud actuel, tel qu'il est vu de son parent. Nous obtenons un résultat tout à fait différent, en basculant simplement la feuille de style. Ainsi, vous pouvez utiliser en toute sécurité le XML pour décrire vos données, même si certaines applications ont besoin des informations sous un autre format. Utilisez simplement le XSLT pour générer cet autre format.

La génération de transformations XSL sur la plate-forme Java est extrêmement simple. Définissez un factory de transformation pour chaque feuille de style. Puis récupérez un objet de transformation et indiquez-lui de transformer une source en résultat.

```

File styleSheet = new File(filename);
StreamSource styleSource = new StreamSource(styleSheet);
Transformer t = TransformerFactory
    .newInstance().newTransformer(styleSource);
t.transform(source, result);

```

Les paramètres de la méthode `transform` sont des objets de classes qui implémentent les interfaces `Source` et `Result`. Il existe trois implémentations de l'interface `Source` :

```

DOMSource
SAXSource
StreamSource

```

Vous pouvez construire un `StreamSource` à partir d'un fichier, d'un flux, d'un lecteur ou d'une URL et un `DOMSource` à partir d'un nœud d'un arbre DOM. Par exemple, dans la section précédente, nous avons invoqué la transformation d'identité comme suit :

```

t.transform(new DOMSource(doc), result);

```

Dans notre programme d'exemple, nous effectuons une action un peu plus intéressante. Plutôt que commencer avec un fichier XML existant, nous produisons un lecteur XML SAX qui donne l'illusion d'analyser un fichier XML, en émettant des événements SAX appropriés. En réalité, le lecteur XML lit un fichier uniforme, tel qu'il est décrit au Chapitre 12 du Volume 1. Le fichier d'entrée ressemble à celui-ci :

```

Carl Cracker|75000.0|1987|12|15
Harry Hacker|50000.0|1989|10|1
Tony Tester|40000.0|1990|3|15

```

Le lecteur XML génère des événements SAX lorsqu'il traite l'entrée. Voici une partie de la méthode `parse` de la classe `EmployeeReader` qui implémente l'interface `XMLReader`.

```

AttributesImpl attributes = new AttributesImpl();
handler.startDocument();
handler.startElement("", "staff", "staff", attributes);
while ((line = in.readLine()) != null)
{
    handler.startElement("", "employee", "employee", attributes);
    StringTokenizer t = new StringTokenizer(line, "|");
    handler.startElement("", "name", "name", attributes);
    String s = t.nextToken();
    handler.characters(s.toCharArray(), 0, s.length());
}

```

```

        handler.endElement("", "name", "name");
        .
        handler.endElement("", "employee", "employee");
    }
    handler.endElement("", rootElement, rootElement);
    handler.endDocument();

```

La `SAXSource` pour le transformateur est construite à partir du lecteur XML :

```

t.transform(new SAXSource(new EmployeeReader(),
    new InputSource(new FileInputStream(filename))), result);

```

Il s'agit là d'une astuce ingénieuse permettant de convertir des données existantes non XML en données XML. Bien entendu, la plupart des applications XSLT posséderont déjà des données d'entrée XML et il vous suffira d'invoquer la méthode `transform` sur un `StreamSource`, comme ceci :

```

t.transform(new StreamSource(file), result);

```

Le résultat de la transformation est un objet d'une classe qui implémente l'interface `Result`. La bibliothèque Java propose trois classes :

```

DOMResult
SAXResult
StreamResult

```

Pour stocker le résultat dans un arbre DOM, utilisez un `DocumentBuilder` pour générer un nouveau nœud de document et emballez-le dans un `DOMResult` :

```

Document doc = builder.newDocument();
t.transform(source, new DOMResult(doc));

```

Pour enregistrer le résultat dans un fichier, utilisez un `StreamResult` :

```

t.transform(source, new StreamResult(file));

```

L'Exemple 10.10 contient la totalité du code source. Les Exemples 10.11 et 10.12 contiennent deux feuilles de style. Cet exemple conclut notre discussion sur la prise en charge XML dans la bibliothèque Java. Vous devriez maintenant avoir une bonne idée des principaux avantages du XML, en particulier de l'analyse et de la validation automatisées et du puissant mécanisme de transformation. Bien entendu, toute cette technologie ne fonctionnera que si vous concevez bien votre format XML. Vous devrez vous assurer que les formats sont suffisamment riches pour exprimer tous vos besoins, qu'ils sont stables dans le temps et que vos partenaires professionnels sont d'accord pour accepter vos documents XML. Ces problèmes peuvent être bien plus prenants que traiter les analyseurs, les DTD ou les transformations.

Exemple 10.10 : *TransformTest.java*

```

import java.io.*;
import java.util.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.sax.*;
import javax.xml.transform.stream.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

```


802 *Au cœur de Java 2 - Fonctions avancées*

```

/**
 Ce programme montre les transformations XSL. Il applique
 une transformation à un ensemble d'enregistrements d'employés. Les
 enregistrements sont stockés dans le fichier employee.dat et
 transformés au format XML. Spécifiez la feuille de style sur la ligne
 de commande, par ex. java TransformTest makeprop.xsl
 */
public class TransformTest
{
    public static void main(String[] args) throws Exception
    {
        String filename;
        if (args.length > 0) filename = args[0];
        else filename = "makehtml.xsl";
        File styleSheet = new File(filename);
        StreamSource styleSource = new StreamSource(styleSheet);

        Transformer t = TransformerFactory
            .newInstance().newTransformer(styleSource);
        t.transform(new SAXSource(new EmployeeReader(),
            new InputSource(new FileInputStream("employee.dat"))),
            new StreamResult(System.out));
    }
}

/**
 Cette classe lit le fichier uniforme employee.dat et signale les
 événements de l'analyseur SAX pour agir à la manière d'une analyse
 d'un fichier XML.
 */
class EmployeeReader implements XMLReader
{
    public void parse(InputSource source)
        throws IOException, SAXException
    {
        InputStream stream = source.getByteStream();
        BufferedReader in = new BufferedReader(
            new InputStreamReader(stream));
        String rootElement = "staff";
        AttributesImpl atts = new AttributesImpl();

        if (handler == null)
            throw new SAXException("Pas de gestionnaire de contenu");

        handler.startDocument();
        handler.startElement("", rootElement, rootElement, atts);
        String line;
        while ((line = in.readLine()) != null)
        {
            handler.startElement("", "employee", "employee", atts);
            StringTokenizer t = new StringTokenizer(line, "|");

            handler.startElement("", "name", "name", atts);
            String s = t.nextToken();
            handler.characters(s.toCharArray(), 0, s.length());
            handler.endElement("", "name", "name");

            handler.startElement("", "salary", "salary", atts);
            s = t.nextToken();
            handler.characters(s.toCharArray(), 0, s.length());
        }
    }
}

```

```

        handler.endElement("", "salary", "salary");

        atts.addAttribute("", "year", "year", "CDATA",
            t.nextToken());
        atts.addAttribute("", "month", "month", "CDATA",
            t.nextToken());
        atts.addAttribute("", "day", "day", "CDATA",
            t.nextToken());
        handler.startElement("", "hiredate", "hiredate", atts);
        handler.endElement("", "hiredate", "hiredate");
        atts.clear();

        handler.endElement("", "employee", "employee");
    }

    handler.endElement("", rootElement, rootElement);
    handler.endDocument();
}

public void setContentHandler(ContentHandler newValue)
{ handler = newValue; }

public ContentHandler getContentHandler()
{ return handler; }

// les méthodes suivantes sont simplement des implémentations
// inactives
public void parse(String systemId)
    throws IOException, SAXException {}
public void setErrorHandler(ErrorHandler handler) {}
public ErrorHandler getErrorHandler() { return null; }
public void setDTDHandler(DTDHandler handler) {}
public DTDHandler getDTDHandler() { return null; }
public void setEntityResolver(EntityResolver resolver) {}
public EntityResolver getEntityResolver() { return null; }
public void setProperty(String name, Object value) {}
public Object getProperty(String name) { return null; }
public void setFeature(String name, boolean value) {}
public boolean getFeature(String name) { return false; }

private ContentHandler handler;
}

```

Exemple 10.11 : *makehtml.xml*

```

<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

  <xsl:output method="html"/>

  <xsl:template match="/staff">
    <table border="1"><xsl:apply-templates/></table>
  </xsl:template>

```

804 *Au cœur de Java 2 - Fonctions avancées*

```
<xsl:template match="/staff/employee">
  <tr><xsl:apply-templates/></tr>
</xsl:template>

<xsl:template match="/staff/employee/name">
  <td><xsl:apply-templates/></td>
</xsl:template>

<xsl:template match="/staff/employee/salary">
  <td>${<xsl:apply-templates/></td>
</xsl:template>

<xsl:template match="/staff/employee/hiredate">
  <td><xsl:value-of select="@year"/>-<xsl:value-of
    select="@month"/>-<xsl:value-of select="@day"/></td>
</xsl:template>

</xsl:stylesheet>
```

Exemple 10.12 : *makeprop.xsl*

```
<?xml version="1.0"?>

<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

  <xsl:output method="text"/>

  <xsl:template match="/staff/employee">
employee.<xsl:value-of select="position()" />.name=<xsl:value-of
  select="name/text()" />

employee.<xsl:value-of select="position()" />.salary=<xsl:value-
  of select="salary/text()" />

employee.<xsl:value-of select="position()" />.hire
  date=<xsl:value-of select="hiredate/@year"/>-<xsl:value-of
  select="hiredate/@month"/>-<xsl:value-of select="hiredate/@day" />
</xsl:template>

</xsl:stylesheet>
```

javax.xml.transform.TransformerFactory 1.4

- Transformer newTransformer(Source from)
Renvoie une instance de la classe Transformer qui lit une feuille de style à partir d'une source donnée.

javax.xml.transform.stream.StreamSource 1.4

- StreamSource(File f)
- StreamSource(InputStream in)
- StreamSource(Reader in)
- StreamSource(String systemID)

Construisent une source de flux à partir d'un fichier, d'un flux, d'un lecteur ou d'une ID système (généralement une URL absolue ou relative).

javax.xml.transform.sax.SAXSource 1.4

- `SAXSource(XMLReader reader, InputSource source)`
Construit une source SAX qui obtient des données à partir de la source d'entrée donnée et utilise le lecteur donné pour analyser l'entrée.

org.xml.sax.XMLReader 1.4

- `void setContentHandler(ContentHandler handler)`
Définit le gestionnaire averti des événements d'analyse lorsque l'entrée est analysée.
- `void parse(InputSource source)`
Analyse l'entrée de la source donnée et envoie les événements d'analyse au gestionnaire du contenu.

javax.xml.transform.dom.DOMResult 1.4

- `DOMResult(Node n)`
Construit une source à partir du nœud donné. Généralement, n est un nouveau nœud de document.

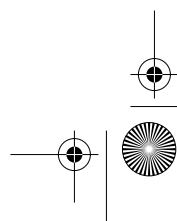
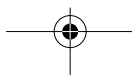
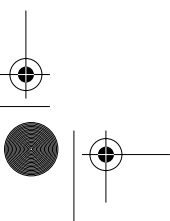
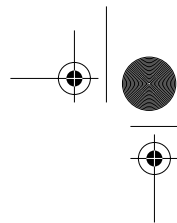
org.xml.sax.helpers.AttributesImpl 1.4

- `void addAttribute(String uri, String lname, String qname, String type, String value)`

Ajoute un attribut à cette collection d'attributs.

<i>Paramètres :</i>	<code>uri</code>	l'URI de l'espace de noms
	<code>lname</code>	le nom local sans le préfixe alias
	<code>qname</code>	le nom qualifié sans le préfixe alias
	<code>type</code>	le type, parmi "CDATA", "ID", "IDREF", "IDREFS", "NMTOKEN", "NMTOKENS", "ENTITY", "ENTITIES" ou "NOTATION"
	<code>value</code>	la valeur d'attribut

- `void clear()`
Supprime tous les attributs de cette collection d'attributs.



11

Annotations

Au sommaire de ce chapitre

- ✓ Ajout de métadonnées à des programmes
- ✓ Un exemple : annoter des gestionnaires d'événements
- ✓ La syntaxe des annotations
- ✓ Les annotations standard
- ✓ L'outil apt pour le traitement des annotations au niveau de la source
- ✓ La création de bytecode

Les annotations sont des indicateurs que vous insérez dans votre code source et qui seront traités par des outils. Les indicateurs peuvent être traités au niveau de la source, mais le compilateur peut aussi les inclure dans les fichiers de classe.

Les annotations ne changent pas la manière dont vos programmes sont compilés. Le compilateur JDK 5.0 utilise un couple d'annotations afin de générer ou de supprimer les alarmes, mais il génère les mêmes fichiers de classe, avec ou sans les annotations. Ces annotations sont simples et, par certaines caractéristiques, atypiques car une tierce partie ne peut ajouter de nouvelles annotations au compilateur.

Pour pouvoir profiter des annotations, vous devez sélectionner un *outil de traitement*, insérer des annotations qu'il comprendra, puis exécuter cet outil de traitement. A l'heure où nous rédigeons, il n'existe pas d'outil de traitement industriel. Malgré tout, nous allons essayer de vous donner un avant-goût de ce que pourraient être ces outils et des capacités que vous pourrez en attendre.

Les annotations peuvent avoir plusieurs utilisations possibles, ce qui peut être déroutant au début. En voici quelques-unes :

- la génération automatique de fichiers auxiliaires, comme les descripteurs de déploiement ou les classes d'information de beans ;
- la génération automatique de code pour le test, la consignation, la sémantique de transaction, etc.

808 Au cœur de Java 2 - Fonctions avancées

Nous commencerons ce chapitre par des concepts de base que nous illustrerons ensuite par des exemples concrets : nous marquons les méthodes comme écouteurs d'événements pour les composants AWT et nous vous montrons un outil de traitement qui analyse les annotations et accroche les écouteurs. Nous verrons ensuite en détail les règles de syntaxe. Nous terminerons ce chapitre par deux exemples avancés. L'un d'entre eux utilise apt, un outil de traitement des annotations qui fait partie du JDK. L'autre utilise la bibliothèque Apache Bytecode Engineering Library pour injecter des bytecodes dans des méthodes annotées.

Ajout de métadonnées aux programmes

Les métadonnées sont des données s'appliquant à d'autres données. Dans le contexte de la programmation, il s'agit de données qui concernent du code. On peut citer comme exemple les commentaires en Javadoc. Ceux-ci décrivent le code mais ne modifient pas sa signification.

Depuis le JDK 5.0, les *annotations* permettent d'insérer des données arbitraires dans votre code source. Vous verrez cependant que les métadonnées ne sont utiles qu'en coordination avec les outils. Comme nous l'indiquons dans ce chapitre, les différents fabricants promettent la sortie prochaine d'outils utiles. Vous verrez ensuite la syntaxe des annotations et des exemples montrant certaines de leurs possibilités. Ceci devrait vous aider à évaluer les outils qui seront bientôt disponibles et pourrait même vous donner des idées pour élaborer les vôtres.

Voici un exemple d'annotation simple :

```
public class MyClass
{
    ...
    @TestCase public void checkRandomInsertions()
}
```

L'annotation `@TestCase` annote la méthode `checkRandomInsertions`.

En Java, une annotation s'utilise comme un *modificateur* et se place avant l'élément annoté, *sans point-virgule* (un modificateur est un mot clé comme `public` ou `static`). Le nom de chaque annotation est précédé du symbole `@`, comme les commentaires Javadoc. Toutefois, les commentaires Javadoc sont insérés entre les délimiteurs `/**` et `*/`, tandis que les annotations font partie du code.

En elle-même, l'annotation `@TestCase` ne fait rien. Elle a besoin d'un outil pour être utile. Un outil de test, par exemple, pourrait appeler toutes les méthodes étiquetées `@TestCase` lorsqu'il teste une classe. Un autre outil pourrait supprimer toutes les méthodes de test d'un fichier de classe de sorte qu'elles ne soient pas livrées avec le programme après le test.

INFO

JUnit, qui est disponible à l'adresse <http://junit.org>, est un outil de test bien connu. Il utilise une convention de nommage permettant de reconnaître les méthodes qui doivent être appelées lors du test. Les noms de méthode doivent commencer par le préfixe `test`. Les conventions de nommage sont généralement fragiles, un nom de méthode comme `testAndSet` serait déroutant pour JUnit. L'article à l'adresse <http://www.langrsoft.com/articles/annotations.html> montre comment modifier JUnit pour utiliser les annotations. Il est possible que cette modification fasse partie des prochaines versions du logiciel.

Les annotations peuvent être définies de manière à avoir des *éléments* comme ceci :

```
@TestCase(id="3352627")
```

Ces éléments peuvent être traités par les outils qui lisent les annotations. D'autres formes d'éléments sont possibles ; nous les verrons plus loin dans ce chapitre.

Outre les méthodes, vous pouvez annoter des classes, des champs et des variables locales ; une annotation peut se trouver n'importe où, tant que vous pouvez placer un modificateur comme `public` ou `static`.

Chaque annotation doit être définie par une *interface d'annotation*. Les méthodes de l'interface correspondent aux éléments de l'annotation. Par exemple, une annotation `TestCase` pourrait être définie par l'interface suivante :

```
import java.lang.annotation.*;
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestCase
{
    String id() default "[none]";
}
```

La déclaration `@interface` crée une vraie interface Java. Les outils de traitement des annotations reçoivent des objets qui implémentent l'interface d'annotation. Un outil appellerait la méthode `id` pour récupérer l'élément `id` d'une annotation `TestCase` particulière.

Les annotations `Target` et `Retention` sont des *méta-annotations*. Elles annotent l'annotation `TestCase`, en la marquant comme une annotation applicable aux méthodes seulement et conservée lorsque le fichier de classe est chargé dans la machine virtuelle. Nous verrons ces méta-annotations en détail dans la suite de ce chapitre.

Vous avez maintenant vu les principes de base des métadonnées et des annotations. Dans la prochaine section, vous verrez un exemple concret de traitement des annotations.

Un exemple : annotation des gestionnaires d'événements

L'une des nombreuses tâches ennuyeuses du programmeur d'interface utilisateur réside dans l'écriture des écouteurs pour les sources d'événements. De nombreux écouteurs ont la forme suivante :

```
myButton.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            doSomething();
        }
    });
```

Dans cette section, nous concevons une annotation qui évitera ce travail fastidieux. L'annotation a la forme

```
@ActionListenerFor(source="myButton") void doSomething() { ... }
```


810 *Au cœur de Java 2 - Fonctions avancées*

Le programmeur n'a plus à appeler `addActionListener`. Chaque méthode est simplement balisée par une annotation. L'Exemple 11.1 montre le programme `ButtonTest` du Chapitre 8 du Volume 1 réimplémenté avec ces annotations.

Nous devons aussi définir une interface d'annotation. Le code se trouve dans l'Exemple 11.2.

Exemple 11.1 : `ButtonTest.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonTest
{
    public static void main(String[] args)
    {
        ButtonFrame frame = new ButtonFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

/**
 * Un cadre avec un panneau de boutons.
 */
class ButtonFrame extends JFrame
{
    public ButtonFrame()
    {
        setTitle("ButtonTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        panel = new JPanel();
        add(panel);

        // Créer les boutons

        yellowButton = new JButton("Jaune");
        blueButton = new JButton("Bleu");
        redButton = new JButton("Rouge");

        // Ajouter les boutons au panneau

        panel.add(yellowButton);
        panel.add(blueButton);
        panel.add(redButton);

        ActionListenerInstaller.processAnnotations(this);
    }

    @ActionListenerFor(source="yellowButton")
    public void yellowBackground()
    {
        panel.setBackground(Color.YELLOW);
    }

    @ActionListenerFor(source="blueButton")
    public void blueBackground()
    {
    }
}
```

```

    {
        panel.setBackground(Color.BLUE);
    }

    @ActionListenerFor(source="redButton")
    public void redBackground()
    {
        panel.setBackground(Color.RED);
    }

    public static final int DEFAULT_WIDTH = 300;
    public static final int DEFAULT_HEIGHT = 200;

    private JPanel panel;
    private JButton yellowButton;
    private JButton blueButton;
    private JButton redButton;
}

```

Exemple 11.2 : ActionListenerFor.java

```

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
{
    String source();
}

```

Bien entendu, les annotations ne font rien par elles-mêmes. Elles se trouvent dans le fichier source. Le compilateur les place ensuite dans le fichier de classe, et la machine virtuelle les charge. Nous avons maintenant besoin d'un mécanisme pour les analyser et installer des écouteurs d'action. C'est le travail de la classe `ActionListenerInstaller`. Le constructeur `ButtonFrame` appelle

```
ActionListenerInstaller.processAnnotations(this);
```

La méthode statique `processAnnotations` énumère toutes les méthodes de l'objet qu'elle a reçu. Pour chaque méthode, elle récupère l'objet d'annotation `ActionListenerFor` et le traite.

```

Class cl = obj.getClass();
for (Method m : cl.getDeclaredMethods())
{
    ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
    if (a != null)...
}

```

Ici, nous utilisons la méthode `getAnnotation` définie dans l'interface `AnnotatedElement`. Les classes `Method`, `Constructor`, `Field`, `Class` et `Package` implémentent cette interface.

Le nom du champ `source` est stocké dans l'objet d'annotation. Nous le récupérons en appelant la méthode `source`, puis recherchons le champ correspondant :

```

String fieldName = a.source();
Field f = cl.getDeclaredField(fieldName);

```

Ceci montre une limite de notre annotation. L'élément `source` doit être le nom d'un champ. Ce ne peut pas être une variable locale.

812 *Au cœur de Java 2 - Fonctions avancées*

Le reste du code est plutôt technique. Pour chaque méthode annotée, nous construisons un objet proxy qui implémente l'interface `ActionListener` et dont la méthode `actionPerformed` appelle la méthode annotée (pour en savoir plus sur les proxys, voir le Chapitre 5 du Volume 1). Les détails ne sont pas importants. La principale observation à faire est que la fonctionnalité de l'annotation a été établie par la méthode `processAnnotations`.

Dans cet exemple, les annotations ont été traitées au moment de l'exécution. Il aurait aussi été possible de les traiter au niveau de la source. Un générateur de code source pourrait avoir produit le code permettant d'ajouter les écouteurs. Les annotations pourraient aussi avoir été traitées au niveau du bytecode. Un éditeur de bytecode pourrait avoir injecté les appels à `addActionListener` dans le constructeur du cadre. Ceci semble complexe, mais des bibliothèques existent qui simplifient grandement cette tâche. Vous en verrez un exemple par la suite.

Notre exemple n'avait pas pour but de devenir un outil sérieux destiné aux programmeurs d'interfaces utilisateur. Une méthode utilitaire ajoutant un écouteur pourrait être aussi pratique pour le programmeur que l'annotation. En fait, la classe `java.beans.EventHandler` tente de le faire. Vous pourriez aisément affiner la classe afin qu'elle soit véritablement utile, en fournissant une méthode qui ajoute le gestionnaire d'événements au lieu de le construire.

Toutefois, cet exemple montre le mécanisme de l'annotation d'un programme et de l'analyse des annotations. Ayant vu un exemple concret, vous êtes maintenant mieux préparé aux sections suivantes, qui décrivent la syntaxe des annotations plus en détail.

Exemple 11.3 : `ActionListenerInstaller.java`

```
import java.awt.event.*;
import java.lang.annotation.*;
import java.lang.reflect.*;

public class ActionListenerInstaller
{
    /**
     * Traite toutes les annotations ActionListenerFor
     * dans l'objet donné.
     * @param obj Un objet dont les méthodes peuvent avoir des
     * annotations ActionListenerFor
     */
    public static void processAnnotations(Object obj)
    {
        try
        {
            Class cl = obj.getClass();
            for (Method m : cl.getDeclaredMethods())
            {
                ActionListenerFor a = m.getAnnotation(
                    ActionListenerFor.class);
                if (a != null)
                {
                    Field f = cl.getDeclaredField(a.source());
                    f.setAccessible(true);
                    addListener(f.get(obj), obj, m);
                }
            }
        }
    }
}
```

```

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }

    /**
     * Ajoute un écouteur d'action qui appelle une méthode donnée.
     * @param source La source de l'événement auquel un écouteur
     * d'action est ajouté
     * @param param Le paramètre implicite de la méthode que
     * l'écouteur appelle
     * @param m La méthode appelée par l'écouteur
     */
    public static void addListener(Object source,
        final Object param, final Method m)
        throws NoSuchMethodException, IllegalAccessException,
        InvocationTargetException
    {
        InvocationHandler handler = new
            InvocationHandler()
        {
            public Object invoke(Object proxy, Method mm, Object[] args)
                throws Throwable
            {
                return m.invoke(param);
            }
        };

        Object listener = Proxy.newProxyInstance(null,
            new Class[] { java.awt.event.ActionListener.class },
            handler);
        Method adder = source.getClass().getMethod("addActionListener",
            ActionListener.class);
        adder.invoke(source, listener);
    }
}

```

java.lang.AnnotatedElement 5.0

- **boolean isAnnotationPresent(Class<? extends Annotation> annotationType)**
Renvoie true si cet élément possède une annotation du type donné.
- **<T extends Annotation> T getAnnotation(Class<T> annotationType)**
Récupère l'annotation du type donné ou null si cet élément n'en possède pas.
- **Annotation[] getAnnotations()**
Récupère toutes les annotations présentes pour cet élément, y compris celles qui sont héritées. S'il n'y en a pas, renvoie un tableau de longueur 0.
- **Annotation[] getDeclaredAnnotations()**
Récupère toutes les annotations déclarées pour cet élément, à l'exception des annotations héritées. S'il n'y en a pas, renvoie un tableau de longueur 0.

Syntaxe des annotations

Dans cette section, nous verrons tout ce qu'il vous faut savoir sur la syntaxe des annotations.

Une annotation se définit par une interface :

```
modificateurs @interface NomAnnotation
{
    déclaration élément1
    déclaration élément2
    ...
}
```

Chaque déclaration d'élément prend la forme

```
type NomElément();
```

ou

```
type NomElément() default valeur;
```

Par exemple, l'annotation suivante possède deux éléments, `assignedTo` et `severity`.

```
public @interface BugReport
{
    String assignedTo() default "[none]";
    int severity() = 0;
}
```

Chaque annotation a le format suivant :

```
@NomAnnotation(NomElément1=valeur1, NomElément2=Valeur2, ...)
```

Par exemple :

```
@BugReport(assignedTo="Harry", severity=10)
```

L'ordre des éléments importe peu. L'annotation

```
@BugReport(severity=10, assignedTo="Harry")
```

est identique à la précédente.

La valeur par défaut de la déclaration est utilisée si une valeur d'élément n'est pas spécifiée. Par exemple, voyez l'annotation suivante :

```
@BugReport(severity=10)
```

La valeur de l'élément `assignedTo` correspond à la chaîne "[none]".

ATTENTION

Les paramètres par défaut ne sont pas stockés avec l'annotation. Ils sont calculés dynamiquement. Par exemple, si vous modifiez la valeur par défaut de l'élément `assignedTo` pour "[]" et que vous recompilez l'interface `BugReport`, l'annotation `@BugReport(severity=10)` utilisera les nouveaux paramètres par défaut, même dans les fichiers de classe qui ont été compilés avant la modification.

Deux raccourcis permettent de simplifier les annotations.

Lorsque aucun élément n'est spécifié, soit parce que l'annotation n'en a pas, soit parce que tous utilisent la valeur par défaut, vous n'avez pas besoin d'utiliser des parenthèses. Par exemple,

```
@BugReport
```

équivalait à

```
@BugReport(assignedTo="[none]", severity=0)
```

Cette annotation est dite *annotation marqueur*.

L'autre raccourci est l'*annotation de valeur unique*. Si un élément possède le nom spécial `value` et qu'aucun autre élément n'est spécifié, vous pouvez omettre le nom de l'élément et le symbole `=`. Par exemple, si nous avons défini l'interface de l'annotation `ActionListenerFor` de la section précédente sous la forme

```
public @interface ActionListenerFor
{
    String value();
}
```

nous aurions pu écrire les annotations sous la forme

```
@ActionListenerFor("yellowButton")
```

au lieu de

```
@ActionListenerFor(value="yellowButton")
```

Toutes les interfaces d'annotation étendent implicitement l'interface `java.lang.annotation.Annotation`. Cette interface est une interface ordinaire, *et non* une interface d'annotation. Voyez les notes d'API à la fin de cette section pour connaître les méthodes fournies par cette interface.

Vous ne pouvez pas étendre les interfaces d'annotation. En d'autres termes, toutes les interfaces d'annotation étendent directement `java.lang.annotation.Annotation`.

Vous ne fournissez jamais les classes qui implémentent les interfaces d'annotation. Au lieu de cela, la machine virtuelle génère les classes proxy et les objets au moment voulu. Par exemple, lorsque vous demandez une annotation `ActionListenerFor`, la machine virtuelle réalise une opération semblable à ceci :

```
return Proxy.newProxyInstance(ChargeurClasse, ActionListenerFor.class,
    new
    InvocationHandler()
    {
        public Object invoke(Object proxy, Method m, Object[] args)
            throws Throwable
        {
            if (m.getName().equals("source"))
                return valeur de source annotation;
            ...
        }
    });
```

Les déclarations d'éléments dans l'interface d'annotation sont en fait des déclarations de méthode. Les méthodes d'une interface d'annotation ne peuvent pas avoir de paramètres ni de clauses `throws` et elles ne peuvent pas être génériques.

816 *Au cœur de Java 2 - Fonctions avancées*

Le type d'un élément d'annotation est l'un de ceux-ci :

- un type primitif (int, short, long, byte, char, double, float ou boolean) ;
- String ;
- Class (avec un paramètre de type optionnel comme `Class<? extends MyClass>`) ;
- un type enum ;
- un type d'annotation ;
- un tableau des types qui précèdent.

INFO

Un tableau de tableaux n'est pas autorisé.

Voici quelques exemples de déclarations d'éléments valides.

```
public @interface BugReport
{
    enum Status { UNCONFIRMED, CONFIRMED, FIXED, NOTABUG };
    boolean showStopper() default false;
    String assignedTo() default "[none]";
    Class<? extends Testable> testCase() default Testable.class;
    Status status() default Status.UNCONFIRMED;
    TestCase testCase();
    String[] reportedBy();
}
```

Les annotations étant évaluées par le compilateur, toutes les valeurs d'éléments doivent être des constantes de compilation. Par exemple :

```
@BugReport(showStopper=true, assignedTo="Harry",
           testCase=MyTestCase.class
           status=BugReport.Status.CONFIRMED, ...)
```

ATTENTION

Un élément d'annotation ne peut jamais être défini sur `null`. `default` ou `null` ne sont pas non plus autorisés. Ceci peut être quelquefois gênant. Vous devrez trouver d'autres paramètres par défaut, comme `""` ou `Void.class`.

Lorsqu'une valeur d'élément correspond à un tableau, entourez ses valeurs entre accolades, comme ceci :

```
@BugReport(..., reportedBy={"Harry", "Carl"})
```

Vous pouvez omettre les accolades si l'élément possède une seule valeur :

```
@BugReport(..., reportedBy="Joe") // revient à écrire {"Joe"}
```

Un élément d'annotation pouvant être une autre annotation, vous pouvez construire des annotations arbitrairement complexes. Par exemple :

```
@BugReport(testCase=@testCase(id="3352627"), ...)
```

INFO

L'introduction de dépendances circulaires dans les annotations est un tort. Par exemple, si `BugReport` possède un élément du type d'annotation `TestCase`, ce dernier ne peut pas avoir d'élément du type `BugReport`.

Vous pouvez ajouter des annotations aux éléments suivants :

- packages ;
- classes (y compris enum) ;
- interfaces (y compris les interfaces d'annotation) ;
- méthodes ;
- constructeurs ;
- champs d'instance (y compris les constantes enum) ;
- variables locales ;
- variables de paramètres.

Toutefois, les annotations pour les variables locales ne peuvent être traitées qu'au niveau de la source. Les fichiers de classe ne décrivent pas les variables locales. Ainsi, toutes les annotations de variables locales sont rejetées lors de la compilation d'une classe. De même, les annotations pour les packages ne sont pas conservées au-delà du niveau de la source.

INFO

Si vous souhaitez traiter des annotations de variables locales, vous devez écrire votre propre analyseur. L'outil `apt` du JDK (que nous verrons plus loin) ne parcourt pas les variables locales.

Un élément peut avoir plusieurs annotations, à condition qu'elles appartiennent à des types différents. Vous ne pouvez pas utiliser le même type d'annotation plusieurs fois lorsque vous annotez un élément particulier. Par exemple,

```
@BugReport(showStopper=true, reportedBy="Joe")
@BugReport(reportedBy={"Harry", "Carl"})
void myMethod()
```

provoque une erreur de compilation. En cas de problème, vous pouvez concevoir une annotation dont la valeur est un tableau d'annotations plus simples :

```
@BugReport({
    @BugReport(showStopper=true, reportedBy="Joe"),
    @BugReport(reportedBy={"Harry", "Carl"})
})
void myMethod()
```


818 Au cœur de Java 2 - Fonctions avancées

java.lang.annotation.Annotation 5.0

- **Class<? extends Annotation> annotationType()**
Renvoie l'objet `Class` qui représente l'interface d'annotation de cet objet d'annotation. Appeler `getClass` sur un objet d'annotation renverrait la classe elle-même, et non l'interface.
- **boolean equals(Object other)**
Renvoie `true` si `other` est un objet qui implémente la même interface d'annotation que cet objet d'annotation et si tous les éléments de cet objet et `other` sont égaux l'un à l'autre.
- **int hashCode()**
Renvoie un code de hachage compatible avec la méthode `equals`, dérivé du nom de l'interface d'annotation et des valeurs des éléments.
- **String toString()**
Renvoie une représentation de chaîne qui contient le nom de l'interface d'annotation et les valeurs des éléments, par exemple `@BugReport(assignedTo=[none], severity=0)`.

Annotations standard

Le JDK 5.0 définit sept interfaces d'annotation. Trois d'entre elles sont des annotations ordinaires que vous pouvez utiliser pour annoter des éléments de votre code source. Les quatre autres sont des méta-annotations qui décrivent le comportement des interfaces d'annotation. Le Tableau 11.1 vous les présente. Nous les verrons plus en détail par la suite.

Tableau 11.1 : Annotations standard

<i>Interface d'annotation</i>	<i>Applicable à</i>	<i>Objet</i>
<code>Deprecated</code>	Tous	Désigne un élément comme obsolète
<code>SuppressWarnings</code>	Tous sauf les packages et les annotations	Supprime les avertissements du type donné
<code>Override</code>	Méthodes	Vérifie que cette méthode surcharge une méthode de superclasse
<code>Target</code>	Annotations	Spécifie les éléments auxquels cette annotation peut être appliquée
<code>Retention</code>	Annotations	Spécifie la durée pendant laquelle cette annotation est conservée
<code>Documented</code>	Annotations	Indique que cette annotation doit être incluse dans la documentation des éléments annotés
<code>Inherited</code>	Annotations	Indique que cette annotation, lorsqu'elle est appliquée à une classe, est automatiquement héritée à ses sous-classes

Annotations ordinaires

L'annotation `@Deprecated` peut être attachée à tout élément dont l'utilisation n'est plus conseillée. Le compilateur vous avertira lorsque vous en utiliserez un. Cette annotation joue le même rôle que la balise Javadoc `@deprecated`.

L'annotation `@SuppressWarnings` indique au compilateur de supprimer les avertissements d'un type particulier, par exemple :

```
@SuppressWarnings("unchecked cast")
```

La première version du compilateur du JDK 5.0 ne prend pas cette annotation en charge.

L'annotation `@Override` ne s'applique qu'aux méthodes. Le compilateur vérifie qu'une méthode ayant cette annotation surcharge réellement une méthode de la superclasse. Par exemple, si vous déclarez

```
public MyClass
{
    @Override public boolean equals(MyClass other);
    . . .
}
```

le compilateur signale une erreur. Après tout, la méthode `equals` ne surcharge *pas* la méthode `equals` de la classe `Object`. Cette méthode possède un paramètre du type `Object`, et non `MyClass`.

Méta-annotations

La méta-annotation `@Target` est appliquée à une annotation, ce qui restreint les éléments auxquels l'annotation s'applique. Par exemple :

```
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
public @interface TestCase
```

Le Tableau 11.2 indique toutes les valeurs possibles. Elles appartiennent au type énuméré `ElementType`. Vous pouvez spécifier autant de types d'éléments que nécessaire, inclus dans des accolades.

Tableau 11.2 : Types d'éléments pour l'annotation `@Target`

Type d'élément	Annotation applicable à
ANNOTATION_TYPE	Déclarations de type d'annotation
PACKAGE	Packages
TYPE	Classes (en incluant <code>enum</code>) et interfaces (en incluant les types d'annotation)
METHOD	Méthodes
CONSTRUCTOR	Constructeurs
FIELD	Champs (en incluant les constantes <code>enum</code>)
PARAMETER	Paramètres de méthode ou du constructeur
LOCAL_VARIABLE	Variables locales

820 Au cœur de Java 2 - Fonctions avancées

Une annotation ne contenant pas la restriction `@Target` peut être appliquée à n'importe quel élément. Le compilateur vérifie que vous appliquez l'annotation à un élément autorisé uniquement. Par exemple, si vous appliquez `@TestCase` à un champ, il en résultera une erreur de compilation.

La méta-annotation `@Retention` spécifie la durée pendant laquelle une annotation est retenue. Vous spécifierez au moins une des valeurs énoncées dans le Tableau 11.3. Par défaut, la valeur est `RetentionPolicy.CLASS`.

Tableau 11.3 : Règles de rétention pour l'annotation `@Retention`

Règle de rétention	Description
SOURCE	Les annotations ne sont pas incluses dans les fichiers de classe.
CLASS	Les annotations sont incluses dans les fichiers de classe mais la machine virtuelle n'a pas besoin de les charger.
RUNTIME	Les annotations sont incluses dans les fichiers de classe et chargées par la machine virtuelle.

Elles sont disponibles dans l'API de réflexion.

Dans l'Exemple 11.2, l'annotation `@ActionListenerFor` a été déclarée avec `RetentionPolicy.RUNTIME` car nous avons utilisé la réflexion pour traiter les annotations. Dans les deux sections suivantes, vous pourrez voir des exemples de traitement d'annotations au niveau de la source et des fichiers de classe.

La méta-annotation `@Documented` aide les outils de documentation comme Javadoc. Les annotations documentées doivent être traitées strictement de la même manière que les autres modificateurs tels que `protected` ou `static` dans le but de la documentation. L'utilisation d'autres annotations n'est pas incluse dans la documentation. Par exemple, supposons que nous déclarions `@ActionListenerFor` comme une annotation documentée :

```
@Documented
@target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
```

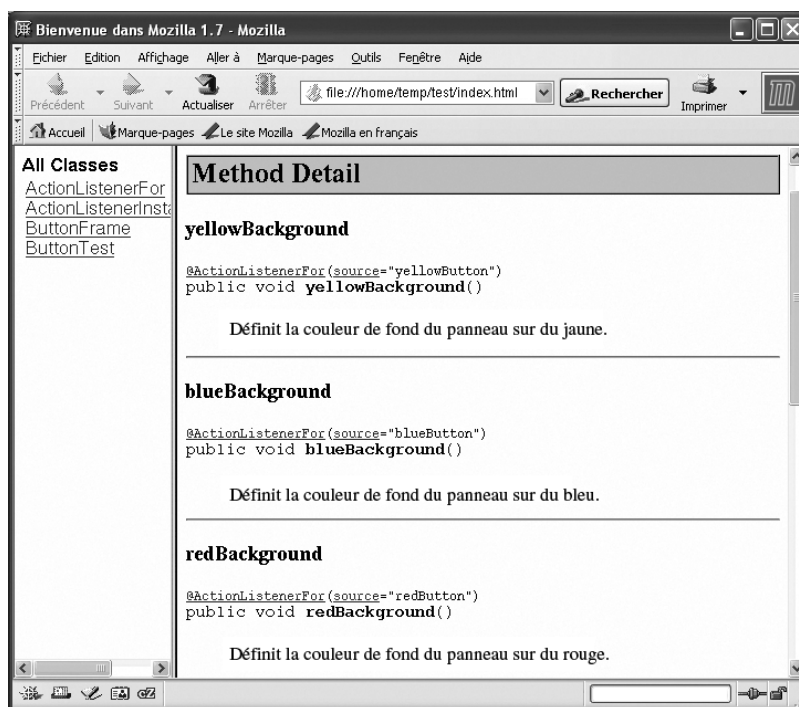
Désormais, la documentation de chaque méthode annotée contient l'annotation, comme indiqué à la Figure 11.1.

Si une annotation est transitoire (comme `@BugReport`) ou constitue un détail de l'implémentation (comme `@ActionListenerFor`), ne documentez pas son utilisation.

INFO

Une annotation peut être appliquée à elle-même. Par exemple, l'annotation `@Documented` est elle-même annotée comme `@Documented`. De plus, la documentation Javadoc sur les annotations indique si elles sont ou non documentées.

Figure 11.1
Annotations
documentées.



La méta-annotation `@Inherited` ne s'applique qu'aux annotations de classes. Lorsqu'une classe possède une annotation héritée, toutes ses sous-classes reçoivent automatiquement la même annotation. Cela facilite la création d'annotations qui se comportent de la même manière que les interfaces marqueur comme `Serializable`.

En fait, l'annotation `@Serializable` serait plus appropriée que l'interface marqueur `Serializable` utilisée sans méthodes. Une classe est sérialisable lorsqu'il existe une prise en charge d'exécution pour la lecture et l'écriture de ses champs et non parce qu'il existe des principes de conception orientée objet. Une annotation décrit cela mieux que ne le fait l'héritage d'interface. Bien entendu, l'interface `Serializable` a été créée dans le JDK 1.1, bien avant que les annotations n'existent.

Imaginons que vous définissiez une annotation héritée `@Persistent` pour indiquer que les objets d'une classe peuvent être sauvegardés dans une base de données. Les sous-classes des classes persistantes sont automatiquement annotées comme persistantes.

```
@Inherited @Persistent { }
@Persistent class Employee { ... }
class Manager extends Employee { ... } // aussi @Persistent
```

Lorsque le mécanisme de persistance recherchera des objets à stocker dans la base de données, il détectera les objets `Employee` et `Manager`.

L'outil apt pour le traitement d'annotations au niveau de la source

L'une des utilisations des annotations est la génération automatique des "fichiers annexes", qui contiennent des informations additionnelles sur les programmes. "L'Enterprise Edition" de Java est célèbre pour les ennuis procurés aux programmeurs du fait de la quantité de code passe-partout. L'entreprise fait des efforts pour développer une série d'annotations standardisées, avec génération automatique de la plupart.

Dans cette section, nous ferons la démonstration de cette technique avec un exemple plus simple. Nous écrirons un programme qui produira automatiquement des classes d'informations sur les beans. Vous balisez les propriétés de bean avec une annotation, puis vous exécutez un outil qui analyse le fichier source et les annotations et écrit ensuite le fichier source de la classe d'information sur les beans. Au lieu d'écrire notre propre analyseur, nous utilisons un outil de traitement des annotations appelé apt qui fait partie du JDK. Nous décrirons en premier lieu comment générer la classe d'information sur les beans, puis nous verrons comment utiliser l'outil apt.

Nous avons vu au Chapitre 6 que la classe `beanInfo` décrit un bean plus précisément que ne peut le faire le processus d'introspection automatique. La classe `beanInfo` répertorie toutes les propriétés du bean. Les propriétés peuvent avoir des éditeurs optionnels. La classe `ChartBeanBeanInfo` du Chapitre 6 en constitue un exemple typique.

Pour éviter d'avoir à écrire des classes `beanInfo`, nous fournissons une annotation `@Property`. Vous pouvez baliser la propriété `get` ou `set` de la propriété, comme ceci :

```
@Property String getTitle() { return title; }
```

ou ainsi :

```
@Property(editor="TitlePositionEditor")
public void setTitlePosition(int p) { titlePosition = p; }
```

L'Exemple 11.4 contient la définition de l'annotation `@Property`. Sachez que l'annotation possède la règle de rétention `SOURCE`. Nous analysons l'annotation au niveau de la source uniquement. Elle n'est pas incluse dans les fichiers de classe et n'est donc pas disponible pendant la réflexion.

Exemple 11.4 : `Property.java`

```
import java.lang.annotation.*;

@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Property
{
    String editor() default "";
}
```

INFO

Il aurait été logique de déclarer l'élément `editor` pour qu'il ait le type `Class`. Toutefois, l'outil de traitement des annotations ne peut pas récupérer les annotations du type `Class` parce que la signification d'une classe peut dépendre de facteurs externes (comme le chemin ou les chargeurs de la classe). Nous utilisons donc une chaîne pour spécifier le nom de la classe de l'éditeur.

Pour générer automatiquement la classe d'information sur le bean pour une classe nommée *BeanClass*, procédez comme suit :

1. Ecrivez un fichier source *BeanClassBeanInfo.java*. Déclarez la classe *BeanClassBeanInfo* de sorte qu'elle étende *SimpleBeanInfo* et surchargez la méthode *getPropertyDescriptors*.
2. Pour chaque méthode annotée, récupérez le nom de la propriété en enlevant le préfixe *get* ou *set* et en enlevant la majuscule du début.
3. Pour chaque propriété, écrivez une instruction permettant de construire un *PropertyDescriptor*.
4. Si la propriété possède un éditeur, écrivez un appel de méthode à *setPropertyEditorClass*.
5. Ecrivez le code permettant de renvoyer un tableau de tous les descripteurs de propriétés.

Par exemple, l'annotation

```
@Property(editor="TitlePositionEditor")
public void setTitlePosition(int p) { titlePosition = p; }
```

de la classe *ChartBean* est traduite en

```
public class ChartBeanBeanInfo extends java.beans.SimpleBeanInfo
{
    public java.beans.PropertyDescriptor[] getProperties()
    {
        java.beans.PropertyDescriptor titlePositionDescriptor
            = new java.beans.PropertyDescriptor(
                "titlePosition", ChartBean.class);
        titlePositionDescriptor.setPropertyEditorClass(
            TitlePositionEditor.class)
        . . .
        return new java.beans.PropertyDescriptor[]
        {
            titlePositionDescriptor,
            . . .
        }
    }
}
```

(Le code passe-partout est imprimé en gris clair.)

Tout ceci est assez simple, à condition que nous puissions trouver toutes les méthodes qui ont été balisées avec l'attribut *@Property*. Plutôt qu'écrire notre propre analyseur, nous utilisons *apt*, l'outil de traitement des annotations, qui fait partie du JDK.

INFO

apt n'est pas lié à l'outil Advanced Packaging distribué par Debian Linux.

Vous trouverez la documentation de l'outil *apt* à l'adresse <http://java.sun.com/j2se/5.0/docs/guide/apt/>. L'outil traite les annotations des fichiers source en utilisant des "factory" d'annotation pour retrouver les éléments de traitement. Il existe un mécanisme permettant de découvrir automatiquement les factory mais, pour des raisons de simplicité, nous spécifions explicitement le factory sur la ligne de commande. Pour appeler l'outil de traitement des informations du bean, exécutez

```
apt -factory BeanInfoAnnotationFactory BeanClass.java
```

824 Au cœur de Java 2 - Fonctions avancées

Le programme apt retrouve les annotations des fichiers source spécifiés sur la ligne de commande. Il choisit ensuite les outils de traitement des annotations à utiliser. Chacun est exécuté à son tour. Si un outil de traitement crée un nouveau fichier source, le processus se répète. Lorsque le traitement ne produit plus de fichiers source, apt appelle le compilateur javac sur tous les fichiers source.

L'analyseur de apt analyse chaque fichier source et énumère les classes, méthodes, champs et variables. L'API utilisée par apt s'appelle "API miroir" car elle ressemble à l'API de réflexion, même si elle opère au niveau de la source. Actuellement, l'API miroir fait partie de la hiérarchie `com.sun` ; or la documentation indique que l'API risque de changer à l'avenir. Nous ne la verrons pas en détail, mais nous examinerons le programme de l'Exemple 11.5 pour vous donner une idée de ses capacités. Vous trouverez la référence complète de l'API dans la documentation de apt.

Les deux premières méthodes de la classe `BeanInfoAnnotationFactory` prennent en charge la recherche de factory (que nous n'utilisons pas). Le factory accepte de traiter les annotations nommées `Property` et ne prend en charge aucune option de la ligne de commande. La troisième méthode produit l'outil de traitement concret.

`BeanInfoAnnotationProcessor` possède une seule méthode publique, `process`, appelée pour chaque fichier. Dans la méthode `process`, nous parcourons toutes les classes, ignorons celles qui ne sont pas publiques, puis parcourons toutes les méthodes et ignorons celles qui ne sont pas annotées avec `@Property`. Voici le squelette du code :

```
public void process()
{
    for (TypeDeclaration t : env.getSpecifiedTypeDeclarations())
    {
        if (t.getModifiers().contains(Modifier.PUBLIC))
        {
            for (MethodDeclaration m : t.getMethods())
            {
                Property p = m.getAnnotation(Property.class);
                if (p != null)
                {
                    traiter les propriétés
                }
            }
        }
        écrire le fichier source d'information sur les beans
    }
}
```

Le code permettant d'écrire le fichier source est simple, ce n'est qu'une succession d'instructions `out.print`. Sachez que nous créons le fichier de résultat par un appel à

```
env.getFile().createSourceFile(beanClassName + "BeanInfo");
```

L'objet `env` encapsule l'environnement de traitement apt. La classe `Filer` est chargée de créer les nouveaux fichiers. `Filer` assure le suivi des fichiers source nouvellement créés, de sorte qu'ils puissent être traités au prochain tour.

Lorsqu'un outil de traitement des annotations détecte une erreur, il fait appel à `Messenger` pour communiquer avec l'utilisateur. Nous émettons, par exemple, un message d'erreur si une méthode a été annotée avec `@Property` mais que son nom ne commence pas par `get`, `set` ou `is` :

```
if (!found)
    env.getMessenger().printError(m.getPosition(),
        "@Property doit être appliqué à
        une méthode getXxx, setXxx ou isXxx");
```

Lors de la compilation de la classe `BeanInfoAnnotationFactory`, vous devez ajouter `tools.jar` à son chemin. Ce fichier se trouve dans le sous-répertoire `lib` de votre installation du JDK :

```
javac -classpath .:jdk/lib/tools.jar BeanInfoAnnotationFactory.java
```

INFO

N'ajoutez le fichier `tools.jar` au chemin de classe que lorsque vous compilez les outils de traitement des annotations. L'outil `apt` retrouve automatiquement la bibliothèque.

Dans le code qui accompagne cet ouvrage, nous vous proposons un fichier annoté intitulé `ChartBean.java`.

Exécutez

```
apt -factory BeanInfoAnnotationFactory ChartBean.java
```

et étudiez le fichier `ChartBeanBeanInfo.java`, généré automatiquement.

Cet exemple montre comment les outils collectent les annotations des fichiers source pour produire d'autres fichiers. Les fichiers générés n'ont pas à être des fichiers source. Les outils de traitement des annotations peuvent choisir de générer des descripteurs XML, des fichiers de propriété, des scripts de shell, de la documentation HTML, etc.

INFO

Certains ont suggéré d'utiliser les annotations pour éliminer un autre élément fastidieux. N'aimeriez-vous pas que les méthodes `get` et `set` soient générées automatiquement ? Par exemple, l'annotation

```
@Property private String title;
```

pourrait produire les méthodes

```
public String getTitle() {return title;}
public void setTitle(String title) { this.title = title; }
```

Ces méthodes doivent toutefois être ajoutées à la même classe. Ceci, qui oblige à modifier un fichier source et non pas seulement à générer un autre fichier, dépasse les capacités de `apt`. On pourrait construire un autre outil dans ce but, mais il dépasserait probablement l'objectif attribué aux annotations. Une annotation doit servir de description pour un élément de code et non de directive pour le modifier.

La structure des annotations assure beaucoup de pouvoir aux programmeurs, mais aussi la possibilité d'abuser. Si les annotations sont utilisées sans limites, elles peuvent générer un code incompréhensible tant pour les programmeurs que pour les outils de développement (les programmeurs C et C++ ont eu la même expérience avec une utilisation inconsidérée du préprocesseur C).

Nous militons pour une utilisation responsable des annotations : vos fichiers source doivent se compiler sans erreurs, même si toutes les annotations en sont retirées. Cette règle ne concerne pas les "méta-sources", qui ne sont transformées en source valide que par un outil de traitement des annotations.

826 *Au cœur de Java 2 - Fonctions avancées*

Exemple 11.5 : BeanInfoAnnotationFactory.java

```
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import com.sun.mirror.type.*;
import com.sun.mirror.util.*;

import java.beans.*;
import java.io.*;
import java.util.*;

/**
 * Cette classe sert à exécuter un outil de traitement des
 * annotations qui crée un fichier BeanInfo.
 */
public class BeanInfoAnnotationFactory implements
    AnnotationProcessorFactory
{
    public Collection<String> supportedAnnotationTypes()
    {
        return Arrays.asList("Property");
    }

    public Collection<String> supportedOptions()
    {
        return Arrays.asList(new String[0]);
    }

    public AnnotationProcessor getProcessorFor(
        Set<AnnotationTypeDeclaration> atds,
        AnnotationProcessorEnvironment env)
    {
        return new BeanInfoAnnotationProcessor(env);
    }

    /**
     * Cette classe est l'outil de traitement qui analyse
     * les annotations @Property.
     */
    private static class BeanInfoAnnotationProcessor
        implements AnnotationProcessor
    {
        BeanInfoAnnotationProcessor(
            AnnotationProcessorEnvironment env)
        {
            this.env = env;
        }

        public void process()
        {
            for (TypeDeclaration t : env.getSpecifiedTypeDeclarations())
            {
                if (t.getModifiers().contains(Modifier.PUBLIC))
                {
                    System.out.println(t);
                    Map<String, Property> props =
                        new TreeMap<String, Property>();
                    for (MethodDeclaration m : t.getMethods())
                    {

```

```

        Property p = m.getAnnotation(Property.class);
        if (p != null)
        {
            String mname = m.getSimpleName();
            String[] prefixes = { "get", "set", "is" };
            boolean found = false;
            for (int i = 0; !found && i < prefixes.length; i++)
                if (mname.startsWith(prefixes[i]))
                {
                    found = true;
                    int start = prefixes[i].length();
                    String name = Introspector.decapitalize(
                        mname.substring(start));
                    props.put(name, p);
                }

            if (!found)
                env.getMessager().printError(m.getPosition(),
                    "@Property doit être appliqué à une méthode
                    getXxx, setXxx ou isXxx");
        }
    }

    try
    {
        if (props.size() > 0)
            writeBeanInfoFile(t.getQualifiedName(), props);
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}

}

/**
 * Ecrit le fichier source de la classe BeanInfo.
 * @param beanClassName Le nom de la classe de bean
 * @param props Une carte des noms de propriété
 * et leurs annotations
 */
private void writeBeanInfoFile(
    String beanClassName, Map<String, Property> props)
    throws IOException
{
    PrintWriter out = env.getFiler().createSourceFile(
        beanClassName + "BeanInfo");
    int i = beanClassName.lastIndexOf(".");
    if (i > 0)
    {
        out.print("package ");
        out.println(beanClassName.substring(0, i));
    }
    out.print("classe publique ");
    out.print(beanClassName.substring(i + 1));
    out.println("BeanInfo extends java.beans.SimpleBeanInfo");
    out.println("{");
    out.println("    public java.beans.PropertyDescriptor[]

```

828 *Au cœur de Java 2 - Fonctions avancées*

```

        getPropertyDescriptors());
    out.println("    {");
    out.println("        try");
    out.println("    {");
    for (Map.Entry<String, Property> e : props.entrySet())
    {
        out.print("        java.beans.PropertyDescriptor ");
        out.print(e.getKey());
        out.println("Descripteur");
        out.print("        = new
            java.beans.PropertyDescriptor(\"");
        out.print(e.getKey());
        out.print("\", ");
        out.print(beanClassName);
        out.println(".class);");
        String ed = e.getValue().editor().toString();
        if (!ed.equals(""))
        {
            out.print("        ");
            out.print(e.getKey());
            out.print("Descriptor.setPropertyEditorClass(");
            out.print(ed);
            out.println(".class);");
        }
    }
    out.println("        return new
        java.beans.PropertyDescriptor[]");
    out.print("    {");
    boolean first = true;
    for (String p : props.keySet())
    {
        if (first) first = false; else out.print(",");
        out.println();
        out.print("        ");
        out.print(p);
        out.print("Descripteur");
    }
    out.println();
    out.println("    };");
    out.println("    }");
    out.println("    catch (
        java.beans.IntrospectionException e)");
    out.println("    {");
    out.println("        e.printStackTrace();");
    out.println("        return null;");
    out.println("    }");
    out.println("    }");
    out.println("}");
    out.close();
}

private AnnotationProcessorEnvironment env;
}
}

```

Ingénierie des bytecodes

Vous avez vu maintenant comment traiter les annotations à l'exécution ou au niveau du code source. Il existe toutefois une troisième possibilité : le traitement au niveau du bytecode. A moins que les annotations ne soient supprimées au niveau de la source, elles existent dans les fichiers de classe. Le format du fichier de classe est documenté (voir <http://java.sun.com/docs/books/vmspec>). Il est plutôt complexe, et il serait compliqué de traiter les fichiers de classe sans posséder de bibliothèques spécialisées. On peut citer par exemple la BCEL, la bibliothèque d'ingénierie des bytecodes, disponible à l'adresse <http://jakarta.apache.org/bcel>.

Dans cette section, nous utilisons la BCEL pour ajouter des messages de consignation aux méthodes annotées. Si une méthode est annotée avec

```
@LogEntry(logger=loggerName)
```

nous ajoutons les bytecodes pour l'instruction suivante au début de la méthode :

```
Logger.getLogger(loggerName).entering(className, methodName);
```

Par exemple, si vous annotez la méthode `hashCode` de la classe `Item` comme suit :

```
@LogEntry(logger="global") public int hashCode()
```

un message semblable à celui-ci s'affiche dès que la méthode est appelée :

```
Aug 17, 2004 9:32:59 PM Item hashCode
FINER: ENTRY
```

Pour réaliser cette tâche :

- Chargez les bytecodes dans le fichier de classe.
- Retrouvez toutes les méthodes.
- Pour chacune, vérifiez si elle possède une annotation `LogEntry`.
- Si oui, ajoutez les bytecodes pour les instructions suivantes au début de la méthode

```
ldc nomConsignateur
invokestatic java/util/logging/Logger.getLogger:(
   Ljava/lang/String;)Ljava/util/logging/Logger;
ldc nomClasse
ldc nomMéthode
invokevirtual java/util/logging/Logger.entering:(
   Ljava/lang/String;Ljava/lang/String;)V
```

L'insertion de ces bytecodes peut paraître difficile, mais la bibliothèque BCEL simplifie grandement l'opération. Toutefois, à l'heure où nous mettons sous presse, BCEL 5.1 ne prend pas en charge le traitement des annotations. Celles-ci sont stockées sous la forme de sections "attribute" dans le fichier de classe. Ces sections stockent une grande diversité d'informations, comme les paramètres des modificateurs `public`, `private`, `protected` et `static` et les numéros de ligne pour la prise en charge du débogage. Le format du fichier de classe est extensible et les compilateurs sont libres d'ajouter des sections d'attributs personnalisées. La BCEL permet donc d'intégrer des lecteurs pour les sections des nouveaux attributs.

Nous avons fourni une classe `AnnotationsAttributeReader` qui gère la section d'attributs contenant les annotations. Pour que le code reste simple, nous ne gérons que les annotations dont les

830 *Au cœur de Java 2 - Fonctions avancées*

éléments ont le type `String` et non les paramètres par défaut. Les prochaines versions de la BCEL devraient assurer la prise en charge de la gestion des attributs.

Nous ne décrivons pas en détail la procédure d'analyse et d'insertion des bytecodes. L'important, c'est que le programme de l'Exemple 11.6 édite un fichier de classe et insère un appel de consignation au début des méthodes annotées avec `LogEntry`. Si vous souhaitez connaître les détails de l'ingénierie des bytecodes, nous vous suggérons le manuel de la BCEL, disponible à l'adresse <http://jakarta.apache.org/bcel/manual.html>.

La bibliothèque BCEL vous sera nécessaire pour compiler et exécuter le programme `EntryLogger`. Voici, par exemple, comment ajouter les instructions de consignation au fichier `Item.java` de l'Exemple 11.7 :

```
javac Item.java
javac -classpath .:bcel-5.1.jar EntryLogger.java
java -classpath .:bcel-5.1.jar EntryLogger Item
```

Essayez d'exécuter

```
javap -c Item
```

avant et après la modification du fichier de classe `Item`. Vous voyez les instructions insérées au début des méthodes `hashCode`, `equals` et `compareTo` :

```
public int hashCode();
Code:
  0:   ldc     #85; //Chaîne global
  2:   invokestatic  #80; //Méthode java/util/logging/
      Logger.getLogger:(Ljava/lang/String;)Ljava/util/
      logging/Logger;
  5:   ldc     #86; //Chaîne Item
  7:   ldc     #88; //Chaîne hashCode
  9:   invokevirtual  #84; //Méthode
      java/util/logging/Logger.entering:(
      Ljava/lang/String;Ljava/lang/String;)V
 12:   bipush  13
 14:   aload_0
 15:   getfield      #2; //Description du champ :Ljava/lang/String;
 18:   invokevirtual  #15; //Méthode java/lang/String.hashCode:()I
 21:   imul
 22:   bipush  17
 24:   aload_0
 25:   getfield      #3; //Champ partNumber:I
 28:   imul
 29:   iadd
 30:   ireturn
```

Le programme `SetTest` de l'Exemple 11.8 insère des objets `Item` dans un jeu de hachage. Exécutez-le avec le fichier de classe modifié pour voir les messages de consignation :

```
Aug 18, 2004 10:57:59 AM Item hashCode
FINER: ENTRY
Aug 18, 2004 10:57:59 AM Item hashCode
FINER: ENTRY
Aug 18, 2004 10:57:59 AM Item hashCode
```

```
FINER: ENTRY
Aug 18, 2004 10:57:59 AM Item equals
FINER: ENTRY
[[description=Toaster, partNumber=1729], [description=Microwave,
partNumber=4562]]
```

Remarquez l'appel à `equals` lorsque nous insérons deux fois le même élément.

Cet exemple montre le pouvoir de l'ingénierie des bytecodes. Les annotations servent à ajouter des directives à un programme. Un outil d'édition des bytecodes choisit les directives et modifie les instructions de la machine virtuelle.

Exemple 11.6 : EntryLogger.java

```
import java.lang.annotation.*;
import java.lang.reflect.Proxy;
import java.lang.reflect.InvocationHandler;
import java.io.*;
import java.util.*;

import org.apache.bcel.*;
import org.apache.bcel.Repository;
import org.apache.bcel.classfile.*;
import org.apache.bcel.classfile.FieldOrMethod;
import org.apache.bcel.generic.*;
import org.apache.bcel.util.*;

/**
 * Ajouter des entrées de journal à toutes les méthodes d'une classe qui
 * possède l'annotation LogEntry.
 */
public class EntryLogger
{
    /**
     * Ajoute le code d'entrée de journal à la classe donnée
     * @param args Le nom du fichier de classe à coller
     */
    public static void main(String[] args)
    {
        try
        {
            if (args.length == 0)
                System.out.println("USAGE: java EntryLogger classname");
            else
            {
                Attribute.addAttributeReader("RuntimeVisibleAnnotations",
                    new AnnotationsAttributeReader());
                JavaClass jc = Repository.lookupClass(args[0]);
                ClassGen cg = new ClassGen(jc);
                EntryLogger el = new EntryLogger(cg);
                el.convert();
                File f = new File(
                    Repository.lookupClassFile(cg.getClassName()).getPath());
                cg.getJavaClass().dump(f.getPath());
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

832 *Au cœur de Java 2 - Fonctions avancées*

```
    }  
  }  
  
  /**  
   * Construit un EntryLogger qui insère une consignation dans les  
   * méthodes annotées d'une classe donnée.  
   * @param cg La classe  
   */  
  public EntryLogger(ClassGen cg)  
  {  
    this.cg = cg;  
    cpg = cg.getConstantPool();  
  }  
  
  /**  
   * Convertit la classe en insérant les appels de consignation.  
   */  
  public void convert() throws IOException  
  {  
    for (Method m : cg.getMethods())  
    {  
      AnnotationsAttribute attr  
        = (AnnotationsAttribute) getAttribute(m,  
        "RuntimeVisibleAnnotations");  
      if (attr != null)  
      {  
        LogEntry logEntry = attr.getAnnotation(LogEntry.class);  
        if (logEntry != null)  
        {  
          String loggerName = logEntry.logger();  
          if (loggerName == null) loggerName = "";  
          cg.replaceMethod(m, insertLogEntry(m, loggerName));  
        }  
      }  
    }  
  }  
  
  /**  
   * Ajoute un appel "entering" au début d'une méthode.  
   * @param m La méthode  
   * @param loggerName Le nom du consignateur à appeler  
   */  
  private Method insertLogEntry(Method m, String loggerName)  
  {  
    MethodGen mg = new MethodGen(m, cg.getClassName(), cpg);  
    String className = cg.getClassName();  
    String methodName = mg.getMethod().getName();  
    System.out.printf("Ajout d'instructions de consignation à %s.%s%n",  
      className, methodName);  
  
    int getLoggerIndex = cpg.addMethodref(  
      "java.util.logging.Logger",  
      "getLogger",  
      "(Ljava/lang/String;)Ljava/util/logging/Logger;");  
    int enteringIndex = cpg.addMethodref(  
      "java.util.logging.Logger",  
      "entering",  
      "(Ljava/lang/String;Ljava/lang/String;)V");  
  }
```

```

        InstructionList il = mg.getInstructionList();
        InstructionList patch = new InstructionList();
        patch.append(new PUSH(cpg, loggerName));
        patch.append(new INVOKESTATIC(getLoggerIndex));
        patch.append(new PUSH(cpg, className));
        patch.append(new PUSH(cpg, methodName));
        patch.append(new INVOKEVIRTUAL(enteringIndex));
        InstructionHandle[] ihs = il.getInstructionHandles();
        il.insert(ihs[0], patch);

        mg.setMaxStack();
        return mg.getMethod();
    }

    /**
     * Récupère l'attribut d'un champ ou d'une méthode avec le nom donné.
     * @param fm Le champ ou la méthode
     * @param name Le nom de l'attribut
     * @return L'attribut ou null si aucun attribut n'est trouvé
     *         avec le nom donné
     */
    public static Attribute getAttribute(FieldOrMethod fm, String name)
    {
        for (Attribute attr : fm.getAttributes())
        {
            int nameIndex = attr.getNameIndex();
            ConstantPool cp = attr.getConstantPool();
            String attrName = cp.constantToString(
                cp.getConstant(nameIndex));
            if (attrName.equals(name))
                return attr;
        }
        return null;
    }

    private ClassGen cg;
    private ConstantPoolGen cpg;
}

/**
 * Lecteur à intégrer pour un attribut d'annotation dans le
 * cadre de la BCEL.
 */
class AnnotationsAttributeReader implements
org.apache.bcel.classfile.AttributeReader
{
    public Attribute createAttribute(int nameIndex,
        int length, DataInputStream in,
        ConstantPool constantPool)
    {
        AnnotationsAttribute attribute = new AnnotationsAttribute(
            nameIndex, length, constantPool);
        try
        {
            attribute.read(in, constantPool);
            return attribute;
        }
        catch (IOException e)
        {
        }
    }
}

```


834 *Au cœur de Java 2 - Fonctions avancées*

```

        e.printStackTrace();
        return null;
    }
}

/**
 * Cet attribut décrit un jeu d'annotations.
 * Seuls les attributs d'annotation en chaîne sont pris en charge.
 */
class AnnotationsAttribute extends Attribute
{
    /**
     * Lit cette annotation.
     * @param nameIndex L'indice pour le nom de cet attribut
     * @param length Le nombre d'octets de cet attribut
     * @param cp Le pool constant
     */
    public AnnotationsAttribute (int nameIndex,
                                int length, ConstantPool cp)
    {
        super(Constants.ATTR_UNKNOWN, nameIndex, length, cp);
        annotations = new HashMap<String, Map<String, String>>();
    }

    /**
     * Lit cette annotation.
     * @param in Le flux d'entrée
     * @param cp Le pool constant
     */
    public void read(DataInputStream in, ConstantPool cp)
        throws IOException
    {
        short numAnnotations = in.readShort();
        for (int i = 0; i < numAnnotations; i++)
        {
            short typeIndex = in.readShort();
            String type = cp.constantToString(cp.getConstant(typeIndex));
            Map<String, String> nvPairs = new HashMap<String, String>();
            annotations.put(type, nvPairs);
            short numElementValuePairs = in.readShort();
            for (int j = 0; j < numElementValuePairs; j++)
            {
                short nameIndex = in.readShort();
                String name = cp.constantToString(cp.getConstant(nameIndex));
                byte tag = in.readByte();
                if (tag == 's')
                {
                    short constValueIndex = in.readShort();
                    String value = cp.constantToString(
                        cp.getConstant(constValueIndex));
                    nvPairs.put(name, value);
                }
                else
                {
                    throw new UnsupportedOperationException(
                        "Ne peut gérer que les attributs String");
                }
            }
        }
    }
}

```

```

public void dump(DataOutputStream out)
    throws IOException
{
    ConstantPoolGen cpg = new ConstantPoolGen(getConstantPool());

    out.writeShort(getNameIndex());
    out.writeInt(getLength());
    out.writeShort(annotations.size());
    for (Map.Entry<String, Map<String, String>> entry :
        annotations.entrySet())
    {
        String type = entry.getKey();
        Map<String, String> nvPairs = entry.getValue();
        out.writeShort(cpg.lookupUtf8(type));
        out.writeShort(nvPairs.size());
        for (Map.Entry<String, String> nv : nvPairs.entrySet())
        {
            out.writeShort(cpg.lookupUtf8(nv.getKey()));
            out.writeByte('s');
            out.writeShort(cpg.lookupUtf8(nv.getValue()));
        }
    }
}

/**
 * Récupère une annotation de ce jeu d'annotations.
 * @param annotationClass La classe de l'annotation à récupérer
 * @return L'objet annotation ou null si aucune annotation
 *         correspondante n'est présente
 */
public <A extends Annotation> A getAnnotation(
    Class<A> annotationClass)
{
    String key = "L" + annotationClass.getName() + ";";
    final Map<String, String> nvPairs = annotations.get(key);
    if (nvPairs == null) return null;

    InvocationHandler handler = new
        InvocationHandler()
    {
        public Object invoke(Object proxy,
            java.lang.reflect.Method m, Object[] args)
            throws Throwable
        {
            return nvPairs.get(m.getName());
        }
    };

    return (A) Proxy.newProxyInstance(
        getClass().getClassLoader(),
        new Class[] { annotationClass },
        handler);
}

public void accept(org.apache.bcel.classfile.Visitor v)
{
    throw new UnsupportedOperationException();
}

```

836 *Au cœur de Java 2 - Fonctions avancées*

```

    public Attribute copy(ConstantPool cp)
    {
        throw new UnsupportedOperationException();
    }

    public String toString ()
    {
        return annotations.toString();
    }

    private Map<String, Map<String, String>> annotations;
}

```

Exemple 11.7 : Item.java

```

public class Item
{
    /**
     * Construit un élément.
     * @param aDescription La description de l'élément
     * @param aPartNumber Le numéro de pièce de l'élément
     */
    public Item(String aDescription, int aPartNumber)
    {
        description = aDescription;
        partNumber = aPartNumber;
    }

    /**
     * Récupère la description de cet élément.
     * @return La description
     */
    public String getDescription()
    {
        return description;
    }

    public String toString()
    {
        return "[description=" + description
            + ", partNumber=" + partNumber + "];"
    }

    @LogEntry(logger="global") public boolean equals(Object otherObject)
    {
        if (this == otherObject) return true;
        if (otherObject == null) return false;
        if (getClass() != otherObject.getClass()) return false;
        Item other = (Item) otherObject;
        return description.equals(other.description)
            && partNumber == other.partNumber;
    }

    @LogEntry(logger="global") public int hashCode()
    {
        return 13 * description.hashCode() + 17 * partNumber;
    }

    private String description;
    private int partNumber;
}

```

Exemple 11.8 : SetTest.java

```
import java.util.*;
import java.util.logging.*;

/**
 * Ce programme consigne les appels de méthode equals et hashCode
 * lors de l'insertion d'éléments dans un jeu de hachage.
 */
public class SetTest
{
    public static void main(String[] args)
    {
        Logger.global.setLevel(Level.FINEST);
        Handler handler = new ConsoleHandler();
        handler.setLevel(Level.FINEST);
        Logger.global.addHandler(handler);

        Set<Item> parts = new HashSet<Item>();
        parts.add(new Item("Grille-pain", 1279));
        parts.add(new Item("Micro-ondes", 4562));
        parts.add(new Item("Grille-pain", 1279));
        System.out.println(parts);
    }
}
```

Modifier les bytecodes au moment du chargement

Dans la section précédente, vous avez vu un outil qui modifie les fichiers de classe. Il peut toutefois être ennuyeux d'ajouter un nouvel outil dans la procédure de construction. Une alternative heureuse consisterait à reporter l'ingénierie des bytecodes au moment du chargement de la classe.

Avant le JDK 5.0, il fallait écrire un chargeur de classe personnalisé pour y parvenir. Désormais, l'API d'instrumentation possède une accroche pour installer un transformateur de bytecodes. Celui-ci doit être installé avant que la méthode `main` du programme ne soit appelée. Pour gérer cette exigence, définissez un agent, une bibliothèque chargée pour surveiller un programme. Le code de l'agent peut réaliser les initialisations dans une méthode `premain`.

Voici les étapes nécessaires à la construction d'un agent :

- implémenter une classe avec une méthode

```
public static void premain(String arg, Instrumentation instr)
```

Cette méthode est appelée au chargement de l'agent. L'agent peut obtenir un argument d'une seule ligne de commande, passé dans le paramètre `arg`. Le paramètre `instr` peut servir à installer diverses accroches.

- créer un fichier de manifeste qui définit l'attribut `Premain-Class`, par exemple :

```
Premain-Class: EntryLoggingAgent
```

- emballer le code de l'agent et le manifeste dans un fichier JAR, par exemple :

```
jar cvfm EntryLoggingAgent.jar EntryLoggingAgent.mf *.class
```

Pour lancer un programme Java en même temps que l'agent, utilisez les options suivantes de la ligne de commande :

```
java -javaagent:AgentJARFile=agentArgument . . .
```

838 *Au cœur de Java 2 - Fonctions avancées*

Par exemple, pour exécuter le programme SetTest avec l'agent de consignment, appelez

```
java -javaagent:EntryLoggingAgent.jar=Item -classpath .:bc1-5.1.jar SetTest
```

L'argument Item est le nom de la classe que l'agent doit modifier.

L'Exemple 11.9 montre le code de l'agent. Il installe un transformateur de fichier de classe. Celui-ci vérifie d'abord si le nom de classe correspond à l'argument de l'agent. Dans l'affirmative, il utilise la classe EntryLogger de la section précédente pour modifier les bytecodes. Toutefois, les bytecodes modifiés ne sont pas enregistrés dans un fichier. Le transformateur les renvoie pour chargement dans la machine virtuelle. Autrement dit, cette technique réalise une modification des bytecodes en "juste à temps".

Exemple 11.9 : EntryLoggingAgent.java

```
import java.lang.instrument.*;
import java.io.*;
import java.security.*;

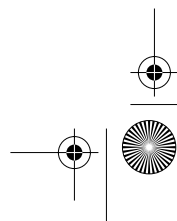
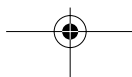
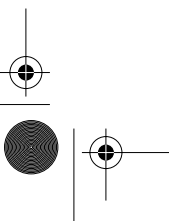
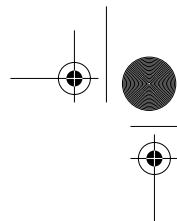
import org.apache.bcel.classfile.*;
import org.apache.bcel.generic.*;

public class EntryLoggingAgent
{
    public static void premain(final String arg, Instrumentation instr)
    {
        System.out.println(instr);
        instr.addTransformer(new
            ClassFileTransformer()
        {
            public byte[] transform(ClassLoader loader,
                String className, Class cl,
                ProtectionDomain pd, byte[] data)
            {
                if (!className.equals(arg)) return null;
                try
                {
                    Attribute.addAttributeReader(
                        "RuntimeVisibleAnnotations",
                        new AnnotationsAttributeReader());
                    ClassParser parser = new ClassParser(
                        new ByteArrayInputStream(data),
                        className + ".java");
                    JavaClass jc = parser.parse();
                    ClassGen cg = new ClassGen(jc);
                    EntryLogger el = new EntryLogger(cg);
                    el.convert();
                    return cg.getJavaClass().getBytes();
                }
                catch (Exception e)
                {
                    e.printStackTrace();
                    return null;
                }
            }
        });
    }
}
```

Dans ce chapitre, vous avez appris à

- ajouter des annotations aux programmes Java ;
- concevoir vos propres interfaces d'annotation ;
- implémenter des outils qui utilisent les annotations.

L'utilisation d'annotations conçues par une autre personne est simple, tout comme la conception d'une interface d'annotation. Mais les annotations sont inutiles en l'absence d'outils. Or concevoir des outils d'annotation est indéniablement plus complexe. Les exemples que nous avons présentés dans ce chapitre vous donnent une idée des possibilités qui s'offrent à vous, et nous espérons que des outils bien plus sophistiqués apparaîtront à l'avenir. Ce chapitre vous a apporté les connaissances de base pour évaluer les outils d'annotation et aura peut-être éveillé votre intérêt pour développer vos propres outils.



Index

Symboles

.class, extension 524
.txt, extension 556
@Deprecated 819
@Documented 820
@Inherited 821
@Override 819
@Retention 820
@SuppressWarnings 819
@Target 819

A

AbstractCollection, classe 84
Accès
 base de registres Windows 719
 fichier, surveillance 556
 tableau 709
Action et permission standard 545
Activatable, classe 263
Activation, descripteurs 263
Activé, thread 22
activeCount, méthode 25
Activer
 gestion des espaces de noms 783
 objets du serveur 263
Actualiser, résultats 177, 180
Adaptateur de pilote, JDBC 153
add, méthode 81
addMouseListener, méthode 365

Adresse Internet 535, 539, 544, 659

Affichage

cellules 370
 choix de propriétés 478
 colonnes 386
 enregistrements de base de données 358
 valeur
 de propriété 490
 des listes 308

Afficheur

cellules
 d'arbre 332
 de liste 308

Agrandissement d'un tableau 457

Agrégation 468

Ajouter

lignes dans le modèle de tableau par défaut 387
 métadonnées 808
 propriété *Voir* Bean

Algorithme 127

de signature 601
 DSA *Voir* DSA
 MD5 578
 personnalisé 133
 RSA *Voir* RSA
 SHA1 578
 sûr de hachage n° 1 *Voir* SHA1

Alias 596, 598

AllPermission 550, 559

Analyse document XML 741

Analyseur 741

DOM 741
 SAX 784

Annotations 807

@Deprecated 819
 @Documented 820
 @Inherited 821
 @Override 819
 @Retention 820
 @SuppressWarnings 819
 @Target 819
 apt 822, 823
 de valeur unique 815
 déclaration d'élément 816
 écouteur 809
 éléments 809
 interface 809
 marqueur 815
 méta-annotations 809, 819
 métadonnées 808
 ordinaires 819
 outil de traitement 807
 standard 818
 syntaxe 814
 utilisations 807

Annulation, transaction 203

API

invocation 715
 Java platform security 612
 JDBC 148

842 Index**Appel**

- C à partir de Java 679
- fonctions JNI 687
- Java dans C/C++ 715
- méthodes
 - contexte 557
 - Java 699
 - non statiques 699
 - statiques 700

Applet

- origine 577
- permission 606
- sécurité 523
- signée 148, 577
 - intranet/Internet 606

appletviewer 543, 608**Application**

- Java 538
- JDBC 147
- localisation *Voir* Localisation
- serveur d'application 147
- Visual Basic 442

applyPattern, méthode 655**apt, outil** 822-823**Arbre** 98, 312, 314

- afficheur de cellules 332
- chemin 322
- infini 347
- modèle 324
 - personnalisé 344
- modification 322
- rouge-noir 99
- structure, modification 322
- terminologie 314

Arguments, ligne de commande 543**ArrayIndexOutOfBoundsException, exception** 711**ArrayList** 94

- classe 86
- et Vector, synchronisation 95

ArrayStoreException, exception 711**Arrêt, thread** 18**ASCII pur** 658**ASN.1** 593**Aspect Metal** 318**Atomiques, opérations** 32**Attribution de nom, convention** 659

- standard 444

Attributs 739, 756**Authentification** 591

- clés publiques/privées *Voir* Clé
- message 583
- utilisateurs 528, 563

Automatique, validation 203**Automatisé**

- mise à jour 204
- mode 204

Autorisations, sémaphores 70**autoscrolls, propriété** 451**Auto-signature de certificat** 597**available, méthode** 409**Avancés, types SQL** 161**await, méthode** 37, 69**AWT** 612**B****Bac à sable** 539, 606, 611**Barres de progression** 399**Barrières cycliques** 69**Base de données**

- COREJAVA 153
- Derby 153
- enregistrements, affichage 358
- hiérarchique 207
- HSQldb 153
- McKoi 153
- programmation 143
- remplissage 164
- structure 184
- tester l'installation 156
- URL 154

Base de registres 719

- clé 720
- éditeur 719
- service de démarrage 234

Basic encoding rules *Voir* BER**BasicPermission, classe** 547**Bean** 441

- agrégation 468
- ajout, propriété 470
- Chart 456
- classe 453

- composer 448
- construction 456
- descripteur de propriétés 471
- écriture 443
- événements par défaut 468
- ImageViewerBean 445-446
- JAR 446
- propriétés
 - indexées 456
 - liées 457
 - par défaut 468
 - simples 456
- représentation graphique 467
- types de propriétés 456
- visuel/non visuel 453

Beanbox

- construction d'une application 446
- éditeur de propriété *Voir* Editeur

BeanInfo, classes 465**BER** 593**Bibliothèque**

- chargée dynamiquement 682
- de méthodes natives 678
- standard 477, 538
- Sun 602

Binaire

- code 32
- recherche 131

binarySearch, méthode 131**Bits, ensembles** 137**BitSet, classe** 137**Blob, objet** 161**Blocage, queue** 52**Blocs synchronisés** 47**Bloqué**

- sortir un thread de l'état 22
- thread 21

Boîte de dialogue dans des fenêtres internes 430**Boolean, type** 693**booleanValue, méthode** 459**Booléenne**

- propriété 454
- valeur, extraire 459

bootstrap registry service 234**breadthFirstEnumeration, méthode** 330

- bulk, operation** 123
- Byte, type** 693, 721
- Bytecode** 527
 - modifier au chargement 837
 - vérification 532
- C**
- C, fichiers** 681
- C binding** *Voir Liaison C*
- C#** 454
- C/C++**
 - appels en Java 715
 - Voir aussi* Langage 678
- C/Java, correspondance de types** 685
- CachedRowSet** 194
- Cacher des colonnes** 386
- Cadre**
 - collections 114
 - extension 125
- Caesar, clé** 528
- Calcul de condensé de message** 579
- calendar, objet** 640
- Callable** 59
- CallNonvirtualXxxMethod, fonction** 702
- Canonical decomposition** *Voir Décomposition*
- Caractères, Unicode** 646, 658, 681, 686
- Carte** 105
 - synchronisée 120
- Cartes de hachage**
 - d'identité 112
 - faibles 109
 - liées 110
- Cascade, présentation des fenêtres** 426
- CDATA**
 - sections 740
 - valeur d'attribut 757
- Cellule**
 - affichage 370
 - d'arbre, afficheur 332
 - éditeur 374
 - modification 370
- sélection 385
- Certificat**
 - auto-signé 597
 - de développeur 610
 - de sécurité 539
 - génération 596
 - empreinte 600
 - fichier 597
 - racine 599
 - signature 599
- Certificate, types** 601
- Certification**
 - Voir aussi* Thawte/Verisign 607
 - X.509 593
- Chaînée (liste), manipulation** 92
- Champs**
 - Objet 692
 - statiques/non-statiques 697
- Changement**
 - propriété 458
 - événement 495
- char, type** 687
- Chargement**
 - bibliothèque 682
 - bytecodes 837
 - modèle 324
- Chargeur de classe** 524
 - d'amorces 524-525
 - d'extensions 524-525
 - systèmes 524-525
- Chart, bean** 456
- checkPermission, méthode** 541, 557
- checkRead, méthode** 557
- Chemin**
 - de l'arbre 322
 - des objets 322
- Chiffres**
 - flux 618
 - symétriques 612
- ChoiceFormat**
 - classe 656
 - objet 656
- Cibles de permission**
 - fichier 547
 - propriété 458
 - socket 548
- Cipher, classe** 612
- Class**
 - classe 659
 - type 693
- Classe**
 - AbstractCollection 84
 - Activatable 263
 - ArrayList 86
 - BasicPermission 547
 - BeanInfo 465
 - Beans 453
 - BitSet 137
 - bytecodes 527
 - chargement dynamique 229
 - chargeur *Voir* Chargeur
 - ChoiceFormat 656
 - Cipher 612
 - Class 659
 - Collator 647
 - Component 453
 - Customizer *Voir* Customizer
 - DateFormat 630, 638
 - de permission
 - hiérarchie 540
 - implémentation 550
 - publique 552
 - DefaultListModel 307
 - DefaultMutableTreeNode 330
 - DesktopManager 432
 - domaine de protection 542
 - DriverManager 155
 - éditeur 476
 - égalité 284
 - enumeration 722
 - étendre 453, 458, 477, 494, 550, 579
 - factory 579
 - FilePermission 539
 - génération, rmic 233
 - Hashtable 135
 - IdentityHashMap 112
 - information de bean *Voir* Information
 - java.beans.Beans 453
 - java.text 632
 - JComponent 458
 - JEditorPane 393

844 Index

Classe (suite)

- JInternalFrame 429
- JLabel 309
- JNIEnv 688
- JTextArea 393
- TextField 393
- LinkedHashMap 110
- LinkedHashSet 110
- LinkedList 87
- Locale 627
 - constructeur 629
- MessageFormat 653
- NumberFormat 632
- Permission 550
- Policy 539
 - standard JDK 1.2 542
- PrintWriter 699
- Properties 736
- PropertyChangeSupport 458
- Random 586
- résolution 524
- ressource, convention
 - d'attribution nom 659
- ResultSet 177-178
- SecureRandom 586
- SecurityManager 557
- sérialisable 251
- SimpleDateFormat 654
- Stack 137
- String 646
- Swing 458
- synchronizedMap 120
- System 538, 700
- système 541
- TreeSet 98
- WeakHashMap 109
- Win32RegKeyNameEnumeration 722

Classement

- collection 88
- forces de tri 647

Clauses grant 608

Clé

- alias 598
- authentification 584
- base de registres 720
- Caesar 528
- clé/valeur, paire 106

- cryptographie clé publique 619
- de décryptage 528
- DSA 586
- génération 584, 596
- magasin *Voir* Magasin de clés
- obtenir 614
- processus complet 588
- publique 619
- publique/privée 583, 597, 611

Clic sur les listes 298

Client

- et serveur 224
- exécution 239
- objet de 227

client.policy 244

Clob, objet 161

Clonage, objets distants 262

CloneNotSupportedException, exception 262

Codage, fichiers texte 657

Code

- ASCII pur 658
- binaire 32
- de hash 95
- GB 658
- ISO 662
- langue 627
- location *Voir* Emplacement du code
- natif 678
 - conversion en ASCII pur 658
- original PC 658
- pays 627
- signature 606
- source de 539
- UTF 686
- UTF-8 modifié 658
- vérification provenance 606

CollationKey, objet 648

Collator

- classe 647
- objet 647

Collection

- cadre 114
- classée 88
- Collection et Map, interface 114
- concrète 85
- de permissions 540

- emballage léger 118
- et threads 58
- interface de 78, 80
- interface pour les collections 114
- structure 114

Collision de hash 96

Colonne

- afficher 386
- cacher 386
- déplacement 353
- et lignes 384
- modifier
 - la largeur 354
 - la taille 384
- nom 148
- sélection 385

Color, propriété 477, 495

COM+ 226

Commande

- javap 698
- SQL, exécution 159
- start 237

Commentaires 741

Comparable, interface 99

Comparaison d'objets 99

compareTo, méthode 99, 646

Compilateur

- Java IDL 270
- JIT 716
- Microsoft C++ 682
- Sun Solaris 682

Compilation Just in Time *Voir* JIT

Complète, décomposition *Voir* Décomposition

Component, classe 453

Composant

- de noms 276
- de texte stylisé 393
- initialisation 494
- JList 295-296, 302
- organismes 414
- Swing 295
- X.500 597

Composition, beans 448

Concrète, collection 85

- ConcurrentModificationException, exception** 90, 121
- Condensé de message** 577, 587
calcul 579
signature 585
- Condition, objet** 36, 37
- Connexion**
établissement 154
gestion 206
mise en commun 207
- Consignation, activité des RMI** 247
- Constructeur** 701
classe Locale 629
de groupe de threads 25
GridBagConstraints 763
par défaut 477, 493
ThreadGroup 25
- Construction**
bean 456
exception 711
gestionnaire de sécurité 556
- Contexte, appel de méthode** 557
- Continue (mise en page), séparateur** 415
- Contour**
de fenêtre, déplacer 431
déplacement 431
- Contraintes**
de grilles 764
propriétés 459
- Contrôle de permission**
modèle standard *Voir* Modèle
pile d'appel 541
- Contrôleurs de progression** 404
- Convention d'attribution de nom**
classes Ressource 659
standard 444
- Conversion, collections et tableaux** 124
- Coopératif, multitâches** 5
- Copie, tableau** 708
- CORBA** 225, 274
et IDL 268
serveurs, implémentation 283
- COREJAVA, base de données** 153
- Correspondance de types (Java/C)** 685
tableaux 706
- Couleur, éditeur de propriété** 494
- CountdownLatch** 69
- CREATE TABLE, SQL** 152
- createSubcontext, méthode** 213
- Création**
fichier JAR 559
tableau Java 709
- Crible d'Ératosthène** 138
- Cryptage** 612
clés publiques/privées *Voir* Clé
de fichier 527
restrictions 528
- Cryptographie, clé publique** 583-584, 619
- Curseur, faire défiler** 179
- Customizer** 491
classe, écriture 493
interface 493
modèle de nom *Voir* Modèle
- CyclicBarrier, classe** 69
- Cygwin** 683, 717
- D**
- Data Workshop** 535
- Date**
formatage 626
interprétation 639
- DateFormat, classe** 630, 638
- DateFormat.xxx** 640
- Débogage**
JDBC 156
JNI 716
- Décapitalisation** 455
- Déclaration**
du type de document *Voir* DTD
DOCTYPE 753
- Décodage des paramètres** 250
- Décompiler un fichier** 32
- Décomposition, mode** 647
Voir aussi Ordre de tri 647
- Décryptage** 528
- DefaultListModel, classe** 307
- DefaultMutableTreeNode, classe** 330
- Défiler**
curseurs 179
listes 296
onglets 419
résultats 177-178
- Définition d'interfaces, langage** 270
- Délégation de tâche** 458
- Délégué**
persistant 504
prédéfini 507
- DELETE, SQL** 152
- DeleteGlobalRef, méthode** 694
- Démarrage**
serveur 236
service de base de registre 234
- Démon, Thread** 25
- Déplacement**
colonnes 353
contour 431
- depthFirstEnumeration, méthode** 330
- Derby** 153
- Désactivation d'une exception** 712
- Describeur**
d'activation 263
de propriété 471
- Description des nœuds** 324
- Désignation, service** 269
- DesktopManager, classe** 432
- Développeur, certificats** 610
- Devises**
identifiants 638
mise en forme 637
- Différence**
entre caractères *Voir* Force de tri
entre les interfaces et les implémentations 78
- Distant, objet**
clonage 262
passage 259
- Distante, méthode**
invocation 230
passage de paramètres 248
- DOCTYPE, déclaration** 753

846 Index

Document

- objet 742
- structure en XML 738
- XML, générer 789

DOM, analyseur 741

- package org.w3c.dom 741

Domaine de protection 540

- classe 542

Données, redondance 149

Double, type 632, 693

doubleValue, méthode 633

DriverManager, classe 155

DSA 584, 585, 586

DTD 738, 752

- commentaires 741
 - élément
 - enfant 739
 - racine 739
 - entités 757
 - instructions de traitement 741
 - références
 - d'entité 740
 - de caractères 740
 - sections CDATA 740
- ### Dynamique, chargement de classes 229

E

Ecouteur

- annotation 809
- d'événement 338
 - ajout 455
 - suppression 455
- de changement de propriété 458
- de listes 302
- de sélection 338
- du modèle d'un tableau 366
- enregistrement 457, 463
- envoi d'événement à 458
- gestion 458
- mise à jour 495
- modèle 459
- suppression 463

Ecriture seule, propriété 454

Editeur

- base de registres 719
- d'applet 609
- de cellule 374

de classe 476

de propriété 470

à interface graphique 480

affichage choix 478

classe 477

couleurs 494

écriture 477

questionnaire 471

multiple 491

par défaut 471

simple 478

tableau 483

hexa 534

personnalisé 374

Edition de valeur de propriété 480

Egalité, classe 284

Élément

- d'annotations 809
- enfant 739
- plus grand
 - d'un tableau 127
 - d'une liste
 - chaînée 128
 - de tableau 128
- racine 739
- supprimer 83

ELEMENT, règle 755

Élimination de la redondance 501

Emballages

- et vues 117
- légers de collection 118

Emplacement

- du code 539
- fichiers de règles
 - modification 543
- géographique *Voir* Paramètres régionaux
- réserve 653
- ressources 659

Empreinte numérique 578

Empreinte *Voir aussi* Certificat

Encodage, paramètres 227, 250

Enfants 314

Enregistrement 148

- d'écouteurs 457, 463

de base de données, affichage 358

Ensemble

- de bits 137
- de propriétés 136
- sous-ensemble 119

Entités de DTD 757

ENTITY, valeur d'attribut 757

Entrées, énumération 106

Énumération 135, 506

- classe 722
- entrées 106
- nœud 330

env, pointeur 687, 716

Enveloppe 458, 478, 632, 719

Environnement

- Cygwin 683
- de génération
 - beans 448
- Forte Community 446

Environnement d'exécution 716

Java 597, 680

Voir aussi JRE 597

Envoi d'événement aux écouteurs 458

equals, méthode 552

Eratosthène, crible 138

Erreur

- de gestion 692, 711
- UnsatisfiedLinkError 680, 684

Espaces

- de noms 782
 - activer la gestion 783
 - traiter 783
- début de chaîne, suppression 633

Etablissement d'une connexion 154

Etats d'un thread 24

Étendre une classe *Voir* Classe

Événement

- changement de propriété 495
- écouter 338
- envoi aux écouteurs 458
- modèle de nom 455
- par défaut 468
- PropertyChangeEvent 495

Exception 535, 633

- ArrayIndexOutOfBoundsException 711
- ArrayStoreException 711
- CloneNotSupportedException 262
- ConcurrentModificationException 90, 121
- construction 711
- dans méthodes natives 711-712
- de sécurité 538
- désactivation 712
- en attente 712
- gestion dans méthodes natives 712
- IllegalAccessException 347
- IllegalArgumentException 640, 712
- IllegalStateException 83
- interception 559
- InterruptedException 7, 12
- IOException 396
- MissingResourceException 660
- non récupérée 27
- NoSuchAlgorithmException 583
- NoSuchElementException 81, 723
- NullPointerException 712
- objet 711
- OutOfMemoryError 712
- ParseException 633
- PropertyVetoException 425
- RemoteException 230, 251, 262
- SecurityException 538, 540
- UnsupportedOperationException 106, 120

ExceptionOccurred, méthode 712

Exchanger 70

Exécutable, thread 21

executeUpdate, méthode 159

Exécution

- client 239
- commandes SQL 159
- programmée 67
- requêtes 168

Executors, classe 63-64, 67

Exemple

- d'arbres 314
- en CORBA 274

exit, méthode 538

exitInternal, méthode 538

Explicite, permission 559

Extensions

- .class 524
- .txt 556

Externe, fonction C 682

Extraction

- valeur
 - booléenne 459
 - de fonctionnalité 468
 - de propriété 478

F

Factory 579, 586, 632, 639

Fenêtre

- de propriétés *Voir* Visual Basic
- déplacer contour 431
- interne 423
 - boîte de dialogue 430
- présentation, cascade et juxtaposition 426

Feuille 314

- de propriétés
 - NetBeans 457
- de style 797

Fichier

- crypté 527
- d'en-tête C 681
- de certificat 597
- de permission 539, 547
- de propriété 155, 660
 - XML 736
- de règles de sécurité 239, 542
 - modification 543, 549
 - propriétés système 548
- décompiler 32
- extensions *Voir* Extensions
- JAR 446, 539, 598, 659
 - création 559
 - signature 606, 607

- ressources *Voir* Ressources
- surveillance d'accès 556
- texte, codage 657

fieldID 693

file read permission 556

FilePermission, classe 539

Filtre

- de tri 364
- modèle 364

FindClass

- fonction 693
- méthode 697

findEditor, méthode 472

firePropertyChange, méthode 458

Flux

- chiffre 618
- progression 408

Fonction

- CallNonvirtualXxxMethod 702
- externe (C) 682
- FindClass 693
- fprintf 699
- GetArrayLength 708
- GetObjectClass 693
- GetStringUTFChars 688
- GetStringUTFLength 688
- GetXxxArrayElements 708
- implémentation 682
- IsAssignableFrom 722
- JNI dans C++ 688
- membre 688
- NewObject 701
- NewStringUTF 688
- NewXxxArray 709
- printf 679
- prototype 682
- ReleaseXxxArrayElements 708
- sprintf 690
- startNameEnumeration 723
- Throw 711
- ThrowNew 711

Force de tri 647

Forêt 314

Format de choix (message) 655

Formatage

- date/heure 638
- dates 626
- masque de 654
- message 653
- monnaies 627, 632
- nombres 626, 632
 - avec printf 685
 - en virgule flottante 691

formatter, objet 632

Forte Community, environnement 446

fprintf, fonction 699

FROM, SQL 150

Full decomposition *Voir*
Décomposition

Full-width *Voir* **Décomposition**

Fusion

- de tableaux 149
- tri 129

Future, interface 59

G

Garbage collector *Voir* **Ramasse-miettes**

GB, code 658

generateCertificate, méthode 601

Générateur de nombre aléatoire 586

Génération

- certificats de sécurité 596
- clés 584
- de classes, rmic 233
- documents XML 789
- stubs 259

Générique, méthode 478

Gestion

- connexions 206
- noms d'utilisateur 206

Gestionnaire

- de sécurité 537
 - instances 558
 - modification 560
 - par défaut 538, 543
 - personnalisé 556
 - remplacement 538

éditeur de propriété 471

exceptions non récupérées 27

GET, méthode 444

GetArrayLength, fonction 708

getAttributes, méthode 212, 745

getAvailableLocales, méthode 633

getBlob, méthode 161

GetBooleanArrayElements, méthode 709

getBundle, méthode 660

GetByteArrayElements, méthode 722

getClassContext, méthode 557

getClob, méthode 161

getCollationKey, méthode 648

getDefault, méthode 630

getDescription, méthode 231, 241

getDisplayName, méthode 630

getFileName, propriété 454

getInstance, méthode 579, 586, 646

GetIntField, méthode 722

getKey, méthode 600

getLocale, méthode 630

GetObjectArrayElement, méthode 708

GetObjectClass, fonction 693

getResource, méthode 659

GetStringUTFChars

fonction 688

méthode 722

GetStringUTFLength, fonction 688

getValue, méthode 720

GetXxxArrayElements, fonction 708

GetXxxArrayRegion, méthode 709

grant, clause 544

Graphique, représentation de bean 467

GridBagConstraints, constructeur 763

Grille

- contraintes 764
- en XML 763, 764

Groupe

- constructeur de thread 25
- de ressources 659
- de threads 25

H

Half-width *Voir* **Décomposition**

Hash

- code 95
- collision 96
- table 95

Hashtable, classe 135

hasMoreElements, méthode 720, 723

Héritage

- hiérarchie 706
- superclasse 455

Hiérarchie

- classes de permission 540
- d'héritage 706

Holder, suffixe 271

HTML et XML 737

HTTP, URL 782

HyperlinkListener, interface 395

I

ID 757

- de champ 693

Identificateur de composant de nom 276

Identification, JAAS 568

IdentityHashMap, classe 112

IDL 226, 270

- et CORBA 268
- et Java 271

méthodes surchargées 272

IllegalAccessException, exception 347

IllegalArgumentException 640, 712

IllegalStateException, exception 83

ImageViewerBean, bean 445-446

Implémentation

- classe de permissions 550
- et interfaces 78, 230
- fonction 682
- queue 79
- serveurs, CORBA 283

- Implication de permission** 550
 - Importation dans magasin de clés** 598
 - Impression, méthode print** 354
 - Indicateur de progression** 399
 - Indulgente, interprétation date** 639
 - Infini, arbre** 347
 - Information de bean** 455
 - init, méthode** 541
 - Initialisation**
 - composant 494
 - machine virtuelle 715
 - initSign, méthode** 587
 - initVerify, méthode** 587
 - INSERT, SQL** 152
 - Installation de JDBC** 153
 - Instances du gestionnaire de sécurité** 558
 - Instructions**
 - de traitement 741
 - préparées 168
 - return 711
 - typedef 685
 - valeur hexa 534
 - Integer, type** 721
 - Interception, exception** 559
 - Interface**
 - Comparable 99
 - Customizer *Voir* Customizer
 - d'annotation 809
 - d'itération 80
 - de collection 78, 80
 - et implémentations 230
 - différence 78
 - graphique, éditeur de propriété 480
 - HyperlinkListener 395
 - langage de définition 270
 - List 115
 - native brute 679
 - pour les collections, Collection et Map 114
 - Remote 230
 - VetoableChangeListener 430
 - International Standards Organization** *Voir* ISO
 - Internationalisation, noms de classe** 658
 - Interne, fenêtre** 423
 - Interpréteur** 543
 - Interrompre thread** 17
 - interrupt, méthode** 25
 - interrupted, méthode** 19
 - InterruptedException, exception** 7, 12, 18
 - Introspection** 470
 - intValue, méthode** 722
 - Inverse, propriété** 481
 - Invocation**
 - API 715
 - de méthode 702
 - distante 227, 230
 - IOException, exception** 396
 - IOR, localiser des objets** 282
 - IsAssignableFrom, fonction** 722
 - isInterrupted, méthode** 19
 - ISO, code** 627, 662
 - isRunning, méthode** 454
 - Itérateurs, Java et STL** 82
 - Itération**
 - interface 80
 - requêtes 160
 - Iterator, méthode** 81
- J**
- JAAS**
 - module d'identification 568
 - service 563
 - JAR**
 - et beans 446
 - fichiers 446
 - jarray, type** 706
 - jarsigner** 598
 - Java**
 - appel
 - dans C/C++ 715
 - de méthodes 699
 - application 538
 - environnement d'exécution 680
 - et IDL 271
 - machine virtuelle *Voir* Machine
 - programme 718
 - sécurité *Voir* Sécurité
 - Java DataBase Connectivity** 143
 - Java IDL, compilateur** 270
 - Java Native Interface** *Voir* JNI
 - Java Plug-in** 606, 608
 - java.beans.Beans, classe** 453
 - java.text, classe** 632
 - Java/C, correspondance de types** 685
 - JavaBeans**
 - persistance 500, 504, 510
 - Voir* Beans
 - javac, programme** 718
 - javah** 681, 723
 - JavaHelp** 393
 - javap, commande** 698
 - jboolean, type** 709
 - jchar, type** 686
 - jclass, type** 692, 701
 - JComponent**
 - classe 458
 - superclasse 458
 - JDBC** 143
 - adaptateur de pilote 153
 - API 148
 - applications typiques 147
 - débogage 156
 - installation 153
 - pilote 146
 - programmation 154
 - programme 164
 - JDBC 3** 144
 - JDK** 599, 602, 629, 678, 718
 - JEditorPane, classe** 393
 - jfieldID, type** 693
 - JInternalFrame, classe** 429
 - JIT** 716
 - JKS** 607
 - JLabel, classe** 309
 - JList, composant** 295-296, 302
 - JNDI, interface** 212
 - JNI**
 - débogage 716
 - fonction *Voir* Fonction
 - JNI_OnLoad, méthode** 684
 - JNIEnv, classe** 688
 - jobject, type** 692, 701
 - JPanel, superclasse** 451

JRE 597

jstring, type 687, 701

JTextArea, classe 393

TextField, classe 393

Juxtaposition de fenêtres 426

JVM 709

JWS DP, téléchargement 290

K

key store Voir **Magasin de clés**

keytool 596, 599, 607

L

Langage

C 678

appel à partir de Java 679

pseudo-code 680

tableaux Java 709

C++ 678

fonctions JNI 688

de définition d'interfaces 270

Fortran 678

Langue 625

code (paramètres régionaux)
627

Voir aussi **Localisation** 625

**Largeur des colonnes,
modification** 354

LDAP 207

accéder aux informations 212

attributs 207, 208

bases hiérarchiques 207

classes d'objet 208

configurer un serveur 208

nom distinctif 208

serveur 208

LDIF 210

Lecture

seule, propriété 454

valeur de propriété 482

Légers, emballages de collection
118

lenient Voir **Indulgent**

Lexicographique, ordre 99

Liaison

C 679

Java/C 692

Ligne

de commande, arguments 543

et colonne 384

sélection 385

LIKE, SQL 151

LinkedHashMap, classe 110

LinkedHashSet, classe 110

LinkedList, classe 87

List, interface 115

Liste 295

afficher des valeurs 308

chaînée 86, 92

plus grand élément 128

pour une queue 79

clic 298

de tableaux 94

plus grand élément 128

défilement 296

écouteurs 302

fonctionnement 298

modèle 302, 306

modifier 306

sélections 298

load, méthode 600

loadClass, méthode 527

Locale

classe 627

constructeur 629

méthode 251

Localisation

application 625

objets

avec des IOR 282

de serveur 234

Location Voir **Paramètres
régionaux**

Locaux (objets), passage 248

Long, type 632

lookup, méthode 241

M

Machine virtuelle 524, 680

initialisation 715

Java 1.2 542

Java Sun Voir **JVM**

suppression 716

Macintosh, ressources 659

Magasin de clés 596-597, 607

importation de certificat 598

Main, méthode 524, 558

malloc 712

Manifestes 447

Manipulation

liste chaînée 92

tableaux Java dans C 709

Map 105

interface pour les collections
114

Marqueur, annotation 815

Masque de formatage 654

Masse, opérations 123

MAX_PRIORITY 24

McKoi 153

MD5 578

MDI 423

Mélanger 129

Message

authentification 583

d'erreur 535, 552, 598, 633

formatage 653

formats de choix 655

signature 587

message digest Voir **Condensé de
message**

MessageFormat, classe 653

Méta-annotations 809

Métadonnée 184, 359

ajouter 808

Metal, aspect 318

Méthode

activeCount 25

add 81

addMouseListener 365

applyPattern 655

binarySearch 131

booleanValue 459

breadthFirstEnumeration 330

checkPermission 541, 557

checkRead 557

compareTo 99, 646

contexte d'appel 557

contrôles de sécurité 557

d'accès 444

d'altération 444

de vérification 540

- DeleteGlobalRef 694
- depthFirstEnumeration 330
- distante 288
 - appel 288
 - invocation 227, 230
 - passage de paramètres 248
- doubleValue 633
- equals 552
- ExceptionOccurred 712
- executeUpdate 159
- exit 538
- exitInternal 538
- factory 632, 639
 - construire un objet 506
- FindClass 697
- findEditor 472
- firePropertyChange 458
- generateCertificate 601
- générique 478
- get 444
- getAttributes 745
- getAvailableLocales 633
- getBlob 161
- GetBooleanArrayElements 709
- getBundle 660
- GetByteArrayElements 722
- getClassContext 557
- getClob 161
- getCollationKey 648
- getDefault 630
- getDescription 231, 241
- getDisplayName 630
- getInstance 579, 586, 646
- GetIntField 722
- getKey 600
- getLocale 630
- GetObjectArrayElement 708
- getResource 659
- GetStringUTFChars 722
- getValue 720
- GetXxxArrayRegion 709
- hasMoreElements 720, 723
- init 541
- initSign 587
- initVerify 587
- interrupt 25
- interrupted 19
- intValue 722
- invocation 702
- isInterrupted 19
- isRunning 454
- Iterator 81
- Java, appel 699
- JNI_OnLoad 684
- load 600
- loadClass 527
- locale 251
- lookup 241
- main 524, 558
- modèle de nom 455
- names 720
- native 719
 - bibliothèque 678
 - exception 711, 712
 - paramètres chaîne 686
 - privée 538
 - surcharge 681
- NewGlobalRef 694
- nextBytes 586
- nextElement 720, 723
- non statique, appel 699
- notify 49
- par défaut 477
- parse 632
- pathFromAncestorEnumeration 330
- postOrderTraversal 330
- preOrderTraversal 330
- print 699
- remplacer 491, 492
- reshape 425
- run 12
- set 444
- setCellRenderer 309
- setFileName 454
- SetIntField 722
- setObject 494
- SetObjectArrayElement 708
- setRunning 454
- setSecurityManager 538
- setSeed 586
- setSelected 425
- setTitle 495
- SetXxxArrayRegion 709
- sign 587
- signature, mutilation 697
- startElement 786
- statique 524, 538, 579, 680
 - appel 700
- stop 50
- subList 119
- substring 119
- surcharge 527, 556
 - et IDL 272
- suspend 50, 51
- synchronisation 251
- System.loadLibrary 680, 683
- update 587
- verify 588
- vetoableChange 430
- Microsoft Query** 150
- MIN_PRIORITY** 24
- Mise à jour**
 - automatisée 204
 - écouteurs 495
 - transaction 203
 - valeur de propriété *Voir* Propriété
- Mise en commun des connexions** 207
- Mise en œuvre**
 - client/serveur 243
 - RMI, vérification 246
- Mise en page continue, séparateur** 415
- MissingResourceException, exception** 660
- Mode**
 - automatisé 204
 - décomposition 647
- Modèle**
 - chargement 324
 - de capitalisation 455
 - de conception 453
 - de filtre 364
 - de nom 453
 - customizer 492
 - événements 455
 - méthodes 455
 - propriétés 453
 - standard 453

852 Index

Modèle (suite)

- de sécurité
 - JDK 1.0 577
 - plate-forme Java 2 538
- de sélection 385
- de tableau 352, 356
 - écouteur 366
 - par défaut, ajouter et supprimer des lignes 387
- écouteur 459
- en trois niveaux 147
- listes 302, 306
- permission standard 540
- personnalisé d'arbre 344

Modification

- arbres et leur structure 322
- cellules 370
- fichier de règles 549
 - emplacement 543
- gestionnaire de sécurité 560
- largeur des colonnes 354
- listes 306
- sélection 339
- taille des colonnes 384

modulo 587

Moniteurs 46

Monnaie, formatage 627

Mort

- thread 23
- verrou 48

Mot clé 453

- native 679
- SQL 150
- static 680

Mot de passe 596

Multiple Document Interface 423

Multitâche 5

- coopératif 5
- préemptif 5

Multithreads 6

Mutilation, signature des méthodes 697

N

names, méthode 720

Natif, code 678

Native

- méthode *Voir* Méthode
- mot clé 679

native2ascii 658, 661, 664

NetBeans, feuille de propriétés 457

newCondition, méthode 37

NewGlobalRef, méthode 694

NewObject, fonction 701

NewStringUTF, fonction 688

NewXxxArray, fonction 709

nextBytes, méthode 586

nextElement, méthode 720, 723

NMTOKENS 757

No decomposition *Voir* Décomposition

Nœud 314

- description 324
- énumération 330

Nom

- classes et packages 526
- composants 276
- d'utilisateur, gestion 206
- de classes, internationalisation 658
- de colonne 148
- de package 693
- local 783
- qualifié 783

Nombre

- aléatoire 586
- générateur 586
- formatage 626
- virgule flottante 632, 636
- formatage 691

Non-statique *Voir* Champ

NORM_PRIORITY 24

NoSuchAlgorithmException 583

NoSuchElementException 81, 723

Notify, méthode 49

Nouveau, thread 20

-noverify, option 535

NullPointerException 712

NumberFormat

- classe 632
- type 633

O

OBDC 145

Object, type 693, 721

Objet

- activer sur le serveur 263
- Blob et Clob 161
- calendar 640
- champ 692
- chemin 322
- CollationKey 648
- collator 647
- comparaison 99
- construire avec une méthode
 - factory 506
- construire depuis les propriétés 505
- de client 227
- de condition 36- 37
- de serveur 227
 - localisation 234
- de stub 229
- distant
 - clonage 262
 - passage 259
- Document 742
- exception 711
- formatter 632
- localisation avec des IOR 282
- locaux, passage 248
- PrintWriter 699
- ResultSetMetaData 359
- TimeZone 645
- verrou 33
- wrapper *Voir* Enveloppe

Obtention clé 614

Onglet 418

- défilement 419
- panneau à 418

Opérations

- atomiques 32
- de masse 123
- optionnelles 122

Option

- certreq 600
- Djava.security.manager 544
- encoding 658
- list 597
- noverify 535

- reverse 658
 - s 698
 - verify 598
 - Optionnelles, opérations** 122
 - ORB** 268
 - Ordre lexicographique** 99
 - org.w3c.dom, package** 741
 - Organisateurs de composants** 414
 - Orientation, de séparateur** 414
 - original PC, code** 658
 - Origine d'applet** 577
 - Outils**
 - apt 822-823
 - Data Workshop 535
 - jarsigner 598
 - keytool 596, 599, 607
 - native2ascii 658, 661, 664
 - policytool 549
 - OutOfMemoryError** 712
- P**
- Package**
 - noms 526, 693
 - org.w3c.dom 741
 - Page de code** 437 658
 - Paire, clé/valeur** 106
 - Panneau à onglets** 418
 - Par défaut**
 - constructeur *Voir* Constructeur
 - éditeur de propriété 471
 - événement 468
 - gestionnaire de sécurité *Voir* Gestionnaire
 - méthode 477
 - paramètres régionaux 630
 - propriétés 468
 - Paramètres**
 - chaîne (String) 686
 - de propriétés, refuser 429
 - encodage 227
 - décodage 250
 - numériques 685
 - passage de, méthodes distantes 248
 - PATH/CLASSPATH** 716
 - régionaux 626
 - code ISO 627
 - par défaut 630
 - polices 640
 - prédéfinis 629
 - variantes 627
 - Parent** 314
 - parse, méthode** 632
 - ParseException, exception** 633
 - Partager un accès entre plusieurs threads** 28
 - Passage**
 - de paramètres aux méthodes distantes 248
 - objets distants 259
 - locaux 248
 - PATH/CLASSPATH** 716
 - pathFromAncestorEnumeration, méthode** 330
 - Pays, code** 627
 - Permission** 537, 539
 - classe 550
 - contrôle, modèle standard 540
 - d'applet 606
 - de lecture de fichier 556
 - explicite 559
 - fichier
 - cibles 547
 - de permission *Voir* Fichier
 - hiérarchie des classes 540
 - implicite 550
 - personnalisée 549
 - propriété (cibles) 548
 - socket (cibles) 548
 - standard 545
 - utilisateurs 563
 - vérification 540
 - Persistance**
 - délégué 504
 - JavaBeans 500, 504, 510
 - Personnalisation**
 - algorithme 133
 - éditeur 374
 - modèle d'arbre 344
 - permission 549
 - Piles** 137
 - d'appel 540, 558
 - machine virtuelle 680
 - Pilote JDBC** 146
 - Planification**
 - coopérative 21
 - préemptive 21
 - Plate-forme Java 2, sécurité** 539
 - Plus grand élément**
 - d'un tableau 127
 - d'une liste
 - chaînée 128
 - de tableau 128
 - Poignées** 318
 - Pointeur env** 687, 716
 - Points**
 - d'insertion 678
 - de sauvegarde 203
 - Polices installées, paramétrages régionaux** 640
 - Policy** *Voir* Fichier de règles de sécurité
 - Policy, classe** 539, 542
 - policytool** 549
 - Pool de threads** 63
 - Portabilité** 678, 719
 - Post-construction** 507
 - postOrderTraversal, méthode** 330
 - Préemptif, multitâches** 5
 - preOrderTraversal, méthode** 330
 - Préparées, instructions** 168
 - Préparer requête** 169
 - Présentation des fenêtres, cascade et juxtaposition** 426
 - Primaire, différence entre caractères** 647
 - print, méthode** 354, 699
 - printf** 679
 - formatage de nombres 685
 - PrintWriter**
 - classe 699
 - objet 699
 - Priorités**
 - d'un thread 24
 - queues 104
 - Privilèges** *Voir* Sécurité
 - Processus et thread** 6

Programmation

des bases de données 143

JDBC 154

Programmeexécutable, signature *Voir* Code

JDBC 164

Progression

barres 399

contrôleurs 404

flux d'entrée 408

indicateurs 399

Properties, classe 736**PropertyChangeEvent, événement** 495**PropertyVetoException, exception** 425**Propriété**ajout *Voir* Bean

autoscrolls 451

booléenne 454

cibles de permission 548

Color 477, 495

d'un thread 24

découvrir 444

descripteur 471

éditeur multiple 491

éditeur *Voir* Editeur

en écriture seulement 454

en lecture seule 454

ensembles 136

fichier de 155

getFileName 454

indexée *Voir* Bean

inverse 481

liée *Voir* Bean

mise à jour valeur 478

modèle de nom 453

nom

local 783

qualifié 783

par défaut 468

paramètres, refuser 429

read/write 454

requestFocusEnabled 451

running 454

setFileName 454

simple *Voir* Bean

String 456

système, fichiers de règles 548

title 495

type 456

URI d'espace de noms 783

valeur

affichage 490

édition 480

extraction 478

lecture 482

mise à jour 482, 495

Propriétés

construire les objets 505

contraintes 459

fichier 660

transitoires 508

Protection, domaine *Voir* Domaine**Prototype** 691

de fonction 682

Proxy 225**Pseudo-code C** 680**Q****QBE** 150**Query, Microsoft** 150**Queue**

de blocage 52

de priorité 104

implémentations 79

liste chaînée 79

synchrone 70

tableau circulaire 79

R**Racine** 314**Ramasse-miettes** 600, 688

de copie 708

Random, classe 586**read, propriété** 454**Recherche binaire** 131**Redondance**

de données 149

élimination 501

ReentrantLock, classe 33**Références**

d'entité 740

de caractères 740

Réflexion 453, 640**Refuser des paramètres de propriétés** 429**REG_BINARY, type** 721**REG_DWORD, type** 721**REG_SZ, type** 721**Registry Windows** *Voir* Base de registres**Règle**

d'encodage de base 593

de mutilation, signature de méthode 697

de sécurité 539, 606

fichier 239, 542

par défaut 608

ELEMENT 755

ReleaseStringUTFChars, fonction 688**ReleaseXxxArrayElements, fonction** 708**Remote, interface** 230**RemoteException, exception** 230, 251, 262**Remplacement**

gestionnaire de sécurité 538

méthode *Voir* Remplacement**Remplissage, base de données** 164**requestFocusEnabled, propriété** 451**Requête**

exécution 168

itération 160

préparation 169

résultat 149

reshape, méthode 425**Résolution de classe** 524**Ressources**

groupes 659

Macintosh 659

trouver l'emplacement 659

Windows 659

Résultats

actualisables 177, 180

de requête 149

défilants 177-178

ResultSet, classe 177-178**ResultSetMetaData, objet** 359

return, instruction 711
Rich Text Format 393
RMI 225, 226
 consignation 247
 vérification d'une mise en
 œuvre 246
Rmic 233
Rmiregistry 236, 269
Rollback 203
Rouge-noir, arbre 99
RowSet 194
RSA 584
RTF 393
run, méthode 12
running, propriété 454

S

sandbox *Voir* Bac à sable
Sauvegarde, points 203
SAX, analyseur 784
**Secondaire, différence entre
 caractères** 647
SecureRandom, classe 586
Sécurité
 applets 523
 certificat *Voir* Certificat
 exception 538
 fichier de règles 239
 gestionnaire *Voir* Gestionnaire
 de sécurité
 Java 539, 711
 méthodes de contrôle 557
 modèle *Voir* Modèle
 modification de fichier de règles
 549
 règles
 Voir aussi Règles 606
security policy *Voir* Règles de
 sécurité
SecurityException 538, 540
SecurityManager
 classe 557
 superclasse 559
SELECT, SQL 150
Sélection
 écouteur de sélection 338
 lignes, colonnes et cellules 385

 listes 298
 modèle 385
 modification 339
Sémaphores, autorisations 70
Séparateurs 414
Séparé, thread 11
Sérialisable, classe 251
Serveur
 activer les objets 263
 CORBA, implémentation 283
 d'application 147
 démarrage 236
 et client 224
 objet de serveur 227
 localisation 234
Service
 de désignation 269
 démarrage de base de registre
 234
 Web 289
Set, méthode 444
setCellRenderer, méthode 309
setFileName
 méthode 454
 propriété 454
SetIntField, méthode 722
setMaximum 400
setMinimum, méthode 400
setObject, méthode 494
**SetObjectArrayElement,
 méthode** 708
setRunning, méthode 454
Sets de hachage liés 110
setSecurityManager, méthode
 538
setSeed, méthode 586
setSelected, méthode 425
setTitle, méthode 495
SetXxxArrayRegion, méthode
 709
SHA1 578
shutdown, méthode 64
sign, méthode 587
signal, méthode 38
signalAll, méthode 37
Signature 577
 applet 148

 authentification 591
 certificat 599
 codée de méthode 700
 condensé de message 585
 d'applet 577
 de code 606
 de programme exécutable 606
 fichiers JAR 606, 607
 message 587
 numérique 577, 583
 règles de mutilation 697
Simple tableau 352
SimpleDateFormat, classe 654
Sites Web
 BER- ASN.1 596
 formats ISO papier 662
 ISO 628
 securingjava 541
 thawte 598
 verisign 592
SOAP 226, 288
 méthode distante 288
Sockets, cibles de permission 548
Source, de code 539
Sous-ensembles 119
Spécification XSL 797-799
sprintf, fonction 690
SQL 143, 148
 commandes, exécuter 159
 CREATE TABLE 152
 DELETE 152
 FROM 150
 INSERT 152
 LIKE 151
 mot clé 150
 SELECT 150
 types avancés 161
 UPDATE 152
 WHERE 151
Stack, classe 137
Standard Template Library *Voir*
 STL
start, commande 237
startElement, méthode 786
**startNameEnumeration,
 fonction** 723
static, mot clé 680

856 Index

- Statiques**
 - champs 697
 - méthodes *Voir* Méthode
 - STL** 78
 - stop, méthode** 50
 - String**
 - classe 646
 - propriété 456
 - type 550, 721
 - Structure**
 - arbre, modification 322
 - base de données 184
 - collections 114
 - document XML 738
 - hiérarchique, XML 737
 - Structured Query Language** *Voir* SQL
 - Stub** 227
 - générations 259
 - objet de stub 229
 - Stylisés, composants de texte** 393
 - subList, méthode** 119
 - substring, méthode** 119
 - Suffixe, Holder** 271
 - Superclasses**
 - héritage 455
 - JComponent 458
 - JPanel 451
 - SecurityManager 559
 - Suppression**
 - écouteur 463
 - éléments 83
 - espaces début de chaîne 633
 - lignes dans le modèle de tableau par défaut 387
 - machine virtuelle 716
 - Surcharger**
 - méthode 527, 556
 - et IDL 272
 - native 681
 - Surveillance, accès fichier** 556
 - suspend, méthode** 50-51
 - SVG** 790
 - Swing** 295
 - classe 458
 - Symétrie, chiffres** 612
 - Synchronisation**
 - blocs 47
 - carte 120
 - communication des threads sans synchronisation 28
 - des accès entre plusieurs threads 32
 - méthodes 251
 - queues 70
 - thread 28
 - vues 120
 - synchronized, mot clé** 42, 43
 - synchronizedMap, classe** 120
 - Synchronizer** 68
 - SynchronousQueue, classe** 70
 - Syntaxe, annotations** 814
 - System, classe** 538, 700
 - System.loadLibrary, méthode** 680, 683
- T**
- Table de hash** 95
 - Tableau** 352
 - accès 709
 - booléen 709
 - circulaire, pour une queue 79
 - compacté 709
 - copie 708, 709
 - de propriétés 483
 - de type primitif 709
 - écouteur de modèle de tableau 366
 - fusion 149
 - Java
 - création 709
 - dans code C 709
 - liste 94
 - modèle 352, 356
 - par défaut, ajouter et supprimer des lignes 387
 - plus grand élément 127
 - simple 352
 - taille 457
 - type 701
 - correspondance Java/C 706
 - Voir aussi* Type tableau 706
 - Tâches, délégation** 458
 - Taille**
 - des colonnes 384
 - tableau 457
 - Tampon Exchanger** 70
 - Terminologie des arbres** 314
 - Tertiaire, différence entre caractères** 647
 - Test, base de données** 156
 - Texte stylisé, composant** 393
 - Thawte** 597, 607, 611
 - this** 692
 - Thread** 6
 - activé 22
 - arrêt 18
 - barrières 69
 - bloqué 21
 - sortir 22
 - collections 58
 - communication sans synchronisation 28
 - constructeur de groupe 25
 - démon 25
 - et processus 6
 - états 24
 - exécutable 21
 - groupes 25
 - interrompre 17
 - mort 23
 - multithread 6
 - nouveau 20
 - partager un accès 28
 - pool 63
 - priorités 24
 - propriétés 24
 - séparé 11
 - synchronisation 28
 - synchroniser des accès 32
 - ThreadDeath** 23
 - ThreadGroup, constructeur** 25
 - Throw, fonction** 711
 - Throwable, type** 711
 - ThrowNew, fonction** 711
 - TimeZone, objet** 645
 - title, propriété** 495
 - toArray** 125
 - Traitement, outil** 807
 - Transactions** 202
 - annulation 203
 - mise à jour 203

Transformation XSL 797

Transtypage 701

TreeSet, classe 98

Tri 129

filtre 364

force 647

fusion 129

strength *Voir* Force de tri

trim 633

Trois niveaux, modèle 147

Type

Boolean 693

Byte 693, 721

Certificate 601

char 687

Class 693

de composant de nom 276

de propriété de bean *Voir* Bean

Double 632, 693

Integer 721

jarray 706

jboolean 709

jchar 686

jclass 692, 701

jfieldID 693

jobject 692, 701

jstring 687, 701

Long 632

NumberFormat 633

Object 693, 721

primitif 708

tableau 709

REG_BINARY 721

REG_DWORD 721

REG_SZ 721

SQL avancé 161

String 550, 721

tableau 701, 706

correspondance (Java/C) 706

hiérarchie d'héritage 706

Throwable 711

void 701

X509Certificate 601

typedef, instruction 685

U

Unicode 270, 646, 657-658, 681, 686

UnsatisfiedLinkError, erreur 684

UnsupportedOperationException, exception 106, 120

update, méthode 587

UPDATE, SQL 152

URI 782

d'espace de noms 783

URL

bases de données 154

HTTP 782

Voir aussi Adresse 539

UTF, code 686

UTF-8, code modifié 658

Utilisateurs

authentifier 563

permissions 563

V

va_list 702

Valeur

afficher pour les listes 308

composant, initialisation 494

fonctionnalité, extraction 468

instructions en hexa 534

propriété

affichage *Voir* Propriété, valeur

édition *Voir* Propriété, valeur

extraction *Voir* Propriété, valeur

lecture *Voir* Propriété, valeur

mise à jour *Voir* Propriété, valeur

renvoyée 685

Valeur/clé, paire 106

Validation automatique 203

Variantes, paramètres régionaux 627

Vector, et ArrayList, synchronisation 95

Vérificateur 532

désactivation 532

Vérification

mise en œuvre RMI 246

permission 540

provenance du code 606

verify, méthode 588

VeriSign 597, 607, 611

Verrou

Countdown 69

implicite 42

mort 48

objet 33

vetoableChange, méthode 430

vetoableChangeListener, interface 430

Virgule flottante

Voir aussi Nombre 632

Visual Basic 454

code 442

construction d'une application 442

contrôle 442

événement 442

fenêtre de propriétés 442

Visuels/non visuels, beans 453

void, type 701

Vues

non-modifiables 119

sous contrôle 121

synchronisées 120

Vues et emballages 117

W

wchar 270

WDSL 288

WeakHashMap, classe 109

Web, services 289

WHERE, SQL 151

Win32RegKey 723

Win32RegKeyNameEnumeration, classe 722, 723

Windows

base de registres 719

ressources 659

Wrapper *Voir* Enveloppe

write, propriété 454

WSDL 226

X

X.50x 593, 597

X509Certificate, type 601

XML 735

analyser un document 741

attributs 739

DTD 738

et HTML 737

fichiers de propriétés 736

générer des documents 789

grille 763-764

structure

d'un document 738

hiérarchique 737

XML Schema 752, 760

XMLDecoder 502

XMLEncoder 502

XPath, localiser des informations
776

XSL

spécification 797-799

transformations 797