



Introduction à JavaTM



VERSION 6

Borland[®]
JBuilderTM

Consultez le fichier DEPLOY.TXT situé dans le répertoire `redist` de JBuilder pour avoir la liste complète des fichiers que vous pouvez distribuer en accord avec votre contrat de licence JBuilder.

Les applications mentionnées dans ce manuel sont brevetées par Borland Software Corporation ou en attente de brevet. Ce document ne donne aucun droit sur ces brevets.

COPYRIGHT © 1997, 2001 Borland Software Corporation. Tous droits réservés. Tous les produits Borland sont des marques commerciales ou des marques déposées de Borland Software Corporation aux Etats-Unis et dans les autres pays. Les autres noms de produits sont des marques commerciales ou des marques déposées de leurs fabricants respectifs.

Pour connaître les conditions et limites des fournisseurs tiers, lisez les Remarques version sur votre CD JBuilder.

JBE0060WWxxxxgsjava 2E2R1001
0102030405-9 8 7 6 5 4 3 2 1
PDF

Table des matières

Chapitre 1		
Introduction	1-1	
Conventions de la documentation	1-3	Utilisation des constructeurs 3-15
Remarque pour les utilisateurs du Macintosh	1-5	Accès aux membres 3-16
Comment contacter le support développeur de Borland	1-6	Tableaux. 3-16
Ressources en ligne.	1-6	
World Wide Web	1-6	Chapitre 4
Groupe de discussion Borland	1-7	Contrôle du langage Java
Groupe de discussion Usenet	1-7	4-1
Rapports de bogues.	1-7	Termes 4-1
		Gestion des chaînes 4-1
Chapitre 2		Transtypage et conversion 4-2
Éléments du langage Java	2-1	Types et instructions de retour 4-3
Termes.	2-1	Instructions de contrôle du déroulement . . . 4-3
Identificateur	2-1	Application des concepts. 4-4
Type de données	2-2	Séquences d'échappement 4-4
Types de données primitifs	2-2	Gestion des chaînes 4-5
Types de données composites	2-3	Détermination des accès 4-6
Chaînes	2-4	Gestion des méthodes. 4-7
Tableaux	2-4	Utilisation des conversions de types 4-8
Variable	2-4	Transtypage implicite 4-8
Littéral	2-4	Conversion explicite. 4-8
Application des concepts	2-5	Contrôle du déroulement. 4-9
Déclaration des variables	2-5	Boucles 4-9
Méthodes	2-6	Instructions de contrôle des boucles. . . 4-11
		Instructions conditionnelles 4-12
Chapitre 3		Gestion des exceptions 4-14
Structure du langage Java	3-1	
Termes.	3-1	Chapitre 5
Mots clés.	3-1	Les bibliothèques des classes Java
Opérateurs	3-2	5-1
Commentaires	3-3	Editions de la plate-forme Java 2 5-1
Instructions	3-5	Standard Edition. 5-2
Blocs de code	3-5	Enterprise Edition. 5-2
Comprendre la portée	3-5	Micro Edition 5-3
Application des concepts	3-7	Paquets de Java 2 Standard Edition. 5-3
Utilisation des opérateurs	3-7	Le paquet du langage : java.lang 5-4
Opérateurs arithmétiques	3-7	Le paquet des utilitaires : java.util. 5-4
Opérateurs logiques	3-9	Le paquet des E/S : java.io 5-5
Opérateurs d'affectation	3-10	Le paquet de texte : java.text 5-5
Opérateurs de comparaison	3-10	Le paquet mathématique : java.math 5-5
Opérateurs au niveau bits	3-11	Le paquet AWT : java.awt 5-6
? : — L'opérateur ternaire.	3-12	Le paquet Swing : javax.swing. 5-6
Utilisation des méthodes	3-13	Les paquets Javax : javax 5-6
Utilisation des tableaux	3-14	Le paquet Applet : java.applet. 5-7
		Le paquet Beans : java.beans 5-7
		Le paquet des réflexions : java.lang.reflect . . 5-8
		Le paquet SQL : java.sql 5-8
		Le paquet RMI : java.rmi 5-9
		Le paquet réseau : java.net 5-9
		Le paquet de sécurité : java.security. 5-9

Chapitre 6	
Programmation orientée objet	
 dans Java	6-1
Classes	6-2
Déclaration et instanciation des classes . . .	6-2
Données membre	6-3
Méthodes de classe	6-3
Constructeurs et finaliseurs	6-4
Etude de cas : Exemple simple d'OOP . . .	6-4
Héritage de classe	6-9
Appel du constructeur du parent . . .	6-12
Modificateurs d'accès	6-12
Accès depuis l'intérieur du paquet	
d'une classe	6-12
Accès depuis l'extérieur d'un paquet . .	6-13
Méthodes d'accès	6-13
Classes abstraites	6-16
Polymorphisme	6-17
Utilisation des interfaces	6-18
Ajout de deux nouveaux boutons	6-22
Exécution de votre application	6-24
Paquets Java	6-25
L'instruction import	6-25
Déclaration des paquets	6-25

Chapitre 7	
Techniques de thread	7-1
Cycle de vie d'un thread	7-2
Personnalisation de la méthode run() . . .	7-2
Sous-classement de la classe Thread . . .	7-2
Implémentation de l'interface Runnable .	7-3
Définition d'un thread	7-5
Démarrage d'un thread	7-5
Rendre un thread non exécutable	7-6
Arrêt d'un thread	7-7
Priorité attribuée aux threads	7-7
Temps partagé	7-7
Threads synchronisés	7-8
Groupe de threads	7-8

Chapitre 8	
Sérialisation	8-1
Pourquoi sérialiser ?	8-1
Sérialisation Java	8-2
Utilisation de l'interface Serializable . . .	8-3
Utilisation des flux de sortie	8-4
Méthodes ObjectOutputStream	8-6
Utilisation des flux d'entrée	8-6
Méthodes InputStream	8-8
Ecriture et lecture des flux d'objets	8-8

Chapitre 9	
Introduction à la machine	
 virtuelle Java	9-1
Sécurité de la machine virtuelle Java	9-3
Le modèle de sécurité	9-3
Le vérificateur Java	9-3
Le gestionnaire de sécurité et le paquet	
java.security	9-4
Le chargeur de classe	9-6
Les compilateurs Just-In-Time	9-7

Chapitre 10	
Utilisation de l'interface native	
 Java (JNI)	10-1
Comment fonctionne l'interface JNI	10-2
Utilisation du mot clé native	10-2
Utilisation de l'outil javah	10-3

Chapitre 11	
Référence rapide du langage Java	11-1
Editions de la plate-forme Java 2	11-1
Bibliothèques des classes Java	11-2
Mots clés	11-3
Types de données, types de retour	
et termes	11-3
Paquets, classes, membres et interfaces . .	11-3
Modificateurs d'accès	11-4
Boucles et contrôles de boucles	11-4
Gestion des exceptions	11-5
Réservés	11-5
Conversion et transtypage des types	
de données	11-5
Primitif en primitif	11-6
Primitif en chaîne	11-7
Primitif en référence	11-8
Chaîne en primitif	11-10
Référence en primitif	11-12
Référence en référence	11-14
Séquences d'échappement	11-18
Opérateurs	11-18
Opérateurs de base	11-18
Opérateurs arithmétiques	11-19
Opérateurs logiques	11-20
Opérateurs d'affectation	11-20
Opérateurs de comparaison	11-21
Opérateurs au niveau bits	11-21
Opérateur ternaire	11-22

Index	I-1
--------------	------------

Introduction

Java est un langage de programmation *orienté objet*. Le passage à la programmation orientée objet (POO) à partir d'autres modèles de programmation peut être difficile. Java est axé sur la création d'objets (structures de données ou comportements) qui peuvent être répartis et manipulés par le programme.

Comme d'autres langages de programmation, Java prend en charge la lecture et l'écriture des données dans différents dispositifs d'entrée et de sortie. Java utilise des processus qui augmentent les performances des entrées/sorties, facilitent l'internationalisation et offrent un meilleur support pour les plates-formes non UNIX. Java examine votre programme au fil de l'exécution et libère automatiquement la mémoire qui n'est plus nécessaire. Cela signifie que vous n'avez pas besoin de suivre les pointeurs de mémoire ni de libérer manuellement la mémoire. Cette fonctionnalité diminue les risques de panne du programme et ne laisse pas la possibilité de mal utiliser la mémoire.

Ce manuel est une introduction à Java conçue pour les programmeurs qui utilisent d'autres langages de programmation. Il présente au lecteur les principaux éléments de la programmation Java et fournit des liens et des suggestions de lecture pour un approfondissement ultérieur. Vous pouvez commander la plupart des titres chez Fatbrain (<http://fatbrain.com/>). Les titres non liés peuvent ne plus être édités, mais sont encore utiles et généralement disponibles. Pour une liste plus complète des ressources, voir "Pour mieux connaître Java" dans *Introduction à JBuilder*.

Ce manuel comprend les chapitres suivants :

- Syntaxe Java : [Chapitre 2, “Eléments du langage Java”](#), [Chapitre 3, “Structure du langage Java”](#), et [Chapitre 4, “Contrôle du langage Java”](#)

Ces trois chapitres définissent la syntaxe Java de base et introduisent les concepts de la programmation orientée objet. Ils suggèrent également d’autres ressources. Ces suggestions sont simplement des points d’entrée. Quiconque apprenant Java dans les livres aura besoin pour cela de trois ou quatre manuels, car chacun met l’accent sur différents points et aucun ne couvre la totalité du langage.

Chaque section est divisée en deux parties principales : “Termes” et “Application des concepts”. “Termes” construit votre vocabulaire, en vous présentant de nouveaux concepts qui s’ajoutent à ceux que vous avez déjà compris. “Application des concepts” illustre l’utilisation réelle des concepts qui ont été présentés jusque là, en fournissant un exposé répété et plus complexe. Certains concepts sont réétudiés plusieurs fois, à des niveaux croissants de complexité. Cette conception itérative améliore la compréhension et la mémorisation.

- [Chapitre 5, “Les bibliothèques des classes Java”](#)

Ce chapitre présente un aperçu des bibliothèques des classes Java 2 et des versions des plates-formes Java 2.

- [Chapitre 6, “Programmation orientée objet dans Java”](#)

Ce chapitre présente les fonctionnalités orientées objet de Java. Vous créez des classes Java, instanciez des objets et accédez aux variables membres au cours d’un bref tutoriel. Vous apprendrez à utiliser l’héritage pour créer de nouvelles classes, les interfaces pour ajouter de nouvelles fonctionnalités à vos classes, le polymorphisme pour que les classes reliées répondent de façons différentes au même message, et les paquets pour regrouper les classes reliées.

- [Chapitre 7, “Techniques de thread”](#)

Un thread est un simple flux séquentiel de contrôle dans un programme. Un des aspects les plus puissants du langage Java est qu’il est facile de programmer plusieurs threads qui s’exécuteront dans le même programme. Ce chapitre explique comment créer des programmes multithreads et fournit des liens vers d’autres ressources contenant des informations plus détaillées.

- [Chapitre 8, “Sérialisation”](#)

La sérialisation permet d’enregistrer et de restaurer l’état d’un objet Java. Ce chapitre décrit comment sérialiser des objets en utilisant Java. Il décrit l’interface `Serializable`, et montre comment écrire un objet sur le disque et comment en retour le lire en mémoire.

- [Chapitre 9, “Introduction à la machine virtuelle Java”](#)

La JVM est le logiciel natif qui permet à un programme Java de s’exécuter sur une machine particulière. Ce chapitre explique la structure et l’objectif général de la JVM. Il décrit les rôles principaux de la JVM, et en particulier la sécurité. Il approfondit trois fonctionnalités de sécurité spécifiques : le vérificateur Java, le gestionnaire de sécurité et le chargeur de classe.

- [Chapitre 10, “Utilisation de l’interface native Java \(JNI\)”](#)

Ce chapitre explique comment appeler des méthodes natives dans les applications Java en utilisant JNI, l’interface de méthodes natives Java. Il commence par expliquer comment fonctionne l’interface JNI, puis décrit le mot clé `native` et montre comment toute méthode Java peut devenir une méthode `native`. Enfin, il étudie l’outil `javah` du JDK, qui permet de générer des fichiers d’en-tête C pour des classes Java.

- [Chapitre 11, “Référence rapide du langage Java”](#)

Ce chapitre contient une liste partielle des bibliothèques de classes et leurs fonctions principales, la liste des versions des plates-formes Java2, la liste complète des mots clés Java du JDK 1.3, les tableaux de conversions entre les types de données primitifs et les types de référence, les séquences d’échappement Java et les tableaux des opérateurs avec leurs actions.

Conventions de la documentation

La documentation Borland pour JBuilder utilise les caractères et symboles décrits dans le tableau ci-dessous pour signaler du texte particulier.

Il y a des considérations spéciales concernant la plate-forme Macintosh. Voir [“Remarque pour les utilisateurs du Macintosh”, page 1-5](#), pour plus d’informations.

Tableau 1.1 Conventions des caractères et des symboles

Caractère	Signification
Police à pas fixe	La police à pas fixe représente ce qui suit : <ul style="list-style-type: none"> le texte tel qu’il apparaît à l’écran tout ce que vous devez taper, comme dans “Entrez Hello World dans le champ Titre de l’expert application.” les noms de fichiers les noms de chemins les noms de répertoires et de dossiers les commandes, comme <code>SET PATH</code>, <code>CLASSPATH</code> le code Java les types de données Java, comme <code>boolean</code>, <code>int</code> et <code>long</code>. les identificateurs Java, comme les noms des variables, classes, interfaces, composants, propriétés, méthodes et événements les noms de paquets les noms de d’arguments les noms de champs les mots clés de Java, comme <code>void</code> et <code>static</code>
Gras	Le gras est utilisé pour les outils java, <code>bmj</code> (Borland Make for Java), <code>bcj</code> (Borland Compiler for Java) et les options de compilation. Par exemple: <code>javac</code> , <code>bmj</code> , <code>-classpath</code> .
<i>Italique</i>	L’italique est utilisé pour les termes nouveaux lors de leur définition et les titres de manuels.
<i>Touches</i>	Cette police indique une touche de votre clavier. Par exemple, “Appuyez sur <i>Echap</i> pour quitter un menu.”
[]	Les crochets dans des listings de syntaxe ou de texte renferment des éléments facultatifs. Ne tapez pas les crochets.
< >	Les crochets angulaires dans le texte ou dans la syntaxe indiquent une chaîne variable ; entrez la chaîne appropriée à votre code. Ne tapez pas ces crochets. Les crochets angulaires sont également utilisés pour les balises HTML. Il sont en outre utilisés pour les répertoires de JBuilder et du JDK. Par exemple, <code><jbuilder></code> est une marque de réservation pour la version courante du répertoire de JBuilder, comme <code>jbuilder5</code> , et <code><.jbuilder></code> représente le répertoire dans lequel sont stockés les paramètres de JBuilder, comme <code>.jbuilder5</code> . De même, <code><jdk></code> est une marque de réservation pour le répertoire en cours du JDK.
...	Dans les exemples de code, les points de suspension indiquent du code manquant. Sur un bouton, les points de suspension indiquent que ce bouton ouvre un dialogue de sélection.

JBuilder est disponible sur plusieurs plates-formes. Le tableau ci-dessous décrit les conventions utilisées dans la documentation pour les plates-formes et les répertoires.

Tableau 1.2 Conventions pour les plates-formes et les répertoires

Élément	Signification
Chemins	Tous les chemins dans la documentation sont signalés par une barre oblique (/). Pour la plate-forme Windows, utilisez une barre oblique inverse (\).
Répertoire de base	L'emplacement du répertoire de base varie d'une plate-forme à l'autre et est indiqué par une marque de réservation, <home>. <ul style="list-style-type: none">• Pour UNIX et Linux, le répertoire de base peut varier. Par exemple, il peut être /user/<nomutilisateur> ou /home/<nomutilisateur>• Pour Windows 95/98, le répertoire de base est C:\Windows• Pour Windows NT, le répertoire de base est C:\Winnt\Profiles\<nomutilisateur>• Pour Windows 2000, le répertoire de base est C:\Documents and Settings\<nomutilisateur>
Répertoire <jbuilder>	Le répertoire <jbuilder> contient l'installation de JBuilder, y compris les programmes, la documentation, les bibliothèques, le JDK, les exemples et autres fichiers. Ce répertoire est nommé d'après la version courante de JBuilder, par exemple jbuilder5.
Répertoire <. <jbuilder>< td=""><td>Le répertoire <.<jbuilder>, .jbuilder5.<="" base.="" ce="" courante="" d'après="" dans="" de="" est="" exemple="" jbuilder,="" la="" le="" lequel="" les="" nommé="" par="" paramètres="" répertoire="" se="" sont="" stockés="" td="" trouve="" version=""></jbuilder>,></td></jbuilder><>	Le répertoire <. <jbuilder>, .jbuilder5.<="" base.="" ce="" courante="" d'après="" dans="" de="" est="" exemple="" jbuilder,="" la="" le="" lequel="" les="" nommé="" par="" paramètres="" répertoire="" se="" sont="" stockés="" td="" trouve="" version=""></jbuilder>,>
Répertoire jbproject	Le répertoire jbproject, qui contient les fichiers projet, classe, source, de sauvegarde et d'autres fichiers, se trouve dans le répertoire de base. JBuilder enregistre les fichiers dans ce chemin par défaut.
Répertoire <jdk>	Le répertoire <jdk> représente le kit de développement Java en cours. Par exemple, jbuilder5/jdk1.3/ sera représenté dans la documentation par <jbuilder>/<jdk>/.
Captures d'écran	Les captures d'écran reflètent l'apparence Metal sur de nombreuses plates-formes.

Remarque pour les utilisateurs du Macintosh

JBuilder a été conçu pour supporter le Macintosh OS X si étroitement qu'il prend l'apparence d'une application native. La plate-forme Macintosh a des conventions d'apparence et de style qui diffèrent de celles de JBuilder ; quand cela se produit, JBuilder prend l'apparence du Mac. Cela veut dire qu'il y a quelques différences entre la façon dont JBuilder se présente sur le Mac et ce qui est décrit dans la documentation. Par exemple, la documentation utilise le mot "répertoire" alors que sur le Mac,

on dit “dossier”. Pour plus d’informations sur les chemins, la terminologie et les conventions de l’interface utilisateur de Macintosh OS X, consultez la documentation livrée avec votre OS X.

Comment contacter le support développeur de Borland

Borland offre aux développeurs diverses options de support. Elles comprennent des services gratuits sur Internet, où vous pouvez consulter notre importante base d’informations et entrer en contact avec d’autres utilisateurs de produits Borland. En outre, vous pouvez choisir parmi plusieurs catégories de support, allant de l’installation des produits Borland au support tarifé de niveau consultant, en passant par une assistance complète.

Pour obtenir des informations sur les services Borland de support aux développeurs, veuillez consulter notre site Web, à l’adresse <http://www.borland.fr/Support/>.

Quand vous contacterez le support, préparez des informations complètes sur votre environnement, la version du produit que vous utilisez et une description détaillée du problème.

Pour avoir de l’aide sur les outils tiers, ou leur documentation, contactez votre fournisseur.

Ressources en ligne

Vous pouvez obtenir des informations depuis les sources ci-après :

World Wide Web <http://www.borland.fr/>

FTP <ftp.borland.com>
Documents techniques accessibles par anonymous ftp.

Listserv Pour vous abonner aux bulletins électroniques, utilisez le formulaire en ligne :
<http://www.borland.com/contact/listserv.html>
ou, pour l’international,
<http://www.borland.com/contact/intlist.html>

World Wide Web

Consultez régulièrement www.borland.fr/jbuilder. L’équipe produit de JBuilder y place notes techniques, analyses des produits concurrents, réponses aux questions fréquemment posées, exemples d’applications, mises à jour du logiciel et informations sur les produits existants ou nouveaux.

Vous pouvez vous connecter en particulier aux URL suivantes :

- <http://www.borland.fr/Produits/jbuilder/> (mises à jour du logiciel et autres fichiers)
- <http://community.borland.com/> (contient notre magazine d'informations web pour les développeurs)

Groupes de discussion Borland

Vous pouvez vous inscrire à JBuilder et participer à de nombreux groupes de discussion dédiés à JBuilder.

Vous trouverez des groupes de discussion, animés par les utilisateurs, pour JBuilder et d'autres produits Borland, à l'adresse <http://www.borland.fr/Newsgroups/>

Groupes de discussion Usenet

Les groupes Usenet suivants sont dédiés à Java et concernent la programmation :

- news:comp.lang.java.advocacy
- news:comp.lang.java.announce
- news:comp.lang.java.beans
- news:comp.lang.java.databases
- news:comp.lang.java.gui
- news:comp.lang.java.help
- news:comp.lang.java.machine
- news:comp.lang.java.programmer
- news:comp.lang.java.security
- news:comp.lang.java.softwaretools

Remarque Ces groupes de discussion sont maintenus par les utilisateurs et ne sont pas des sites Borland officiels.

Rapports de bogues

Si vous pensez avoir trouvé un bogue dans le logiciel, merci de le signaler dans la page du support développeur de JBuilder, à l'adresse <http://www.borland.fr/Support/jbuilder/>. Sur ce site, vous pouvez également demander une fonctionnalité ou consulter la liste des bogues déjà signalés.

Quand vous signalez un bogue, indiquez toutes les étapes nécessaires à la reproduction de ce bogue, ainsi que tout paramètre spécial de votre environnement et les autres programmes utilisés avec JBuilder. Précisez bien le comportement attendu et ce qui s'est réellement passé.

Si vous avez des commentaires (compliments, suggestions ou questions) concernant la documentation de JBuilder, vous pouvez envoyer un e-mail à jgpubs@borland.com. Uniquement pour la documentation. Les questions de support doivent être adressées au support développeur.

JBuilder est fait par des développeurs pour des développeurs. Nous apprécions vraiment vos remarques, car elles nous aident à améliorer notre produit.

Éléments du langage Java

Cette section présente les concepts fondamentaux relatifs aux éléments du langage de programmation Java qui seront utilisés tout au long de ce chapitre. Elle suppose que vous avez assimilé les concepts généraux de programmation, mais que vous avez peu ou pas du tout d'expérience Java.

Termes

Les termes et concepts suivants sont traités dans ce chapitre :

- Identificateur
- Type de données
- Chaînes
- Tableaux
- Variable
- Littéral

Identificateur

L'identificateur est le nom que vous choisissez pour appeler un élément (une variable ou une méthode, par exemple). Java accepte tout identificateur correct, mais, pour un bon usage, il vaut mieux utiliser un terme simple du langage modifié pour répondre aux exigences suivantes :

- Il doit commencer par une lettre. A vrai dire, il peut commencer par un symbole monétaire Unicode ou par un caractère de soulignement (`_`), mais certains de ces symboles peuvent être utilisés dans les fichiers importés ou par les traitements internes. Il vaut donc mieux les éviter.

- Ensuite, il peut contenir tous les caractères alphanumériques (lettres ou chiffres), des caractères de soulignement ou des symboles monétaire Unicode (£ ou \$, par exemple), mais pas d'autres caractères spéciaux.
- Il doit être constitué d'un seul mot (sans espace ni tiret).

L'utilisation des majuscules dans un identificateur dépend de son type. Java distingue les majuscules des minuscules, soyez donc attentif à l'utilisation des majuscules. L'utilisation correcte des majuscules est mentionnée dans le contexte.

Type de données

Les types de données permettent de répertorier le type d'information que peuvent contenir certains éléments de programmation Java. Les types de données se divisent en deux catégories :

- Primitifs ou de base
- Composites ou de référence

Naturellement, différents types de données peuvent contenir différentes quantités d'informations. Vous pouvez convertir le type de données d'une variable en un autre type, dans les limites suivantes : vous ne pouvez pas convertir un type en `boolean` et réciproquement, et vous ne pouvez pas transtyper un objet en un objet d'une classe non liée.

Java vous empêche de faire courir un risque à vos données. Cela veut dire qu'il vous laissera convertir une variable ou un objet en un type plus grand, mais essaiera de vous empêcher de les convertir en un type plus petit. Pour changer un type de données ayant une grande capacité en un type de capacité moindre, vous devez utiliser une instruction particulière, qu'on appelle le *transtypage*.

Types de données primitifs

Les types de données primitifs, ou de base, sont les booléens (qui spécifient un état actif ou inactif), les caractères (simples ou Unicode), les entiers (nombres entiers) ou les nombres en virgule flottante (nombres décimaux). Dans le code, les types primitifs sont tout en minuscules.

Le type de données booléen est appelé `boolean`, et prend une des deux valeurs : `true` ou `false`. Java ne stocke pas ces valeurs sous forme numérique, mais utilise pour cela le type de données `boolean`.

Le type de données caractère est appelé `char` et prend comme valeurs des caractères Unicode simples d'une longueur de 16 bits. Dans Java, les caractères Unicode (lettres, caractères spéciaux et marques de ponctuation) sont placés entre apostrophes : `'b'`. La valeur par défaut d'un caractère Unicode dans Java est `\u0000`, et les valeurs vont de `\u0000` à `\uFFFF`.

En bref, le système de numérotation Unicode accepte des nombres de 0 à 65535, mais les nombres doivent être spécifiés en notation hexadécimale, précédée de la séquence d'échappement `\u`.

Les caractères spéciaux ne peuvent pas tous être représentés de cette façon. Java fournit son propre jeu de séquences d'échappement, dont vous trouverez de nombreux exemples dans le tableau [“Séquences d'échappement”](#), page 11-18.

En Java, la taille des types de données primitifs est absolue, elle ne dépend pas de la plate-forme. Cela renforce la portabilité.

Les types de données numériques acceptent les nombres de différentes sortes et de différentes tailles. Leurs noms et leurs capacités sont énumérés ci-dessous :

Type	Attributs	Etendue
double	Type par défaut de Java. Type en virgule flottante qui prend un nombre sur 8 octets avec deux décimales.	+/- 9.00x10 ¹⁸
int	Option la plus commune. Type entier qui prend un nombre entier sur 4 octet.	+/- 2x10 ⁹
long	Type entier qui prend un nombre entier sur 8 octets.	+/- 9x10 ¹⁸
float	Type en virgule flottante qui prend un nombre sur 4 octets avec une décimale.	+/- 2.0x10 ⁹
short	Type entier qui prend un nombre entier sur 2 octet.	+/- 32768
byte	Type entier qui prend un nombre entier sur 1 octet.	+/- 128

Types de données composites

Chacun des types de données précédents accepte un nombre, un caractère ou un état. Les types de données composites, ou de référence, sont constitués de plusieurs éléments. Les types de données composites sont de deux sortes : classes et tableaux. Les noms de classes et de tableaux commencent par une lettre majuscule et la première lettre de chaque mot qui les constitue est en majuscule (capitalisation en dents de scie), par exemple, `NomDeClasse`.

Une classe est un morceau de code complet et cohérent qui définit un ensemble d'objets unifiés logiquement ainsi que leur comportement. Pour plus d'information sur les classes, voir [Chapitre 6, “Programmation orientée objet dans Java”](#).

Toute classe peut être utilisée comme type de données une fois qu'elle a été créée et importée dans le programme. Comme la classe `String` est la plus souvent utilisée comme type de données, c'est à elle que nous nous consacrerons dans ce chapitre.

Chaînes

La type de données `String` est en réalité la classe `String`. La classe `String` stocke toute séquence de caractères alphanumériques, espaces et ponctuation normale (ce qu'on appelle des *chaînes*), entourée de guillemets. Les chaînes peuvent contenir n'importe quelle séquence d'échappement Unicode et nécessitent `\` pour placer des guillemets à l'intérieur de la chaîne, mais, en général, la classe `String` elle-même indique au programme comment interpréter correctement les caractères.

Tableaux

Un tableau est une structure de données contenant un groupe de valeurs du même type. Par exemple, un tableau accepte un groupe de valeurs `String`, un groupe de valeurs `int` ou un groupe de valeurs `boolean`. Tant que les valeurs sont du même type, elles peuvent être placées dans le même tableau.

Les tableaux sont caractérisés par une paire de crochets droits. Quand vous déclarez un tableau dans Java, vous pouvez placer les crochets soit après l'identificateur, soit après le type de données :

```
int idEtudiant[];  
char[] grades;
```

Remarquez que la taille du tableau n'est pas spécifiée. La déclaration d'un tableau ne lui alloue pas de mémoire. Dans la plupart des autres langages, la taille du tableau doit être indiquée dans la déclaration, mais dans Java, vous ne spécifiez pas sa taille jusqu'au moment où vous l'utilisez. La mémoire appropriée est alors allouée.

Variable

Une variable est une valeur qu'un programmeur nomme et définit. Les variables requièrent un identificateur et une valeur.

Littéral

Un littéral est la représentation réelle d'un nombre, d'un caractère, d'un état ou d'une chaîne. Un littéral représente la valeur d'un identificateur.

Les littéraux alphanumériques comprennent les chaînes entre guillemets, des caractères `char` uniques entre apostrophes et les valeurs `boolean` `true`/`false`.

Les littéraux entiers peuvent être stockés en décimal, octal ou hexadécimal, mais soyez attentif à votre syntaxe : tout entier commençant par 0 (une date, par exemple) sera interprété comme un octal. Les littéraux

en virgule flottante peuvent s'exprimer seulement en décimal. Ils seront traités comme `double` sauf si vous spécifiez le type.

Pour une explication plus détaillée des littéraux et de leurs capacités, voir *The Java Handbook* de Patrick Naughton.

Application des concepts

Les sections suivantes montrent comment appliquer les termes et les concepts présentés auparavant dans ce chapitre.

Déclaration des variables

L'acte de déclarer une variable réserve de la mémoire pour cette variable. La déclaration d'un variable requiert seulement deux choses : un type de données et un identificateur, dans cet ordre. Le type de données indique au programme combien de mémoire allouer. L'identificateur donne un libellé à la mémoire allouée.

Déclarez une seule fois la variable. Une fois que vous avez déclaré la variable de façon appropriée, il suffit de faire référence à son identificateur pour accéder à ce bloc de mémoire.

Les déclarations de variables ressemblent à ceci :

<code>boolean estActif;</code>	Le type de données <code>boolean</code> peut être défini par <code>true</code> ou par <code>false</code> . L'identificateur <code>estActif</code> est le nom que le programmeur a donné à la mémoire allouée à cette variable. Le nom <code>estActif</code> est significatif pour celui qui le lit, et représente quelque chose qui devrait logiquement accepter les valeurs <code>true/false</code> .
<code>int étudiantsInscrits;</code>	Le type de données <code>int</code> indique que vous allez manipuler un nombre entier de moins de dix chiffres. L'identificateur <code>étudiantsInscrits</code> suggère ce que signifie ce nombre. Comme les étudiants sont des gens entiers, le type de données approprié implique des nombres entiers.
<code>double ventesCarteCrédit;</code>	Le type de données <code>double</code> est approprié car les valeurs monétaires sont généralement représentées avec deux décimales. Vous savez ce dont il s'agit, car le programmeur a nommé clairement la variable <code>ventesCarteCrédit</code> .

Méthodes

Les *méthodes* Java sont équivalentes aux fonctions ou aux sous-routines des autres langages. La méthode définit une action à effectuer sur un objet.

Les méthodes sont constituées d'un nom et d'une paire de parenthèses :

```
obtenirDonnées()
```

Ici, `obtenirDonnées` est le nom et les parenthèses indiquent au programme qu'il s'agit d'une méthode.

Si la méthode a besoin d'informations particulières pour faire son travail, cela se trouve entre les parenthèses. Ce qui est à l'intérieur des parenthèses s'appelle l'*argument* (abrégé parfois en *arg*). Dans la déclaration d'une méthode, l'argument inclut un type de données et un identificateur :

```
afficherChaîne(String remarque)
```

Ici, `afficherChaîne` est le nom de la méthode et `String remarque` est le type de données et le nom de variable de la chaîne que la méthode doit afficher.

Vous devez dire au programme quel type de données la méthode doit renvoyer ou si elle ne doit rien renvoyer du tout. Cela s'appelle le *type de retour*. Votre méthode peut renvoyer des données de n'importe quel type primitif. Si une méthode n'a pas besoin de renvoyer quelque chose (comme dans la plupart des méthodes qui exécutent une action), le type de retour doit être `void`.

Le type de retour, le nom et les parenthèses avec tous les arguments nécessaires forment une déclaration de méthode très basique :

```
String afficherChaîne(String remarque);
```

Votre méthode est probablement plus compliquée que cela. Une fois que vous lui avez attribué un type et un nom et que vous lui avez indiqué les arguments dont elle aura besoin (le cas échéant), vous devez la définir complètement. Vous faites cela en-dessous du nom de la méthode, en incorporant le corps de la définition dans une paire d'accolades. Cela donne une déclaration de méthode plus complexe :

```
String afficherChaîne(String remarque) {                //Déclare la méthode
    String remarque = "Que vous avez de grandes dents !" //Définit ce qui est
                                                         //dans la méthode.
} //Ferme le corps de la méthode.
```

Une fois que la méthode est définie, il vous reste à y faire référence par son nom et à lui passer les arguments dont elle a éventuellement besoin pour faire correctement son travail : `afficherChaîne(remarque);`

Structure du langage Java

Cette section présente les concepts fondamentaux relatifs à la structure du langage de programmation Java qui sera utilisée tout au long de ce chapitre. Elle suppose que vous avez assimilé les concepts généraux de programmation, mais que vous avez peu ou pas du tout d'expérience Java.

Termes

Les termes et concepts suivants sont traités dans ce chapitre :

- Mots clés
- Opérateurs
- Commentaires
- Instructions
- Blocs de code
- Portée

Mots clés

Les mots clés sont des termes Java réservés permettant de modifier d'autres éléments de syntaxe. Les mots clés peuvent définir l'accessibilité à un objet, le déroulement d'une méthode ou le type de données d'une variable. Les mots clés ne peuvent jamais servir d'identificateurs.

De nombreux mots clés Java sont empruntés à C/C++. Les mots clés sont toujours écrits en minuscules, comme dans C/C++. De façon générale, les mots clés Java peuvent être classés par catégories selon leurs fonctions (exemples entre parenthèses) :

- Types de données, types de retour et termes (`int`, `void`, `return`)

- Paquet, classe, membre et interface (package, class, static)
- Modificateurs d'accès (public, private, protected)
- Boucles et contrôles de boucles (if, switch, break)
- Gestion des exceptions (throw, try, finally)
- Mots réservés — non encore utilisés, mais non disponibles (goto, assert, const)

Certains mots clés sont décrits dans leur contexte au cours de ces chapitres. Pour une liste complète des mots clés et de leur signification, voir les tableaux “Mots clés” commençant [page 11-3](#).

Opérateurs

Les opérateurs vous permettent d'accéder ou de faire référence à des éléments du langage Java, des variables aux classes, et de les manipuler ou de les lier. Les opérateurs ont des propriétés de *priorité* et d'*associativité*. Quand plusieurs opérateurs agissent sur le même élément (ou *opérande*), la priorité des opérateurs détermine lequel agit en premier. Quand plusieurs opérateurs ont la même priorité, les règles d'associativité s'appliquent. Ces règles sont généralement mathématiques ; par exemple, les opérateurs sont habituellement utilisés de gauche à droite, et les expressions d'opérateurs qui sont à l'intérieur de parenthèses sont évaluées avant celles qui sont à l'extérieur des parenthèses.

Les opérateurs sont généralement classés en six catégories : affectation, arithmétique, logique, comparaison, niveau bits et ternaire.

L'*affectation* signifie le stockage de la valeur qui est à la *droite* du signe = dans la variable qui est à sa *gauche*. Vous pouvez soit affecter une valeur à une variable *quand* vous la déclarez ou *après* l'avoir déclarée. Il n'y a pas de règle ; vous décidez de ce qui est valable dans votre programme et vous le faites :

```
double soldeBanque;           //Déclaration
soldeBanque = 100.35;         //Affectation

double soldeBanque = 100.35;   //Déclaration avec affectation
```

Dans les deux cas, la valeur de 100.35 est stockée dans la mémoire réservée par la déclaration de la variable `soldeBanque`.

Les opérateurs d'affectation vous permettent de donner des valeurs aux variables. Ils vous permettent également d'effectuer une opération sur une expression et puis d'affecter la nouvelle valeur à l'opérande de droite, en utilisant une seule expression combinée.

Les opérateurs arithmétiques effectuent des calculs mathématiques à la fois sur des valeurs entières et sur des valeurs en virgule flottante. Les signes mathématiques usuels s'appliquent : + ajoute, - soustrait, * multiplie et / divise deux nombres.

Les opérateurs logiques, ou booléens, permettent au programmeur de regrouper des expressions de type `boolean`, en indiquant au programme exactement comment déterminer une condition spécifique.

Les opérateurs de comparaison comparent des expressions uniques à d'autres parties du code. Les comparaisons plus complexes (les comparaisons de chaînes, par exemple) sont faites par programme.

Les opérateurs au niveau bits agissent sur les nombres binaires 0 et 1. Les opérateurs au niveau bits de Java peuvent préserver le signe du nombre initial ; tous les langages ne le font pas.

L'opérateur ternaire, `?:`, fournit un raccourci pour écrire une instruction `if-then-else` très simple. La première expression est évaluée ; si elle vaut `true`, la deuxième expression est évaluée ; si la deuxième expression vaut `false`, la troisième expression est utilisée.

Voici une liste partielle des autres opérateurs et de leurs attributs :

Opérateur	Opérande	Comportement
<code>.</code>	membre objet	Accède à un membre d'un objet.
<code>(<type>)</code>	type de données	Convertit un type de données. ¹
<code>+</code>	chaîne	Joint des chaînes (concaténation).
	nombre	Additionne.
<code>-</code>	nombre	C'est le moins unaire ² (inverse le signe du nombre).
	nombre	Soustrait.
<code>!</code>	booléen	C'est l'opérateur <code>boolean NOT</code> .
<code>&</code>	entier, booléen	C'est à la fois l'opérateur au niveau bits (entier) et <code>boolean AND</code> . Quand il est doublé (<code>&&</code>), c'est le <code>AND</code> conditionnel <code>boolean</code> .
<code>=</code>	la plupart des éléments avec des variables	Affecte un élément à un autre élément (par exemple, une valeur à une variable, ou une classe à une instance). Il peut être combiné à d'autres opérateurs pour effectuer une opération et affecter la valeur résultante. Par exemple, <code>+=</code> ajoute la valeur de gauche à celle de droite, puis affecte la nouvelle valeur au côté droit de l'expression.

1. Il est important de faire la distinction entre opération et ponctuation. Les parenthèses sont utilisées comme ponctuation autour des arguments. Elles sont utilisées autour d'un type de données dans une opération qui change le type de données d'une variable en celui qui est à l'intérieur des parenthèses.
2. Un opérateur unaire prend un seul opérande, un opérateur binaire prend deux opérandes et un opérateur ternaire prend trois opérandes.

Commentaires

Commenter le code est une excellente habitude de programmation. De bons commentaires peuvent vous aider à analyser plus rapidement votre code, à garder la trace de ce que vous faites au fur et à mesure que vous

construisez un programme complexe et à vous souvenir de ce qu'il faudra ajouter ou modifier. Vous pouvez utiliser les commentaires pour cacher des parties du code que vous gardez uniquement pour certaines situations ou que vous mettez de côté pendant que vous travaillez sur quelque chose qui pourrait provoquer un conflit. Les commentaires peuvent vous aider à vous rappeler ce que vous étiez en train de faire lorsque vous reprenez un projet après avoir travaillé sur un autre ou quand vous revenez de congé. Dans un environnement de développement en équipe ou chaque fois que le code passe d'un programmeur à un autre, les commentaires aident les autres à comprendre le but et les associations de tout ce que vous avez commenté, sans qu'il leur soit nécessaire d'analyser chaque ligne pour être sûrs d'avoir compris.

Java utilise trois sortes de commentaires : les commentaires sur une ligne, les commentaires sur plusieurs lignes et les commentaires Javadoc.

Commentaire	Balise	But
Une ligne	// ...	Convient à de courtes remarques sur la fonction ou la structure d'une instruction ou d'une expression. Ils nécessitent uniquement une balise d'ouverture : dès que vous commencez une nouvelle ligne, vous revenez au code.
Plusieurs lignes	/* ... */	Destiné à tout commentaire dépassant une ligne, comme lorsque vous voulez détailler ce qui se passe dans le code ou quand vous avez besoin d'incorporer au code des informations légales. Il requiert une balise d'ouverture et une balise de fermeture.
Javadoc	/** ... */	Il s'agit d'un commentaire multiligne que l'utilitaire Javadoc du JDK peut lire et transformer en documentation HTML. Javadoc a recours à des balises que vous pouvez utiliser pour étendre ses fonctionnalités. Il est utilisé pour fournir de l'aide aux API, générer des listes à faire et incorporer des drapeaux dans le code. Il requiert une balise d'ouverture et une balise de fermeture. Pour en savoir plus sur l'outil Javadoc, allez à la page Javadoc de Sun, à l'adresse http://java.sun.com/products/jdk/1.2/docs/tooldocs/javadoc/ .

Voici quelques exemples :

```
/* Vous pouvez utiliser autant de lignes
   d'explication ou autant de pages de
   passe-partout que vous voulez entre
   ces deux balises.
*/

/* Remarquez que, si vous voulez vraiment aller plus loin,
   vous pouvez insérer des commentaires sur une ligne
   //à l'intérieur des commentaires multilignes
   et cela ne posera aucun problème
   au compilateur.
*/
```

```

/* Mais, n'essayez pas d'imbriquer
/* des commentaires multilignes
*/
/** quels qu'ils soient
*/
car cela générerait une
erreur de compilation.
*/

/**Les informations utiles sur ce que fait le code
se placent entre les balises Javadoc. Des balises spéciales
comme @todo peuvent être utilisées ici pour tirer
profit des fonctionnalités d'aide de Javadoc.
*/

```

Instructions

Une instruction est une seule commande. Une commande peut occuper plusieurs lignes de code, mais le compilateur lit l'ensemble comme une seule commande. Les instructions individuelles (habituellement sur une seule ligne) se terminent par un point-virgule (;), et les groupes d'instructions (sur plusieurs lignes) se terminent par une accolade fermante (}). Les instructions sur plusieurs lignes sont généralement appelées *blocs de code*.

Par défaut, Java exécute les instructions dans l'ordre où elles ont été écrites, mais Java autorise les références anticipées aux termes non encore définis.

Blocs de code

Un bloc de code est tout ce qui est entre accolades plus l'expression qui introduit la partie entre accolades :

```

class obtenirArrondi {
    ...
}

```

Comprendre la portée

Les règles de portée déterminent où est reconnue une variable dans un programme. Les variables appartiennent à deux catégories de portée principales :

- Variables globales : Variables reconnues dans toute une classe.
- Variables locales : Variables reconnues uniquement dans le bloc de code où elles sont déclarées.

Les règles de portée sont étroitement liées aux blocs de code. La règle de portée générale est : une variable déclarée dans un bloc de code n'est visible que dans ce bloc et dans les blocs qui y sont imbriqués. Le code suivant illustre cette règle :

```
class Portée {
    int x = 0;
    void méthode1() {
        int y;
        y = x; // Cela fonctionne. méthode1 peut accéder à y.
    }
    void méthode2() {
        int z = 1;
        z = y; // Cela ne fonctionne pas :
                // y est définie hors de la portée de méthode2.
    }
}
```

Ce code déclare une classe appelée *Portée*, qui contient deux méthodes : *méthode1()* et *méthode2()*. La classe elle-même est considérée comme étant le bloc de code principal et les deux méthodes sont ses blocs imbriqués.

La variable *x* est déclarée dans le bloc principal et elle est donc *visible* (reconnue par le compilateur) dans la méthode *méthode1()* et dans la méthode *méthode2()*. D'autre part, les variables *y* et *z*, sont déclarées dans deux blocs imbriqués mais indépendants ; ainsi, essayer d'utiliser *y* dans *méthode2()* provoque une erreur, puisque *y* n'est visible que dans son bloc.

Remarque Un programme qui repose sur des variables globales peut être sujet à des erreurs pour deux raisons :

- 1 Les variables globales sont difficile à suivre.
- 2 La modification d'une variable globale à un endroit du programme peut avoir un effet secondaire imprévu à un autre endroit.

Les variables locales sont plus sûres, puisqu'elles ont une durée de vie limitée. Par exemple, une variable déclarée dans une méthode n'est accessible qu'à partir de cette méthode et ne risque pas d'être utilisée de façon erronée ailleurs dans le programme.

Terminez chaque instruction simple par un point-virgule. Vérifiez que chaque accolade a sa correspondante. Organisez vos accolades de façon cohérente (comme dans les exemples précédents) afin de pouvoir repérer les paires. De nombreux EDI Java (comme JBuilder) imbriquent automatiquement les accolades selon vos préférences.

Application des concepts

Les sections suivantes montrent comment appliquer les termes et les concepts présentés auparavant dans ce chapitre.

Utilisation des opérateurs

Rappel Il y a six sortes d'opérateurs (arithmétique, logique, affectation, comparaison, niveau bits et ternaire), et les opérateurs affectent un, deux ou trois opérandes, selon qu'ils sont unaires, binaires ou ternaires. Ils possèdent des propriétés de priorité et d'associativité, qui déterminent l'ordre dans lequel ils sont traités.

On affecte des numéros aux opérateurs pour établir leur priorité. Plus le numéro est élevé, plus la priorité de l'opérateur est haute (c'est-à-dire, qu'il sera évalué avant les autres). Un opérateur dont la priorité est 1 (la plus basse) sera évalué en dernier, et un opérateur dont la priorité est 15 (la plus haute) sera évalué en premier.

Les opérateurs de même priorité sont normalement évalués de gauche à droite.

La priorité est évaluée avant l'associativité. Par exemple, l'expression $a + b - c * d$ ne sera pas évaluée de gauche à droite ; la multiplication étant prioritaire sur l'addition, $c * d$ sera évaluée d'abord. L'addition et la soustraction ayant la même priorité, l'associativité s'applique : a et b seront additionnés d'abord, puis cette somme sera soustraite du produit $c * d$.

C'est une bonne habitude d'utiliser des parenthèses autour des expressions mathématiques que vous voulez évaluer en premier, indépendamment de leur priorité, par exemple : $(a + b) - (c * d)$. Le programme évaluera cette opération de la même façon, mais pour celui qui lit le programme, ce sera plus clair.

Opérateurs arithmétiques

Java fournit un jeu complet d'opérateurs de calcul. A la différence de certains langages, Java peut exécuter des fonctions mathématiques sur des valeurs entières et sur des valeurs en virgule flottante. Ces opérateurs vous sont sans doute familiers.

Voici les opérateurs arithmétiques :

Opérateur	Définition	Priorité	Associativité
++/--	Incrémentation/décrémentation automatique : Ajoute un à ou soustrait un de son opérande unique. Si la valeur de <i>i</i> est 4, ++ <i>i</i> vaut 5. Une pré-incrémentation (++ <i>i</i>) incrémente de un la valeur et affecte la nouvelle valeur à la variable initiale <i>i</i> . Une post-incrémentation (<i>i</i> ++) incrémente la valeur mais laisse intacte la variable initiale <i>i</i> . Voir plus loin pour plus d'informations.	1	Droite
+/-	Plus/moins unaire : définit ou modifie la valeur positive/négative d'un seul nombre.	2	Droite
*	Multiplication.	4	Gauche
/	Division.	4	Gauche
%	Modulo : Divise le premier opérande par le second et renvoie le reste. Voir ci-dessous pour un bref rappel mathématique.	4	Gauche
+/-	Addition/soustraction	5	Gauche

Utilisez la pré- ou post-incrémentation/décrémentation selon que vous voulez affecter ou non la nouvelle valeur à la variable :

```
int y = 3, x; //1. déclaration des variables
int b = 9;    //2.
int a;        //3.
x = ++y;      //4. pré-incrémentation
a = b--;      //5. post-décrémentation
```

Dans l'instruction 4, pré-incrémentation, la variable *y* est incrémentée de 1, puis sa nouvelle valeur (4) est attribuée à *x*. *x* et *y* valaient toutes les deux 3 initialement; maintenant, elles valent toutes les deux 4.

Dans l'instruction 5, post-décrémentation, la valeur en cours (9) de *b* est affectée à *a* et *ensuite* la valeur de *b* est décrémentée (à 8). Initialement, *b* valait 9 et *a* n'avait pas de valeur ; maintenant, *a* vaut 9 et *b* vaut 8.

L'opérateur modulo exige une explication pour ceux qui ont étudié les mathématiques il y a très longtemps. Quand vous divisez deux nombres, ils ne se divisent pas toujours exactement. Le reliquat de la division (si vous n'ajoutez pas de décimale) est le *reste*. Par exemple, 3 va dans 5 une fois et il reste 2. Le reste (ici, 2) est ce que calcule l'opérateur modulo. Comme les restes d'un cycle de division se répètent de façon prévisible (par exemple, l'heure est calculée modulo 60), l'opérateur modulo est particulièrement utile pour indiquer à un programme de répéter un processus à intervalles réguliers.

Opérateurs logiques

Les opérateurs logiques (ou booléens) permettent au programmeur de regrouper des expressions de type `boolean` pour déterminer certaines conditions. Ces opérateurs exécutent les opérations booléennes standard (AND, OR, NOT et XOR).

Le tableau suivant donne la liste des opérateurs logiques :

Opérateur	Définition	Priorité	Associativité
!	NOT booléen (unaire) Change <code>true</code> en <code>false</code> ou <code>false</code> en <code>true</code> . En raison de sa priorité basse, vous devez inclure cette instruction entre parenthèses.	2	Droite
&	AND évaluation (binaire) Renvoie <code>true</code> seulement si les deux opérandes valent <code>true</code> . Evalue toujours deux opérandes. Rarement utilisé comme opérateur logique.	9	Gauche
^	XOR évaluation (binaire) Renvoie <code>true</code> si un des deux opérandes seulement vaut <code>true</code> . Evalue deux opérandes.	10	Gauche
	OR évaluation (binaire) Renvoie <code>true</code> si un ou les deux opérandes valent <code>true</code> . Evalue deux opérandes.	11	Gauche
&&	AND conditionnel (binaire) Renvoie <code>true</code> seulement si les deux opérandes valent <code>true</code> . Il est dit "conditionnel" car il n'évalue le second opérande que si le premier vaut <code>true</code> .	12	Gauche
	OR conditionnel (binaire) Renvoie <code>true</code> si un ou les deux opérandes valent <code>true</code> ; renvoie <code>false</code> si les deux valent <code>false</code> . Le second opérande n'est pas évalué si le premier vaut <code>true</code> .	13	Gauche

Les opérateurs d'évaluation évaluent toujours deux opérandes. Les opérateurs conditionnels, eux, évaluent toujours le premier opérande et, si cela suffit à déterminer la valeur de la totalité de l'expression, ils n'évaluent pas le deuxième. Par exemple :

```

if ( !estHautePression && (température1 > température2)) {
    ...
} //Instruction 1 : conditionnelle

boolean1 = (x < y) || ( a > b); //Instruction 2 : conditionnelle

booléen2 = (10 > 5) & (5 > 1); //Instruction 3 : évaluation

```

La première instruction évalue d'abord `!estHautePression`. Si `!estHautePression` vaut `false` (c'est-à-dire si la pression *est* haute ; notez la double négation logique de `!` et de `false`), le deuxième opérande,

`température1 > température2`, n'a pas besoin d'être évalué. && il suffit d'une valeur `false` pour déterminer la valeur renvoyée.

Dans la deuxième instruction, la valeur de `booléen1` sera `true` si `x` est inférieur à `y`. Si `x` est supérieur à `y`, la deuxième expression sera évaluée ; si `a` est inférieur à `b`, la valeur de `booléen1` sera encore `true`.

Dans la troisième instruction, cependant, le compilateur calculera la valeur des deux opérandes avant d'affecter `true` ou `false` à `booléen2`, car `&` est un opérateur d'évaluation et non un opérateur conditionnel.

Opérateurs d'affectation

Vous savez que l'opérateur d'affectation de base (`=`) vous permet d'affecter une valeur à une variable. Avec l'ensemble des opérateurs d'affectation de Java, vous pouvez en une seule étape effectuer une opération sur un opérande et affecter la valeur à une variable.

Le tableau suivant donne la liste des opérateurs d'affectation :

Opérateur	Définition	Priorité	Associativité
<code>=</code>	Affecte la valeur de droite à la variable de gauche.	15	Droite
<code>+=</code>	Ajoute la valeur de droite à la valeur de la variable de gauche ; affecte la nouvelle valeur à la variable initiale.	15	Droite
<code>-=</code>	Soustrait la valeur de droite de la valeur de la variable de gauche ; affecte la nouvelle valeur à la variable initiale.	15	Droite
<code>*=</code>	Multiplie la valeur de droite avec la valeur de la variable de gauche ; affecte la nouvelle valeur à la variable initiale.	15	Droite
<code>/=</code>	Divise la valeur de la variable de gauche par la valeur de droite ; affecte la nouvelle valeur à la variable initiale.	15	Droite

Le premier opérateur vous est maintenant familier. Les autres opérateurs d'affectation commencent par une opération, puis stockent le résultat de l'opération dans l'opérande situé sur le côté gauche de l'expression. Voici quelques exemples :

```
int y = 2;
y *= 2;    //identique à (y = y * 2)

boolean b1 = true, b2 = false;
b1 &= b2;  //identique à (b1 = b1 & b2)
```

Opérateurs de comparaison

Les opérateurs de comparaison vous permettent de comparer une valeur à une autre.

Le tableau suivant donne la liste des opérateurs de comparaison :

Opérateur	Définition	Priorité	Associativité
<	Inférieur à	7	Gauche
>	Supérieur à	7	Gauche
<=	Inférieur ou égal à	7	Gauche
>=	Supérieur ou égal à	7	Gauche
==	Egal à	8	Gauche
!=	Différent de	8	Gauche

L'opérateur d'égalité peut être utilisé pour comparer deux variables objet de même type. Dans ce cas, le résultat de la comparaison n'a la valeur `true` que si les deux variables font référence au même objet. Voici une illustration :

```
m1 = new Mammifère();
m2 = new Mammifère();
boolean b1 = m1 == m2; //b1 a la valeur false

m1 = m2;
boolean b2 = m1 == m2; //b2 a la valeur true
```

Comme `m1` et `m2` font référence à des objets différents, le premier test d'égalité donne le résultat `false` (bien qu'ils soient du même type). La deuxième comparaison donne le résultat `true`, puisque les deux variables représentent maintenant le même objet.

Remarque La plupart du temps, cependant, la méthode `equals()` de la classe `Object` est utilisée à la place de l'opérateur de comparaison. La classe de comparaison doit être sous-classée depuis `Object` pour que ses objets puissent être comparés à l'aide de `equals()`.

Opérateurs au niveau bits

Il y a deux types d'opérateurs au niveau bits : opérateurs de décalage et opérateurs booléens. Les opérateurs de décalage permettent de décaler les chiffres binaires d'un entier vers la droite ou vers la gauche. Etudiez l'exemple suivant (le type entier `short` est utilisé à la place du type `int` pour plus de concision) :

```
short i = 13; //i a la valeur 0000000000001101
i = i << 2; //i a la valeur 0000000000110100
```

Dans la deuxième ligne, l'opérateur de décalage des bits décale tous les bits de `i` de deux positions vers la gauche.

Remarque L'opération de décalage fonctionne différemment dans Java et dans C/C++, principalement en ce qui concerne les nombres entiers *signés*. Dans un nombre entier signé, le bit le plus à gauche indique le signe positif ou négatif du nombre entier : le bit a la valeur 1 si l'entier est négatif, 0 s'il est positif. Dans Java, les nombres entiers sont toujours signés, alors que dans

C/C++, ils sont signés par défaut. Dans la plupart des implémentations de C/C++, une opération de décalage des bits ne conserve pas le signe du nombre entier ; le bit du signe serait décalé. Cependant, dans Java, les opérateurs de décalage conservent le bit du signe (sauf si vous utilisez l'opérateur >>> pour effectuer un *décalage non signé*). Cela veut dire que le bit du signe est dupliqué, puis décalé. Par exemple, décaler à droite 10010011 de 1 donne 11001001.

Le tableau suivant donne la liste des opérateurs au niveau bits de Java :

Opérateur	Définition	Priorité	Associativité
~	NOT au niveau bits Inverse chaque bit de l'opérande, 0 devient 1 et réciproquement.	2	Droite
<<	Décalage gauche signé Décale à gauche les bits de l'opérande gauche, du nombre de chiffres spécifié dans l'opérande droit, complète la droite par des 0. Les bits de poids fort sont perdus.	6	Gauche
>>	Décalage droit signé Décale à droite les bits de l'opérande gauche, du nombre de chiffres spécifié dans l'opérande droit. Si l'opérande gauche est négatif, la partie gauche est complétée par des 0 ; s'il est positif, elle est complétée par des 1. Cela préserve le signe initial.	6	Gauche
>>>	Décalage droit par ajout de zéros Décale à droite et remplit toujours par des 0.	6	Gauche
&	AND au niveau bits Peut être utilisé avec = pour affecter la valeur.	9	Gauche
	OR au niveau bits Peut être utilisé avec = pour affecter la valeur.	10	Gauche
^	XOR au niveau bits Peut être utilisé avec = pour affecter la valeur.	11	Gauche
<<=	Décalage gauche avec affectation	15	Gauche
>>=	Décalage droit avec affectation	15	Gauche
>>>=	Décalage droit par ajout de zéros avec affectation	15	Gauche

?: — L'opérateur ternaire

?: est un opérateur ternaire que Java a emprunté au C. Il fournit un raccourci pratique pour créer une instruction de type if-then-else très simple.

En voici la syntaxe :

```
<expression 1> ? <expression 2> : <expression 3>;
```

expression 1 est évaluée en premier. Si elle est vraie, alors expression 2 est évaluée. Si expression 2 est fausse, expression 3 est utilisée. Par exemple :

```
int x = 3, y = 4, max;
max = (x > y) ? x : y;
```

Dans ce code, max reçoit la valeur de x ou de y, selon celui qui est le plus grand.

Utilisation des méthodes

Vous savez que les méthodes sont ce qui permet de faire des choses. Les méthodes ne peuvent pas contenir d'autres méthodes, mais elles peuvent contenir des variables et des références à des classes.

Voici un court exemple à examiner. Cette méthode aide à faire l'inventaire d'un magasin de musique :

```
//Déclare la méthode : type de retour, nom, arguments :
public int obtenirTotalCDs(int nbCDsRock, int nbCDsJazz, int nbCDsPop) {
    //Déclare la variable totalCDs.
    //Les trois autres variables ont été déclarées ailleurs :
    int totalCDs = nbCDsRock+ nbCDsJazz+ nbCDsPop;
    //Lui faire faire quelque chose d'utile.
    //Dans ce cas, imprimer cette ligne à l'écran :
    System.out.println("Nombre total de CD en stock = " + totalCDs);
}
```

Dans Java, vous pouvez définir plusieurs méthodes portant le même nom, pourvu qu'elles aient des arguments différents. Par exemple, il est correct d'avoir public int obtenirTotalCDs(int nbCDsRock, int nbCDsJazz, int nbCDsPop) et public int obtenirTotalCDs(int ventesCDsDétail, int ventesCDsGros) dans la même classe. Java reconnaîtra les deux modèles d'arguments (les *signatures de la méthode*) et appliquera la bonne méthode lorsque vous y ferez appel. L'attribution du même nom à des méthodes différentes s'appelle la *surcharge des méthodes*.

Pour accéder à une méthode à partir d'autres endroits d'un programme, vous devez d'abord créer une instance de la classe dans laquelle réside la méthode, puis utiliser cet objet pour appeler la méthode :

```
//Crée une instance totalCD de la classe Inventaire :
Inventaire totalCD = new Inventaire();

//Accède à la méthode obtenirTotalCDs() à l'intérieur d'Inventaire,
//et stocke la valeur dans total :
int total = totalCD.obtenirTotalCDs(mesNbCDsRock, mesNbCDsJazz, mesNbCDsPop);
```

Utilisation des tableaux

Notez que la taille d'un tableau ne fait pas partie de sa déclaration. La mémoire requise par un tableau n'est allouée que lorsque vous initialisez le tableau.

Pour initialiser le tableau (et donc allouer réellement de la mémoire), vous devez utiliser l'opérateur `new` comme ceci :

```
int idEtudiant[] = new int[20];           //Crée un tableau de 20 éléments int.
char[] grades = new char[20];            //Crée un tableau de 20 éléments char.
float[][] coordonnées = new float[10][5]; //tableau bidimensionnel
                                           //de 10x5 éléments float.
```

Remarque Quand vous créez des tableaux bidimensionnels, le premier nombre définit le nombre de colonnes et le second le nombre de lignes.

Java compte les positions en commençant par 0. Cela veut dire que les éléments d'un tableau de 20 éléments seront numérotés de 0 à 19 : le premier élément sera 0, le deuxième 1, etc. Ne l'oubliez pas lorsque vous manipulez des tableaux.

Quand un tableau est créé, la valeur de tous ses éléments est à `null` ou à 0 ; les valeurs sont affectées plus tard.

Remarque Dans Java, l'utilisation de l'opérateur `new` est semblable à l'utilisation de la commande `malloc` en C et de l'opérateur `new` en C++.

Pour initialiser un tableau, spécifiez les valeurs des éléments du tableau à l'intérieur d'un ensemble d'accolades. Pour les tableaux à plusieurs dimensions, il faut utiliser des accolades imbriquées. Par exemple :

```
char[] grades = {'A', 'B', 'C', 'D', 'F'};
float[][] coordonnées = {{0.0, 0.1}, {0.2, 0.3}};
```

La première instruction crée un tableau de type `char` appelé `grades`. Elle initialise les éléments du tableau avec des valeurs comprises entre 'A' et 'F'. Nous n'avons pas eu besoin d'utiliser l'opérateur `new` pour créer ce tableau ; en initialisant le tableau, la mémoire nécessaire est automatiquement affectée au tableau pour y mettre les valeurs initialisées. Par conséquent, la première instruction crée un tableau de type `char` de 5 éléments.

La deuxième instruction crée un tableau bidimensionnel de type `float` appelé `coordonnées`, avec une taille de 2 sur 2. Au fond, `coordonnées` est un tableau constitué de deux éléments de tableau : la première ligne du tableau est initialisée à 0.0 et 0.1, et la deuxième ligne à 0.2 et 0.3.

Utilisation des constructeurs

Une *classe* est un morceau de code complet, entouré d'accolades, qui définit un ensemble logique et cohérent de variables, d'attributs et d'actions. Un *paquet* est un ensemble de classes réunies de façon logique.

Notez qu'une classe est simplement un ensemble d'instructions. Elle ne fait rien elle-même. On peut la comparer à une recette : vous faites un gâteau avec la bonne recette, mais la recette ne fait pas le gâteau, elle se contente de donner les instructions pour le faire. Le gâteau est un *objet* que vous avez créé à partir des instructions de la recette. Dans Java, nous dirions que nous avons créé une *instance* du gâteau à partir de la recette Gâteau.

L'acte de créer une instance d'une classe est appelé *instanciation* de cet objet. Vous *instanciez* un *objet* d'une *classe*.

Pour instancier un objet, utilisez l'opérateur d'affectation (=), le mot clé `new` et un type spécial de méthode appelé un *constructeur*. L'appel à un constructeur est le nom de la classe à instancier suivi d'une paire de parenthèses. Bien qu'il ressemble à une méthode, il prend le nom d'une classe ; c'est pourquoi il prend une majuscule :

```
<NomClasse> <nomInstance> = new <Constructeur>();
```

Par exemple, pour instancier un nouvel objet de la classe `Geek` et nommer l'instance `ceProgrammeur` :

```
Geek ceProgrammeur = new Geek();
```

Un constructeur crée une nouvelle instance d'une classe : il initialise toutes les variables de cette classe, ce qui les rend immédiatement disponibles. Il peut aussi exécuter n'importe quelle routine de démarrage requise par l'objet.

Par exemple, quand vous avez besoin de conduire votre voiture, la première chose à faire est d'ouvrir la porte, de monter dedans, de mettre le contact et de démarrer le moteur. (Après, vous pouvez faire tout ce que vous faites habituellement pour conduire, comme passer une vitesse et accélérer.) Le constructeur gère les équivalents en termes de programmation des actions et objets impliqués dans l'ouverture et le démarrage d'une voiture.

Une fois que vous avez créé une instance, vous pouvez utiliser son nom pour accéder aux membres de cette classe.

Pour plus d'informations sur les constructeurs, voir ["Etude de cas : Exemple simple d'OOP", page 6-4.](#)

Accès aux membres

L'opérateur d'accès (.) est utilisé pour accéder aux membres se trouvant à l'intérieur d'un objet instancié. La syntaxe de base est :

```
<nomInstance>.<nomMembre>
```

La syntaxe exacte du nom du membre dépend du membre dont il s'agit. Il peut y avoir des variables (<nomMembre>), des méthodes (<nomMembre>()) ou des sous-classes (<NomMembre>).

Vous pouvez utiliser cette opération à l'intérieur d'autres éléments de syntaxe partout où vous avez besoin d'accéder à un membre. Par exemple :

```
setColor(Color.pink);
```

Cette méthode a besoin d'une couleur pour effectuer son travail. Le programmeur a utilisé une opération d'accès comme argument pour accéder à la variable `pink` à l'intérieur de la classe `Color`.

Tableaux

Pour accéder aux éléments d'un tableau, il faut *indexer* la variable tableau. Pour indexer une variable tableau, faites suivre son nom du numéro de l'élément (indice) entre crochets droits. *Les tableaux sont toujours indexés en commençant par 0.* (Programmer comme s'ils étaient numérotés à partir de 1 est une erreur commune.)

Dans le cas d'un tableau à plusieurs dimensions, il faut utiliser un indice par dimension. Le premier indice est la ligne et le deuxième est la colonne.

Par exemple :

```
premierElément = grades[0];    //premierElément = 'A'
cinquièmeElément = grades[4];  //cinquièmeElément = 'F'
ligne2Col1 = coordonnées[1][0]; //ligne2Col1 = 0.2
```

Le morceau de code suivant illustre une utilisation des tableaux. Elle crée un tableau de 5 éléments de type `int` appelé `tableauInt`, puis utilise une boucle `for` pour mettre les nombres entiers 0 à 4 dans les éléments du tableau :

```
int[] tableauInt = new int [5];
int indice;
for (indice = 0; indice < 5; indice++) tableauInt [indice] = indice;
```

Ce code incrémente la variable `indice` de 0 à 4 et, à chaque passage, sa valeur est mise dans l'élément de `tableauInt` indexé par la variable `indice`.

Contrôle du langage Java

Cette section présente les concepts fondamentaux relatifs au contrôle du langage de programmation Java qui sera utilisé tout au long de ce chapitre. Elle suppose que vous avez assimilé les concepts généraux de programmation, mais que vous avez peu ou pas du tout d'expérience Java.

Termes

Les termes et concepts suivants sont traités dans ce chapitre :

- Gestion des chaînes
- Transtypage et conversion
- Types et instructions de retour
- Instructions de contrôle du déroulement

Gestion des chaînes

La classe `String` fournit les méthodes qui vous permettent de récupérer des sous-chaînes ou *d'indexer* des caractères à l'intérieur d'une chaîne. Cependant, la valeur d'une `String` déclarée ne peut pas être modifiée. Si vous avez besoin de changer la valeur `String` associée à cette variable, vous devez faire pointer la variable vers une nouvelle valeur :

```
String text1 = new String("Bonsoir."); // Déclare text1 et lui affecte une
                                     // valeur.
text1 = "Chéri, je suis rentrée !"     // Affecte une nouvelle valeur à text1.
```

L'indexation vous permet de pointer sur un caractère particulier d'une chaîne. Java numérote chaque position d'une chaîne, en commençant

par 0, de sorte que la première position est 0, la deuxième 1, etc. Ce qui donne à la huitième position d'une chaîne l'indice 7.

La classe `StringBuffer` fournit une solution. Elle offre aussi plusieurs autres moyens de manipuler le contenu d'une chaîne. La classe `StringBuffer` stocke votre chaîne dans un tampon (une zone de mémoire spéciale) dont vous contrôlez explicitement la taille ; cela vous permet de changer la chaîne autant de fois que nécessaire avant de déclarer une `String` pour la rendre permanente.

En général, la classe `String` sert au stockage des chaînes, alors que la classe `StringBuffer` sert à leur manipulation.

Transtypage et conversion

Les valeurs des types de données peuvent être convertis dans un autre type. Les valeurs d'une classe peuvent être converties d'une classe à une autre classe de la même hiérarchie. Notez que cette conversion ne change pas le type initial de la valeur, mais uniquement la perception qu'en a le compilateur pour cette seule opération.

Des restrictions logiques évidentes s'appliquent. Une conversion d'agrandissement — d'un type petit vers un plus grand — est facile, mais une conversion de raccourcissement — d'un type grand (par exemple, `double` ou `Mammifère`) vers un plus petit (par exemple, `float` ou `Ours`) — est dangereuse pour vos données, sauf si vous êtes certain que les données tiendront dans les paramètres du nouveau type. Une conversion de raccourcissement requiert une opération spéciale qu'on appelle le *transtypage*.

Le tableau suivant montre des conversions d'agrandissement des valeurs primitives. Elles ne présentent aucun risque pour vos données :

Type initial	Convertit en type
<code>byte</code>	<code>short</code> , <code>char</code> , <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code>
<code>short</code>	<code>int</code> , <code>long</code> , <code>float</code> , <code>double</code>
<code>char</code>	<code>int</code> , <code>long</code> , <code>float</code> , <code>double</code>
<code>int</code>	<code>long</code> , <code>float</code> , <code>double</code>
<code>long</code>	<code>float</code> , <code>double</code>
<code>float</code>	<code>double</code>

Pour transtyper un type de données, placez le type *vers* lequel transtyper entre parenthèses, immédiatement avant la variable à transtyper : `(int)x`. Voici le cas où `x` est la variable à transtyper, `float` le type de données

`initial`, `int` le type de destination et `y` est la variable contenant la nouvelle valeur :

```
float x = 1.00; //déclaration de x en float
int y = (int)x; //transtypage de x vers un int nommé y
```

Cela suppose que la valeur de `x` tiendra dans `int`. Notez que les décimales de `x` sont perdues dans la conversion. Java arrondit les décimales au nombre entier le plus proche.

Types et instructions de retour

Vous savez qu'une méthode requiert un type de retour, comme une déclaration de variable requiert un type de données. Les types de retour sont identiques aux types de données (`int`, `boolean`, `String`, etc.), à l'exception de `void`.

`void` est un type de retour spécial. Il signifie que la méthode ne renvoie rien lorsqu'elle est terminée. Il est le plus communément utilisé dans les méthodes d'action qui servent uniquement à faire quelque chose et non à transmettre des informations.

Tous les autres types de retour nécessitent une instruction de retour, `return`, à la fin de la méthode. Vous pouvez utiliser l'instruction `return` dans une méthode `void` pour quitter la méthode à un certain endroit, sinon, elle n'est pas nécessaire.

Une instruction de retour est constituée du mot `return` et de la chaîne, des données, du nom de variable ou de la concaténation nécessaires :

```
return nombreCD;
```

Il est courant d'utiliser des parenthèses pour les concaténations :

```
return ("Nombre de fichiers : " + nbFichiers);
```

Instructions de contrôle du déroulement

Les instructions de contrôle de déroulement indiquent au programme comment classer et utiliser les informations que vous lui donnez. Avec les instructions de contrôle de déroulement, vous pouvez répéter des instructions, insérer des conditions, créer des boucles récursives et contrôler le comportement des boucles.

Les instructions de contrôle du déroulement peuvent être regroupées en trois sortes : les instructions d'*itération* comme `for`, `while` et `do-while`, qui créent des boucles ; les instructions de *sélection* comme `switch`, `if`, `if-else`, `if-then-else` et `if-else-if`, qui mettent des conditions à l'utilisation des instructions, et les instruction de *branchement* `break`, `continue` et `return`, qui passent le contrôle à une autre partie du programme.

La *gestion des exceptions* est une forme spéciale de contrôle du déroulement. La gestion des exceptions offre des moyens structurés de capturer les erreurs d'exécution de votre programme et de fournir des informations significatives à leur sujet. Vous pouvez aussi définir le gestionnaire d'exception pour qu'il effectue certaines actions avant de permettre au programme de s'arrêter.

Application des concepts

Les sections suivantes montrent comment appliquer les termes et les concepts présentés auparavant dans ce chapitre.

Séquences d'échappement

Une *séquence d'échappement* est un type spécial de littéral caractère. Comme C/C++, Java utilise les séquences d'échappement pour représenter des caractères de contrôle spéciaux et des caractères non imprimables. Une séquence d'échappement est représentée par une barre oblique inversée (\) suivie du code d'un caractère. Le tableau suivant résume ces séquences d'échappement :

Caractère	Séquence d'échappement
Barre oblique inversée	\\
Retour arrière	\b
Retour chariot	\r
Guillemet	\"
Saut de page	\f
Tabulation horizontale	\t
Nouvelle ligne	\n
Caractère octal	\DDD
Apostrophe	\'
Caractère Unicode	\uHHHH

Les caractères numériques non décimaux sont des séquences d'échappement. Un caractère octal est représenté par une suite de trois chiffres octaux ; un caractère Unicode est représenté par un `u` minuscule suivi de quatre chiffres hexadécimaux. Par exemple, le nombre décimal 57 est représenté par le code octal `\071` et par la séquence Unicode `\u0039`.

La chaîne exemple de l'instruction suivante imprime sur la première ligne les mots `Nom` et `"Hildegard von Bingen"` séparés par deux tabulations et, sur la deuxième ligne, `ID` et `"1098"`, également séparés par deux tabulations :

```
String démoEchappement = new
    String("Nom\t\t\"Hildegard von Bingen\"\nID\t\t\"1098\"");
```

Gestion des chaînes

La chaîne de caractères que vous spécifiez dans une `String` est un littéral ; le programme l'utilisera telle quelle, sans aucune modification.

Cependant, la classe `String` fournit les moyens de chaîner les chaînes entre elles (ce qu'on appelle la *concaténation des chaînes*), de voir et d'utiliser ce qui est à l'intérieur des chaînes (comparer des chaînes, rechercher dans les chaînes ou extraire une sous-chaîne d'une chaîne) et de convertir en chaînes d'autres sortes de données. Voici quelques exemples :

- Déclarer des variables de type `String` et leur affecter des valeurs :

```
String prénoms = "Joseph, Elvira et Hans";
String modificateur = " aiment vraiment ";
String goûts = "le chocolat.";
```

- Obtenir une sous-chaîne d'une chaîne, en sélectionnant de la neuvième colonne à la fin de la chaîne :

```
String sous = prénoms(9); // "Elvira et Hans"
```

- Comparer une partie de la sous-chaîne à une autre chaîne, convertir une chaîne en majuscules, puis la concaténer avec d'autres chaînes afin d'obtenir une valeur de retour :

```
boolean bPrénom = prénoms.startsWith("Emine"); // Renvoie false dans ce cas.
String caps = modificateur.toUpperCase(); // Donne " AIMENT VRAIMENT "
return prénoms + caps + goûts; // Renvoie la ligne :
// Elvira et Hans AIMENT VRAIMENT le chocolat.
```

Pour plus d'informations sur la façon d'utiliser la classe `String`, voir la documentation API de Sun, à l'adresse <http://java.sun.com/j2se/1.3/docs/api/java/lang/String.html>, ou *The Java Class Libraries: An Annotated Reference*, de Patrick Chan et Rosanna Lee. Pour une couverture plus complète, voir aussi le supplément.

Pour une liste des gestionnaires de `String`, voir *Java in a Nutshell*, de David Flanagan. Pour une étude plus complète de la gestion de `String`, voir *The Java Handbook*, de Patrick Naughton.

StringBuffer

Pour mieux contrôler vos chaînes, utilisez la classe `StringBuffer`. Cette classe fait partie du paquet `java.lang`.

`StringBuffer` stocke vos chaînes dans un tampon, de sorte que vous déclarez une `String` qu'au moment où vous avez besoin d'une chaîne permanente. Entre autres avantages, vous n'avez pas à redéclarer de `String` lorsque le contenu de la chaîne change. Vous pouvez réserver une taille de tampon plus grande que ce qui s'y trouve déjà.

`StringBuffer` fournit des méthodes supplémentaires par rapport à `String` ; elles offrent de nouveaux moyens de modifier le contenu des chaînes. Par exemple, la méthode `setCharAt()` de `StringBuffer` remplace le caractère

dont la position est spécifiée par le premier paramètre par le caractère donné en second paramètre.

```
StringBuffer word = new StringBuffer ("jaune");
word.setCharAt (0, 'f'); //le mot est maintenant "faune"
```

Détermination des accès

Par défaut les classes sont disponibles pour tous les membres qu'elles contiennent, et ceux-ci sont disponibles les uns pour les autres. Mais, cet accès peut être grandement modifié.

Les *modificateurs d'accès* déterminent la visibilité des informations d'une classe ou d'un membre par rapport aux autres membres et aux autres classes. Les modificateurs d'accès sont :

- **public** : Un membre public est visible par les membres qui sont hors de sa portée, tant que la classe parent est visible. Une classe publique est visible par toutes les autres classes de tous les autres paquets.
- **private** : L'accès à un membre privé est limité à la classe de ce membre.
- **protected** : Un membre protégé peut être accédé par les autres membres de sa classe et par les membres des classes du même paquet (tant que la classe parent du membre est accessible), mais pas par les autres paquets. Une classe protégée est accessible aux autres classes du même paquet, mais pas aux autres paquets.
- Si aucun modificateur d'accès n'est déclaré, le membre est disponible pour toutes les classes se trouvant dans le paquet parent, mais pas à l'extérieur du paquet.

Voyons cela dans le contexte :

```
class tailleFine {
    private boolean invitationDonnée = false; // C'est privé.
    private int poids = 75                    // C'est privé aussi.

    public void accepterInvitation() {        // C'est public.
        invitationDonnée = true;
    }

    //La classe PetitFour est déclarée et l'objet petitFour est instancié
    //ailleurs :
    public void manger(PetitFour petitFour) {

        /*Cet objet n'accepte davantage de petits-fours que s'il a une invitation
        * et s'il est capable d'accepter. Remarquez que estNourritureAcceptée()
        * cherche à vérifier si l'objet est trop gros pour accepter davantage de
        * nourriture :
        */
        if (invitationDonnée && estNourritureAcceptée()) {
```



```

        /*Le nouveau poids de cet objet sera son poids initial
        * plus le poids du petitFour. Le poids est incrémenté
        * dès qu'on ajoute un petitFour :
        */
        poids += petitFour.obtenirPoids();
    }

    //Seul l'objet sait s'il accepte la nourriture :
    private boolean estNourritureAcceptée() {
        //Cet objet n'acceptera de la nourriture que s'il a la place :
        return (estTropGros() ? false : true);
    }

    //Les objets du même paquet peuvent voir si cet objet est trop gros :
    protected boolean estTropGros() {
        //Il peut accepter de la nourriture si son poids est inférieur à 80 :
        return (poids > 80) ? true : false;
    }
}

```

Remarquez que `estNourritureAcceptée()` et `invitationDonnée` sont **private**. Seuls les membres à l'intérieur de cette classe savent si l'objet est capable d'accepter de la nourriture ou s'il a une invitation.

`estTropGros()` est **protected**. Seules les classes de ce paquet peuvent voir si le poids de cet objet excède ou non sa limite.

Les seules méthodes qui sont exposées à l'extérieur sont `accepterInvitation()` et `manger()`. N'importe quelle classe peut percevoir ces méthodes.

Gestion des méthodes

La méthode `main()` requiert une attention particulière. Elle est le point d'entrée des programmes (à l'exception des applets). Elle s'écrit ainsi :

```

public static void main(String[] args) {
    ...
}

```

Des variantes sont possibles entre les parenthèses, mais la forme générale reste analogue.

Le mot clé **static** est important. Une méthode **static** est toujours associée à sa classe entière, plutôt qu'à une instance particulière de cette classe. (Le mot clé **static** peut aussi s'appliquer aux classes. Tous les membres d'une classe **static** sont associés à la classe parent entière de la classe.) Les méthodes **static** sont également appelées *méthodes de classe*.

Comme la méthode `main()` est le point de départ à l'intérieur du programme, elle doit être **static** afin de rester indépendante des nombreux objets que le programme peut générer à partir de sa classe parent.

L'association de `static` au niveau de la classe affecte la façon dont vous faites appel à une méthode `static` et dont vous faites appel aux autres méthodes depuis une méthode `static`. Les membres `static` peuvent être appelés depuis d'autres types de membres simplement à l'aide du nom de la méthode, et peuvent s'appeler les uns les autres de la même façon. Vous n'avez pas besoin de créer une instance de la classe pour accéder à une méthode `static` se trouvant dans cette classe.

Pour accéder à des membres non `static` d'une classe non `static` depuis une méthode `static`, vous devez instancier la classe du membre à atteindre et manipuler cette instance avec l'opérateur d'accès, exactement comme vous le feriez pour tout autre appel de méthode.

Remarquez que la méthode `main()` est un tableau de `String`, avec d'autres arguments possibles. N'oubliez pas que cette méthode est l'endroit où le compilateur commence son travail. Quand vous passez un argument depuis la ligne de commande, il est passé sous forme de chaîne au tableau de `String` dans la déclaration de la méthode `main()`, et cet argument est utilisé pour commencer l'exécution du programme. Quand vous passez un type de données autre que `String`, il est quand même reçu comme une chaîne. Vous devez programmer dans le corps de la méthode `main()` la conversion de `String` vers le type de données requis.

Utilisation des conversions de types

Rappel La conversion de type est le processus de conversion du type de données d'une variable pour la durée d'une opération particulière. La forme standard d'une conversion de raccourcissement s'appelle le transtypage ; il peut présenter un danger pour vos données.

Transtypage implicite

Dans certains cas, un transtypage peut être réalisé implicitement par le compilateur. Voici un exemple :

```
if (3 > 'a') {  
    ...  
}
```

Dans ce cas, la valeur `'a'` est convertie en nombre entier (valeur ASCII de la lettre `a`) avant d'être comparée au nombre `3`.

Conversion explicite

La syntaxe d'une conversion d'agrandissement est simple :

```
<nomAncienneValeur> = (<nouveau type>) <nomNouvelleValeur>
```

Java ne veut pas que vous fassiez une conversion de raccourcissement, c'est pourquoi vous devez être plus explicite lorsque vous le faites :

```

valeurFlottant = (float)valeurDouble; // Vers le float "valeurFlottant"
                                         // depuis le double "valeurDouble".

valeurLong = (long)valeurFlottant;    // Vers le long "valeurLong"
                                         // depuis le float "valeurFlottante".

// C'est une des quatre constructions possibles.

```

(Notez que par défaut les décimales sont arrondies à la valeur inférieure.) Assurez-vous de connaître parfaitement la syntaxe des types à transtyper ; ce processus peut devenir confus.

Pour plus d'informations, voir ["Conversion et transtypage des types de données", page 11-5.](#)

Contrôle du déroulement

Rappel Il y a trois types d'instructions de boucle : les instructions d'itération (`for`, `while` et `do-while`) créent des boucles, les instructions de sélection (`switch` et toutes les instructions `if`) indiquent au programme les cas d'utilisation des instructions, et les instruction de branchement (`break`, `continue` et `return`) passent le contrôle à une autre partie du programme.

Boucles

Dans un programme, chaque instruction est exécutée une fois. Cependant, il est quelquefois nécessaire d'exécuter des instructions plusieurs fois jusqu'à ce qu'une condition soit satisfaite. Avec Java, il y a trois façons de créer des boucles : boucles `while`, `do` et `for`.

- **La boucle while**

La boucle `while` est utilisée pour créer un bloc de code qui sera exécuté tant qu'une condition particulière est satisfaite. Voici la syntaxe générale de la boucle `while` :

```

while ( <instruction de condition booléenne> ) {
    <code à exécuter tant que la condition est vraie>
}

```

La boucle commence par tester la condition. Si la condition a la valeur `true`, la boucle exécute la totalité du bloc. Ensuite, elle teste la condition une nouvelle fois et répète ce processus jusqu'à ce que la condition prenne la valeur `false`. A ce moment, la boucle arrête de s'exécuter. Par exemple, pour imprimer "Bouclage" 10 fois :

```

int x = 0;                                //Initialise x à 0.
while (x < 10){                            //Instruction de condition booléenne.
    System.out.println("Bouclage");        //Imprime "Bouclage" une fois.
    x++;                                   //Incrémence x pour l'itération suivante.
}

```

Quand la boucle commence à s'exécuter, elle vérifie si la valeur de `x` est inférieure à 10. Comme c'est le cas, le corps de la boucle est exécuté. Le mot "Bouclage" est affiché à l'écran, puis la valeur de `x` est incrémentée. Cette boucle continue jusqu'à ce que la valeur de `x` soit égale à 10, auquel cas la boucle arrête de s'exécuter.

A moins que vous souhaitiez écrire une boucle infinie, vérifiez qu'il existe un cas où la valeur de la condition deviendra `false` et que la boucle pourra s'arrêter. Vous pouvez également arrêter une boucle à l'aide des instructions `return`, `continue` ou `break`.

- **La boucle do-while**

La boucle `do-while` ressemble à la boucle `while`, sauf qu'elle évalue la condition *après* les instructions et non avant. Le code suivant montre la boucle `while` précédente convertie en boucle `do` :

```
int x = 0;
do{
    System.out.println("Bouclage");
    x++;
}
while (x < 10);
```

La différence principale entre les deux boucles est que la boucle `do-while` s'exécute toujours au moins une fois, alors que la boucle `while` ne s'exécute pas si la condition initiale n'est pas remplie.

- **La boucle for**

La boucle `for` est la plus puissante des constructions de boucles. Voici la syntaxe générale d'une boucle `for` :

```
for ( <initialisation> ; <condition booléenne> ; <itération> ) {
    <code d'exécution>
}
```

La boucle `for` est constituée de trois parties : une expression d'initialisation, une expression conditionnelle booléenne et une expression d'itération. La troisième expression met généralement à jour la variable de la boucle initialisée par la première expression. Voici la boucle `for` équivalente à la boucle `while` précédente :

```
for (int x = 0; x < 10; x++){
    System.out.println("Bouclage");
}
```

Cette boucle `for` et la boucle `while` équivalente sont pratiquement identiques. Pour presque chaque boucle `for`, il existe une boucle `while` équivalente.

La boucle `for` est la plus souple des constructions de boucles, tout en restant très performante. Par exemple, une boucle `while` et une boucle `for` peuvent toutes les deux additionner les nombres de 1 à 20, mais la boucle `for` le fait avec une ligne de moins.

While :

```
int x = 1, z = 0;
while (x <= 20) {
    z += x;
    x++;
}
```

For :

```
int z = 0;
for (int x=1; x <= 20; x++) {
    z+= x;
}
```

Nous pouvons compresser la boucle `for` pour qu'elle s'exécute la moitié moins de fois :

```
for (int x=1,y=20, z=0; x<=10 && y>10; x++, y--) {
    z+= x+y;
}
```

Décomposons cette boucle en quatre parties principales :

- 1 L'expression d'initialisation : `int x =1, y=20, z=0`
- 2 La condition booléenne : `x<=10 && y>10`
- 3 L'expression d'itération : `x++, y--`
- 4 Le corps principal de code exécutable : `z+= x + y`

Instructions de contrôle des boucles

Ces instructions contrôlent les instructions de boucles.

- **L'instruction `break`**

L'instruction `break` permet de sortir d'une structure de boucle avant que la condition du test soit remplie. Quand la boucle rencontre une instruction `break`, elle se termine immédiatement en ignorant le code restant. Par exemple :

```
int x = 0;
while (x < 10){
    System.out.println("Bouclage");
    x++;
    if (x == 5)
        break;
    else
        ... //faire quelque chose d'autre
}
```

Dans cet exemple, la boucle s'arrête quand `x` est égal à 5.

- **L'instruction continue**

L'instruction continue permet d'ignorer le reste de la boucle et de reprendre l'exécution à l'itération suivante de la boucle.

```
for ( int x = 0 ; x < 10 ; x++){  
    if(x == 5)  
        continue; //revient au début de la boucle avec x=6  
    System.out.println("Bouclage");  
}
```

Cet exemple n'imprime pas "Bouclage" si x a la valeur 5, mais continue à l'imprimer pour 6, 7, 8 et 9.

Instructions conditionnelles

Avec les instructions conditionnelles, votre code a la possibilité de prendre des décisions. Dans Java, il existe deux structures conditionnelles : l'instruction if-else et l'instruction switch.

- **L'instruction if-else**

Voici la syntaxe d'une instruction if-else :

```
if (<condition1>) {  
    ... //bloc de code 1  
}  
else if (<condition2>) {  
    ... //bloc de code 2  
}  
else {  
    ... //bloc de code 3  
}
```

L'instruction if-else est généralement constituée de plusieurs blocs. Quand l'instruction if-else s'exécute, un seul des blocs est exécuté (celui dont les conditions sont vraies).

Les blocs else-if et else sont facultatifs. En outre, l'instruction if-else n'est pas limitée à trois blocs : elle peut contenir autant de blocs else-if que nécessaire.

Les exemples suivants montrent l'utilisation de l'instruction if-else :

```
if ( x % 2 == 0)  
    System.out.println("x est pair");  
else  
    System.out.println("x est impair");  
if (x == y)  
    System.out.println("x est égal à y");  
else if (x < y)  
    System.out.println("x est inférieur à y");  
else  
    System.out.println("x est supérieur à y");
```

• L'instruction switch

L'instruction `switch` est semblable à l'instruction `if-else`. Voici la syntaxe générale de l'instruction `switch` :

```
switch (<expression>){
    case <value1>: <blocCode1>;
        break;
    case <value2>: <blocCode2>;
        break;
    default      : <blocCode3>;
}
```

Remarquez ceci :

- S'il n'y a qu'une seule instruction dans un bloc de code, il n'est pas nécessaire de l'entourer d'accolades.
- Le bloc de code `default` correspond au bloc `else` d'une instruction `if-else`.
- Les blocs de code sont exécutés selon la valeur d'une variable ou d'une expression, pas d'une condition.
- La valeur de `<expression>` doit être de type entier (ou d'un type qui peut être converti en `int` sans risque, comme `char`).
- Les valeurs `case` doivent être des expressions constantes du même type que l'expression initiale.
- Le mot clé `break` est facultatif. Il est utilisé pour terminer l'exécution de l'instruction `switch` une fois qu'un code de bloc est exécuté. S'il n'est pas utilisé après `blocCode1`, alors `blocCode2` s'exécute immédiatement après l'exécution de `blocCode1`.
- Si un bloc de code doit s'exécuter quand `expression` prend une valeur parmi un certain nombre de valeurs, chacune doit être spécifiée comme ceci : `case <valeur>;`.

Voici un exemple, où `c` est de type `char` :

```
switch (c){
    case '1': case '3': case '5': case '7': case '9':
        System.out.println("c est un chiffre impair");
        break;
    case '0': case '2': case '4': case '6': case '8':
        System.out.println("c est un chiffre pair");
        break;
    case ' ':
        System.out.println("c est un espace");
        break;
    default :
        System.out.println("c n'est ni un chiffre ni un espace");
}
```

L'instruction `switch` évalue `c` et passe directement à l'instruction `case` dont la valeur est égale à `c`. Si aucune des valeurs `case` n'est égale à `c`, la

section `default` est exécutée. Remarquez la façon d'utiliser plusieurs valeurs pour chaque bloc.

Gestion des exceptions

La gestion des exceptions offre des moyens structurés de capturer les erreurs d'exécution de votre programme et de fournir des informations significatives à leur sujet. Vous pouvez aussi définir le gestionnaire d'exception pour qu'il effectue certaines actions avant de permettre au programme de s'arrêter. La gestion des exceptions utilise les mots clés `try`, `catch` et `finally`. Une méthode peut déclarer une exception à l'aide des mots clés `throws` et `throw`.

Dans Java, une exception peut être une sous-classe de la classe `java.lang.Exception` ou de la classe `java.lang.Error`. Quand une méthode déclare qu'une exception est survenue, nous disons qu'elle *déclenche* une exception. *Capter* une exception gère l'exception.

Les exceptions déclarées explicitement dans la déclaration de la méthode *doivent* être capturées, sinon le code ne se compilera pas. Les exceptions déclarées explicitement dans la déclaration de la méthode provoqueront quand même l'interruption du programme à l'exécution, mais il se compilera. Notez qu'une bonne gestion des exceptions rend votre code plus robuste.

Pour capturer une exception, vous imbriquez le code qui peut la provoquer dans un bloc `try`, puis vous imbriquez le code qui va la gérer dans un bloc `catch`. S'il y a du code important (par exemple du code effectuant un nettoyage) que vous voulez exécuter même si une exception est déclenchée et que le programme s'arrête, placez ce code à la fin, dans un bloc `finally`. Voici un exemple de ce fonctionnement :

```
try {
    ... // Insérer ici du code pouvant déclencher une exception.
}
catch( Exception e ) {
    ... // Insérer ici le code de gestion de l'exception.
    // La ligne suivante ouvre un suivi de pile de l'exception :
    e.printStackTrace();
}
finally {
    ... // Le code inséré ici sera toujours exécuté,
        // que l'exception ait été déclenchée dans le bloc try ou non.
}
```

Le bloc `try` doit être utilisé pour entourer tout code susceptible de déclencher une exception ayant besoin d'être gérée. Si aucune exception n'est déclenchée, tout le code du bloc `try` est exécuté. Mais, si une exception est déclenchée, le code du bloc `try` arrête l'exécution à l'endroit où l'exception a été déclenchée et le contrôle passe au bloc `catch`, dans lequel l'exception est gérée.

Vous pouvez faire tout ce dont vous avez besoin pour gérer l'exception dans un ou plusieurs blocs `catch`. Le moyen le plus simple de gérer des exceptions est de les gérer toutes dans un seul bloc `catch`. Pour cela, l'argument entre les parenthèses suivant `catch` doit indiquer la classe `Exception`, suivie d'un nom de variable à affecter à cette exception. Cela indique que toute exception qui est une instance de `java.lang.Exception` ou de n'importe laquelle de ses sous-classes sera capturée ; en d'autres termes, toute exception.

Si vous avez besoin d'écrire un code de gestion différent selon le type de l'exception, vous pouvez utiliser plusieurs blocs `catch`. Dans ce cas, au lieu de passer `Exception` comme type d'exception dans l'argument `catch`, indiquez le nom de classe du type d'exception particulier que vous voulez capturer. Ce peut être toute sous-classe de `Exception`. N'oubliez pas que le bloc `catch` capturera toujours le type d'exception indiqué ou une de ses sous-classes.

Le code se trouvant dans le bloc `finally` sera toujours exécuté, même si le bloc `try` ne se termine pas pour une raison quelconque. Par exemple, le code se trouvant dans le bloc `try` peut être interrompu parce qu'il déclenche une exception, mais le code se trouvant dans le bloc `finally` s'exécutera quand même. Le bloc `finally` est un bon endroit pour placer du code de nettoyage.

Si vous savez qu'une méthode que vous écrivez va être appelée par d'autre code, vous pouvez laisser le code appelant gérer l'exception que votre méthode peut déclencher. Dans ce cas, il suffit de déclarer que la méthode peut déclencher une exception. Le code qui pourrait déclencher une exception peut utiliser le mot clé `throws` pour déclarer l'exception. Ce procédé peut être une alternative à la capture de l'exception, puisque si une méthode déclare qu'elle déclenche une exception, elle n'a pas besoin de la gérer.

Voici un exemple d'utilisation de `throws` :

```
public void maMéthode() throws UneException {
    ... // Le code inséré ici peut déclencher UneException
        // ou une de ses sous-classes.
        // UneException est supposée être une sous-classe de Exception.
}
```

Vous pouvez également utiliser le mot clé `throw` pour indiquer que quelque chose ne va pas bien. Par exemple, vous pouvez l'utiliser pour déclencher une exception à vous quand un utilisateur entre des informations incorrectes et que vous voulez lui afficher un message d'erreur. Pour cela, utilisez une instruction du type :

```
throw new UneException("saisie incorrecte");
```


Les bibliothèques des classes Java

Pour prendre en charge certaines fonctionnalités, la plupart des langages de programmation reposent sur des bibliothèques de classes déjà construites. Dans Java, ces groupes de classes liées, que l'on appelle des paquets, varient selon l'édition du langage Java. Chaque édition est utilisée à des fins spécifiques, par exemple pour les applications, pour les applications d'entreprise et pour les produits grand public.

Editions de la plate-forme Java 2

La plate-forme Java 2 est disponible dans plusieurs éditions utilisées à des fins diverses. Comme Java est un langage qui peut s'exécuter partout et sur n'importe quelle plate-forme, il est utilisé dans divers environnements : Internet, intranets, électronique grand public et applications sur ordinateur. En raison de ses diverses applications, Java est fourni en plusieurs éditions : Java 2 Standard Edition (J2SE), Java 2 Enterprise Edition (J2EE) et Java 2 Micro Edition (J2ME). Dans certains cas, comme pour le développement d'applications d'entreprise, un ensemble important de paquets est utilisé. Dans d'autres cas, comme pour les produits d'électronique grand public, seule une petite partie du langage est utilisée. Chaque édition contient un kit de développement

Java 2 (SDK) pour développer des applications et un environnement d'exécution Java 2 (JRE) pour les exécuter.

Tableau 5.1 Editions de la plate-forme Java 2

Plate-forme Java 2	Abréviation	Description
Standard Edition	J2SE	Contient les classes qui forment le cœur du langage Java.
Enterprise Edition	J2EE	Contient les classes J2SE plus d'autres classes pour le développement d'applications d'entreprise.
Micro Edition	J2ME	Contient un sous-ensemble des classes J2SE destiné aux produits électroniques grand public.

Standard Edition

La plate-forme Java 2, Standard Edition (J2SE), fournit aux développeurs un environnement de développement riche en fonctionnalités, stable, sécurisé et multiplate-forme. Cette édition de Java prend en charge des fonctionnalités majeures comme la connectivité aux bases de données, la création d'interfaces utilisateur, les entrées/sorties, la programmation en réseau, et contient les paquets fondamentaux du langage Java.

Voir aussi Java 2 Platform Standard Edition Overview
 (<http://java.sun.com/j2se/1.3/>)
 "Introducing the Java™ Platform"
 (<http://developer.java.sun.com/developer/onlineTraining/new2java/programming/intro/>)
 "Paquets de Java 2 Standard Edition", page 5-3

Enterprise Edition

La plate-forme Java 2, Enterprise Edition (J2EE) fournit au développeur des outils permettant de construire et de déployer des applications d'entreprise multiniveaux. J2EE comprend les paquets de J2SE plus d'autres paquets supportant le développement des Enterprise JavaBeans, les servlets Java, les pages JSP (JavaServer Pages), XML et un contrôle souple des transactions.

Voir aussi Java 2 Platform Enterprise Edition Overview
 (<http://java.sun.com/j2ee/overview.html>)
 Java 2 Enterprise Edition technical articles
 (<http://developer.java.sun.com/developer/technicalArticles/J2EE/index.html>)

Micro Edition

La plate-forme Java 2 Micro Edition (J2ME) est utilisée pour divers produits électroniques grand public, comme les pageurs, les cartes à mémoire, les téléphones cellulaires, les assistants personnels numériques et les STB. J2ME fournit les mêmes avantages du langage Java concernant la portabilité du code sur diverses plates-formes, la capacité à s'exécuter partout et les livraisons sécurisées sur le réseau que J2SE et J2EE, elle utilise un ensemble de paquets plus petit. J2ME comprend un sous-ensemble des paquets J2SE plus un paquet spécifique à la Micro Edition, `javax.microedition.io`. En outre, on peut faire évoluer les applications J2ME pour qu'elles fonctionnent avec J2SE et J2EE.

Voir aussi Java 2 Platform Micro Edition Overview
(<http://java.sun.com/j2me/>)
Consumer & Embedded Products technical articles
(<http://developer.java.sun.com/developer/technicalArticles/ConsumerProducts/index.html>)

Paquets de Java 2 Standard Edition

La plate-forme Java 2 Standard Edition (J2SE) est livrée avec une importante bibliothèque qui comprend le support de la connectivité aux bases de données, la conception des interfaces utilisateurs, les entrées/sorties et la programmation en réseau. Ces bibliothèques sont réparties en groupes de classes liées, appelés paquets. Le tableau suivant décrit brièvement certains de ces paquets.

Tableau 5.2 Paquets de J2SE

Paquet	Nom du paquet	Description
Langage	<code>java.lang</code>	Classes qui forment le cœur du langage Java.
Utilitaires	<code>java.util</code>	Prise en charge des structures de données utilitaires.
E/S	<code>java.io</code>	Prise en charge de divers types d'entrée/sortie
Texte	<code>java.text</code>	Support de localisation pour la gestion du texte, des dates, des nombres et des messages.
Math	<code>java.math</code>	Classes permettant des calculs en précision entière et en virgule flottante.
AWT	<code>java.awt</code>	Conception d'interfaces utilisateur et gestion des événements
Swing	<code>javax.swing</code>	Classes pour créer des composants légers tout-Java qui se comportent de la même façon sur toutes les plates-formes.
Javax	<code>javax</code>	Extensions du langage Java.
Applet	<code>java.applet</code>	Classes pour créer des applets.

Tableau 5.2 Paquets de J2SE (suite)

Paquet	Nom du paquet	Description
Beans	<code>java.beans</code>	Classes pour développer des JavaBeans.
Réflexion	<code>java.lang.reflect</code>	Classes utilisées pour obtenir à l'exécution des informations sur les classes
SQL	<code>java.sql</code>	Accès aux données des bases de données et traitement.
RMI	<code>java.rmi</code>	Prise en charge de la programmation distribuée.
Réseau	<code>java.net</code>	Classes qui supportent le développement d'applications en réseau.
Sécurité	<code>java.security</code>	Prise en charge de la sécurité par cryptographie.

Remarque Les paquets Java dépendent de l'édition de la plate-forme Java 2. Le kit de développement Java 2 (SDK) est disponible dans plusieurs éditions utilisées à des fins diverses : Standard Edition (J2SE), Enterprise Edition (J2EE) et Micro Edition (J2ME).

Voir aussi [“Éditions de la plate-forme Java 2”, page 5-1](#)
“Java 2 Platform, Standard Edition, API Specification” dans la spécification de l'API du JDK
Le tutoriel de Sun, “Creating and using packages”
(<http://www.java.sun.com/docs/books/tutorial/java/interpack/packages.html>)
“Paquets” dans “Création et gestion des projets” de *Introduction à JBuilder*.

Le paquet du langage : `java.lang`

Le paquet `java.lang` est un des plus importants de la bibliothèque des classes Java. Ce paquet, qui est importé automatiquement dans chaque programme Java, contient les principales classes relatives au langage qui sont les bases de la conception du langage de programmation Java.

Voir aussi `java.lang` dans la documentation de l'API du JDK
[“Principales classes `java.lang`”, page 5-10](#)

Le paquet des utilitaires : `java.util`

Le paquet `java.util` contient plusieurs classes et interfaces utilitaires cruciales pour le développement Java. Les classes de ce paquet prennent en charge le cadre de travail des collections et les fonctionnalités de date et heure.

Voir aussi `java.util` dans la documentation de l'API du JDK
[“Principales classes `java.util`”, page 5-16](#)

Le paquet des E/S : `java.io`

Le paquet `java.io` prend en charge la lecture et l'écriture de données sur différentes unités. Java supporte également les flux de caractères en entrée et en sortie. En outre, la classe `File` du paquet `java.io` utilise une représentation abstraite et indépendante du système des noms de chemins des fichiers et des répertoires, afin d'offrir un meilleur support des plates-formes non UNIX. Les classes de ce paquet sont réparties dans les groupes suivants : classes de flux d'entrée, classes de flux de sortie, classes de fichiers et classe `StreamTokenizer`.

Voir aussi `java.io` dans la documentation de l'API du JDK
[“Principales classes `java.io`”, page 5-19](#)

Le paquet de texte : `java.text`

Le paquet `java.text` contient des classes et des interfaces qui fournissent le support de localisation pour la gestion du texte, des dates, des nombres et des messages. Les classes de ce paquet, comme `NumberFormat`, `DateFormat` et `Collator`, permettent de formater des nombres, des dates, des heures et des chaînes conformément aux usages locaux. D'autres classes prennent en charge l'analyse, la recherche et le tri des chaînes de caractères.

Voir aussi `java.text` dans la documentation de l'API du JDK
 “Internationalisation des programmes avec `JBuilder`” dans *Construction d'applications avec `JBuilder`*

Le paquet mathématique : `java.math`

Le paquet `java.math`, à ne pas confondre avec la classe `java.lang.Math`, fournit des classes pour le calcul en précision entière (`BigInteger`) et en virgule flottante (`BigDecimal`).

La classe `BigInteger` fournit le support de la représentation des entiers arbitrairement grands.

La classe `BigDecimal` est utilisée pour les calculs nécessitant le support des décimales, comme les calculs de conversion des monnaies, et fournit des opérations arithmétiques de base, la manipulation des échelles, les comparaisons, la conversion des formats et le hachage.

Voir aussi `java.math` dans la documentation de l'API du JDK
`java.lang.Math` dans la documentation de l'API du JDK
 “Arbitrary-Precision Math” dans le JDK Guide to Features

Le paquet AWT : java.awt

La paquet AWT (Abstract Window Toolkit), qui fait partie des Java Foundation Classes (JFC), fournit le support de la programmation des interfaces graphiques (GUI) et comprend des fonctionnalités comme les composants d'interface utilisateur, les modèles de gestionnaires d'événements, les gestionnaires de dispositions, les outils graphiques et les classes de transfert de données pour les opérations de couper/coller.

Voir aussi `java.awt` dans la documentation de l'API du JDK
 "Abstract Window Toolkit (AWT)" dans le JDK Guide to Features
 "AWT Fundamentals"
 (<http://developer.java.sun.com/developer/onlineTraining/awt/>)
 "Tutoriel : Construction d'une applet" dans *Introduction à JBuilder*
 "Construction d'une interface utilisateur" et "Utilisation des gestionnaires de dispositions" dans *Conception d'interfaces utilisateur avec JBuilder*

Le paquet Swing : javax.swing

La paquet `javax.swing` fournit un ensemble de composants "légers" (langage tout-Java) qui prennent automatiquement l'apparence du système d'exploitation de n'importe quelle plate-forme. Les composants Swing sont des versions 100 % pur Java des composants AWT existants (bouton, barre de défilement ou libellé), plus un autre ensemble de composants (vue arborescente, table ou volet à onglets).

Remarque Les paquets `javax` sont des extensions du langage Java de base.

Voir aussi `java.swing` dans la documentation de l'API du JDK
 "Project Swing (Java™ Foundation Classes) Software" dans le JDK Guide to Features
 "Java Foundation Classes (JFC)"
 (<http://java.sun.com/docs/books/tutorial/post1.0/preview/jfc.html>)
 Le tutoriel Swing de Sun, "Trail: Creating a GUI with JFC/Swing"
 (<http://www.java.sun.com/docs/books/tutorial/uiswing/index.html>)
 "Construction d'un éditeur de texte Java" dans *Conception d'interfaces utilisateur avec JBuilder*
 Chapitres concernés dans *Conception d'interfaces utilisateur avec JBuilder* :

- "Construction d'une interface utilisateur"
- "Disposition de votre interface utilisateur"
- "Utilisation des gestionnaires de disposition"
- "Utilisation des panneaux et des dispositions imbriqués"

Les paquets Javax : javax

Les nombreux paquets `javax` sont des extensions du langage Java de base. Ils comprennent des paquets comme `javax.swing`, `javax.sound`, `javax.rmi`,

`javax.transactions` et `javax.naming`. Les développeurs peuvent aussi créer leurs propres paquets `javax` personnalisés.

Voir aussi `javax.accessibility`, `javax.naming`, `javax.rmi`, `javax.sound.midi`, `javax.sound.sampled`, `javax.swing` et `javax.transaction` dans la documentation de l'API du JDK

Le paquet Applet : `java.applet`

Le paquet `java.applet` fournit les classes permettant de créer des applets, ainsi que les classes utilisées par ces applets pour communiquer avec le contexte des applets, généralement un navigateur web. Les applets sont des programmes Java non prévus pour s'exécuter tout seuls et qu'il faut incorporer dans une autre application. Communément, les applets sont stockées sur un serveur Internet/intranet, appelées par une page HTML et téléchargées sur diverses plates-formes client où elles peuvent s'exécuter dans une machine virtuelle Java (JVM) fournie par le navigateur de la machine client. La livraison et l'exécution se font sous la surveillance d'un gestionnaire de sécurité, qui peut empêcher les applets d'exécuter certaines tâches, comme le formatage du disque dur ou l'ouverture de connexions vers des machines "douteuses".

Pour des raisons de sécurité et de compatibilité du navigateur JDK, il est important de bien comprendre les applets avant d'en développer. Les applets ne possèdent pas toutes les fonctionnalités des programmes Java et ce pour des raisons de sécurité. Les applets sont en outre basées sur la version du JDK du navigateur qui peut ne pas être celui en cours. Au moment où nous écrivons ces lignes, de nombreux navigateurs ne supportent pas complètement le dernier JDK. Par exemple, la plupart des navigateurs incluent une ancienne version du JDK qui ne prend pas en charge Swing. Par conséquent, les applets utilisant des composants Swing ne fonctionnent pas dans ces navigateurs.

Voir aussi `java.applet` dans la documentation de l'API du JDK
 Le tutoriel de Sun, "Trail: Writing applets"
 (<http://www.java.sun.com/docs/books/tutorial/applet/index.html>)
[Chapitre 9, "Introduction à la machine virtuelle Java"](#)
 "Utilisation des applets" dans *Développement d'applications web* de JBuilder
 "Tutoriel : Construction d'une applet" dans *Introduction à JBuilder*

Le paquet Beans : `java.beans`

La paquet `java.beans` contient les classes relatives au développement des JavaBeans. Les JavaBeans, classes Java qui servent de composants auto-contenus et réutilisables, étendent la capacité "écrire une fois, exécuter partout" de la plate-forme Java au développement de composants réutilisables. Ces morceaux de code réutilisables peuvent être

manipulés et mis à jour avec un minimum de conséquences sur les tests du programme.

Voir aussi `java.beans` dans la documentation de l'API du JDK
JavaBeans™ Technology technical articles
(<http://developer.java.sun.com/developer/technicalArticles/jbeans/index.html>)
“Création de JavaBeans avec BeansExpress” dans *Construction d'applications avec JBuilder*

Le paquet des réflexions : `java.lang.reflect`

Le paquet `java.lang.reflect` fournit des classes et des interfaces permettant d'examiner et de manipuler les classes à l'exécution. La réflexion permet l'accès aux informations relatives aux champs, méthodes et constructeurs des classes chargées. Le code Java peut utiliser ces informations réfléchies pour opérer sur les objets correspondants.

Les classes de ce paquet fournissent des applications comme les débogueurs, les interpréteurs, les inspecteurs d'objets ou les navigateurs de classes, ainsi que des services comme la sérialisation des objets et l'accès des JavaBeans aux membres publics ou aux membres déclarés par une classe.

Voir aussi `java.lang.reflect` dans la documentation de l'API du JDK
“Reflection” dans le JDK Guide to Features

Le paquet SQL : `java.sql`

Le paquet `java.sql` contient des classes qui fournissent l'API permettant d'accéder aux données d'une source de données et de les traiter. Le paquet `java.sql` est également appelé API JDBC 2.0 (Java Database Connectivity). Cette API inclut un cadre de travail permettant d'installer de façon dynamique différents pilotes afin d'accéder à différents types de sources de données. JDBC est un standard qui permet à la plate-forme Java de se connecter avec presque toutes les bases de données, y compris celles écrites dans d'autres langages comme SQL (Structured Query Language).

Le paquet `java.sql` contient des classes, interfaces et méthodes permettant d'établir des connexions aux bases de données, d'envoyer des instructions à une base de données, de récupérer et de mettre à jour les résultats d'une requête, de mettre en correspondance des valeurs SQL, de fournir des informations sur une base de données, de déclencher des exceptions et d'assurer la sécurité.

Voir aussi `java.sql` dans la documentation de l'API du JDK
Le tutoriel de Sun, “Trail: JDBC(TM) Database Access”
(<http://web2.java.sun.com/docs/books/tutorial/jdbc/index.html>)

“Référence SQL” dans le *Guide du développeur JDataStore* de JBuilder

Le paquet RMI : java.rmi

Le paquet `java.rmi` fournit des classes pour RMI (Remote Method Invocation) Java. RMI permet une communication à distance entre des programmes écrits en langage Java. RMI est un mécanisme qui permet à un objet sur une machine virtuelle Java (JVM) d'appeler les méthodes d'un autre objet sur une autre JVM.

Voir aussi `java.rmi` dans la documentation de l'API du JDK
 “Java Remote Method Invocation (RMI)” dans le JDK Guide to Features
 “The Java Remote Method Invocation - Distributed Computing for Java (a White Paper)”
 (<http://java.sun.com/marketing/collateral/javarmi.html>)
 “Tutoriel : Exploration des applications distribuées basées sur RMI Java dans JBuilder” dans le *Guide du développeur d'applications distribuées* de JBuilder
 Les applications RMI exemple de JBuilder : `samples/RMI/et DataExpress/StreamableDataSets/` dans le répertoire d'installation de JBuilder

Le paquet réseau : java.net

Le paquet `java.net` contient des classes permettant de développer des applications en réseau. En utilisant les classes socket, vous pouvez communiquer avec n'importe quel serveur sur Internet ou implémenter votre propre serveur Internet. Les classes sont également fournies pour récupérer des données depuis Internet.

Voir aussi `java.net` dans la documentation de l'API du JDK
 “Networking Features” dans le JDK Guide to Features

Le paquet de sécurité : java.security

Le paquet de sécurité, `java.security`, définit des classes et des interfaces permettant de mettre en œuvre la sécurité. Il existe deux catégories de classes :

- Les classes qui implémentent le contrôle des accès et empêchent le code douteux d'exécuter des opérations sensibles.
- Les classes d'authentification qui implémentent les condensés de messages et les signatures numériques et authentifient les classes et autres objets.

A l'aide de ces classes, les développeurs peuvent protéger l'accès aux applets et au code Java, y compris aux applications, aux beans et aux servlets, en créant des permissions et des politiques de sécurité. Quand le

code est chargé, de permissions lui sont affectées en fonction des politiques de sécurité. Les permissions spécifient les ressources auxquelles on peut accéder, comme les accès en lecture/écriture ou à une connexion. La politique, qui détermine quelles permissions sont disponibles, est généralement initialisée à partir d'un fichier externe configurable qui définit la politique de sécurité du code. L'utilisation des permissions et d'une politique permet de contrôler l'accès au code de façon souple, configurable et extensible.

Voir aussi `java.security` dans la documentation de l'API du JDK
 "Security" dans le JDK Guide to Features

Principales classes `java.lang`

La classe `Object` : `java.lang.Object`

La classe `Object` du paquet `java.lang` est la classe parent ou superclasse de toutes les classes Java. Cela signifie simplement que la classe `Object` est la racine de la hiérarchie des classes et que toutes les classes Java en sont dérivées. La classe `Object` elle-même contient un constructeur et plusieurs méthodes importantes, dont `clone()`, `equals()` et `toString()`.

Méthode	Argument	Description
<code>clone</code>	<code>()</code>	Crée et renvoie la copie d'un objet.
<code>equals</code>	<code>(Object obj)</code>	Indique si un autre objet est égal à l'objet spécifié.
<code>toString</code>	<code>()</code>	Renvoie la représentation d'un objet sous forme de chaîne

Un objet qui utilise la méthode `clone()` fait simplement une copie de lui-même. Quand la copie se fait, une certaine quantité de mémoire est d'abord affectée au clone, puis le contenu de l'objet initial est copié dans l'objet clone. Dans l'exemple suivant où la classe `Document` implémente l'interface `Cloneable`, une copie de la classe `Document` contenant une propriété `text` et `author` est créée à l'aide de la méthode `clone()`. Un objet n'est considéré comme clonable que lorsqu'il implémente l'interface `Cloneable`.

```
Document document1 = new Document("docText.txt", "Jean Dupont");
Document document2 = document1.clone();
```

La méthode `equals()` compare deux objets du même type sur la base de leurs propriétés. Elle renvoie une valeur booléenne qui dépend de l'objet qui a appelé la méthode et de l'objet qui lui a été transmis. Par exemple, si `equals()` est appelée par un objet qui lui transmet un objet identique, la méthode `equals()` renvoie la valeur `true`.

La méthode `toString()` donne un résultat de type `String` qui représente la valeur de l'objet. Pour que cette méthode renvoie des informations correctes quel que soit le type d'objet, la classe de l'objet doit la redéfinir.

Voir aussi `java.lang.Object` dans la documentation de l'API du JDK

Classes d'enveloppe de type

Pour des raisons de performance, les types de données primitifs ne sont pas utilisés en tant qu'objets dans Java. Ces types de données primitifs sont les nombres, les booléens et les caractères.

Cependant, certaines classes et méthodes Java nécessitent que les types de données primitifs soient des objets. Java utilise des classes pour envelopper ou encapsuler le type primitif en tant qu'objet, comme indiqué dans le tableau suivant.

Type primitif	Description	Enveloppe
<code>boolean</code>	True ou False (1 bit)	<code>java.lang.Boolean</code>
<code>byte</code>	-128 à 127 (nombre entier signé sur 8 bits)	<code>java.lang.Byte</code>
<code>char</code>	Caractère Unicode (16 bits)	<code>java.lang.Character</code>
<code>double</code>	+1.79769313486231579E+308 à +4.9406545841246544E-324 (64 bits)	<code>java.lang.Double</code>
<code>float</code>	+3.40282347E+28 à +1.40239846E-45 (32 bits)	<code>java.lang.Float</code>
<code>int</code>	-2147483648 à 2147483647 (nombre entier signé sur 32 bits)	<code>java.lang.Integer</code>
<code>long</code>	-9223372036854775808 à 9223372036854775807 (nombre entier signé sur 64 bits)	<code>java.lang.Long</code>
<code>short</code>	-32768 à 32767 (nombre entier signé sur 16 bits)	<code>java.lang.Short</code>
<code>void</code>	Une classe de réservation non instanciable pour contenir l'objet classe représentant le type Java primitif <code>void</code> .	<code>java.lang.Void</code>

Le constructeur d'une classe d'enveloppe, comme `Character(char valeur)`, prend simplement comme argument le type de classe qu'elle enveloppe. Par exemple, le code suivant montre la construction d'une classe d'enveloppe du type `Character`.

```
Character charWrapper = new Character('T');
```

Bien que chacune de ces classes contienne ses propres méthodes, plusieurs d'entre elles sont standard pour plusieurs objets. Ces méthodes sont des méthodes qui renvoient un type primitif, `toString()` et `equals()`.

Chaque classe d'enveloppe a une méthode, comme `charValue()`, qui renvoie le type primitif de cette classe. Le code suivant illustre l'utilisation de l'objet `charWrapper`. Remarquez que `charPrimitive` est un type de données

primitif (déclaré `char`). De cette façon, des types de données primitifs peuvent être attribués aux enveloppes de types.

```
char charPrimitive = charWrapper.charValue();
```

Les méthodes `toString()` et `Equals()` sont utilisées de la même façon que dans la classe `Object`.

La classe `Math` : `java.lang.Math`

La classe `Math` du paquet `java.lang`, à ne pas confondre avec le paquet `java.math`, fournit des méthodes utiles permettant d'implémenter les fonctions mathématiques communes. Cette classe n'est pas instanciée et est déclarée comme `final`, ce qui signifie qu'elle ne peut pas avoir de sous-classe. Parmi les méthodes de cette classe, se trouvent : `sin()`, `cos()`, `exp()`, `log()`, `max()`, `min()`, `random()`, `sqrt()`, et `tan()`. Voici plusieurs exemples de ces méthodes.

```
double d1 = Math.sin(45);
double d2 = 23.4;
double d3 = Math.exp(d2);
double d4 = Math.log(d3);
double d5 = Math.max(d2, Math.pow(d1, 10));
```

Certaines de ces méthodes sont surchargées pour accepter et renvoyer différents types de données.

La classe `Math` déclare également les constantes `PI` et `E`.

Remarque Le paquet `java.math`, contrairement à `java.lang.Math`, fournit des classes de support permettant de manipuler des nombres arbitrairement grands.

Voir aussi `java.lang.Math` dans la documentation de l'API du JDK
`java.math` dans la documentation de l'API du JDK

La classe `String` : `java.lang.String`

La classe `String` du paquet `java.lang` est utilisé pour représenter des chaînes de caractères. A la différence de C/C++, Java n'utilise pas de tableaux de caractères pour représenter des chaînes. Les chaînes sont constantes, c'est-à-dire que leur valeur ne peut plus changer une fois qu'elles ont été créées. La classe `String` est habituellement construite quand le compilateur Java rencontre une chaîne de caractères entre guillemets. Il y a cependant plusieurs façons de construire des chaînes. Le

tableau suivant contient plusieurs constructeurs de `String` et les arguments qu'ils acceptent.

Constructeur	Argument	Description
<code>String</code>	<code>()</code>	Initialise un nouvel objet <code>String</code> .
<code>String</code>	<code>(String valeur)</code>	Initialise un nouvel objet <code>String</code> avec le contenu de l'argument <code>String</code> .
<code>String</code>	<code>(char[] valeur)</code>	Crée un nouvel objet <code>String</code> contenant le tableau dans le même ordre.
<code>String</code>	<code>(char[] valeur, int offset, int compte)</code>	Crée un nouvel objet <code>String</code> contenant un sous-tableau de l'argument.
<code>String</code>	<code>(StringBuffer buffer)</code>	Initialise un nouvel objet <code>String</code> avec le contenu de <code>StringBuffer</code> .

La classe `String` contient plusieurs méthodes importantes, essentielles dans le traitement des chaînes de caractères. Ces méthodes sont utilisées pour modifier, comparer et analyser les chaînes. Comme les chaînes sont immuables et ne peuvent être modifiées, aucune de ces méthodes ne change la séquence de caractères. La classe `StringBuffer`, traitée dans la prochaine section, fournit des méthodes pour modifier les chaînes.

Le tableau suivant dresse la liste de quelques méthodes parmi les plus utiles, avec ce qu'elles acceptent et renvoient.

Méthode	Argument	Renvoie	Description
<code>length</code>	<code>()</code>	<code>int</code>	Renvoie le nombre de caractères de la chaîne.
<code>charAt</code>	<code>(int indice)</code>	<code>char</code>	Renvoie le caractère à l'indice spécifié dans la chaîne.
<code>compareTo</code>	<code>(String valeur)</code>	<code>int</code>	Compare une chaîne à la chaîne de l'argument.
<code>indexOf</code>	<code>(int car)</code>	<code>int</code>	Renvoie l'indice de la première occurrence du caractère spécifié.
<code>substring</code>	<code>(int indiceDébut, int indiceFin)</code>	<code>String</code>	Renvoie une nouvelle chaîne qui est une sous-chaîne de la chaîne.
<code>concat</code>	<code>(String chaîne)</code>	<code>String</code>	Insère l'objet <code>String</code> spécifié à la fin de cette chaîne.
<code>toLowerCase</code>	<code>()</code>	<code>String</code>	Renvoie la chaîne en minuscules.
<code>toUpperCase</code>	<code>()</code>	<code>String</code>	Renvoie la chaîne en majuscules.
<code>valueOf</code>	<code>(Object obj)</code>	<code>String</code>	Renvoie la représentation sous forme de chaîne de l'argument <code>Object</code> .

Comme elles sont surchargées pour plus de souplesse, ces méthodes sont encore plus puissantes. Les exemples suivants illustrent l'utilisation de la classe `String` et de quelques-unes de ses méthodes.

```
String s1 = new String("Hello World.");

char cArray[] = {'J', 'B', 'u', 'i', 'l', 'd', 'e', 'r'};
String s2 = new String(cArray);    //s2 = "JBuilder"

int i = s1.length();               //i = 12
char c = s1.charAt(6);             //c = 'W'
i = s1.indexOf('e');               //i = 1 (indice de 'e' dans "Hello World.")

String s3 = "abcdef".substring(2, 5); //s3 = "cde"
String s4 = s3.concat("f");         //s4 = "cdef"
String s5 = String.valueOf(i);      //s5 = "1" (valueOf() est statique)
```

Remarque Souvenez-vous que les indices de tableau et de `String` commencent à zéro.

Voir aussi `java.lang.String` dans la documentation de l'API du JDK

La classe `StringBuffer` : `java.lang.StringBuffer`

La classe `StringBuffer` du paquet `java.lang`, comme la classe `String`, représente une séquence de caractères. Contrairement à une chaîne, le contenu d'un `StringBuffer` peut être modifié. À l'aide de diverses méthodes `StringBuffer`, la longueur et le contenu du tampon de chaînes peut être modifié. De plus, l'objet `StringBuffer` peut devenir plus long, si nécessaire. Enfin, après la modification de `StringBuffer`, vous pouvez créer une nouvelle chaîne représentant le contenu de `StringBuffer`.

La classe `StringBuffer` a plusieurs constructeurs indiqués dans le tableau suivant.

Constructeur	Argument	Description
<code>StringBuffer</code>	<code>()</code>	Crée un tampon de chaînes vide pouvant contenir 16 caractères.
<code>StringBuffer</code>	<code>(int longueur)</code>	Crée un tampon de chaînes vide pouvant contenir le nombre de caractères spécifié par <code>longueur</code> .
<code>StringBuffer</code>	<code>(String chaîne)</code>	Crée un tampon de chaînes contenant une copie de <code>String chaîne</code> .

Plusieurs méthodes importantes différencient la classe `StringBuffer` et la classe `String`, dont : `capacity()`, `setLength()`, `setCharAt()`, `append()`, `insert()` et `toString()`. Les méthodes `append()` et `insert()` sont surchargées pour accepter divers types de données.

Méthode	Argument	Description
<code>setLength</code>	<code>(int nouvelleLongueur)</code>	Définit la longueur du <code>Stringbuffer</code> .
<code>capacity</code>	<code>()</code>	Renvoie la quantité de mémoire allouée au <code>StringBuffer</code> .

Méthode	Argument	Description
<code>setCharAt</code>	<code>(int indice, char car)</code>	Définit par <code>car</code> le caractère dont l'indice dans le <code>StringBuffer</code> est spécifié.
<code>append</code>	<code>(char c)</code>	Ajoute au <code>StringBuffer</code> la représentation sous forme de chaîne du type de données de l'argument. Cette méthode est surchargée pour accepter divers types de données.
<code>insert</code>	<code>(int offset, char c)</code>	Insère dans ce <code>StringBuffer</code> la représentation sous forme de chaîne du type de données de l'argument. Cette méthode est surchargée pour accepter divers types de données.
<code>toString</code>	<code>()</code>	Convertit le <code>StringBuffer</code> en <code>String</code> .

La méthode `capacity()`, qui renvoie la quantité de mémoire allouée au `StringBuffer`, peut renvoyer une valeur plus grande que la méthode `length()`. La mémoire allouée à un `StringBuffer` peut être définie avec le constructeur `StringBuffer(int longueur)`.

Le code suivant illustre quelques-unes des méthodes associées à la classe `StringBuffer`.

```
StringBuffer s1 = new StringBuffer(10);

int c = s1.capacity();      //c = 10
int lon = s1.length();      //lon = 0

s1.append("Bor");           //s1 = "Bor"
s1.append("land");          //s1 = "Borland"

c = s1.capacity();          //c = 10
lon = s1.length();          //lon = 7

s1.setLength(2);            //s1 = "Bo"

StringBuffer s2 = new StringBuffer("Helo World");
s2.insert(3, "l");           //s2 = "Hello World"
```

Voir aussi `java.lang.StringBuffer` dans la documentation de l'API du JDK

La classe `System` : `java.lang.System`

La classe `System` du paquet `java.lang` contient plusieurs champs et méthodes utiles pour accéder aux ressources et aux informations système indépendantes de la plate-forme, copier des tableaux, charger des fichiers et des bibliothèques, lire et écrire des propriétés. Par exemple, la méthode `currentTimeMillis()` fournit l'accès à l'heure système. Il est également possible d'extraire et de modifier les ressources du système avec les méthodes `getProperty` et `setProperty`. La classe `System` contient aussi la méthode `gc()` qui demande au ramasse-miettes d'effectuer son ramassage (garbage collection) ; et enfin, la classe `System` permet aux développeurs de

charger des bibliothèques de liens dynamiques avec la méthode `loadLibrary()`.

La classe `System` est déclarée en tant que classe `final` et ne peut pas être sous-classée. Elle déclare `static` ses méthodes et ses variables. Cela leur permet d'être disponible, sans que la classe soit instanciée.

La classe `System` déclare aussi plusieurs variables qui sont utilisées pour interagir avec le système. Il s'agit des variables `in`, `out` et `err`. La variable `in` représente le flux d'entrée standard du système, alors que la variable `out` représente le flux de sortie standard. La variable `err` est le flux d'erreur standard. Les flux sont décrits en détail dans la section consacrée au paquet des E/S.

Méthode	Argument	Description
<code>arrayCopy</code>	<code>arraycopy(Object src, int src_position, Object dst, int dst_position, int longueur)</code>	Copie le tableau source spécifié, à partir de la position spécifiée, et à la position spécifiée du tableau de destination.
<code>currentTimeMillis</code>	<code>()</code>	Renvoie l'heure courante en millisecondes.
<code>loadLibrary</code>	<code>(String nombiblio)</code>	Charge la bibliothèque système spécifiée par l'argument.
<code>getProperty</code>	<code>(String clé)</code>	Obtient la propriété système indiquée par clé.
<code>gc</code>	<code>()</code>	Exécute le ramasse-miettes qui supprime les objets qui ne sont plus utilisés.
<code>load</code>	<code>(String nomfichier)</code>	Charge un fichier de code du système de fichiers local en tant que bibliothèque dynamique.
<code>exit</code>	<code>(int état)</code>	Quitte le programme en cours.
<code>setProperty</code>	<code>(String clé, String valeur)</code>	Définit la propriété système indiquée par clé.

Voir aussi `java.lang.System` dans la documentation de l'API du JDK

Principales classes `java.util`

L'interface `Enumeration` : `java.util.Enumeration`

L'interface `Enumeration` du paquet `java.util` sert à implémenter une classe pouvant énumérer des valeurs. Une classe qui implémente l'interface `Enumeration` facilite l'investigation de structures de données.

Avec les méthodes définies dans l'interface `Enumeration`, l'objet `Enumeration` peut extraire en continu tous les éléments d'un ensemble de valeurs, un par un. Il y a seulement deux méthodes déclarées dans l'interface `Enumeration`, `hasMoreElements()` et `nextElement()`.

La méthode `hasMoreElements()` renvoie `True` s'il reste des éléments dans la structure de données. La méthode `nextElement()` renvoie la prochaine valeur de la structure de données en cours d'énumération.

L'exemple suivant crée une classe appelée `CanEnumerate`, qui implémente l'interface `Enumeration`. Une instance de cette classe est utilisée pour imprimer tous les éléments de l'objet `Vector`, `v`.

```
CanEnumerate enum = v.elements();

while (enum.hasMoreElements()) {
    System.out.println(enum.nextElement());
}
```

Un objet `Enumeration` est limité en ce qu'il ne peut être utilisé qu'une seule fois. L'interface ne dispose pas de méthode permettant à l'objet `Enumeration` de revenir sur les éléments précédents. Ainsi, une fois l'ensemble des valeurs entièrement énuméré, l'objet est consommé.

Voir aussi `java.lang.Enumeration` dans la documentation de l'API du JDK

La classe `Vector` : `java.util.Vector`

Java n'inclut pas le support de toutes les structures de données dynamiques ; il définit seulement la classe `Stack`. Cependant, la classe `Vector` du paquet `java.util` permet d'implémenter facilement des structures de données dynamiques.

La classe `Vector` est efficace, car elle affecte plus de mémoire que nécessaire pendant l'ajout de nouveaux éléments. Par conséquent, la capacité d'un objet `Vector` est généralement supérieure à sa taille réelle. L'argument `incrémentCapacité` du quatrième constructeur, dans le tableau suivant, définit l'augmentation de la capacité du `Vector` chaque fois qu'un élément lui est ajouté.

Constructeur	Argument	Description
<code>Vector</code>	<code>()</code>	Construit un vecteur vide de taille de tableau 10 et dont l'incrément de capacité est zéro.
<code>Vector</code>	<code>(Collection c)</code>	Construit un vecteur contenant les éléments de la collection, dans l'ordre où ils ont été renvoyés par l'itérateur de la collection

Constructeur	Argument	Description
Vector	(int capacitéInitiale)	Construit un vecteur vide ayant la capacité initiale spécifiée et un incrément de capacité nul.
Vector	(int capacitéInitiale, int incrémentCapacité)	Construit un vecteur vide ayant la capacité initiale et l'incrément de capacité spécifiés.

Le tableau suivant dresse la liste de quelques méthodes de la classe `Vector` parmi les plus importantes, avec les arguments qu'elles acceptent.

Méthode	Argument	Description
<code>setSize</code>	(int nouvelleTaille)	Définit la taille d'un vecteur.
<code>capacity</code>	()	Renvoie la capacité d'un vecteur.
<code>size</code>	()	Renvoie le nombre d'éléments stockés dans un vecteur.
<code>elements</code>	()	Renvoie l'énumération des éléments d'un vecteur.
<code>elementAt</code>	(int)	Renvoie l'élément à l'indice spécifié.
<code>firstElement</code>	()	Renvoie le premier élément d'un vecteur (indice 0).
<code>lastElement</code>	()	Renvoie le dernier élément d'un vecteur.
<code>removeElementAt</code>	(int indice)	Supprime l'élément à l'indice spécifié.
<code>addElement</code>	(Object obj)	Ajoute l'objet spécifié à la fin d'un vecteur, augmentant sa taille de un.
<code>toString</code>	()	Renvoie la représentation sous forme de chaîne de chaque élément d'un vecteur.

Le code suivant illustre l'utilisation de la classe `Vector`. Un objet `Vector` appelé `vector1` est créé et énumère ses éléments de trois façons : en utilisant la méthode `nextElement()` de `Enumeration`, en utilisant la méthode `elementAt()` de `Vector` et en utilisant la méthode `toString()` de `Vector`. Un composant AWT, `TextArea`, est créé pour afficher le résultat. La propriété `text` est définie avec la méthode `setText()`.

```
Vector vector1 = new Vector();

for (int i = 0; i < 10; i++) {
    vector1.addElement(new Integer(i)); //addElement accepte les types objet ou
}                                     //composites mais pas les types primitifs

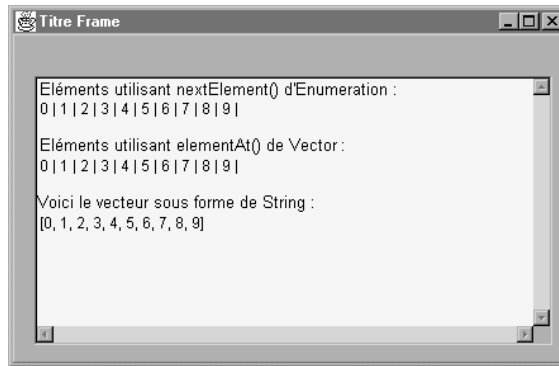
//énumérer vector1 en utilisant nextElement()
Enumeration e = vector1.elements();
textArea1.setText("Les éléments en utilisant nextElement() de Enumeration :\n");
while (e.hasMoreElements()) {
    textArea1.append(e.nextElement()+ " | ");
}
textArea1.append("\n\n");
```

```
//énumérer en utilisant la méthode elementAt()
textAreal.append("Les éléments en utilisant elementAt() de Vector :\n");
for (int i = 0; i < vector1.size();i++) {
    textAreal.append(vector1.elementAt(i) + " | ");
}
textAreal.append("\n\n");

//énumérer en utilisant la méthode toString()
textAreal.append("Voici le vecteur sous forme de chaîne :\n");
textAreal.append(vector1.toString());
```

La figure suivante montre l'action de ce code dans une application.

Figure 5.1 Exemple de Vector et Enumeration



Voir aussi `java.lang.Vector` dans la documentation de l'API du JDK

Principales classes java.io

Classes de flux d'entrée

Un flux d'entrée sert à lire les données depuis une source d'entrée, telle un fichier, une chaîne de caractères ou la mémoire. Les classes de flux d'entrée du paquet `java.io` sont par exemple `InputStream`, `BufferedInputStream`, `DataInputStream` et `FileInputStream`.

La méthode de base de lecture des données à l'aide d'une classe de flux d'entrée est toujours la même :

- 1 Créez une instance d'une classe de flux d'entrée.
- 2 Indiquez-lui où lire les données.

Remarque Les classes de flux d'entrée lisent les données sous forme d'un flux d'octets continu. S'il ne reste plus de données disponibles, la classe de flux d'entrée se bloque (attend que de nouvelles données soient disponibles).

Outre les classes de flux d'entrée, le paquet `java.io` fournit des classes de lecture (sauf pour `DataInputStream`). Les classes de lecture sont par exemple `Reader`, `BufferedReader`, `FileReader` et `StringReader`. Les classes de lecture sont identiques aux classes de flux d'entrée, sauf qu'elles lisent des caractères Unicode et non des octets.

Classe `InputStream` : `java.io.InputStream`

La classe `InputStream` du paquet `java.io` est une classe abstraite et la superclasse de toutes les autres classes de flux d'entrée. Elle fournit l'interface de base pour la lecture des flux d'octets. Le tableau suivant dresse la liste de quelques méthodes définies dans la classe `InputStream` avec les arguments qu'elles acceptent. Chacune de ces méthodes renvoie une valeur de type `int`, sauf la méthode `close()`.

Méthode	Argument	Description
<code>read</code>	<code>()</code>	Lit l'octet suivant dans le flux d'entrée et le renvoie sous forme d'un entier. Quand elle atteint la fin du flux, elle renvoie -1.
<code>read</code>	<code>(byte b[])</code>	Lit plusieurs octets et les place dans le tableau <code>b</code> . Elle renvoie le nombre d'octets lus, ou -1 quand la fin du flux est atteinte.
<code>read</code>	<code>(byte b[], int off, int lon)</code>	Lit <code>lon</code> octets de données à partir de l'emplacement <code>off</code> du flux d'entrée et les met dans un tableau.
<code>available</code>	<code>()</code>	Renvoie le nombre d'octets qui peuvent être lus dans un flux d'entrée sans blocage par le prochain appelant d'une méthode pour ce flux d'entrée.
<code>skip</code>	<code>(long n)</code>	Saute et ignore <code>n</code> octets de données d'un flux d'entrée.
<code>close</code>	<code>()</code>	Ferme le flux d'entrée et libère les ressources système utilisées par ce flux.

Voir aussi `java.lang.InputStream` dans la documentation de l'API du JDK

Classe `FileInputStream` : `java.io.FileInputStream`

La classe `FileInputStream` du paquet `java.io` ressemble beaucoup à la classe `InputStream`, sauf qu'elle a été spécialement conçue pour lire des fichiers. Elle contient trois constructeurs : `FileInputStream(String nomfichier)`, `FileInputStream(File objetfichier)` et `FileInputStream(FileDescriptor objDf)`. Le premier constructeur prend comme paramètre le nom du fichier, alors que le deuxième prend simplement un objet fichier. Le troisième

constructeur prend un objet descripteur de fichier. Les “Classes de fichiers” sont traitées [page 5-25](#).

Constructeur	Argument	Description
<code>FileInputStream</code>	<code>(String nomfichier)</code>	Crée un <code>FileInputStream</code> en ouvrant une connexion vers le fichier nommé par le nom de chemin <code>nomfichier</code> dans le système de fichiers.
<code>FileInputStream</code>	<code>(File objetfichier)</code>	Crée un <code>FileInputStream</code> en ouvrant une connexion vers le fichier nommé par le fichier <code>objetfichier</code> dans le système de fichiers.
<code>FileInputStream</code>	<code>(FileDescriptor objDf)</code>	Crée un <code>FileInputStream</code> en utilisant le descripteur de fichier <code>objDf</code> , qui représente une connexion existante vers un fichier réel du système de fichiers.

L'exemple suivant illustre une utilisation de la classe `FileInputStream`.

```
import java.io.*;

class FileReader {
    public static void main(String args[]) {
        byte buff[] = new byte[80];
        try {
            InputStream fileIn = new FileInputStream("Readme.txt");
            int i = fileIn.read(buff);
            String s = new String(buff);
            System.out.println(s);
        }
        catch(FileNotFoundException e) {
        }
        catch(IOException e) {
        }
    }
}
```

Dans cet exemple, un tableau de caractères est créé pour recevoir les données en entrée. Ensuite, un objet `FileInputStream` est instancié et transmet le nom du fichier en entrée à son constructeur. Ensuite, la méthode `read()` de `FileInputStream` lit un flux de caractères qu'elle met dans le tableau `buff`. Les 80 premiers octets du fichier `Readme.txt` sont lus et mis dans le tableau `buff`.

Remarque La classe `FileReader` pourrait être utilisée à la place de la méthode `FileInputStream()`. Les seules modifications nécessaires seraient le remplacement d'un tableau de type `byte` par un tableau de type `char` et l'objet `reader` serait instancié de la manière suivante :

```
Reader fileIn = new FileReader("Readme.txt");
```

Enfin, pour voir le résultat de l'appel de `read`, un objet `String` est créé en utilisant le tableau `buff`, puis il est transmis à la méthode `System.out.println`.

Comme cela a déjà été dit, la classe `System` est définie dans `java.lang` et donne accès aux ressources du système. `System.out` est un membre statique (static) de la classe `System` et représente le périphérique de sortie standard. La méthode `println()` est appelée pour envoyer la sortie sur le périphérique de sortie standard. L'objet `System.out` est du type `PrintStream`, présenté dans ["Classes de flux de sortie"](#), page 5-22.

L'objet `System.in`, un autre membre statique de la classe `System`, est du type `InputStream` et représente le périphérique d'entrée standard.

Voir aussi `java.io.FileInputStream` dans la documentation de l'API du JDK

Classes de flux de sortie

Les classes de flux de sortie sont la contrepartie des classes de flux d'entrée. Elles permettent d'envoyer des flux de données en sortie sur diverses unités. Les classes de flux de sortie principales de Java, situées dans le paquet `java.io`, sont `OutputStream`, `PrintStream`, `BufferedOutputStream`, `DataOutputStream` et `FileOutputStream`.

Classe `OutputStream` : `java.io.OutputStream`

Pour envoyer un flux de données en sortie, il faut créer un objet `OutputStream` et diriger les données en sortie vers une unité particulière. Comme prévu, il existe une classe d'écriture correspondant à chaque classe, sauf pour la classe `DataOutputStream`. La classe `OutputStream` définit les méthodes suivantes.

Méthode	Argument	Description
<code>write</code>	<code>(int b)</code>	Ecrit <code>b</code> dans un flux de sortie.
<code>write</code>	<code>(byte b[])</code>	Ecrit le tableau <code>b</code> dans un flux de sortie.
<code>write</code>	<code>(byte b[], int off, int lon)</code>	Ecrit dans le flux de sortie <code>lon</code> octets issus du tableau d'octets en commençant à l'emplacement <code>off</code> .
<code>flush</code>	<code>()</code>	Vide le flux de sortie et force la sortie de toutes les données stockées dans un tampon.
<code>close</code>	<code>()</code>	Ferme le flux de sortie et libère toutes les ressources système qui lui étaient associées.

Voir aussi `java.io.OutputStream` dans la documentation de l'API du JDK

Classe `PrintStream` : `java.io.PrintStream`

La classe `PrintStream` du paquet `java.io`, principalement destinée à la sortie des données sous forme de texte, a deux constructeurs. Le premier constructeur vide les données de la mémoire tampon sur la base de conditions spécifiées, alors que le deuxième vide les données quand il trouve un caractère de nouvelle ligne (si `autoflush` a la valeur `true`).

Constructeur	Argument	Description
<code>PrintStream</code>	<code>(OutputStream out)</code>	Crée un nouveau flux d'impression.
<code>PrintStream</code>	<code>(OutputStream out, boolean autoflush)</code>	Crée un nouveau flux d'impression.

Plusieurs méthodes définies dans la classe `PrintStream` figurent dans le tableau suivant.

Méthode	Argument	Description
<code>checkError</code>	<code>()</code>	Vide le flux et renvoie la valeur <code>false</code> si une erreur est détectée.
<code>print</code>	<code>(Object obj)</code>	Imprime un objet.
<code>print</code>	<code>(String s)</code>	Imprime une chaîne de caractère.
<code>println</code>	<code>()</code>	Imprime et termine la ligne par la chaîne séparateur de ligne définie par la propriété système <code>line.separator</code> , qui n'est pas nécessairement un simple caractère nouvelle ligne (<code>'\n'</code>).
<code>println</code>	<code>(Object obj)</code>	Imprime un objet et termine la ligne. Cette méthode se comporte comme si on appelait <code>print(Object)</code> et puis <code>println()</code> .

Les méthodes `print()` et `println()` sont surchargées pour recevoir différents types de données.

Voir aussi `java.io.PrintStream` dans la documentation de l'API du JDK

Classe `BufferedOutputStream` : `java.io.BufferedOutputStream`

La classe `BufferedOutputStream` du paquet `java.io` implémente un flux de sortie en tampon et accroît l'efficacité des sorties, en stockant les valeurs dans un tampon et en les écrivant uniquement lorsque ce dernier est plein ou lorsque la méthode `flush()` est appelée.

Constructeur	Argument	Description
<code>BufferedOutputStream</code>	<code>(OutputStream out)</code>	Crée un nouveau flux de sortie dans un tampon de 512 octets pour écrire les données dans le flux de sortie.
<code>BufferedOutputStream</code>	<code>(OutputStream out, int taille)</code>	Crée un nouveau flux de sortie en tampon pour écrire les données dans le flux de sortie, en spécifiant la taille du tampon.

`BufferedOutputStream` a trois méthodes pour vider le flux de sortie et y écrire.

Méthode	Argument	Description
<code>flush</code>	<code>()</code>	Vide le flux de sortie en tampon.
<code>write</code>	<code>(byte[] b, int off, int lon)</code>	Ecrit dans le flux de sortie en tampon <code>lon</code> octets issus du tableau d'octets en commençant à l'emplacement <code>off</code> .
<code>write</code>	<code>(int b)</code>	Ecrit l'octet dans le flux de sortie en tampon.

Voir aussi `java.io.BufferedOutputStream` dans la documentation de l'API du JDK

Classe `DataOutputStream` : `java.io.DataOutputStream`

Un flux de sortie de données permet à une application d'écrire dans un flux de sortie des types de données Java primitifs sous un format binaire portable. Une application peut utiliser ensuite un flux d'entrée de données pour lire les données en retour.

La classe `DataOutputStream` du paquet `java.io` a un seul constructeur, `DataOutputStream(OutputStream out)`, qui crée un nouveau flux de sortie de données utilisé pour écrire des données dans un flux de sortie.

La classe `DataOutputStream` utilise diverses méthodes `write()` pour sortir des types de données primitifs, ainsi qu'une méthode `flush()` et une méthode `size()`.

Méthode	Argument	Description
<code>flush</code>	<code>()</code>	Vide le flux de sortie de données.
<code>size</code>	<code>()</code>	Renvoie le nombre d'octets écrits dans le flux de sortie de données.
<code>write</code>	<code>(int b)</code>	Ecrit l'octet dans le flux de sortie.
<code>writeType</code>	<code>(type v)</code>	Ecrit dans le flux de sortie le type primitif spécifié sous forme d'octets.

Voir aussi `java.io.DataOutputStream` dans la documentation de l'API du JDK

Classe `FileOutputStream` : `java.io.FileOutputStream`

Un flux de sortie de fichier est un flux de sortie permettant d'écrire des données dans un fichier ou dans un `FileDescriptor`. Qu'un fichier soit disponible ou puisse être créé dépend de la plate-forme sous-jacente. Certaines plates-formes autorisent un fichier à être ouvert en écriture par un seul `FileOutputStream` à la fois. Dans de telles situations les constructeurs de cette classe échouent si le fichier est déjà ouvert. La classe `FileOutputStream` du paquet `java.io`, sous-classe de `OutputStream`, a plusieurs constructeurs.

Constructeur	Argument	Description
<code>FileOutputStream</code>	<code>(File fichier)</code>	Crée un flux de sortie de fichier pour écrire dans le fichier spécifié.
<code>FileOutputStream</code>	<code>(FileDescriptor objDf)</code>	Crée un flux de sortie de fichier pour écrire dans le descripteur de fichier, qui représente une connexion existante vers un fichier réel du système de fichiers.
<code>FileOutputStream</code>	<code>(String nom)</code>	Crée un flux de sortie de fichier pour écrire dans le fichier dont le nom est spécifié.
<code>FileOutputStream</code>	<code>(String nom, boolean append)</code>	Crée un flux de sortie de fichier pour écrire dans le fichier dont le nom est spécifié.

`FileOutputStream` a plusieurs méthodes, dont `close()`, `finalize()`, et plusieurs méthodes `write()`.

Méthode	Argument	Description
<code>close</code>	<code>()</code>	Ferme le flux de sortie de fichier et libère toutes les ressources système qui lui étaient associées.
<code>finalize</code>	<code>()</code>	Nettoie la connexion au fichier et appelle la méthode <code>close()</code> quand il n'y a plus de référence au flux.
<code>getFD</code>	<code>()</code>	Renvoie le descripteur de fichier associé au flux.
<code>write</code>	<code>(byte[] b, int off, int lon)</code>	Écrit dans le flux de sortie de fichier <code>lon</code> octets issus du tableau d'octets en commençant à l'emplacement <code>off</code> .

Voir aussi `java.io.FileOutputStream` dans la documentation de l'API du JDK

Classes de fichiers

Les classes `FileInputStream` et `FileOutputStream` du paquet `java.io` fournissent uniquement les fonctions de base de la gestion des entrées et sorties de fichiers. Le paquet `java.io` fournit la classe `File` et la classe `RandomAccessFile` pour une prise en charge évoluée des fichiers. La classe `File` fournit un accès facile aux attributs et fonctions des fichiers, alors que la classe `RandomAccessFile` fournit diverses méthodes pour lire et écrire dans un fichier.

Classe `File` : `java.io.File`

La classe `File` de Java utilise une représentation abstraite, indépendante de la plate-forme, des noms de chemins de fichiers et de répertoires. La classe `File` a trois constructeurs indiqués dans le tableau suivant.

Constructeur	Argument	Description
<code>File</code>	<code>(String chemin)</code>	Crée une nouvelle instance de <code>File</code> en convertissant la chaîne de nom de chemin donnée en un nom de chemin abstrait.
<code>File</code>	<code>(String parent, String enfant)</code>	Crée une nouvelle instance de <code>File</code> à partir d'une chaîne de nom de chemin parent et d'une chaîne de nom de chemin enfant.
<code>File</code>	<code>(File parent, String enfant)</code>	Crée une nouvelle instance de <code>File</code> à partir d'un nom de chemin abstrait parent et d'une chaîne de nom de chemin enfant.

La classe `File` implémente également de nombreuses méthodes importantes qui contrôlent l'existence, la possibilité de lecture, la possibilité d'écriture, le type, la taille et le moment de la modification des fichiers et des répertoires, ainsi que la création de nouveaux répertoires et le changement de nom ou la suppression des fichiers et des répertoires.

Méthode	Argument	Renvoie	Description
<code>delete</code>	<code>()</code>	<code>boolean</code>	Supprime les fichiers ou les répertoires.
<code>canRead</code>	<code>()</code>	<code>boolean</code>	Teste si l'application peut lire le fichier désigné par le nom de chemin abstrait.
<code>canWrite</code>	<code>()</code>	<code>boolean</code>	Teste si l'application peut écrire dans le fichier.
<code>renameTo</code>	<code>(File dest)</code>	<code>boolean</code>	Renomme le fichier.
<code>getName</code>	<code>()</code>	<code>String</code>	Renvoie la chaîne du nom du fichier ou du répertoire.
<code>getParent</code>	<code>()</code>	<code>String</code>	Renvoie la chaîne du nom de chemin du répertoire parent du fichier ou du répertoire.
<code>getPath</code>	<code>()</code>	<code>String</code>	Convertit le nom de chemin abstrait en une chaîne de nom de chemin.

Voir aussi [java.io.File](#) dans la documentation de l'API du JDK

Classe `RandomAccessFile` : `java.io.RandomAccessFile`

La classe `RandomAccessFile` du paquet `java.io` est plus puissante que les classes `FileInputStream` et `FileOutputStream`, qui fournissent seulement l'accès séquentiel à un fichier. La classe `RandomAccessFile` vous permet de lire et d'écrire arbitrairement des octets, du texte et des types de données Java à n'importe quel emplacement spécifié dans un fichier. Elle a deux constructeurs : `RandomAccessFile(String nom, String mode)` et

`RandomAccessFile(File fichier, String mode)`. Le paramètre `mode` indique si l'objet `RandomAccessFile` est utilisé en lecture ("`r`") ou en lecture/écriture ("`rw`").

Constructeur	Argument	Description
<code>RandomAccessFile</code>	<code>(String nom, String mode)</code>	Crée un flux de fichier d'accès direct pour lire, et de façon facultative écrire, dans un fichier dont le nom est spécifié.
<code>RandomAccessFile</code>	<code>(File fichier, String mode)</code>	Crée un flux de fichier d'accès direct pour lire, et de façon facultative écrire, dans le fichier spécifié par le paramètre <code>fichier</code> .

Il existe de nombreuses méthodes puissantes implémentées par `RandomAccessFile` comme indiqué dans le tableau suivant.

Méthode	Argument	Description
<code>seek</code>	<code>(long pos)</code>	Définit le décalage du pointeur de fichier, mesuré depuis le début de ce fichier, auquel va se produire la prochaine lecture ou écriture.
<code>read</code>	<code>()</code>	Lit le prochain octet de données dans le flux d'entrée.
<code>read</code>	<code>(byte b[], int off, int lon)</code>	Lit <code>lon</code> octets de données à partir de l'emplacement <code>off</code> du flux d'entrée et les met dans un tableau.
<code>readType</code>	<code>()</code>	Lit dans un fichier le type de données spécifié, par exemple <code>readChar</code> , <code>readByte</code> , <code>readLong</code> .
<code>write</code>	<code>(int b)</code>	Ecrit dans un fichier l'octet spécifié.
<code>write</code>	<code>(byte b[], int off, int lon)</code>	Ecrit dans le flux de sortie <code>lon</code> octets issus du tableau d'octets en commençant à l'emplacement <code>off</code> .
<code>length</code>	<code>()</code>	Renvoie la longueur du fichier.
<code>close</code>	<code>()</code>	Ferme le fichier et libère toutes les ressources système qui lui étaient associées.

Voir aussi `java.io.RandomAccessFile` dans la documentation de l'API du JDK

La classe `StreamTokenizer` : `java.io.StreamTokenizer`

La classe `StreamTokenizer` du paquet `java.io` est utilisée pour lire un flux d'entrée et le décomposer ou l'analyser en jetons individuels qui peuvent être traités un par un. Les jetons sont des groupes de caractères qui représentent un nombre ou un mot. La décomposition des flux en jetons permet de reconnaître les chaînes, les nombres, les identificateurs et les

commentaires. Cette technique de traitement des flux en jetons est probablement la plus utilisée dans l'écriture des analyseurs, compilateurs ou programmes qui traitent l'entrée des caractères.

Cette classe a un constructeur, `StreamTokenizer(Reader r)`, et définit les quatre constantes suivantes.

Constante	Description
<code>TT_EOF</code>	Indique que la fin du fichier a été lue.
<code>TT_EOL</code>	Indique que la fin de la ligne a été lue.
<code>TT_NUMBER</code>	Indique qu'un jeton nombre a été lu.
<code>TT_WORD</code>	Indique qu'un jeton mot a été lu.

La classe `StreamTokenizer` utilise les variables d'instance `nval`, `sval` et `ttype` pour stocker respectivement la valeur numérique, la valeur de chaîne et le type du jeton.

La classe `StreamTokenizer` implémente plusieurs méthodes utilisées pour définir la syntaxe lexicale des jetons.

Méthode	Argument	Description
<code>nextToken</code>	<code>()</code>	Analyse le prochain jeton dans le flux d'entrée. Renvoie <code>TT_NUMBER</code> si le prochain jeton est un nombre, <code>TT_WORD</code> si c'est un mot ou un caractère.
<code>parseNumbers</code>	<code>()</code>	Analyse les nombres.
<code>lineno</code>	<code>()</code>	Renvoie le numéro de ligne en cours.
<code>pushBack</code>	<code>()</code>	Renvoie la valeur en cours du champ <code>ttype</code> au prochain appel de la méthode <code>nextToken()</code> .
<code>toString</code>	<code>()</code>	Renvoie l'équivalent sous forme de chaîne du jeton en cours.

Suivez ces étapes lorsque vous utilisez la décomposition des flux en jetons :

- 1 Créez un objet `StreamTokenizer` pour un `Reader`.
- 2 Définissez la façon de traiter les caractères.
- 3 Utilisez la méthode `nextToken()` pour obtenir le prochain jeton.
- 4 Lisez la variable d'instance `ttype` pour connaître le type du jeton.
- 5 Lisez la valeur du jeton dans la variable d'instance.
- 6 Traitez le jeton.
- 7 Répétez les étapes 3 à 6 jusqu'à ce que `nextToken()` renvoie `StreamTokenizer.TT_EOF`.

Voir aussi `java.io.StreamTokenizer` dans la documentation de l'API du JDK

Programmation orientée objet dans Java

La programmation orientée objet existe depuis l'arrivée du langage Simula '67 en 1967. Cependant, elle n'est vraiment devenue un des paradigmes de la programmation qu'au milieu des années 1980.

Au contraire de la programmation structurée traditionnelle, la programmation orientée objet met dans une même et unique structure les données et les opérations qui leurs sont associées. En programmation traditionnelle, les données et les opérations sur les données sont séparées, les structures de données sont donc envoyées aux procédures et fonctions qui les utilisent. La programmation orientée objet résout de nombreux problèmes inhérents à cette conception en mettant dans une même entité les attributs et les opérations. Cela est plus proche du monde réel, dans lequel tous les objets disposent d'attributs auxquels sont associés des activités.

Java est un *pur* langage orienté objet, ce qui signifie que le niveau le plus externe de la structure des données est l'*objet*. Il n'y a pas de constante, de variable ni de fonction indépendante en Java. On accède à toute chose via les classes et les objets. C'est un des aspects les plus agréables de Java. D'autres langages orientés objet plus hybrides ont conservé des aspects des langages structurés en plus de leurs extensions objet. Par exemple, C++ et Pascal Objet sont des langages orientés objet, mais permettent toujours d'écrire des programmes structurés, ce qui diminue l'efficacité des extensions orientées objet. Vous ne pouvez tout simplement pas faire cela en Java !

Ce chapitre suppose que vous ayez une certaine connaissance de la programmation avec d'autres langages orientés objet. Si ce n'est pas le cas, vous devrez rechercher ailleurs une explication détaillée de la

programmation orientée objet. Ce chapitre a pour objectif de souligner et de résumer les fonctionnalités orientées objet de Java.

Classes

Les classes et les objets ne sont pas la même chose. Une classe est la définition d'un type, alors qu'un objet est la déclaration d'une instance d'un type de classe. Après avoir créé une classe, vous pouvez créer autant d'objets que voulu basés sur cette classe. Entre les classes et les objets, il y a la même relation qu'entre une recette de clafoutis et le clafoutis lui-même ; à partir d'une même recette de clafoutis, vous pouvez faire autant de clafoutis que vous voulez.

Le processus de création d'un objet à partir d'une classe est appelé *instanciation* d'un objet ou création d'une *instance* d'une classe.

Déclaration et instanciation des classes

Une classe Java peut être très simple. Voici la définition d'une classe vide :

```
class MaClasse {  
}
```

Alors que cette classe n'est pas encore utile, elle est correcte dans Java. Une classe plus utile contiendra quelques données membre et méthodes, que nous allons ajouter sous peu. Premièrement, examinez la syntaxe d'instanciation de la classe. Pour créer une instance de cette classe, utilisez l'opérateur `new` avec la nom de la classe. Vous devez déclarer une variable d'instance pour l'objet :

```
MaClasse monObjet;
```

Mais, déclarer simplement une variable d'instance n'alloue pas de mémoire ni aucune autre des ressources nécessaires à l'objet. Cela crée une référence appelée `monObjet`, mais n'instancie pas l'objet. C'est le rôle de l'opérateur `new`.

```
monObjet = new MaClasse();
```

Remarquez que le nom de la classe est utilisé comme s'il s'agissait d'une méthode. Il ne s'agit pas d'une coïncidence, comme vous le verrez dans une section ultérieure. Après l'exécution de cette ligne de code, il est possible d'accéder aux variables et aux méthodes membre de la classe, qui n'existent pas encore, avec l'opérateur `."`.

Une fois l'objet créé, vous n'avez pas à vous préoccuper de sa destruction. Les objets en Java sont automatiquement éliminés par le ramasse-miettes (garbage collector), autrement dit, lorsqu'une référence à l'objet n'est plus utilisée, la machine virtuelle désalloue automatiquement toutes les ressources allouées par l'opérateur `new`.

Données membre

Comme nous l'avons dit, une classe Java peut contenir des données membre et des méthodes. Une donnée membre ou une variable membre est une variable déclarée dans la classe. Une méthode est une fonction ou une routine effectuant une certaine tâche. Voici une classe qui ne contient que des données membre :

```
public class ClasseChien {  
    String nom, couleurYeux;  
    int age;  
    boolean hasQueue;  
}
```

Cet exemple crée une classe appelée `ClasseChien` qui contient les données membre : `nom`, `couleurYeux`, `age`, ainsi qu'un indicateur appelé `hasQueue`. Vous pouvez inclure n'importe quel type de données comme variable membre d'une classe. Pour accéder à une donnée membre, il faut d'abord créer une instance de la classe puis accéder aux données avec l'opérateur `"."`.

Méthodes de classe

Les classes peuvent contenir aussi des méthodes. En fait, il n'existe pas de fonction ni de procédure indépendante dans Java. Toutes les sous-routines sont définies comme méthodes de classes. Voici un exemple de la classe `ClasseChien` à laquelle est ajoutée la méthode `dit()` :

```
public class ClasseChien {  
    String nom, couleurYeux;  
    int age;  
    boolean hasQueue;  
  
    public void dit() {  
        JOptionPane.showMessageDialog(null, "Wouah ! Wouah !");  
    }  
}
```

Remarquez que, lors de la définition des méthodes, l'implémentation de la méthode figure juste sous la déclaration. Cela est différent de certains autres langages orientés objet dans lesquels la classe est définie à un emplacement et le code d'implémentation est situé ailleurs. Une méthode doit spécifier un type de retour et tous les paramètres qu'elle accepte. La méthode `dit()` ne prend pas de paramètre. Elle ne renvoie pas de valeur non plus, son type de retour est donc `void`.

L'accès à une méthode s'effectue de la même façon que l'accès aux variables membre, c'est-à-dire en utilisant l'opérateur `"."`. Par exemple,

```
ClasseChien chien = new ClasseChien();  
chien.age = 4;  
chien.dit();
```

Constructeurs et finaliseurs

Chaque classe Java possède une méthode spécialisée appelée un *constructeur*. Le constructeur a toujours le même nom que la classe et il ne peut spécifier de valeur de retour. Le constructeur affecte toutes les ressources dont l'objet a besoin et renvoie une instance de l'objet. Quand vous utilisez l'opérateur *new*, vous appelez en réalité le constructeur. Vous n'avez pas besoin de spécifier un type de retour pour le constructeur car l'instance de l'objet est toujours le type renvoyé.

Dans la plupart des langages orientés objet, il existe une méthode correspondante nommée *destructeur*, appelée pour libérer les ressources affectées par le constructeur. Mais, comme Java désalloue pour vous toutes les ressources automatiquement, il n'existe pas de mécanisme destructeur en Java.

Cependant, certaines situations nécessitent un nettoyage spécial que le ramasse-miettes ne peut pas effectuer lorsque la classe disparaît. Par exemple, certains fichiers ont été ouverts pendant la durée de vie de l'objet et vous voulez vérifier qu'ils sont correctement fermés quand l'objet est détruit. Pour cela, une autre méthode spéciale, appelée *finaliseur*, peut être définie. Cette méthode (si elle est présente) est appelée par le ramasse-miettes immédiatement avant la destruction de l'objet. Ainsi, si un nettoyage spécial quelconque doit avoir lieu, le finaliseur peut le faire pour vous. Cependant, le ramasse-miettes s'exécute dans la machine virtuelle sous forme de thread de faible priorité, ce qui ne permet pas de savoir à quel moment il détruira réellement votre objet. C'est pourquoi il vaut mieux éviter de mettre dans le finaliseur du code dépendant du temps, puisque vous ne pouvez pas savoir quand il sera appelé.

Etude de cas : Exemple simple d'OOP

Dans cette section, nous allons voir un exemple simple de définition de classes et d'instanciation d'objets. Nous allons développer une application qui crée deux objets (un chien et un homme) et montrer leurs attributs sur une fiche.

Si vous débutez dans JBuilder, abandonnez provisoirement ce chapitre et étudiez l'environnement de développement intégré de JBuilder avant de commencer cet exemple. Commencez par *Introduction à JBuilder*. Dans ce document, étudiez le chapitre "Environnement de JBuilder". Vous devez également vous intéresser à *Conception d'interfaces utilisateur avec JBuilder*. Pour que tout cela se mette en place, créez l'application exemple du chapitre "Tutoriel : Construction d'une application" de *Introduction à JBuilder*.

Vous pourrez reprendre ce chapitre lorsque vous serez à l'aise avec les tâches suivantes :

- Commencer une application en utilisant l'expert application de JBuilder.
- Sélectionner des composants dans la palette de composants et les placer dans le concepteur d'interface utilisateur.
- Définir les propriétés des composants en utilisant l'inspecteur.
- Passer de l'éditeur au concepteur d'interface utilisateur dans le volet contenu de JBuilder.
- Utiliser l'éditeur.

Si vous avez des questions concernant l'utilisation de l'environnement de développement de JBuilder, vous trouverez des informations supplémentaires dans les chapitres de *Construction d'applications avec JBuilder*.

Voici à quoi ressemble l'application exemple que vous construirez lorsqu'elle s'exécute :

Figure 6.1 Application exemple montrant deux objets instanciés



Suivez les étapes énumérées dans cette section afin de créer une interface utilisateur simple pour votre application exemple.

- 1 Commencez à créer l'application et à concevoir son interface utilisateur :
 - 1 Créez un nouveau projet. Choisissez Fichier | Nouveau projet pour démarrer l'expert projet.
 - 2 Entrez `oop1` dans le champ Nom du projet et cliquez sur Terminer. Un nouveau projet est ouvert.
 - 3 Choisissez Fichier | Nouveau et cliquez sur l'icône Application pour lancer l'expert application.

- 4 Acceptez le nom de classe par défaut. Le nom du paquet sera `oop1`.
 - 5 Cliquez sur Suivant et ensuite sur Terminer pour créer un fichier `Cadrel.java` et un fichier `Application1.java`.
 - 6 Cliquez sur l'onglet Conception dans le volet contenu afin d'afficher le concepteur d'interface utilisateur pour `Cadrel.java`.
 - 7 Sélectionnez `contentPane` dans le volet structure. Dans l'inspecteur, définissez la propriété `layout` de `contentPane` par `XYLayout` (si vous utilisez l'édition JBuilder Personnel, définissez `layout` par `null`). `XYLayout` et `null` ne sont pas souvent des dispositions appropriées à une application, mais tant que l'utilisation des dispositions ne vous est pas familière, vous pouvez vous en servir pour créer une interface utilisateur "vite fait, bien fait".
- 2 Placez les composants nécessaires dans le concepteur d'interface utilisateur, en utilisant la capture d'écran précédente comme référence :
 - 1 Sélectionnez l'onglet Swing de la palette des composants et cliquez sur le composant `JTextField`. (Lorsque vous positionnez votre curseur sur un composant, une bulle d'aide apparaît qui décrit le composant. Cliquez sur le composant dont le libellé indique `javax.swing.JTextField`.) Cliquez dans le concepteur d'interface utilisateur et maintenez enfoncée le bouton de la souris pendant que vous dessinez le composant sur l'écran. Répétez cette étape cinq fois jusqu'à ce que vous ayez deux groupes de trois `JTextField` sur votre fiche.
 - 2 Maintenez enfoncée la touche Maj pendant que vous cliquez sur chaque `JTextField` dans le concepteur d'interface utilisateur afin de les sélectionner tous. Sélectionnez la propriété `text` dans l'inspecteur et supprimez le texte qui s'y trouve. Cela supprimera tout le texte de tous les composants `JTextField`.
 - 3 Modifiez la valeur de la propriété `name` de chaque `JTextField`. Appelez le premier `txtfldNomChien`, le deuxième `txtfldCouleurYeuxChien`, le troisième `txtfldAgeChien`, le quatrième `txtfldNomHomme`, le cinquième `txtfldCouleurYeuxHomme` et le sixième `txtfldAgeHomme`.
 - 4 Dessinez six composants `JLabel` dans la fiche, chacun étant situé à côté d'un composant `JTextField`.
 - 5 Modifiez les valeurs de la propriété `text` de ces composants afin d'associer un libellé approprié au composant `JTextField` correspondant. Par exemple, la propriété `text` du `JLabel` situé en haut de la fiche sera `Nom`, celle du second composant `Couleur des yeux`, etc.
 - 6 Placez deux composants `JCheckBox` sur la fiche. Placez le premier sous le premier groupe de composants `JTextField` et le second sous le second groupe de composants `JTextField`.

- 7 Sélectionnez chaque composant `JCheckBox` de la fiche et modifiez la propriété `text` du premier en `Queue` et la propriété `text` du second en `Marié`.
- 8 Changez la valeur de la propriété `name` de la première case à cocher en `chkboxChien` et changez le nom de la seconde case à cocher en `chkboxHomme`.
- 9 Placez deux composants `JButton` sur la fiche, un à droite du groupe de composants situé en haut et un à droite du groupe de composants situé en bas.
- 10 Changez la propriété `text` du premier bouton en `Créer Chien` et changez la propriété `text` du second bouton en `Créer Homme`.

L'étape finale consiste à enregistrer le projet en choisissant `Fichier | Tout enregistrer`.

Vous êtes désormais prêt à programmer. D'abord, créez une nouvelle classe :

- 1 Choisissez `Fichier | Nouvelle classe` pour démarrer l'expert classe.
- 2 Conservez le nom du paquet, `oop1`, spécifiez `ClasseChien` dans le champ `Nom de classe`, ne modifiez pas la `Classe de base`.
- 3 Cochez uniquement les options `Publique` et `Créer un constructeur par défaut`, désactivez toutes les autres.
- 4 Cliquez sur `OK`.

L'expert classe crée pour vous le fichier `ClasseChien.java`. Modifiez le code qu'il a créé pour qu'il ressemble à ce qui suit :

```
package oop1;

public class ClasseChien {
    String nom, couleurYeux;
    int age;
    boolean hasQueue;

    public ClasseChien() {
        nom = "Snoopy";
        couleurYeux = "Marron";
        age = 2;
        hasQueue = true;
    }
}
```

Vous avez défini `ClasseChien` avec certaines variables membre. Il y a aussi un constructeur pour instancier les objets `ClasseChien`.

A l'aide de l'expert classe, créez un fichier `ClasseHomme.java` en suivant les étapes précédentes mais en spécifiant `ClasseHomme` comme Nom de classe. Modifiez le code résultant pour qu'il ressemble à ceci :

```
package oop1;

public class ClasseHomme {
    String nom, couleurYeux;
    int age;
    boolean isMarié;

    public ClasseHomme() {
        nom = "Steven";
        couleurYeux = "Bleu";
        age = 35;
        isMarié = true;
    }
}
```

Les deux classes sont très semblables. Vous tirerez avantage de cette ressemblance dans une prochaine section.

Cliquez sur l'onglet `Cadre1` en haut du volet contenu pour revenir à la classe `Cadre1`. Cliquez sur l'onglet `Source` en bas pour ouvrir l'éditeur. Déclarez deux variables d'instance comme références aux objets. Voici le source des déclarations des variables de `Cadre1`, en gras ; ajoutez à votre classe les lignes apparaissant en gras :

```
public class Cadre1 extends JFrame {
    // Crée une référence pour les objets chien et homme
    ClasseChien chien;
    ClasseHomme homme;

    JPanel contentPane;
    JPanel jPanel1 = new JPanel();
    . . .
```

Cliquez sur l'onglet `Conception` en bas du volet contenu pour revenir à l'interface utilisateur en cours de conception. Double-cliquez sur le bouton `Créer chien`. `JBuilder` crée le début d'un gestionnaire d'événement pour ce bouton et place votre curseur à l'intérieur du code du gestionnaire. Remplissez le code du gestionnaire d'événement de façon à instancier un objet chien et à remplir les champs texte de l'objet chien créé. Votre code doit ressembler à ceci :

```
void jButton1_actionPerformed(ActionEvent e) {
    chien = new ClasseChien();
    txtfldNomChien.setText(chien.nom);
    txtfldCouleurYeuxChien.setText(chien.couleurYeux);
    txtfldAgeChien.setText(Integer.toString(chien.age));
    chkboxChien.setSelected(true);
}
```

Comme le code le montre, nous appelons le constructeur de l'objet chien puis accédons à ses variables membre.

Cliquez sur l'onglet Conception pour revenir au concepteur d'interface utilisateur. Double-cliquez sur le bouton Créer homme. JBuilder crée un gestionnaire d'événement pour le bouton Créer homme. Remplissez le gestionnaire d'événement pour qu'il ressemble à ce qui suit :

```
void jButton2_actionPerformed(ActionEvent e) {
    homme = new ClasseHomme();
    txtfldNomHomme.setText(homme.nom);
    txtfldCouleurYeuxHomme.setText(homme.couleurYeux);
    txtfldAgeHomme.setText(Integer.toString(homme.age));
    chkboxHomme.setSelected(true);
}
```

Vous pouvez à présent compiler et exécuter votre application. Choisissez Projet | Construire le projet "oop1.jpx" pour le compiler. S'il n'y a pas d'erreur, choisissez Exécuter | Exécuter le projet.

Si tout se passe bien, la fiche apparaît sur votre écran. Lorsque vous cliquez sur le bouton Créer chien, un objet `chien` est créé et des valeurs décrivant ce chien apparaissent dans les champs appropriés. Lorsque vous cliquez sur le bouton Créer homme, un objet `homme` est créé et des valeurs décrivant cet homme apparaissent dans les champs appropriés.

Héritage de classe

Les objets chien et homme créés ont de nombreuses ressemblances. Un des avantages de la programmation orientée objet est la possibilité de gérer de telles similitudes, comme dans une hiérarchie. Cette possibilité s'appelle *héritage*. Quand une classe hérite d'une autre classe, la classe enfant hérite automatiquement de toutes les caractéristiques (variables membre) et du comportement (méthodes) de la classe parent. L'héritage est toujours additif ; il n'est pas possible d'hériter d'une classe et de recevoir moins que ce que possède la classe parent.

Dans Java, l'héritage est géré avec le mot clé `extends`. Quand une classe hérite d'une autre classe, la classe enfant étend la classe parent. Par exemple,

```
public class ClasseChien extends ClasseMammifere {
    . . .
}
```

Les éléments communs aux hommes et aux chiens sont communs à tous les mammifères, ce qui permet de créer une `ClasseMammifere` pour gérer ces similitudes. Nous pouvons ensuite supprimer les déclarations des éléments communs de `ClasseChien` et de `ClasseHomme`, les déclarer dans `ClasseMammifere` à la place, puis créer les sous-classes `ClasseChien` et `ClasseHomme` à partir de `ClasseMammifere`.

En utilisant l'expert classe, créez une `ClasseMammifère`. Modifiez le code résultant pour qu'il ressemble à ceci :

```
package oop1;

public class ClasseMammifère {
    String nom, couleurYeux;
    int age;

    public ClasseMammifère() {
        nom = "Le nom";
        couleurYeux = "Marron";
        age = 0;
    }
}
```

Remarquez que `ClasseMammifère` a les caractéristiques communes aux deux classes `ClasseChien` et `ClasseHomme`. Maintenant, écrivons `ClasseChien` et `ClasseHomme` d'une autre façon pour tirer parti de l'héritage.

Modifiez le code de `ClasseChien` pour qu'il ressemble à ceci :

```
package oop1;

public class ClasseChien extends ClasseMammifère {

    boolean hasQueue;

    public ClasseChien() {
        // super() implicite
        nom = "Snoopy";
        age = 2;
        hasQueue = true;
    }
}
```

Modifiez le code de `ClasseHomme` pour qu'il ressemble à ceci :

```
package oop1;

public class ClasseHomme extends ClasseMammifère{

    boolean isMarié;

    public ClasseHomme() {
        nom = "Steven";
        couleurYeux = "Bleu";
        age = 35;
        isMarié = true;
    }
}
```

Remarquez que `ClasseChien` n'attribue pas une valeur `couleurYeux` mais que `ClasseHomme` le fait. `ClasseChien` n'a pas besoin d'attribuer une valeur à `couleurYeux` ; en effet, le chien Snoopy a les yeux marrons et la `ClasseChien` hérite des yeux marrons de la `ClasseMammifère`, qui déclare une variable

couleurYeux et lui assigne la valeur "Marron". L'homme Steven, lui, a les yeux bleus ; il est donc nécessaire d'attribuer la valeur "Bleu" à la variable couleurYeux héritée de ClasseMammifère.

Essayez de compiler et d'exécuter votre projet une nouvelle fois. (Choisir Exécuter | Exécuter le projet compilera, puis exécutera votre application.) Vous verrez que l'interface utilisateur de votre programme ressemble exactement à ce qu'elle était auparavant mais que, désormais, les objets chien et homme héritent de toutes les variables membre communes de ClasseMammifère.

Dès que ClasseChien étend ClasseMammifère, ClasseChien dispose de toutes les variables membre et de toutes les méthodes de ClasseMammifère. En fait, même ClasseMammifère a hérité d'une autre classe. Dans Java, toutes les classes étendent la classe Object ; donc, si une classe est déclarée sans étendre une autre classe, elle étend implicitement la classe Object.

Dans Java, les classes ne peuvent hériter que d'une seule classe à la fois (*héritage unique*). Au contraire de Java, certains langages (comme C++) permettent qu'une classe hérite de plusieurs classes à la fois (*héritage multiple*). Une classe ne peut étendre qu'une classe à la fois. Bien qu'il n'y ait aucune restriction sur le nombre de fois où l'héritage peut être utilisé pour étendre la hiérarchie, il ne peut y avoir qu'une extension à la fois. L'héritage multiple est une fonctionnalité sympathique, mais elle donne des hiérarchies d'objets très complexes. Java met en place un mécanisme qui apporte un grand nombre de ces avantages sans entraîner autant de complexité, comme vous allez le voir.

La ClasseMammifère a un constructeur qui définit des valeurs par défaut pratiques et appropriées. Il serait intéressant que les sous-classes accèdent à ce constructeur.

Elles le peuvent en réalité. Il y a deux façons d'y arriver dans Java. Si vous n'appellez pas explicitement le constructeur de la classe parent, **Java appelle automatiquement le constructeur par défaut de la classe parent comme première ligne du constructeur de la classe enfant**. La seule façon d'éviter ce comportement consiste à appeler vous-même un des constructeurs de la classe parent comme première ligne du constructeur de la classe enfant. Les appels au constructeur sont toujours chaînés de cette façon et vous ne pouvez pas contourner ce mécanisme. C'est une fonctionnalité très sympathique du langage Java, puisque, dans les autres langages orientés objet, une erreur habituelle consiste à ne pas appeler le constructeur de la classe parent. Java le fait toujours pour vous si vous ne le faites pas. C'est la signification du commentaire de la première ligne du constructeur de ClasseChien (`// super() implicite`). Le constructeur de ClasseMammifère est appelé à cet emplacement automatiquement. Ce mécanisme repose sur l'existence d'un constructeur sans paramètre dans la superclasse (classe parent). Si le constructeur n'existe pas et si vous n'en appelez pas un autre sur la première ligne du constructeur enfant, la classe ne pourra pas se compiler.

Appel du constructeur du parent

Comme vous avez fréquemment besoin d'appeler explicitement le constructeur de la classe de niveau supérieur, le mot clé Java `super()` a été défini. `super()` appellera le constructeur de la classe parent doté des paramètres nécessaires.

Il est également possible d'avoir plus d'un constructeur par classe. Lorsque plusieurs méthodes portant le même nom existent dans la même classe, les méthodes sont dites *surchargées*. Il est fréquent qu'une classe ait plusieurs constructeurs.

Pour l'application exemple, le changement de hiérarchie est la seule différence entre les deux versions de l'exemple. L'instanciation des objets et la fiche principale ne sont absolument pas modifiées. Cependant, la conception de l'application est plus efficace, puisque, pour modifier une quelconque des caractéristiques des mammifères, il suffit de le faire dans la `ClasseMammifère` et de recompiler les classes enfant. Ces changements se reportent automatiquement sur les classes enfant.

Modificateurs d'accès

Il est important de comprendre à quel moment les membres (variables et méthodes) de la classe sont accessibles. Dans Java, plusieurs options permettent de personnaliser l'accessibilité aux membres.

En règle générale, il vaut mieux limiter autant que possible la portée des éléments d'un programme, y compris celle des membres des classes. Moins un élément est accessible, moins il risque d'être mal utilisé.

Dans Java, il y a quatre modificateurs d'accès différents pour les membres des classes : `private`, `protected`, `public` et `default` (ou l'absence de tout modificateur). Le fait que les classes d'un même paquet disposent d'accès différents par rapport aux classes situées en dehors du paquet ajoute un peu de complexité. Les deux tableaux suivants montrent l'accessibilité et l'héritage des classes et des variables membre depuis l'intérieur du même paquet et depuis l'extérieur du paquet (les paquets sont présentés plus loin).

Accès depuis l'intérieur du paquet d'une classe

Modificateur d'accès	Héritage	Accessible
Par défaut (pas de modificateur)	Oui	Oui
Public	Oui	Oui
Protected	Oui	Oui
Private	Non	Non

Ce tableau montre comment les autres membres du même paquet accèdent aux membres d'une classe et en héritent. Par exemple, un membre déclaré privé est inaccessible depuis les autres membres du même paquet et il est impossible d'en hériter. Les membres déclarés avec les autres modificateurs sont accessibles depuis les autres membres de ce paquet qui peuvent aussi en hériter. Les diverses parties de l'application exemple font toutes partie du paquet `oop1`, vous n'avez pas à vous soucier d'accéder aux classes d'un autre paquet.

Accès depuis l'extérieur d'un paquet

Les règles changent si vous accédez par du code situé en dehors du paquet de la classe :

Modificateur d'accès	Héritage	Accessible
Par défaut (pas de modificateur)	Non	Non
Public	Oui	Oui
Protected	Oui	Non
Private	Non	Non

Par exemple, ce tableau montre qu'il est possible d'hériter d'un membre de type protégé mais qu'il est impossible d'y accéder, pour des classes situées en dehors de son paquet.

Dans les deux tableaux précédents, il faut noter que les membres déclarés de type public sont disponibles à qui souhaite y accéder (les constructeurs sont toujours de type public) alors que les membres de type privé sont inaccessibles et qu'il est impossible d'en hériter en dehors de leur classe. Ainsi, vous devez déclarer privée toute variable ou méthode membre devant rester interne à la classe.

En programmation orientée objet, il est recommandé de cacher les informations à l'intérieur de la classe en rendant privées toutes les variables membre de la classe, et en y accédant au moyen de méthodes, de format spécifique, appelées méthodes d'accès.

Méthodes d'accès

Les méthodes d'accès (parfois appelées getter et setter) fournissent l'interface publique de la classe accessible depuis l'extérieur, tout en conservant au stockage des données de la classe son caractère privé. C'est une bonne idée puisque vous pourrez ensuite à tout moment modifier la représentation interne des données dans la classe sans toucher aux méthodes qui définissent réellement ces valeurs internes. Tant que vous ne modifiez pas l'interface publique d'accès à la classe, vous ne détruisez aucun code qui repose sur cette classe et ses méthodes publiques.

Dans Java, les méthodes d'accès sont fournies par paires : une pour obtenir la valeur interne (*get*) et une autre pour définir la valeur interne (*set*). Par convention, la méthode *Get* utilise le nom de la variable interne privée associé au préfixe "get" (obtient). La méthode *Set* fait de même avec "set" (définit). Une propriété en lecture seule possède uniquement une méthode *Get*. En général, les méthodes *Get* booléennes utilisent "is" (est) ou "has" (a) comme préfixe à la place de "get". Les méthodes d'accès permettent aussi de valider facilement les données attribuées à une variable membre particulière.

En voici un exemple. Pour notre *ClasseChien*, déclarez *private* toutes les variables membre internes et ajoutez des méthodes d'accès aux valeurs internes. *ClasseChien* crée simplement une nouvelle variable membre, *queue*.

```
package oop1;

public class ClasseChien extends ClasseMammifère{

    // méthodes d'accès aux propriétés
    // Queue
    public boolean hasQueue() {
        return queue;
    }

    public void setQueue( boolean valeur ) {
        queue= valeur;
    }

    public ClasseChien() {
        setNom("Snoopy");
        setAge(2);
        setQueue(true);
    }

    private boolean queue;
}
```

La variable *queue* a été déplacée vers le bas de la classe et elle est maintenant déclarée comme privée. L'emplacement de la définition est sans importance, mais il est habituel dans Java de mettre les membres privés de la classe en bas de la définition (après tout, vous ne pouvez pas y accéder depuis l'extérieur de la classe ; par conséquent, si vous lisez le code, vous êtes intéressé en premier par les aspects publics). *ClasseChien* a désormais des méthodes publiques pour lire et écrire la valeur de *queue*. Le getter est *hasQueue()* et le setter est *setQueue()*.

Suivez le même modèle et modifiez le code de *ClasseHomme* pour qu'il ressemble à ceci :

```
package oop1;

public class ClasseHomme extends ClasseMammifère {
```

```

public boolean isMarié() {
    return marié;
}

public void setMarié(boolean valeur) {
    marié= valeur;
}

public ClasseHomme() {
    setNom("Steven");
    setAge(35);
    setCouleurYeux("Bleu");
    setMarié(true);
}

private boolean marié;
}

```

Notez que les constructeurs pour ces deux classes utilisent à présent des méthodes d'accès pour définir la valeur des variables de `ClasseMammifère`. Mais la `ClasseMammifère` n'a pas encore de méthode d'accès pour définir ces valeurs, aussi devez-vous les ajouter à la `ClasseMammifère`.

Modifiez `ClasseMammifère` pour que son code ressemble à ceci :

```

public class ClasseMammifère {

    // méthodes d'accès aux propriétés
    // nom
    public String getNom() {
        return nom;
    }

    public void setNom(String valeur) {
        nom = valeur;
    }

    // couleurYeux
    public String getCouleurYeux() {
        return couleurYeux;
    }

    public void setCouleurYeux(String valeur) {
        couleurYeux = valeur;
    }

    // son
    public String getSon() {
        return son;
    }

    public void setSon( String valeur ) {
        son = valeur;
    }
}

```

```

// age
public int getAge() {
    return age;
}

public void setAge(int valeur) {
    if (valeur > 0) {
        age = valeur;
    }
    else
        age = 0;
}

public ClasseMammifère() {
    setNom( "Le nom" );
    setCouleurYeux("Marron");
    setAge(0);
}

private String nom, couleurYeux, son;
private int age;
}

```

Remarquez qu'une nouvelle variable membre, `son`, a été ajoutée à `ClasseMammifère`. Elle aussi a des méthodes d'accès. Comme `ClasseChien` et `ClasseHomme` étendent `ClasseMammifère`, elles ont aussi une propriété `son`.

Les gestionnaires d'événements de `Cadre1.java` doivent également utiliser les méthodes d'accès. Modifiez les gestionnaires d'événements pour qu'ils ressemblent à ceci :

```

void jButton1_actionPerformed(ActionEvent e) {
    chien = new ClasseChien();
    txtfldNomChien.setText(chien.getNom());
    txtfldCouleurYeuxChien.setText(chien.getCouleurYeux());
    txtfldAgeChien.setText(Integer.toString(chien.getAge()));
    chkboxChien.setSelected(true);
}

void jButton2_actionPerformed(ActionEvent e) {
    homme = new ClasseHomme();
    txtfldNomHomme.setText(homme.getNom());
    txtfldCouleurYeuxHomme.setText(homme.getCouleurYeux());
    txtfldAgeHomme.setText(Integer.toString(homme.getAge()));
    chkboxHomme.setSelected(true);
}

```

Classes abstraites

Dans une classe, il est possible de déclarer une méthode comme étant *abstraite*, ce qui signifie qu'il n'y a pas d'implémentation de la méthode dans cette classe, mais que toutes les classes qui étendent celle-ci doivent fournir une implémentation.

Par exemple, supposons que vous vouliez que tous les mammifères précisent leur vitesse de course maximale et que chaque mammifère puisse indiquer une vitesse différente. Vous devez créer une méthode abstraite appelée *vitesse()* dans la classe des mammifères. Ajoutez une méthode *vitesse()* à *ClasseMammifère*, juste au-dessus des déclarations des variables membre privées, à la fin du code source :

```
abstract public void vitesse();
```

Quand une classe contient une méthode abstraite, la totalité de la classe doit être déclarée comme abstraite. Cela signifie qu'une classe qui inclut au moins une méthode abstraite (et qui est donc une classe abstraite) ne peut pas être instanciée. Aussi, ajoutez le mot clé `abstract` au début de la déclaration de *ClasseMammifère* pour qu'elle ressemble à ceci :

```
abstract public class ClasseMammifère {

    public String getNom() {
        ...
    }
}
```

A présent, chaque classe étendant *ClasseMammifère* doit implémenter une méthode *vitesse()*. Ajoutez donc cette méthode au code de *ClasseChien* sous le constructeur de *ClasseChien()* :

```
public void vitesse() {
    JOptionPane.showMessageDialog(null, "50 km/h", "Vitesse du chien", 1);
}
```

Ajoutez cette méthode *vitesse()* au code de *ClasseHomme* :

```
public void vitesse() {
    JOptionPane.showMessageDialog(null, "30 km/h", "Vitesse de l'homme", 1);
}
```

Comme chaque méthode *vitesse()* crée un composant *JOptionPane*, qui est un composant Swing, ajoutez cette instruction immédiatement après l'instruction du paquet au début de *ClasseChien* et de *ClasseHomme* :

```
import javax.swing.*;
```

Cette instruction rend toute la bibliothèque Swing accessible à ces classes. Bientôt, nous vous en dirons plus sur l'instruction `import`.

Polymorphisme

Le polymorphisme est la possibilité pour deux classes séparées, mais reliées, de recevoir le même message mais d'agir dessus de différentes façons. En d'autres termes, deux classes différentes (mais reliées) peuvent avoir la même méthode mais l'implémenter de façon différente.

Vous pouvez ainsi avoir une méthode dans une classe, également implémentée dans une classe enfant, et accéder au code depuis la classe parent (ce qui est similaire au chaînage automatique du constructeur déjà

évoqué). Tout comme dans l'exemple du constructeur, le mot clé `super` permettra d'accéder à toutes les méthodes ou variables membre de la classe de niveau supérieur.

Voici un exemple simple. Nous avons deux classes, `Parent` et `Enfant`.

```
class Parent {
    int uneValeur = 1;
    int uneMéthode(){
        return uneValeur;
    }
}

class Enfant extends Parent {
    int uneValeur;      // cette valeur uneValeur fait partie de cette classe
    int uneMéthode() { // ceci redéfinit la méthode de la classe parent
        uneValeur = super.uneValeur + 1; // accès à la valeur uneValeur de Parent
                                         // avec super
        return super.uneMéthode() + uneValeur;
    }
}
```

La méthode `uneMéthode()` de `Enfant` surcharge la méthode `uneMéthode()` de `Parent`. Une méthode de la classe enfant qui porte le même nom qu'une méthode de la classe parent, mais qui est implémentée différemment et qui a, par conséquent, un comportement différent est une méthode surchargée.

Pouvez-vous imaginer dans quelles circonstances la méthode `uneMéthode()` de la classe `Enfant` retournerait la valeur 3 ? La méthode accède à la variable `uneValeur` de `Parent` en utilisant le mot clé `super`, lui ajoute la valeur 1, et assigne la valeur résultante (2) à sa propre variable `uneValeur`. La dernière ligne de la méthode appelle la méthode `uneMéthode()` de `Parent`, qui renvoie simplement `Parent.uneValeur` dont la valeur est 1. Pour cela, elle ajoute la valeur de `Enfant.uneValeur`, à qui la ligne précédente avait attribué la valeur 2. Ainsi, $1 + 2 = 3$.

Utilisation des interfaces

Une interface ressemble à une classe abstraite mais avec une différence importante : une interface ne peut pas inclure de code. Dans Java, le mécanisme d'interface est un moyen destiné à remplacer l'héritage multiple.

Une interface est une déclaration de classe spécialisée qui peut déclarer des constantes et des déclarations de méthodes, mais pas leur implémentation. Vous ne pouvez pas placer de code dans une interface.

Voici la déclaration d'une interface pour notre application exemple. Vous pouvez utiliser l'expert interface de JBuilder pour démarrer une interface :

- 1 Choisissez Fichier | Nouveau pour ouvrir la galerie d'objets.
Double-cliquez sur l'icône Interface pour afficher l'expert interface.
- 2 Spécifiez le nom de l'interface, par exemple `InterfaceSon`, en conservant inchangées toutes les autres valeurs. (Vous pouvez désélectionner Générer les commentaires d'en-tête pour omettre les en-têtes.)
- 3 Cliquez sur OK pour générer la nouvelle interface.

Dans la nouvelle `InterfaceSon`, ajoutez la déclaration d'une méthode `dit()` de façon à ce que l'interface ressemble à ceci :

```
package oop1;

public interface InterfaceSon {

    public void dit();
}
```

Remarquez l'utilisation du mot clé `interface` à la place de `class`. Par défaut, toutes les méthodes déclarées dans une interface sont publiques ; il est donc inutile de préciser leur accessibilité. Une classe peut implémenter une interface en utilisant le mot clé `implements`. Une classe ne peut étendre qu'une seule autre classe, mais elle peut implémenter autant d'interfaces que nécessaire. C'est de cette façon que les interfaces Java gèrent des situations qui sont normalement gérées par l'héritage multiple dans d'autres langages. Dans de nombreux cas, vous pouvez traiter l'interface comme s'il s'agissait d'une classe. En d'autres termes, pour plus de facilité, vous pouvez traiter des objets qui implémentent une interface comme des sous-classes de l'interface. Cependant, remarquez que vous ne pouvez accéder aux méthodes définies par cette interface que si vous transtypez un objet qui implémente l'interface.

L'exemple ci-dessous illustre le polymorphisme et les interfaces. Nous voulons que la définition de `ClasseMammifère` implémente la nouvelle `InterfaceSon`. Vous faites cela en ajoutant les mots `implements InterfaceSon` à la définition de la classe. Ensuite, vous devez définir et implémenter une méthode `dit()` pour `ClasseMammifère`. Modifiez votre `ClasseMammifère` de façon à ce qu'elle implémente `InterfaceSon` et une méthode `dit()`. Voici le code de `ClasseMammifère` en entier :

```
package oop1;

import javax.swing.*;

abstract public class ClasseMammifère implements InterfaceSon {

    // méthodes d'accès aux propriétés
    // nom
    public String getNom() {
```

```
        return nom;
    }

    public void setNom( String value ) {
        nom = valeur;
    }

    // couleurYeux
    public String getCouleurYeux() {
        return couleurYeux;
    }

    public void setCouleurYeux( String value ) {
        couleurYeux = valeur;
    }

    // son
    public String getSon() {
        return son;
    }

    public void setSon( String valeur ) {
        son = valeur;
    }

    // age
    public int getAge() {
        return age;
    }

    public void setAge( int value ) {
        if (valeur > 0)
        {
            age = valeur;
        }
        else
            age = 0;
    }

    public ClasseMammifère() {
        setNom( "Le nom" );
        setCouleurYeux("Marron");
        setAge(0);
    }

    public void dit() {
        JOptionPane.showMessageDialog(null, this.getSon(),
            this.getNom() + " Dit", 1);
    }

    abstract public void vitesse();

    private String nom, couleurYeux, son;
```

```
    private int age;
}
```

Désormais, la définition de `ClasseMammifère` implémente complètement l'interface `InterfaceSon`. Comme l'implémentation de la méthode `dit()` utilise un composant `JOptionPane` qui fait partie de la bibliothèque Swing, vous devez ajouter une instruction importante vers le début du fichier :

```
import javax.swing.*;
```

Cette instruction rend toute la bibliothèque Swing accessible à `ClasseMammifère`. Nous vous en dirons plus sur l'instruction `import` dans ["L'instruction import", page 6-25](#).

Comme `ClasseChien` et `ClasseHomme` étendent `ClasseMammifère`, elles ont à présent automatiquement accès à la méthode `dit()` définie dans `ClasseMammifère`. Elles n'ont pas besoin d'implémenter `dit()` elles-mêmes. La valeur de la variable `son` transmise à la méthode `dit()` est définie dans les constructeurs de `ClasseChien` et `ClasseHomme`. Voici à quoi doit ressembler la classe `ClasseChien` :

```
package oop1;
import javax.swing.*;

public class ClasseChien extends ClasseMammifère{

    public boolean hasQueue() {
        return queue;
    }

    public void setQueue(boolean valeur) {
        queue= valeur;
    }

    public ClasseChien() {
        setNom("Snoopy");
        setSon("Wouah, Wouah !");
        setAge(2);
        setQueue(true);
    }

    public void vitesse() {
        JOptionPane.showMessageDialog(null, "50 km/h", "Vitesse du chien", 1);
    }

    private boolean queue;
}
```

Voici à quoi doit ressembler `ClasseHomme` :

```
package oop1;
import javax.swing.*;

public class ClasseHomme extends ClasseMammifère {
```

```
public boolean isMarié() {
    return marié;
}

public void setMarié(boolean valeur) {
    marié= valeur;
}

public ClasseHomme() {
    setNom("Steven");
    setCouleurYeux("Bleu");
    setSon("Bonjour à tous ! Je suis " + this.getNom() + ".");
    setAge(35);
    setMarié(true);
}

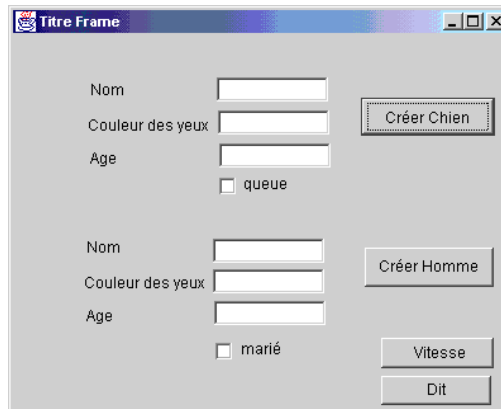
public void vitesse() {
    JOptionPane.showMessageDialog(null, "30 km/h", "Vitesse de l'homme", 1);
}

private boolean marié;
}
```

Ajout de deux nouveaux boutons

Alors que vous avez ajouté deux méthodes, `dit()` et `vitesse()` à l'application exemple, celle-ci ne les appelle jamais. Pour que cela change, ajoutez deux nouveaux boutons à la classe `Cadre1.java` :

- 1 Cliquez sur l'onglet `Cadre1` dans le volet contenu.
- 2 Cliquez sur l'onglet `Conception` pour afficher le concepteur d'interface utilisateur.
- 3 Placez deux nouveaux boutons sur la fiche.
- 4 Dans l'inspecteur, changez la valeur de la propriété `text` du premier bouton en `Vitesse` et celle du deuxième bouton en `Dit`.

Figure 6.2 Nouvelle version de l'application exemple avec les boutons Vitesse et Dit

Cliquez sur l'onglet Source pour revenir au code de `Cadre1.java` et ajoutez le code apparaissant en gras ci-dessous à la définition de la classe :

```
// Crée une référence pour les objets
ClasseChien chien;
ClasseHomme homme;

// Crée un tableau d'interfaces son
InterfaceSon listeSons[] = new InterfaceSon[2];

// Crée un tableau de mammifères
ClasseMammifère listeMammifères[] = new ClasseMammifère[2];
```

Vous avez ajouté le code créant deux tableaux : un tableau de mammifères et un d'interfaces son.

Ajoutez encore du code aux gestionnaires d'événements Créer chien et Créer homme pour ajouter aux tableaux des références aux objets chien et homme :

```
void button1_actionPerformed(ActionEvent e) {
    chien = new ClasseChien();
    txtfldNomChien.setText(chien.getNom());
    txtfldCouleurYeuxChien.setText(chien.getCouleurYeux());
    txtfldAgeChien.setText(Integer.toString(chien.getAge()));
    chkboxChien.setSelected(true);
    listeMammifères[0] = chien;
    listeSons[0] = chien;
}

void button2_actionPerformed(ActionEvent e) {
    homme = new ClasseHomme();
    txtfldNomHomme.setText(homme.getNom());
    txtfldCouleurYeuxHomme.setText(homme.getCouleurYeux());
    txtfldAgeHomme.setText(Integer.toString(homme.getAge()));
    chkboxHomme.setSelected(true);
    listeMammifères[1] = homme;
}
```

```
listeSons[1] = homme;
}
```

Revenez au concepteur d'interface utilisateur, double-cliquez sur le bouton Vitesse et remplissez le gestionnaire d'événement que JBuilder a commencé pour vous de sorte que le code ressemble à ceci :

```
void button3_actionPerformed(ActionEvent e) {
    for (int i = 0; i <= 1; i++) {
        listeMammifères[i].vitesse();
    }
}
```

Ce code parcourt dans une boucle la liste des mammifères contenue dans le tableau et demande à chaque objet d'indiquer sa vitesse. Au premier accès dans la liste, le chien affiche sa vitesse ; au deuxième accès, l'homme affiche sa vitesse. C'est le polymorphisme en action : deux objets séparés mais reliés recevant le même message et réagissant de façon différente.

Le code pour le bouton Dit est semblable.

```
void button4_actionPerformed(ActionEvent e) {
    for (int i = 0; i <= 1; i++) {
        listeSons[i].dit();
    }
}
```

Choisissez Fichier | Tout enregistrer pour enregistrer toutes vos modifications.

Vous pouvez traiter `InterfaceSon` comme une classe si besoin est. Remarquez que l'interface apporte de nombreux avantages de l'héritage multiple tout en évitant sa complexité.

Exécution de votre application

Vous voilà prêt à exécuter l'application modifiée. Choisissez Exécuter | Exécuter le projet pour recompiler, puis exécuter votre projet.

Lorsque votre application commence à s'exécuter, cliquez sur les boutons Créer chien et Créer homme pour créer les objets `chien` et `homme` avant d'essayer les boutons Vitesse et Dit, ou vous obtiendrez une exception `NullPointerException`.

Lorsque les objets existent et que vous cliquez sur le bouton Vitesse, une boîte message apparaît indiquant la vitesse du premier mammifère du tableau `listeMammifères`, le chien. Lorsque vous cliquez sur OK pour supprimer la boîte message, la deuxième boîte message apparaît. Elle indique la vitesse du deuxième mammifère, l'homme. Cliquer sur le bouton Dit provoque le même comportement, mais les messages affichés sont les sons produits par chacun des mammifères.

Paquets Java

Pour faciliter la réutilisation du code, Java permet de regrouper plusieurs définitions de classes dans un groupe logique appelé *paquet*. Si, par exemple, vous créez un groupe des règles de gestion qui modélisent les traitements de gestion de votre entreprise, vous pouvez les réunir dans un paquet. Cela facilite la réutilisation du code précédemment créé.

L'instruction import

Le langage Java est fourni avec de nombreux paquets prédéfinis. Par exemple, le paquet `java.applet` contient des classes permettant de travailler avec les applets Java.

```
public class Bonjour extends java.applet.Applet {
```

Ce code fait référence à la classe appelée `Applet` dans le paquet `java.applet` de Java. Cela peut s'avérer fastidieux de répéter le nom de classe complet `java.applet.Applet` chaque fois que vous voulez faire référence à cette classe. Java propose une autre solution. Vous pouvez importer un paquet fréquemment utilisé :

```
import java.applet.*;
```

Cela revient à dire au compilateur "si tu rencontres un nom de classe inconnu, cherche dans le paquet `java.applet`". Maintenant, pour déclarer une nouvelle classe, vous pouvez dire,

```
public class Bonjour extends Applet {
```

C'est plus concis. Cependant, cela pose un problème si deux paquets importés contiennent deux classes qui portent le même nom. Dans ce cas, il faut utiliser le nom complet.

Déclaration des paquets

La création de vos propres paquets est presque aussi simple que leur utilisation. Par exemple, pour créer un paquet appelé `monpaquet`, il suffit d'utiliser une instruction `package` au début du fichier :

```
package monpaquet;
```

```
public class Bonjour extends java.applet.Applet {  
  
    public void init() {  
        add(new java.awt.Label("Bonjour à tous sur le Web!"));  
    }  
  
} // fin classe
```

Maintenant, n'importe quel autre programme peut, pour accéder aux classes déclarées dans `monpaquet` en utilisant l'instruction :

```
import monpaquet.*;
```

Rappelez-vous que ce fichier doit se trouver dans un sous-répertoire appelé `monpaquet`. Cela permet au compilateur Java de localiser facilement votre paquet. L'expert projet de JBuilder définit automatiquement le répertoire pour qu'il corresponde au nom du projet. N'oubliez donc pas que le répertoire de base de tout paquet importé doit figurer dans le Chemin du source de l'EDI de JBuilder ou dans le Chemin du source de votre projet. Pensez-y si vous décidez de mettre un paquet dans un répertoire de base différent.

Pour plus d'informations sur l'utilisation des paquets dans JBuilder, voir "Paquets" dans *Introduction à JBuilder*.

Techniques de thread

Les threads sont impliqués dans tous les programmes Java. Un thread est un ordre d'exécution séquentiel unique dans un programme. Il possède un début, un déroulement et une fin. Un thread ne s'exécute pas de lui-même, il s'exécute dans un programme. Si votre programme est une séquence d'exécution unique, vous n'avez pas à définir de thread explicitement, la machine virtuelle Java (VM) s'en charge à votre place.

Un des aspects les plus puissants du langage Java est que vous pouvez facilement programmer plusieurs threads d'exécution afin qu'ils s'exécutent simultanément dans le même programme. Par exemple, un navigateur Web pour, simultanément, télécharger un fichier d'un site et accéder à un autre site. Si le navigateur ne peut pas faire deux tâches en même temps, vous devrez attendre que tout le fichier soit téléchargé pour naviguer jusqu'à l'autre site.

La machine virtuelle Java a toujours plusieurs threads, appelés *threads démon*, en train de s'exécuter. Par exemple, un thread démon en constante exécution effectue les tâches du ramasse-miettes (garbage collection). Un autre thread démon gère les événements souris et clavier. Il se peut que votre programme bloque un des threads de la VM Java. Si votre programme semble mort, aucun événement ne lui étant envoyé, essayez d'utiliser les threads.

Voici quelques uns des manuels donnant des informations détaillées sur l'utilisation des threads dans Java.

- *Java Threads, Second Edition* de Scott Oaks, Henry Wong et al.
- *Taming Java Threads* d'Allen Holub
- *Mastering Java Threads* de DDC Publishing

Cycle de vie d'un thread

Chaque thread a un cycle de vie défini – il démarre et s'arrête, il peut s'interrompre et attendre un événement, il peut envoyer une notification à un autre thread pendant qu'il s'exécute. Cette section présente certains des aspects les plus courants du cycle de vie d'un thread.

Personnalisation de la méthode run()

Utilisez la méthode `run()` pour implémenter le comportement d'exécution du thread. Il peut s'agir de tout ce qu'une instruction Java peut accomplir — calculs, tris, animations, etc.

Vous pouvez utiliser une des deux techniques d'implémentation de la méthode `run()` pour un thread :

- Sous-classer la classe `java.lang.Thread`
- Implémenter l'interface `java.lang.Runnable`

Sous-classement de la classe Thread

Si vous créez une nouvelle classe dont vous voulez exécuter les objets dans des threads séparés, vous devez sous-classer la classe `java.lang.Thread`. La méthode `run()` par défaut de la classe `Thread` ne fait rien, votre classe devra donc surcharger cette méthode. L'exécution de la méthode `run()` est ce qui se produit en premier au démarrage d'un thread.

Comme exemple, la classe suivante, `ThreadComptage`, sous-classe `Thread` et surcharge sa méthode `run()`. Dans cet exemple, la méthode `run()` identifie un thread et affiche son nom à l'écran. La boucle `for` compte les entiers successifs, de la valeur `début` à la valeur `fin`, et affiche chaque nombre à l'écran. Puis, avant la fin de la boucle, la méthode affiche une chaîne indiquant que le thread a fini de s'exécuter.

```
public class ThreadComptage extends Thread {
    private int début;
    private int fin;

    public ThreadComptage (int de, int à) {
        this.début = de;
        this.fin = à;
    }

    public void run() {
        System.out.println(this.getName()+ " : début d'exécution...");
        for (int i = début; i <= fin; i++) {
            System.out.print (i + " ");
        }
        System.out.println(this.getName() + " : fin d'exécution.");
    }
}
```

Pour tester la classe `ThreadComptage`, vous pouvez créer une classe de test :

```
public class ThreadTester {
    static public void main(String[] args) {
        ThreadComptage thread1 = new ThreadComptage(1, 10);
        ThreadComptage thread2 = new ThreadComptage(20, 30);
        thread1.start();
        thread2.start();
    }
}
```

La méthode `main()` dans l'application de test crée deux objets

`ThreadComptage` : `thread1`, qui compte de 1 à 10, et `thread2`, qui compte de 20 à 30. Les deux threads sont ensuite démarrés en appelant leur méthode `start()`. La sortie de cette application test doit ressembler à ceci :

```
Thread-0 : début d'exécution...
1 2 3 4 5 6 7 8 9 10 Thread-0 : fin d'exécution.
Thread-1 : début d'exécution...
20 21 22 23 24 25 26 27 28 29 30 Thread-1 : fin d'exécution.
```

Remarquez que le résultat ne garde pas les noms `thread1` et `thread2`. Sauf si vous attribuez un nom spécifique à un thread, Java lui donne automatiquement un nom de la forme `Thread-n`, où `n` est un numéro unique, commençant à 0. Vous pouvez donner un nom à un thread dans le constructeur de la classe ou avec la méthode `setName(String)`.

Dans cet exemple, `Thread-0` démarre et se termine en premier. Mais, il aurait pu démarrer en premier et se terminer en dernier, ou encore être partiellement démarré puis interrompu par `Thread-1`. C'est que, dans Java, les threads ne s'exécutent pas toujours dans le même ordre. En fait, à chaque exécution de `ThreadTester`, vous pouvez obtenir un résultat différent. L'ordre d'exécution des threads est contrôlé par l'ordonnancement de threads de Java et non pas par le programmeur. Pour plus d'informations, voir "[Priorité attribuée aux threads](#)", page 7-7.

Implémentation de l'interface `Runnable`

Si vous voulez que des objets d'une classe existant déjà s'exécutent dans leurs propres threads, vous pouvez implémenter l'interface `java.lang.Runnable`. Cette interface sert à ajouter le support des threads aux classes qui ne dérivent pas de la classe `Thread`. Elle fournit une seule méthode, la méthode `run()`, que vous devez implémenter pour votre classe.

Remarque Si votre classe dérive d'une autre classe que `Thread`, par exemple, `Applet`, vous devez utiliser l'interface `Runnable` pour créer des threads.

Pour créer une nouvelle classe `ThreadComptage` qui implémente l'interface `Runnable`, vous devez modifier la définition de la classe `ThreadComptage`. Le code de définition de la classe, les modifications apparaissant en gras, doit ressembler à ce qui suit :

```
public class ThreadComptage implements Runnable {
```

Vous devez aussi modifier la façon dont est obtenu le nom du thread. Comme vous n'instanciez pas la classe `ThreadComptage`, vous ne pouvez pas appeler la méthode `getName()` de la superclasse de `ThreadComptage` (dans ce cas, `java.lang.Object`). Cette méthode n'est pas disponible. En revanche, vous devez utiliser spécifiquement la méthode `Thread.currentThread()`, qui renvoie le nom du thread dans un format légèrement différent de la méthode `getName()`.

La classe entière, les modifications apparaissant en gras, doit ressembler à ce qui suit :

```
public class ThreadComptage implements Runnable {
    private int début;
    private int fin;

    public ThreadComptage (int de, int à) {
        this.début = de;
        this.fin = à;
    }

    public void run() {
        System.out.println(Thread.currentThread() + " : début d'exécution...");
        for (int i = début; i <= fin; i++) {
            System.out.print (i + " ");
        }
        System.out.println(Thread.currentThread() + " : fin d'exécution.");
    }
}
```

L'application test doit modifier sa façon de créer les objets. Au lieu d'instancier `ThreadComptage`, l'application doit créer un objet `Runnable` à partir de la nouvelle classe et le transmettre à un des constructeurs du thread. Le code, les modifications apparaissant en gras, doit ressembler à ce qui suit :

```
public class ThreadTester {
    static public void main(String[] args) {
        ThreadComptageRun thread1 = new ThreadComptageRun(1, 10);
        new Thread(thread1).start();
        ThreadComptageRun thread2 = new ThreadComptageRun(20, 30);
        new Thread(thread2).start();
    }
}
```

La sortie de cette application test doit ressembler à ceci :

```
Thread[Thread-0,5,main] : début d'exécution...
1 2 3 4 5 6 7 8 9 10 Thread[Thread-0,5,main] : fin d'exécution.
Thread[Thread-1,5,main] : début d'exécution...
20 21 22 23 24 25 26 27 28 29 30 Thread[Thread-1,5,main] : fin d'exécution.
```

`Thread-0` est le nom du thread, `5` la priorité qui lui a été accordée lors de sa création et `main` le `ThreadGroup` par défaut auquel le thread a été affecté. (La priorité et le groupe sont assignés par la machine virtuelle Java si vous ne les spécifiez pas.)

Voir aussi [“Priorité attribuée aux threads”, page 7-7](#)
[“Groupes de threads”, page 7-8](#)

Définition d'un thread

La classe `Thread` définit sept constructeurs. Ces constructeurs combinent les trois paramètres suivants de différentes façons :

- Un objet `Runnable` dont la méthode `run()` s'exécutera à l'intérieur du thread.
- Un objet `String` pour identifier le thread.
- Un objet `ThreadGroup` auquel attribuer le thread. La classe `ThreadGroup` organise les groupes de threads liés entre eux.

Constructeur	Description
<code>Thread()</code>	Alloue un nouvel objet <code>Thread</code> .
<code>Thread(Runnable cible)</code>	Alloue un nouvel objet <code>Thread</code> ayant cible comme objet d'exécution.
<code>Thread(Runnable cible, String nom)</code>	Alloue un nouvel objet <code>Thread</code> ayant cible comme objet d'exécution et le nom spécifié.
<code>Thread(String nom)</code>	Alloue un nouvel objet <code>Thread</code> ayant le nom spécifié.
<code>Thread(ThreadGroup groupe, Runnable cible)</code>	Alloue un nouvel objet <code>Thread</code> appartenant au groupe de threads référencé par groupe et ayant cible comme objet d'exécution.
<code>Thread(ThreadGroup groupe, Runnable cible, String nom)</code>	Alloue un nouvel objet <code>Thread</code> ayant cible comme objet d'exécution, le nom spécifié et appartenant au groupe de threads référencé par groupe.
<code>Thread(ThreadGroup groupe, String nom)</code>	Alloue un nouvel objet <code>Thread</code> appartenant au groupe de threads référencé par groupe et ayant le nom spécifié.

Si vous voulez associer un état à un thread, utilisez un objet `ThreadLocal` lorsque vous créez le thread. Cette classe permet à chaque thread de posséder sa propre copie initialisée indépendamment d'une variable statique privée, par exemple, un ID d'utilisateur ou de transaction.

Démarrage d'un thread

Pour démarrer un thread, appelez la méthode `start()`. Cette méthode crée les ressources système nécessaires pour l'exécution du thread, ordonnance le thread et appelle sa méthode `run()`.

Après le retour de la méthode `start()`, le thread s'exécute et se trouve dans l'état exécutable. Comme la majorité des ordinateurs n'ont qu'un CPU, la machine virtuelle Java doit ordonnancer les threads. Pour plus d'informations, voir ["Priorité attribuée aux threads", page 7-7](#).

Rendre un thread non exécutable

Pour placer un thread dans l'état non exécutable, utilisez une des techniques suivantes :

- Une méthode `sleep()` : ces méthodes vous permettent de spécifier un nombre de secondes et de nanosecondes pendant lesquelles le thread ne s'exécute pas.
- La méthode `wait()` : cette méthode oblige le thread en cours à attendre que la condition spécifiée soit remplie.
- Blocage du thread sur une entrée ou sur une sortie.

Lorsque le thread est inexécutable, il ne s'exécute pas, même si le processeur devient disponible. Pour quitter l'état inexécutable, la condition de l'entrée dans l'état inexécutable ne doit plus être remplie. Par exemple, si vous avez utilisé la méthode `sleep()`, le nombre de secondes spécifié doit être écoulé. Si vous avez utilisé la méthode `wait()`, un autre objet doit indiquer au thread en attente (avec `notify()` ou `notifyAll()`) le changement de condition. Si un thread est bloqué sur une entrée ou une sortie, l'entrée ou la sortie doit être terminée.

Vous pouvez également utiliser la méthode `join()` pour qu'un thread attende la fin de l'exécution d'un autre thread. Vous appelez cette méthode pour le thread attendu. Vous pouvez préciser un délai en passant à la méthode un paramètre exprimé en millisecondes. La méthode `join()` attend sur le thread jusqu'à ce que le délai soit écoulé ou que le thread soit terminé. Cette méthode fonctionne en conjonction avec la méthode `isAlive()` - `isAlive()` renvoie `true` si le thread a été démarré et non stoppé.

Notez que les méthodes `suspend()` et `resume()` ont été désapprouvées. La méthode `suspend()` est susceptible de provoquer des verrouillages mortels. Si le thread cible verrouille un moniteur protégeant une ressource système critique lorsqu'il est suspendu, aucun thread ne peut accéder à cette ressource jusqu'à ce que le thread cible ait repris. Un moniteur est un objet Java servant à vérifier qu'un seul thread à la fois exécute les méthodes synchronisées pour l'objet que le moniteur contrôle. Pour plus d'informations, voir ["Threads synchronisés", page 7-8](#).

Arrêt d'un thread

Vous ne pouvez plus stopper un thread avec la méthode `stop()`. Cette méthode a été désapprouvée, car elle n'est pas sûre. Stopper un thread provoquera le déverrouillage de tous les moniteurs qu'il avait verrouillé. Si un objet préalablement protégé par un de ces moniteurs se trouve dans un état inconsistant, d'autres threads verront cet objet comme inconsistant. Cela risque d'endommager votre programme.

Pour stopper un thread, terminez la méthode `run()` avec un boucle finie.

Pour plus d'informations, voir la rubrique appelée "Why are Thread.stop, Thread.suspend, Thread.resume and runtime.runFinalizersOnExit Deprecated?" dans la documentation du JDK.

Priorité attribuée aux threads

Lorsqu'un thread Java est créé, il hérite sa priorité du thread qui le crée. Vous pouvez définir la priorité d'un thread avec la méthode `setPriority()`. Les priorités des threads sont représentées par des valeurs entières comprises entre `MIN_PRIORITY` et `MAX_PRIORITY` (constantes de la classe `Thread`). Le thread ayant la plus forte priorité est exécuté.

Lorsque ce thread stoppe, relâche le CPU ou devient non exécutable, un thread de plus faible priorité est exécuté. Si deux threads de même priorité sont en attente, l'ordonnanceur Java choisira de les exécuter à tour de rôle. Le thread s'exécutera jusqu'à ce que :

- Un thread de priorité supérieure devienne exécutable.
- Le thread relâche, par l'utilisation de la méthode `yield()` ou à la fin de sa méthode `run()`.
- Le temps qui lui a été dévolu expire. Cela s'applique uniquement aux systèmes supportant le temps partagé.

Ce type d'ordonnancement est basé sur un algorithme appelé *ordonnancement selon des priorités fixes*. Les threads sont exécutés en fonction de leur priorité comparée à celles des autres threads. Le thread ayant la plus forte priorité sera toujours en train de s'exécuter.

Temps partagé

Certains systèmes d'exploitation utilisent un mécanisme d'ordonnancement désigné sous les termes de *temps partagé*. Ce mécanisme divise le temps CPU en tranches. Le système fait s'exécuter les threads de priorités égales les plus hautes, jusqu'à ce qu'un ou plusieurs d'entre eux finissent, ou jusqu'à ce qu'un thread de priorité supérieure

passer dans l'état exécutable. Le temps partagé n'étant pas supporté par tous les systèmes d'exploitation, votre programme ne doit pas dépendre d'un tel mécanisme d'ordonnancement.

Threads synchronisés

Un des problèmes centraux du traitement multithread est la gestion de situations où plusieurs threads ont accès à la même structure de données. Par exemple, si un thread essaie de mettre à jour les éléments d'une liste, tandis qu'un autre thread essaie de les trier, votre programme risque un verrouillage mortel ou l'apparition de résultats incorrects. Pour éviter ce problème, vous devez *synchroniser les threads*.

La façon la plus simple d'empêcher deux objets d'accéder à la même méthode en même temps est d'exiger qu'un thread obtienne un verrou. Lorsqu'un thread pose un verrou, un autre thread nécessitant un verrou, doit attendre que le premier thread libère le verrou. Pour qu'une méthode soit *sûre par rapport aux threads*, utilisez le mot clé `synchronized` lors de la déclaration des méthodes ne pouvant être exécutées que par un seul thread à la fois. Vous pouvez également synchroniser un objet.

Par exemple, si vous créez une méthode `swap()`, qui intervertit deux valeurs en utilisant une variable locale, et si vous créez deux threads différents qui exécutent cette méthode, votre programme peut produire des résultats incorrects. Le premier thread, du fait de l'ordonnanceur Java, risque d'exécuter seulement la première moitié de la méthode. Puis, le second thread peut exécuter toute la méthode, en utilisant des valeurs incorrectes (puisque le premier thread n'a pas achevé l'opération). Le premier thread peut revenir ensuite pour terminer la méthode. Dans ce cas, il semblera que l'inversion des valeurs n'a jamais eu lieu. Pour empêcher que cela se produise, utilisez le mot clé `synchronized` dans la déclaration de votre méthode.

Comme règle de base, toute méthode qui modifie les propriétés des objets devrait être déclarée `synchronized`.

Groupes de threads

Chaque thread Java fait partie d'un *groupe de threads*. Un groupe de threads rassemble plusieurs threads en un objet unique afin de les manipuler tous en même temps. Les groupes de threads sont implémentés par la classe `java.lang.ThreadGroup`.

Le système d'exécution place un thread dans un groupe au moment de la construction du thread. Le thread est placé soit dans le groupe par défaut soit dans le groupe de threads spécifié au moment de la création du

thread. Vous ne pouvez pas placer le thread dans un nouveau groupe après que le thread a été créé.

Si vous créez un thread sans spécifier de nom de groupe dans le constructeur, le système d'exécution place le nouveau thread dans le même groupe que le thread qui l'a créé. Habituellement, les threads non spécifiés sont placés dans le groupe du thread principal. Mais, si vous créez un thread dans une applet, le nouveau thread risque d'être placé dans un autre groupe que le groupe principal, selon le navigateur ou le visualiseur où s'exécute l'applet.

Si vous construisez un thread avec `ThreadGroup`, ce groupe peut être :

- Un nom de votre choix
- Un groupe créé par le runtime Java
- Un groupe créé par l'application dans laquelle votre applet s'exécute

Pour connaître le nom du groupe dont fait partie votre thread, utilisez la méthode `getThreadGroup()`. Lorsque vous connaissez le nom d'un groupe, vous pouvez connaître les autres threads appartenant au groupe et les manipuler tous en même temps.

Sérialisation

La *sérialisation d'un objet* est le processus de stockage d'un objet complet sur disque ou tout autre système de stockage, d'où il pourra être restauré à tout moment. Le processus inverse de la restauration est connu sous le nom de *désérialisation*. Dans cette section, nous allons voir l'utilité de la sérialisation et la façon dont Java implémente la sérialisation et la désérialisation.

Un objet sérialisé est dit *persistant*. La plupart des objets en mémoire sont *transitoires*, ce qui veut dire qu'ils disparaissent quand leur référence est hors de portée ou que l'ordinateur est éteint. Les objets persistants existent tant qu'un exemplaire d'eux reste stocké quelque part sur un disque, une cartouche ou un CD-ROM.

Pourquoi sérialiser ?

Traditionnellement, enregistrer des données sur disque ou une autre unité de stockage nécessite la définition d'un format de données spécial, l'écriture de fonctions de lecture et d'écriture pour ce format et la création d'une mappe entre le format du fichier et le format des données. Les fonctions de lecture et d'écriture des données étaient soit simples et sans possibilité d'extension, soit complexes et difficiles à créer et à maintenir.

Java, qui est complètement basé sur les objets et la programmation orientée objet, fournit un mécanisme de stockage des objets, appelé sérialisation. Si vous utilisez les moyens offerts par Java, vous n'aurez plus à vous préoccuper des détails concernant le format des fichiers ou les entrées/sorties. A la place, vous pourrez vous concentrer sur la résolution des cas réels en concevant et en implémentant des objets. Si, par exemple, vous rendez une classe persistante et si vous lui ajoutez ultérieurement de nouveaux champs, vous n'avez pas à vous préoccuper de la modification

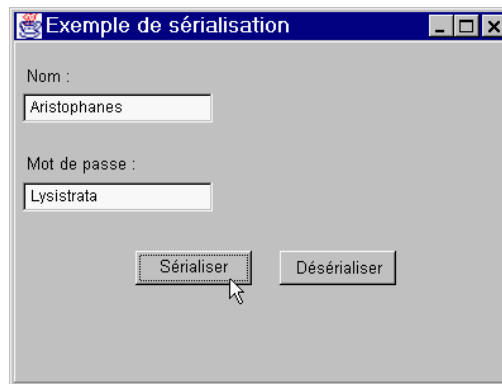
des programmes de lecture et d'écriture des données. Tous les champs qui se trouvent dans un objet sérialisé sont automatiquement écrits et restaurés.

Sérialisation Java

La sérialisation est une fonctionnalité apparue la première fois dans le JDK 1.1. La prise en charge par Java de la sérialisation se compose de l'interface `Serializable`, des classes `ObjectOutputStream` et `ObjectInputStream` ainsi que de quelques autres classes et interfaces. Nous allons illustrer ces trois éléments par une application qui enregistre sur disque des informations sur les utilisateurs et les relit.

Par exemple, supposons que vous vouliez enregistrer des informations sur un utilisateur particulier comme l'illustre la figure présentée ici.

Figure 8.1 Enregistrement d'un nom utilisateur et d'un mot de passe



Une fois que l'utilisateur a tapé son nom et son mot de passe dans les champs appropriés, l'application doit enregistrer ces informations sur disque. Naturellement, cet exemple est très simple, mais il est applicable à l'enregistrement du dernier document ouvert, des préférences des utilisateurs pour leurs applications, etc.

En utilisant JBuilder, vous pouvez concevoir une interface utilisateur comme celle montrée plus haut. Voir "Conception d'une interface utilisateur" dans *Conception d'interfaces utilisateur avec JBuilder*, si vous avez besoin d'aide à ce sujet. Nommez le champ de saisie Nom `champTexteNom`, et le champ Mot de passe `champTexteMotPasse`. En plus des deux libellés que vous pouvez voir, ajoutez-en un troisième, en bas du cadre, et nommez-le `sortieLibellé`.

Utilisation de l'interface `Serializable`

Créez une nouvelle classe représentant l'utilisateur en cours. Ses propriétés doivent représenter le nom de l'utilisateur et son mot de passe.

Pour créer la nouvelle classe,

- 1 Choisissez Fichier | Nouvelle classe pour afficher l'expert classe.
- 2 Dans la section Informations classe, spécifiez le nom de la nouvelle classe, `InfoUtil`. Laissez inchangés tous les autres champs.
- 3 Dans la section Options, cochez juste les options Publique et Créer un constructeur par défaut, en laissant les autres non cochées.
- 4 Choisissez OK.

L'expert classe crée pour vous le nouveau fichier classe et l'ajoute au projet. Modifiez le code généré pour qu'il ressemble à ceci :

```
package serialize;

public class InfoUtil implements java.io.Serializable {
    private String nomUtil = "";
    private String motPasseUtil = "";

    public InfoUtil() {
    }

    public String obtientNomUtil() {
        return nomUtil;
    }

    public void définitNomUtil(String s) {
        nomUtil = s;
    }

    public String obtientMotPasseUtil() {
        return motPasseUtil;
    }

    public void définitMotPasseUtil(String s) {
        motPasseUtil = s;
    }
}
```

Vous avez ajouté une variable contenant le nom de l'utilisateur et une autre contenant son mot de passe. Vous avez également ajouté des méthodes d'accès aux deux champs.

Vous remarquerez que la classe `InfoUtil` implémente l'interface `java.io.Serializable`. `Serializable` est connue sous le nom d'*interface de balisage*, puisqu'elle ne spécifie aucune méthode à implémenter, mais "balise" plutôt ses objets comme étant d'un type particulier.

Tout objet prévu pour être sérialisé doit implémenter l'interface `Serializable`. Cela est important, car, sinon, les techniques utilisées dans la suite de ce chapitre ne fonctionnent pas. Si, par exemple, vous essayez de sérialiser un objet qui n'implémente pas cette interface, une `NotSerializableException` est émise.

A ce stade, il faut importer le paquet `java.io` pour que votre application ait accès aux classes et interfaces d'entrée et de sortie et que vous puissiez écrire et lire des objets. Ajoutez cette instruction d'importation au début de votre classe cadre :

```
import java.io.*
```

Utilisation des flux de sortie

Avant de sérialiser l'objet `InfoUtil`, donnez-lui les valeurs saisies par l'utilisateur dans les zones de texte. Quand l'utilisateur entre des informations dans les champs et clique sur le bouton **Sérialiser**, les valeurs qu'il a entrées sont stockées dans l'instance de l'objet `InfoUtil` :

```
void jButton1_actionPerformed(ActionEvent e) {
    InfoUtil user = new InfoUtil(); // instancie un objet utilisateur
    user.définitNomUtil(champTexteNom.getText());
    user.définitMotPasseUtil(champTexteMotPasse.getText());
}
```

Si vous utilisez le concepteur d'interface utilisateur de `JBuilder`, double-cliquez sur le bouton **Sérialiser** et `JBuilder` va commencer à écrire pour vous le code de l'événement `jButton1_actionPerformed()`. Instanciez un objet utilisateur, puis ajoutez au gestionnaire d'événement les appels aux méthodes `user.définitNomUtil()` et `user.définitMotPasseUtil()`.

Ensuite, ouvrez un `FileOutputStream` vers le fichier qui va contenir les données sérialisées. Dans cet exemple, le fichier s'appelle `C:\infoUtil.ser`. Ajoutez ce code au gestionnaire d'événement du bouton **Sérialiser** :

```
try {
    FileOutputStream file = new FileOutputStream("c:\infoUtil.ser");
```

Créez un `ObjectOutputStream` qui sérialisera l'objet et l'enverra au `FileOutputStream` en ajoutant ce code au gestionnaire d'événement :

```
ObjectOutputStream out = new ObjectOutputStream(file);
```

Vous pouvez maintenant envoyer l'objet `InfoUtil` au fichier. Pour cela, appelez la méthode `writeObject()` de `ObjectOutputStream`. Appeler la méthode `flush()` videra le tampon de sortie pour garantir l'écriture effective de l'objet dans le fichier.

```
out.writeObject(u);
out.flush();
```

Fermez le flux de sortie pour libérer les ressources utilisées par le flux, comme les descripteurs du fichier.

```
    out.close();
}
```

Ajoutez au gestionnaire le code qui capture une `IOException` en cas de problème d'écriture sur le fichier ou si l'objet ne prend pas en charge l'interface `Serializable`.

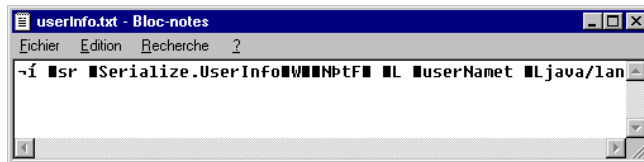
```
catch (java.io.IOException IOE) {
    sortieLibellé.setText("IOException");
}
```

Voici comment devrait se présenter en entier le gestionnaire d'événement du bouton Sériàliser :

```
void jButton1_actionPerformed(ActionEvent e) {
    InfoUtil user = new InfoUtil();
    user.définirNomUtil(champTexteNom.getText());
    user.définirMotPasseUtil(champTexteMotPasse.getText());
    try {
        FileOutputStream file = new FileOutputStream("c:\\infoUtil.ser");
        ObjectOutputStream out = new ObjectOutputStream(file);
        out.writeObject(user);
        out.flush();
    }
    catch (java.io.IOException IOE) {
        sortieLibellé.setText("IOException");
    }
    finally {
        out.close();
    }
}
```

Compilez maintenant votre projet et exécutez-le. Entrez des valeurs dans les champs Nom et Mot de passe, puis cliquez sur le bouton Sériàliser. Vous pouvez vérifier dans un éditeur de texte que l'objet a été écrit. (N'essayez pas de le modifier, sinon le fichier risquerait d'être endommagé !) Notez qu'un objet sérialisé contient un mélange de texte ASCII et de données binaires :

Figure 8.2 L'objet sérialisé



Méthodes ObjectOutputStream

La classe `ObjectOutputStream` contient plusieurs méthodes utiles pour mettre des données dans un flux. Vous n'êtes pas limité aux objets d'écriture. Appeler `writeInt()`, `writeFloat()`, `writeDouble()`, etc., écrira dans un flux n'importe quel type fondamental. Pour écrire plusieurs objets ou types fondamentaux dans le même flux, appelez ces méthodes plusieurs fois sur le même objet `ObjectOutputStream`. Cependant, faites attention à relire les objets *dans le même ordre*.

Utilisation des flux d'entrée

L'objet a maintenant été écrit sur disque, mais comment le récupérer ? Une fois que l'utilisateur a cliqué sur le bouton Désérialiser, il vous faut relire les données du disque et les mettre dans un nouvel objet.

Pour commencer le processus, créez un nouvel objet `FileInputStream` pour lire le fichier que vous venez d'écrire. Si vous utilisez JBuilder, double-cliquez dans le concepteur d'interface utilisateur sur le bouton Désérialiser et, dans le gestionnaire d'événement créé pour vous par JBuilder, ajoutez le code mis en évidence :

```
void jButton2_actionPerformed(ActionEvent e) {
    try {
        FileInputStream file = new FileInputStream("c:\\infoUtil.ser");
```

Ensuite, créez un `ObjectInputStream`, ce qui permet de lire les objets de ce fichier.

```
ObjectInputStream input = new ObjectInputStream(file);
```

Ensuite, appelez la méthode `ObjectInputStream.readObject()` pour lire le premier objet de ce fichier. `readObject()` renvoie un type `Object` qu'il faut transtyper dans le type approprié (`InfoUtil`).

```
InfoUtil user = (InfoUtil)input.readObject();
```

Après la lecture, n'oubliez pas de fermer `ObjectInputStream` pour libérer les ressources qui lui sont associées, comme les descripteurs du fichier.

```
input.close();
```

Enfin, servez-vous de l'objet `user` comme vous utiliserez n'importe quel objet de la classe `InfoUtil` : Dans ce cas, vous affichez le nom et le mot de passe dans le troisième champ libellé que vous avez ajouté à la boîte de dialogue :

```
sortieLibellé.setText("Le nom est " + user.obtientNomUtil() +
    ", le mot de passe est : " + user.obtientMotPasseUtil());
```


La lecture à partir d'un fichier peut provoquer une `IOException` qu'il vous faut gérer. Vous risquez de recevoir aussi une `StreamCorruptedException` (sous-classe de `IOException`) si le fichier a été endommagé d'une manière ou d'une autre :

```
catch (java.io.IOException IOE) {
    sortieLibellé.setText("IOException");
}
```

Tenez également compte de l'exception suivante. La méthode `readObject()` peut déclencher une `ClassNotFoundException`. Cette exception peut survenir si vous tentez de lire un objet pour lequel il n'y a pas d'implémentation. Par exemple, si l'objet a été écrit par un autre programme ou si vous avez renommé la classe `InfoUtil` depuis la dernière écriture du fichier, vous obtenez une `ClassNotFoundException`.

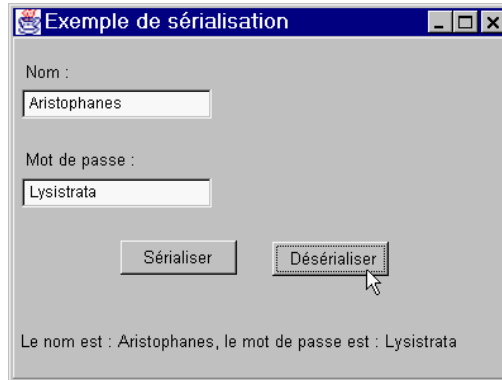
```
catch (ClassNotFoundException cnfe) {
    sortieLibellé.setText("ClassNotFoundException");
}
}
```

Voici le gestionnaire d'événement du bouton Désérialiser, en entier :

```
void jButton2_actionPerformed(ActionEvent e) {
    try {
        FileInputStream file = new FileInputStream("c:\\infoUtil.ser");
        ObjectInputStream input = new ObjectInputStream(file);
        InfoUtil user = (InfoUtil)input.readObject();
        input.close();
        sortieLibellé.setText("Le nom est " + user.obtientNomUtil() +
            ", le mot de passe est : " + user.obtientMotPasseUtil());
    }
    catch (java.io.IOException IOE) {
        sortieLibellé.setText("IOException");
    }
    catch (ClassNotFoundException cnfe) {
        sortieLibellé.setText("ClassNotFoundException");
    }
}
```

Maintenant, vous pouvez compiler et exécuter votre projet, entrer les valeurs du Nom et du Mot de passe, et cliquer sur le bouton Sériialiser pour stocker sur disque les informations. Ensuite, cliquez sur le bouton Désérialiser pour lire à nouveau l'objet `InfoUtil` sérialisé.

Figure 8.3 L'objet restauré



Méthodes `ObjectInputStream`

`ObjectInputStream` contient également des méthodes comme `readDouble()`, `readFloat()`, etc., qui sont les contreparties des méthodes `writeDouble()`, `writeFloat()`, etc. Vous pouvez appeler chaque méthode en séquence, de la même façon que les objets ont été écrits dans le flux.

Ecriture et lecture des flux d'objets

Vous pouvez vous demander ce qui se passe quand un objet en cours de sérialisation contient un champ qui fait référence à un autre objet plutôt qu'à un type primitif. Dans ce cas, l'objet de base, ainsi que l'objet secondaire, sont écrits dans le flux. Cependant, n'oubliez pas que les deux objets écrits dans le flux doivent implémenter l'interface `Serializable`. Si ce n'est pas le cas, un appel de la méthode `writeObject()` provoquera une `NotSerializableException`.

Cette sérialisation des objets peut poser des problèmes de sécurité. Dans l'exemple ci-dessus, vous avez écrit un mot de passe dans un objet sérialisé. Bien que cette technique soit acceptable dans certains cas, veillez aux problèmes de sécurité quand vous décidez de sérialiser un objet.

Finalement, pour créer un objet persistant sans utiliser le mécanisme de sérialisation par défaut, l'interface `Serializable` documente deux méthodes, `writeObject()` et `readObject()`, que vous pouvez implémenter pour exécuter une sérialisation personnalisée. L'interface `Externalizable` fournit aussi un mécanisme similaire. Pour avoir des informations sur ces techniques, consultez la documentation JDK.

Introduction à la machine virtuelle Java

Ce chapitre est une introduction à la machine virtuelle Java (JVM). Bien qu'il soit important que vous assimiliez les informations élémentaires concernant la JVM, si vous ne vous aventurez pas dans une programmation Java particulièrement avancée, la JVM est justement quelque chose dont vous n'avez pas à vous préoccuper. Ce chapitre n'est là qu'à titre informatif.

Avant d'explorer la machine virtuelle Java, nous allons présenter certains des termes utilisés dans ce chapitre. Tout d'abord, la machine virtuelle Java (Java Virtual Machine ou JVM) est l'environnement d'exécution des programmes Java. Elle définit un ordinateur abstrait et précise les instructions que ce dernier peut exécuter. Ces instructions sont appelées *bytecodes*. De façon générale, les bytecodes Java sont à la JVM ce que le jeu d'instructions est à une CPU. Un bytecode est une instruction d'un octet de long générée par le compilateur Java et exécutée par l'interpréteur Java. Lors de la compilation d'un fichier .java, le compilateur produit une suite de bytecodes qu'il stocke dans un fichier .class. L'interpréteur Java peut ensuite exécuter les bytecodes stockés dans le fichier .class.

Dans ce chapitre, il sera également fait mention d'applications et d'applets Java. Il est parfois utile de faire la distinction entre une application Java et un applet Java. Dans certaines sections de ce chapitre, la distinction est superflue. Nous utiliserons dans ce cas le mot *app* pour faire référence aux applications Java comme aux applets Java.

Il est important d'expliquer clairement ici ce qu'est réellement Java. Java est plus qu'un simple langage informatique ; c'est un *environnement* informatique. C'est parce que Java est composé de deux éléments principaux distincts, chacun étant une partie essentielle de Java : Java de

conception (le langage Java lui-même) et Java d'exécution (la JVM). C'est une interprétation technique du mot *Java*.

L'interprétation pratique du mot *Java* est qu'il représente l'environnement d'exécution, et non le langage. Une phrase comme "cet ordinateur peut exécuter Java" signifie en fait que cet ordinateur prend en charge l'environnement d'exécution de Java (le JRE) et, plus précisément, qu'il implémente une machine virtuelle Java.

La distinction doit être faite entre la *spécification JVM* et une *implémentation* de la JVM. La spécification JVM est un document (disponible sur le site Sun <http://java.sun.com/docs/books/vmspec/index.html>) qui définit comment implémenter une JVM. Quand une implémentation de la JVM suit correctement cette spécification, cela garantit essentiellement que les apps Java s'exécuteront sur cette implémentation de la JVM avec les mêmes résultats que produiraient les mêmes applications exécutées sur toute autre implémentation de la JVM. La spécification JVM garantit que les programmes Java seront capables de s'exécuter sur n'importe quelle plate-forme.

La spécification JVM est indépendante des plates-formes, car elle peut être implémentée sur n'importe quelle plate-forme. Remarquez qu'une implémentation particulière de la JVM est dépendante des plates-formes. C'est parce que l'implémentation JVM est la seule partie de Java qui interagit directement avec le système d'exploitation de votre ordinateur. Comme chaque système d'exploitation est différent, toute implémentation JVM spécifique doit savoir comment interagir avec le système d'exploitation spécifique pour lequel elle est conçue.

L'exécution des programmes Java sous une implémentation de la JVM garantit un environnement d'exécution prévisible, car toutes les implémentations de la JVM sont conformes à la spécification JVM. Même s'il y a différentes implémentations de la JVM, elles suivent toutes un certain nombre d'exigences qui garantissent la portabilité. En d'autres termes, les différences éventuelles entre les diverses implémentations n'affectent pas la portabilité.

La JVM est responsable de l'exécution des fonctions suivantes :

- Allocation de mémoire aux objets créés
- Récupération des données périmées (garbage collection)
- Gestion du recensement et des piles
- Appel du système hôte pour certaines fonctions, comme l'accès aux périphériques
- Suivi de la sécurité des apps Java

Dans la suite de ce chapitre, nous allons nous concentrer sur la dernière fonction : la sécurité.

Sécurité de la machine virtuelle Java

Un des rôles les plus importants de la JVM consiste à gérer la sécurité des apps Java. La JVM utilise un mécanisme spécifique pour imposer certaines restrictions de sécurité aux apps Java. Ce mécanisme (ou modèle de sécurité) joue les rôles suivants :

- Il détermine jusqu'à quel niveau le code qui s'exécute est "fiabilisé" et lui attribue le niveau d'accès approprié
- Il contrôle que les bytecodes n'effectuent pas d'opérations interdites
- Il vérifie que chaque bytecode est généré correctement

Dans les sections suivantes, nous allons voir comment cette sécurité est assurée par Java.

Le modèle de sécurité

Dans cette section, nous allons nous intéresser aux différents éléments du modèle de sécurité de Java. Nous allons en particulier examiner les rôles du vérificateur Java, du gestionnaire de sécurité et du paquet `java.security`, ainsi que du chargeur de classe. Ce sont des composants qui sécurisent les apps Java.

Le vérificateur Java

A chaque chargement d'une classe, il faut commencer par une vérification. Le rôle principal de cette vérification consiste à s'assurer que chaque bytecode de la classe ne viole pas les spécifications de la machine virtuelle Java. Par exemple, les erreurs de type et les opérations arithmétiques en débordement ou en sous-débordement sont des violations. La vérification est effectuée par le vérificateur Java et se décompose en quatre étapes :

- 1 Vérification de la structure des fichiers classe.
- 2 Exécution des vérifications au niveau système.
- 3 Validation des bytecodes.
- 4 Exécution des contrôles de types et d'accès au moment de l'exécution.

La première étape du vérificateur concerne le contrôle de la structure du fichier classe. Tous les fichiers classe ont une structure commune ; ils doivent, par exemple, toujours commencer par ce qu'il est convenu d'appeler le nombre *magique* dont la valeur est `0xCAFEBABE`. Pendant cette étape, le vérificateur contrôle aussi la validité de la réserve de constantes (la réserve de constantes est l'emplacement où sont stockés les chaînes et les nombres des fichiers classe). En outre, le vérificateur s'assure qu'aucun octet n'a été ajouté à la fin du fichier classe.

La deuxième étape exécute des vérifications au niveau système : validité de toutes les références à la réserve de constantes et assurance que toutes les sous-classes sont correctes.

La troisième étape valide les bytecodes. C'est l'étape du processus de vérification la plus importante et la plus complexe. Valider un bytecode signifie contrôler la validité de son type ainsi que le nombre et le type de ses arguments. Le vérificateur contrôle aussi que les appels aux méthodes transmettent des arguments dont le nombre et le type sont corrects et que chaque fonction externe renvoie le type exact.

L'étape finale effectue les contrôles d'exécution. Les classes référencées en externe sont chargées et leurs méthodes sont contrôlées. Ce contrôle vérifie que les appels aux méthodes correspondent à la signature des méthodes dans les classes externes. Le vérificateur suit aussi les tentatives d'accès par la classe actuellement chargée pour s'assurer qu'elle ne viole pas les restrictions d'accès. Un autre contrôle d'accès a lieu sur les variables pour vérifier que les variables `private` et `protected` ne subissent aucun accès interdit.

Avec ce processus de vérification exhaustive, nous pouvons mesurer l'importance du vérificateur Java pour le modèle de sécurité. Il est également important de noter que le processus de vérification doit avoir lieu au niveau du vérificateur et non à celui du compilateur, puisque n'importe quel compilateur peut être programmé pour générer des bytecodes Java. Il est donc clair que se fier au compilateur pour exécuter la vérification est dangereux, puisque le compilateur peut être programmé pour ne pas en tenir compte. Ce point montre pourquoi la JVM est nécessaire.

Pour plus d'informations sur le vérificateur JAVA, consultez *Virtual Machine Specification* (<http://java.sun.com/docs/books/vmspec/index.html>).

Le gestionnaire de sécurité et le paquet `java.security`

La classe `SecurityManager` est une des classes définies dans le paquet `java.lang`. Cette classe contrôle la politique de sécurité des apps Java pour vérifier si l'app qui s'exécute possède la permission d'effectuer certaines opérations dangereuses. Le rôle principal de cette stratégie est de déterminer les droits d'accès. Dans Java 1.1, la classe `SecurityManager` était uniquement responsable de la définition de la politique de sécurité, mais dans Java 2 et dans les versions ultérieures, un modèle de sécurité robuste et plus détaillé a été établi grâce au nouveau paquet `java.security`. La classe `SecurityManager` dispose de plusieurs méthodes dont le nom commence par "check". Dans Java 1.1, l'implémentation par défaut de ces méthodes "check" consistait à déclencher une `SecurityException`. Depuis Java 2, l'implémentation par défaut de la plupart des méthodes "check" appelle `SecurityManager.checkPermission()`, et l'implémentation par défaut de cette méthode appelle à son tour

`java.security.AccessController.checkPermission()`. C'est `AccessController` qui est responsable de l'algorithme de vérification des permissions.

La classe `SecurityManager` contient le nombreuses méthodes qui servent à contrôler si une opération particulière est autorisée. Par exemple, les méthodes `checkRead()` et `checkWrite()` contrôlent si l'appelant de la méthode a le droit d'effectuer une lecture ou une écriture sur un fichier donné. Ils le font en appelant `checkPermission()`, qui à son tour appelle `AccessController.checkPermission()`. Dans le JDK, de nombreuses méthodes utilisent la classe `SecurityManager` avant d'effectuer des opérations dangereuses. Le JDK fait cela pour des raison de continuité ; `SecurityManager` existait dans des versions précédentes du JDK quand le modèle de sécurité était bien plus limité. Dans vos apps, vous appellerez probablement `AccessController.checkPermission()` directement, au lieu d'utiliser la classe `SecurityManager` (qui appelle la même méthode indirectement).

La méthode statique `System.setSecurityManager()` peut être utilisée pour charger le gestionnaire de sécurité par défaut dans l'environnement. Ainsi, dès qu'une app Java doit effectuer une opération dangereuse, elle peut consulter l'objet `SecurityManager` chargé dans l'environnement.

Généralement, toutes les apps java utilisent la classe `SecurityManager` de la même façon. Une instance de `SecurityManager` est tout d'abord créée, soit à l'aide d'un argument spécial de la ligne de commande au démarrage de l'app ("`-Djava.security.manager`"), soit dans du code similaire à celui-ci :

```
SecurityManager security = System.getSecurityManager();
```

La méthode `System.getSecurityManager()` renvoie une occurrence de la classe `SecurityManager` actuellement chargée. Si aucune classe `SecurityManager` n'a été définie avec la méthode `System.setSecurityManager()`, `System.getSecurityManager()` renvoie `null` ; sinon, elle renvoie une occurrence de la classe `SecurityManager` chargée dans l'environnement. Maintenant, considérons que l'app veut vérifier si elle a le droit de lire un fichier. Elle le fait de la façon suivante :

```
if (security != null) {
    security.checkRead (nomFichier);
}
```

L'instruction `if` regarde d'abord si un objet `SecurityManager` existe, puis elle appelle la méthode `checkRead()`. Si `checkRead()` n'autorise pas cette opération, une `SecurityException` est déclenchée et l'opération n'a pas lieu ; sinon, elle continue correctement.

Habituellement, un gestionnaire de sécurité est chargé lorsqu'une applet s'exécute, car la plupart des navigateurs Java en utilisent un automatiquement. Une application, en revanche, n'utilise pas automatiquement de gestionnaire de sécurité, sauf si on en charge un dans l'environnement, à l'aide de la méthode `System.setSecurityManager()` ou depuis la ligne de commande au démarrage de l'application. Pour utiliser

la même politique de sécurité pour une application que pour une applet, vous devez vous assurer que le gestionnaire de sécurité est chargé.

Pour spécifier votre propre politique de sécurité, vous devez travailler avec les classes du paquet `java.security`. Les classes importantes de ce paquet sont `Policy`, `Permission` et `AccessController`. Vous ne devez pas sous-classer `SecurityManager` sauf en dernier recours et, dans ce cas, avec d'extrêmes précautions. Une discussion détaillée du paquet `security` sortirait du cadre de ce manuel. La politique de sécurité par défaut devrait suffire à la plupart des développeurs Java. Quand vous estimerez être concerné par des sujets de sécurité plus évolués ou quand vous voudrez simplement plus d'informations sur le paquet `java.security`, consultez "Security Architecture" dans la documentation JDK.

Le chargeur de classe

Le chargeur de classe s'associe au gestionnaire de sécurité pour gérer la sécurité des apps Java. Les rôles principaux du chargeur de classe sont résumés ci-dessous :

- Il détermine si la classe qu'il essaye de charger ne l'est pas déjà
- Il charge les fichiers classe dans la machine virtuelle
- Il détermine les permissions attribuées à la classe chargée en accord avec la politique de sécurité
- Il fournit au gestionnaire de sécurité certaines informations relatives aux classes chargées
- Il détermine le chemin à partir duquel la classe doit être chargée (les classes système sont toujours chargées depuis le `BOOTCLASSPATH`)

Chaque instance d'une classe est associée à un objet chargeur de classe, qui est une instance d'une sous-classe de la classe abstraite `java.lang.ClassLoader`. Le chargement est automatique lorsqu'une classe est instanciée. Il est possible de créer un chargeur de classe personnalisé en sous-classant `ClassLoader` ou une de ses sous-classes existantes, mais, dans la plupart des cas, ce n'est pas nécessaire. Si vous avez besoin d'avoir plus d'informations sur le mécanisme de chargement des classes, lisez la documentation sur `java.lang.ClassLoader` et le document "Security Architecture" dans la documentation JDK.

Jusqu'ici, nous avons vu comment le vérificateur Java, la classe `SecurityManager` et le chargeur de classe fonctionnent pour assurer la sécurité des apps Java. D'autres mécanismes non décrits dans ce chapitre, comme ceux du paquet `java.security`, augmentent la sécurité des apps Java. Il existe également une mesure de sécurité intégrée au langage Java lui-même, mais cela dépasse la cadre de ce chapitre.

Les compilateurs Just-In-Time

Dans ce chapitre, il convient d'inclure une brève remarque sur les compilateurs Just-In-Time (JIT). Ces compilateurs convertissent les bytecodes Java en instructions machine natives exécutées directement par la CPU. Cela augmente de façon évidente les performances des apps Java. Mais si des instructions natives sont exécutées à la place des bytecodes, qu'advient-il du processus de vérification déjà mentionné ? En fait, le processus de vérification ne change pas, puisque le vérificateur Java vérifie toujours les bytecodes avant leur conversion.

Utilisation de l'interface native Java (JNI)

Ce chapitre explique comment appeler des méthodes natives dans les applications Java en utilisant JNI, l'interface de méthodes natives Java. Il commence par expliquer comment fonctionne l'interface JNI, puis décrit le mot clé **native** et montre comment toute méthode Java peut devenir une méthode native. Enfin, il étudie l'outil `javah` du JDK, qui permet de générer des fichiers d'en-tête C pour des classes Java.

Même si le code Java est conçu pour être exécuté sur différentes plates-formes, il ne peut, dans certains cas, se suffire à lui-même. Par exemple,

- La bibliothèque de classes standard Java ne supporte pas les fonctionnalités dépendantes de la plate-forme que requiert votre application.
- Vous voulez accéder à une bibliothèque existant dans un autre langage et la rendre accessible à votre code Java.
- Vous avez du code que vous voulez implémenter dans un programme de bas-niveau comme l'assembleur, puis le faire appeler par votre application Java.

L'interface JNI est une interface de programmation multiplate-forme standard incluse dans le JDK. Elle vous permet d'écrire des programmes Java qui peuvent fonctionner avec des applications et des bibliothèques écrites dans d'autres langages de programmation, comme le C, le C++ et l'assembleur.

Grâce à l'interface JNI, vous pouvez écrire des *méthode Java natives* pour créer, inspecter et mettre à jour des objets Java (dont les tableaux et les chaînes), appeler des méthodes Java, capturer et déclencher des

exceptions, charger des classes et obtenir des informations sur ces classes, et enfin, effectuer des vérifications de type à l'exécution.

En outre, vous pouvez utiliser l'API Invocation pour incorporer la JVM dans vos applications natives, puis utiliser le pointeur d'interface JNI pour accéder aux fonctionnalités de la machine virtuelle. Cela vous permet de rendre compatibles Java des applications existantes sans avoir à les lier au code source de la machine virtuelle.

Comment fonctionne l'interface JNI

Pour que Java atteigne son but principal, c'est-à-dire l'indépendance vis-à-vis des plates-formes, Sun n'a pas standardisé son implémentation de la machine virtuelle Java ; la société a voulu conserver à l'architecture de la JVM une certaine souplesse, qui permet aux fournisseurs de personnaliser leurs implémentations. Java reste indépendant des plates-formes, puisque chaque implémentation de la JVM doit se conformer à certaines règles impératives pour respecter l'indépendance vis-à-vis des plates-formes (comme la structure standard d'un fichier .class).

Avec ce scénario, le seul problème est un accès difficile aux bibliothèques natives à partir des apps Java, puisque le système d'exécution diffère selon les implémentations de la JVM. Pour cette raison, Sun propose la JNI comme moyen standard d'accès aux bibliothèques natives à partir des applications Java.

La façon d'accéder aux méthodes natives à partir des applications Java a changé dans le JDK 1.1. Dans l'ancien JDK, une classe Java pouvait accéder directement aux méthodes d'une bibliothèque native. La nouvelle implémentation utilise la JNI comme couche intermédiaire entre une classe Java et une bibliothèque native. Au lieu d'appeler directement les méthodes natives, la JVM utilise pour ses appels réels un pointeur vers la JNI. De cette façon, même si les implémentations de la JVM sont différentes, la couche utilisée pour accéder aux méthodes natives (la JNI) est toujours la même.

Utilisation du mot clé native

Rendre natives les méthodes Java est très simple. Voici un résumé des étapes nécessaires :

- 1 Supprimer le corps de la méthode.
- 2 Ajouter un point-virgule à la fin de la signature de la méthode.
- 3 Ajouter le mot clé **native** au début de la signature de la méthode.

- 4 Inclure le corps de la méthode dans une bibliothèque native à charger lors de l'exécution.

Considérons par exemple que la méthode suivante existe dans une classe Java :

```
public void nativeMethod () {
    //corps de la méthode
}
```

Voici comment rendre la méthode native :

```
public native void nativeMethod ();
```

Une fois la méthode déclarée comme native, son implémentation réelle sera incluse dans une bibliothèque native. C'est la classe à laquelle cette méthode appartient qui doit appeler la bibliothèque, ce qui rend son implémentation globalement disponible. Pour qu'une classe appelle la bibliothèque, le plus simple consiste à ajouter ce qui suit à la classe :

```
static
{
    System.loadLibrary (nomDeBibliothèque);
}
```

Un bloc de code **static** est toujours exécuté une fois, lors du premier chargement de la classe. Vous pouvez inclure pratiquement n'importe quoi dans un bloc de code **static**. Mais, son utilisation la plus commune est le chargement des bibliothèques. Si, pour une raison quelconque, le chargement de la bibliothèque échoue, une exception **UnsatisfiedLinkError** est émise dès qu'une méthode de cette bibliothèque est appelée. La JVM ajoutant la bonne extension à son nom (.dll dans Windows et .so dans UNIX), vous n'avez pas à la spécifier dans le nom de la bibliothèque.

Utilisation de l'outil javah

Le JDK fournit un outil appelé `jvah`, qui génère des fichiers d'en-tête C pour des classes Java. Voici la syntaxe générale d'utilisation de `jvah` :

```
jvah [options] nomClasse
```

`nomClasse` représente le nom de la classe (sans l'extension .class) pour laquelle vous voulez générer un fichier d'en-tête C. Vous pouvez préciser plusieurs classes sur une ligne de commande. Pour chaque classe, `jvah` ajoute par défaut un fichier .h au répertoire des classes. Si vous voulez que les fichiers .h soient placés ailleurs, utilisez l'option `-o`. Si une classe se trouve dans un paquet, précisez le paquet et le nom de la classe.

Par exemple, pour générer un fichier d'en-tête pour la classe `maClasse` du paquet `monPaquet`, procédez comme suit :

```
jvah monPaquet.maClasse
```

Le fichier d'en-tête généré inclura le nom du paquet, (`monPaquet_maClasse.h`).

Voici une liste de quelques options de `javah` :

Option	Description
<code>-jni</code>	Crée un fichier d'en-tête JNI
<code>-verbose</code>	Affiche des informations sur l'avancement des opérations
<code>-version</code>	Affiche la version de <code>javah</code>
<code>-o nomRépertoire</code>	Met le fichier <code>.h</code> résultant dans le répertoire indiqué
<code>-classpath chemin</code>	Redéfinit le chemin par défaut de la classe

Le contenu du fichier `.h` file généré par `javah` inclut tous les prototypes de fonctions pour les méthodes **native** de la classe. Les prototypes sont modifiés pour que le système d'exécution de Java trouve et appelle les méthodes natives. Il s'agit principalement d'un changement du nom de la méthode pour suivre la convention d'appel des méthodes natives. Le nom modifié est constitué par le préfixe `Java_` et les noms de la classe et de la méthode. Ainsi, si une méthode native appelée `méthodeNative` se trouve dans la classe `maClasse`, le nom qui figure dans le fichier `maClasse.h` est `Java_maClasse_méthodeNative`.

Pour plus d'informations sur JNI, voir :

- Java Native Interface (<http://java.sun.com/j2se/1.3/docs/guide/jni/>)
- Java Native Interface Specification (<http://java.sun.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html>)
- *Le tutorial Java, "Trail: Java Native Interface"* (<http://java.sun.com/docs/books/tutorial/native1.1/index.html>)

Les manuels suivants concernant l'interface JNI sont également disponibles :

- *The Java Native Interface: Programmer's Guide and Specification (Java Series)* de Sheng Liang
 - amazon.com (<http://www.amazon.com/exec/obidos/ASIN/0201325772/inprisecorporati/107-5674331-0009332>)
 - fatbrain.com (<http://www1.fatbrain.com/asp/bookinfo/bookinfo.asp?theisbn=0201325772&from=SWP279>)
- *Essential Jni: Java Native Interface (Essential Java)*, de Rob Gordon
 - amazon.com (<http://www.amazon.com/exec/obidos/ASIN/0136798950/inprisecorporati/107-5674331-0009332>)
 - fatbrain.com (<http://www1.fatbrain.com/asp/bookinfo/bookinfo.asp?theisbn=0136798950&from=SWP279>)

Référence rapide du langage Java

Editions de la plate-forme Java 2

La plate-forme Java 2 est disponible dans plusieurs éditions utilisées à des fins diverses. Comme Java est un langage qui peut s'exécuter partout et sur n'importe quelle plate-forme, il est utilisé dans divers environnements et a été livré dans plusieurs éditions : Java 2 Standard Edition (J2SE), Java 2 Enterprise Edition (J2EE) et Java 2 Micro Edition (J2ME). Dans certains cas, comme pour le développement d'applications d'entreprise, un ensemble important de paquets est utilisé. Dans d'autres cas, comme pour les produits d'électronique grand public, seule une petite partie du langage est utilisée. Chaque édition contient un kit de développement Java 2 (SDK) pour développer des applications et un environnement d'exécution Java 2 (JRE) pour les exécuter.

Tableau 11.1 Editions de la plate-forme Java 2

Plate-forme Java 2	Abréviation	Description
Standard Edition	J2SE	Contient les classes qui forment le cœur du langage Java.
Enterprise Edition	J2EE	Contient les classes J2SE plus d'autres classes pour le développement d'applications d'entreprise.
Micro Edition	J2ME	Contient un sous-ensemble des classes J2SE destiné aux produits électroniques grand public.

Bibliothèques des classes Java

Java, comme la plupart des langages de programmation, repose en grande partie sur des bibliothèques déjà construites pour prendre en charge certaines fonctionnalités. Dans Java, ces groupes de classes liées, que l'on appelle des paquets, varient selon l'édition du langage Java. Chaque édition est utilisée à des fins spécifiques, par exemple pour les applications, pour les applications d'entreprise et pour les produits grand public.

La plate-forme Java 2, Standard Edition (J2SE), fournit aux développeurs un environnement de développement riche en fonctionnalités, stable, sécurisé et multiplate-forme. Cette édition de Java prend en charge des fonctionnalités majeures comme la connectivité aux bases de données, la création d'interfaces utilisateur, les entrées/sorties, la programmation en réseau, et contient les paquets fondamentaux du langage Java. Certains de ces paquets J2SE figurent dans le tableau suivant.

Tableau 11.2 Paquets de J2SE

Paquet	Nom du paquet	Description
Langage	java.lang	Classes qui forment le cœur du langage Java.
Utilitaires	java.util	Prise en charge des structures de données utilitaires.
E/S	java.io	Prise en charge de divers types d'entrée/sortie
Texte	java.text	Support de localisation pour la gestion du texte, des dates, des nombres et des messages.
Math	java.math	Classes permettant des calculs en précision entière et en virgule flottante.
AWT	java.awt	Conception d'interfaces utilisateur et gestion des événements
Swing	javax.swing	Classes pour créer des composants légers tout-Java qui se comportent de la même façon sur toutes les plates-formes.
Javax	javax	Extensions du langage Java.
Applet	java.applet	Classes pour créer des applets.
Beans	java.beans	Classes pour développer des JavaBeans.
Réflexion	java.lang.reflect	Classes utilisées pour obtenir à l'exécution des informations sur les classes
SQL	java.sql	Accès aux données des bases de données et traitement.
RMI	java.rmi	Prise en charge de la programmation distribuée.
Réseau	java.net	Classes qui supportent le développement d'applications en réseau.
Sécurité	java.security	Prise en charge de la sécurité par cryptographie.

Mots clés

Ces tableaux couvrent les types de mots clés suivants :

- Types de données, types de retour et termes
- Paquets, classes, membres et interfaces
- Modificateurs d'accès
- Boucles et contrôles de boucles
- Gestion des exceptions
- Mots réservés

Types de données, types de retour et termes

Types et termes	Mot clé	Usage
Types numériques	byte	1 octet, entier.
	double	8 octets.
	float	4 octets.
	int	4 octets, entier.
	long	8 octets, entier.
	short	2 octets, entier.
	strictfp	(proposé) Méthode ou classe pour utiliser la précision standard dans les calculs intermédiaires en virgule flottante.
Autres types	booléen	Valeurs booléennes.
	char	16 bits, un caractère.
Termes de retour	return	Quitte le bloc de code en cours sans valeur résultante.
	void	Type de retour quand aucune valeur de retour n'est requise.

Paquets, classes, membres et interfaces

Mot clé	Usage
abstract	Cette méthode ou classe doit être étendue pour être utilisée.
class	Déclare une classe Java.
extends	Crée une sous-classe. Donne à une classe l'accès aux membres publics et protégés d'une autre classe. Permet à une interface d'hériter d'une autre.
final	Cette classe ne peut pas être étendue.
implements	Dans une définition de classe, implémente une interface définie.
import	Rend visibles par le programme en cours toutes les classes de la classe ou du paquet importé.

Mot clé	Usage
<code>instanceof</code>	Contrôle l'héritage d'un objet.
<code>interface</code>	Rend abstraite l'interface d'une classe depuis son implémentation (indique quoi faire, et non comment le faire).
<code>native</code>	Le corps de cette méthode est fourni par un lien vers une bibliothèque native.
<code>new</code>	Instancie une classe.
<code>package</code>	Déclare un nom de paquet pour toutes les classes définies dans les fichiers source avec la même déclaration de paquet.
<code>static</code>	Le membre est disponible pour la classe entière, et pas seulement pour un objet.
<code>super</code>	A l'intérieur d'une classe, fait référence à la superclasse.
<code>synchronized</code>	Rend un bloc de code sécurisé par rapport aux threads.
<code>this</code>	Fait référence à l'objet en cours.
<code>transient</code>	La valeur de cette variable n'est pas conservée quand l'objet est enregistré.
<code>volatile</code>	La valeur de cette variable peut changer de façon inattendue.

Modificateurs d'accès

Mot clé	Usage
<code>package</code>	Niveau d'accès par défaut ; ne l'utilisez pas de façon explicite. Ne peut pas être sous-classé par un autre paquet.
<code>private</code>	Accès limité à la propre classe du membre.
<code>protected</code>	Accès limité au paquet de la classe du membre.
<code>public</code>	Classe : accessible de partout. Sous-classe : accessible tant que sa classe l'est.

Boucles et contrôles de boucles

Type d'instruction	Mot clé
Instructions de sélection	<code>if</code>
	<code>else</code>
	<code>switch</code>
	<code>case</code>
Instruction de sortie de boucle	<code>break</code>
Instruction de repli	<code>default</code>
Instructions d'itération	<code>for</code>
	<code>do</code>
	<code>while</code>
	<code>continue</code>

Gestion des exceptions

Mot clé	Usage
<code>throw</code>	Passe le contrôle de la méthode au gestionnaire d'exception.
<code>throws</code>	Enumère les exceptions qu'une méthode peut déclencher.
<code>try</code>	Instruction d'ouverture d'un gestionnaire d'exception.
<code>catch</code>	Capture l'exception.
<code>finally</code>	Exécute son code avant l'achèvement du programme.

Réservés

Mot clé	Usage
<code>assert</code>	Réservé à un usage futur.
<code>const</code>	Réservé à un usage futur.
<code>goto</code>	Réservé à un usage futur.

Conversion et transtypage des types de données

Le type de données d'un objet ou d'une variable peut être changé en une seule opération lorsqu'un type différent est requis. Les conversions d'agrandissement (d'une classe ou d'un type de données plus petit vers un plus grand) peuvent être implicites, mais c'est une bonne habitude de les faire de manière explicite. Les conversions de raccourcissement doivent être explicites, on les appelle *transtypages*. Il vaut mieux que les programmeurs débutants évitent le transtypage ; il peut devenir une source inépuisable d'erreurs et de confusion.

Pour transtyper un type de données, placez le type *vers* lequel transtyper entre parenthèses, immédiatement avant la variable à transtyper : `(int)x`. Voici le cas où `x` est la variable à transtyper, `float` le type de données initial, `int` le type de destination et `y` est la variable contenant la nouvelle valeur :

```
float x = 1.00; //déclaration de x en float
int y = (int)x; //transtypage de x vers un int nommé y
```

Cela suppose que la valeur de `x` tiendra dans `int`. Notez que les décimales de `x` sont perdues dans la conversion. Java arrondit les décimales au nombre entier le plus proche.

Notez que les séquences Unicode peuvent représenter des chiffres, des lettres ou des caractères non imprimables comme la rupture de ligne ou la tabulation. Pour plus d'informations sur Unicode, voir <http://www.unicode.org/>

Cette section contient les tableaux des conversions suivantes :

- Primitif en primitif
- Primitif en chaîne
- Primitif en référence
- Chaîne en primitif
- Référence en primitif
- Référence en référence

Primitif en primitif

Java ne supporte pas le transtypage de ou vers les valeurs `boolean`. Pour contourner la gestion des types logiques de Java, vous devez affecter à la variable une valeur équivalente appropriée et la convertir ensuite. 0 et 1 sont souvent utilisés pour représenter les valeurs `false` et `true`.

Syntaxe	Commentaires
D'un autre type primitif p En <code>boolean t</code> : <code>t = p != 0;</code>	Les autres types primitifs sont <code>byte</code> , <code>short</code> , <code>char</code> , <code>int</code> , <code>long</code> , <code>double</code> , <code>float</code> .
De <code>boolean t</code> En <code>byte b</code> : <code>b = (byte)(t ? 1 : 0);</code>	
De <code>boolean t</code> En <code>int</code> , <code>long</code> , <code>double</code> , ou <code>float m</code> : <code>m = t ? 1 : 0;</code>	
De <code>boolean t</code> En <code>short s</code> : <code>s = (short) (t ? 1 : 0);</code>	
De <code>boolean t</code> En <code>byte b</code> : <code>b = (byte) (t?1:0);</code>	
De <code>boolean t</code> En <code>char c</code> : <code>c = (char) (t?'1':'0');</code>	Vous pouvez omettre les apostrophes, mais elles sont recommandées.
De <code>short</code> , <code>char</code> , <code>int</code> , <code>long</code> , <code>double</code> ou <code>float n</code> En <code>byte b</code> : <code>b = (byte)n;</code>	
De <code>byte b</code> En <code>short</code> , <code>int</code> , <code>long</code> , <code>double</code> ou <code>float n</code> : <code>n = b;</code>	
De <code>byte b</code> En <code>char c</code> : <code>c = (char)b;</code>	

Primitif en chaîne

Les types de données primitifs sont muables ; les types de référence sont des objets immuables. Le transtypage de ou vers un type de référence est dangereux.

Java ne supporte pas le transtypage de ou vers les valeurs `boolean`. Pour contourner la gestion des types logiques de Java, vous devez affecter à la variable une valeur équivalente appropriée et la convertir ensuite. 0 et 1 sont souvent utilisés pour représenter les valeurs `false` et `true`.

Syntaxe	Commentaires
De <code>boolean t</code> En <code>String gg</code> : <code>gg = t ? "true" : "false";</code>	
De <code>byte b</code> En <code>String gg</code> : <code>gg = Integer.toString(b);</code> ou <code>gg = String.valueOf(b);</code>	Ce qui suit peut remplacer <code>toString</code> , le cas échéant : <code>toBinaryString</code> <code>toOctalString</code> <code>toHexString</code> Où vous utilisez une base autre que 10 ou 2 (comme 8) : <code>gg = Integer.toString(b, 7);</code>
De <code>short</code> ou <code>int n</code> En <code>String gg</code> : <code>gg = Integer.toString(n);</code> ou <code>gg = String.valueOf(n);</code>	Ce qui suit peut remplacer <code>toString</code> , le cas échéant : <code>toBinaryString</code> <code>toOctalString</code> <code>toHexString</code> Où vous utilisez une base autre que 10 (comme 8) : <code>gg = Integer.toString(n, 7);</code>
De <code>char c</code> En <code>String gg</code> : <code>gg = String.valueOf(c);</code>	
De <code>long n</code> En <code>String gg</code> : <code>g = Long.toString(n);</code> ou <code>gg = String.valueOf(n);</code>	Ce qui suit peut remplacer <code>toString</code> , le cas échéant : <code>toBinaryString</code> <code>toOctalString</code> <code>toHexString</code> Où vous utilisez une base autre que 10 ou 2 (comme 8) : <code>gg = Integer.toString(n, 7);</code>

Syntaxe	Commentaires
De float f En String gg : gg = Float.toString(f); ou gg = String.valueOf(f); Pour la protection des décimales ou la notation scientifique, voir la colonne suivante.	Ces transtypes protègent davantage de données. Deux décimales : java.text.DecimalFormat df2 = new java.text.DecimalFormat("###,##0.00"); gg = df2.format(f); Notation scientifique (protège les exposants) (JDK 1.2.x et versions ultérieures) : java.text.DecimalFormat de = new java.text.DecimalFormat("0.000000E00"); gg = de.format(f);
De double d En String gg : gg = Double.toString(d); ou gg = String.valueOf(d); Pour la protection des décimales ou la notation scientifique, voir la colonne suivante.	Ces transtypes protègent davantage de données. Deux décimales : java.text.DecimalFormat df2 = new java.text.DecimalFormat("###,##0.00"); gg = df2.format(d); Notation scientifique (JDK 1.2.x et versions ultérieures) : java.text.DecimalFormat de = new java.text.DecimalFormat("0.000000E00"); gg = de.format(d);

Primitif en référence

Java fournit des classes qui correspondent aux types de données primitifs et possèdent des méthodes pour faciliter les conversions.

Notez que les types de données primitifs sont muables ; les types de référence sont des objets immuables. Le transtypage de ou vers un type de référence est dangereux.

Java ne supporte pas le transtypage de ou vers les valeurs `boolean`. Pour contourner la gestion des types logiques de Java, vous devez affecter à la variable une valeur équivalente appropriée et la convertir ensuite. 0 et 1 sont souvent utilisés pour représenter les valeurs `false` et `true`.

Syntaxe	Commentaires
De boolean t En Boolean tt : tt = new Boolean(t);	
De type primitif p (autre que <code>boolean</code>) En Boolean tt : tt = new Boolean(p != 0); Pour char, voir la colonne suivante.	Pour char c, placez des apostrophes autour du zéro : tt = new Boolean(c != '0');

Syntaxe	Commentaires
De boolean t En Character cc : cc = new Character(t ? '1' : '0');	
De byte b En Character cc : cc = new Character((char) b);	
De char c En Character cc : cc = new Character(c);	
De short, int, long, float, ou double n En Character cc : cc = new Character((char)n);	
De boolean t En Integer ii : ii = new Integer(t ? 1 : 0);	
De byte b En Integer ii : ii = new Integer(b);	
De short, char ou int n En Integer ii : ii = new Integer(n);	
De long, float ou double f En Integer ii : ii = new Integer((int) f);	
De boolean t En Long nn : nn = new Long(t ? 1 : 0);	
De byte b En Long nn : nn = new Long(b);	
De short, char, int ou long s En Long nn : nn = new Long(s);	
De float, double f En Long nn : nn = new Long((long) f);	
De boolean t En Float ff : ff = new Float(t ? 1 : 0);	
De byte b En Float ff : ff = new Float(b);	

Syntaxe	Commentaires
De short, char, int, long, float ou double n En Float ff: <pre>ff = new Float(n);</pre>	
De boolean t En Double dd: <pre>dd = new Double(t ? 1 : 0);</pre>	
De byte b En Double dd: <pre>dd = new Double(b);</pre>	
De short, char, int, long, float ou double n En Double dd: <pre>dd = new Double(n);</pre>	

Chaîne en primitif

Notez que les types de données primitifs sont muables ; les types de référence sont des objets immuables. Le transtypage de ou vers un type de référence est dangereux.

Java ne supporte pas le transtypage de ou vers les valeurs `boolean`. Pour contourner la gestion des types logiques de Java, vous devez affecter à la variable une valeur équivalente appropriée et la convertir ensuite. Les nombres 0 et 1, les chaînes “true” et “false” ou les valeurs intuitives d’égalité sont utilisés ici pour représenter les valeurs `true` et `false`.

Syntaxe	Commentaires
De String gg En boolean t: <pre>t = new Boolean(gg.trim()).booleanValue();</pre>	Attention : t sera <code>true</code> seulement quand la valeur de gg sera “true” (majuscules/minuscules indifférentes) ; si la chaîne vaut “1”, “oui” ou tout autre affirmation, cette conversion renverra une valeur <code>false</code> .
De String gg En byte b: <pre>try { b = (byte)Integer.parseInt(gg.trim()); } catch (NumberFormatException e) { ... }</pre>	Remarque : Si la valeur de gg est null, <code>trim()</code> déclenchera une <code>NullPointerException</code> . Si vous n’utilisez pas <code>trim()</code> , vérifiez qu’il n’y ait pas d’espaces à droite. Pour des bases autres que 10, comme 8 : <pre>try { b = (byte)Integer.parseInt(gg.trim(), 7); } catch (NumberFormatException e) { ... }</pre>

Syntaxe	Commentaires
De String gg En short s : <pre> try { s = (short)Integer.parseInt(gg.trim()); } catch (NumberFormatException e) { ... } </pre>	<p>Remarque : Si la valeur de gg est null, trim() déclenchera une NullPointerException. Si vous n'utilisez pas trim(), vérifiez qu'il n'y ait pas d'espaces à droite.</p> <p>Pour des bases autres que 10, comme 8 :</p> <pre> try { s = (short)Integer.parseInt(gg.trim(), 7); } catch (NumberFormatException e) { ... } </pre>
De String gg En char c : <pre> try { c = (char)Integer.parseInt(gg.trim()); } catch (NumberFormatException e) { ... } </pre>	<p>Remarque : Si la valeur de gg est null, trim() déclenchera une NullPointerException. Si vous n'utilisez pas trim(), vérifiez qu'il n'y ait pas d'espaces à droite.</p> <p>Pour des bases autres que 10, comme 8 :</p> <pre> try { c = (char)Integer.parseInt(gg.trim(), 7); } catch (NumberFormatException e) { ... } </pre>
De String gg En int i : <pre> try { i = Integer.parseInt(gg.trim()); } catch (NumberFormatException e) { ... } </pre>	<p>Remarque : Si la valeur de gg est null, trim() déclenchera une NullPointerException. Si vous n'utilisez pas trim(), vérifiez qu'il n'y ait pas d'espaces à droite.</p> <p>Pour des bases autres que 10, comme 8 :</p> <pre> try { i = Integer.parseInt(gg.trim(), 7); } catch (NumberFormatException e) { ... } </pre>
De String gg En long n : <pre> try { n = Long.parseLong(gg.trim()); } catch (NumberFormatException e) { ... } </pre>	<p>Remarque : Si la valeur de gg est null, trim() déclenchera une NullPointerException. Si vous n'utilisez pas trim(), vérifiez qu'il n'y ait pas d'espaces à droite.</p>
De String gg En float f : <pre> try { f = Float.valueOf(gg.trim()).floatValue; } catch (NumberFormatException e) { ... } </pre>	<p>Remarque : Si la valeur de gg est null, trim() déclenchera une NullPointerException. Si vous n'utilisez pas trim(), vérifiez qu'il n'y ait pas d'espaces à droite.</p> <p>Pour le JDK 1.2.x ou les versions ultérieures :</p> <pre> try { f = Float.parseFloat(gg.trim()); } catch (NumberFormatException e) { ... } </pre>

Syntaxe	Commentaires
De String gg En double d : <pre> try { d = Double.valueOf(gg.trim()).doubleValue; } catch (NumberFormatException e) { ... } </pre>	Remarque : Si la valeur de gg est null, trim() déclenchera une NullPointerException. Si vous n'utilisez pas trim(), vérifiez qu'il n'y ait pas d'espaces à droite. Pour le JDK 1.2.x ou les versions ultérieures : <pre> try { d = Double.parseDouble(gg.trim()); } catch (NumberFormatException e) { ... } </pre>

Référence en primitif

Java fournit des classes qui correspondent aux types de données primitifs. Ce tableau montre comment convertir une variable depuis une de ces classes en un type de données primitif, en une seule opération.

Pour convertir un type de référence en type primitif, vous devez d'abord obtenir la valeur de la référence sous forme de primitif, puis transtyper le primitif.

Les types de données primitifs sont muables ; les types de référence sont des objets immuables. La conversion de ou vers un type de référence est dangereuse.

Java ne supporte pas le transtypage de ou vers les valeurs boolean. Pour contourner la gestion des types logiques de Java, vous devez affecter à la variable une valeur équivalente appropriée et la convertir ensuite. 0 et 1 sont souvent utilisés pour représenter les valeurs false et true.

Syntaxe	Commentaires
De Boolean tt En boolean t : <pre> t = tt.booleanValue(); </pre>	
De Boolean tt En byte b : <pre> b = (byte)(tt.booleanValue() ? 1 : 0); </pre>	
De Boolean tt En short s : <pre> s = (short)(tt.booleanValue() ? 1 : 0); </pre>	
De Boolean tt En char c : <pre> c = (char)(tt.booleanValue() ? '1' : '0'); </pre>	Vous pouvez omettre les apostrophes, mais elles sont recommandées.
De Boolean tt En int, long, float ou double n : <pre> n = tt.booleanValue() ? 1 : 0; </pre>	

Syntaxe	Commentaires
De Character cc En boolean t : t = cc.charValue() != 0;	
De Character cc En byte b : b = (byte)cc.charValue();	
De Character cc En short s : s = (short)cc.charValue();	
De Character cc En char, int, long, float ou double n : n = cc.charValue();	
De Integer ii En boolean t : t = ii.intValue() != 0;	
De Integer ii En byte b : b = ii.byteValue();	
De Integer, Long, Float ou Double nn En short s : s = nn.shortValue();	
De Integer, Long, Float ou Double nn En char c : c = (char)nn.intValue();	
De Integer, Long, Float ou Double nn En int i : i = nn.intValue();	
De Integer ii En long n : n = ii.longValue();	
De Long, Float ou Double dd En long n : n = dd.longValue();	
De Integer, Long, Float ou Double nn En float f : f = nn.floatValue();	
De Integer, Long, Float ou Double nn En double d : d = nn.doubleValue();	

Référence en référence

Java fournit des classes qui correspondent aux types de données primitifs. Ce tableau montre comment convertir une variable depuis une de ces classes en une autre, en une seule opération.

Remarque Pour des conversions de classe en classe correctes autres que celles indiquées ici, les conversions d'agrandissement sont implicites. Les transtypes de raccourcissement utilisent cette syntaxe :

```
nomObjetATranstyper = (ClasseObjetVersQuiTranstyper)nomObjetVersQuiTranstyper;
```

Vous devez utiliser le transtypage entre les classes se trouvant dans la même hiérarchie d'héritage. Si vous transtypez un objet en une classe incompatible, cela déclenchera une `ClassCastException`.

Les types de référence sont des objets immuables. La conversion entre type de référence est dangereuse.

Syntaxe	Commentaires
De String gg En Boolean tt: <pre>tt = new Boolean(gg.trim());</pre>	Remarque : Si la valeur de gg est null, trim() déclenchera une <code>NullPointerException</code> . Si vous n'utilisez pas trim(), vérifiez qu'il n'y ait pas d'espaces à droite. Alternative : <pre>tt = Boolean.valueOf(gg.trim());</pre>
De String gg En Character cc: <pre>cc = new Character(g.charAt(<index>));</pre>	
De String gg En Integer ii: <pre>try { ii = new Integer(gg.trim()); } catch (NumberFormatException e) { ... }</pre>	Remarque : Si la valeur de gg est null, trim() déclenchera une <code>NullPointerException</code> . Si vous n'utilisez pas trim(), vérifiez qu'il n'y ait pas d'espaces à droite. Alternative : <pre>try { ii = Integer.valueOf(gg.trim()); } catch (NumberFormatException e) { ... }</pre>
De String gg En Long nn: <pre>try { nn = new Long(gg.trim()); } catch (NumberFormatException e) { ... }</pre>	Remarque : Si la valeur de gg est null, trim() déclenchera une <code>NullPointerException</code> . Si vous n'utilisez pas trim(), vérifiez qu'il n'y ait pas d'espaces à droite. Alternative : <pre>try { nn = Long.valueOf(gg.trim()); } catch (NumberFormatException e) { ... }</pre>

Syntaxe	Commentaires
De String gg En Float ff : <pre> try { ff = new Float(gg.trim()); } catch (NumberFormatException e) { ... } </pre>	Remarque : Si la valeur de gg est null, trim() déclenchera une NullPointerException. Si vous n'utilisez pas trim(), vérifiez qu'il n'y ait pas d'espaces à droite. Alternative : <pre> try { ff = Float.valueOf(gg.trim()); } catch ... } </pre>
De String gg En Double dd : <pre> try { dd = new Double(gg.trim()); } catch ... } </pre>	Remarque : Si la valeur de gg est null, trim() déclenchera une NullPointerException. Si vous n'utilisez pas trim(), vérifiez qu'il n'y ait pas d'espaces à droite. Alternative : <pre> try { dd = Double.valueOf(gg.trim()); } catch (NumberFormatException e) { ... } </pre>
De Boolean tt En Character cc : <pre> cc = new Character(tt.booleanValue() ? '1' : '0'); </pre>	
De Boolean tt En Integer ii : <pre> ii = new Integer(tt.booleanValue() ? 1 : 0); </pre>	
De Boolean tt En Long nn : <pre> nn = new Long(tt.booleanValue() ? 1 : 0); </pre>	
De Boolean tt En Float ff : <pre> ff = new Float(tt.booleanValue() ? 1 : 0); </pre>	
De Boolean tt En Double dd : <pre> dd = new Double(tt.booleanValue() ? 1 : 0); </pre>	
De Character cc En Boolean tt : <pre> tt = new Boolean(cc.charValue() != '0'); </pre>	
De Character cc En Integer ii : <pre> ii = new Integer(cc.charValue()); </pre>	
De Character cc En Long nn : <pre> nn = new Long(cc.charValue()); </pre>	

Syntaxe	Commentaires
De toute classe <code>rr</code> En <code>String gg</code> : <pre>gg = rr.toString();</pre>	
De <code>Float ff</code> En <code>String gg</code> : <pre>gg = ff.toString();</pre>	Ces variantes protègent davantage de données. Deux décimales : <pre>java.text.DecimalFormat df2 = new java.text.DecimalFormat("###,##0.00"); gg = df2.format(ff.floatValue());</pre> Notation scientifique (JDK 1.2.x et versions ultérieures) : <pre>java.text.DecimalFormat de = new java.text.DecimalFormat("0.000000E00"); gg = de.format(ff.floatValue());</pre>
De <code>Double dd</code> En <code>String gg</code> : <pre>gg = dd.toString();</pre>	Ces variantes protègent davantage de données. Deux décimales : <pre>java.text.DecimalFormat df2 = new java.text.DecimalFormat("###,##0.00"); gg = df2.format(dd.doubleValue());</pre> Notation scientifique (JDK 1.2.x et versions ultérieures) : <pre>java.text.DecimalFormat de = new java.text.DecimalFormat("0.0000000000E00"); gg = de.format(dd.doubleValue());</pre>
De <code>Integer ii</code> En <code>Boolean tt</code> : <pre>tt = new Boolean(ii.intValue() != 0);</pre>	
De <code>Integer ii</code> En <code>Character cc</code> : <pre>cc = new Character((char)ii.intValue());</pre>	
De <code>Integer ii</code> En <code>Long nn</code> : <pre>nn = new Long(ii.intValue());</pre>	
De <code>Integer ii</code> En <code>Float ff</code> : <pre>ff = new Float(ii.intValue());</pre>	
De <code>Integer ii</code> En <code>Double dd</code> : <pre>dd = new Double(ii.intValue());</pre>	
De <code>Long nn</code> En <code>Boolean tt</code> : <pre>tt = new Boolean(nn.longValue() != 0);</pre>	
De <code>Long nn</code> En <code>Character cc</code> : <pre>cc = new Character((char)nn.intValue());</pre>	Remarque : Certaines valeurs Unicode peuvent être rendues en caractères non imprimables. Consultez www.unicode.org/

Syntaxe	Commentaires
De Long nn En Integer ii : ii = new Integer(nn.intValue());	
De Long nn En Float ff : ff = new Float(nn.longValue());	
De Long nn En Double dd : dd = new Double(nn.longValue());	
De Float ff En Boolean tt : tt = new Boolean(ff.floatValue() != 0);	
De Float ff En Character cc : cc = new Character((char)ff.intValue());	Remarque : Certaines valeurs Unicode peuvent être rendues en caractères non imprimables. Consultez www.unicode.org/
De Float ff En Integer ii : ii = new Integer(ff.intValue());	
De Float ff En Long nn : nn = new Long(ff.longValue());	
De Float ff En Double dd : dd = new Double(ff.floatValue());	
De Double dd En Boolean tt : tt = new Boolean(dd.doubleValue() != 0);	
De Double dd En Character cc : cc = new Character((char)dd.intValue());	Remarque : Certaines valeurs Unicode peuvent être rendues en caractères non imprimables. Consultez www.unicode.org/
De Double dd En Integer ii : ii = new Integer(dd.intValue());	
De Double dd En Long nn : nn = new Long(dd.longValue());	
De Double dd En Float ff : ff = new Float(dd.floatValue());	

Séquences d'échappement

Un caractère octal est représenté par une suite de trois chiffres octaux ; un caractère Unicode est représenté par une suite de quatre chiffres hexadécimaux. Les caractères octaux sont précédés de la marque d'échappement standard, `\`, et les caractères Unicode sont précédés de `\u`. Par exemple, le nombre décimal 57 est représenté par le code octal `\071` et par la séquence Unicode `\u0039`. Les séquences Unicode peuvent représenter des chiffres, des lettres ou des caractères non imprimables comme la rupture de ligne ou la tabulation. Pour plus d'informations sur Unicode, voir <http://www.unicode.org/>

Caractère	Séquence d'échappement
Barre oblique inversée	<code>\\</code>
Retour arrière	<code>\b</code>
Retour chariot	<code>\r</code>
Guillemet	<code>\"</code>
Saut de page	<code>\f</code>
Tabulation horizontale	<code>\t</code>
Nouvelle ligne	<code>\n</code>
Caractère octal	<code>\DDD</code>
Apostrophe	<code>\'</code>
Caractère Unicode	<code>\uHHHH</code>

Opérateurs

Cette section énumère les éléments suivants :

- Opérateurs de base
- Opérateurs arithmétiques
- Opérateurs logiques
- Opérateurs d'affectation
- Opérateurs de comparaison
- Opérateurs au niveau bits
- Opérateur ternaire

Opérateurs de base

Opérateur	Opérande	Comportement
<code>.</code>	membre objet	Accède à un membre d'un objet.
<code>(<type>)</code>	type de données	Transtype une variable en un type de données différent. ¹

Opérateur	Opérande	Comportement
+	chaîne	Joint des chaînes (concaténation).
	nombre	Additionne.
-	nombre	C'est le moins unaire ² (inverse le signe du nombre).
	nombre	Soustrait.
!	booléen	C'est l'opérateur <code>boolean</code> NOT.
&	entier, booléen	C'est à la fois l'opérateur au niveau bits (entier) et <code>boolean</code> AND. Quand il est doublé (&&), c'est le AND conditionnel <code>boolean</code> .
=	la plupart des éléments avec des variables	Affecte un élément à un autre élément (par exemple, une valeur à une variable, ou une classe à une instance). Il peut être combiné à d'autres opérateurs pour effectuer une opération et affecter la valeur résultante. Par exemple, += ajoute la valeur de gauche à celle de droite, puis affecte la nouvelle valeur au côté droit de l'expression.

1. Il est important de faire la distinction entre un opérateur et un délimiteur. Les parenthèses sont utilisées autour des arguments (par exemple) en tant que délimiteurs marquant les arguments dans l'instruction. Elles sont utilisées autour d'un type de données en tant qu'opérateur changeant le type de données d'une variable en celui qui est à l'intérieur des parenthèses.
2. Un *opérateur unaire* affecte un seul opérande, un *opérateur binaire* affecte deux opérandes et un *opérateur ternaire* affecte trois opérandes.

Opérateurs arithmétiques

Opérateur	Priorité	Associativité	Définition
++/--	1	Droite	Incréméntation/décréméntation automatique : Ajoute un à ou soustrait un de son opérande unique. Si la valeur de <code>i</code> est 4, <code>++i</code> vaut 5. Une <i>pré-incréméntation</i> (<code>++i</code>) incrémente de un la valeur et affecte la nouvelle valeur à la variable. Une <i>post-incréméntation</i> (<code>i++</code>) incrémente la valeur mais laisse la variable à sa valeur initiale.
+/-	2	Droite	Plus/moins unaire : définit ou modifie la valeur positive/négative d'un seul nombre.
*	4	Gauche	Multiplication.
/	4	Gauche	Division.
%	4	Gauche	Modulo : Divise le premier opérande par le second et renvoie le reste (pas le résultat).
+/-	5	Gauche	Addition/soustraction

Opérateurs logiques

Opérateur	Priorité	Associativité	Définition
!	2	Droite	NOT booléen (unaire) Change <code>true</code> en <code>false</code> ou <code>false</code> en <code>true</code> . En raison de sa priorité basse, vous devez inclure cette instruction entre parenthèses.
&	9	Gauche	AND évaluation (binaire) Renvoie <code>true</code> seulement si les deux opérandes valent <code>true</code> . Evalue toujours deux opérandes.
^	10	Gauche	XOR évaluation (binaire) Renvoie <code>true</code> si un des deux opérandes seulement vaut <code>true</code> . Evalue deux opérandes.
	11	Gauche	OR évaluation (binaire) Renvoie <code>true</code> si un ou les deux opérandes valent <code>true</code> . Evalue deux opérandes.
&&	12	Gauche	AND conditionnel (binaire) Renvoie <code>true</code> seulement si les deux opérandes valent <code>true</code> . Il est dit “conditionnel” car il n’évalue le second opérande que si le premier vaut <code>true</code> .
	13	Gauche	OR conditionnel (binaire) Renvoie <code>true</code> si un ou les deux opérandes valent <code>true</code> ; renvoie <code>false</code> si les deux valent <code>false</code> . Le second opérande n’est pas évalué si le premier vaut <code>true</code> .

Opérateurs d’affectation

Opérateur	Priorité	Associativité	Définition
=	15	Droite	Affecte la valeur de droite à la variable de gauche.
+=	15	Droite	Ajoute la valeur de droite à la valeur de la variable de gauche ; affecte la nouvelle valeur à la variable initiale.
-=	15	Droite	Soustrait la valeur de droite de la valeur de la variable de gauche ; affecte la nouvelle valeur à la variable initiale.
*=	15	Droite	Multiplie la valeur de droite avec la valeur de la variable de gauche ; affecte la nouvelle valeur à la variable initiale.
/=	15	Droite	Divise la valeur de la variable de gauche par la valeur de droite ; affecte la nouvelle valeur à la variable initiale.

Opérateurs de comparaison

Opérateur	Priorité	Associativité	Définition
<	7	Gauche	Inférieur à
>	7	Gauche	Supérieur à
<=	7	Gauche	Inférieur ou égal à
>=	7	Gauche	Supérieur ou égal à
==	8	Gauche	Egal à
!=	8	Gauche	Différent de

Opérateurs au niveau bits

Remarque Dans un nombre entier signé, le bit le plus à gauche indique le signe positif ou négatif du nombre entier : le bit a la valeur 1 si l'entier est négatif, 0 s'il est positif. Dans Java, les nombres entiers sont toujours signés, alors que dans C/C++, ils sont signés par défaut. Cependant, dans Java, les opérateurs de décalage conservent le bit du signe, de sorte que le bit du signe est dupliqué, puis décalé. Par exemple, décaler à droite 10010011 de 1 donne 11001001.

Opérateur	Priorité	Associativité	Définition
~	2	Droite	NOT au niveau bits Inverse chaque bit de l'opérande, 0 devient 1 et réciproquement.
<<	6	Gauche	Décalage gauche signé Décale à gauche les bits de l'opérande gauche, du nombre de chiffres spécifié dans l'opérande droit, complète la droite par des 0. Les bits de poids fort sont perdus.
>>	6	Gauche	Décalage droit signé Décale à droite les bits de l'opérande gauche, du nombre de chiffres spécifié dans l'opérande droit. Si l'opérande gauche est négatif, la partie gauche est complétée par des 0 ; s'il est positif, elle est complétée par des 1. Cela préserve le signe initial.
>>>	6	Gauche	Décalage droit par ajout de zéros Décale à droite et remplit toujours par des 0.
&	9	Gauche	AND au niveau bits Peut être utilisé avec = pour affecter la valeur.

Opérateur	Priorité	Associativité	Définition
	10	Gauche	OR au niveau bits Peut être utilisé avec = pour affecter la valeur.
^	11	Gauche	XOR au niveau bits Peut être utilisé avec = pour affecter la valeur.
<<=	15	Gauche	Décalage gauche avec affectation
>>=	15	Gauche	Décalage droit avec affectation
>>>=	15	Gauche	Décalage droit par ajout de zéros avec affectation

Opérateur ternaire

L'opérateur ternaire `?:` effectue une opération if-then-else très simple à l'intérieur d'une seule instruction. Si le premier terme est vrai, il évalue le deuxième ; si le deuxième est faux, il utilise le troisième. En voici la syntaxe :

```
<expression 1> ? <expression 2> : <expression 3>;
```

Par exemple :

```
int x = 3, y = 4, max;
max = (x > y) ? x : y;
```

Cette opération affecte à `max` la valeur de `x` ou de `y`, selon celui qui est le plus grand.

Index

Symboles

. (point), opérateur 6-2
?: opérateur 3-3
syntaxe 11-22

A

abstract, mot clé 11-3
accès aux membres 3-16
AccessController, classe 9-4
allocation de mémoire
 obtenir un StringBuffer 5-15
appel des méthodes 6-3
applications
 exemple de développement 6-4
arrêt d'un thread 7-7

B

bibliothèques
 accès aux natives 10-2
 classes Java 5-1
 static, blocs de code 10-3
bibliothèques de classes
 présentation 5-1
Bibliothèques des classes Java
 présentation 5-1
bits
 décalage, signé et non signé 3-11
blocs de code
 définition 3-5
 static 10-3
boolean, mot clé 11-3
Borland
 contacter 1-6
Borland Online 1-6
boucles
 achèvement 4-10
 contrôle de l'exécution 4-11
 instructions conditionnelles 4-12
 mots clés, tableau 11-4
boucles do
 utilisation 4-10
boucles for
 utilisation 4-10
boucles while
 utilisation 4-9
boucles, utilisation 4-9
do 4-10
for 4-10

if-else 4-12
switch 4-13
while 4-9
break, mot clé 11-4
BufferedOutputStream, classe
 présentation 5-23
byte, mot clé 11-3
bytecodes 9-1
 traduction en instructions natives 9-7
 violations 9-3

C

caractères de contrôle 4-4
 Voir aussi séquences d'échappement
caractères non imprimables 4-4
 Voir aussi séquences d'échappement
case, mot clé 11-4
catch, mot clé 11-5
chaînes 4-1
 construction 5-12
 gestion 4-1, 4-5
 manipulation 4-1
char, mot clé 11-3
chargeur de classe 9-6
checkPermission() 9-4
checkRead() 9-5
checkWrite() 9-5
class, mot clé 11-3
classes
 accès aux membres 6-12
 définition 6-2
 enveloppe de type 5-11
 implémentation des interfaces 6-18
 objets et 6-2
classes abstraites 6-16
classes d'enveloppe de type 5-11
classes de fichiers 5-25
 RandomAccessFile, classe 5-26
classes de flux d'entrée 5-19
 FileInputStream 5-20
 InputStream 5-20
classes de flux de sortie 5-22
 BufferedOutputStream 5-23
 DataOutputStream 5-24
 FileOutputStream 5-24
 FileOutputStream 8-4
 OutputStream 5-22
 PrintStream 5-23
classes enfant 6-9
classes parent 6-9

- classes utilitaires
 - présentation 5-4
- classes, définitions 6-2
 - regroupement 6-25
- ClassLoader, classe 9-6
- ClassNotFoundException, exception 8-7
- code
 - commentaires 3-3
 - réutilisation 6-25
- code source
 - réutilisation 6-25
- commentaires 3-3
- compilateurs
 - just-in-time (JIT) 9-7
- conditions de test, annulation 4-11
- constructeurs 6-4
 - appel du parent 6-12
 - multiple 6-12
 - superclasses 6-12
 - syntaxe 3-15
 - utilisation 3-15
- constructeurs des threads 7-5
- contacter Borland 1-6
 - groupes de discussion 1-7
 - World Wide Web 1-6
- continue, mot clé 11-4
- contrôle du déroulement
 - définition 4-3
 - utilisation 4-9
- contrôles de boucles
 - instructions break 4-11
 - instructions continue 4-11
- conventions de la documentation 1-3
 - relatives au Macintosh 1-5
- conversions
 - chaîne en primitif 11-10
 - primitif en chaîne 11-7
 - primitif en primitif 11-6
 - référence en primitif 11-12
 - référence en référence 11-14
 - tableaux 11-5
 - types primitifs en types de référence 11-8
- conversions d'agrandissement
 - tableau 4-2
- conversions de types 4-2
 - agrandissement, tableau 4-2
 - raccourcissement explicite 4-8
 - tableaux 11-5
 - transtypage implicite 4-8
- création d'un objet 3-15
- création d'un thread 7-5

D

- DataOutputStream, classe
 - présentation 5-24

- décalages au niveau bits 3-11
- déclaration d'une variable 2-5
- déclaration des classes 6-2
- déclaration des paquets 6-25
- default, mot clé 6-12, 11-4
- définition de la priorité d'un thread 7-7
- définition des classes 6-2
- démarrage d'un thread 7-5
- désérialisation
 - définition 8-1
 - exemple 8-6
- désérialisation des objets 8-1
- développement des applications 6-4
- do, mot clé 11-4
- données membre 6-3
 - accès 6-12
- double, mot clé 11-3

E

- écriture dans des flux fichiers 8-4
- écriture des flux d'objets 8-8
- éditions Java 5-1
 - présentation 11-1
 - tableau 5-1, 11-1
- else, mot clé 11-4
- enregistrement des objets 8-1
- Enterprise Edition 5-2
- en-tête C, fichiers 10-3
- entrée/sortie des fichiers 5-25
- Enumeration, interface
 - présentation 5-16
- enveloppe, classes 5-11
- environnement d'exécution Java 9-2
 - Voir aussi JRE*
- exceptions
 - blocs catch 4-14
 - blocs finally 4-15
 - blocs try 4-14
 - instructions 4-14
 - throw, mot clé 4-15
 - throws, mot clé 4-15
- exceptions, gestion 4-14
- expert interface 6-18
- extends, mot clé 6-9, 11-3
- Externalizable, interface 8-8

F

- fichiers classe
 - compilation 9-1
 - structure 9-3
- fichiers d'en-tête 10-3
- File, classe
 - présentation 5-25

- FileInputStream, classe 8-6
 - présentation 5-20
- FileOutputStream, classe 8-4
 - présentation 5-24
- final, mot clé 11-3
- finaliseurs 6-4
- finally, mot clé 11-5
- float, mot clé 11-3
- flush() 8-4
- flux 8-4, 8-6
 - flux d'entrée 5-19
 - flux de sortie 5-22
 - lectures/écritures 8-8
 - partitionnement en jetons 5-27
- flux d'entrée 8-6
- flux d'objets
 - lectures/écritures 8-8
- fonctions 2-6
 - Voir aussi* méthodes
- fonctions mathématiques 5-12
- for, mot clé 11-4

G

- gestion de la mémoire
 - rôle de JVM 9-2
- gestion des exceptions 4-14
 - Voir aussi* exceptions
 - définition 4-3
 - mots clés, tableau 11-5
- gestionnaire de sécurité 9-4
- getter, méthodes 6-13
- groupes de discussion 1-7
 - Borland 1-7
- groupes de threads 7-8

H

- héritage 6-9
 - multiple 6-11
 - unique 6-11
- héritage de classe 6-9
- héritage multiple 6-11
 - remplacé par des interfaces 6-18
- héritage unique 6-11

I

- identificateurs
 - définition 2-1
- if, mot clé 11-4
- implements, mot clé 6-18, 11-3
- import, mot clé 11-3
- indépendance des plates-formes 10-2
- InputStream, classe
 - présentation 5-20

- instanceof, mot clé 11-3
- instanciation
 - classes 6-2
 - classes abstraites 6-16
 - définition 3-15, 6-2
- instructions
 - définition 3-5
- instructions break 4-11
- instructions conditionnelles 4-12
 - if-else 4-12
 - switch 4-13
- instructions continue 4-11
- instructions d'importation 6-25
- instructions de boucle
 - définition 4-3
- instructions de contrôle 4-11
- instructions de retour 4-3
- instructions if-else
 - utilisation 4-12
- instructions machine natives 9-7
- instructions switch 4-13
- int, mot clé 11-3
- interface native Java 10-1
 - Voir aussi* JNI
- interface, mot clé 6-18, 11-3
- interfaces
 - définition 6-18
 - interface native Java 10-1, 10-2
 - remplacement de l'héritage multiple 6-18

J

- J2EE 5-2
 - Voir aussi* Java 2 Enterprise Edition
- J2ME 5-3
 - Voir aussi* Java 2 Micro Edition
- J2SE 5-2, 11-2
 - Voir aussi* Java 2 Standard Edition
- Java
 - définition 9-2
 - langage orienté objet 6-1
- Java 2 Enterprise Edition 5-2
- Java 2 Micro Edition 5-3
- Java 2 Standard Edition 5-2
- Java 2 Standard Edition, présentation 11-2
- Java Runtime Environment 9-2
 - Voir aussi* JRE
- Java, bytecodes 9-1
- Java, chargeur de classe 9-6
- Java, machine virtuelle 9-1
 - Voir aussi* JVM
- Java, paquet de sécurité 9-4
- java.applet, paquet
 - présentation 5-7
- java.awt, paquet
 - présentation 5-6

- java.beans, paquet
 - présentation 5-7
- java.io, paquet
 - présentation 5-5
- java.lang, paquet
 - présentation 5-4
- java.lang.reflect, paquet
 - présentation 5-8
- java.math, paquet
 - présentation 5-5
- java.net, paquet
 - présentation 5-9
- java.rmi, paquet
 - présentation 5-9
- java.security, paquet
 - présentation 5-9
- java.sql, paquet
 - présentation 5-8
- java.swing, paquet
 - présentation 5-6
- java.text, paquet
 - présentation 5-5
- java.util, paquet
 - présentation 5-4
- javah 10-3
 - options 10-4
- jetons 5-27
- JIT, compilateurs just-in-time 9-7
- JNI (Java Native Interface) 10-1, 10-2
- JRE (Java Runtime Environment)
 - relation avec JVM 9-2
- just-in-time (JIT), compilateurs 9-7
- JVM (Java Virtual Machine)
 - avantages 9-2
 - chargeur de classe 9-6
 - définition 9-1
 - et JNI 10-2
 - gestion de la mémoire 9-2
 - instructions 9-1
 - introduction 9-1
 - portabilité 9-2
 - relation avec JRE 9-2
 - rôles principaux 9-2
 - sécurité 9-2
 - spécification et. implémentation 9-2
 - vérificateur 9-3

L

- lecture des flux d'objets 8-8
- lecture, types de données 8-8
- libération des ressources de flux 8-6
- littéraux 4-4
 - Voir aussi* séquences d'échappement
- définition 2-4
- long, mot clé 11-3

M

- machine virtuelle Java 9-1
 - Voir aussi* JVM
- Macintosh
 - support dans JBuilder 1-5
- Math, classe
 - présentation 5-12
- membres, accès 3-16
- méthodes 2-6
 - accès 10-2
 - déclaration 6-3
 - définition 6-3
 - implémentation 6-3
 - main 4-7
 - redéfinition 6-17
 - static 4-7
 - surcharge 6-12
 - utilisation 3-13
- méthodes d'accès 6-13
- méthodes, appels 6-3, 10-2
- Micro Edition 5-3
- modificateurs d'accès 4-6, 6-12
 - dans un paquet 6-12
 - extérieur d'un paquet 6-13
 - non spécifié 6-12
 - par défaut 6-12
 - tableau 11-4
- mots clés
 - boucles 11-4
 - définition 3-1
 - gestion des exceptions 11-5
 - modificateurs d'accès 4-6, 11-4
 - paquets, classes, membres et interfaces 11-3
 - réservés 11-5
 - tableaux 11-3
 - types de données et de retour 11-3

N

- interface de code natif
 - Voir aussi* JNI
- native code interface 10-1, 10-2
- native, mot clé 10-2, 11-3
- new, mot clé 11-3
- new, opérateur 6-2
- NotSerializableException, exception 8-4

O

- Object, classe
 - présentation 5-10
- ObjectInputStream, classe 8-2, 8-6
 - méthodes 8-8
- ObjectOutputStream, classe 8-2, 8-4
 - méthodes 8-6

objets

- allocation de mémoire pour 6-2
- classes et 6-2
- définition 6-2
- désérialisation 8-1
- libération de mémoire pour 6-2
- référencement 8-8
- sérialisation 8-1
- objets persistants 8-1
- objets transitoires 8-1
- objets, référencement 6-2
- opérateur ternaire 3-12, 11-22
 - définition 3-3
- opérateurs
 - accès 3-16
 - affectation, tableau 3-10, 11-20
 - arithmétiques 3-7
 - arithmétiques, tableau 11-19
 - au niveau bits, tableau 11-21
 - comparaison, tableau 3-10, 11-21
 - de base 11-18
 - définition 3-2
 - logique ou booléen 3-9
 - logiques, tableau 11-20
 - niveau bits 3-11
 - tableaux 11-18
 - ternaire 3-12, 11-22
 - utilisation 3-7
- opérateurs arithmétiques
 - définition 3-2
 - tableau 3-8, 11-19
 - utilisation 3-7
- opérateurs au niveau bits 3-11
 - définition 3-3
 - tableau 3-12, 11-21
- opérateurs booléens
 - définition 3-2
 - tableau 3-9
- opérateurs d'affectation
 - définition 3-2
 - tableau 3-10, 11-20
- opérateurs de base
 - tableau 11-18
- opérateurs de comparaison
 - définition 3-3
 - tableau 3-10, 11-21
- opérateurs logiques
 - définition 3-2
 - tableau 3-9, 11-20
- opérateurs mathématiques
 - tableau 3-8
 - utilisation 3-7
- OutputStream, classe
 - présentation 5-22

P

- package, instructions 6-25
- package, mot clé 11-3, 11-4
- paquet applet
 - présentation 5-7
- paquet AWT
 - présentation 5-6
- paquet de beans
 - présentation 5-7
- paquet de sécurité 9-4
 - présentation 5-9
- paquet de texte
 - présentation 5-5
- paquet des entrées/sorties
 - présentation 5-5
- paquet des réflexions
 - présentation 5-8
- paquet des utilitaires
 - Enumeration, interface, présentation 5-16
 - présentation 5-4
 - Vector, classe, présentation 5-17
- paquet du langage
 - classes d'enveloppe de type 5-11
 - Math, classe, présentation 5-12
 - Object, classe, présentation 5-10
 - présentation 5-4
 - String, classe, présentation 5-12
 - StringBuffer, classe, présentation 5-14
 - System, classe 5-15
- paquet mathématique
 - présentation 5-5
- paquet réseau
 - présentation 5-9
- paquet RMI
 - présentation 5-9
- paquet SQL
 - présentation 5-8
- paquet Swing
 - présentation 5-6
- paquets
 - accès aux membres de classes 6-12
 - accès aux membres depuis l'extérieur 6-13
 - déclaration 6-25
 - définition 6-25
 - importation 6-25
 - Java, tableau 5-3
- Paquets de Java 2 Standard Edition 5-3
- paquets externes
 - importation 6-25
- paquets javax
 - présentation 5-6
- pointeurs 10-2
- politique de sécurité 9-4

- polymorphisme 6-17
 - exemple 6-19
- portabilité
 - de Java 9-2
- portée
 - définition 3-5
- primitifs, types de données
 - conversion en chaînes 11-7
 - conversion en d'autres types primitifs 11-6
 - conversion en référence 11-8
 - définition 5-11
- PrintStream, classe
 - présentation 5-23
- private, mot clé 6-12, 11-4
- programmation orientée objet 6-1
 - exemple 6-4
- protected, mot clé 6-12, 11-4
- prototypes 10-4
- public, mot clé 4-6, 6-12, 11-4

R

- raccourcissement, conversions de types 4-8
- ramasse-miettes 6-2, 6-4
 - rôle de JVM 9-2
- RandomAccessFile, classe
 - présentation 5-26
- readObject() 8-6, 8-8
- redéfinition des méthodes 6-17
- référencement des objets 8-8
- références d'objets 6-2
- réservés, mots clés
 - tableau 11-5
- ressources
 - libération de flux 8-6
- ressources de flux
 - libération 8-6
- ressources en ligne 1-6
- restauration des objets 8-1
- return, mot clé 11-3
- run() 7-2
- Runnable, interface
 - implémentation 7-3

S

- sécurité
 - applet et. application 9-6
 - chargeur de classe 9-6
 - dans la JVM 9-2
 - sérialisation et 8-8
- SecurityManager, classe 9-4
- séquences d'échappement 4-4
 - tableau 11-18
- sérialisation
 - définition 8-1

- raisons pour 8-1
- sécurité et 8-8
- sérialisation des objets 8-1
- Serializable, interface 8-2, 8-3
- setSecurityManager() 9-5
- setter, méthodes 6-13
- short, mot clé 11-3
- sous-routines 2-6
 - Voir aussi* méthodes
- Standard Edition 11-2
- static, blocs de code 10-3
- static, mot clé 4-6, 11-3
- stockage des objets sur disque 8-1
- StreamTokenizer, classe
 - présentation 5-27
- strictfp, mot clé 11-3
- String, classe
 - présentation 5-12
- StringBuffer, classe
 - présentation 5-14
- super, mot clé 6-12, 11-3
- superclasses 6-11
- support aux développeurs 1-6
- support technique 1-6
- surcharge des méthodes 6-12
- switch, mot clé 11-4
- synchronisation des threads 7-8
- synchronized, mot clé 11-3
- System, classe 5-15

T

- tableaux
 - accès 3-16
 - définition 2-4
 - indexation 3-16
 - représentation des chaînes 5-17
 - utilisation 3-14
- tableaux de caractères 5-17
- temps partagé 7-7
- this, mot clé 11-3
- Thread, classe
 - sous-classement 7-2
- ThreadGroup, classe 7-8
- threads 7-1
 - arrêt 7-7
 - création 7-5
 - cycle de vie 7-2
 - démarrage 7-5
 - groupes 7-8
 - implémentation de l'interface Runnable 7-3
 - non exécutables 7-6
 - personnalisation de la méthode run() 7-2
 - priorité 7-7
 - synchronisation 7-8
 - temps partagé 7-7

- threads démon 7-1
 - threads multiples 7-1
- threads démon 7-1
- threads multiples 7-1
- throw, mot clé 11-5
- throws, mot clé 11-5
- transient, mot clé 11-3
- transtypage 4-2
 - Voir aussi* conversions de types
- transtypage implicite
 - définition 4-8
- try, mot clé 11-5
- type de données chaîne
 - conversion en primitif 11-10
 - définition 2-4
- types
 - écriture dans des flux 8-6
 - lecture 8-8
- types de données
 - chaînes 2-4
 - composite 2-3
 - conversion et transtypage 4-2
 - définition 2-2
 - écriture dans des flux 8-6
 - lecture 8-8
 - numérique, tableau 2-3
 - primitifs 2-2, 5-11
 - tableaux 2-4
- types de données composites 2-3
 - chaînes 2-4
 - tableaux 2-4
- types de données de base 2-2
- types de données de référence
 - conversion en autres types de référence 11-14
 - conversion en primitif 11-12

- types de données et de retour
 - tableau 11-3
- types de données numériques, tableau 2-3
- types de données primitifs 2-2
- types de retour 4-3

U

- UnsatisfiedLineError, exceptions 10-3
- Usenet, groupes de discussion 1-7

V

- valeurs
 - comparaison 3-10
- variables
 - définition 2-4
 - instance 6-2
 - membre 6-3
 - objets en tant que 6-2
- variables d'instances 6-2
- variables membre 6-3
- variables, déclarations 2-5
- Vector, classe
 - présentation 5-17
- vérificateur Java 9-3
- vérification
 - des bytecodes Java 9-3
- void, mot clé 4-6, 11-3
- void, type de retour
 - définition 4-3

W

- while, mot clé 11-4
- writeObject() 8-4, 8-8

