Timothy Fisher

LE GUIDE DE SURVIE

Java®

L'ESSENTIEL DU CODE ET DES COMMANDES

Java

Timothy R. Fisher

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

Réalisation PAO : Léa B

Auteur : Timothy R. Fisher

ISBN : 978-2-7440-4004-7

Titre original : *Java® Phrasebook*

Traduit de l'américain par :
Patrick Fabre

ISBN original : 0-672-32907-7

Sams Publishing
800 East 96th,
Indianapolis, Indiana 46240 USA

Table des matières

Au sujet de l'auteur

Timothy Fisher est un professionnel du développement de logiciels Java depuis 1997. Il a revêtu de nombreuses casquettes, dont celle de développeur, de chef d'équipe et d'architecte en chef. Il est actuellement consultant pour l'entreprise Compuware Corporation à Détroit dans le Michigan. Il aime écrire sur des sujets techniques et a contribué aux deux ouvrages *Java Developer's Journal* et *XML Journal*.

Tim est également passionné par l'éducation et l'utilisation des technologies Internet avancées dans ce domaine. Pour le contacter et consulter son blog (en anglais), rendez-vous à l'adresse **www.timothyfisher.com**.

Introduction

En début d'année, un éditeur de Pearson m'a demandé d'écrire ce *Guide de survie* consacré au Java, opus d'une collection regroupant un certain nombre d'autres ouvrages, dont Christian Wenz avait écrit le premier consacré au PHP (première partie du *Guide de survie PHP et MySQL*). L'idée de la collection *Guide de survie* est tirée des guides de conversation pour le tourisme dans les pays étrangers qui proposent des listes de phrases pour s'exprimer dans une autre langue. Ces manuels sont très utiles pour ceux qui ne connaissent pas la langue locale. Le principe des ouvrages de la collection *Guide de survie* est analogue. Ils montrent au lecteur comment réaliser des tâches courantes dans le cadre d'une technologie particulière.

Le but de ce *Guide de survie* est de fournir une liste d'exemples de code couramment utilisés en programmation Java. Ce livre doit être utile à la fois au programmeur Java confirmé et à celui qui débute avec ce langage. S'il peut être lu de bout en bout afin d'acquérir une vue d'ensemble du langage Java, il est avant tout conçu comme un ouvrage de référence qui peut être consulté à la demande lorsque le programmeur doit savoir comment réaliser une tâche courante en Java. Vous pouvez aussi explorer ce livre afin de découvrir des fonctionnalités et des techniques Java que vous n'avez pas encore maîtrisées.

Ce livre n'est pas un manuel d'apprentissage ou d'introduction au Java ni une référence complète de ce langage. Il existe bien d'autres classes et API que celles présentées ici. D'excellents livres ont déjà été publiés qui vous permettront d'apprendre le Java ou serviront de référence en abordant toutes les fonctionnalités possibles et imaginables. Si votre but est d'acquérir une compréhension très vaste d'une technologie spécifique, il est préférable de consulter un livre adapté.

La plupart des exemples présentés dans ce livre n'incluent pas de code de gestion des erreurs. Bon nombre de ces fragments de code sont cependant susceptibles de lever des exceptions qu'il vous faudra impérativement gérer dans vos propres applications. Le code de gestion des erreurs et des exceptions est volontairement omis ici, afin que le lecteur puisse se concentrer spécifiquement sur la notion illustrée par l'exemple, sans être distrait par d'autres considérations. Si les exemples avaient inclus l'ensemble du code de gestion des exceptions, ils n'auraient pour la plupart pas pu prendre la forme compacte et synthétique qui est la leur et vous n'auriez pas mieux compris les notions abordées. La JavaDoc du JDK Java est une excellente source d'informations pour retrouver les exceptions qui peuvent être levées par les méthodes contenues dans les classes Java rencontrées dans ce livre. Pour la consulter, rendez-vous à l'adresse **http://java.sun.com/j2se/1.5.0/docs/api/**.

Les exemples de ce livre doivent être indépendants du système d'exploitation. Le mot d'ordre de la plate-forme Java ("programmé une fois, exécuté partout") doit s'appliquer à tous les exemples contenus dans ce livre. Le code a été testé sous le JDK 1.5 aussi appelé Java 5.0.

La plupart des exemples fonctionnent aussi sous les versions précédentes du JDK, sauf mention spéciale à ce sujet.

Tous les exemples ont été testés et doivent être exempts d'erreurs. Je souhaite pour ma part que le livre ne contienne pas la moindre erreur, mais il faut évidemment admettre qu'aucun livre technique ne peut par définition y prétendre. Toutes les erreurs qui pourraient être trouvées dans l'ouvrage seront signalées sur le site **www.samspublishing.com**.

En rédigeant ce livre, j'ai tenté de trouver les exemples les plus utiles tout en m'astreignant à l'exigence de concision de la collection *Guide de survie*. Il est immanquable qu'à un moment ou un autre, vous rechercherez un exemple qui ne figurera pas dans ce livre. Si vous estimez qu'un exemple important manque, signalez-le moi. Si vous pensez à l'inverse que d'autres exemples du livre ne sont pas si utiles, indiquez-le moi également. En tant qu'auteur, j'apprécie toujours de connaître le sentiment des lecteurs. A l'avenir, il est possible qu'une seconde édition du livre voit le jour. Vous pouvez me contacter en consultant mon site Web à l'adresse **www.timothyfisher.com**.

1

Les bases

Ce chapitre présente les premiers exemples avec lesquels vous aurez besoin de vous familiariser pour démarrer un développement Java. Ils sont en fait requis pour réaliser quelque action que ce soit en Java. Vous devez pouvoir compiler et exécuter votre code Java et comprendre les chemins de classe Java. A la différence d'autres langages comme le PHP ou le Basic, le code source Java doit être compilé sous une forme appelée "code-octet" (*bytecode*) avant de pouvoir être exécuté. Le compilateur place le code-octet dans des fichiers de classe Java. Tout programmeur Java doit donc comprendre comment compiler son code source en fichiers de classe et savoir exécuter ces fichiers de classe. La compréhension des chemins de classe Java est importante à la fois pour compiler et pour exécuter le code Java. Nous commencerons donc par ces premiers exemples.

Il est aujourd'hui courant de travailler au développement Java dans un EDI (environnement de développement intégré) comme le projet libre Eclipse (**http://www.eclipse.org**). Pour ce chapitre, nous considérerons que vous réaliserez vos tâches en ligne de commande.

Si l'essentiel de votre développement peut parfaitement se faire avec un EDI, tout développeur se doit cependant d'être familiarisé avec la configuration et la réalisation de ces tâches en dehors d'un EDI. La procédure propre aux tâches effectuées dans un EDI varie selon l'EDI : il est donc préférable dans ce cas de consulter le manuel de l'EDI concerné pour obtenir de l'aide à ce sujet.

Pour exécuter les instructions contenues dans ce chapitre, vous devez obtenir une distribution Java auprès de Sun. Sun diffuse la technologie Java sous plusieurs formes. Les distributions Java les plus courantes sont le Java Standard Edition (SE), le Java Enterprise Edition (EE) et le Java Micro Edition (ME). Pour suivre tous les exemples de ce livre, vous n'aurez besoin que du paquetage Java SE. Java EE contient des fonctionnalités supplémentaires permettant de développer des applications d'entreprise. Java ME est destiné au développement d'applications pour les périphériques tels que les téléphones cellulaires et les assistants personnels. Tous ces paquetages peuvent être téléchargés depuis le site Web de Sun à l'adresse **http://java.sun.com**. A l'heure où ces lignes sont écrites, J2SE 5.0 est la version la plus récente du Java SE. A moins que vous n'ayez une raison particulière d'utiliser une version antérieure, utilisez donc cette version avec ce livre. Vous trouverez deux paquetages à télécharger dans le J2SE 5.0 : le JDK et le JRE. Le JDK est le kit de développement Java. Il est nécessaire pour développer des applications Java. Le JRE (*Java runtime edition*) ne permet que d'exécuter des applications Java et non d'en développer. Pour ce livre, vous aurez donc besoin de la distribution JDK du paquetage J2SE 5.0.

Info

J2SE 5.0 et JDK 5.0 sont souvent aussi mentionnés sous la référence JDK 1.5. Sun a décidé de changer officiellement le nom de la version 1.5 en l'appelant 5.0.

Pour obtenir de l'aide lors de l'installation de la version la plus récente du Java J2SE JDK, voir **http://java.sun.com/j2se/1.5.0/install.html**.

Compiler un programme Java

```
javac HelloWorld.java
```

Cet exemple compile le fichier source `HelloWorld.java` en code-octet. Le code-octet est la représentation Java indépendante de toute plate-forme des instructions d'un programme. La sortie sera placée dans le fichier `HelloWorld.class`.

L'exécutable `javac` est inclus dans la distribution Java JDK. Il est uti'lisé pour compiler les fichiers source Java que vous écrivez dans des fichiers de classe Java. Un fichier de classe Java est une représentation en code-octet de la source Java compilée. Pour plus d'informations sur la commande `javac`, consultez la documentation du JDK. De nombreuses options peuvent être utilisées avec `javac` qui ne sont pas traitées dans ce livre.

Pour la plupart de vos projets de programmation, à l'exception des petits programmes très simples, vous utiliserez sans doute un EDI ou un outil comme Ant d'Apache pour réaliser votre compilation. Si vous compilez autre

chose qu'un très petit projet avec des fichiers source minimaux, il est vivement conseillé de vous familiariser avec Ant. Si vous connaissez l'outil de compilation Make utilisé par les programmeurs C, vous comprendrez l'importance d'Ant. Ant est en quelque sorte l'équivalent de Make pour Java. Il permet de créer un script de compilation pour spécifier les détails de la compilation d'une application complexe puis de générer automatiquement l'application complète à l'aide d'une seule commande. Pour plus d'informations sur Ant et pour télécharger le programme, rendez-vous à l'adresse **http://ant.apache.org**.

Exécuter un programme Java

```
javac HelloWorld.java // compilation du fichier source
java HelloWorld // exécution du code-octet
```

Dans cet exemple, nous utilisons d'abord le compilateur `javac` pour compiler notre source Java dans un fichier `HelloWorld.class`. Ensuite, nous exécutons le programme HelloWorld en utilisant la commande `java` à laquelle nous passons le nom de la classe compilée, `HelloWorld`. Notez que l'extension `.class` n'est pas incluse dans le nom qui est passé à la commande `java`.

L'exécutable `java` est inclus avec la distribution Java JDK ou la distribution Java JRE. Il est utilisé pour exécuter les fichiers de classe Java compilés. Il fait office d'interpréteur et compile en temps réel le code-octet en code natif exécutable sur la plate-forme utilisée. L'exécutable `java` est un élément de Java dépendant de la plate-forme d'exécution. Chaque plate-forme qui supporte Java possède ainsi son propre exécutable `java` compilé spécifiquement pour elle. Cet élément est aussi appelé la *machine virtuelle*.

Définir le chemin de classe

```
set CLASSPATH = /utilisateur/projets/classes
java -classpath =
    ➥CLASSPATH%;classes/classfile.class;libs/stuff.jar
```

Le chemin de classe est utilisé par l'exécutable `java` et le compilateur `java` pour trouver les fichiers de classe compilés et toutes les bibliothèques empaquetées sous forme de fichiers JAR requis pour exécuter ou compiler un programme. Les fichiers JAR sont le moyen standard d'empaqueter des bibliothèques dans une ressource prenant la forme d'un fichier unique. L'exemple précédent montre comment le chemin de classe peut être défini lors de l'exécution d'un programme Java en ligne de commande. Par défaut, le chemin de classe est obtenu depuis la variable d'environnement système `CLASSPATH`. Dans cet exemple, une classe spécifique appelée `classfile.class` et située dans le dossier `classes` est ajoutée au chemin de classe défini par la variable d'environnement. Une bibliothèque appelée `stuff.jar` située dans le répertoire `libs` est également ajoutée au chemin de classe. Si la variable d'environnement `CLASSPATH` n'est pas définie et que l'option `–classpath` n'est pas utilisée, le chemin de classe par défaut correspond au répertoire courant. Si le chemin de classe est défini avec l'une de ces options, le répertoire courant n'est pas automatiquement inclus dans le chemin de classe. Il s'agit là d'une source fréquente de problèmes. Si vous définissez un chemin de classe, vous devez explicitement rajouter le répertoire courant. Vous pouvez toutefois ajouter le répertoire courant au chemin de classe en le spécifiant par "." dans le chemin de classe.

Attention

Notez que si toutes les classes se trouvent dans un répertoire inclus dans le chemin de classe, il faut néanmoins que les fichiers JAR soient explicitement inclus dans le chemin de classe pour être trouvés. Il ne suffit pas d'inclure le répertoire dans lequel ils résident à l'intérieur du chemin de classe.

Les problèmes liés aux chemins de classe sont très courants chez les programmeurs novices comme chez les programmeurs expérimentés et peuvent souvent être très agaçants à résoudre. Si vous prenez le temps de bien comprendre le fonctionnement des chemins de classe et de bien savoir les définir, vous devriez pouvoir éviter ces problèmes dans vos applications. Pour plus d'informations sur la configuration et l'utilisation des chemins de classe, consultez la page **http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/classpath.html**.

2

Interagir avec l'environnement

Ce chapitre regroupe l'ensemble des exemples qui vous permettront d'interagir avec l'environnement d'exécution sur lequel votre application s'exécute. Plusieurs d'entre eux utilisent l'objet Java `System`, un objet Java central destiné à interagir avec l'environnement qui entoure votre application Java. Il faut être très prudent lorsque vous utilisez cet objet et plus généralement lorsque vous interagissez avec l'environnement, car vous risquez par inadvertance de créer du code dépendant de votre plate-forme. L'objet `System` interagit avec l'environnement et ce dernier est bien sûr propre à la plate-forme sur laquelle vous travaillez. Les effets de l'utilisation d'une méthode ou d'une propriété de `System` sur une plate-forme peuvent ainsi ne pas être les mêmes sur l'ensemble des autres plates-formes.

Obtenir des variables d'environnement

```
String envPath = System.getenv("PATH");
```

Cet exemple montre comment récupérer une variable d'environnement avec la méthode `System.getenv()`. Cette méthode a été déconseillée dans les versions du JDK comprises entre la version 1.2 et la version 1.4. Avec le JDK 1.5, Sun a pris une mesure exceptionnelle en revenant sur sa décision et en réhabilitant cette méthode. Si vous utilisez une version du JDK pour laquelle cette méthode est déconseillée, vous verrez des avertissements apparaître au moment de la compilation lorsque vous tentez d'utiliser cette méthode. Les méthodes déconseillées ne doivent pas être utilisées dans les nouveaux projets de développement mais restent généralement prises en charge pour des raisons de compatibilité arrière. Il n'existe pas de garantie que les méthodes déconseillées continuent d'être prises en charge dans les versions futures du JDK, mais dans le cas précis de cette méthode, il se trouve que la version la plus récente du JDK l'a réhabilitée : vous pouvez donc raisonnablement supposer qu'elle continuera d'être prise en charge.

En général, on considère qu'il est de mauvais usage d'utiliser des variables d'environnement dans les applications Java. Celles-ci dépendent en effet de la plate-forme, or le Java a justement pour vocation d'être indépendant de toute plate-forme. Certaines plates-formes Java (notamment Macintosh) ne proposent d'ailleurs pas de variable d'environnement. Votre code ne se comportera donc pas comme prévu dans ces environnements.

L'exemple suivant montre comment obtenir et définir des propriétés système. Cette approche est préférable à celle qui consiste à utiliser des variables d'environnement.

Définir et obtenir des propriétés système

```
System.setProperty("timezone", "EasternStandardTime");
String zone = System.getProperty("timezone");
```

Les propriétés système sont des paires clé/valeur externes à votre application Java. L'objet Java `System` propose un mécanisme permettant de lire les noms et les valeurs de ces propriétés système externes depuis votre application Java. L'exemple précédent montre comment définir et lire une propriété système à l'aide de l'objet Java `System`. Vous pouvez aussi récupérer toutes les propriétés système dans un objet de propriétés à l'aide de l'instruction suivante :

```
Properties systemProps = System.getProperties();
```

Une autre méthode permet également de récupérer les noms des propriétés système. Le fragment de code suivant indique comment récupérer tous les noms des propriétés système puis récupérer chaque propriété avec son nom :

```
Properties props = System.getProperties();
Enumeration propertyNames = props.propertyNames();
String key = "";
while (propertyNames.hasMoreElements()) {
     key = (String) propertyNames.nextElement();
     System.out.println(key + "=" +
              ➥props.getProperty(key));
}
```

Parser des arguments en ligne de commande

```
java my_program arg1 arg2 arg3

public static void main(String[] args) {
     String arg1 = args[0];
     String arg2 = args[1];
     String arg3 = args[2];
}
```

Dans cet exemple, nous stockons les valeurs de trois arguments en ligne de commande dans trois variables `String` séparées, `arg1`, `arg2` et `arg3`.

Toutes les classes Java peuvent inclure une méthode `main()` exécutable en ligne de commande. La méthode `main()` accepte un tableau `String` d'arguments en ligne de commande. Les arguments sont contenus dans le tableau dans l'ordre où ils sont entrés dans la ligne de commande. Pour les récupérer, il vous suffit donc d'extraire les éléments du tableau des arguments passé à la méthode `main()`.

Si votre application utilise un grand nombre d'arguments en ligne de commande, il peut être utile de passer du temps à concevoir un parseur d'arguments en ligne de commande personnalisé pour comprendre et gérer les différents types d'arguments en ligne de commande, comme les paramètres à un caractère, les paramètres avec des tirets (-), les paramètres immédiatement suivis par un autre paramètre lié, etc.

Info

De nombreux exemples de processeurs d'arguments en ligne de commande peuvent être trouvés sur Internet afin de gagner du temps. Deux bonnes bibliothèques peuvent être utilisées pour démarrer :

http://jargs.sourceforge.net

https://args4j.dev.java.net/

Ces deux bibliothèques peuvent analyser les arguments d'une ligne de commande complexe à l'aide d'une interface relativement simple.

3

Manipuler des chaînes

En programmation, quel que soit le langage utilisé, une grande partie des opérations réalisées concerne la manipulation des chaînes. A l'exception des données numériques, presque toutes les données sont gérées sous forme de chaînes. Les données numériques sont d'ailleurs parfois elles-mêmes manipulées sous cette forme. On s'imagine difficilement comment il serait possible d'écrire un programme complet sans utiliser la moindre chaîne.

Les exemples de ce chapitre présentent des tâches courantes liées à la manipulation des chaînes. Le langage Java propose une excellente prise en charge des chaînes. A la différence du langage C, les chaînes sont des types prédéfinis dans le langage Java. Celui-ci contient une classe `String` spécifiquement destinée à contenir les données de chaîne. En Java, les chaînes ne doivent pas être considérées à la manière de tableaux de caractères comme elles le sont en C.

Chaque fois que vous souhaitez représenter une chaîne en Java, vous devez utiliser la classe `String` et non un tableau.

La classe `String` possède une propriété importante : une fois créée, la chaîne est immuable. Les objets Java `String` ne peuvent donc plus être changés après qu'ils sont créés. Vous pouvez attribuer le nom donné à une chaîne à un autre objet `String`, mais vous ne pouvez pas changer le contenu de la chaîne. Vous ne trouverez donc aucune méthode `set` dans la classe `String`. Si vous souhaitez créer une chaîne à laquelle des données peuvent être ajoutées (par exemple, dans une routine qui construit progressivement une chaîne), vous devez utiliser la classe `StringBuilder` dans le JDK 1.5 ou la classe `StringBuffer` dans les versions antérieures du Java, et non la classe `String`. Les classes `StringBuilder` et `StringBuffer` sont muables : leur contenu peut être modifié. Il est très courant de construire des chaînes en utilisant la classe `StringBuilder` ou `StringBuffer` et de passer ou stocker des chaînes en utilisant la classe `String`.

Comparer des chaînes

```
boolean result = str1.equals(str2);
boolean result2 = str1.equalsIgnoreCase(str2);
```

La valeur de `result` et `result2` doit être `true` si les chaînes ont le même contenu. Si leur contenu est différent, `result` et `result2` valent `false`. La première méthode, `equals()`, tient compte de la casse des caractères dans les chaînes. La seconde, `equalsIgnoreCase()`, ignore la casse des caractères et retourne `true` si le contenu est identique indépendamment de la casse.

Les opérations de comparaison de chaînes sont une source courante de bogues pour les programmeurs débutants en Java. Ces derniers s'efforcent souvent de comparer leurs chaînes avec l'opérateur de comparaison ==. Or ce dernier compare les références d'objet et non le contenu des objets. Deux objets chaîne qui contiennent les mêmes données de chaîne mais correspondent à des instances d'objet physiquement distinctes ne sont dès lors pas considérés comme égaux selon cet opérateur.

La méthode `equals()` de la classe `String` fait porter la comparaison sur le contenu de la chaîne et non sur sa référence d'objet. En général, il s'agit de la méthode de comparaison souhaitée pour les comparaisons de chaînes. Voyez l'exemple suivant :

```
String name1 = new String("Timmy");
String name2 = new String("Timmy");
if (name1 == name2) {
     System.out.println("The strings are equal.");
}
else {
     System.out.println("The strings are not equal.");
}
```

La sortie obtenue après l'exécution de ces instructions est la suivante :

```
The strings are not equal.
```

A présent, utilisez la méthode `equals()` et observez le résultat :

```
String name1 = new String("Timmy");
String name2 = new String("Timmy");
if (name1.equals(name2)) {
     System.out.println("The strings are equal.");
```

```
}
else {
     System.out.println("The strings are not equal.");
}
```

La sortie obtenue après l'exécution de ces instructions est la suivante :

```
The strings are equal.
```

La méthode `compareTo()` est une autre méthode apparentée de la classe `String`. Elle compare alphabétiquement deux chaînes en retournant une valeur entière : positive, négative ou égale à 0. La valeur 0 n'est retournée que si la méthode `equals()` est évaluée à `true` pour les deux chaînes. Une valeur négative est retournée si la chaîne sur laquelle la méthode est appelée précède dans l'ordre alphabétique celle qui est passée en paramètre à la méthode. Une valeur positive est retournée si la chaîne sur laquelle la méthode est appelée suit dans l'ordre alphabétique celle qui est passée en paramètre. En fait, la comparaison s'effectue en fonction de la valeur Unicode de chaque caractère dans les chaînes comparées. La méthode `compareTo()` possède également une méthode `compareToIgnoreCase()` correspondante qui opère de la même manière mais en ignorant la casse des caractères. Considérez l'exemple suivant :

```
String name1="Camden";
String name2="Kerry";
int result = name1.compareTo(name2);
if (result == 0) {
      System.out.println("The names are equal.");
}
```

```
else if (result > 0) {
System.out.println(
"name2 comes before name1 alphabetically.");
}
else if (result < 0) {
     System.out.println(
          "name1 comes before name2 alphabetically.");
}
```

La sortie de ce code est :

```
name1 comes before name2 alphabetically.
```

Rechercher et récupérer des sous-chaînes

```
int result = string1.indexOf(string2);
int result = string1.indexOf(string2, 5);
```

Dans la première méthode, la valeur de `result` contient l'index de la première occurrence de `string2` à l'intérieur de `string1`. Si `string2` n'est pas contenu dans `string1`, la valeur `-1` est retournée.

Dans la seconde méthode, la valeur de `result` contient l'index de la première occurrence de `string2` à l'intérieur de `string1` qui intervient après le cinquième caractère dans `string1`. Le second paramètre peut être n'importe quel entier valide supérieur à 0. Si la valeur est supérieure à la longueur de `string1`, la valeur `-1` est retournée.

Outre rechercher une sous-chaîne dans une chaîne, il peut arriver que vous sachiez où se trouve la sous-chaîne et que vous souhaitiez simplement l'atteindre. La méthode

`substring()` de la chaîne vous permet de l'atteindre. Cette méthode est surchargée, ce qui signifie qu'il existe plusieurs moyens de l'appeler. Le premier consiste à lui passer simplement un index de départ. Cette méthode retourne une sous-chaîne qui commence à l'index de départ et s'étend jusqu'à la fin de la chaîne. L'autre moyen d'utiliser `substring()` consiste à l'appeler avec deux paramètres – un index de départ et un index de fin.

```
String string1 = "My address is 555 Big Tree Lane";
String address = string1.substring(14);
System.out.println(address);
```

Ce code produit la sortie suivante :

```
555 Big Tree Lane
```

Le premier caractère 5 se trouve à la position 14 de la chaîne. Il s'agit donc du début de la sous-chaîne. Notez que les chaînes sont toujours indexées en commençant à 0 et que le dernier caractère se trouve à l'emplacement (*longueur de la chaîne*) - 1.

Traiter une chaîne caractère par caractère

```
for (int index = 0; index < string1.length();
index++) {
      char aChar = string1.charAt(index);
}
```

La méthode `charAt()`permet d'obtenir un unique caractère de la chaîne à l'index spécifié. Les caractères sont indexés en commençant à 0, de 0 à (*longueur de la chaîne*) - 1.

Cet exemple parcourt en boucle chaque caractère contenu dans `string1`. Il est aussi possible de procéder en utilisant la classe `StringReader`, comme ceci :

```
StringReader reader = new StringReader(string1);
int singleChar = reader.read();
```

Avec ce mécanisme, la méthode `read()` de la classe `StringReader` retourne un caractère à la fois, sous forme d'entier. A chaque fois que la méthode `read()` est appelée, le caractère suivant de la chaîne est retourné.

Renverser une chaîne par caractère

```
String letters = "ABCDEF";
StringBuffer lettersBuff = new StringBuffer(letters);
String lettersRev = lettersBuff.reverse().toString();
```

La classe `StringBuffer` contient une méthode `reverse()` qui retourne un `StringBuffer` contenant les caractères du `StringBuffer` original, mais inversés. Un objet `StringBuffer` peut aisément être converti en objet `String` à l'aide de la méthode `toString()` de l'objet `StringBuffer`. En utilisant temporairement un objet `StringBuffer`, vous pouvez ainsi produire une seconde chaîne avec les caractères d'une chaîne d'origine, en ordre inversé.

Si vous utilisez le JDK 1.5, vous pouvez utiliser la classe `StringBuilder` au lieu de la classe `StringBuffer`.`StringBuilder` possède une API compatible avec la classe `StringBuffer`. Elle propose de meilleures performances, mais ses méthodes ne sont pas synchronisées. Elle n'est donc pas *thread-safe*. En cas de multithreading, vous devez continuer à utiliser la classe `StringBuffer`.

Renverser une chaîne par mot

```
String test = "Reverse this string";
Stack stack = new Stack();
StringTokenizer strTok = new StringTokenizer(test);

while(strTok.hasMoreTokens()) {
     stack.push(strTok.nextElement());
}

StringBuffer revStr = new StringBuffer();
while(!stack.empty()) {
     revStr.append(stack.pop());
     revStr.append(" ");
}
System.out.println("Original string: " + test);
System.out.println("\nReversed string: " + revStr);
```

La sortie de ce fragment de code est la suivante :

```
Original string: Reverse this string

Reversed string: string this Reverse
```

Comme vous pouvez le voir, le renversement d'une chaîne par mot est plus complexe que le renversement d'une chaîne par caractère. C'est qu'il existe un support intégré dans Java pour le renversement par caractère, et non pour le renversement par mot. Pour réaliser cette dernière tâche, nous utilisons les classes `StringTokenizer` et `Stack`. Avec `StringTokenizer`, nous parsons chaque mot de la chaîne et le poussons dans notre pile. Une fois la chaîne entière traitée, nous parcourons la pile en boucle en dépilant chaque mot et en l'ajoutant à un `StringBuffer` qui stocke la chaîne inversée.

Dans la pile, le dernier élément entré est par principe le premier sorti – une propriété baptisée LIFO (*last in, first out*) que nous exploitons pour effectuer l'inversion.

Consultez l'exemple traité dans la section "Parser une chaîne séparée par des virgules" de ce chapitre pour d'autres utilisations de la classe `StringTokenizer`.

Info

Nous n'en traiterons pas ici, mais un nouvel ajout du JDK 1.5 peut vous intéresser : la classe `Scanner`. Cette classe est un analyseur de texte élémentaire permettant de parser des types primitifs et des chaînes à l'aide d'expressions régulières.

Convertir une chaîne en majuscules ou en minuscules

```
String string = "Contains some Upper and some Lower.";
String string2 = string.toUpperCase();
String string3 = string.toLowerCase();
```

Ces deux méthodes transforment une chaîne en majuscules ou en minuscules uniquement. Elles retournent toutes deux le résultat transformé. Ces méthodes n'affectent pas la chaîne d'origine. Celle-ci reste intacte avec une casse mixte.

Ces méthodes peuvent notamment être utiles lors du stockage d'informations dans une base de données, par exemple si vous souhaitez stocker des valeurs de champ en majuscules ou en minuscules uniquement. Grâce à ces méthodes, l'opération de conversion est un jeu d'enfant.

La conversion de la casse est également utile pour la gestion des interfaces d'authentification des utilisateurs. Le champ d'ID de l'utilisateur est normalement considéré comme étant un champ qui ne doit pas tenir compte de la casse, alors que le champ de mot de passe en tient compte. Lors de la comparaison de l'ID utilisateur, vous pouvez ainsi convertir l'ID en majuscules ou en minuscules puis le comparer à une valeur stockée dans la casse appropriée. Vous pouvez aussi utiliser la méthode `equalsIgnoreCase()` de la classe `String`, qui réalise une comparaison sans tenir compte de la casse.

Supprimer les espaces au début et à la fin d'une chaîne

```
String result = str.trim();
```

La méthode `trim()` supprime les espaces de début et de fin d'une chaîne et retourne le résultat. La chaîne originale reste inchangée. S'il n'y a pas d'espace de début ou de fin à supprimer, la chaîne d'origine est retournée. Les espaces et les caractères de tabulation sont supprimés.

Cette méthode est très utile lorsqu'il s'agit de comparer des entrées saisies par l'utilisateur avec des données existantes. Bien des programmeurs se sont creusé la tête de nombreuses heures en se demandant pourquoi les données saisies n'étaient pas identiques à la chaîne stockée et se sont finalement aperçu que la différence ne tenait qu'à un simple espace blanc à la fin de la chaîne. La suppression des espaces avant les comparaisons élimine entièrement ce problème.

Parser une chaîne séparée par des virgules

```
String str = "tim,kerry,timmy,camden";
String[] results = str.split(",");
```

La méthode `split()` de la classe `String` accepte une expression régulière comme unique paramètre et retourne un tableau d'objets `String` décomposé selon les règles de l'expression régulière passée. Le parsing des chaînes séparées par des virgules devient ainsi un jeu d'enfant. Dans cet exemple, nous passons simplement une virgule à la méthode `split()` et obtenons un tableau de chaînes contenant les données séparées par des virgules. Le tableau de résultat de notre exemple contient ainsi les données suivantes :

```
results[0] = tim
results[1] = kerry
results[2] = timmy
results[3] = camden
```

La classe `StringTokenizer` est elle aussi utile pour décomposer des chaînes. L'exemple précédent peut être repris avec cette classe au lieu de la méthode `split()` :

```
String str = "tim,kerry,timmy,Camden";
StringTokenizer st = new StringTokenizer(str, ",");
while (st.hasMoreTokens()) {
      System.out.println(st.nextToken());
}
```

Cet exemple de code imprime chacun des noms contenus dans la chaîne d'origine (`str`) sur une ligne séparée, comme ceci :

```
tim
kerry
timmy
camden
```

Notez que les virgules sont supprimées (elles ne sont pas imprimées). La classe `StringTokenizer` peut être construite avec un, deux ou trois paramètres. Lorsqu'elle est appelée avec un seul paramètre, le paramètre correspond à la chaîne à diviser. Dans ce cas, le délimiteur utilisé correspond aux limites naturelles de mot par défaut. Le *tokenizer* utilise ainsi le jeu de délimiteurs `" \t\n\r\f"` (le caractère d'espace, le caractère de tabulation, le caractère de nouvelle ligne, le caractère de retour chariot et le caractère d'avancement de page).

Le deuxième moyen de construire un objet `StringTokenizer` consiste à passer deux paramètres au constructeur. Le premier paramètre correspond alors à la chaîne à décomposer et le second à la chaîne contenant les délimiteurs en fonction desquels la chaîne doit être divisée. Ce paramètre vient remplacer les délimiteurs par défaut.

Pour finir, vous pouvez passer un troisième argument au constructeur `StringTokenizer` qui indique si les délimiteurs doivent être retournés sous forme de jetons ou supprimés. Ce paramètre est booléen. Si vous passez la valeur `true`, les délimiteurs sont retournés sous forme de jetons. Par défaut, le paramètre vaut `false`. Il supprime les délimiteurs et ne les traite pas comme des jetons.

Examinez aussi les exemples du Chapitre 6. Avec l'ajout du support des expressions régulières en Java proposé par le JDK 1.4, il devient souvent possible d'utiliser des expressions régulières au lieu de la classe `StringTokenizer`. La documentation JavaDoc officielle stipule que la classe `StringTokenizer` est une classe héritée dont l'usage doit être déconseillé dans le nouveau code. Dans la mesure du possible, utilisez donc la méthode `split()` de la classe `String` ou le paquetage Java pour les expressions régulières.

4

Travailler avec des structures de données

On appelle "structure de données" un dispositif servant à organiser les données utilisées par un programme. Chaque fois que vous travaillez avec des groupes d'éléments de données similaires, il est judicieux d'utiliser une structure de données. Le Java offre une excellente gestion des différents types de structures de données, dont les tableaux, les listes, les dictionnaires et les ensembles. La plupart des classes Java servant à travailler avec les structures de données sont livrées dans le *framework* Collections, une architecture unifiée pour la représentation et la manipulation de collections ou de structures de données. Les classes de structure de données les plus couramment utilisées sont `ArrayList` et `HashMap`. La plupart des exemples de ce chapitre s'y réfèrent.

L'expression "structure de données" peut s'appliquer à la manière dont les données sont ordonnées dans un fichier ou une base de données tout autant qu'en mémoire. Tous les exemples de ce chapitre traitent des structures de données en mémoire.

Info

Sun met à disposition un document (en anglais) qui offre une bonne vue d'ensemble du framework Collections et propose des didacticiels sur l'utilisation des différentes classes. Pour consulter ce document, rendez-vous à l'adresse suivante : http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html.

Redimensionner un tableau

```
// Utiliser un ArrayList
List myArray = new ArrayList();
```

En Java, les tableaux d'objets normaux ou de types primitifs ne peuvent pas être redimensionnés de manière dynamique. Si vous souhaitez qu'un tableau soit agrandi par rapport à sa déclaration d'origine, vous devez déclarer un nouveau tableau plus grand, puis copier le contenu du premier tableau dans le nouveau. La procédure peut prendre la forme suivante :

```
int[] tmp = new int[myArray.length + 10];
System.arraycopy(myArray, 0, tmp, 0, myArray.length);
myArray = tmp;
```

Dans cet exemple, nous souhaitons agrandir la taille d'un tableau d'entiers appelé `myArray` afin de lui ajouter dix éléments. Nous créons donc un nouveau tableau appelé `tmp` et l'initialisons en lui attribuant la longueur de `myArray + 10`. Nous utilisons ensuite la méthode `System.arrayCopy()` pour copier le contenu de `myArray` dans le tableau `tmp`. Pour finir, nous positionnons `myArray` de manière à ce qu'il pointe sur le tableau `tmp` nouvellement créé.

En général, la meilleure solution pour ce type de problème consiste à utiliser un objet `ArrayList` au lieu d'un tableau d'objets classique. L'objet `ArrayList` peut contenir

n'importe quel type d'objet. Son principal intérêt tient à ce qu'il se redimensionne automatiquement selon les besoins. Lorsque vous utilisez un `ArrayList`, vous n'avez plus à vous soucier de la taille de votre tableau en vous demandant si vous risquez de manquer de place. L'implémentation `ArrayList` est en outre bien plus efficace que la méthode précédente qui consiste à copier le tableau à redimensionner dans un nouveau tableau. L'objet `ArrayList` fait partie du paquetage `java.util`.

Parcourir une collection en boucle

```
// Pour un ensemble ou une liste
// collection est l'objet set ou list
for (Iterator it= collection.iterator(); it.hasNext(); )
{
  Object element = it.next();
}

// Pour les clés d'un dictionnaire
for (Iterator it = map.keySet().iterator(); it.hasNext();
) {
  Object key = it.next();
}

// Pour les valeurs d'un dictionnaire
for (Iterator it = map.values().iterator(); it.hasNext();
) {
  Object value = it.next();
}

// Pour les clés et les valeurs d'un dictionnaire
for (Iterator it = map.entrySet().iterator();
it.hasNext(); ) {
  Map.Entry entry = (Map.Entry)it.next();
  Object key = entry.getKey();
  Object value = entry.getValue();
}
```

Le paquetage `java.util` contient une classe `Iterator` qui facilite le parcours en boucle des collections. Pour parcourir en boucle un objet collection, vous devez d'abord obtenir un objet `Iterator` en appelant la méthode `iterator()` de l'objet `collection`. Lorsque vous avez l'objet `Iterator`, il ne reste plus qu'à le parcourir pas à pas avec la méthode `next()`. Cette méthode retourne l'élément suivant de la collection.

La méthode `next()` retourne un type `Object` générique. Vous devez donc transtyper la valeur de retour afin de lui attribuer le type attendu. La méthode `hasNext()` vous permet quant à elle de vérifier s'il existe d'autres éléments qui n'ont pas encore été traités. En combinant ces deux méthodes, vous pouvez ainsi aisément créer une boucle `for` pour parcourir un à un chacun des éléments d'une collection, comme le montre l'exemple précédent.

L'exemple précédent indique aussi comment parcourir en boucle un ensemble ou une liste, les clés d'un dictionnaire, les valeurs d'un dictionnaire et les clés et les valeurs d'un dictionnaire.

Info

Les itérateurs peuvent être utiles pour exposer des collections via une API. L'avantage qu'apporte l'exposition des données à l'aide d'un itérateur tient à ce que le code appelant n'a pas à se soucier de la manière dont les données sont stockées. Cette implémentation permet de changer le type de collection sans avoir à modifier l'API.

Créer une collection mappée

```
HashMap map = new HashMap();
map.put(key1, obj1);
map.put(key2, obj2);
map.get(key3, obj3);
```

Cet exemple utilise un `HashMap` pour créer une collection mappée d'objets. Le `HashMap` possède une méthode `put()` qui prend deux paramètres. Le premier est une valeur de clé et le second l'objet que vous souhaitez stocker dans le dictionnaire. Dans cet exemple, nous stockons donc trois objets (`obj1`, `obj2` et `obj3`) en les indexant avec des clés (respectivement `key1`, `key2` et `key3`). La classe `HashMap` est l'une des classes Java les plus couramment utilisées. Dans un `HashMap`, les objets placés dans un dictionnaire doivent tous être du même type de classe. Si `obj1` est un objet `String`, `obj2` et `obj3` doivent donc également être des objets `String`.

Pour récupérer les objets placés dans la collection, vous utilisez la méthode `get()` du `HashMap`. Elle prend un unique paramètre correspondant à la clé de l'élément à récupérer. Si l'élément est trouvé, il est retourné sous forme d'objet générique `Object` : il faut donc le transtyper pour lui attribuer le type désiré. Si l'élément que vous tentez de récupérer n'existe pas, la valeur `null` est retournée.

Info

Le JDK 1.5 introduit une nouvelle fonctionnalité du langage, les génériques, permettant de récupérer des éléments d'un `HashMap` sans avoir à réaliser de transtypage. Sun propose un excellent article sur l'utilisation des génériques à l'adresse suivante : http://java.sun.com/developer/technicalArticles/J2SE/generics/index.html.

Les objets utilisés comme valeurs de clé dans un `HashMap` doivent implémenter les méthodes `equals()` et `hashCode()`. Ces méthodes sont utilisées par l'implémentation `HashMap` pour retrouver les éléments dans le dictionnaire. Si elles ne sont pas implémentées dans un objet utilisé comme valeur de clé, les objets clés sont retrouvés en fonction de leur identité uniquement. Dans ce cas, pour trouver une clé concordante, vous devrez passer l'instance d'objet elle-même lorsque vous essayerez de récupérer un objet : *a priori*, ce n'est pas le but recherché !

Stocker une collection

```
// Trier un tableau
int[] myInts = {1,5,7,8,2,3};
Arrays.sort(myInts);

// Trier une liste
List myList = new ArrayList();
myList.put(obj1);
myList.put(obj2);
Collections.sort(myList);
```

La classe `Arrays` est une classe du paquetage `java.util` contenant un grand nombre de méthodes statiques servant à manipuler des tableaux. La plus utile d'entre elles est sans doute la méthode `sort()`. Cette méthode prend un tableau d'objets ou de types primitifs et des index de début et de fin. L'index de début spécifie l'index du premier élément à trier et l'index de fin celui du dernier élément à trier.

Les primitifs sont triés par ordre croissant. Lorsque cette méthode est utilisée pour trier des objets, tous les objets doivent implémenter l'interface `Comparable`, à défaut de quoi un objet `Comparator` peut être passé.

Dans l'exemple précédent, nous commençons avec un tableau d'entiers de type `int`. Nous passons ce tableau à la méthode `Arrays.sort()` qui le trie. Notez bien que c'est le tableau lui-même qui est passé : il est donc directement trié et modifié. La méthode `sort()` ne retourne pas de nouveau tableau trié : son type de retour est `void`.

La classe `Collections`, autre classe du paquetage `java.util`, contient des méthodes statiques qui opèrent sur d'autres objets de collection. La méthode `sort()` prend un objet `List` en entrée et trie les éléments dans la liste par ordre croissant, selon l'ordre naturel des éléments. Comme avec la méthode `sort()` de l'objet `Arrays`, tous les éléments dans l'objet `List` passé à la méthode doivent implémenter l'interface `Comparable` à défaut de quoi un objet `Comparator` peut être passé avec l'objet `List`. La liste passée à la méthode `sort()` est elle-même modifiée.

Dans la seconde partie de notre exemple, nous créons un objet `ArrayList` et utilisons la méthode `Collections.sort()` pour le trier. Dans cet exemple, aucun objet `Comparator` n'a été passé, aussi les objets `obj1` et `obj2` doivent impérativement implémenter l'interface `Comparable`.

Si l'ordre de tri par défaut ne vous convient pas, vous pouvez implémenter l'interface `Comparator` pour définir votre propre mécanisme de tri. Le comparateur que vous définissez peut ensuite être passé comme second argument à la méthode `sort()` de la classe `Collections` ou `Arrays`.

En plus des classes que nous venons de citer, le framework Collections contient des classes dont le tri est inhérent, comme les objets `TreeSet` et `TreeMap`. Si vous utilisez ces classes, les éléments sont automatiquement triés lorsqu'ils sont placés dans la collection. Dans le cas d'un `TreeSet`, les éléments sont triés par ordre croissant

d'après l'interface `Comparable` ou d'après le `Comparator` fourni au moment de la création. Dans le cas d'un `TreeMap`, les éléments sont triés par ordre croissant de clé d'après l'interface `Comparable` ou d'après le `Comparator` fourni au moment de la création.

Trouver un objet dans une collection

```
// Trouver un objet dans un ArrayList
int index = myArrayList.indexOf(myStringObj);

// Trouver un objet par valeur dans un HashMap
myHashMap.containsValue(myStringObj);

// Trouver un objet par clé dans un HashMap
myHashMap.containsKey(myStringObj);
```

Ces exemples montrent comment retrouver des objets dans les collections les plus couramment utilisés : `ArrayList` et `HashMap`. La méthode `indexOf()` de l'objet `ArrayList` permet de retrouver la position dans le tableau à laquelle se trouve un objet particulier. Si l'objet passé à la méthode `indexOf()` n'est pas retrouvé, la méthode retourne `-1`. L'objet `HashMap` indexe les éléments par objets et non par valeurs entières comme le fait l'objet `ArrayList`. Les méthodes `containsValue()` ou `containsKey()` peuvent être utilisées pour déterminer si le `HashMap` contient l'objet passé comme valeur ou comme clé dans le dictionnaire. Elles retournent une valeur booléenne.

Deux autres méthodes, `binarySearch()` et `contains()`, permettent également de retrouver des objets dans des collections. La méthode `binarySearch()` est une méthode

des classes utilitaires `Arrays` et `Collections`. Elle effectue une recherche dans un tableau selon l'algorithme de recherche binaire. Le tableau doit être trié avant d'appeler la méthode `binarySearch()` de la classe `Arrays`. Sans cela, les résultats sont indéfinis. Le tri du tableau peut être réalisé avec la méthode `Arrays.sort()`. Si le tableau contient plusieurs éléments possédant la valeur spécifiée comme valeur de recherche, rien ne permet de déterminer celui qui sera retrouvé. Selon la même logique, la méthode `binarySearch()` de la classe `Collections` ne doit être utilisée que sur une collection déjà triée par ordre croissant selon l'ordre naturel de ses éléments. Ce tri peut être réalisé avec la méthode `Collections.sort()`. Comme pour les tableaux, l'emploi de `binarySearch()` sur une collection non triée produit des résultats indéfinis. S'il existe plusieurs éléments correspondant à l'objet recherché, rien ne permet non plus de déterminer celui qui sera retrouvé.

Lorsque la collection n'est pas déjà triée, il peut être préférable d'utiliser la méthode `indexOf()` plutôt que de réaliser le tri (`sort()`) puis la recherche binaire (`binarySearch()`). L'opération de tri (`sort()`) peut être coûteuse dans le cas de certaines collections.

L'exemple suivant utilise la méthode `binarySearch()` pour effectuer une recherche dans un tableau d'entiers :

```
int[] myInts = new int[]{7, 5, 1, 3, 6, 8, 9, 2};
Arrays.sort(myInts);
int index = Arrays.binarySearch(myInts, 6);
System.out.println("Value 6 is at index: " + index);
```

Ce code produit la sortie suivante :

```
The value 6 is at index 4.
```

La classe `ArrayList` possède également une méthode `contains()` qui peut être utilisée pour vérifier si un objet donné est membre d'un `ArrayList` donné.

Convertir une collection en un tableau

```
// Convertir un ArrayList en un tableau d'objets
Object[] objects = aArrayList.toArray();

// Convertir un HashMap en un tableau d'objets
Object[] mapObjects = aHashMap.entrySet().toArray();
```

Comme le montre cet exemple, il est assez simple en Java de convertir une collection telle qu'un `ArrayList` ou un `HashMap` en un tableau d'objets standard.

L'objet `ArrayList` possède une méthode `toArray()` qui retourne un tableau d'objets. La conversion d'un `HashMap` en un tableau est légèrement différente. Il faut d'abord obtenir les valeurs stockées dans le `HashMap` sous forme de tableau en utilisant la méthode `entrySet()`.

La méthode `entrySet()` retourne les valeurs de données sous forme de `Set` Java. Une fois que l'objet `Set` est obtenu, nous pouvons appeler la méthode `toArray()` pour récupérer un tableau contenant les valeurs stockées dans le `HashMap`.

5

Dates et heures

La plupart des programmes Java sont immanquablement conduits à devoir gérer des dates et des heures à un moment ou un autre. Fort heureusement, la gestion des dates et des heures est fort bien intégrée à Java. Trois classes principales sont utilisées dans la plupart des programmes pour stocker et manipuler les heures et les dates : `java.util.Date`, `java.sql.Date` et `java.util.Calendar`.

Bon nombre des méthodes de la classe `java.util.Date` sont maintenant déconseillées : vous devez donc éviter de les utiliser pour vos nouveaux projets de développement. Les méthodes déconseillées concernent pour la plupart la création et la manipulation des dates. Dans le cas de ces opérations, il est préférable d'utiliser le mécanisme de la classe `java.util.Calendar`. Les conversions entre les objets `Date` et `Calendar` sont faciles. Si vous préférez passer vos dates sous forme d'objets `Date`, il est donc parfaitement possible d'éviter les méthodes déconseillées. Vous devez alors simplement convertir vos dates en objets `Calendar` au moment de les manipuler. L'un des exemples de ce chapitre montre comment effectuer la conversion entre les objets `Date` et les objets `Calendar`.

Retrouver la date d'aujourd'hui

```
Date today = new java.util.Date();
System.out.println("Today's Date is " + today.toString());
```

Il est impératif que vous soyez familiarisé avec l'objet `Date` du paquetage `java.util` si vos programmes traitent avec des dates et des heures, car cet objet est très fréquemment utilisé. Il permet notamment de récupérer très simplement la date et l'heure courantes. Lorsque vous créez une instance de l'objet `Date`, celle-ci est initialisée avec l'heure et la date courantes.

La classe `Calendar` propose une autre méthode pour récupérer la date et l'heure courantes, comme ceci :

```
Calendar cal = Calendar.getInstance();
```

Cette ligne produit un objet `Calendar` nommé `cal` et l'initialise avec la date et l'heure courantes.

Conversion entre les objets Date et Calendar

```
// Conversion de Date à Calendar
Date myDate = new java.util.Date();
Calendar myCal = Calendar.getInstance();
myCal.setTime(myDate);

// Conversion de Calendar à Date
Calendar newCal = Calendar.getInstance();
Date newDate = newCal.getTime();
```

En travaillant avec des heures et des dates, vous constaterez souvent qu'il est nécessaire d'effectuer une conversion entre des objets Java `Date` et `Calendar`. Cette tâche est

heureusement très simple, comme en atteste l'exemple précédent. L'objet `Calendar` possède une méthode `setTime()` qui prend un objet `java.util.Date` en entrée et positionne l'objet `Calendar` en lui attribuant la date et l'heure contenues dans l'objet `Date` passé. Pour la conversion inverse, vous pouvez utiliser la méthode `getTime()` de la classe `Calendar` qui retourne la date et l'heure du calendrier sous forme d'objet `java.util.Date`.

La plupart des applications Java se servent des classes `Date` et `Calendar`. De là l'importance qu'il y a à se familiariser avec le processus de conversion d'un type vers l'autre. Il est conseillé de créer des méthodes utilitaires pour réaliser ces conversions afin de pouvoir les utiliser depuis n'importe quel emplacement du code au moyen d'un simple appel de méthode. Voici des méthodes simples pour la conversion des objets `Calendar` en objets `Date` et des objets `Date` en objets `Calendar` :

```
public static Date calToDate(Calendar cal) {
   return cal.getTime();
}

public static Calendar dateToCal(Date date) {
     Calendar myCal = Calendar.getInstance();
     myCal.setTime(date);
     return myCal;
}
```

Imprimer une date/une heure dans un format spécifié

```
Date todaysDate = new java.util.Date();
SimpleDateFormat formatter =
      new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss");
String formattedDate = formatter.format(todaysDate);
System.out.println("Today's Date and Time is: " +
➥formattedDate);
```

Le Java contient des classes de formatage qui peuvent être utilisées pour appliquer à une date un format désiré. La classe la plus couramment utilisée pour le formatage des dates est la classe `SimpleDateFormat`. Elle prend une chaîne de format en entrée de son constructeur et retourne un objet de format qui peut être utilisé ensuite pour formater des objets `Date`. La méthode `format()` de l'objet `SimpleDateFormat` retourne une chaîne contenant la représentation formatée de la `Date` passée en paramètre.

Voici la sortie de l'exemple précédent :

```
Today's Date and Time is: jeu., 04 janv. 2007 16:48:38
```

La chaîne de format passée au constructeur `SimpleDateFormat` peut paraître un peu obscure si vous ne connaissez pas les codes de formatage à utiliser. Le Tableau 5.1 présente les codes qui peuvent être passés au constructeur de `SimpleDateFormat`. Dans notre exemple, nous avons utilisé la chaîne de format suivante :

```
"EEE, dd MMM yyyy HH:mm:ss"
```

Décomposons cette chaîne en nous référant au Tableau 5.1, afin de comprendre le format demandé :

- **`EEE`** : représentation sur trois caractères du jour de la semaine (par exemple, `Mar`).
- **`,`** : place une virgule dans la sortie.

- **dd** : représentation sur deux caractères du jour du mois (01 à 31).
- **MMM** : représentation sur trois caractères du mois de l'année (par exemple, Jul).
- **yyyy** : représentation sur quatre caractères de l'année (par exemple, 2006).
- **HH:mm:ss** : heure, minutes et secondes, séparées par des deux-points (par exemple, 11:18:33).

Ces éléments combinés produisent la chaîne de date suivante :

```
jeu., 04 janv. 2007 16:48:38
```

Tableau 5.1 : Codes de format de date et d'heure

Lettre	Composant de date ou d'heure	Présentation	Exemples
G	Désignateur d'ère	Texte	AD
y	Année	Année	1996 ; 96
M	Mois de l'année	Mois	Juillet ; Jul ; 07
w	Semaine de l'année	Nombre	27
W	Semaine du mois	Nombre	2
D	Jour de l'année	Nombre	189
d	Jour du mois	Nombre	10
F	Jour de la semaine dans le mois	Nombre	2
E	Jour de la semaine	Texte	Mardi ; Mar
a	Marqueur A.M./P.M.	Texte	PM

Tableau 5.1 : Codes de format de date et d'heure *(suite)*

Lettre	Composant de date ou d'heure	Présentation	Exemples
`H`	Heure du jour (`0` à `23`)	Nombre	`0`
`k`	Heure du jour (`1` à `24`)	Nombre	`24`
`K`	Heure au format A.M./P.M. (`0` à `11`)	Nombre	`0`
`h`	Heure au format A.M./P.M. (`1` à `12`)	Nombre	`12`
`m`	Minutes dans l'heure	Nombre	`30`
`s`	Secondes dans la minute	Nombre	`55`
`S`	Millisecondes	Nombre	`978`
`z`	Fuseau horaire	Fuseau horaire général	`Pacific Standard Time` ; `PST` ; `GMT-08:00`
`Z`	Fuseau horaire	Fuseau horaire RFC 822	`-0800`

En plus de créer vos propres chaînes de format, vous pouvez utiliser l'une des chaînes de format prédéfinies avec les méthodes `getTimeInstance()`, `getDateInstance()` ou `getDateTimeInstance()` de la classe `DateFormat`. Par exemple, le code suivant retourne un objet formateur qui utilisera un format de date pour vos paramètres régionaux par défaut :

```
DateFormat df = DateFormat.getDateInstance();
```

Le formateur `df` peut ensuite être utilisé exactement comme nous avons utilisé l'objet `SimpleDateFormat` dans notre exemple précédent. Consultez la documentation JavaDoc disponible pour la classe `DateFormat` pour obtenir des informations détaillées sur les objets de formatage d'heure et de date standard disponibles à l'adresse **http://java.sun.com/j2se/1.5.0/docs/api/java/text/DateFormat.html**.

Parser des chaînes en dates

```
String dateString = "January 12, 1952 or 3:30:32pm";
DateFormat df = DateFormat.getDateInstance();
Date date = df.parse(dateString);
```

L'objet `DateFormat` est utilisé pour parser un objet `String` et obtenir un objet `java.util.Date`. La méthode `getDateInstance()` crée un objet `DateFormat` avec le format de date standard de votre pays. Vous pouvez ensuite utiliser la méthode `parse()` de l'objet `DateFormat` retourné pour parser votre chaîne de date et obtenir un objet `Date`, comme le montre cet exemple.

La méthode `parse()` accepte aussi un second paramètre qui spécifie la position dans la chaîne à partir de laquelle l'analyse doit être effectuée.

Les classes `java.sql.Date`, `java.sql.Time` et `java.sql.Timestamp` contiennent une méthode statique appelée `valueOf()` qui peut également être utilisée pour parser des chaînes de date simples au format aaaa-mm-jj. Elles sont très utiles pour convertir en objets `Date` des dates utilisées dans des chaînes SQL avec des objets JDBC.

Ces techniques sont utiles pour convertir des données de date entrées par l'utilisateur en objets `Date` Java pour un traitement ultérieur dans votre application. Vous pouvez récupérer les dates entrées par l'utilisateur sous forme de chaîne et les convertir en objets `Date` à l'aide de ces techniques.

Additions et soustractions avec des dates ou des calendriers

```
// Arithmétique de dates utilisant des objets Date
Date date = new Date();
long time = date.getTime();
time += 5*24*60*60*1000;
Date futureDate = new Date(time);

// Arithmétique de dates utilisant des objets Calendar
Calendar nowCal = Calendar.getInstance();
nowCal.add(Calendar.DATE, 5);
```

Si vous utilisez un objet `Date`, la technique d'ajout ou de soustraction des dates consiste à convertir d'abord l'objet en une valeur `long` en utilisant la méthode `getTime()` de l'objet `Date`. La méthode `getTime()` retourne l'heure mesurée en millisecondes depuis le début de "l'ère" UNIX (1er janvier 1970, 00 h 00 s 00 ms GMT). Ensuite, vous devez réaliser l'opération arithmétique avec des valeurs `long` et reconvertir le résultat en objet `Date`. Dans l'exemple précédent, nous ajoutons 5 jours à l'objet `Date`. Nous convertissons les 5 jours en millisecondes en multipliant 5 par le nombre d'heures dans une journée (`24`), le nombre de minutes dans une heure (`60`), le nombre de secondes dans une minute (`60`) et finalement par `1 000` pour convertir les secondes en millisecondes.

Les opérations arithmétiques de date peuvent être réalisées directement sur des objets `Calendar` en utilisant la méthode `add()`. Cette méthode prend deux paramètres, un champ et une quantité, tous deux de type `int`. La quantité spécifiée est ajoutée au champ spécifié. Le champ peut correspondre à n'importe quel champ de date valide, comme un jour, une semaine, un mois, une année, etc. Pour soustraire une heure, vous devez utiliser une valeur négative. En positionnant le paramètre de champ à la constante `Calendar` appropriée, vous pouvez directement ajouter ou soustraire des jours, des semaines, des mois, des années, etc. La seconde partie de l'exemple précédent montre comment ajouter 5 jours à un objet `Calendar`.

Calculer la différence entre deux dates

```
long time1 = date1.getTime();
long time2 = date2.getTime();
long diff = time2 - time1;
System.out.println("Difference in days = " +
 ➥diff/(1000*60*60*24));
```

Cet exemple convertit deux objets de date `date1` et `date2` en millisecondes – chacun représenté sous forme de `long`. La différence est calculée en soustrayant `time1` à `time2`. La différence calculée en jours est ensuite imprimée en réalisant l'opération arithmétique nécessaire pour convertir la différence en millisecondes en une différence en jours.

Il arrivera souvent que vous souhaitiez déterminer la durée qui sépare deux dates, par exemple en calculant le nombre de jours restants avant l'expiration d'un produit.

Si vous connaissez la date d'expiration d'un produit, vous pouvez calculer le nombre de jours avant expiration en calculant la différence entre la date d'expiration et la date courante.

Voici un exemple de méthode pour réaliser ce calcul :

```
public static void daysTillExpired(Date expDate) {
    Date currentDate = new Date();
    long expTime = expDate.getTime();
    long currTime = currentDate.getTime();
    long diff = expTime - currTime;
    return diff/(1000*60*60*24);
}
```

Cette méthode prend une date d'expiration en entrée et calcule le nombre de jours jusqu'à la date d'expiration. Cette valeur en jours est retournée par la méthode. Elle peut fournir un nombre négatif si la date d'expiration est dépassée.

Comparer des dates

```
if (date1.equals(date2)) {
  System.out.println("dates are the same.");
}
else {
  if (date1.before(date2)) {
    System.out.println("date1 before date2");
  }
  else {
    System.out.println("date1 after date2");
  }
}
```

Cet exemple utilise les méthodes `equals()` et `before()` de la classe `Date`. La méthode `equals()` retourne `true` si les valeurs de données sont les mêmes. Sinon, elle retourne `false`. Les dates doivent être les mêmes à la milliseconde près pour que la méthode `equals()` retourne `true`. La méthode `before()` retourne `true` si la date sur laquelle elle est appelée intervient avant la date qui lui est passée en paramètre.

La classe `Date` possède également une méthode `after` qui est utilisée de manière analogue à la méthode `before()` pour déterminer si la date à partir de laquelle elle est appelée intervient après la date passée en paramètre.

La méthode `compareTo()` de la classe `Date` est aussi utile pour comparer deux dates. Elle prend un argument de det retourne une valeur d'entier. Elle retourne 0 si la date à partir de laquelle elle est appelée équivaut à celle passée en argument, une valeur négative si elle lui est antérieure et une valeur positive si elle lui est ultérieure.

Retrouver le jour de la semaine/ du mois/de l'année ou le numéro de la semaine

```
Calendar cal = Calendar.getInstance();
System.out.println("Day of week: " +
➥cal.get(Calendar.DAY_OF_WEEK));
System.out.println("Month: " + cal.get(Calendar.MONTH));
System.out.println("Year: " + cal.get(Calendar.YEAR));
System.out.println("Week number: " +
 ➥cal.get(Calendar.WEEK_OF_YEAR));
```

Vous pouvez aisément déterminer des valeurs comme le jour de la semaine, le mois, l'année ou le numéro de semaine avec la méthode `get()` de l'objet `Calendar`. Dans l'exemple précédent, nous obtenons un objet `Calendar` représentant la date et l'heure courantes avec la méthode `getInstance()`. Nous imprimons ensuite le jour de la semaine, le mois, l'année puis la semaine de l'année en utilisant la méthode `get()` et en passant la constante `Calendar` appropriée pour spécifier le champ à récupérer.

Pour obtenir ces valeurs avec un objet `Date`, vous devez d'abord convertir l'objet `Date` en un objet `Calendar` en utilisant la méthode `setTime()` d'une instance `Calendar` et en passant l'objet `Date` à convertir. La conversion entre les objets `Date` et `Calendar` a été présentée précédemment dans ce chapitre.

Calculer une durée écoulée

```
long start = System.currentTimeMillis();
// Réaliser une autre action…
long end = System.currentTimeMillis();
long elapsedTime = end - start;
```

En calculant une durée écoulée, vous pouvez déterminer le temps requis pour réaliser une action ou le temps de progression d'un processus. Pour cela, vous devez utiliser la méthode `System.currentTimeMillis()` afin d'obtenir l'heure courante en millisecondes. Cette méthode doit être utilisée au début et à la fin de la tâche à chronométrer, afin de calculer la différence entre les deux mesures. La valeur retournée par la méthode `System.currentTimeMillis()` correspond au temps écoulé depuis le 1er janvier 1970, 00 h 00 s 00 ms, en millisecondes.

Le JDK 1.5 introduit une méthode `nanoTime()` dans la classe `System`, qui permet d'obtenir une mesure plus précise encore, à la nanoseconde. Toutes les plates-formes ne prennent cependant pas en charge cette précision : bien que la méthode `nanoTime()` soit disponible, il n'est donc pas toujours possible de compter sur une mesure en nanosecondes. Ce niveau de précision est souvent utile pour les tests, le profilage et l'analyse des performances.

6

Retrouver des motifs avec des expressions régulières

Les expressions régulières ont été introduites en Java à la sortie du JDK 1.4. Les expressions régulières spécifient des motifs pouvant être retrouvés dans des séquences de caractères. Elles sont particulièrement utiles pour l'analyse des chaînes et économisent souvent au programmeur bien du temps et des efforts par rapport aux solutions qui n'y font pas appel. Avant d'être ajoutées au Java, elles ont été utilisées pendant des années par les programmeurs UNIX. Les outils UNIX standard comme sed et awk les emploient notamment. Les expressions régulières sont aussi couramment utilisées dans le langage de programmation Perl. Leur ajout au JDK représente un intéressant renforcement des capacités du Java.

Dans ce chapitre, vous allez apprendre à utiliser les fonctionnalités liées aux expressions régulières du Java afin de retrouver et remplacer des portions de texte ou d'en déterminer la concordance au regard d'un modèle.

Grâce à ces acquis, vous pourrez déterminer les cas où l'usage d'un traitement par les expressions régulières peut être utile dans vos applications.

Les expressions régulières en Java

Les classes Java `Matcher` et `Pattern` que vous utiliserez pour les opérations liées aux expressions régulières sont contenues dans le paquetage `java.util.regex`. Elles permettent à la fois de retrouver des séquences de caractères et d'en déterminer la concordance d'après des motifs d'expression régulière. Quelle différence faut-il faire entre "retrouver" et "déterminer la concordance" ? L'opération de recherche permet de retrouver des correspondances dans une chaîne. L'opération de détermination de la concordance requiert que la chaîne entière soit une correspondance précise de l'expression régulière.

Les tâches pour lesquelles vous faisiez auparavant appel à la classe `StringTokenizer` sont en général toutes désignées pour être simplifiées à l'aide d'expressions régulières.

Info

Si vous ne pouvez pas utiliser une version de Java contenant le paquetage des expressions régulières (autrement dit, une version 1.4 ou ultérieure), il existe une bonne solution de remplacement avec le paquetage Jakarta RegExp d'Apache. Ce livre ne couvre pas le paquetage Jakarta, mais vous trouverez des informations et une documentation complète le concernant à l'adresse http://jakarta.apache.org/regexp.

Le Tableau 6.1 présente les caractères spéciaux courants utilisés dans les expressions régulières. Vous pourrez vous référer à ce tableau en consultant les exemples du chapitre.

Tableau 6.1 : Tableau des expressions régulières – caractères spéciaux couramment utilisés

Caractère spécifique	Description
`^`	Début de la chaîne
`$`	Fin de la chaîne
`?`	0 ou 1 fois (fait référence à l'expression régulière précédente)
`*`	0 ou plusieurs fois (fait référence à l'expression régulière précédente)
`+`	1 ou plusieurs fois (fait référence à l'expression régulière précédente)
`[…]`	Classe de caractères
`|`	Opérateur union
`.`	N'importe quel caractère
`\d`	Un chiffre
`\D`	Caractère autre qu'un chiffre
`\s`	Caractère d'espace blanc (espace, tabulation, nouvelle ligne, saut de page, retour chariot)
`\S`	Caractère autre qu'espace blanc
`\w`	Caractère de mot `[a-zA-Z_0-9]`
`\W`	Caractère autre qu'un caractère de mot `[^\w]`

Dans le Tableau 6.1, les expressions régulières échappées sont présentées en étant précédées par une unique barre oblique inversée. Notez cependant qu'à l'intérieur des chaînes Java, il convient d'utiliser deux barres obliques inversées à chaque fois. Le caractère de barre oblique inversée possède en effet une signification spéciale en Java : la double barre oblique inversée échappe dès lors le caractère de barre oblique inversée et équivaut à un unique caractère de barre oblique inversée.

La JavaDoc de la classe `Pattern` propose une liste plus complète des caractères utilisés pour exprimer des expressions régulières. Elle est disponible à l'adresse suivante : **http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html**.

Retrouver une portion de texte à l'aide d'une expression régulière

```
String pattern = "[TJ]im";
Pattern regPat = Pattern.compile(pattern);
String text = "This is jim and Timothy.";
Matcher matcher = regPat.matcher(text);
if (matcher.find()) {
  String matchedText = matcher.group();
}
```

Cet exemple utilise les classes `Pattern` et `Matcher`. Pour commencer, nous utilisons la méthode `compile()` de la classe `Pattern` afin de compiler une chaîne de motif en un objet `Pattern`. Une fois que nous avons l'objet `Pattern` `regPat`, nous utilisons la méthode `matcher()` et passons la chaîne de texte en fonction de laquelle la correspondance

doit être établie. La méthode `matcher()` retourne une instance de la classe `Matcher`. Pour finir, nous appelons la méthode `group()` de la classe `Matcher` pour obtenir le texte correspondant. Le texte correspondant dans cet exemple est la chaîne `"Tim"`. Notez que la chaîne `"jim"` n'est pas une occurrence correspondante, car par défaut, les expressions régulières tiennent compte de la casse des caractères. Pour opérer une recherche en ignorant la casse, ce code doit être légèrement modifié, comme ceci :

```
String patt = "[TJ]im";
Pattern regPat = Pattern.compile(patt,
    ➥Pattern.CASE_INSENSITIVE);
String text = "This is jim and Timothy.";
Matcher matcher = regPat.matcher(text);
if (matcher.find()) {
    String matchedText = matcher.group();
}
```

Le texte retourné est maintenant la chaîne de caractères `"jim"`. La mise en correspondance n'étant cette fois pas sensible à la casse, la première correspondance `"jim"` intervient avant la correspondance portant sur `"Tim"`.

Notez que la seule différence par rapport au précédent exemple tient à ce que nous avons ajouté un paramètre supplémentaire à la méthode `compile()` en créant notre objet `Pattern`. Cette fois-ci, nous passons le drapeau `CASE_INSENSITIVE` afin d'indiquer que nous souhaitons que la correspondance soit établie sans tenir compte de la casse. Lorsque ce drapeau n'est pas inclus, la correspondance s'établit par défaut en tenant compte de la casse.

Si votre code doit s'exécuter sous différents paramètres régionaux, vous pouvez également passer le drapeau de casse Unicode. La ligne de compilation ressemblera alors à ceci :

```
Pattern regPat = Pattern.compile(pattern,
➥Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE);
```

Comme vous le voyez, les différents drapeaux passés à la méthode `compile()` sont séparés les uns des autres par un signe OU logique. Les drapeaux `Pattern` doivent être passés au moment où le `Pattern` est créé en utilisant la méthode `compile()`. Une fois que l'objet `Pattern` est créé, il est immuable, ce qui signifie qu'il ne peut plus être changé.

Dans les précédents exemples, nous avons utilisé la méthode `find()` de la classe `Matcher` pour retrouver la première correspondance dans notre chaîne d'entrée. La méthode `find()` peut cependant être appelée de manière répétitive afin de retourner toutes les correspondances successives dans la chaîne d'entrée. La méthode `find()` retourne `true` chaque fois qu'une correspondance est trouvée. Elle retourne `false` lorsqu'il n'y a plus de correspondance. Si vous appelez `find()` de nouveau après qu'elle a retourné `false`, elle se réinitialise et retrouve la première occurrence de nouveau. Il existe une autre méthode `find()` qui prend un paramètre `int` spécifiant un index à partir duquel la recherche doit démarrer. Pour le reste, cette méthode `find()` se comporte exactement comme la méthode `find()` sans paramètre.

D'autres solutions sont possibles pour obtenir le résultat de la correspondance. Nous avons utilisé la méthode `group()` de la classe `Matcher`, mais il existe aussi des méthodes pratiques `start()` et `end()`. La méthode `start()`

retourne l'index au début de la précédente correspondance. La méthode `end()` retourne l'index après le dernier caractère mis en correspondance.

Remplacer du texte mis en correspondance

```
String pattern = "[TJ]im";
Pattern regPat = Pattern.compile(pattern);
String text = "This is jim and Tim.";
Matcher matcher = regPat.matcher(text);
String string2 = matcher.replaceAll("John");
```

Cet exemple montre comment remplacer le texte retrouvé avec notre chaîne de motif par un texte de remplacement. La valeur de `string2` à la fin de cet exemple est la suivante :

```
This is jim and John.
```

L'occurrence de `"jim"` n'est pas remplacée, car la mise en correspondance des expressions régulières tient par défaut compte de la casse. Pour établir une correspondance en ignorant la casse, référez-vous à l'exercice précédent.

Nous utilisons les classes `Pattern` et `Matcher` comme nous l'avons fait lors de la mise en correspondance élémentaire de l'exercice précédent. L'étape nouvelle concerne ici notre appel à la méthode `replaceAll()` de `Matcher`. Nous passons en paramètre le texte à utiliser comme texte de remplacement. Celui-ci vient remplacer toutes les occurrences retrouvées du motif. Cette technique est très efficace pour remplacer des portions d'une chaîne par une chaîne de remplacement.

L'autre technique utile pour le remplacement du texte consiste à utiliser les méthodes `appendReplacement()` et `appendTail()` de la classe `Matcher`. L'usage combiné de ces méthodes permet de remplacer des occurrences d'une sous-chaîne à l'intérieur d'une chaîne. Le code suivant présente un exemple de cette technique :

```
Pattern p = Pattern.compile("My");
Matcher m = p.matcher("My dad and My mom");
StringBuffer sb = new StringBuffer();
boolean found = m.find();
while(found) {
    m.appendReplacement(sb, "Our");
    found = m.find();
}
m.appendTail(sb);
System.out.println(sb);
```

La sortie de ce code produit l'impression de la ligne suivante avec la méthode `System.out.println()` :

```
Our dad and Our mom
```

Dans cet exemple, nous créons un objet `Pattern` afin de retrouver les occurrences du texte "My". La méthode `appendReplacement()` écrit les caractères de la séquence d'entrée ("My dad and my mom") dans le tampon `StringBuffer sb`, jusqu'au dernier caractère avant la précédente correspondance. Elle ajoute ensuite au `StringBuffer` la chaîne de remplacement passée en second paramètre. Pour finir, elle fixe la position de la chaîne courante au niveau de la fin de la dernière correspondance. Ce processus se répète jusqu'à ce qu'il n'y ait plus de correspondance. A ce moment-là, nous appelons la méthode `appendTail()` pour ajouter la portion restante de la séquence d'entrée au `StringBuffer`.

Retrouver toutes les occurrences d'un motif

```
String pattern = "\\st(\\w)*o(\\w)*";
Pattern regPat = Pattern.compile(pattern);
String text = "The words are town tom ton toon house.";
Matcher matcher = regPat.matcher(text);
while (matcher.find()) {
  String matchedText = matcher.group();
  System.out.println("match - " + matchedText);
}
```

Dans les précédents exemples du chapitre, nous n'avons trouvé qu'une unique correspondance d'un motif. Dans cet exemple, nous retrouvons toutes les occurrences d'un motif de correspondance donné dans une chaîne. Le motif utilisé est `"\\st(\\w)*o(\\w)*"`. Cette expression régulière retrouve tous les mots qui commencent par *t* et contiennent la lettre *o*. La sortie imprimée par notre instruction `System.out.println()` est la suivante :

```
town
tom
ton
toon
```

Décomposons cette expression régulière et voyons ce que chaque élément nous apporte :

- **`\\s`** : caractère spécial d'expression régulière correspondant à un caractère d'espace blanc.
- **`t`** : correspond à la lettre *t*.
- **`\\w*`** : caractère spécial d'expression régulière correspondant à zéro, un ou plusieurs caractères de mot (qui ne sont pas des espaces blancs).

- `o` : correspond à la lettre *o*.
- `\\w*` : caractère spécial d'expression régulière correspondant à zéro, un ou plusieurs caractères de mot (qui ne sont pas des espaces blancs).

Cette expression régulière ne correspond pas au premier mot de la chaîne, quand bien même celui-ci commencerait par un *t* et contiendrait un *o*, car le premier élément de l'expression régulière correspond à un caractère d'espace blanc or généralement, les chaînes ne commencent pas par un espace blanc.

Imprimer des lignes contenant un motif

```
String pattern = "^a";
Pattern regPat = Pattern.compile(pattern);
Matcher matcher = regPat.matcher("");
BufferedReader reader =
     new BufferedReader(new FileReader("file.txt"));
String line;
while ((line = reader.readLine()) != null) {
  matcher.reset(line);
  if (matcher.find()) {
    System.out.println(line);
  }
}
```

Cet exemple montre comment effectuer une recherche dans un fichier afin de trouver toutes les lignes contenant un motif donné. Ici, nous utilisons la classe `BufferedReader` pour lire des lignes depuis un fichier texte. Nous tentons de mettre en correspondance chaque ligne avec notre motif en utilisant la méthode `find()` de la classe `Matcher`.

Cette méthode retourne `true` si le motif est trouvé dans la ligne passée en paramètre. Nous imprimons toutes les lignes qui correspondent au motif donné. Notez que ce fragment de code peut lever des exceptions `FileNotFoundException` et `IOException`, qu'il convient de gérer dans votre code. Dans cet exemple, l'expression régulière correspond à n'importe quelle ligne contenue dans notre fichier d'entrée qui commence par la lettre *a* minuscule.

Le motif d'expression régulière que nous utilisons se décompose de la manière suivante :

- `^` : caractère spécial d'expression régulière correspondant au début d'une chaîne.
- **`a`** : correspond à la lettre *a*.

Retrouver des caractères de nouvelle ligne dans du texte

```
String pattern = "\\d$";
String text =
    "This is line 1\nHere is line 2\nThis is line 3\n";
Pattern regPat =
    Pattern.compile(pattern, Pattern.MULTILINE);
Matcher matcher = regPat.matcher(text);
while (matcher.find()) {
  System.out.println(matcher.group());
}
```

Dans cet exemple, nous utilisons le drapeau `Pattern.MULTILINE` pour retrouver des caractères de nouvelle ligne dans une chaîne de texte. Par défaut, les caractères d'expression régulière `^` et `$` ne correspondent qu'au début et à la fin d'une chaîne entière. Si une chaîne con-

tenait plusieurs lignes distinguées par des caractères de nouvelle ligne, l'expression régulière `^` ne correspondrait toujours qu'au début de la chaîne par défaut. Si nous passons le drapeau `Pattern.MULTILINE` à la méthode `Pattern.compile()` comme nous le faisons dans cet exemple, le caractère `^` correspond maintenant au premier caractère suivant un terminateur de ligne et le caractère `$` au caractère précédent un terminateur de ligne. Avec le drapeau `Pattern.MULTILINE`, le `^` correspond maintenant au début de chaque ligne dans une chaîne contenant plusieurs lignes séparées par des caractères de nouvelle ligne.

La sortie de cet exemple est la suivante :

```
1
2
3
```

Nous utilisons le motif `"\\d$"`. Dans cette expression régulière, le `\\d` correspond à n'importe quel chiffre unique. En mode `MULTILINE`, le `$` correspond au caractère intervenant juste avant un terminateur de ligne. L'effet intéressant est que notre expression régulière correspond à tout caractère chiffre unique présent à la fin d'une ligne. Nous obtenons donc la sortie précédente.

7

Nombres

Le travail avec des nombres en Java est un domaine dans lequel tout bon programmeur se doit d'être aguerri, car presque tous les programmes ont affaire à des nombres sous une forme ou une autre. Dans ce chapitre, nous utiliserons principalement les types numériques de base du Java, leurs objets encapsuleurs (*wrappers*) et la classe `java.lang.Math`.

Le Tableau 7.1 présente les types prédéfinis du Java et liste leurs objets encapsuleurs disponibles. Notez que le type booléen ne possède pas de taille en bits parce qu'il ne peut contenir que deux valeurs discrètes, `true` ou `false`.

Tableau 7.1 : Types prédéfinis du Java

Type	Taille en bits	Objet encapsuleur
`byte`	8	`Byte`
`short`	16	`Short`
`int`	32	`Integer`
`long`	64	`Long`

Tableau 7.1 : Types prédéfinis du Java *(suite)*

Type	Taille en bits	Objet encapsuleur
`float`	32	`Float`
`double`	64	`Double`
`char`	16	`Character`
`boolean`	--	`Boolean`

Les classes d'objet encapsuleur sont utiles lorsque vous souhaitez traiter un type de base comme un objet. Cette approche peut notamment être utile si vous souhaitez définir une API manipulant uniquement des objets. En encapsulant vos nombres sous forme d'objets, vous pouvez aussi sérialiser les types de base.

Vérifier si une chaîne est un nombre valide

```
try {
  int result = Integer.parseInt(aString);
}
catch (NumberFormatException ex) {
  System.out.println("The string does not contain a valid
  ➥number.");
}
```

Dans cet exemple, nous utilisons la méthode statique `parseInt()` de la classe `Integer` pour tenter de convertir le paramètre chaîne en un entier. Si le paramètre chaîne ne peut pas être converti en un entier valide, l'exception

`NumberFormatException` est levée. Si l'exception `NumberFormatException` n'est pas levée, on peut donc en conclure *a contrario* que la méthode `parseInt()` est parvenue à parser la chaîne en une valeur d'entier.

Il est aussi possible de déclarer la variable `int` en dehors du bloc `try` afin de pouvoir attribuer une valeur par défaut à la variable dans le bloc `catch` si l'exception `NumberFormatException` est levée. Le code devient alors le suivant :

```
int result = 0;
try {
    result = Integer.parseInt(aString);
}
catch (NumberFormatException ex) {
    result = DEFAULT_VALUE;
}
```

Comparer des nombres à virgule flottante

```
Float a = new Float(3.0f);
Float b = new Float(3.0f);
if (a.equals(b)) {
  // Ils sont égaux
}
```

Une prudence toute particulière est recommandée lors de la comparaison des nombres à virgule flottante en raison des erreurs d'arrondi. Au lieu de comparer les types Java de base à virgule flottante `float` et `double` avec l'opérateur ==, il est préférable de comparer leurs équivalents objet. La méthode `equals()` de `Float` et `Double` retourne `true` si et

seulement si les deux valeurs sont exactement identiques au bit près ou si elles correspondent toutes deux à la valeur `NaN`. Cette valeur désigne une valeur autre qu'un nombre (NaN pour *not a number*) – qui n'est pas un nombre valide.

En pratique, lors de la comparaison des nombres à virgule flottante, il n'est pas toujours souhaitable d'effectuer une comparaison exacte. Il est parfois plus judicieux de la fixer à une plage différentielle acceptable, aussi appelée *tolérance*. Les classes et les types Java ne possèdent malheureusement pas de fonctionnalité prédéfinie de ce type, mais vous pouvez assez aisément créer votre propre méthode `equals()` pour cela. Voici un fragment de code qui peut être utilisé pour créer une méthode de ce type :

```
float f1 = 2.99f;
float f2 = 3.00f;
float tolerance = 0.05f;
if (f1 == f2) System.out.println("they are equal");
else {
    if (Math.abs(f1-f2) < tolerance) {
     System.out.println("within tolerance");
    }
}
```

Nous comparons d'abord les nombres à virgule flottante en utilisant l'opérateur ==. S'ils sont égaux, nous imprimons un message correspondant. S'ils ne le sont pas, nous vérifions si la valeur absolue de leur différence est inférieure à la valeur de tolérance désirée. Cette technique permet de créer une méthode utile prenant en paramètre deux valeurs à virgule flottante et une tolérance, et retournant un résultat indiquant si les valeurs sont égales, à la plage de tolérance près.

Arrondir des nombres à virgule flottante

```
// Arrondir une valeur double
long longResult = Math.round(doubleValue);

// Arrondir une valeur float
int intResult = Math.round(floatValue);
```

Il convient d'être prudent si vous souhaitez convertir un nombre à virgule flottante en un entier. Si vous vous contentez de transtyper le nombre à virgule flottante en un `int` ou un `long`, la conversion en un `int` ou un `long` s'opère en tronquant simplement la portion décimale. Avec une valeur décimale comme 20.99999, vous obtiendrez donc 20 après le transtypage en une valeur `int` ou `long`. La méthode appropriée pour réaliser une conversion de nombre à virgule flottante en entier consiste à utiliser la méthode `Math.round()`. L'exemple précédent montre comment arrondir une valeur `double` et une valeur `float`. Si vous passez une valeur `double` à la méthode `Math.round()`, le résultat retourné est un `long`.

Si vous passez une valeur `float` à la méthode `Math.round()`, le résultat retourné est un `int`. La méthode `Math.round()` arrondit la valeur vers le haut si la partie décimale du nombre à virgule flottante est 0,5 ou plus et vers le bas pour les nombres dont la partie décimale est inférieure à 0,5.

Formater des nombres

```
double value = 1623542.765;
NumberFormat numberFormatter;
String formattedValue;
numberFormatter = NumberFormat.getNumberInstance();
formattedValue = numberFormatter.format(value);
System.out.format("%s%n", formattedValue);
```

Dans la plupart des applications, il est nécessaire d'afficher des nombres. Le Java permet par chance le formatage des nombres, afin de leur donner l'apparence désirée lorsque vous souhaitez les afficher dans votre application.

Cet exemple génère le nombre formaté suivant en sortie :

```
1 623 542,765
```

Dans cet exemple, nous utilisons la classe `NumberFormat` pour formater une valeur `double` en une représentation sous forme de chaîne séparée par des espaces. La classe `NumberFormat` se trouve dans le paquetage `java.text`. Elle est aussi très utile pour le code que vous devez diffuser dans plusieurs pays. La classe `NumberFormat` supporte le formatage de nombres et de devises et sait également représenter des nombres et des devises à l'aide de différents paramètres régionaux.

La classe `NumberFormat` peut aussi être utilisée pour formater des valeurs de pourcentage à afficher. Voici un exemple utilisant la classe `NumberFormat` pour formater un pourcentage à afficher :

```
double percent = 0.80;
NumberFormat percentFormatter;
String formattedPercent;
percentFormatter = NumberFormat.getPercentInstance();
```

```
formattedPercent = percentFormatter.format(percent);
System.out.format("%s%n", formattedPercent);
```

La sortie de ce code est la suivante :

```
80%
```

La classe `NumberFormat` possède également une méthode `parse()` qui peut être utilisée pour parser des chaînes contenant des nombres en un objet `Number`, à partir duquel peut s'obtenir un type numérique.

Le JDK 1.5 a introduit la classe `java.util.Formatter`, un objet de formatage de portée générale permettant de formater un grand nombre de types. En plus des nombres, cette classe peut également formater des dates et des heures. Elle est très bien documentée dans la documentation du JDK 1.5 à l'adresse suivante : **http://java.sun.com/j2se/1.5.0/docs/api/java/util/Formatter.html**.

Le JDK 1.5 ajoute aussi deux méthodes utilitaires à la classe `java.io.PrintStream` pour un formatage aisé des objets `OutputStream` : `format()` et `printf()`. Toutes deux prennent une chaîne de format et un nombre variable d'arguments `Object` en entrée. Ces méthodes sont très proches des méthodes de formatage des chaînes classiques `printf` et `scanf` du C. Pour plus d'informations sur ces méthodes, référez-vous à la documentation du JDK 1.5 à l'adresse **http://java.sun.com/j2se/1.5.0/docs/api/java/io/PrintStream.html**.

Dans l'exemple suivant, nous allons voir comment formater des valeurs de devise à afficher avec la classe `NumberFormat`.

Formater des devises

```
double currency = 567123678.99;
NumberFormat currencyFormatter;
String formattedCurrency;

currencyFormatter = NumberFormat.getCurrencyInstance();
formattedCurrency = currencyFormatter.format(currency);
System.out.format("%s%n", formattedCurrency);
```

Comme pour l'exemple précédent concernant le formatage des nombres, nous utilisons ici la classe `NumberFormat`, mais afin de formater cette fois une valeur de devise. Nous utilisons la méthode `getCurrencyInstance()` de la classe `NumberFormat` pour obtenir une instance de formatage de devise de la classe. Avec cette instance, nous pouvons passer une valeur à virgule flottante et récupérer en retour une valeur de devise formatée. La sortie de cet exemple produit la chaîne suivante :

```
567 123 678,99
```

En plus de placer des virgules aux emplacements appropriés, le formateur de devises ajoute automatiquement le signe dollar après la chaîne (ou avant, selon les paramètres régionaux).

Convertir un entier en nombre binaire, octal et hexadécimal

```
int intValue = 24;
String binaryStr = Integer.toBinaryString(intValue);
String octalStr = Integer.toOctalString(intValue);
String hexStr = Integer.toHexString(intValue);
```

La classe `Integer` permet aisément de convertir une valeur d'entier en un nombre binaire, octal ou hexadécimal. Les méthodes statiques concernées de la classe `Integer` sont les méthodes `toBinaryString()`, `toOctalString()` et `toHexString()`. Dans cet exemple, nous les utilisons chacune en passant dans chaque cas une valeur d'entier et en obtenant en retour un objet `String` contenant respectivement l'entier au format binaire, octal et hexadécimal.

Générer des nombres aléatoires

```
Random rn = new Random();
int value = rn.nextInt();
double dvalue = rn.nextDouble();
```

La classe `Random` du paquetage `java.util` peut être utilisée pour générer des nombres aléatoires. Par défaut, elle utilise l'heure courante du jour comme valeur de graine pour son générateur de nombres aléatoires. Vous pouvez aussi définir vous-même une valeur de graine en la passant comme paramètre au constructeur de `Random`. La méthode `nextInt()` produit un nombre aléatoire entier 32 bits.

L'autre moyen de générer des nombres aléatoires consiste à utiliser la méthode `random()` de la classe `Math` dans le paquetage `java.lang`.

```
double value = Math.random();
```

Cette méthode retourne une valeur `double` avec un signe positif, supérieure ou égale à 0,0 et inférieure à 1,0. Pour générer une valeur comprise dans un intervalle spécifique, vous pouvez ajouter la limite inférieure au résultat de `Math.random()` et multiplier par l'intervalle.

Le code suivant produit ainsi un nombre aléatoire compris entre 5 et 20 :

```
double value = (5+Math.random())*15;
```

La classe `Random` et la méthode `random()` de la classe `Math` fournissent en fait un nombre pseudo-aléatoire et non un véritable nombre aléatoire, car ce nombre est généré en utilisant une formule mathématique et une valeur de graine d'entrée. Si l'on connaît la valeur de graine et le mécanisme interne de la classe `Random`, il est possible de prédire la valeur obtenue. Ces classes ne constituent donc généralement pas une bonne solution pour les générateurs de nombres aléatoires à utiliser dans les applications fortement sécurisées. Dans la plupart des cas cependant, il s'agit de générateurs de nombres aléatoires parfaitement acceptables.

Calculer des fonctions trigonométriques

```
// Calcul du cosinus
double cosine = Math.cos(45);
// Calcul du sinus
double sine = Math.sin(45);
// Calcul de la tangente
double tangent = Math.tan(45);
```

La classe `Math` du paquetage `java.lang` contient des méthodes permettant de calculer aisément toutes les fonctions trigonométriques. Cet exemple montre comment il est possible de récupérer le cosinus, le sinus et la tangente d'un angle donné. La classe `Math` possède également des

méthodes pour calculer l'arc cosinus, l'arc sinus et l'arc tangente ainsi que le sinus, le cosinus et la tangente hyperboliques. Chacune de ces méthodes prend un unique paramètre de type `double` en entrée et retourne un résultat de type `double`.

Calculer un logarithme

```
double logValue = Math.log(125.5);
```

Cet exemple utilise la méthode `log()` de la classe `java.lang.Math` pour calculer le logarithme du paramètre passé. Nous passons une valeur de type `double` et la valeur de retour est également un `double`. La méthode `log()` calcule le logarithme naturel de base *e*, où *e* correspond à la valeur standard de 2,71828.

Le JDK 1.5 a ajouté une nouvelle méthode à la classe `Math` afin de calculer directement un algorithme de base 10 : `log10()`. Cette méthode, analogue à `log()`, prend en entrée un paramètre `double` et retourne un `double`. Elle peut aisément être utilisée pour calculer un logarithme de base 10, comme ceci :

```
double logBase10 = Math.log10(200);
```

8

Entrée et sortie

Dans la plupart des cas, l'entrée et la sortie constituent le but ultime des applications. Les programmes seraient parfaitement inutiles s'ils ne pouvaient produire en sortie des résultats ni récupérer en entrée des données à traiter fournies par l'utilisateur ou l'ordinateur. Dans ce chapitre, nous présenterons quelques exemples de base pour les opérations d'entrée et de sortie.

Les paquetages `java.io` et `java.util` abritent la plupart des classes utilisées dans ce chapitre pour les tâches d'entrée et de sortie. Nous verrons comment lire et écrire des fichiers, travailler avec des archives ZIP, formater la sortie et travailler avec les flux de système d'exploitation standard.

A mesure que vous lisez les exemples de ce chapitre, gardez à l'esprit que bon nombre des exercices peuvent lever des exceptions dans n'importe quel programme réel, telle l'exception `java.io.IOException`. Dans les exemples, nous n'inclurons pas le code de gestion des exceptions. Dans une application réelle, il est cependant impératif qu'il soit présent.

Lire du texte à partir d'une entrée standard

```
BufferedReader inStream =
    new BufferedReader (new InputStreamReader(System.in));
String inLine = "";
while ( !(inLine.equalsIgnoreCase("quit"))) {
  System.out.print("prompt> ");
  inLine = inStream.readLine();
}
```

Dans un programme de console, il est courant de lire l'entrée standard qui provient le plus souvent de la ligne de commande. Cet exemple montre comment lire l'entrée dans une variable `String` Java. Le Java contient trois flux connectés aux flux du système d'exploitation. Il s'agit des flux standard d'entrée, de sortie et d'erreur. Ils sont respectivement définis en Java par `System.in`, `System.out` et `System.err`. Ils peuvent être utilisés pour lire ou écrire vers et depuis l'entrée et la sortie standard du système d'exploitation.

Dans cet exemple, nous créons un `BufferedReader` afin de lire le flux `System.in`. Nous lisons avec ce lecteur les lignes de l'entrée standard en poursuivant jusqu'à ce que l'utilisateur tape le mot `"quit"`.

Ecrire vers une sortie standard

```
System.out.println("Hello, World!");
```

`System.out` est un `PrintStream` qui écrit sa sortie vers la sortie standard (en général, la console). `System.out` est l'un des trois flux défini par le Java pour se connecter aux flux

de système d'exploitation standard. Les autres flux sont `System.in` et `System.err`, permettant de lire l'entrée standard et d'écrire dans le flux d'erreur standard.

Le flux `System.out` est probablement le flux de système d'exploitation standard le plus fréquemment utilisé. Il est mis à contribution par la quasi-totalité des programmeurs pour le débogage de leurs applications. Ce flux écrit dans la console : il s'agit ainsi d'un outil pratique pour voir ce qui se passe à un point particulier de l'application.

En général, les instructions `System.out` ne doivent cependant pas être conservées dans les programmes après le débogage initial, car elles peuvent en affecter les performances. A long terme, il est préférable de récolter les informations de débogage dans votre application à l'aide d'un dispositif de journalisation comme celui que proposent `java.util.logging` ou le paquetage populaire Log4J d'Apache.

Formater la sortie

```
float hits=3;
float ab=10;
String formattedTxt =
    String.format("Batting average: %.3f", hits/ab);
```

Cet exemple utilise la méthode `format()` pour formater une chaîne de sortie qui imprime une moyenne à la batte en baseball sous sa forme classique à trois chiffres après la virgule. La moyenne est définie en divisant le nombre de coups réussis par le nombre de "présences à la batte" *(at-bats)*. Le spécificateur de format `%.3f` demande au formateur d'imprimer la moyenne sous forme de nombre à virgule flottante avec trois chiffres après la virgule.

Le JDK 1.5 a introduit la classe `java.util.Formatter` qui peut être utilisée pour simplifier le formatage du texte. La classe `Formatter` opère à la manière de la fonction `printf` du langage C et offre une prise en charge de la justification et de l'alignement pour la mise en page, des formats courants pour les données numériques, les chaînes et les dates et heures, ainsi qu'une sortie spécifique en fonction des paramètres régionaux.

Voici un exemple d'utilisation directe de la classe `Formatter` :

```
StringBuffer buffer = new StringBuffer();
Formatter formatter = new Formatter(buffer, Locale.FRANCE);
formatter.format("Value of PI: %6.4f", Math.PI);
System.out.println(buffer.toString());
```

Ce code produit la sortie suivante :

```
Value of PI: 3,1416
```

Dans cet exemple, nous créons une instance `Formatter` et l'utilisons pour formater la valeur mathématique standard du nombre pi. Celle-ci contient un nombre infini de chiffres après la virgule, mais il est généralement préférable d'en restreindre le nombre au moment de l'imprimer.

Dans cet exemple, nous avons utilisé le spécificateur de format `%6.4f`. La valeur `6` indique que la sortie pour ce nombre ne doit pas dépasser 6 caractères de longueur en comptant la virgule. La valeur `4` indique que la précision de la valeur décimale doit être de 4 chiffres après la virgule. La valeur imprimée atteint donc 6 caractères de longueur et possède 4 chiffres après la virgule : `3,1416`.

En plus d'utiliser directement la classe `Formatter`, vous pouvez utiliser les méthodes `format()` et `printf()` des flux `System.out` et `System.err`. Voici par exemple comment

imprimer l'heure locale en utilisant la méthode `format()` du flux `System.out` :

```
System.out.format("Local time: %tT",
Calendar.getInstance());
```

Cette méthode imprime l'heure locale, comme ceci :

```
Local time: 16:25:14
```

La classe `String` contient également une méthode statique `format()` qui peut être utilisée pour formater directement des chaînes. Nous pouvons par exemple utiliser cette méthode statique pour formater aisément une chaîne de date comme ceci :

```
Calendar c = new GregorianCalendar(1999,
➥Calendar.JANUARY, 6);
String s = String.format("Timmy's Birthday: %1$tB %1$te,
➥%1$tY", c);
```

Ce code crée la valeur `String` formatée suivante :

```
Timmy's Birthday: Janvier 6, 1999
```

Toutes les méthodes qui produisent une sortie formatée dont nous avons traité prennent une chaîne de format et une liste d'arguments en paramètres. La chaîne de format est un objet `String` qui peut contenir du texte et un ou plusieurs spécificateurs de format.

Pour notre exemple de formatage précédent, la chaîne de format serait `"Timmy's Birthday: %1$tm %1$te,%1$tY"`, les éléments `%1$tm`, `%1$te` et `$1$tY` étant les spécificateurs de format.

Le reste de la chaîne correspond à du texte statique. Ces spécificateurs de format indiquent comment les arguments doivent être traités et l'endroit de la chaîne où ils doivent être placés.

Pour faire référence à notre exemple de nouveau, la liste d'arguments correspond simplement à l'objet `Calendar c`. Dans cet exemple, nous n'avons qu'un seul argument, mais la liste peut en contenir plusieurs. Tous les paramètres passés aux méthodes de formatage après la chaîne de format sont considérés comme des arguments.

Les spécificateurs de format possèdent le format suivant :

```
%[index_argument$][drapeaux][largeur][.précision]
➥conversion
```

`index_argument` fait référence à un argument dans la liste des arguments passée à la méthode de formatage. La liste est indexée en commençant à 1. Pour faire référence au premier argument, vous devez donc utiliser `1$`.

L'élément *`drapeaux`* désigne un ensemble de caractères qui modifient le format de sortie. L'ensemble de drapeaux valides dépend du type de conversion.

L'élément *`largeur`* est un entier décimal positif indiquant le nombre minimal de caractères à écrire dans la sortie.

L'élément *`précision`* est un entier décimal positif normalement utilisé pour restreindre le nombre de caractères. Le comportement spécifique dépend du type de conversion.

L'élément *`conversion`* est un caractère indiquant comment l'argument doit être formaté. L'ensemble des conversions valides pour un argument donné dépend du type de données de l'argument.

Tous les éléments spécificateurs sont facultatifs à l'exception du caractère de conversion.

Le Tableau 8.1 présente une liste des caractères de conversion valides. Pour plus d'informations sur les conversions de date et d'heure, référez-vous au JavaDoc de la classe `Formatter` à l'adresse suivante : **http://java.sun.com/j2se/1.5.0/docs/api/**.

Tableau 8.1 : Codes de format de `Formatter`

Code	Description
`b`	Si l'argument *`arg`* est `null`, le résultat est `false`. Si *`arg`* est un `boolean` ou un `Boolean`, le résultat est la chaîne retournée par `String.valueOf()`. Sans cela, le résultat est `true`.
`h`	Si l'argument *`arg`* est `null`, le résultat est `"null"`. Sans cela, le résultat est obtenu en invoquant `Integer.toHexString(arg.hashCode())`.
`s`	Si l'argument *`arg`* est `null`, le résultat est `"null"`. Si *`arg`* implémente `Formattable`, `arg.formatTo` est invoquée. Sans cela, le résultat est obtenu en invoquant `arg.toString()`.
`c`	Le résultat est un caractère Unicode.
`d`	Le résultat est formaté sous forme d'entier décimal.
`o`	Le résultat est formaté sous forme d'entier octal.
`x`	Le résultat est formaté sous forme d'entier hexadécimal.
`f`	Le résultat est formaté comme un nombre décimal.
`e`	Le résultat est formaté sous forme de nombre décimal en notation scientifique informatisée.
`g`	Le résultat est formaté en utilisant une notation scientifique informatisée ou le format décimal, selon la précision et la valeur après l'arrondi.
`a`	Le résultat est formaté sous forme de nombre hexadécimal à virgule flottante avec une mantisse et un exposant.
`t`	Préfixe pour les caractères de conversion de date et d'heure.
`n`	Le résultat est le séparateur de ligne spécifique à la plate-forme.
`%`	Le résultat est un `%` littéral.

Pour obtenir une liste complète des codes de format disponibles, référez-vous à la documentation JavaDoc de la classe `Formatter` qui peut être consultée à l'adresse **http://java.sun.com/j2se/1.5.0/docs/api/java/util/Formatter.html**.

Ouvrir un fichier par son nom

```
// Ouvrir un fichier en lecture
BufferedReader is =
    new BufferedReader(new FileReader("file.txt"));
// Ouvrir un fichier en écriture
BufferedWriter out =
    new BufferedWriter(new FileWriter("afile.txt"));
```

Cet exemple montre comment créer un `BufferedReader` pour lire l'entrée d'un fichier spécifié par un nom de fichier (ici, `file.txt`) et comment créer un `BufferedWriter` pour écrire du texte vers un fichier de sortie spécifié par son nom (`afile.txt`).

Il est très facile d'ouvrir un fichier désigné par son nom en Java. La plupart des classes de flux d'entrée et de sortie ou de lecteur possèdent une option permettant de spécifier le fichier par nom dans le flux ou le constructeur du lecteur.

Lire un fichier dans un tableau d'octets

```
File file = new File(fileName);
InputStream is = new FileInputStream(file);
long length = file.length();
byte[] bytes = new byte[(int)length];
int offset = 0;
int numRead = 0;
while ((offset < bytes.length)
 && ((numRead=is.read(bytes, offset,
  bytes.length-offset))>= 0)) {
  offset += numRead;
}
is.close();
```

Cet exemple lit le fichier spécifié par `fileName` dans le tableau d'octets `bytes`. Notez que la méthode `file.length()` retourne la longueur du fichier en octets sous forme de valeur `long`, mais nous devons utiliser une valeur `int` pour initialiser le tableau d'octets. Nous transtypons donc d'abord la valeur `long` en une valeur `int`. Dans un véritable programme, il conviendrait préalablement de s'assurer que la valeur de longueur tient effectivement dans un type `int` avant de la transtyper à l'aveuglette. Avec la méthode `read()` du `InputStream`, nous continuons à lire les octets du fichier jusqu'à ce que le tableau d'octets soit rempli ou qu'il n'y ait plus d'octets à lire dans le fichier.

Lire des données binaires

```
InputStream is = new FileInputStream(fileName);
int offset = 0;
int bytesRead = is.read(bytes, offset, bytes.length-offset);
```

La méthode `read()` permet de lire des données binaires depuis un fichier dans un tableau d'octets. Dans cet exemple, nous lisons le flux d'entrée `is` dans le tableau d'octets `bytes`. Le tableau `bytes` est ici supposé avoir été précédemment initialisé sous forme de tableau d'octets et la variable `fileName` correspondre au nom d'un fichier valide. La variable `offset` pointe l'emplacement du tableau d'octets à partir duquel l'écriture des données doit être commencée. Elle est utile lorsque vous vous trouvez dans une boucle, que vous lisez les données d'un fichier et que vous ne souhaitez pas écraser les données précédemment stockées dans le tableau d'octets. A chaque passage dans la boucle, vous pouvez mettre à jour ce décalage, comme nous l'avons vu dans l'exemple précédent, "Lire un fichier dans un tableau d'octets". Voici la portion de code concernée :

```
while ( (offset < bytes.length)
    && ( (numRead=is.read(bytes, offset,
    ➥bytes.length-offset)) >= 0) ) {
    offset += numRead;
}
```

Dans cet exemple de code, nous écrivons des données depuis le flux d'entée `is` dans le tableau `bytes`. Nous poursuivons la lecture depuis le fichier jusqu'à ce que nous ayons rempli le tableau `bytes` ou qu'il n'y ait plus de données à lire dans le fichier.

Atteindre une position dans un fichier

```
File file = new File("somefile.bin");
RandomAccessFile raf = new RandomAccessFile(file, "rw");
raf.seek(file.length());
```

La méthode `seek()` de la classe `RandomAccessFile` permet d'atteindre n'importe quelle position désirée dans un fichier. Dans cet exemple, nous créons d'abord un objet `File`, qui est ensuite utilisé pour créer une instance `RandomAccessFile`. Avec l'instance `RandomAccessFile` (`raf`), nous recherchons la fin du fichier en passant la valeur `file.length()` en paramètre à la méthode `seek()`.

Après avoir utilisé la méthode `seek()` pour trouver la position désirée dans le fichier, nous pouvons ensuite utiliser les méthodes `read()` ou `write()` de la classe `RandomAccessFile` pour lire ou écrire des données à partir de cette position exacte.

Lire une archive JAR ou ZIP

```
// Lire un fichier ZIP
ZipFile file = new ZipFile(filename);
Enumeration entries = file.entries();
while (entries.hasMoreElements()) {
  ZipEntry entry = (ZipEntry)entries.nextElement();
  if (entry.isDirectory()) {
    // Traiter le répertoire
  }
  else {
    // Traiter le fichier
  }
}
file.close();
```

Le Java offre un support intégré pour la lecture et l'écriture de fichiers d'archive ZIP. Les fichiers ZAR n'étant autres que des fichiers ZIP possédant un contenu précis, les classes et les méthodes des fichiers ZIP peuvent également être utilisées pour les lire. Les classes ZIP sont contenues dans le paquetage `java.util.zip` qui fait partie du JDK standard.

Dans cet exemple, nous créons d'abord un objet `ZipFile` en passant le nom de fichier d'un fichier ZIP existant au constructeur de la classe `ZipFile`. Nous obtenons ensuite l'ensemble des entrées du fichier ZIP dans un type d'énumération en appelant la méthode `entries()` de l'objet `ZipFile`. Une fois en possession des entrées de fichier ZIP sous forme d'énumération, nous pouvons parcourir au pas à pas les entrées et instancier un objet `ZipEntry` pour chaque entrée. Avec l'objet `ZipEntry`, nous pouvons déterminer si l'entrée particulière qui est traitée est un répertoire ou un fichier et la traiter en fonction.

Créer une archive ZIP

```
// Ecrire un fichier ZIP
ZipOutputStream out =
 ➥new ZipOutputStream(new FileoutputStream(zipFileName));
FileInputStream in = new FileInputStream(fileToZip1);
out.putNextEntry(new ZipEntry(fileToZip1));
int len;
byte[] buf = new byte[1024];
while ((len = in.read(buf)) > 0) {
  out.write(buf,0,len);
}
out.closeEntry();
in.close();
out.close();
```

Dans l'exemple précédent, nous avons vu comment lire dans un fichier ZIP. Cette fois, nous créons un fichier ZIP. Pour cela, nous commençons par construire un `ZipOutputStream` en passant à son constructeur un objet `FileOutputStream` pointant vers le fichier que nous souhaitons compresser sous forme de fichier ZIP. Ensuite, nous créons un `FileInputStream` pour le fichier que nous souhaitons ajouter à notre archive ZIP. Nous utilisons la méthode `putNextEntry()` du `ZipOutputStream` pour ajouter le fichier à l'archive.

La méthode `putNextEntry()` prend un objet `ZipEntry` en entrée : nous devons donc construire le `ZipEntry` à partir du nom du fichier que nous ajoutons à notre archive. Dans une boucle `while`, nous lisons ensuite notre fichier en utilisant le `FileInputStream` et l'écrivons dans le `ZipOutputStream`. Une fois cela fait, nous fermons l'entrée en utilisant la méthode `closeEntry()` du `ZipOutputStream`, puis fermons chacun de nos flux ouverts.

Dans cet exemple, nous n'avons ajouté qu'un seul fichier à notre archive ZIP, mais ce code peut aisément être étendu afin d'ajouter autant de fichiers que nécessaire à l'archive. La classe `ZipOutputStream` accepte aussi bien les entrées compressées que les entrées non compressées.

9

Travailler avec des répertoires et des fichiers

L'une des tâches courantes dans la plupart des applications Java consiste à travailler avec le système de fichiers et notamment ses répertoires et ses fichiers. Ce chapitre présente un certain nombre d'exemples destinés à vous aider à travailler avec des fichiers et des répertoires en Java.

La principale classe que nous utiliserons pour ces exemples est la classe `java.io.File`. Elle permet de lister, créer, renommer et supprimer des fichiers, mais encore de travailler avec des répertoires.

Bon nombre des exemples de ce chapitre peuvent lever une exception `SecurityException`. En Java, le système de fichiers est protégé par le gestionnaire de sécurité. Pour certaines applications, il peut falloir en utiliser une implémentation personnalisée. Parmi les applications Java, les applets sont les plus restreintes en ce qui concerne l'accès

aux fichiers et aux répertoires sur l'ordinateur local de l'utilisateur. En tirant parti du gestionnaire de sécurité et du framework de stratégie de sécurité lié vous pouvez contrôler précisément l'accès aux fichiers et aux répertoires. Pour plus d'informations sur les options de sécurité disponibles en Java, consultez la documentation de sécurité disponible (en anglais) sur le site Web Java officiel à l'adresse **http://java.sun.com/javase/technologies/security.jsp**.

Pour plus d'informations sur le gestionnaire de sécurité, consultez le didacticiel suivant (en anglais) proposé par Sun : **http://java.sun.com/docs/books/tutorial/essential/system/securityIntro.html**.

Créer un fichier

```
File f = new File("myfile.txt");
boolean result = f.createNewFile();
```

Cet exemple utilise la méthode `createNewFile()` pour créer un nouveau fichier portant le nom spécifié en paramètre (ici, `myfile.txt`) en construisant l'objet `File`. La méthode `createNewFile()` retourne la valeur booléenne `true` si le fichier a bien été créé et `false` si le nom de fichier spécifié existe déjà.

La classe `File` propose une autre méthode statique pour créer un fichier temporaire : `createTempFile()`. L'exemple suivant montre comment l'utiliser pour créer un fichier temporaire :

```
File tmp = File.createTempFile("temp", "txt", "/temp");
```

Les paramètres que nous passons à la méthode `createTempFile()` sont le préfixe du fichier temporaire, son suffixe et son répertoire. Il existe aussi une autre version de cette méthode qui ne prend que deux paramètres et utilise le répertoire temporaire par défaut. Le fichier spécifié doit déjà exister pour que l'une ou l'autre forme des méthodes `createTempFile()` puisse fonctionner.

Si vous utilisez des fichiers temporaires, la méthode `deleteOnExit()` de la classe `File` peut aussi vous intéresser. Elle doit être appelée sur un objet `File` qui représente un fichier temporaire. L'appel de la méthode `deleteOnExit()` requiert que le fichier soit automatiquement supprimé lorsque la machine virtuelle Java se ferme.

Renommer un fichier ou un répertoire

```
File f = new File("myfile.txt");
File newFile = new File("newname.txt");
boolean result = f.renameTo(newFile);
```

Dans cet exemple, nous renommons le fichier `myfile.txt` en l'appelant `newname.txt`. Pour cela, nous devons créer deux objets `File`. Le premier est construit avec le nom courant du fichier. Ensuite, nous créons un nouvel objet `File` en utilisant le nom de remplacement que nous souhaitons donner au fichier. Nous appelons la méthode `renameTo()` de l'objet `File` d'origine et lui passons l'objet `File` spécifiant le nouveau nom de fichier. La méthode `renameTo()` retourne la valeur booléenne `true` si l'opération de modification du nom réussit et `false` sinon, quelle que soit la raison.

Cette technique peut également être utilisée pour renommer un répertoire. Le code est alors exactement le même, à la différence que nous passons cette fois les noms de répertoire aux constructeurs de l'objet `File` au lieu des noms de fichier. Voici comment procéder :

```
File f = new File("directoryA");
File newDirectory = new File("newDirectory");
boolean result = f.renameTo(newDirectory);
```

Rappelez-vous que le nouveau nom de fichier ou de répertoire doit être spécifié dans un objet `File` passé à la méthode `renameTo()`. L'une des erreurs courantes consiste à tenter de passer un objet `String` contenant le nouveau nom de fichier ou de répertoire à la méthode `renameTo()`. Une erreur de compilation est générée si un objet `String` est passé à la méthode `renameTo()`.

Supprimer un fichier ou un répertoire

```
File f = new File("somefile.txt");
boolean result = f.delete();
```

La classe `File` permet aisément de supprimer un fichier. Dans cet exemple, nous créons d'abord un objet `File` en spécifiant le nom du fichier à supprimer. Ensuite, nous appelons la méthode `delete()` de l'objet `File`. Elle retourne la valeur booléenne `true` si le fichier a bien été supprimé et `false` sinon.

La méthode `delete()` peut aussi être utilisée pour supprimer un répertoire. Dans ce cas, vous devez créer l'objet

`File` en spécifiant le nom du répertoire au lieu d'un nom de fichier, comme ceci :

```
File directory = new File("files/images");
directory.delete();
```

Le répertoire n'est supprimé que s'il est vide. S'il ne l'est pas, la méthode `delete()` retourne la valeur booléenne `false`. Si le fichier ou le répertoire que vous essayez de supprimer n'existe pas, `delete()` retourne aussi `false`, sans lever d'exception.

La classe `File` propose une autre méthode utile liée à la suppression des fichiers et des répertoires : `deleteOnExit()`. Lorsqu'elle est appelée, le fichier ou le répertoire représenté par l'objet `File` sont automatiquement supprimés lorsque la machine virtuelle Java se ferme.

Modifier des attributs de fichier

```
File f = new File("somefile.txt");
boolean result = f.setReadOnly();
long time = (new Date()).getTime();
result = f.setLastModified(time);
```

L'objet `File` permet aisément de modifier l'horodatage de dernière modification et l'état de lecture/écriture d'un fichier. Pour réaliser ces tâches, vous devez utiliser les méthodes `setReadOnly()` et `setLastModified()` de la classe `File`. La méthode `setReadOnly()` positionne en lecture seule le fichier sur lequel elle est appelée. La méthode `setLastModified()` prend un unique paramètre en entrée spécifiant une date en millisecondes et positionne l'horodatage de dernière modification du fichier à cette date.

La valeur temporelle passée est mesurée en millisecondes écoulées depuis l'époque UNIX (1er janvier 1970, 00 h 00 m 00 s, GMT). Ces deux méthodes retournent la valeur booléenne `true` uniquement si l'opération réussit. Si l'opération échoue, elles retournent `false`.

Obtenir la taille d'un fichier

```
File file = new File("infilename");
long length = file.length();
```

Dans cet exemple, nous retrouvons la taille d'un fichier en utilisant la méthode `length()` de l'objet `File`. Cette méthode retourne la taille du fichier en octets. Si le fichier n'existe pas, elle retourne 0.

Cette méthode est souvent utile avant de lire un fichier sous forme de tableau d'octets. Grâce à la méthode `length()`, vous pouvez déterminer la longueur du fichier afin de connaître la taille requise pour que le tableau d'octets contienne la totalité du fichier. Le code suivant est ainsi souvent utilisé pour lire un fichier dans un tableau d'octets :

```
File myFile = new File("myfile.bin");
InputStream is = new FileInputStream(myFile);
// Obtenir la taille du fichier
long length = myFile.length();
if (length > Integer.MAX_VALUE) {
    // Le fichier est trop grand
}
byte[] bytes = new byte[(int)length];
int offset = 0;
int numRead = 0;
```

```
while (offset < bytes.length
    && (numRead=is.read(bytes, offset,
    ➥bytes.lengthoffset)) >= 0) {
    offset += numRead;
}
is.close();
```

Déterminer si un fichier ou un répertoire existe

```
boolean exists = (new File("filename")).exists();
if (exists) {
  // Le fichier ou le répertoire existe
}
else {
  // Le fichier ou le répertoire n'existe pas
}
```

Cet exemple utilise la méthode `exists()` de l'objet `File` pour déterminer si le fichier ou le répertoire représenté par cet objet existe. Elle retourne `true` si le fichier ou le répertoire existe et `false` sinon.

Déplacer un fichier ou un répertoire

```
File file = new File( "filename ");
File dir = new File( "directoryname ");
boolean success =
    ➥file.renameTo(new File(dir, file.getName()));
if (!success) {
  // Le fichier n'a pas pu être déplacé
}
```

La méthode `renameTo()` de la classe `File` permet de déplacer un fichier ou un répertoire dans un autre répertoire. Dans cet exemple, nous créons un objet `File` afin de représenter le fichier ou le répertoire à déplacer. Nous créons un autre objet `File` représentant le répertoire de destination dans lequel nous souhaitons déplacer le fichier ou le répertoire, puis appelons la méthode `renameTo()` du fichier déplacé en lui passant un unique paramètre d'objet `File`. L'objet `File` passé en paramètre est construit en utilisant le répertoire de destination et le nom de fichier d'origine. Si l'opération de déplacement réussit, la méthode `renameTo()` retourne la valeur booléenne `true`. En cas d'échec, elle retourne `false`.

Lorsque vous utilisez la méthode `renameTo()`, gardez à l'esprit que bien des aspects de ce comportement dépendent de la plate-forme d'exécution. Certains sont signalés dans le JavaDoc pour cette méthode, et notamment les suivants :

- L'opération `renameTo` peut ne pas être capable de déplacer un fichier d'un système de fichiers à un autre.
- L'opération `renameTo` peut ne pas être atomique. Autrement dit, l'implémentation de l'opération `renameTo` peut se décomposer en plusieurs étapes au niveau du système d'exploitation : cette particularité peut poser problème en cas d'incident, comme lors d'une coupure de courant survenant entre les étapes.
- L'opération `renameTo` peut échouer si un fichier possédant le nom du chemin abstrait de destination existe déjà.

Lorsque vous utilisez cette méthode, vérifiez toujours la valeur de retour afin de vous assurer que l'opération a réussi.

Obtenir un chemin de nom de fichier absolu à partir d'un chemin relatif

```
File file = new File( "somefile.txt ");
File absPath = file.getAbsoluteFile();
```

Cet exemple retrouve le chemin absolu d'un fichier dont le chemin relatif est spécifié. Le chemin absolu définit le chemin complet du fichier en commençant à partir du répertoire racine du système de fichiers, comme ceci :

`c:\project\book\somefile.txt.`

Le nom de fichier relatif spécifie le nom et le chemin du fichier par rapport au répertoire courant, comme `somefile.txt` si le répertoire courant est `c:\project\book`. Dans bien des cas, le nom de chemin relatif n'est constitué que du nom de fichier. La méthode `getAbsoluteFile()` de la classe `File` retourne une objet `File` représentant le nom de fichier absolu pour le fichier représenté par l'objet `File` sur lequel elle est appelée.

Une autre méthode similaire, `getAbsolutePath()`, retourne le chemin absolu sous forme de `String` et non d'objet `File`. Le code suivant présente cette méthode :

```
File file = new File( "filename.txt ");
String absPath = file.getAbsolutePath();
```

Dans cet exemple, `absPath` contient la chaîne `"c:\project\book\somefile.txt"`.

Déterminer si un chemin de nom de fichier correspond à un fichier ou à un répertoire

```
File testPath = new File( "directoryName ");
boolean isDir = testPath.isDirectory();
if (isDir) {
  // testPath est un répertoire
}
else {
  // testPath est un fichier
}
```

Cet exemple détermine si l'objet `File` désigné représente un fichier ou un répertoire. La méthode `isDirectory()` de la classe `File` retourne `true` si l'objet `File` sur lequel elle est appelée représente un répertoire et `false` s'il représente un fichier. Elle est utile lorsque vous souhaitez parcourir l'ensemble des fichiers et sous-répertoires d'un répertoire donné. Par exemple, vous pourriez souhaiter écrire une méthode qui liste tous les fichiers d'un répertoire et parcoure de manière récurrente chacun de ses sous-répertoires.

La méthode `isDirectory()` peut être utilisée lorsque vous parcourez la liste des éléments contenus dans chaque répertoire afin de déterminer s'il s'agit d'un fichier ou d'un répertoire.

Voici un exemple de ce type de méthode qui utilise la méthode `isDirectory()` :

```
static void listAllFiles(File dir) {
    String[] files = dir.list();
    for (int i = 0; i < files.length; i++) {
      File f = new File(dir, files[i]);
```

```
    if (f.isDirectory()) {
      listAllFiles(f);
    }
    else {
      System.out.println(f.getAbsolutePath());
    }
  }
}
```

Si vous appelez cette méthode et lui passez un objet `File` représentant un répertoire, elle imprime les chemins complets de tous les fichiers contenus dans le répertoire et dans l'ensemble de ses sous-répertoires.

La classe `File` contient aussi une méthode `isFile()` qui retourne `true` si l'objet `File` sur lequel elle est appelée représente un fichier et `false` sinon.

Lister un répertoire

```
File directory = new File("users/tim");
String[] result = directory.list();
```

La classe `File` peut aussi être utilisée pour lister le contenu d'un répertoire. Cet exemple utilise la méthode `list()` de la classe `File` pour obtenir un tableau d'objets `String` contenant tous les fichiers et sous-répertoires contenus dans le répertoire spécifié par l'objet `File`. Si le répertoire n'existe pas, la méthode retourne `null`. Les chaînes retournées sont des noms de fichiers et des noms de répertoire simples et non des chemins complets. L'ordre des résultats n'est pas garanti.

La méthode `list()` possède une autre implémentation qui prend un paramètre `java.io.FilenameFilter` et permet de filtrer les fichiers et répertoires retournés dans les résultats de la méthode. En voici un exemple :

```
File directory = new File("users/tim");
FilenameFilter fileFilter = new HTMLFileFilter();
String[] result = directory.list(fileFilter);
```

Voici maintenant l'implémentation correspondante de la classe `HTMLFileFilter` :

```
class HTMLFileFilter extends FilenameFilter {
    public boolean accept(File f) {
     return f.isDirectory() || f.getName()
     .toLowerCase() .endsWith(".html"));
    }
    public String getDescription() }
     return "/html files";
}
```

`FilenameFilter` est une interface définissant une méthode nommée `accept()`. Celle-ci prend deux paramètres : un objet `File` et un objet `String`. L'objet `File` spécifie le répertoire dans lequel le fichier a été trouvé et l'objet `String` spécifie le nom du fichier. La méthode `accept()` retourne `true` si le fichier doit être inclus dans la liste et `false` sinon. Dans cet exemple, nous avons créé un filtre qui amène la méthode `list()` à n'inclure que les fichiers qui se terminent par l'extension `.html`.

Si le filtre passé est `null`, la méthode se comporte comme la précédente méthode `list()` sans paramètre.

En plus des méthodes `list()`, la classe `File` propose deux versions d'une méthode appelée `listFiles()`.

`listFiles()` retourne un tableau d'objets `File` au lieu d'un tableau de chaînes. L'exemple utilise sa variante sans paramètre :

```
File directory = new File("users/tim");
File[] result = directory.listFiles();
```

Les objets `File` résultants contiennent des chemins relatifs ou absolus selon l'objet `File` depuis lequel la méthode `listFiles()` a été appelée. Si l'objet `File` de répertoire dans cet exemple contient un chemin absolu, le résultat contient des chemins absolus. Si l'objet `File` de répertoire contient un chemin relatif, les résultats sont des chemins relatifs.

L'autre version de `listFiles()` prend un paramètre `FileFilter`, de manière analogue à l'exemple présenté pour la méthode `list()`. En voici un exemple :

```
File directory = new File("users/tim");
FileFilter fileFilter = new HTMLFileFilter();
String[] result = directory.listFiles(fileFilter);
```

Voici maintenant l'implémentation correspondante de la classe `HTMLFileFilter` :

```
class HTMLFileFilter extends FileFilter {
    public boolean accept(File f) {
      return f.isDirectory() ||
      ➥f.getName().toLowerCase().endsWith(".html");
    }
    public String getDescription() {
      return ".html files";
    }
}
```

`FileFilter` est une interface définissant deux méthodes, `accept()` et `getDescription()`. A la différence de la méthode `accept()` de `FilenameFilter`, la méthode `accept()` de `FileFilter` ne prend qu'un paramètre, un objet `File`. L'objet `File` spécifie un fichier ou un répertoire. La méthode `accept()` retourne `true` si le fichier ou le répertoire doivent être inclus dans la liste et `false` sinon. Dans cet exemple, nous avons créé un filtre qui amène la méthode `list()` à n'inclure que les répertoires ou les fichiers qui se terminent par l'extension `.html`.

Créer un nouveau répertoire

```
boolean success = (new File("users/tim")).mkdir();
```

Cet exemple utilise la méthode `mkdir()` de la classe `File` pour créer un nouveau répertoire. `mkdir()` retourne `true` si le répertoire a pu être créé et `false` sinon. Elle ne crée de répertoire que si tous les répertoires parents spécifiés existent déjà. Ici, il est donc nécessaire que le répertoire `users` existe déjà pour que `mkdir()` parvienne à créer le répertoire `users/tim`.

La méthode `mkdirs()` de la classe `File` est une méthode similaire qui permet de créer un arbre de répertoires complet en générant tous les répertoires parents s'ils n'existent pas. En voici un exemple :

```
boolean success = (new File("/users/tim/Web")).mkdirs( );
```

A l'exécution de cette instruction, la méthode `mkdirs()` créera tous les répertoires (`users`, `tim`, `Web`) qui n'existent pas.

10

Clients réseau

La plupart des applications écrites aujourd'hui requièrent différents types de fonctionnalités réseau. Les applications Java entièrement autonomes se font de plus en plus rares. Ce chapitre sur les clients réseau a donc toutes les chances d'être utile à la plupart des développeurs Java qui conçoivent des applications aujourd'hui.

Les programmes réseau impliquent une communication entre un client et un serveur. En général, le client est l'application qui transmet une demande de contenu ou de services et le serveur une application réseau qui sert ce contenu et ces services à de nombreux clients. Dans ce chapitre, nous nous concentrerons spécifiquement sur le client. Le Chapitre 11, "Serveurs réseau" traite des exemples liés au serveur.

A l'exception d'un exemple concernant la lecture d'une page Web *via* HTTP, les exemples de ce chapitre opèrent tous au niveau de la programmation avec les sockets. Les sockets sont une implémentation réseau de bas niveau. Dans la plupart des cas, vous chercherez à utiliser un protocole situé au niveau juste supérieur à celui de la couche des sockets, comme le HTTP, le SMTP ou le POP.

D'autres API Java ou tierces permettent de travailler avec ces protocoles réseau de plus haut niveau. Le paquetage `java.net` fournit les fonctionnalités de communication réseau côté client que nous utiliserons dans ce chapitre.

La plate-forme J2EE (que nous n'abordons pas dans ce livre) propose de nombreux services réseau supplémentaires dont un support complet du développement Web Java côté serveur. Parmi les technologies réseau incluses dans la plate-forme J2EE, on peut citer les servlets, les EJB et le JMS.

Contacter un serveur

```
String serverName = "www.timothyfisher.com";
Socket sock = new Socket(serverName, 80);
```

Dans cet exemple, nous nous connectons à un serveur *via* TCP/IP à l'aide de la classe Java `Socket`. Lors de la construction de l'instance `sock`, une connexion de socket est opérée au serveur spécifié par `serverName` – ici, `www.timothyfisher.com`, sur le port `80`.

A chaque fois qu'un `Socket` est créé, vous devez veiller à fermer le socket lorsque vous avez terminé en appelant la méthode `close()` sur l'instance `Socket` avec laquelle vous travaillez.

Le Java prend en charge d'autres méthodes de connexion au serveur dont nous ne traiterons pas en détail ici. Par exemple, vous pouvez utiliser la classe `URL` pour ouvrir une URL et la lire. Pour plus de détails sur l'utilisation de la classe `URL`, consultez l'exemple "Lire une page Web *via* HTTP" de ce chapitre.

Retrouver des adresses IP et des noms de domaine

```
// Trouver l'adresse IP d'un domaine
String hostName = www.timothyfisher.com";
String ip =
  InetAddress.getByName(hostName).getHostAddress();
// Trouver le nom de domaine de l'adresse IP
String ipAddress = "66.43.127.5";
String hostName =
  InetAddres.getByName(ipAddress).getHostName();
```

Dans cet exemple, nous récupérons le nom d'hôte correspondant à une adresse IP connue, puis nous récupérons l'adresse IP correspondant à un nom d'hôte distant. Pour ces deux tâches, nous faisons appel à la classe `InetAddress`.

Nous utilisons la méthode statique `getByName()` de la classe `InetAddress` pour créer une instance `InetAddress`. Nous pouvons passer une adresse IP ou un nom d'hôte à la méthode `getByName()` pour créer l'instance `InetAddress`.

Une fois l'instance `InetAddress` créée, nous pouvons appeler la méthode `getHostAddress()` pour retourner l'adresse IP sous forme de `String`. Si nous connaissons déjà l'adresse IP, nous pouvons appeler la méthode `getHostName()` pour retourner le nom d'hôte sous forme de `String`. Si le nom d'hôte ne peut pas être résolu, la méthode `getHostName()` retourne l'adresse IP.

Gérer les erreurs réseau

```
try {
// Connexion à l'hôte réseau
// Réalisation des E/S réseau
}
catch (UnknownHostException ex) {
  System.err.println("Unknown host.");
}
catch (NoRouteToHostException ex) {
  System.err.println("Unreachable host.");
}
catch (ConnectException ex) {
  System.err.println("Connect refused.");
}
catch (IOException ex) {
  System.err.println(ex.getMessage());
}
```

Cet exemple présente les exceptions que vous devez tenter de capturer lorsque vous réalisez des opérations réseau. La première exception que nous essayons de capturer est l'exception `UnknownHostException`. Il s'agit d'une sous-classe d'`IOException`. Elle est levée afin d'indiquer que l'adresse IP d'un hôte ne peut être déterminée.

`NoRouteToHostException` et `ConnectException` sont des sous-classes de `SocketException`. `NoRouteToHostException` signale qu'une erreur s'est produite lors de la tentative de connexion à un socket à une adresse et un port distants. En général, l'hôte distant ne peut pas être atteint en raison d'un problème lié à un pare-feu ou un routeur interposé.

L'exception `ConnectException` est levée si une connexion à l'hôte distant est refusée. `IOException` est une exception de portée plus générale qui peut également être levée à partir d'appels réseau.

Les exemples de ce chapitre et du suivant n'incluent pas de mécanisme de gestion des erreurs. Il convient cependant de capturer ces exceptions dans vos applications Java qui utilisent des fonctionnalités réseau.

Lire du texte

```
BufferedReader in = new BufferedReader
 ➥(new InputStreamReader(socket.getInputStream()));
String text = in.readLine();
```

Cet exemple suppose que vous avez précédemment créé un socket au serveur à partir duquel vous souhaitez lire du texte. Pour plus d'informations sur la création d'une instance de socket, consultez l'exemple "Contacter un serveur" de ce chapitre. Une fois l'instance `socket` obtenue, nous appelons la méthode `getInputStream()` pour obtenir une référence au flux d'entrée du socket. Avec cette référence, nous créons un `InputStreamReader` et l'utilisons pour instancier un `BufferedReader`. Nous lisons enfin le texte sur le réseau avec la méthode `readLine()` du `BufferedReader`.

Cet usage du `BufferedReader` permet d'effectuer une lecture efficace des caractères, des tableaux et des lignes. Si vous cherchez simplement à lire une très petite quantité de données, vous pouvez cependant aussi procéder directement depuis `InputStreamReader`, sans utiliser de `BufferedReader`.

Voici comment lire des données dans un tableau de caractères en n'utilisant qu'un `InputStreamReader` :

```
InputStreamReader in =
     ➥new InputStreamReader(socket.getInputStream()));
String text = in.read(charArray, offset, length);
```

Dans cet exemple, les données sont lues depuis le flux d'entrée dans le tableau de caractères spécifié par `charArray`. Les caractères sont placés dans le tableau en commençant à une position spécifiée par le paramètre de décalage `offset`, et le nombre maximal de caractères lus est spécifié par le paramètre `length`.

Ecrire du texte

```
PrintWriter out =
    ➥new PrintWriter(socket.getOutputStream(), true);
out.print(msg);
out.flush();
```

Cet exemple requiert que vous ayez précédemment créé un socket au serveur à destination duquel vous souhaitez écrire du texte. Pour plus de détails sur la création de l'instance de socket, consultez l'exemple "Contacter un serveur" de ce chapitre. Une fois l'instance de socket obtenue, nous appelons la méthode `getOutputStream()` pour obtenir une référence au flux de sortie du socket. Lorsque la référence est acquise, nous instancions un `PrintWriter` afin d'écrire du texte sur le réseau vers le serveur avec lequel nous sommes connectés. Le second paramètre que nous passons au constructeur `PrintWriter` dans cet exemple définit l'option de purge automatique. La valeur `true` amène les méthodes `println()`, `printf()` et `format()` à vider automatiquement le tampon de sortie. Dans notre exemple, nous utilisons la méthode `print()`. Nous devons donc la faire suivre par un appel à la méthode `flush()` pour forcer l'envoi des données sur le réseau.

Lire des données binaires

```
DataInputStream in =
  new DataInputStream(socket.getInputStream());
in.readUnsignedByte();
```

Cet exemple montre comment lire des données binaires sur un réseau. Il requiert que vous ayez précédemment créé un socket au serveur à partir duquel vous souhaitez lire du texte. Pour plus de détails sur la création de l'instance de socket, consultez l'exemple "Contacter un serveur" de ce chapitre.

Dans cet exemple, nous appelons la méthode `getInputStream()` de l'instance de socket afin d'obtenir une référence au flux d'entrée du socket. En passant le flux d'entrée en paramètre, nous instancions un `DataInputStream`, que nous pouvons utiliser pour lire des données binaires sur le réseau. Nous utilisons la méthode `readUnsignedByte()` pour lire un unique octet non signé sur le réseau.

Si le volume de données que vous lisez est important, il est préférable d'encapsuler le flux d'entrée du socket dans une instance `BufferedInputStream`, comme ceci :

```
DataInputStream in = new DataInputStream(new
➥BufferedInputStream(socket.getInputStream()));
```

Ici, au lieu de passer directement le flux d'entrée du socket au constructeur `DataInputStream`, nous créons d'abord une instance `BufferedInputStream` et la passons au constructeur `DataInputStream`.

Dans cet exemple, nous avons utilisé la méthode `readUnsignedByte()`, mais `DataInputStream` possède bien d'autres méthodes pour lire des données dans n'importe quel type de données Java primitif, dont les suivantes : `read()`,

`readBoolean()`, `readByte()`, `readChar()`, `readDouble()`, `readFloat()`, `readInt()`, `readLong()`, `readShort()`, `readUnsignedByte()` et `readUnsignedShort()`. Consultez le JavaDoc pour plus de détails sur l'utilisation de ces méthodes et d'autres méthodes de la classe `DataInputStream` : **http://java.sun.com/j2se/1.5.0/docs/api/java/io/DataInputStream.html**.

Ecrire des données binaires

```
DataOutputStream out =
    ➥new DataOutputStream(socket.getOutputStream());
out.write(byteArray, 0, 10);
```

Dans l'exemple "Ecrire du texte" vu précédemment, nous avons montré comment écrire des données texte sur un réseau. Cet exemple montre comment écrire des données binaires sur le réseau. Il requiert que vous ayez précédemment créé un socket au serveur à destination duquel vous souhaitez écrire du texte. Pour plus de détails sur la création de l'instance de socket, consultez l'exemple "Contacter un serveur" de ce chapitre.

Dans cet exemple, nous appelons la méthode `getOutputStream()` de l'instance de socket pour obtenir une référence au flux de sortie du socket. Nous instancions ensuite un `DataOutputStream`, que nous pouvons utiliser pour écrire des données binaires sur le réseau. Nous utilisons la méthode `write()` pour écrire un tableau d'octets sur le réseau. La méthode `write()` prend trois paramètres. Le premier est un `byte[]` servant à récupérer les octets à partir desquels écrire. Le second définit un décalage dans le tableau d'octets servant à déterminer la position à partir de laquelle l'écriture doit être effectuée. Le troisième

spécifie le nombre d'octets à écrire. Dans cet exemple, nous écrivons des octets du tableau `byteArray` en commençant à la position `0` et en écrivant 10 octets.

Si le volume des données que vous écrivez est important, il devient plus efficace d'encapsuler le flux de sortie du socket dans une instance `BufferedOutputStream`, comme ceci :

```
DataOutputStream out = new DataOutputStream(new
➥BufferedOutputStream(socket.getOutputStream()));
```

Au lieu de passer directement le flux de sortie de socket au constructeur `DataOutputStream`, nous créons d'abord une instance `BufferedOutputStream` et la passons au constructeur `DataOutputStream`.

Dans cet exemple, nous avons utilisé la méthode `write()`, mais `DataOutputStream` possède bien d'autres méthodes pour écrire des données depuis n'importe quel type de données Java primitif, dont les suivantes : `write()`, `writeBoolean()`, `writeByte()`, `writeBytes()`, `writeChar()`, `writeChars()`, `writeDouble()`, `writeFloat()`, `writeInt()`, `writeLong()` et `writeShort()`. Pour plus de détails sur l'utilisation de ces méthodes de la classe `DataOutputStream`, consultez le JavaDoc à l'adresse **http://java.sun.com/j2se/1.5.0/docs/api/java/io/DataOutputStream.html**.

Lire des données sérialisées

```
ObjectInputStream in =
    ➥new ObjectInputStream(socket.getInputStream());
Object o = in.readObject();
```

Le Java permet de sérialiser les instances d'objet et de les écrire dans un fichier ou sur un réseau. Cet exemple montre comment lire un objet sérialisé depuis un socket réseau.

Il requiert que vous ayez précédemment créé un socket au serveur avec lequel vous souhaitez communiquer. Pour plus de détails sur la création de l'instance de socket, consultez l'exemple "Contacter un serveur" de ce chapitre.

Dans cet exemple, nous appelons la méthode `getInputStream()` de l'instance de socket afin d'obtenir une référence au flux d'entrée du socket. Avec cette référence, nous pouvons instancier un `ObjectInputStream`. La classe `ObjectInputStream` est utilisée pour désérialiser des données et des objets primitifs précédemment écrits en utilisant un `ObjectOutputStream`. Nous utilisons la méthode `readObject()` de l'objet `ObjectInputStream` pour lire un objet depuis le flux. L'objet peut ensuite être transtypé en son type attendu. Par exemple, pour lire un objet `Date` depuis le flux, nous utiliserions la ligne suivante :

```
Date aDate = (Date)in.readObject();
```

Tous les champs de données qui ne sont pas transitoires et statiques retrouvent la valeur qui était la leur lorsque l'objet a été sérialisé.

Seuls les objets qui supportent l'interface `java.io.Serializable` ou `java.io.Externalizable` peuvent être lus depuis des flux. Lors de l'implémentation d'une classe sérialisable, il est vivement recommandé de déclarer un membre de données `serialVersionUID`. Ce champ fournit un numéro de version qui est utilisé lors de la désérialisation pour vérifier que l'émetteur et le récepteur d'un objet sérialisé ont chargé pour cet objet des classes compatibles en termes de sérialisation. Si vous ne déclarez pas explicitement ce champ, un `serialVersionUID` par défaut est automatiquement calculé. Ce `serialVersionUID` par défaut tient finement compte des particularités de détail de la classe. Si vous apportez des modifications mineures à une classe et que vous souhaitiez conserver le même

numéro de version en considérant que cette implémentation reste compatible avec la version courante, vous devez déclarer votre propre `serialVersionUID`.

Ecrire des données sérialisées

```
ObjectOutputStream out =
    ➥new ObjectOutputStream(socket.getOutputStream());
out.writeObject(myObject);
```

Le Java permet de sérialiser les instances d'objet et de les écrire dans un fichier ou sur un réseau. Cet exemple montre comment lire un objet sérialisé depuis un socket réseau. Il requiert que vous ayez précédemment créé un socket au serveur avec lequel vous souhaitez communiquer. Pour plus de détails sur la création de l'instance de socket, consultez l'exemple "Contacter un serveur" de ce chapitre.

Dans cet exemple, nous appelons la méthode `getOutputStream()` de l'instance de socket pour obtenir une référence au flux de sortie du socket. Avec cette référence, nous instancions un `ObjectOutputStream`. La classe `ObjectOutputStream` est utilisée pour sérialiser des données et des objets primitifs. Nous utilisons la méthode `writeObject()` de `ObjectOutputStream` pour écrire un objet dans le flux.

Tous les champs de données qui ne sont pas transitoires et statiques sont préservés dans la sérialisation et restaurés lorsque l'objet est désérialisé. Seuls les objets qui supportent l'interface `java.io.Serializable` peuvent être écrits dans des flux.

Lire une page Web via HTTP

```
URL url = new URL("http://www.timothyfisher.com");
HttpURLConnection http = new HttpURLConnection(url);
InputStream in = http.getInputStream();
```

Cet exemple présente un autre moyen de lire des données depuis le réseau avec une programmation de plus haut niveau que le niveau socket auquel se cantonnaient les précédents exemples. Le Java peut communiquer avec une URL sur le protocole HTTP grâce à la classe `HttpURLConnection`. Ici, nous instancions un objet `URL` en passant une chaîne d'URL valide au constructeur `URL`. Ensuite, nous instancions un `HttpURLConnection` en passant l'instance `url` au constructeur `HttpURLConnection`. La méthode `getInputStream()` est appelée pour obtenir un flux d'entrée afin de lire les données depuis la connexion d'URL. A l'aide du flux d'entrée, nous pouvons ensuite lire le contenu de la page Web.

Il est aussi possible de lire le contenu d'une URL en utilisant directement la classe `URL`, comme ceci :

```
URL url = new URL("http://www.timothyfisher.com");
url.getContent();
```

La méthode `getContent()` retourne un `Object`. Celui-ci peut être un `InputStream` ou un objet contenant les données. Par exemple, la méthode `getContent()` peut retourner un objet `String` stockant le contenu d'une URL. La méthode `getContent()` que nous venons d'utiliser est en fait un raccourci du code suivant :

```
url.openConnection.getContent();
```

La méthode `openConnection()` de la classe `URL` retourne un objet `URLConnection`. Il s'agit de l'objet dans lequel la méthode `getContent()` se trouve en fait implémentée.

`HttpURLConnection` fournit des méthodes spécifiques au HTTP qui ne sont pas disponibles dans les classes plus générales `URL` ou `URLConnection`. Par exemple, on peut utiliser la méthode `getResponseCode()` pour obtenir le code d'état d'un message de réponse HTTP. Le HTTP définit également un protocole pour rediriger les requêtes vers un autre serveur. La classe `HttpURLConnection` contient des méthodes qui comprennent cette fonctionnalité également. Par exemple, si vous souhaitez opérer une requête à un serveur et suivre toutes les redirections qu'il retourne, vous pouvez utiliser le code suivant pour définir cette option :

```
URL url = new URL("http://www.timothyfisher.com");
HttpURLConnection http = new HttpURLConnection(url);
http.setFollowRedircts(true);
```

L'option est en fait positionnée à `true` par défaut. Le cas pratique le plus utile consistera donc au contraire à positionner l'option de suivi des redirections à `false` lorsque vous ne souhaitiez pas automatiquement être redirigé vers un autre serveur que celui sur lequel votre requête portait initialement. Cette mesure restrictive pourrait notamment être envisagée avec certaines applications de sécurité, lorsque vous ne faites confiance qu'à certains serveurs spécifiés.

Les pages Web qui contiennent des données sensibles sont généralement protégées par un protocole de sécurité appelé SSL (*Secure Sockets Layer*). Les pages protégées par SSL sont désignées par le préfixe "https" dans la chaîne d'URL, au lieu du préfixe "http" habituel.

Le JDK standard inclut une implémentation du SSL dans le cadre de JSSE (*Java Secure Socket Extension*). Pour récupérer une page protégée par SSL, vous devez utiliser la classe `HttpsURLConnection` au lieu de `HttpURLConnection`. `HttpsURLConnection` gère tous les détails du protocole SSL, en toute transparence. Pour plus d'informations sur l'utilisation du SSL et les autres fonctionnalités de sécurité fournies par JSSE, consultez le guide de référence JSSE proposé par Sun à l'adresse : **http://java.sun.com/j2se/1.5.0/docs/guide/security/jsse/JSSERefGuide.html**.

11

Serveurs réseau

En pratique, il y a bien plus de chances que vous écriviez du code de client réseau que du code de serveur réseau. Toutefois, bon nombre d'applications intègrent à la fois des fonctionnalités client et des fonctionnalités serveur. Le Java propose par chance un excellent support pour les deux.

Le paquetage `java.net` fournit les fonctionnalités de communication réseau côté serveur que nous utiliserons dans ce chapitre. La plate-forme J2EE (que nous n'abordons pas dans ce livre) propose de nombreux services réseau supplémentaires dont un support complet du développement Web Java côté serveur. Parmi les technologies réseau incluses dans la plate-forme J2EE, on peut citer les servlets, les EJB et le JMS.

Créer un serveur et accepter une requête

```
public static final short PORT = 9988;
ServerSocket server = new ServerSocket(PORT);
while ((clientSock = server.accept( )) != null) {
  // Traiter la demande du client
}
```

Cet exemple utilise une instance `ServerSocket` pour créer un serveur écoutant sur le port 9988. Nous passons le port sur lequel nous souhaitons que le serveur écoute au constructeur du `ServerSocket`. Une fois le socket serveur créé, nous appelons la méthode `accept()` pour attendre une connexion client.

La méthode `accept()` bloque l'exécution jusqu'à ce qu'une connexion avec un client soit opérée. Lorsqu'une connexion est établie, une nouvelle instance `Socket` est retournée.

Si un gestionnaire de sécurité est utilisé, sa méthode `checkAccept()` est appelée avec `clientSock.getInetAddress().getHostAddress()` et `clientSock.getPort()` en arguments, afin de s'assurer que l'opération est autorisée. Cette vérification peut lever une exception `SecurityException`.

Les exemples de ce chapitre utilisent tous la classe `ServerSocket`. Elle est utilisée par le serveur pour attendre et établir les connexions client.

L'exemple précédent l'a montré : lorsque vous commencez par créer une classe `ServerSocket`, vous devez spécifier un port à écouter pour les requêtes entrantes. La classe `ServerSocket` elle-même n'est pas utilisée pour la communication avec le client, mais uniquement pour établir une connexion

avec lui. Lorsque `ServerSocket` accepte une connexion client, une instance `Socket` standard est retournée. C'est cette instance qui est utilisée pour communiquer avec le client.

Pour plus d'informations sur la manière d'écrire votre code lorsque vous vous attendrez à devoir gérer de nombreuses requêtes clientes simultanées, consultez l'exemple "Gérer plusieurs clients" de ce chapitre.

Retourner une réponse

```
Socket clientSock = serverSocket.accept();
DataOutputStream out =
    ➥new DataOutputStream(clientSock.getOutputStream());
out.writeInt(someValue);
out.close();
```

Cet exemple montre comment retourner une réponse du serveur au client. La méthode `accept()` de l'instance `ServerSocket` retourne une instance `Socket` lorsqu'une connexion est établie avec un client. Nous obtenons ensuite le flux de sortie de ce socket en appelant la méthode `getOutputStream()` de cet objet. Nous utilisons le flux de sortie pour instancier un `DataOutputStream` et appelons la méthode `writeInt()` de ce dernier pour écrire une valeur entière envoyée sous forme de données binaires au client. Pour finir, nous fermons le socket en utilisant la méthode `close()` du `Socket`.

Cet exemple utilise la méthode `write()`, mais `DataOutputStream` possède bien d'autres méthodes pour écrire des données depuis n'importe quel type de données Java primitif, et notamment les suivantes : `write()`, `writeBoolean()`, `writeByte()`, `writeBytes()`, `writeChar()`, `writeChars()`, `writeDouble()`, `writeFloat()`, `writeInt()`, `writeLong()` et `writeShort()`.

Pour plus d'informations sur l'utilisation de ces méthodes et des autres méthodes de la classe `DataOutputStream`, consultez la JavaDoc à l'adresse **http://java.sun.com/j2se/1.5.0/docs/api/java/io/DataOutputStream.html**.

Si vous souhaitez écrire des données texte au client, utilisez le code suivant :

```
Socket clientSock = serverSocket.accept();
PrintWriter out = new PrintWriter(new
➥OutputStreamWriter(clientSock.getOutputStream()), true);
out.println("Hello World");
out.close();
```

Au lieu de créer un `DataOutputStream`, nous créons cette fois un `OutputStreamWriter` et un `PrintWriter`. La méthode `print()` du `PrintWriter` permet d'écrire une chaîne de texte à destination du client. Le second paramètre passé au constructeur `PrintWriter` définit l'option de purge automatique. La valeur `true` amène les méthodes `println()`, `printf()` et `format()` à vider automatiquement le tampon de sortie. Comme notre exemple utilise la méthode `println()`, il n'est pas nécessaire d'appeler explicitement la méthode `flush()`. Enfin comme toujours, lorsque nous avons fini d'utiliser le `PrintWriter`, nous appelons la méthode `close()` pour fermer le flux.

Retourner un objet

```
Socket clientSock = serverSocket.accept();
ObjectOutputStream os =
 ➥new ObjectOutputStream(clientSock.getOutputStream( ));
// Retourner un objet
os.writeObject(new Date());
os.close();
```

Cet exemple retourne un objet sérialisé au client. Nous obtenons une instance `Socket` que retourne la méthode `accept()` du `ServerSocket` une fois qu'une connexion à un client est établie. Nous créons alors une instance `ObjectOutputStream` et passons le flux de sortie obtenu depuis le socket client. `ObjectOutputStream` est utilisée pour écrire des types de données primitifs et des graphes d'objets Java vers un `OutputStream`. Dans cet exemple, nous écrivons un objet `Date` dans le flux de sortie puis fermons ce flux.

La méthode `writeObject()` sérialise l'objet passé en paramètre. Dans cet exemple, il s'agit d'un objet `Date`. Tous les champs de données qui ne sont pas transitoires et dynamiques sont préservés dans la sérialisation et restaurés lorsque l'objet est désérialisé. Seuls les objets qui supportent l'interface `java.io.Serializable` peuvent être sérialisés.

Le projet open source XStream de **codehaus.org** propose une alternative intéressante aux classes `ObjectOutputStream` et `ObjectInputStream`. XStream fournit des implémentations de `ObjectInputStream` et `ObjectOutputStream` qui permettent aux flux d'objets d'être sérialisés ou désérialisés en XML. La classe `ObjectInputStream` standard utilise un format binaire pour les données sérialisées. La sortie sérialisée des classes XStream fournit pour sa part les classes sérialisées dans un format XML facile à lire. Pour plus d'informations sur XStream et pour télécharger ce projet, rendez-vous à l'adresse **http://xstream.codehaus.org/index.html**.

Gérer plusieurs clients

```
while (true) {
  Socket clientSock = socket.accept();
  new Handler(clientSock).start();
}
```

Pour gérer plusieurs clients, il suffit de créer un thread pour chaque requête entrante à traiter. Dans cet exemple, nous créons un nouveau thread pour gérer la connexion cliente entrante immédiatement après avoir accepté la connexion. Ce procédé permet de libérer notre thread d'écouteur de serveur qui peut retourner écouter les connexions clientes suivantes.

Dans cet exemple, nous nous trouvons à l'intérieur d'une boucle `while` infinie : dès qu'un thread est engendré pour gérer une requête entrante, le serveur retourne immédiatement attendre la requête suivante. La classe `Handler` que nous utilisons pour démarrer le thread doit être une sous-classe de la classe `Thread` ou doit implémenter l'interface `Runnable`. Le code utilisé dans l'exemple est correct si la classe `Handler` est une sous-classe de la classe `Thread`. Si la classe `Handler` implémente au contraire l'interface `Runnable`, le code de démarrage du thread devient alors le suivant :

```
Thread thd = new Thread(new Handler(clientSock));
thd.start();
```

Voici l'exemple d'une classe `Handler` simple qui étend la classe `Thread` :

```
class Handler extends Thread {
    Socket sock;

    Handler(Socket socket) {
     this.sock = socket;
    }
```

```
    public void run() {
     DataInputStream in =
      ➥new DataInputStream(sock.getInputStream());
     PrintStream out =
      ➥new PrintStream(sock.getOutputStream(), true);

     // Gestion de la requête cliente

     sock.close();
    }
}
```

Cette classe peut être utilisée pour gérer des requêtes clientes entrantes. L'implémentation concrète de la gestion des requêtes spécifiques dans le code est volontairement masquée ici, aux fins de l'illustration. Lorsque la méthode `start()` de cette classe est appelée, comme c'est le cas dans notre exemple précédent, la méthode `run()` définie ici est exécutée. `start()` est implémentée dans la classe de base `Thread` : nous n'avons pas à la redéfinir dans notre implémentation de `Handler`.

Lors de la création d'une solution multithreadée de ce type, il peut aussi être intéressant de créer un système de pooling des threads. Dans ce cas, vous créerez un pool de threads au démarrage de l'application au lieu d'engendrer un nouveau thread pour chaque requête entrante.

Le pool de threads contient un nombre fixe de threads pouvant exécuter des tâches. Ce système évite que l'application ne crée un nombre excessif de threads qui pourraient grever les performances système. Un très bon article (en anglais) concernant le pooling des threads peut être consulté à l'adresse **http://www.informit.com/articles/article.asp?p=30483&seqNum=3&rl=1**. Pour plus d'informations sur l'utilisation des threads, consultez le Chapitre 15, "Utiliser des threads".

Servir du contenu HTTP

```
Socket client = serverSocket.accept();
BufferedReader in = new BufferedReader
➥(new InputStreamReader(client.getInputStream()));
// Avant de servir une réponse, on lit habituellement
// l'entrée cliente et on traite la requête.
PrintWriter out =
      ➥new PrintWriter(client.getOutputStream());
out.println("HTTP/1.1 200");
out.println("Content-Type: text/html");
String html = "<html><head><title>Test Response" +
➥"</title></head><body>Just a test</body></html>";
out.println("Content-length: " + html.length());
out.println(html);
out.flush();
out.close();
```

Cet exemple montre comment servir du contenu HTML très simple *via* HTTP. Pour commencer, nous acceptons une connexion avec un client, créons un `BufferedReader` pour lire la requête cliente et créons un `PrintWriter` que nous utilisons pour retransmettre le HTML *via* HTTP au client. Les données que nous écrivons vers le `PrintWriter` constituent le minimum nécessaire pour créer un message de réponse HTTP valide. Notre réponse est composée de trois champs d'en-tête HTTP et de nos données HTML. Nous commençons notre réponse en spécifiant la version HTTP et un code de réponse dans la ligne suivante :

```
out.println("HTTP/1.1 200");
```

Nous indiquons le HTTP version 1.1 et le code de réponse 200. Ce code signale que la requête a réussi. A la ligne suivante, nous signalons que le type de contenu retourné est du HTML. D'autres types de contenu peuvent être retournés pour un message de réponse HTTP valide.

Par exemple, la ligne suivante spécifie que la réponse correspond à du texte brut et non du code HTML :

```
out.println("Content-Type: text/plain");
```

Ensuite, nous écrivons l'en-tête `Content-length`. Celui-ci spécifie la longueur du contenu retourné, sans tenir compte des champs d'en-tête. Après cela, nous écrivons le message HTML à retourner. Pour finir, nous purgeons le flux `BufferedReader` et le fermons avec les méthodes `flush()` et `close()`.

Info

Cette technique est utile pour les besoins simples en matière de service HTTP, mais il n'est pas recommandé d'écrire de toutes pièces un serveur HTTP complet en Java. Un excellent serveur HTTP, gratuit et open source, est disponible dans le cadre du projet Jakarta d'Apache : le serveur Tomcat. Pour plus d'informations sur Tomcat et pour en télécharger les fichiers, accédez à http://jakarta.apache.org/tomcat/. Tomcat sert du contenu sur HTTP mais fournit également un conteneur de servlets pour gérer les servlets Java et les JSP.

12

Envoyer et recevoir des e-mails

L'e-mail est utilisé dans de nombreuses applications. Il est fort probable qu'à un stade ou un autre de vos projets de développement, vous soyez conduit à prendre en charge des courriers électroniques dans l'une de vos applications Java.

Le Java facilite l'intégration des fonctionnalités de messagerie électronique à vos applications Java grâce à l'API JavaMail. Cette API est une extension du Java que vous devez télécharger séparément. Elle ne fait pas partie du paquetage JDK standard proposé en téléchargement. Les classes utiles qui constituent l'API JavaMail se trouvent dans le paquetage `javax.mail`. L'API JavaMail actuelle requiert le JDK 1.4 ou une version ultérieure. Les versions antérieures du JDK nécessitent des versions également antérieures de l'API JavaMail.

Ce chapitre traite de l'envoi et de la réception d'e-mails à partir d'une application Java. L'intégration des capacités de messagerie électronique à votre application Java constitue un excellent ajout dans de nombreuses applications. En pratique, ces fonctionnalités peuvent être utiles pour envoyer des alertes par e-mail, transmettre automatiquement des journaux et des rapports et plus généralement, communiquer avec les utilisateurs.

Vue d'ensemble de l'API JavaMail

JavaMail fournit des fonctionnalités pour envoyer et recevoir des e-mails. Des fournisseurs de service s'ajoutent comme composants additionnels à l'API JavaMail pour offrir des implémentations de différents protocoles de messagerie. L'implémentation Sun inclut des fournisseurs de services pour IMAP, POP3 et SMTP. JavaMail fait également partie de Java Enterprise dans J2EE.

Pour télécharger l'extension JavaMail, rendez-vous à l'adresse **http://java.sun.com/products/javamail/downloads/index.html**.

Pour utiliser l'API JavaMail, vous devez également télécharger et installer l'extension JAF (*JavaBeans Activation Framework*) depuis **http://java.sun.com/products/javabeans/jaf/downloads/index.html**.

En plus des exemples traités dans ce chapitre, vous pouvez trouver des informations détaillées complètes sur l'utilisation de l'API JavaMail grâce au lien JavaMail de réseau des développeurs Sun (en anglais) : **http://java.sun.com/products/javamail/index.jsp**.

Envoyer des e-mails

```
Properties props = new Properties( );
props.put("mail.smtp.host", "mail.yourhost.com");
Session session = Session.getDefaultInstance(props, null);
Message msg = new MimeMessage(session);
msg.setFrom(new InternetAddress("tim@timothyfisher.com"));
InternetAddress toAddress =
    ➥new InternetAddress("kerry@timothyfisher.com");
```

Cet exemple envoie un message électronique en texte brut à l'aide d'un serveur SMTP. Six étapes de base doivent être impérativement suivies lorsque vous souhaitez envoyer un e-mail avec l'API JavaMail :

1. Vous devez créer un objet `java.util.Properties`, que vous utiliserez pour passer des informations concernant le serveur de messagerie.
2. Vous devez placer le nom d'hôte du serveur de messagerie SMTP dans l'objet `Properties`, ainsi que toutes les autres propriétés à définir.
3. Vous devez créer des objets `Session` et `Message`.
4. Vous devez définir les adresses e-mail du destinataire et de l'expéditeur du message ainsi que son sujet dans l'objet `Message`.
5. Vous devez définir le texte du message dans l'objet `Message`.
6. Vous devez appeler la méthode `Transport.send()` pour envoyer le message.

L'exemple précédent suit chacune de ces étapes pour créer et envoyer un message électronique. Notez que les adresses de provenance (`From`) et de destination (`To`) sont créées sous forme d'objets `InternetAddress`. L'objet `InternetAddress` représente une adresse e-mail valide.

Une exception est levée si vous tentez de créer un objet `InternetAddress` en utilisant un format d'adresse e-mail invalide. Lorsque vous spécifiez les destinataires `To`, vous devez aussi spécifier leur type. Les types valides sont `TO`, `CC` et `BCC`. Ils sont représentés par les constantes suivantes :

```
Message.RecipientType.TO
Message.RecipientType.CC
Message.RecipientType.BCC
msg.addRecipient(Message.RecipientType.TO, toAddress);
msg.setSubject("Test Message");
msg.setText("This is the body of my message.");
Transport.send(msg);
```

La classe `Message` est une classe abstraite définie dans le paquetage `javax.mail`. Une sous-classe qui l'implémente fait partie de l'implémentation standard de JavaMail : la classe `MimeMessage`. C'est cette implémentation que nous utilisons dans l'exemple au début de la section. La classe `MimeMessage` représente un message électronique de style MIME. Elle devrait vous suffire pour la plupart de vos besoins en matière de messagerie électronique.

Dans cet exemple, nous utiliserons l'objet `Properties` pour ne passer que l'hôte de messagerie SMTP. Il s'agit de l'unique propriété qu'il est obligatoire de définir, mais d'autres propriétés supplémentaires peuvent aussi être spécifiées.

Info

Consultez le sommaire concernant le paquetage `javax.mail` dans la JavaDoc pour plus de détails sur d'autres propriétés de messagerie liées qui peuvent être passées dans l'objet `Properties` : http://java.sun.com/javaee/5/docs/api/javax/mail/package-summary.html.

Envoyer des e-mails MIME

```
String html = "<html><head><title>Java Mail</title></head>"
  ➥+ "<body>Some HTML content.</body></html>";
Multipart mp = new MimeMultipart();
BodyPart textPart = new MimeBodyPart( );
textPart.setText("This is the message body.");
BodyPart htmlPart = new MimeBodyPart( );
htmlPart.setContent(html, "text/html");
mp.addBodyPart(textPart);
mp.addBodyPart(htmlPart);
msg.setContent(mp);
Transport.send(msg);
```

MIME est l'acronyme de *Multimedia Internet Mail Extensions*. Ce standard est supporté par tous les principaux clients de messagerie. Il constitue le moyen standard d'associer des pièces jointes aux messages. Il permet de joindre aux e-mails une variété de types de contenu dont des images, des vidéos et des fichiers PDF. L'API JavaMail supporte également les messages MIME. Il est même presque aussi facile de créer un message MIME avec des pièces jointes qu'un message standard en texte brut.

Dans cet exemple, nous créons et envoyons un message MIME contenant un corps en texte brut et une pièce jointe HTML.

Pour créer un message multipartie, nous utilisons la classe `MultiPart` du paquetage `javax.mail`. La classe `MimeMultiPart` du paquetage `javax.mail.Internet` fournit une implémentation concrète de la classe abstraite `MultiPart` et utilise des conventions MIME pour les données multiparties. La classe `MimeMultiPart` permet d'ajouter plusieurs parties de corps de message représentées sous forme d'objets `MimeBodyPart`. Le contenu des parties de corps est défini en utilisant la méthode `setText()` pour les parties de corps en texte brut et `setContent()` pour les autres types de parties.

Ensuite, nous utilisons la méthode `setContent()` en passant un objet contenant la partie de corps avec une chaîne spécifiant le type MIME que nous ajoutons. Ici, nous ajoutons une partie de corps HTML et spécifions donc le type MIME `text/html`.

Le code présenté dans l'exemple se concentre spécifiquement sur les étapes d'envoi du message relatives au standard MIME. Voici un exemple plus complet qui inclut toutes les étapes nécessaires à la réalisation de cette tâche :

```
Properties props = new Properties( );
props.put("mail.smtp.host", "mail.yourhost.com");
Session session = Session.getDefaultInstance(props, null);
Message msg = new MimeMessage(session);
msg.setFrom(new InternetAddress("tim@timothyfisher.com"));
InternetAddress toAddress =
    ➥new InternetAddress("kerry@timothyfisher.com");
msg.addRecipient(Message.RecipientType.TO, toAddress);
msg.setSubject("Test Message");
String html = "<html><head><title>Java Mail</title>
➥</head>" + "<body>Some HTML content.</body></html>";
Multipart mp = new MimeMultipart();
BodyPart textPart = new MimeBodyPart( );
textPart.setText("This is the message body.");
BodyPart htmlPart = new MimeBodyPart( );
htmlPart.setContent(html, "text/html");
mp.addBodyPart(textPart);
mp.addBodyPart(htmlPart);
msg.setContent(mp);
Transport.send(msg);
```

Info

L'IANA (Internet Assigned Numbers Authority) propose une référence précise de tous les types de contenu MIME standard sur son site Web. Le site propose également une application permettant d'enregistrer de nouveaux types MIME. Si vous avez le sentiment qu'aucun type MIME existant ne correspond à votre contenu, vous pouvez utiliser cette application pour demander la création d'un nouveau type de contenu MIME correspondant à votre type de contenu. Pour accéder au site Web de l'IANA, rendez-vous à l'adresse http://www.iana.org. Les types de contenu MIME peuvent être trouvés à l'adresse http://www.iana.org/assignments/media-types/.

Lire un e-mail

```
Properties props = new Properties();
Session session = Session.getDefaultInstance(props, null);
Store store = session.getStore("pop3");
store.connect(host, username, password);
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
Message message[] = folder.getMessages();
for (int i=0, n=message.length; i<n; i++) {
  System.out.println(i + ": " + message[i].getFrom()[0] +
  ➥"\t" + message[i].getSubject());
  String content = message[i].getContent().toString();
  System.out.print(content.substring(0,100));
}
folder.close(false);
store.close();
```

Dans cet exemple, nous nous connectons à un serveur de messagerie POP3 et récupérons tous les messages dans le dossier `INBOX` (boîte de réception). L'API JavaMail facilite considérablement cette tâche.

Voici les étapes générales à suivre lorsque vous utilisez l'API JavaMail pour lire des messages depuis un serveur de messagerie POP :

1. Vous devez obtenir un objet `Session`.
2. Vous devez obtenir un objet `Store` à partir de l'objet `Session`.
3. Vous devez créer un objet `Folder` pour le dossier que vous souhaitez ouvrir.
4. Vous devez ouvrir le dossier et récupérer les messages. Un dossier peut contenir des sous-dossiers : il convient donc de récupérer également les messages de ces dossiers en procédant de manière récursive.

Dans cet exemple, nous obtenons une instance par défaut de l'objet `Session` en utilisant la méthode statique `getDefaultInstance()`. L'objet `Session` représente une session de messagerie. A partir de cet objet, nous obtenons ensuite un objet `Store` qui implémente le protocole POP3. L'objet `Store` représente un entrepôt de messages et son protocole d'accès. Si par exemple, nous souhaitions nous connecter à un serveur de messagerie IMAP au lieu d'un serveur POP3, nous pourrions modifier cette ligne de code afin d'obtenir un entrepôt IMAP au lieu d'un entrepôt POP3. Nous devrions également inclure un fichier JAR supplémentaire qui supporte le protocole IMAP. Sun fournit le fichier `imap.jar` dans le cadre de la distribution JavaMail. Nous nous connectons à un entrepôt POP3 en appelant la méthode `connect()` de l'objet `Store` et en passant un hôte, un nom d'utilisateur et un mot de passe.

Dans le reste de l'exemple, nous récupérons le dossier `INBOX` et tous les messages qu'il contient. Nous imprimons l'expéditeur (`From`), l'objet du message et les cent premiers caractères du corps de chaque message dans le dossier `INBOX`.

La classe `Folder` contient également une méthode `list()` qui n'est pas utilisée dans cet exemple mais permet de récupérer un tableau d'objets `Folder` représentant tous les sous-dossiers du dossier sur lequel elle est appelée. Si le dossier `INBOX` contient de nombreux sous-dossiers, il est ainsi possible d'obtenir une référence à chacun d'entre eux à l'aide du code suivant :

```
Folder folder = store.getFolder("INBOX");

folder.open(Folder.READ_ONLY);

Folder[] subfolders = folder.list();
```

Le tableau `subfolders` de cet exemple contiendra un objet `Folder` pour chaque sous-dossier du dossier `INBOX`. Il sera alors possible de traiter les messages dans chacun d'entre eux, comme nous l'avons fait pour ceux du dossier `INBOX`. La classe `Folder` propose aussi une méthode `getFolder()` qui prend un unique paramètre de chaîne et retourne un dossier dont le nom correspond à la chaîne passée.

Grâce à la classe `Folder`, vous pouvez écrire une méthode qui parcourt l'ensemble d'un compte de messagerie et lit les messages des différents dossiers de l'utilisateur.

13

Accès aux bases de données

Les bases de données fournissent un mécanisme de stockage persistant pour les données d'application et dans bien des cas de figure, elles sont essentielles au fonctionnement des applications. Le Java propose un excellent support pour l'accès aux bases de données relationnelles avec l'API JDBC (*Java Database Connectivity*).

Si votre application ne définit qu'un modèle de données très simple et ne requiert qu'un accès très limité à une base de données, l'API JDBC convient bien. Au-delà de ce cas de figure, il peut cependant être judicieux de considérer l'emploi d'un framework de base de données plutôt que de programmer directement l'API JDBC. Le framework de persistance standard pour les applications d'entreprise est le framework EJB (*Enterprise Java Beans*).

EJB fait partie de Java Enterprise Edition. Il est considéré comme étant excessivement complexe par bon nombre de développeurs Java. Ce défaut a d'ailleurs fait le succès

de certaines solutions open source qui deviennent de plus en plus populaires. La complexité d'EJB et les problèmes qui lui étaient associés ont heureusement été partiellement résolus dans EJB 3.0. EJB 3.0 constitue une avancée majeure dans la bonne direction et devrait faire d'EJB une technologie plus conviviale pour les développeurs.

L'excellent framework de persistance de données open source Hibernate gagne sans cesse en popularité. Il crée une couche de mapping objet de vos données relationnelles. La couche de mapping objet permet de traiter les données persistantes avec une approche orientée objet par opposition à l'interface SQL procédurale. Pour plus d'informations sur le framework Hibernate, rendez-vous à l'adresse **http://www.hibernate.org**.

Ce chapitre se concentre purement sur l'accès aux bases de données *via* JDBC. Si vous utilisez un framework de persistance de plus haut niveau, il reste important de bien comprendre l'API JDBC, car elle définit les fondations sur lesquelles s'appuient la plupart des frameworks de plus haut niveau.

Se connecter à une base de données via JDBC

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection conn =
   DriverManager.getConnection(url, user, password);
```

Pour créer une connexion de base de données avec JDBC, vous devez d'abord charger un pilote. Dans cet exemple, nous chargeons le `JdbcOdbcDriver`. Ce pilote fournit une connectivité à une source de données ODBC.

Nous le chargeons avec la méthode `Class.forName()`. Les pilotes de base de données JDBC sont généralement fournis par les éditeurs de bases de données, bien que Sun propose plusieurs pilotes génériques dont le pilote ODBC utilisé dans cet exemple. Une fois le pilote chargé, nous obtenons une connexion à la base de données avec la méthode `DriverManager.getConnection()`. La syntaxe utilisée pour spécifier la base de données à laquelle nous souhaitons nous connecter prend la forme d'une URL. Nous passons également un nom d'utilisateur et un mot de passe valides pour la connexion à la base de données. L'URL doit commencer par le préfixe `jdbc:`. Le reste du format de spécification d'URL (après le préfixe) est spécifique à l'éditeur. Voici la syntaxe d'URL pour se connecter à une base de données ODBC :

```
jdbc:odbc:nomdebasededonnées
```

La plupart des pilotes requièrent que la chaîne d'URL inclue un nom d'hôte, un port et un nom de base de données. Voici par exemple une URL valide pour la connexion à une base de données MySQL :

```
jdbc:mysql://db.myhost.com:3306/mydatabase
```

Cette URL spécifie une base de données MySQL sur l'hôte `db.myhost.com` pour une connexion sur le port 3306 avec le nom de base de données `mydatabase`. Le format général d'une URL de base de données MySQL est le suivant :

```
jdbc:mysql://hôte:port/base de données
```

L'un des autres moyens d'obtenir une connexion de base de données consiste à utiliser JNDI. C'est en général l'approche que vous adopterez si vous utilisez un serveur d'applications comme WebLogic de BEA ou WebSphere d'IBM.

```
Hashtable ht = new Hashtable();
ht.put(Context.INITIAL_CONTEXT_FACTORY,
➥"weblogic.jndi.WLInitialContextFactory");
ht.put(Context.PROVIDER_URL, "t3://hostname:port");
Context ctx = new InitialContext(ht);
javax.sql.DataSource ds =
  ➥(javax.sql.DataSource) ctx.lookup("myDataSource");
Connection conn = ds.getConnection();
```

Avec JNDI, nous créons une instance `InitialContext` et l'utilisons pour rechercher un `DataSource`. Ensuite, nous obtenons la connexion depuis l'objet de source de données. Une autre version de la méthode `getConnection()` permet aussi de passer un nom d'utilisateur et un mot de passe pour la connexion aux bases de données requérant une authentification.

Il est important de toujours veiller à fermer la connexion en utilisant la méthode `close()` de la classe `Connection` lorsque vous avez terminé d'utiliser l'instance `Connection`.

Envoyer une requête via JDBC

```
Statement stmt = conn.createStatement( );
ResultSet rs =
 ➥stmt.executeQuery("SELECT * from users where name='tim'");
```

Dans cet exemple, nous créons une instruction JDBC avec la méthode `createStatement()` de l'objet `Connection` et l'utilisons pour exécuter une requête qui retourne un `ResultSet` Java. Pour la création de la connexion, référez-vous au précédent exemple, "Se connecter à une base de données via JDBC". Pour réaliser la requête `SELECT`, nous utilisons la méthode `executeQuery()` de l'objet `Statement`.

Si vous souhaitez effectuer une opération `UPDATE` au lieu d'une requête `SELECT`, vous devez utiliser la méthode `executeUpdate()` de l'objet `Statement` au lieu de sa méthode `executeQuery()`. La méthode `executeUpdate()` est utilisée avec des instructions SQL `INSERT`, `UPDATE` et `DELETE`. Elle retourne le compte des lignes pour les instructions `INSERT`, `UPDATE` ou `DELETE` ou 0 si l'instruction SQL ne retourne rien. Voici un exemple d'exécution d'une instruction `UPDATE` :

```
Statement stmt = conn.createStatement( );
int result = stmt.executeUpdate("UPDATE users SET name=
➥'tim' where id='1234'");
```

Il est important de se souvenir qu'il n'est possible d'ouvrir qu'un seul objet `ResultSet` à la fois par objet `Statement`. Toutes les méthodes d'exécution dans l'interface `Statement` ferment l'objet `ResultSet` courant s'il en existe déjà un d'ouvert. Ce point est important à retenir si vous imbriquez des connexions et des requêtes de base de données.

JDBC 3.0 a introduit une fonctionnalité de conservation du jeu de résultats. Elle permet de conserver plusieurs jeux de résultats ouverts si vous spécifiez cette option lorsque l'objet d'instruction est créé. Pour en apprendre plus sur les nouvelles fonctionnalités de JDBC 3.0, consultez l'article suivant (en anglais) sur le site Web DeveloperWorks d'IBM : **http://www.128.ibm.com/developerworks/java/library/j-jdbcnew/**.

Lorsque vous travaillez avec des instructions et des résultats, veillez toujours à bien fermer les objets `Connection`, `Statement` et `ResultSet` lorsque vous en avez terminé. Chacun de ces objets possède une méthode `close()` permettant de le fermer afin de libérer la mémoire et les ressources. Si vous omettez de les fermer, vous risquez de créer des fuites mémoire dans vos applications Java.

Le fait de ne pas fermer une connexion peut également provoquer des cas d'interblocage dans les applications multithreadées.

Si l'une de vos instructions SQL doit être exécutée de nombreuses fois, il est plus efficace d'utiliser une requête `PreparedStatement`. Pour plus d'informations à ce sujet, consultez l'exemple suivant, "Utiliser une instruction préparée".

Utiliser une instruction préparée

```
PreparedStatement stmnt = conn.prepareStatement("INSERT
➥into users values (?,?,?,?)");
stmnt.setString(1, name);
stmnt.setString(2, password);
stmnt.setString(3, email);
stmnt.setInt(4, employeeId);
stmnt.executeUpdate( );
```

Pour créer une instruction préparée dans JDBC, vous devez utiliser un objet `PreparedStatement` au lieu d'un objet `Statement`. Dans cet exemple, nous passons le code SQL à la méthode `prepareStatement()` de l'objet `Connection`. Cette opération crée un objet `PreparedStatement`. Avec une instruction préparée, les valeurs de données dans l'instruction SQL sont spécifiées à l'aide de points d'interrogation. Les véritables valeurs pour ces jokers représentés par des points d'interrogation sont définies plus tard en utilisant les méthodes set de `PreparedStatement`. Les méthodes set disponibles sont `setArray()`, `setAsciiStream()`, `setBigDecimal()`, `setBinaryStream()`, `setBlob()`, `setBoolean()`, `setByte()`, `setBytes()`, `setCharacterStream()`, `setClob()`, `setDate()`, `setDouble()`, `setFloat()`, `setInt()`, `setLong()`, `setNull()`, `setObject()`,

`setRef()`, `setShort()`, `setString()`, `setTime()`, `setTimestamp()` et `setURL()`. Chacune de ces méthodes set est utilisée pour définir un type de données spécifique sous forme de paramètre dans l'instruction SQL. Par exemple, la méthode `setInt()` est utilisée pour définir des paramètres entiers, la méthode `setString()` pour définir des paramètres `String`, etc.

Dans cet exemple, nous positionnons trois valeurs de chaîne et une valeur entière avec les méthodes `setString()` et `setInt()`.

Chaque point d'interrogation qui apparaît dans l'instruction de requête doit avoir une instruction set correspondante qui définisse sa valeur. Le premier paramètre des instructions set spécifie la position du paramètre correspondant dans l'instruction de requête. Si la valeur 1 est passée comme premier paramètre à une instruction set, c'est ainsi la valeur correspondant au premier point d'interrogation qui est positionnée dans l'instruction de requête. Le second paramètre des instructions set spécifie la valeur elle-même du paramètre. Dans notre exemple, les variables `name`, `password` et `email` sont toutes censées être de type `String`. La variable `employeeId` est de type `int`.

Lorsque vous créez une instruction SQL que vous allez réutiliser plusieurs fois, il est plus efficace d'utiliser un objet `PreparedStatement` au lieu d'un objet `Statement` standard. L'instruction préparée est une instruction SQL précompilée qui offre une exécution plus rapide une fois créée.

Récupérer les résultats d'une requête

```
ResultSet rs = stmt.executeQuery("SELECT name, password
➥FROM users where name='tim'");
while (rs.next( )) {
  String name = rs.getString(1);
  String password = rs.getString(2);
}
```

Les requêtes JDBC retournent un objet `ResultSet`. Ce dernier représente une table de données contenant les résultats d'une requête de base de données. Le contenu du `ResultSet` peut être parcouru afin d'obtenir les résultats de la requête exécutée. Le `ResultSet` conserve un curseur qui pointe sur la ligne de données courante. L'objet `ResultSet` possède une méthode `next()` qui déplace le curseur à la ligne suivante. La méthode `next()` retourne `false` lorsqu'il n'y a plus de ligne dans l'objet `ResultSet`. Il est ainsi possible d'utiliser une boucle `while` pour parcourir toutes les lignes contenues dans le `ResultSet`.

Le `ResultSet` possède des méthodes getter permettant de récupérer les valeurs de colonne de la ligne courante. Les valeurs de données peuvent être récupérées à l'aide du numéro d'index ou du nom de la colonne. La numérotation des colonnes commence à 1. La casse n'est pas prise en compte pour les noms de colonne fournis en entrée aux méthodes getter.

Dans cet exemple, nous obtenons un `ResultSet` suite à l'exécution d'une requête `SELECT`. Nous parcourons en boucle les lignes contenues dans le `ResultSet` en utilisant la méthode `next()` et une boucle `while`. Nous obtenons les valeurs de données `name` et `password` avec la méthode `getString()`.

Rappelez-vous qu'il est conseillé de fermer vos instances de `ResultSet` lorsque vous avez fini de les utiliser. Les objets `ResultSet` sont automatiquement fermés lorsque l'objet `Statement` qui les a générés est fermé, réexécuté ou utilisé pour récupérer le résultat suivant d'une séquence de résultats multiples.

Utiliser une procédure stockée

```
CallableStatment cs =
      ➥conn.prepareCall("{ call ListAllUsers }");
ResultSet rs = cs.executeQuery( );
```

Les procédures stockées sont des programmes de base de données stockés et conservés dans la base de données elle-même. Elles peuvent être appelées depuis le code Java en utilisant l'interface `CallableStatement` et la méthode `prepareCall()` de l'objet `Connection`. `CallableStatement` retourne un objet `ResultSet` comme le fait `Statement` ou `PreparedStatement`. Dans cet exemple, nous appelons la procédure stockée `ListAllUsers` sans paramètre.

L'objet `CallableStatement` peut prendre des paramètres d'entrée également. Ceux-ci sont gérés exactement comme ils le sont avec un `PreparedStatement`. Par exemple, le code suivant montre comment appeler une procédure stockée qui utilise des paramètres d'entrée :

```
CallableStatment cs = conn.prepareCall("{ call
➥AddInts(?,?) }");
cs.setInt(1,10);
cs.setInt(2,50);
ResultSet rs = cs.executeQuery( );
```

A la différence des autres types d'instructions JDBC, `CallableStatement` peut également retourner des paramètres, appelés paramètres OUT. Le type JDBC de chaque paramètre OUT doit être enregistré avant que l'objet `CallableStatement` puisse être exécuté. Cette inscription s'opère avec la méthode `registerOutParameter()`. Une fois que l'instruction a été exécutée, les paramètres OUT peuvent être récupérés en utilisant les méthodes getter de `CallableStatement`.

```
CallableStatement cs = con.prepareCall("{call
  ➥getData(?, ?)}");
cs.registerOutParameter(1, java.sql.Types.INT);
cs.registerOutParameter(2, java.sql.Types.STRING);
ResultSet rs = cs.executeQuery();
int intVal = cs.getInt(1);
String strVal = cs.getString(2);
```

Dans cet exemple, nous appelons une procédure stockée nommée `getData()` qui possède deux paramètres OUT. L'un de ces paramètres OUT est une valeur `int` et l'autre une valeur `String`. Une fois ces deux paramètres enregistrés, nous exécutons la requête et obtenons leurs valeurs avec les méthodes `getInt()` et `getString()`.

L'une des autres différences à remarquer tient à ce que les procédures stockées peuvent retourner plusieurs jeux de résultats. Si une procédure stockée retourne plus d'un jeu de résultats, on peut utiliser la méthode `getMoreResults()` de la classe `CallableStatement` pour fermer le jeu de résultats courant et pointer sur le suivant. La méthode `getResultSet()` peut être appelée ensuite pour récupérer le jeu de résultats nouvellement désigné.

Voici un exemple qui retourne plusieurs jeux de résultats et utilise ces méthodes pour récupérer chacun d'entre eux :

```
int i;
String s;
callablestmt.execute();
rs = callablestmt.getResultSet();
while (rs.next()) {
    i = rs.getInt(1);
}
callablestmt.getMoreResults();
rs = callablestmt.getResultSet();
while (rs.next()) {
    s = rs.getString(1);
}
rs.close();
callablestmt.close();
```

Ici, nous positionnons la valeur `int i` avec les résultats du premier jeu de résultats et la variable `String s` avec ceux du second.

14

XML

Le XML (*eXtensible Markup Language*) est dérivé du SGML (*Standard Generalized Markup Language*), tout comme le HTML (*Hypertext Markup Language*). En fait, le XML est même analogue en bien des points au HTML, à ceci près qu'en XML, il vous revient de définir vos propres balises. Vous n'êtes pas cantonné à un jeu prédéfini de balises comme vous l'êtes en HTML. Le XHTML, pour sa part, est une version du HTML compatible avec le standard XML.

Le XML est couramment utilisé comme format générique pour l'échange de données entre serveurs et applications, dans les processus de communication entre couches applicatives ou pour le stockage de données complexes comme les documents de traitement de texte voir les fichiers graphiques.

Le XML a été largement adopté dans tous les secteurs d'industrie et par la majorité des langages de programmation. La plupart d'entre eux proposent maintenant un support pour le traitement des données XML. Le Java n'y fait pas exception et fournit d'excellents outils pour le traitement des documents XML, que ce soit pour créer ou pour lire des données XML.

Ce chapitre requiert des connaissances en XML. Si vous souhaitez apprendre ce langage ou parfaire vos connaissances dans ce domaine, consultez *XML* de Michael Morrison (CampusPress, 2006).

Deux API de parsing XML courantes indépendantes du langage sont définies par le W3C (World Wide Web Consortium) : les API DOM et SAX. Le DOM (*Document Object Model*) est un parseur qui lit un document XML entier et construit un arbre d'objets `Node`, que l'on appelle le DOM ou le modèle objet du document. Le DOM livre une représentation parsée complète du document XML dont vous pouvez extraire des éléments à tout moment. SAX (*Simple API for XML*) n'est pas un véritable parseur en soi, mais plus exactement une API qui définit un mécanisme de gestion des événements pouvant servir à parser des documents XML. Vous pouvez créer des méthodes de rappel qui sont appelées par l'API SAX au moment où des éléments spécifiques du document XML sont atteints.

L'implémentation SAX scanne le document XML en appelant les méthodes de rappel dès qu'elle rencontre le début et la fin d'éléments particuliers du document XML. Avec SAX, le document XML n'est jamais complètement stocké ou représenté en mémoire.

L'implémentation Java du traitement XML est appelée JAXP (*Java API for XML Processing*). JAXP permet aux applications de parser et de transformer des documents XML sans l'aide d'une implémentation de traitement XML. JAXP contient un parseur DOM et un parseur SAX ainsi qu'une API XSLT pour la transformation des documents XML. XSLT est l'acronyme de *eXtensible Stylesheet Language Transformations*. La technologie XSLT permet de transformer les documents XML en les faisant passer d'un format à un autre. JAXP fait partie intégrante du JDK 1.4 et de ses versions ultérieures.

Parser du XML avec SAX

```
XMLReader parser = XMLReaderFactory.createXMLReader
➥("org.apache.xerces.parsers.SAXParser");
parser.setContentHandler(new MyXMLHandler( ));
parser.parse("document.xml");
```

L'API SAX opère en scannant le document XML de bout en bout et en fournissant des rappels pour les événements qui se produisent. Ces événements peuvent correspondre à la rencontre du début d'un élément, de sa fin, du début d'un attribut, de sa fin, etc. Ici, nous créons une instance `XMLReader` en utilisant le `SAXParser`. Une fois l'instance de parseur créée, nous définissons un gestionnaire de contenu avec la méthode `setContentHandler()`. Le gestionnaire de contenu est une classe qui définit les différentes méthodes de rappel appelées par le parseur SAX lorsque le document XML est parsé. Ici, nous créons une instance de `MyXMLHandler`, une classe que nous allons devoir ensuite implémenter, en guise de gestionnaire. Ensuite, nous appelons la méthode `parse()` et lui passons le nom d'un document XML. Dès lors, le traitement SAX démarre.

Le code suivant présente une implémentation d'exemple de la classe `MyXMLHandler`. La classe `DefaultHandler` que nous étendons est une classe de base par défaut pour les gestionnaires d'événements SAX.

```
class MyXMLHandler extends DefaultHandler {

    public void startElement(String uri, String
    ➥localName, String qname, Attributes attributes) {

    // Traiter le début de l'élément

}

public void endElement(String uri, String localName,
➥String qname) {
```

```
    // Traiter la fin de l'élément
}

public void characters(char[] ch, int start, int length) {
    // Traiter les caractères
}

public MyXMLHandler( )
      throws org.xml.sax.SAXException {
     super( );
    }
}
```

Cette implémentation d'exemple n'implémente que trois méthodes : `startElement()`, `endElement()` et `characters()`. La méthode `startElement()` est appelée par le parseur SAX lorsqu'il rencontre le début d'un élément dans le document XML. De la même manière, la méthode `endElement()` est appelée lorsqu'il rencontre la fin d'un élément. La méthode `characters()` est appelée pour signaler la présence de données de caractère dans un élément. Pour obtenir une description complète de toutes les méthodes qui peuvent être redéfinies dans le gestionnaire SAX, consultez la Java-Doc `DefaultHandler` : **http://java.sun.com/j2se/1.5.0/docs/api/org/xml/sax/helpers/DefaultHandler.html**.

Dans cet exemple, le parseur SAX sous-jacent est Xerces. Nous le définissons dans l'appel de méthode suivant :

```
XMLReader parser = XMLReaderFactory.createXMLReader
➥("org.apache.xerces.parsers.SAXParser");
```

JAXP est conçu pour permettre des implémentations de parseur externes : si vous préférez un autre parseur à Xerces, rien ne vous empêche de l'utiliser avec le code

de cet exemple. Veillez cependant à bien inclure l'implémentation du parseur dans votre chemin de classe.

SAX est généralement plus efficace au niveau de la mémoire que le parseur DOM car le document XML n'est pas tout entier stocké en mémoire. L'API DOM lit le document entier en mémoire pour le traiter.

Parser du XML avec DOM

```
File file = new File("document.xml");
DocumentBuilderFactory f =
  ➥DocumentBuilderFactory.newInstance();
DocumentBuilder p = f.newDocumentBuilder();
Document doc = p.parse(file);
```

Dans cet exemple, nous utilisons les trois classes `DocumentBuilderFactory`, `DocumentBuilder` et `Document` pour démarrer le parsing d'un document XML avec un parseur DOM. Le parsing s'opère avec la classe `DocumentBuilder`. Cette dernière définit l'API permettant d'obtenir des instances de `Document` DOM à partir d'un document XML. La classe `DocumentBuilder` peut parser du XML depuis une variété de sources d'entrée, dont des `InputStream`, des `File`, des `URL` et des `SAXInputSources`. Ici, nous parsons le XML à partir d'une source d'entrée `File`. La méthode `parse()` de la classe `DocumentBuilder` parse le document XML et retourne un objet `Document`.

L'objet `Document` représente le DOM du document XML. Cette instance `Document` peut ensuite être utilisée pour accéder aux composants du document XML, comme ses entités, ses éléments, ses attributs, etc.

L'objet `Document` est un conteneur pour une collection hiérarchique d'objets `Node` qui représente la structure du document XML. Les nœuds ont un parent, des enfants ou des attributs associés. Le type `Node` contient trois sous-classes qui représentent les principaux composants du document XML : `Element`, `Text` et `Attr`. Considérons maintenant un exemple de parsing d'un DOM avec la classe `Document`. Voici le document XML d'exemple que nous allons utiliser :

```
<Location>
   <Address>
    <City>Flat Rock</City>
    <State>Michigan</State>
   </Address>
</Location>
```

En supposant que nous avons déjà obtenu une instance `Document` avec la technique de parsing présentée dans l'exemple de départ, il suffira d'utiliser le code Java suivant pour extraire les valeurs de texte de villes (`city`) et d'états (`state`) :

```
NodeList list = document.getElementsByTagName("City");
Element cityEl = (Element)list.item(0);
String city = ((Text)cityEl.getFirstChild()).getData();
NodeList list = document.getElementsByTagName("State");
Element stateEl = (Element)list.item(0);
String state = ((Text)stateEl.getFirstChild()).getData();
```

La méthode `getElementsByTagName()` retourne un `NodeList` contenant tous les éléments qui correspondent au nom passé. Notre document d'exemple ne contient qu'un seul élément `City` et un seul élément `State` : nous récupérons

donc simplement le premier élément (index zéro) de la liste de nœuds et le transtypons en un `Element`. Les éléments `City` et `State` possèdent chacun un enfant, de type `Text`. Nous utilisons la méthode `getData()` du type `Text` pour récupérer la valeur de ville (`city`) et d'état (`state`).

A la différence du parseur SAX, le parseur DOM lit le document XML entier en mémoire, le parse et le traite à cet endroit. Ce procédé est moins efficace en termes de consommation de mémoire que celui du parseur SAX qui ne stocke pas le document XML entier en mémoire mais le scanne progressivement à la manière d'un flux.

Utiliser une DTD pour vérifier un document XML

```
DocumentBuilderFactory factory =
    ➥DocumentBuilderFactory.newInstance();
factory.setValidating(true);
DocumentBuilder builder = factory.newDocumentBuilder();
```

Les DTD (*Document Type Definition*) sont des fichiers qui définissent la manière dont un document XML particulier doit être structuré. Une DTD peut ainsi spécifier quels éléments et quels attributs sont autorisés dans un document. Les documents XML qui se conforment à une DTD sont considérés être valides. Les documents XML syntaxiquement corrects mais qui ne se conforment pas à une DTD sont simplement dits bien formés.

Pour valider un document avec une DTD, vous devez appeler la méthode `setValidating()` de l'instance `DocumentBuilderFactory` et lui passer la valeur `true`. Tous les documents XML parsés sont ensuite validés par rapport

aux DTD spécifiées dans leur en-tête. Voici une déclaration de DTD classique en haut d'un document XML :

```
<!DOCTYPE people SYSTEM "file:baseball.dtd">
```

Cette déclaration attache le fichier `baseball.dtd` stocké dans le système de fichiers local sous forme de DTD au document XML dans lequel elle est déclarée.

Lorsque vous spécifiez la validation DTD, une exception est lancée depuis la méthode `parse()` de la classe `DocumentBuilder` si le document XML que vous parsez ne se conforme pas à la DTD.

Le standard XML Schema est une technologie plus récente qui offre les mêmes avantages que les DTD. Les schémas XML définissent le balisage attendu des documents XML, comme le font les DTD. L'avantage des documents de schéma tient cependant à ce qu'ils sont eux-mêmes des documents XML et que vous n'avez donc pas besoin d'un autre parseur pour les lire, alors que les DTD ne sont pas des documents XML valides. Les DTD respectent la syntaxe XBNF (*eXtended Backus-Naur Form*). Pour utiliser un schéma, vous devez utiliser la méthode `setSchema()` du `DocumentBuilderFactory` au lieu de la méthode `setValidating()`, comme ceci :

```
DocumentBuilderFactory factory =
    ➥DocumentBuilderFactory.newInstance();
factory.setSchema(schema);
DocumentBuilder builder = factory.newDocumentBuilder();
```

La méthode `setSchema()` prend une instance d'un objet `Schema`. Nous n'entrerons pas ici dans les considérations de détail concernant l'utilisation des schémas, mais vous pouvez consulter la JavaDoc de `DocumentBuilderFactory` pour plus d'informations sur l'implémentation des schémas

en Java, à l'adresse suivante : **http://java.sun.com/j2se/1.5.0/docs/api/javax/xml/parsers/DocumentBuilderFactory.html**.

Pour plus d'informations sur les schémas en général, consultez la documentation XML Schema à l'adresse : **http://www.w3.org/TR/xmlschema-1/**.

Créer un document XML avec DOM

```
DocumentBuilderFactory fact =
  ➥DocumentBuilderFactory.newInstance( );
DocumentBuilder builder = fact.newDocumentBuilder( );
Document doc = builder.newDocument( );
Element location = doc.createElement("Location");
doc.appendChild(location);
Element address = doc.createElement("Address");
location.appendChild(address);
Element city = doc.createElement("City");
address.appendChild(city);
line.appendChild(doc.createTextNode("Flat Rock"));
Element state = doc.createElement("State");
address.appendChild(state);
state.appendChild(doc.createTextNode("Michigan"));
((org.apache.crimson.tree.XmlDocument)doc)
 ➥.write(System.out);
```

Cet exemple utilise l'API DOM et JAXP pour créer un document XML. Le segment XML créé est le suivant :

```
<Location>
    <Address>
      <City>Flat Rock</City>
      <State>Michigan</State>
    </Address>
</Location>
```

La principale classe utilisée ici est la classe `org.w3c.dom.Document`. Elle représente le DOM d'un document XML. Nous créons une instance de la classe `Document` en utilisant un `DocumentBuilder` obtenu à partir d'un `DocumentBuilderFactory`. Chaque élément du document XML est représenté dans le DOM sous forme d'instance `Element`. Dans le document XML que nous créons, nous avons construit des objets `Element` appelés `Location`, `Address`, `City` et `State`. Nous ajoutons l'élément de niveau racine (`Location`) à l'objet de document avec la méthode `appendChild()` de l'objet `Document`. La classe `Element` contient elle aussi une méthode `appendChild()` que nous utilisons pour construire la hiérarchie du document sous l'élément racine.

L'API DOM permet tout aussi facilement de créer un `Element` avec des attributs. Le code suivant peut être utilisé pour ajouter un attribut `"id"` possédant la valeur `"home"` à l'élément `Location` :

```
location.setAttribute("id","home");
```

Dans cet exemple, le parseur DOM sous-jacent est Crimson. Cette implémentation apparaît dans la dernière ligne, reproduite ici :

```
((org.apache.crimson.tree.XmlDocument)doc)
➥.write(System.out);
```

JAXP est conçu pour supporter les implémentations de parseur externes : si vous préférez un autre parseur à Crimson, rien ne vous empêche de l'utiliser avec le code de cet exemple. Veillez cependant à bien inclure l'implémentation du parseur dans votre chemin de classe.

L'API JDOM est une alternative à l'API JAXP pour le travail avec les documents XML. Il s'agit d'un projet open source standardisé par le biais du JCP (*Java Community*

Process), sous la référence JSR 102. Pour plus d'informations sur l'API JDOM, consultez le site **www.jdom.org**. JDOM propose une API Java native en remplacement de l'API DOM standard pour lire et créer des documents XML. De nombreux développeurs trouvent l'API JDOM plus facile à utiliser que l'API DOM lors de la création de documents XML.

Transformer du XML avec des XSLT

```
StreamSource input =
  ➥new StreamSource(new File("document.xml"));
StreamSource stylesheet =
  ➥new StreamSource(new File("style.xsl"));
StreamResult output = new StreamResult(new File("out.xml"));
TransformerFactory tf = TransformerFactory.newInstance();
Transformer tx = tf.newTransformer(stylesheet);
tx.transform(input, output);
```

Le XSLT est un standard pour la transformation des documents XML qui permet d'en modifier la structure à l'aide de feuilles de style XSL. Le paquetage `javax.xml.transform` contient l'API permettant d'utiliser des transformations XSLT standard en Java. XSL est l'acronyme de *eXtensible Stylesheet Language*. XSLT est l'acronyme de *XSL Transformation* (transformation XSL), un langage qui permet de restructurer complètement les documents XML. Lorsque vous utiliserez des XSLT, vous aurez en général un document XML d'entrée et une feuille de style XSL d'entrée et produirez en les combinant un document XML de sortie. Le type du document de sortie ne se limite cependant pas au XML. Bien d'autres sortes de documents de sortie peuvent être créées à l'aide de transformations XSL.

Dans cet exemple, nous créons des instances `StreamSource` pour les documents utilisés en entrée du processus de transformation : le document XML à transformer et la feuille de style XSL contenant les instructions de transformation. Nous créons aussi un objet `StreamResult` qui servira à recueillir le résultat de l'écriture du document de sortie. Nous obtenons ensuite une instance `Transformer` générée à partir d'une instance `TranformerFactory`.

Le flux de feuille de style est ensuite passé à la méthode `newTranformer()` de l'objet `TransformerFactory` pour créer un objet `Transformer`. Pour finir, nous appelons la méthode `transform()` du `Transformer` afin de transformer notre document XML d'entrée en un document de sortie mis en forme avec la feuille de style sélectionnée.

Le XSL peut être une technologie très efficace pour les développeurs. Si votre application Web doit être accessible à partir d'une variété de périphériques différents, comme des assistants personnels, des navigateurs Web et des téléphones cellulaires, vous pourrez ainsi utiliser des XSLT pour transformer votre sortie afin d'en adapter le format à chacun des périphériques concernés, sans programmer chaque fois une sortie séparée. Les XSLT sont aussi très utiles pour créer des sites multilingues. La sortie XML peut en effet être transformée en plusieurs langages grâce à des transformations XSLT.

15

Utiliser des threads

Le threading désigne la méthode utilisée par une application logicielle pour accomplir plusieurs processus à la fois. En Java, un thread est une unité d'exécution programme qui s'exécute simultanément à d'autres threads.

Les threads sont fréquemment utilisés dans les applications d'interface utilisateur graphique (GUI). Dans une application de ce type, un thread peut écouter l'entrée du clavier ou d'autres périphériques de saisie pendant qu'un autre traite la commande précédente.

La communication réseau implique aussi souvent l'usage du multithreading. En programmation réseau, un thread peut écouter les requêtes de connexion, pendant qu'un autre traite une requête précédente. Les minuteurs utilisent aussi couramment les threads. Ils peuvent être démarrés sous forme de thread s'exécutant indépendamment du reste de l'application. Dans tous ces exemples, le multithreading permet à une application de poursuivre le traitement tout en exécutant une autre tâche qui peut prendre plus de temps et provoquerait de longs délais en l'absence de multithreading.

Le Java facilite grandement l'écriture d'applications multithreadées. La conception d'applications multithreadées était très complexe en C, mais en Java, tout est bien plus simple.

Lancer un thread

```
public class MyThread extends Thread {
  public void run() {
    // Exécuter des tâches
  }
}

// Code pour utiliser MyThread
new MyThread().start();
```

Il existe deux techniques principales pour écrire du code qui s'exécute dans un thread séparé. Vous pouvez implémenter l'interface `java.lang.Runnable` ou étendre la classe `java.lang.Thread`. Dans l'un ou l'autre cas, vous devez implémenter une méthode `run()` qui contient le code à exécuter dans le thread.

Dans cet exemple, nous avons étendu la classe `java.lang.Thread`. A l'emplacement où nous souhaitons démarrer le thread, nous instancions notre classe `MyThread` et appelons la méthode `start()` héritée de la classe `Thread`.

Voici maintenant le code permettant d'exécuter un thread avec l'autre technique, en implémentant l'interface `Runnable` :

```
public class MyThread2 implements Runnable {
   public void run() {
     // Exécuter des tâches
   }
}

// Code pour utiliser MyThread2
Thread t = new Thread(MyThread2);
t.start();
```

L'interface `Runnable` est en général implémentée lorsqu'une classe étend une autre classe et ne peut donc étendre la classe `Thread` également. Le Java ne supporte que l'héritage unique : une classe ne peut étendre deux classes différentes. La méthode à l'intérieur de laquelle nous démarrons le thread est ici légèrement différente. Au lieu d'instancier la classe précédemment définie, comme nous l'avons fait en étendant l'interface `Thread`, nous instancions un objet `Thread` et passons la classe implémentant `Runnable` en paramètre au constructeur `Thread`. Ensuite, nous appelons la méthode `start()` de `Thread`, qui démarre le thread et planifie son exécution.

L'un des autres moyens pratiques de créer un thread consiste à implémenter l'interface `Runnable` en utilisant une classe interne anonyme, comme ceci :

```
public class MyThread3 {
    Thread t;

    public static void main(String argv[]) {
      new MyThread3();
    }

    public MyThread3() {
      t = new Thread(new Runnable( ) {
        public void run( ) {
          // Exécuter des tâches
        }
      });
      t.start( );
    }
}
```

Dans cet exemple, tout le code est contenu à l'intérieur d'une unique classe et se trouve donc parfaitement encapsulé. On peut ainsi mieux voir ce qui se passe. Notre implémentation `Runnable` est définie sous forme de classe interne au lieu de créer explicitement une classe qui implémente l'interface `Runnable`. Cette solution est idéale pour les petites méthodes qui n'interagissent que très peu avec des classes externes.

Arrêter un thread

```
public class StoppableThread extends Thread {
  private boolean done = false;

  public void run( ) {
    while (!done) {
      System.out.println("Thread running");
      try {
        sleep(500);
      }
      catch (InterruptedException ex) {
        // Ne rien faire
      }
    }
    System.out.println("Thread finished.");
  }

  public void shutDown( ) {
    done = true;
  }
}
```

Si vous souhaitez créer un thread que vous pourrez arrêter avant la fin de son exécution (autrement dit, avant le retour de la méthode `run()`), la meilleure solution consiste à utiliser un drapeau booléen dont vous testerez l'état au début

d'une boucle globale. Dans cet exemple, nous créons un thread en étendant la classe `Thread` avec notre classe `StoppableThread`. A l'intérieur de la méthode `run()`, nous créons une boucle `while` qui vérifie l'état d'un drapeau booléen `done`. Tant que le drapeau `done` vaut `false`, le thread continue. Pour arrêter le thread, il suffit à un processus externe de positionner le drapeau à `true`. La boucle `while` de la méthode `run()` quitte alors et termine ce thread.

La classe `Thread` contient une méthode `stop()` que l'on peut être tenté d'utiliser pour arrêter le thread, mais Sun en déconseille l'usage, car si votre thread opère sur un objet de structure de données et que vous appelez soudainement sa méthode `stop()`, les objets peuvent être laissés dans un état incohérent. Si d'autres threads attendent que cet objet particulier soit libéré, ils risquent alors de se bloquer et d'attendre indéfiniment. Des problèmes d'interblocage peuvent avoir lieu. La méthode `stop()` est également déconseillée depuis le JDK 1.2. Si vous utilisez la méthode `stop()` dans l'un de ces JDK, le compilateur génère des avertissements à ce sujet.

L'article de référence (en anglais) à l'adresse suivante explique les raisons pour lesquelles la méthode `stop()` est déconseillée : **http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html**.

Attendre qu'un thread se termine

```
Thread t = new Thread(MyThread);
t.start();
// Réaliser d'autres opérations
t.join();
// Continue après que le thread t se termine
```

Dans certains cas, vous pourrez souhaiter qu'un thread d'exécution attende qu'un autre thread ait terminé avant de poursuivre. La jonction des threads est une méthode courante pour interrompre un thread jusqu'à ce qu'un autre thread ait achevé son travail. Dans cet exemple, nous démarrons le thread `t` de l'intérieur du thread qui exécute ces lignes de code. Des opérations sont ensuite exécutées, puis nous appelons la méthode `join()` de l'objet de thread lorsque nous souhaitons arrêter l'exécution du thread en attendant que le thread `t` en ait fini. Lorsque `t` a fini, l'exécution se poursuit jusqu'aux instructions qui suivent la ligne dans laquelle nous avons appelé la méthode `join()`. Si le thread `t` a déjà complètement terminé lorsque nous appelons `join()`, cette dernière retourne immédiatement l'exécution.

Une autre forme de la méthode `join()` prend un paramètre `long` contenant une valeur en millisecondes. Lorsque cette méthode est utilisée, le thread appelant attend au maximum le nombre de millisecondes spécifié avant de continuer, même si le thread sur lequel la méthode `join()` est appelée n'a pas fini. Enfin, une troisième implémentation de la méthode `join()` prend deux paramètres, une valeur `long` en millisecondes et une valeur `int` en nanosecondes. Cette méthode se comporte exactement comme la version à un paramètre, sauf que les valeurs en millisecondes et en nanosecondes sont additionnées pour déterminer la durée pendant laquelle le thread appelant doit attendre avant de continuer. Cette méthode offre un contrôle plus fin sur le temps d'attente.

Synchroniser des threads

```
public synchronized void myMethod() {
  // Exécuter quelque chose
}
```

La synchronisation est utilisée pour empêcher que plusieurs threads puissent accéder simultanément à des sections spécifiques du code. Le mot-clé `synchronized` qui apparaît dans cet exemple permet de synchroniser une méthode ou un bloc de code afin qu'un seul thread puisse l'exécuter à la fois. Dans le contexte de cet exemple, si un thread exécute actuellement `myMethod()`, tous les autres threads qui tentent d'exécuter la même méthode sur la même instance d'objet sont bloqués à l'extérieur de la méthode jusqu'à ce que le thread courant termine son exécution et la retourne de `myMethod()`.

Dans le cas des méthodes non statiques, la synchronisation s'applique uniquement à l'instance d'objet sur laquelle un autre thread exécute la méthode. Les autres threads gardent la possibilité d'exécuter la même méthode mais sur une instance différente. Au niveau de l'instance, le verrouillage s'applique à toutes les méthodes synchronisées de l'instance. Aucun thread ne peut appeler la moindre méthode synchronisée d'une instance dont un thread exécute déjà une méthode synchronisée. Dans le cas des méthodes statiques, seul un thread à la fois peut exécuter la méthode.

Le mot-clé `synchronized` peut également être appliqué à des blocs de code, sans nécessairement concerner l'ensemble d'une méthode. Le bloc de code suivant opère une synchronisation de ce type :

```
synchronized(myObject) {
    // Faire quelque chose avec myObject
}
```

Lors de la synchronisation d'un bloc de code, vous devez spécifier l'objet sur lequel la synchronisation doit être opérée. Bien souvent, le but est d'opérer la synchronisation sur l'objet qui contient le bloc de code, ce qui peut être fait en passant l'objet `this` comme objet à synchroniser, comme ceci :

```
synchronized(this) {
   // Faire quelque chose
}
```

L'objet passé au mot-clé `synchronized` est verrouillé lorsqu'un thread exécute le bloc de code qu'il entoure.

La synchronisation est généralement utilisée lorsque l'accès concurrentiel par plusieurs threads peut risquer d'endommager des données partagées.

Une classe dite *thread-safe* garantit qu'aucun thread n'utilise un objet qui se trouve dans un état incohérent. Le bloc de code suivant présente un exemple de classe qui risquerait d'être problématique si elle n'était pas rendue thread-safe en appliquant le mot-clé `synchronized` à la méthode `adjust()`. En général, les classes qui possèdent des membres de données d'instance sont susceptibles de poser des problèmes dans les environnements multithreadés. A titre d'exemple, supposons que deux threads exécutent la méthode `adjust()` et que cette dernière ne soit pas synchronisée. Le thread A exécute la ligne `size=size+1` et se trouve interrompu juste après la lecture de la valeur `size`, mais avant d'attribuer la nouvelle valeur à `size`.

Le thread B s'exécute maintenant et appelle la méthode `reset()`. Cette méthode positionne la variable `size` à 0. Le thread B est ensuite interrompu et retourne le contrôle au thread A, qui reprend maintenant l'exécution de l'instruction `size=size+1`, en incrémentant la valeur de `size` d'une unité. Au final, la méthode `reset()` ne paraît pas avoir été appelée. Ses effets sont contrecarrés par les

imprévus du multithreading. Si le mot-clé `synchronized` est appliqué à ces méthodes, ce cas de figure ne se produit plus, car un seul thread est alors autorisé à exécuter l'une ou l'autre des méthodes. Le deuxième thread doit attendre que le premier ait terminé l'exécution de la méthode.

```
public class ThreadSafeClass {
    private int size;

    public synchronized void adjust() {
      size = size + 1;
       if (size >= 100) {
        size = 0;
       }
      }

    public synchronized void reset() {
      size = 0;
    }
}
```

La programmation *thread-safe* ne s'applique qu'à une application qui possède plusieurs threads. Si vous écrivez une application qui n'utilise pas de multithreading, vos soucis s'envolent. Avant d'opter pour ce choix, considérez cependant aussi que votre application ou votre composant peuvent être réutilisés à d'autres endroits. Quand vous n'utilisez qu'un seul thread, posez-vous cette question : est-il possible qu'un autre projet utilise ce composant dans un environnement multithreadé ?

La synchronisation peut être utilisée pour rendre un objet thread-safe, mais n'oubliez pas qu'elle implique un compromis en termes de performances. L'exécution des méthodes synchronisées est conséquemment plus lente en raison de la surcharge liée au verrouillage des objets. Veillez ainsi à ne synchroniser que les méthodes qui requièrent véritablement d'être *thread-safe*.

Suspendre un thread

```
MyThread thread = new MyThread();
thread.start();
while (true) {

   // Quelques tâches…

synchronized (thread) {
    thread.doWait = true;
  }

   // Quelques tâches…

   synchronized (thread) {
    thread.doWait = false;
    thread.notify();
   }
}

class MyThread extends Thread {
   boolean doWait = false;
   public void run() {
    while (true) {
      // Quelques tâches…
      synchronized (this) {
       while (doWait) {
         wait();
         }
         catch (Exception e) {
         }
       }
      }
    }
   }
}
```

Cet exemple montre comment suspendre un thread depuis un autre thread. Nous utilisons la variable `doWait` comme drapeau. Dans la méthode `run()` de `MyThread`, nous vérifions l'état de ce drapeau après avoir réalisé une tâche dans une boucle afin de déterminer si nous devons suspendre l'exécution du thread. Si le drapeau `doWait` vaut `true`, nous appelons la méthode `Object.wait()` pour suspendre l'exécution du thread.

Lorsque nous souhaitons relancer le thread, nous positionnons le drapeau `doWait` à `false` et appelons la méthode `thread.Notify()` pour relancer le thread et poursuivre sa boucle d'exécution.

La suspension du thread est très simple à réaliser, comme le montre le fragment suivant :

```
long numMilliSecondsToSleep = 5000;

Thread.sleep(numMilliSecondsToSleep);
```

Ce code suspend le thread courant pendant 5 000 millisecondes, soit 5 secondes. En plus de ces méthodes, deux méthodes appelées `Thread.suspend()` et `Thread.resume()` fournissent un mécanisme pour la suspension des threads, mais elles sont déconseillées. Elles peuvent en effet souvent créer des interblocages. Nous ne les mentionnons ici que pour signaler qu'il convient de ne pas les utiliser.

Lister tous les threads

```
public static void listThreads() {
  ThreadGroup root =
     Thread.currentThread().getThreadGroup().getParent();
  while (root.getParent() != null) {
   root = root.getParent();
  }
  visitGroup(root, 0);
}

public static void visitGroup(ThreadGroup group, int level) {
  int numThreads = group.activeCount();
  Thread[] threads = new Thread[numThreads];
  group.enumerate(threads, false);
  for (int i=0; i<numThreads; i++) {
   Thread thread = threads[i];
   printThreadInfo(thread);
  }

  int numGroups = group.activeGroupCount();
  ThreadGroup[] groups = new ThreadGroup[numGroups];
  numGroups = group.enumerate(groups, false);

  for (int i=0; i<numGroups; i++) {
   visitGroup(groups[i], level+1);
  }
}
private static void printThreadInfo(Thread t) {
  System.out.println("Thread: " + t.getName( ) +
  ➥" Priority: " + t.getPriority( ) + (t.isDaemon( )?"
  ➥Daemon":"") + (t.isAlive( )?"":" Not Alive"));
}
```

Cet exemple liste tous les threads en cours d'exécution. Chaque thread réside dans un groupe de threads et chaque groupe de threads peut contenir des threads et d'autres groupes de threads. La classe `ThreadGroup` permet de

regrouper des threads et d'appeler des méthodes qui affectent tous les threads dans le groupe de threads. Les `ThreadGroup` peuvent également contenir des `ThreadGroup` enfants. Les `ThreadGroup` organisent ainsi tous les threads en une hiérarchie complète.

Dans cet exemple, nous parcourons en boucle tous les groupes de threads afin d'imprimer des informations concernant chacun des threads. Nous commençons par retrouver le groupe de threads racine. Ensuite, nous utilisons la méthode `visitGroup()` pour consulter de manière récursive chaque groupe de threads situé sous le groupe racine. Dans la méthode `visitGroup()`, nous énumérons d'abord tous les threads contenus dans le groupe puis appelons la méthode `printThreadInfo()` pour imprimer le nom, la priorité et l'état (démon ou non, vivant ou non) de chaque thread. Après avoir parcouru en boucle tous les threads dans le groupe courant, nous énumérons tous les groupes qu'il contient et opérons un appel récursif à la méthode `visitGroup()` pour chaque groupe. Cet appel de méthode se poursuit jusqu'à ce que tous les groupes et tous les threads aient été énumérés et que les informations concernant chacun des threads aient été imprimées.

Les groupes de threads sont souvent utilisés pour regrouper des threads liés ou similaires, par exemple selon la fonction qu'ils réalisent, leur provenance ou le moment où ils doivent être démarrés et arrêtés.

16

Programmation dynamique par réflexion

La réflexion est un mécanisme permettant de découvrir à l'exécution des données concernant un programme. En Java, elle permet de découvrir des informations concernant des champs, des méthodes et des constructeurs de classes. Vous pouvez aussi opérer sur les champs et méthodes que vous découvrez de cette manière. La réflexion permet ainsi de réaliser en Java ce que l'on appelle couramment une *programmation dynamique*.

La réflexion s'opère en Java à l'aide de l'API Java Reflection. Cette API est constituée de classes dans les paquetages `java.lang` et `java.lang.reflect`.

Entre autres possibilités, l'API Java Reflection permet notamment :

- de déterminer la classe d'un objet ;
- d'obtenir des informations concernant des modificateurs, des champs, des méthodes, des constructeurs et des superclasses ;
- de retrouver les déclarations de constantes et de méthodes appartenant à une interface ;
- de créer une instance d'une classe dont le nom n'est pas connu jusqu'à l'exécution ;
- de retrouver et de définir la valeur d'un champ d'objet ;
- d'invoquer une méthode sur un objet ;
- de créer un nouveau tableau, dont la taille et le type de composant sont inconnus jusqu'au moment de l'exécution.

L'API Java Reflection est couramment utilisée pour créer des outils de développement tels que des débogueurs, des navigateurs de classes et des générateurs d'interfaces utilisateur graphiques. Ces types d'outils requièrent souvent d'interagir avec des classes, des objets, des méthodes et des champs sans que l'on puisse savoir lesquels dès la compilation. L'application doit alors retrouver ces éléments en cours d'exécution et y accéder de manière dynamique.

Obtenir un objet Class

```
MyClass a = new MyClass();
a.getClass();
```

L'opération la plus simple en programmation réflexive consiste à obtenir un objet `Class`. Une fois l'instance

d'objet `Class` récupérée, il est ensuite possible d'obtenir toutes sortes d'informations concernant la classe et même de la manipuler. Dans cet exemple, nous utilisons la méthode `getClass()` pour obtenir un objet `Class`. Cette méthode est souvent utile avec une instance d'objet dont vous ne connaissez pas la classe de provenance.

Plusieurs autres approches permettent d'obtenir un objet `Class`. Dans le cas d'une classe dont le nom de type est connu à la compilation, il existe un moyen plus simple d'obtenir une instance de classe. Vous devez simplement utiliser le mot-clé de compilateur `.class`, comme ceci :

```
Class aclass = String.class;
```

Si le nom de la classe n'est pas connu à la compilation et se trouve seulement disponible à l'exécution, vous devez utiliser la méthode `forName()` pour obtenir un objet `Class`. Par exemple, la ligne de code suivante crée un objet `Class` associé à la classe `java.lang.Thread` :

```
Class c = Class.forName("java.lang.Thread");
```

Vous pouvez aussi utiliser la méthode `getSuperClass()` sur un objet `Class` afin d'obtenir un objet `Class` représentant la superclasse de la classe réfléchie. Par exemple, dans le code suivant, l'objet `Class a` réfléchit la classe `TextField` et l'objet `Class b` réfléchit la classe `TextComponent` car `TextComponent` est la superclasse de `TextField` :

```
TextField textField = new TextField();
Class a = textField.getClass();
Class b = a.getSuperclass();
```

Obtenir un nom de classe

```
Class c = someObject.getClass();
String s = c.getName();
```

Pour obtenir le nom d'un objet `Class`, il suffit d'appeler la méthode `getName()` sur l'objet concerné. L'objet `String` retourné par la méthode `getName()` est un nom de classe pleinement qualifié. Dans cet exemple, si la variable `someObject` est une instance de la classe `String`, le nom retourné par l'appel à `getName()` est :

`java.lang.String`.

Découvrir des modificateurs de classe

```
Class c = someObject.getClass();
int mods = c.getModifiers();
if (Modifier.isPublic(mods))
  System.out.println("public");
if (Modifier.isAbstract(mods))
  System.out.println("abstract");
if (Modifier.isFinal(mods))
  System.out.println("final");
```

Dans les définitions de classe, plusieurs mots-clés appelés *modificateurs* peuvent précéder le mot-clé `class`. Ces modificateurs sont `public`, `abstract` ou `final`. Pour découvrir quel modificateur a été appliqué à une classe, vous devez d'abord obtenir un objet `Class` représentant la classe concernée avec la méthode `getClass()`. Ensuite, vous devez appeler la méthode `getModifiers()` sur l'objet de

classe pour récupérer une valeur `int` codée représentant les modificateurs. Les méthodes statiques de la classe `java.lang.reflect.Modifier` peuvent ensuite être utilisées pour déterminer les modificateurs qui ont été appliqués. Ces méthodes statiques sont `isPublic()`, `isAbstract()` et `isFinal()`.

Info

Si l'un de vos objets de classe peut représenter une interface, il peut aussi être souhaitable d'utiliser la méthode `isInterface()`. Cette méthode retourne `true` si les modificateurs passés incluent le modificateur `interface`. La classe `Modifier` contient des méthodes statiques supplémentaires qui permettent de déterminer quels modificateurs ont été appliqués aux méthodes et variables de la classe, dont les suivantes : `isPrivate()`, `isProtected()`, `isStatic()`, `isSynchronized()`, `isVolatile()`, `isTransient()`, `isNative()` et `isStrict()`.

Trouver des superclasses

```
Class cls = obj.getClass();
Class superclass = cls.getSuperclass();
```

Les ancêtres d'une classe sont appelés ses *superclasses*. Elles peuvent être retrouvées par réflexion. Une fois que vous avez obtenu un objet `Class`, vous pouvez utiliser la méthode `getSuperclass()` pour retrouver la superclasse de la classe. Si la superclasse existe, un objet `Class` est retourné. S'il n'y a pas de superclasse, la méthode retourne `null`. Rappelez-vous que le Java ne supporte que l'héritage unique : chaque classe ne peut donc avoir qu'une seule superclasse.

Plus précisément, chaque classe ne peut avoir qu'une seule superclasse *directe*. En théorie, toutes les classes ancêtres sont considérées être des superclasses. Pour les récupérer toutes, vous devez récursivement appeler la méthode `getSuperclass()` sur chacun des objets `Class` retournés.

La méthode suivante imprime toutes les superclasses associées à l'objet passé :

```
static void printSuperclasses(Object obj) {
    Class cls = obj.getClass();
    Class superclass = cls.getSuperclass();
    while (superclass != null) {
      String className = superclass.getName();
      System.out.println(className);
      cls = superclass;
      superclass = cls.getSuperclass();
    }
}
```

Les EDI (environnements de développement intégré) comme Eclipse incluent souvent un navigateur de classes qui permet au développeur de parcourir visuellement la hiérarchie des classes. La technique que nous venons de présenter est généralement utilisée pour construire ces navigateurs de classes. Pour développer un navigateur de classes visuel, votre application doit pouvoir retrouver les superclasses d'une classe donnée.

Déterminer les interfaces implémentées par une classe

```
Class c = someObject.getClass();
Class[] interfaces = c.getInterfaces();
for (int i = 0; i < interfaces.length; i++) {
  String interfaceName = interfaces[i].getName();
  System.out.println(interfaceName);
}
```

Dans l'exemple précédent, nous avons vu comment retrouver les superclasses associées à une classe donnée. Ces superclasses sont liées au mécanisme d'héritage et d'extension des classes du Java. En plus des possibilités qui vous sont offertes en matière d'extension des classes, le Java vous permet également d'implémenter des interfaces. Les interfaces qu'une classe donnée a implémentées peuvent aussi être retrouvées par réflexion. Une fois que vous avez obtenu un objet `Class`, vous devez utiliser la méthode `getInterfaces()` pour récupérer les interfaces implémentées par la classe. `getInterfaces()` retourne un tableau d'objets `Class`. Chaque objet du tableau représente une interface implémentée par la classe concernée. Vous pouvez utiliser la méthode `getName()` de ces objets `Class` pour récupérer le nom des interfaces implémentées.

Découvrir des champs de classe

```
Class c = someObject.getClass();
Field[] publicFields = c.getFields();
for (int i = 0; i < publicFields.length; i++) {
  String fieldName = publicFields[i].getName();
  Class fieldType = publicFields[i].getType();
  String fieldTypeStr = fieldType.getName();
  System.out.println("Name: " + fieldName);
  System.out.println("Type: " + fieldTypeStr);
}
```

Les champs publics d'une classe peuvent être découverts à l'aide de la méthode `getFields()` de l'objet `Class`.

La méthode `getFields()` retourne un tableau d'objets `Field` contenant un objet par champ public accessible. Les champs publics accessibles retournés ne sont pas nécessairement des champs contenus directement dans la classe avec laquelle vous travaillez. Il peut aussi s'agir des champs contenus :

- dans une superclasse ;
- dans une interface implémentée ;
- dans une interface étendue à partir d'une interface implémentée par la classe.

A l'aide de la classe `Field`, vous pouvez récupérer le nom, le type et les modificateurs du champ. Ici, nous imprimons le nom et le type de chacun des champs. Vous pouvez également retrouver et définir la valeur des champs. Pour plus de détails à ce sujet, consultez les exemples "Récupérer des valeurs de champ" et "Définir des valeurs de champ" de ce chapitre.

Si vous le préférez, vous pouvez aussi récupérer un champ individuel d'un objet au lieu de tous ses champs si vous en

connaissez le nom. L'exemple suivant montre comment récupérer un champ unique :

```
Class c = someObject.getClass();
Field titleField = c.getField("title");
```

Ce code récupère un objet `Field` représentant le champ nommé `"title"`.

Les méthodes `getFields()` et `getField()` ne retournent que les membres de données publics. Si vous souhaitez récupérer tous les champs d'une classe et notamment ses champs privés et protégés, utilisez les méthodes `getDeclaredFields()` ou `getDeclaredField()`. Elles se comportent comme leurs équivalents `getFields()` et `getField()` mais retournent tous les champs en incluant les champs privés et protégés.

Découvrir des constructeurs de classe

```
Class c = someObject.getClass();
Constructor[] constructors = c.getConstructors();
for (int i = 0; i < constructors.length; i++) {
  Class[] paramTypes = constructors[i].getParameterTypes();
  for (int k = 0; k < paramTypes.length; k ++) {
   String paramTypeStr = paramTypes[k].getName();
   System.out.print(paramTypeStr + " ");
  }
  System.out.println();
}
```

La méthode `getConstructors()` peut être appelée sur un objet `Class` afin de récupérer des informations concernant les constructeurs publics d'une classe. Elle retourne un

tableau d'objets `Constructor`, qui peuvent ensuite être utilisés pour retrouver le nom, les modificateurs et les types de paramètre des constructeurs et les exceptions qui peuvent être levées. L'objet `Constructor` possède également une méthode `newInstance()` permettant de créer une nouvelle instance de la classe du constructeur.

Dans cet exemple, nous obtenons tous les constructeurs de la classe `someObject`. Pour chaque constructeur trouvé, nous récupérons un tableau d'objets `Class` représentant tous ses paramètres. A la fin, nous imprimons chacun des types de paramètre pour chaque constructeur.

Info

Le premier constructeur contenu dans le tableau de constructeurs retourné est toujours le constructeur sans argument par défaut lorsque ce dernier existe. S'il n'existe pas, le constructeur sans argument est défini par défaut.

Vous pouvez aussi retrouver directement un constructeur public individuel au lieu de récupérer tous les constructeurs d'un objet pour peu que vous connaissiez les types de ses paramètres. L'exemple suivant montre comment procéder :

```
Class c = someObject.getClass();
Class[] paramTypes = new Class[] {String.class};
Constructor aCnstrct = c.getConstructor(paramTypes);
```

Nous obtenons un objet `Constructor` représentant le constructeur qui prend un unique paramètre `String`. Les méthodes `getConstructors()` et `getConstructor()` ne retournent que les constructeurs publics. Si vous souhaitez retrouver tous les constructeurs d'une classe et notamment ses constructeurs privés, vous devez utiliser les méthodes `getDeclaredConstructors()` ou `getDeclaredConstructor()`.

Celles-ci se comportent comme leurs équivalents `getConstructors()` et `getConstructor()` mais retournent tous les constructeurs en incluant les constructeurs privés.

Découvrir des informations de méthode

```
Class c = someObject.getClass();
Method[] methods = c.getMethods();
for (int i = 0; i < methods.length; i++) {
  String methodName = methods[i].getName();
  System.out.println("Name: " + methodName);
  String returnType = methods[i].getReturnType().getName();
  System.out.println("Return Type: " + returnType);
  Class[] paramTypes = methods[i].getParameterTypes();
  System.out.print("Parameter Types:");
  for (int k = 0; k < paramTypes.length; k ++) {
   String paramTypeStr = paramTypes[k].getName();
   System.out.print(" " + paramTypeStr);
  }
  System.out.println();
}
```

La méthode `getMethods()` peut être appelée sur un objet `Class` afin de récupérer des informations concernant les méthodes publiques d'une classe. Elle retourne un tableau d'objets `Method`, qui peuvent ensuite être utilisés pour retrouver le nom, le type de retour, les types de paramètres, les modificateurs de la classe et les exceptions qui peuvent être levées. La méthode `Method.invoke()` peut également être utilisée pour appeler la méthode. Pour plus d'informations sur l'invocation des méthodes, consultez l'exemple "Invoquer des méthodes" de ce chapitre.

Dans l'exemple précédent, une fois que nous récupérons le tableau de méthodes, nous le parcourons en boucle pour imprimer le nom, le type de retour et une liste des types de paramètres de chacune des méthodes.

Vous pouvez aussi retrouver une méthode publique individuelle au lieu de récupérer toutes les méthodes d'un objet, pourvu que vous connaissiez son nom et les types de ses paramètres. L'exemple suivant montre comment procéder :

```
Class c = someObject.getClass();
Class[] paramTypes =
        ➥new Class[] {String.class, Integer.class};
Method meth = c.getMethod("setValues", paramTypes);
```

Dans cet exemple, nous obtenons un objet `Method` représentant la méthode nommée `setValue` qui prend deux paramètres de type `String` et `Integer`.

Les méthodes `getMethods()` et `getMethod()` dont nous venons de traiter retournent l'ensemble des méthodes publiques auxquelles il est possible d'accéder avec une classe. Des méthodes équivalentes appelées `getDeclaredMethods()` et `getDeclaredMethod()` permettent d'obtenir toutes les méthodes quel que soit leur type d'accès. Elles se comportent exactement de la même manière mais retournent toutes les méthodes de la classe concernée indépendamment de leur type d'accès. Il est ainsi possible d'obtenir les méthodes privées également.

Les EDI (environnements de développement intégré) comme Eclipse incluent souvent un navigateur de classes qui permet au développeur de parcourir visuellement la hiérarchie des classes. La technique que nous venons de présenter est généralement utilisée pour construire ces navigateurs de classes. Pour développer un navigateur de classes visuel, votre application doit avoir un moyen de connaître toutes les méthodes d'une classe donnée.

Retrouver des valeurs de champ

```
Class c = anObject.getClass();
Field titleField = c.getField("title");
String titleVal = (String) titleField.get(anObject);
```

Pour retrouver une valeur de champ, vous devez commencer par récupérer un objet `Field` pour le champ dont vous souhaitez connaître la valeur. Pour plus d'informations sur l'obtention des objets `Field` d'une classe, consultez l'exemple "Découvrir des champs de classe" précédemment dans ce chapitre.

La classe `Field` propose des méthodes spécialisées pour récupérer les valeurs des types primitifs, dont `getInt()`, `getFloat()` et `getByte()`. Pour plus de détails sur les méthodes getter de l'objet `Field`, consultez la JavaDoc (en anglais) à l'adresse suivante : **http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/Field.html**.

Pour retrouver des champs stockés sous forme d'objets et non comme des primitifs, vous devez utiliser la méthode plus générale `get()` et transtyper le résultat de retour sur le type d'objet approprié. Dans cet exemple, nous obtenons le champ nommé `"title"`. Après avoir récupéré le champ sous forme d'objet `Field`, nous obtenons ensuite la valeur du champ en utilisant la méthode `get()` et en transtypant le résultat en type `String`.

Dans cet exemple, nous connaissions le nom du champ dont nous souhaitions retrouver la valeur. Cette valeur pourrait cependant être obtenue sans même connaître son nom à la compilation, en combinant cet exemple avec l'exemple "Découvrir des champs de classe" où nous avons montré comment obtenir des noms de champs. Cette technique pourrait par exemple être utile dans un outil de générateur d'interface utilisateur graphique qui

nécessiterait d'obtenir la valeur de différents champs d'objets de l'interface dont les noms ne seraient pas connus avant l'exécution.

Définir des valeurs de champ

```
String newTitle = "President";
Class c = someObject.getClass();
Field titleField = c.getField("title");
titleField.set(someObject, newTitle);
```

Pour définir une valeur de champ, vous devez d'abord obtenir un objet `Field` pour le champ dont vous souhaitez définir la valeur. Pour plus d'informations sur l'obtention d'objets `Field` à partir d'une classe, consultez l'exemple "Découvrir des champs de classe" précédemment dans ce chapitre. Référez-vous aussi à l'exemple "Retrouver des valeurs de champ" pour plus d'informations sur l'obtention des valeurs de champ.

La classe `Field` possède des méthodes spécialisées pour définir les valeurs des types primitifs, dont `setInt()`, `setFloat()` et `setByte()`. Pour plus de détails sur les méthodes set disponibles pour l'objet `Field`, consultez la JavaDoc (en anglais) à l'adresse : **http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/Field.html**.

Pour positionner des champs stockés sous forme d'objets et non sous forme de types primitifs, vous devez utiliser la méthode `set()` plus générale en passant l'instance d'objet dont vous définissez les valeurs de champ et la valeur de champ comme objet. Dans cet exemple, nous positionnons le champ nommé `"title"`. Après avoir obtenu le champ sous forme d'objet `Field`, nous définissons sa valeur avec `set()`, en passant l'instance d'objet dont nous positionnons les valeurs de champ et la nouvelle valeur pour la chaîne de titre.

Pour cet exemple, nous connaissions le nom du champ dont nous souhaitions définir la valeur. Il est cependant possible de définir la valeur des champs dont vous ne connaissez pas le nom à la compilation, en combinant l'approche de cet exemple avec celle de l'exemple "Découvrir des champs de classe" précédemment dans ce chapitre. Cette technique peut par exemple être utile pour un générateur d'interface utilisateur graphique avec lequel vous devez positionner la valeur de différents champs d'objet d'interface utilisateur graphique dont les noms ne peuvent être connus avant l'exécution.

Les débogueurs vous permettent souvent de modifier la valeur d'un champ au cours d'une session de débogage. Pour implémenter ce type de fonctionnalité, le développeur du programme peut utiliser la technique de cet exemple afin de positionner la valeur des champs car il ne peut connaître à la compilation le champ dont vous souhaiterez positionner la valeur à l'exécution.

Invoquer des méthodes

```
Baseball bbObj = new Baseball();
Class c = Baseball.class;
Class[] paramTypes = new Class[] {int.class, int.class};
Method calcMeth = c.getMethod("calcBatAvg", paramTypes);
Object[] args = new Object[] {new Integer(30),
➥new Integer(100)};
Float result = (Float) calcMeth.invoke(bbObj, args);
```

L'API Reflection permet d'invoquer de manière dynamique des méthodes dont vous ne connaissez pas le nom à la compilation. Vous devez d'abord obtenir un objet `Method` pour la méthode à invoquer. Pour plus d'informations sur l'obtention des objets `Method` à partir d'une

classe, consultez l'exemple "Découvrir des informations de méthode" précédemment dans ce chapitre.

Dans l'exemple précédent, nous tentons d'invoquer une méthode qui calcule une moyenne de réussite en baseball. La méthode nommée `calcBatAvg()` prend deux paramètres entiers, un compte des coups réussis et un compte des "présences à la batte" (*at-bats*). La méthode retourne une moyenne à la batte sous forme d'objet `Float`. Pour l'invoquer, nous suivons ces étapes :

- Nous obtenons un objet `Method` associé à la méthode `calcBatAvg()` de l'objet `Class` qui représente la classe `Baseball`.
- Nous invoquons la méthode `calcBatAvg()` en utilisant la méthode `invoke()` de l'objet `Method`. La méthode `invoke()` prend deux paramètres : un objet dont la classe déclare ou hérite la méthode et un tableau de valeurs de paramètre à passer à la méthode invoquée. Si la méthode est statique, le premier paramètre est ignoré et peut valoir `null`. Si la méthode ne prend aucun paramètre, le tableau d'arguments peut être de longueur nulle ou valoir `null`.

Dans notre exemple, nous passons une instance de l'objet `Baseball` comme premier paramètre à la méthode `invoke()` et un tableau d'objets contenant deux valeurs entières encapsulées en second paramètre. La valeur de retour de la méthode `invoke()` correspond à la valeur retournée par la méthode invoquée – soit ici, la valeur de retour de `calcBatAvg()`. Si la méthode retourne un primitif, la valeur est d'abord encapsulée dans un objet et retournée sous forme d'objet. Si la méthode possède le type de retour `void`, la valeur `null` est retournée. La méthode `calcBatAvg()` retourne une valeur `Float`. Nous transtypons donc l'objet retourné afin d'en faire un objet `Float`.

Cette technique pourrait être utile pour l'implémentation d'un débogueur permettant à l'utilisateur de sélectionner une méthode et de l'invoquer. La méthode sélectionnée ne pouvant être connue avant l'exécution, elle peut être retrouvée de manière réflexive puis invoquée au moyen de cette technique.

Charger et instancier une classe de manière dynamique

```
Class personClass = Class.forName(personClassName);
Object personObject = personClass.newInstance();
Person person = (Person)personObject;
```

Les méthodes `Class.forName()` et `newInstance()` de l'objet `Class` permettent de charger et d'instancier dynamiquement une classe dont vous ne connaissez pas le nom jusqu'à l'exécution. Dans cet exemple, nous chargeons notre classe en utilisant la méthode `Class.forName()` à laquelle nous passons le nom de la classe à charger. `forName()` retourne un objet `Class`.

Nous appelons ensuite la méthode `newInstance()` sur l'objet `Class` pour instancier une instance de la classe. La méthode `newInstance()` retourne un type `Object` général que nous transtypons dans le type attendu.

Cette technique peut être particulièrement utile si vous avez une classe qui étend une classe de base ou implémente une interface et que vous souhaitez stocker le nom de la classe d'extension ou d'implémentation dans un fichier de configuration. L'utilisateur final peut alors ajouter de manière dynamique différentes implémentations sans devoir recompiler l'application. Par exemple, si notre

application incluait le code d'exemple précédent et dans un plug-in, le code suivant, nous pourrions l'amener à instancier dynamiquement un objet `BusinessPerson` à l'exécution en spécifiant le nom de classe complet de l'objet `BusinessPerson` dans un fichier de configuration. Avant d'exécuter notre exemple, nous lirions le nom de classe depuis le fichier de configuration et attribuerions cette valeur à la variable `personClassName`.

```
public class BusinessPerson extends Person {

//Corps de la classe, étends le comportement
    ➥de la classe Person

}
```

Le code de l'application n'inclurait ainsi aucune référence à la classe `BusinessPerson` elle-même. Il ne serait donc nécessaire de coder en dur que la classe de base ou l'interface générique, l'implémentation spécifique pouvant être configurée de manière dynamique à l'exécution en éditant le fichier de configuration.

17

Empaquetage et documentation des classes

Les applications Java sont généralement constituées de nombreuses classes et peuvent parfois même en compter des centaines ou des milliers.

Puisque le Java requiert que chaque classe publique soit définie dans un fichier séparé, vous aurez au moins autant de fichiers que vous avez de classes. Ce foisonnement peut rapidement devenir ingérable lorsqu'il s'agit de travailler avec des classes, de retrouver des fichiers ou d'installer et de distribuer une application. Ce problème a heureusement été anticipé dès la création du Java. Sun a défini un mécanisme d'empaquetage standard permettant de placer les classes liées dans des paquetages. Les paquetages utilisés en Java permettent d'organiser les classes d'après leurs fonctionnalités. Le mécanisme d'empaquetage organise également les fichiers source Java en une structure de répertoires connue définie par rapport aux noms des paquetages utilisés.

Un mécanisme standard en Java est aussi proposé pour empaqueter les classes Java en fichiers d'archive standard. Les applications peuvent être exécutées directement depuis le fichier d'archive et des bibliothèques distribuées sous forme d'archive. Le fichier d'archive Java standard est le fichier JAR, qui possède l'extension `.jar`. Il utilise le protocole d'archivage ZIP. Les fichiers JAR peuvent être extraits en utilisant n'importe quel outil prenant en charge la décompression des archives ZIP. Sun propose également l'outil `jar` pour créer et décompresser des archives JAR. Celui-ci fait partie de la distribution JDK standard. JAR est l'acronyme de *Java Archive*.

Créer un paquetage

```
package com.timothyfisher.book;
```

Dans les applications ou les bibliothèques de grande taille, les classes Java s'organisent généralement sous forme de paquetages. Pour placer une classe dans un paquetage, il vous suffit d'inclure une instruction `package` au début du fichier de classe, comme le montre l'exemple précédent. L'instruction `package` doit correspondre à la première ligne non commentée du fichier de classe. Dans notre exemple, nous attribuons la classe contenue dans le fichier où l'instruction figure au paquetage `com.timothyfisher.book`.

Le nom du paquetage de la classe fait partie de son nom complet. Si nous créons une classe nommée `MathBook` dans le paquetage `com.timothyfisher.book`, le nom complet de la classe est alors `com.timothyfisher.book.MathBook`. Les noms de paquetage régissent également la structure des répertoires dans lesquels les fichiers source

des classes sont stockés. Chaque élément du nom de chemin représente un répertoire. Par exemple, si votre répertoire racine du code source est `project/src`, le code source pour la classe `MathBook` est stocké dans le répertoire suivant :

```
project/src/com/timothyfisher/book/
```

Les bibliothèques Java standard sont toutes organisées dans des paquetages avec lesquels vous devez être familiarisé. Parmi les exemples de paquetages, on peut citer `java.io`, `java.lang` ou bien encore `java.util`.

Les classes stockées dans un paquetage peuvent aussi être importées facilement dans un fichier. Vous pouvez ainsi importer un paquetage entier dans votre fichier source Java avec la syntaxe suivante :

```
import java.util.*;
```

Cette instruction `import` importe toutes les classes contenues dans le paquetage `java.util`. Notez cependant qu'elle n'importe pas les classes contenues dans les sous-paquetages de `java.util`, comme celles contenues dans le paquetage `java.util.logging`. Une instruction `import` séparée est requise pour l'importation de ces classes.

Le Java 5.0 a introduit une nouvelle fonctionnalité liée à l'importation des classes appelée *importations statiques*. Les importations statiques permettent d'importer des membres statiques de classes en leur permettant d'être utilisés sans qualification de classe. Par exemple, pour référencer la méthode `cos()` dans le paquetage `java.lang.Math`, vous pourrez vous y référer de la manière suivante :

```
double val = Math.cos(90);
```

Pour cela, vous devrez importer le paquetage `java.lang.Math` à l'aide d'une importation statique, comme ceci :

```
import static java.lang.Math.*;
```

Vous pouvez faire référence à la méthode `cos()` de la manière suivante :

```
double val = cos(90);
```

Lors de l'exécution d'une application Java depuis la ligne de commande avec l'exécutable java, vous devez inclure le nom de paquetage complet lorsque vous spécifiez la classe exécutable principale. Par exemple, pour exécuter une méthode `main()` dans l'exemple `MathBook` signalé précédemment, vous devrez taper ceci :

```
java com.timothyfisher.book.MathBook
```

Cette commande est exécutée depuis la racine de la structure du paquetage – ici, le répertoire parent du répertoire `com`.

Les classes qui ne sont pas spécifiquement attribuées à un paquetage avec une instruction `package` sont considérées êtres incluses dans un paquetage "par défaut". Le bon usage exige que vous placiez toujours vos classes dans des paquetages définis par vos soins. Les classes qui se trouvent dans le paquetage par défaut ne peuvent pas être importées ni utilisées à l'intérieur des classes des autres paquetages.

Documenter des classes avec JavaDoc

```
javadoc -d \home\html
    -sourcepath \home\src
    -subpackages java.net
```

JavaDoc est un outil permettant de générer de la documentation d'API au format HTML à partir de commentaires placés dans les fichiers de code source Java. L'outil JavaDoc fait partie intégrante de l'installation standard du JDK.

L'exemple présenté ici illustre un type d'usage particulier de l'outil JavaDoc. JavaDoc possède de nombreuses options et de nombreux drapeaux en ligne de commande qui peuvent être utilisés pour documenter des classes et des paquetages. Pour obtenir une description complète des options de JavaDoc, consultez la documentation de Sun (en anglais) à l'adresse **http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/index.html**.

La commande `javadoc` utilisée dans cet exemple génère une documentation JavaDoc pour toutes les classes contenues dans le paquetage `java.net` et tous ses sous-paquetages. Le code source doit se trouver dans le répertoire `\home\src directory`. La sortie de la commande est écrite dans le répertoire `\home\html`.

Voici un exemple classique de commentaire JavaDoc dans un fichier source Java :

```
/**
* Un commentaire décrivant une classe ou une méthode
*
* Les balises spéciales sont précédées par le caractère @
* pour documenter les paramètres de méthode, types de
* retour, nom d'auteur de méthode ou de classe, etc. Voici
* un exemple de paramètre documenté.
* @param input Les données d'entrée pour cette méthode.
*/
```

Les séquences de caractères `/**` et `*/` signalent le début et la fin d'un commentaire JavaDoc.

L'outil JavaDoc produit une sortie analogue à la documentation de classe Java standard que vous aurez inévitablement rencontrée si vous avez consulté des documents Java en ligne par le passé. Le JDK est lui-même documenté avec la documentation JavaDoc. Pour visualiser la JavaDoc du JDK, rendez-vous à l'adresse suivante : **http://java.sun.com/j2se/1.5.0/docs/api/index.html**.

La documentation générée par JavaDoc permet de parcourir facilement les classes qui composent une application ou une bibliothèque. Une page d'index propose une liste de toutes les classes et des liens hypertexte vers chacune d'entre elles. Des index sont également fournis pour chaque paquetage.

La création de la documentation JavaDoc s'intègre souvent au processus de génération des applications. Si vous utilisez l'outil de compilation Ant, il existe ainsi une tâche Ant permettant de générer la JavaDoc dans le cadre de votre processus de génération et de compilation.

La technologie qui permet à JavaDoc de fonctionner a également été utilisée il y a peu pour créer d'autres outils dont les fonctionnalités sortent du simple cadre de la documentation des fichiers Java. L'API Doclet est ainsi utilisée par JavaDoc et des outils tiers. L'un de ces outils tiers, parmi les plus populaires, est le projet open source XDoclet.

XDoclet est un moteur servant à la programmation orientée attribut. Il permet d'ajouter des métadonnées à votre code source afin d'automatiser des tâches telles que la création d'EJB. Pour plus d'informations sur XDoclet, consultez le site **http://xdoclet.sourceforge.net/**.

L'API Taglet est une autre API utile pour le travail avec les commentaires de style JavaDoc qui fait partie du Java standard. Elle permet de créer des programmes appelés Taglets qui peuvent modifier et formater des commentaires de style JavaDoc contenus dans vos fichiers source. Pour plus d'informations sur les Taglets, rendez-vous à l'adresse **http://java.sun.com/j2se/1.4.2/docs/tooldocs/javadoc/taglet/overview.html**.

Archiver des classes avec Jar

```
jar cf project.jar *.class
```

L'utilitaire jar est inclus dans le JDK. Il permet d'empaqueter des groupes de classes en un lot et de créer, mettre à jour, extraire, lister et indexer des fichiers JAR. Avec l'instruction de cet exemple, toutes les classes contenues dans le répertoire courant depuis lequel la commande `jar` est exécutée sont placées dans un fichier JAR nommé `project.jar`.

L'option `c` demande à l'utilitaire jar de créer un nouveau fichier d'archive. L'option `f` est toujours suivie par un nom de fichier spécifiant le nom du fichier JAR à utiliser.

Des applications complètes peuvent être distribuées sous forme de fichiers JAR. Elles peuvent également être exécutées depuis le fichier JAR sans avoir à les extraire au préalable. Pour plus d'informations à ce sujet, consultez l'exemple "Exécuter un programme depuis un fichier JAR" de ce chapitre.

Toutes les classes contenues dans un fichier JAR peuvent aisément être incluses dans le `CLASSPATH` lors de l'exécution

ou de la compilation d'une application ou d'une bibliothèque Java. Pour inclure le contenu d'un fichier JAR dans le `CLASSPATH`, incluez le chemin complet au fichier JAR au lieu du seul répertoire. Par exemple, l'instruction `CLASSPATH` pourrait ressembler à ceci :

```
CLASSPATH=.;c:\projects\fisher.jar;c:\projects\classes
```

Cette instruction inclut toutes les classes contenues dans l'archive `fisher.jar` dans le `CLASSPATH`. Notez bien que pour inclure les classes dans un fichier JAR, vous devez spécifier le nom du fichier JAR dans le chemin de classe. Il ne suffit pas de pointer vers un répertoire contenant plusieurs fichiers JAR comme il est possible de le faire avec les fichiers `.class`.

Pour plus d'informations sur l'utilisation de l'outil jar, consultez la documentation JAR officielle sur le site de Sun à l'adresse **http://java.sun.com/j2se/1.5.0/docs/guide/jar/index.html**.

Exécuter un programme à partir d'un fichier JAR

```
java -jar Scorebook.jar
```

L'exécutable en ligne de commande java permet d'exécuter une application Java empaquetée dans un fichier JAR. Pour cela, vous devez utiliser le commutateur `–jar` à l'exécution de la commande `java`. Vous devez aussi spécifier le nom du fichier JAR contenant l'application à exécuter.

La classe contenant la méthode `main()` que vous souhaitez exécuter doit être déclarée dans un fichier de manifeste.

Par exemple, pour exécuter la classe `com.timothyfisher.Scorebook`, vous pourriez utiliser un fichier de manifeste dont le contenu serait le suivant :

```
Manifest-Version: 1.2
Main-Class: com.timothyfisher.Scorebook
Created-By: 1.4 (Sun Microsystems Inc.)
```

Ce fichier de manifeste devrait alors être placé dans le fichier JAR avec vos classes.

Cette fonctionnalité permet aux développeurs Java de distribuer une application dans un unique fichier JAR et d'inclure un fichier de script comme un fichier BAT Windows ou un script de shell UNIX qui peuvent être utilisés pour lancer l'application à l'aide d'une instruction analogue à celle présentée dans cet exemple.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z

LE GUIDE DE SURVIE

Java®

L'ESSENTIEL DU CODE ET DES COMMANDES

Ce *Guide de survie* vous livre tout le code dont vous avez besoin pour réaliser rapidement et efficacement vos projets de développement en Java.

CONCIS ET MANIABLE

Facile à transporter, facile à utiliser — finis les livres encombrants !

PRATIQUE ET FONCTIONNEL

Plus de 100 fragments de code personnalisables pour programmer du Java fonctionnel dans toutes les situations.

Timothy Fisher est un professionnel du développement de logiciels Java depuis 1997. Il est actuellement consultant pour l'entreprise Compuware Corporation à Détroit dans le Michigan. Il aime écrire sur cette technologie et a contribué aux deux ouvrages *Java Developer's Journal* et *XML Journal*. Tim est également passionné par l'éducation et l'utilisation des technologies Internet avancées dans ce domaine.

Niveau : Intermédiaire
Catégorie : Programmation
Configuration : Multiplate-forme

PEARSON

Pearson Education France
47 bis, rue des Vinaigriers
75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearson.fr

ISBN : 978-2-7440-4004-7

9 782744 040047