Un livre de Wikilivres.

Programmation Java

Une version à jour et éditable de ce livre est disponible sur Wikilivres,
une bibliothèque de livres pédagogiques, à l'URL :
http://fr.wikibooks.org/wiki/Programmation_Java

Vous avez la permission de copier, distribuer et/ou modifier ce document selon les termes de la Licence de documentation libre GNU, version 1.2 ou plus récente publiée par la Free Software Foundation ; sans sections inaltérables, sans texte de première page de couverture et sans Texte de dernière page de couverture. Une copie de cette licence est inclue dans l'annexe nommée « Licence de documentation libre GNU ».

Introduction

Introduction au langage Java

Le 23 mai 1995, Sun Microsystems présentait une nouvelle plateforme, composée d'un langage de programmation et d'une machine virtuelle. Java™ était né.

L'histoire de Java commence en fait en 1991, lorsque Sun décide de lancer un projet destiné à anticiper l'évolution de l'informatique, confié à James Gosling, Patrick Naughton et Mike Sheridan. Ce projet, appelé « Green Project » (en anglais « projet vert »), eu comme résultat une plate-forme baptisée « Oak » (en anglais « chêne »), indépendante du système, orientée objet et légère. Oak était initialement destinée à la télévision interactive. Il a été renommé en Java pour de simples raisons de droit d'auteur.

Lorsqu'il est révélé en 1995, Java profite de l'essor d'Internet en permettant l'un des premiers mécanismes d'interactivité au niveau du poste client : l'appliquette *(applet)* Java.

Langage orienté objet d'usage généraliste, Java est enrichi par des bibliothèques, des outils et des environnements très diversifiés, standardisés par le Java Community Process (JCP), consortium chargé de l'évolution de Java. Ce consortium regroupe des entreprises, comme Sun, IBM, Oracle, Borland, BEA, des organismes de normalisation, comme le NIST, des organismes du monde Open Source, comme la Fondation Apache et le JBoss Group, et des particuliers.

Il est possible d'utiliser Java pour créer des logiciels dans des environnements très diversifiés :

- applications sur client lourd (JFC) ;
- applications Web, côté serveur (servlets, JSP, Struts, JSF) ;
- applications réparties (EJB) ;
- applications embarquées (J2ME) ;
- applications sur carte à puce (JavaCard).

Ces applications peuvent être enrichies par de nombreuses fonctionnalités :

- accès à des bases de données (JDBC et JDO) ;
- accès à des annuaires (JNDI) ;
- traitements XML (JAXP) ;
- connexion à des ERP (JCA) ;
- accès à des traitements en d'autres langages (JNI) ;
- services web (JAX-RPC, JAXM, JAXR) ;
- multimédia (Java Media) ;
- téléphonie (JTAPI) ;
- télévision interactive (Java TV).

Ceci n'est bien sûr qu'un petit échantillon. Il existe bien d'autres bibliothèques.

Historique de Java

- 1991 : Début du projet Oak, qui donnera naissance à Java.
- Eté 1992 : première présentation interne des possibilités de Oak. Un appareil appelé « Star Seven » permet de visualiser une animation montrant Duke, l'actuelle mascotte de Java.
- 1994 : Développement de HotJava, un navigateur internet entièrement écrit en Java.
- 23 mai 1995 : Lancement officiel de Java 1.0
- 23 janvier 1996 : Lancement du JDK 1.0.
- 29 mai 1996 : Première conférence JavaOne. Java Media et les servlets y sont notamment annoncés.
- 16 août 1996 : Premières éditions des livres *Java Tutorial* et *Java Language Specification* par Sun et Addison-Wesley.
- Septembre 1996 : Lancement du site Web *Java Developer Connection Web*.

- 16 octobre 1996 : Spécification des JavaBeans.
- 11 janvier 1997 : Lancement des JavaBeans.
- 18 février 1997 : Lancement du JDK 1.1.
- 28 février 1997 : Netscape annonce que Communicator supporte désormais Java.
- 4 mars 1997 : Lancement des *servlets*.
- 10 mars 1997 : Lancement de JNDI.
- 5 juin 1997 : Lancement de Java Web Server 1.0.
- 5 août 1997 : Lancement de Java Media et de Java Communication API.
- Mars 1998 : Lancement des JFC et notamment de Swing.
- 8 décembre 1998 : Lancement de Java 2 et du JDK 1.2.
- 4 mars 1999 : Support XML.
- 27 mars 1999 : Lancement de la machine virtuelle HotSpot.
- 2 juin 1999 : Lancement des JSP.
- 15 juin 1999 : Formalisation des environnements J2ME, de J2SE et J2EE ; Lancement de Java TV.
- Août 1999 : Lancement de Java Phone.
- 8 mai 2000 : Lancement de J2SE 1.3.
- 9 mai 2002 : Lancement de J2SE 1.4.
- 24 novembre 2003 : Lancement de J2EE 1.4.
- 30 septembre 2004 : Lancement de J2SE 1.5, nommé également « J2SE 5.0 » ou « Tiger ».
- 11 décembre 2006 : Lancement de JavaSE 6, nommé également « Mustang ».
- 13 novembre 2006 : Passage de Java, c'est-à-dire le JDK (JRE et outils de développement) et les environnements Java EE (déjà sous licence CDDL) et Java ME sous licence GPL. Java devient donc un logiciel libre.
- 27 janvier 2010 : Sun Microsystem est racheté par Oracle. Désormais, Java est maintenu par la société Oracle.
- 28 Juillet 2011 : Lancement de JavaSE 7, nommé également « Dolphin ».
- 18 mars 2014 : Lancement de JavaSE 8, nommé également « Kenaï ».

Présentation du langage

Java est un langage typé et orienté objet. Il est compilé et basé sur une architecture logicielle très particulière nécessitant une machine virtuelle Java. Il utilise les notions usuelles de la programmation orientée objet : la notion de classe, d'encapsulation, d'héritage, d'interface, de virtualité, de généricité, … Il est accompagné d'un ensemble énorme de bibliothèques standard couvrant de très nombreux domaines, notamment des bibliothèques graphiques. C'est un langage qui présente d'excellentes propriétés de portabilité du code. Son gros point faible est une relative lenteur, surtout si on le compare à des langages comme le C++. Cependant, ce défaut a été résolu en grande partie par l'introduction de la technologie JIT (compilateur *Just-In-Time*, en anglais « juste à temps »), qui compile le code à la première exécution, permettant une exécution quasiment aussi rapide qu'en C/C++.

Machine virtuelle

Introduction à la machine virtuelle Java

Le langage Java utilise une notion extrêmement importante en informatique : la notion de machine virtuelle. Cette machine virtuelle est composée de trois parties, et est souvent fournie avec l'OS, ou facilement installable comme un logiciel normal.

Sur les ordinateurs, la même machine virtuelle est capable de lancer :

- des applications indépendantes (*standalone* en anglais), lancées et fonctionnant comme toute application installée sur la machine,
- des applets, à partir d'une page HTML (internet, réseau local ou en local sur la machine). Pour cela, il faut que le navigateur possède une extension permettant d'utiliser une machine virtuelle Java pour l'exécution de ces applets.

Environnement d'exécution

Le langage Java est un langage orienté objet qui doit être compilé. Cependant, le compilateur Java ne produit pas directement un fichier exécutable, mais du **code intermédiaire** sous la forme d'un ou plusieurs fichiers dont l'extension est `.class` ; ce code intermédiaire est appelé ***bytecode***. Pour exécuter le programme, il faut utiliser la machine virtuelle Java qui va **interpréter** le code intermédiaire en vue de l'exécution du programme.

Il ne s'agit pas d'une compilation normale, car le compilateur ne produit pas du code compréhensible directement par le microprocesseur, ni d'un langage interprété, car il y a tout de même la notion de compilation, mais une situation intermédiaire entre un langage interprété et un langage complètement compilé.

En d'autres termes, un programme Java, une fois compilé en code intermédiaire, n'est compréhensible que par la machine virtuelle, qui va traduire à la volée (interprétation) les instructions exécutées en code compréhensible par la machine physique.

Depuis Java SE 1.3, les machines virtuelles d'Oracle contiennent un interpréteur capable d'optimiser le code appelé HotSpot.

L'interpréteur effectue plusieurs tâches :

- vérification du code intermédiaire ;
- traduction en code natif (spécifique à la plateforme, au système d'exploitation) ;
- optimisation.

Outre un interpréteur, l'environnement d'exécution fournit également :

- un noyau multitâches pour les machines virtuelles fonctionnant sur des systèmes monotâches *(green virtual machines)* ;
- un « bac à sable » *(sandbox)* pour l'exécution de processus distants.

Chargeur de classe

Le chargeur de classe charge les classes nécessaires à l'exécution, alloue l'espace mémoire nécessaire et établit les liens entre elles *(linkage)*. Le chargeur de classe connait la structure d'un fichier `.class`.

Gestionnaire de mémoire

Le gestionnaire mémoire assure les services liés à la mémoire, en particulier :

- un ramasse-miette *(garbage collector)* ;
- une protection mémoire même sur les machines dépourvues d'unité de gestion mémoire (MMU).

Avantage de l'utilisation de la machine virtuelle

L'utilisation d'une machine virtuelle a l'énorme avantage de garantir une vraie **portabilité**. Il existe des machines virtuelles Java pour de très nombreux environnements : Windows, MacOS, Linux et autres.

Ces machines virtuelles sont capables d'exécuter exactement le même code intermédiaire (les mêmes fichiers Java en *bytecode*) avec une totale compatibilité. C'est là une situation unique et assez remarquable qui a fait le succès de ce langage.

La machine virtuelle Java n'est pas uniquement développée sur des ordinateurs classiques, une multitude d'appareils disposent d'une machine virtuelle Java : téléphones portables, assistants personnels (PDA)…

Les interfaces de programmation (API) Java : des fonctionnalités énormes

La machine virtuelle Java possède un ensemble de bibliothèques extrêmement complètes : des bibliothèques graphiques, des bibliothèques systèmes, etc...

Toutes ces bibliothèques sont totalement portables d'un environnement à un autre.

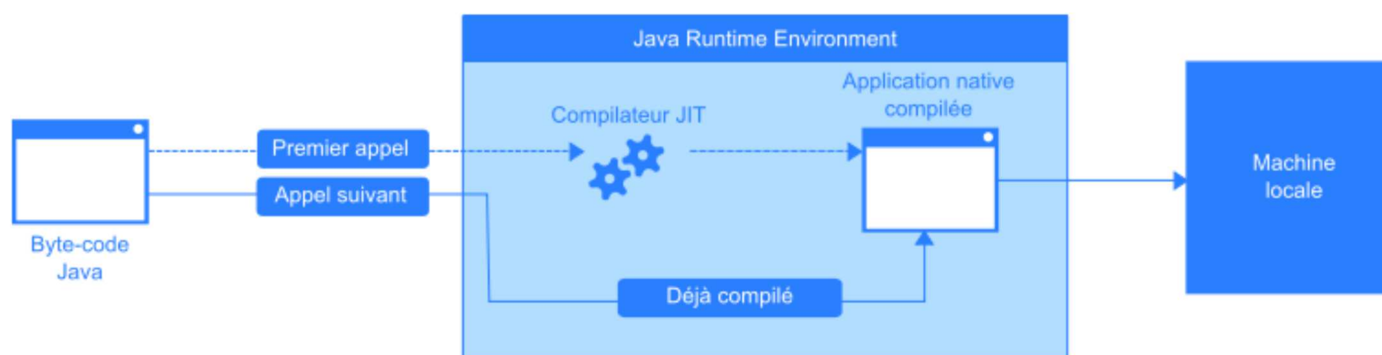
La machine virtuelle Java offre donc un véritable système d'exploitation virtuel, qui fonctionne au dessus du système d'exploitation, de la machine cible et le masque totalement aux applications.

La lenteur de Java

Le gros point faible du concept de machine virtuelle est que le code intermédiaire *(bytecode)* est interprété par la machine virtuelle. Ceci entraîne une baisse importante des performances des programmes.

Toutefois avec les machines virtuelles actuelles, cet argument n'a plus autant de poids. La technique appelée « compilation juste à temps » (JIT : *Just-In-Time*) est employée par la machine virtuelle quand une méthode est appelée. Cette technique consiste à compiler à la volée la méthode appelée (la première fois) en code natif directement exécutable par le processeur.

Toute méthode s'exécute ainsi aussi rapidement que du code natif.



Processeur Java

Un processeur Java est l'implémentation matérielle d'une machine virtuelle Java. C'est à dire que les *bytecodes* constituent son jeu d'instruction.

Actuellement, il y a quelques processeurs Java disponibles :

- en:picoJava, première tentative de construction d'un processeur par Sun Microsystems,
- aJ100 (http://www.ajile.com/) [archive] de aJile. Disponibles sur cartes Systronix (http://jstamp.systronix.com/) [archive],
- Cjip (http://www.imsystech.com/) [archive] (Imsys Technologies),
- Komodo (http://ipr.ira.uka.de/komodo/komodoEng.html) [archive] : micro-controlleur Java multi-thread pour la

recherche sur la planification temps-réel,

- FemtoJava (http://www.inf.ufrgs.br/~lse/) [archive] : projet de recherche,
- ARM926EJ-S : processeur ARM pouvant exécuter du bytecode java,

Bases du langage

Un programme Java est une suite d'instructions exécutées de manière séquentielle. D'une manière générale, les instructions sont séparées par des points-virgules « `;` ».

Ces instructions s'exécutent de gauche à droite et de haut en bas. Elles peuvent être organisées dans des blocs.

Point d'entrée du programme

Tout programme Java contient au moins une classe[1], nommée par convention avec une majuscule contrairement aux méthodes. De plus, il nécessite un point d'entrée. Pour un programme simple, le point d'entrée est la méthode "*main*" qui doit être publique, statique et située dans une classe qui elle même doit être publique (d'où les mots-clés **public** et **static**) :

```
public class ClassTest {
    public static void main(String args[]) {
        // Instructions du programme
    }
}
```

L'argument *args* de cette méthode contient les paramètres passés au programme par la ligne de commande.

D'autres types de programmes, comme les applets, les servlets ou encore les applications Android, utilisent d'autres méthodes comme point d'entrée.

Blocs d'instructions

Un bloc d'instructions est une suite d'instructions commençant par une accolade ouvrante ({) et se terminant par une accolade fermante (}).

Un bloc d'instructions est considéré comme une seule instruction par les instructions *for*, *while*, *if*, *else* et *case*. Nous détaillerons cela plus loin.

Ce bloc délimite également la portée des variables. Toute variable déclarée dans ce bloc n'est accessible qu'à partir de ce bloc, et n'existe que durant l'exécution du bloc.

Commentaires

*Pour plus de détails voir : **Programmation Java/Commentaires**.*

Les commentaires sont, en programmation informatique, des portions du code source ignorées par le compilateur. Ils sont très pratiques pour préciser quelque chose ou pour "mettre de coté" du code sans le supprimer. Java permet d'insérer des commentaires en utilisant deux syntaxes différentes :

- La séquence `//` permet d'insérer un commentaire sur une seule ligne, qui se termine donc à la fin de la ligne.
- La séquence `/*` permet d'insérer un commentaire sur plusieurs lignes. La séquence `*/` marque la fin du commentaire. En d'autres termes, tout ce qui est situé entre `/*` et `*/` est considéré comme faisant partie du commentaire.

Exemple :

```
// Commentaire sur une ligne
public class /* un commentaire au milieu de la ligne */ Exemple {
    /*
        Commentaire sur
        plusieurs lignes
        ...
```

```
    */
    public static void main(String[] args) {
    }
}
```

Commentaire Javadoc

Les commentaires normaux sont totalement ignorés par le compilateur Java.

En revanche, certains commentaires sont interprétés par le générateur automatique de documentation Javadoc.

Ces commentaires commencent par la séquence `/**` et se terminent par `*/`. Le contenu décrit l'entité qui suit (classe, interface, méthode ou attribut), suivi d'une série d'attributs dont le nom commence par un arobase `@`.

La documentation générée étant au format HTML, il est possible d'insérer des balises dans la description.

Exemple :

```
/**
    Une classe pour illustrer les commentaires Javadoc.
    @author Moi :-)
*/
public class Exemple {
    /**
        Une méthode <b>main</b> qui ne fait rien.
        @param args Les arguments de la ligne de commande.
    */
    public static void main(String[] args) {
    }
}
```

En fait, il existe un attribut Javadoc qui est pris en compte par le compilateur : l'attribut `@deprecated`. Cet attribut marque une classe, une méthode ou une variable comme obsolète. Ce qui signifie que si une autre classe l'utilise un avertissement est affiché à la compilation de cette classe.

Exemple :

```
/**
    Une méthode obsolète. Il faut utiliser get() à la place.
    @deprecated
*/
public Object getElement(int index)
{ ... }

/**
    Une nouvelle méthode.
*/
public Object get(int index)
{ ... }
```

Cet attribut permet de modifier une bibliothèque de classe utilisée par plusieurs applications, en la laissant compatible avec les applications utilisant l'ancienne version, mais en indiquant que les anciennes classes / méthodes / variables ne doivent plus être utilisées et pourraient ne plus apparaître dans une version ultérieure.

Références

1. Kathy Sierra, Bert Bates, *Java : tête la première* (https://books.google.fr/books?id=n79bw0RFNUIC&pg=PA39&dq=%22Un+programme+Java+est+un+ensemble+de+classes+%28ou+au+moins+une+classe%29.%22&hl=fr&sa=X&ved=0ahUKEwimiam8lcHKAhWE0xQKHVfRB_sQ6AEIKDAA#v=onepage&q=%22Un%20programme%20Java%20est%20un%20ensemble%20de%20classes

%20%28ou%20au%20moins%20une%20classe%29.%22&f=false) [archive], O'Reilly Media, Inc., 2004

Premier programme

Premier programme

Le fichier source

Ce programme doit être écrit dans le fichier **Exemple.java**.

```
public class Exemple {
	public static void main(String[] args) {
		System.out.println("Hello world!");
	}
}
```

Explications sur le langage

Ce programme est le classique Hello world. Comme son nom l'indique, ce programme va afficher la phrase "Hello world" à l'écran. Analysons-le ligne par ligne :

```
public class Exemple {
```

Cette ligne déclare une classe publique que nous appelons *Exemple*.

Un fichier `.java` ne peut contenir qu'une seule classe publique et le fichier doit porter le nom de cette classe. Ainsi, le fichier de ce premier exemple s'appellera obligatoirement `Exemple.java` (en respectant la classe).

Ce système de nommage permet au compilateur et à l'interpréteur de trouver les fichiers correspondant à une classe.

```
public static void main(String[] args) {
```

Cette ligne déclare une méthode appelée *`main`*. Cette méthode est le point d'entrée du programme (la méthode appelée lorsqu'il sera exécuté).

Elle prend en argument un tableau de chaînes de caractères (`String[] args`) et ne retourne rien (`void`).

Cette méthode est publique et statique, ce qui sera expliqué plus loin.

```
System.out.println("Hello world!");
```

Cette dernière instruction invoque la méthode `println` de l'objet `out` se trouvant dans la classe `System` en lui passant en argument la chaîne `Hello world!`. L'exécution de cette méthode aura pour résultat l'affichage de `Hello world!`.

Cette ligne peut sembler obscure pour l'instant. Les détails seront abordés par la suite.

Compilation du fichier source

Nous allons présenter la compilation de notre programme en utilisant le compilateur gratuit Java très répandu nommé **Javac**, disponible gratuitement auprès d'Oracle. Cependant, il faut savoir qu'il existe aussi de nombreux environnements de développement **Java** permettant de taper, compiler, exécuter ou débuguer des programmes dans ce langage.

Tapez le programme précédent et sauvegardez le dans un fichier *Exemple.java* (pour la raison expliquée précédemment) et tapez dans une fenêtre :

Invite de commande DOS	Terminal Unix

<pre>> javac Exemple.java
> dir
Exemple.class
Exemple.java
></pre>
	<pre>$ javac Exemple.java
	$ ls
	Exemple.class
	Exemple.java
	$</pre>

Le compilateur Javac a produit le fichier *Exemple.class*, contenant le code intermédiaire. Ce fichier n'est normalement pas éditable[1], ce qui peut garantir le copyright.

En cas de problèmes...

Voici les points à vérifier selon le type de problème :

Le système indique que le compilateur est introuvable

1. Vérifiez que vous avez installé un kit de développement Java (le JDK) et pas simplement un environnement d'exécution (JRE) qui ne comporte pas les outils de compilation.
2. Vérifiez que le chemin du répertoire *bin* contenant le compilateur *javac* (*javac.exe* sous Windows) est dans la liste de la variable d'environnement `PATH`.

Le compilateur se lance mais affiche une erreur de classe non trouvée

1. Si la classe ne déclare aucun paquetage (*package*), vérifiez que vous lancez la commande depuis le répertoire où se trouve le fichier source (**.java*). Changez de répertoire si nécessaire avant de recommencer.
2. Sinon, vous devez lancer la commande depuis le répertoire parent du paquetage racine, en donnant le chemin relatif du fichier source depuis ce répertoire parent.
3. Dans les deux cas ci-dessus, en plus de changer de répertoire courant, il peut être nécessaire d'ajouter le chemin de ce répertoire dans le *classpath*. Cela peut être fait soit dans la ligne de commande avec l'option `-classpath` (ou `-cp`), soit dans la variable d'environnement `CLASS_PATH`.

Le compilateur se lance mais affiche une erreur de syntaxe

1. Vérifiez le contenu du fichier source. Pour compiler les exemples de ce livre, le mieux est de faire un copier-coller **complet** du code.
2. Assurez-vous de compiler le bon fichier source et pas un autre.

Exécution du programme

Java est une machine virtuelle java fournie par Oracle et disponible pour de nombreux environnements.

Pour exécuter notre code intermédiaire, il faut taper :

```
java Exemple
```

L'exécution du programme affiche dans une fenêtre console la fameuse phrase *Hello world!*.

En cas de problèmes...

Voici les points à vérifier selon le type de problème :

Le système indique que la commande java est introuvable

1. Vérifiez que vous avez installé un kit de développement Java (le JDK).
2. Vérifiez que le chemin du répertoire *bin* contenant l'interpréteur *java* (*java.exe* sous Windows) est dans la liste de la variable d'environnement `PATH`.

L'interpréteur se lance mais affiche une erreur de classe non trouvée

1. Vérifiez que vous avez compilé le fichier source **.java* (voir section précédente) sans erreur, c'est-à-dire que vous avez obtenu un fichier compilé **.class*.
2. Si la classe ne déclare aucun paquetage (*package*), vérifiez que vous lancez la commande depuis le répertoire où se trouve le fichier compilé (**.class*). Changez de répertoire si nécessaire avant de recommencer.

3. Sinon, vous devez lancer la commande depuis le répertoire parent du paquetage racine, en donnant le nom complet de la classe (incluant le nom du paquetage).
4. Dans les deux cas ci-dessus, en plus de changer de répertoire courant, il peut être nécessaire d'ajouter le chemin de ce répertoire dans le *classpath*. Cela peut être fait soit dans la ligne de commande avec l'option `-classpath` (ou `-cp`), soit dans la variable d'environnement `CLASS_PATH`.

L'interpréteur se lance mais rien ne s'affiche, ou le comportement n'est pas le même

1. Vérifiez le contenu du fichier source. Pour compiler les exemples de ce livre, le mieux est de faire un copier-coller **complet** du code.
2. Assurez-vous de lancer la bonne classe principale et pas une autre.

Voir aussi

- (anglais) http://docs.oracle.com/javase/tutorial/getStarted/index.html

Références

1. http://www.jmdoudoux.fr/java/dej/chap-decompil.htm

Cette page fait partie du livre Programmation

Types de base

Les types de base ou types primitifs de Java sont listés dans le tableau suivant :

Types primitifs de Java

Type	Taille	Syntaxe	Description	Intervalle
`char`	2 octets 16 bits	`'caractère'`	Une unité de code, suffisant à représenter un grand nombre de point de code, et même un caractère Unicode (UTF-16) `'b'` `'\u250C'`	`'\u0000'` à `'\uFFFF'`
`byte`	1 octet 8 bits		Un nombre entier de 8 bits (soit un octet) signé	`-128` à `127`
`short`	2 octets 16 bits		Un nombre entier de 16 bits signé entre -32 768 et +32 767	`-32768` à `32767`
`int`	4 octets 32 bits	`[+|-]chiffres...`	Un nombre entier de 32 bits signé entre -2 147 483 648 et +2 147 483 647	`-2147483648` à `2147483647`
`long`	8 octets 64 bits	`[+|-]chiffres...L`	Un nombre entier de 64 bits signé entre -9 223 372 036 854 775 808 et +9 223 372 036 854 775 807	`-9223372036854775808L` à `9223372036854775807L`
`float`	4 octets 32 bits	`[+|-][chiffres].[chiffres][E[+|-]chiffres]F`	Un nombre à virgule flottante de 32 bits signé (simple précision)	▪ de 2^{-149} (`Float.MIN_VALUE`) à $2^{128} - 2^{104}$ (`Float.MAX_VALUE`), ▪ `0.0F`, ▪ $-\infty$ (`Float.NEGATIVE_INFINITY`), ▪ $+\infty$ (`Float.POSITIVE_INFINITY`), ▪ pas un nombre (`Float.NaN`).
`double`	8 octets 64 bits	`[+|-][chiffres].[chiffres][E[+|-]chiffres][D]`	Un nombre à virgule flottante de 64 bits signé (double précision)	▪ de 2^{-1074} (`Double.MIN_VALUE`) à $2^{1024} - 2^{971}$ (`Double.MAX_VALUE`), ▪ `0.0D`, ▪ $-\infty$ (`Double.NEGATIVE_INFINITY`), ▪ $+\infty$ (`Double.POSITIVE_INFINITY`), ▪ pas un nombre (`Double.NaN`).
`boolean`	1 octet	`false|true`	Une valeur logique	`false` (faux) ou `true` (vrai)

Remarques :

- Une constante numérique de type entier est considérée comme étant de type `int`. Il faut ajouter le suffixe L pour avoir une constante de type `long`.
- Une constante numérique de type décimal est considérée comme étant de type `double`. Il faut ajouter le suffixe F pour avoir une constante de type `float`. Il existe également un suffixe D pour `double` mais son utilisation est optionnelle.
- `String` est le seul type non-primitif (donc absent du tableau) dont les instances possèdent une syntaxe littérale :
  `"caractères..."`

Caractères pour `char` et `String`

Les caractères spéciaux suivants peuvent être utilisés pour les valeurs littérales des types `char` et `String` :

`\r` (ou `\u000D`)
: Retour chariot (*carriage **R**eturn*),

`\n` (ou `\u000A`)
: Nouvelle ligne (***N**ew line*),

`\f` (ou `\u000C`)
: Nouvelle page (***F**eed page*),

`\b` (ou `\u000C`)
: Retour arrière (***B**ackspace*),

`\t` (ou `\u0009`)
: Tabulation (***T**ab*),

`\"`
: Guillemet (pour qu'il soit inclus au lieu de marquer la fin de la chaîne de caractères),

`\'`
: Apostrophe (pour qu'il soit inclus au lieu de marquer la fin du caractère),

`\OOO`
: Caractère 8 bits dont le code OOO est spécifié en octal (jusqu'à 3 chiffres),

`\xXX`
: Caractère 8 bits dont le code XX est spécifié en hexadécimal (jusqu'à 2 chiffres),

`\uXXXX`
: Caractère 16 bits (UTF-16) dont le code XXXX est spécifié en hexadécimal (jusqu'à 4 chiffres).
: Cette séquence d'échappement peut également être utilisée en dehors des chaînes et caractères dans le code source :

```
\u0070ublic \u0063lass UneClasse ... /* <- public class UneClasse ... */
```

: Cette fonctionnalité du langage est cependant à réserver au cas d'utilisation de code Java externe où les identificateurs de classe, attributs ou méthodes utilisent des caractères non supportés (par exemple, en Japonais) par le clavier du développeur.

Variables et classes

Les classes de base comme **String** ne sont pas des variables primitives, il est aisé de les confondre mais les conventions habituelles d'écriture permettent de distinguer les deux types de données. Les types de variables primitives sont toujours écrits en minuscules, par contre les noms des classes ont en principe leur premier caractère en majuscule. Aussi lorsque vous rencontrez un **B**oolean, ce n'est pas un type de base mais bien une classe. En effet les variables primitives peuvent être encapsulées, et Java fournit d'ailleurs pour tous les types primitifs des classes d'encapsulation appelées *wrappers*.

Ceci peut être utile dans certains cas pour bénéficier de certaines caractéristiques de leur classe mère *Object*. Par exemple, la pose d'un verrou de synchronisation (instruction *synchronized*) ne peut se faire que sur un objet.

Les emballages de types primitifs

Une classe d'emballage (*wrapper* en anglais) permet d'englober une valeur d'un type primitif dans un objet. La valeur est stockée sous la forme d'un attribut. Chaque classe permet de manipuler la valeur à travers des méthodes (conversions, ...).

- `Character` pour `char`
- `Byte` pour `byte`
- `Short` pour `short`
- `Long` pour `long`
- `Integer` pour `int`
- `Float` pour `float`
- `Double` pour `double`

- `Boolean` pour `boolean`

Opérateurs

Les opérateurs

Voici une liste des opérateurs utilisables en Java, avec leur signification et leur associativité, dans l'ordre de leur évaluation (du premier au dernier évalué) et le type de données auquel chaque opérateur s'applique :

Opérateur	Description	Type	Associativité
`()`	Appel de méthode	*classes et objets*	de gauche à droite
`[]`	Elément d'un tableau	*tableaux*	
`.`	Membre d'une classe ou d'un objet	*classes et objets*	
`++`	Incrémentation post ou pré-fixée	`byte char short int long float double`	de droite à gauche
`--`	Décrémentation post ou pré-fixée	`byte char short int long float double`	
`+`	Positif	`byte char short int long float double`	
`-`	Négation	`byte short int long float double`	
`!`	Non logique	`boolean`	
`~`	Non binaire	`byte char short int long`	
`(`*type*`)`	Conversion de type	*tous*	
`*`	Multiplication	`byte char short int long float double`	de gauche à droite
`/`	Division	`byte char short int long float double`	
`%`	Modulo (reste de la division entière)	`byte char short int long`	
`+`	Addition	`byte char short int long float double` `String` *(concaténation)*	de gauche à droite
`-`	Soustraction	`byte char short int long float double`	
`<<`	Décalage de bit vers la gauche	`byte char short int long`	de gauche à droite
`>>`	Décalage de bit vers la droite (signe conservé)	`byte char short int long`	
`>>>`	Décalage de bit vers la droite (signe décalé)	`byte char short int long`	
`<`	Inférieur	`byte char short int long float double`	de gauche à droite
`<=`	Inférieur ou égal	`byte char short int long float double`	
`>`	Supérieur	`byte char short int long float double`	
`>=`	Supérieur ou égal	`byte char short int long float double`	
`==`	Egal	`byte char short int long float double` *objet*	de gauche à droite
`!=`	Différent	`byte char short int long float double` *objet*	
`&`	ET binaire	`byte char short int long boolean`	de gauche à droite
`^`	OU exclusif binaire	`byte char short int long boolean`	de gauche à droite

<table>
<tr><td><code>|</code></td><td>OU binaire</td><td><code>byte char short int long boolean</code></td><td>de gauche à droite</td></tr>
<tr><td><code>&&</code></td><td>ET logique</td><td><code>boolean</code></td><td>de gauche à droite</td></tr>
<tr><td><code>||</code></td><td>OU logique</td><td><code>boolean</code></td><td>de gauche à droite</td></tr>
<tr><td><code>?:</code></td><td>Opérateur ternaire de condition</td><td><code>boolean ?</code> tous <code>:</code> tous</td><td>de droite à gauche</td></tr>
<tr><td><code>=</code></td><td>Affectation</td><td>tous</td><td rowspan="13">de droite à gauche</td></tr>
<tr><td><code>+=</code></td><td>Addition et affectation</td><td><code>byte char short int long float double</code>
<code>String</code> (concaténation)</td></tr>
<tr><td><code>-=</code></td><td>Soustraction et affectation</td><td><code>byte char short int long float double</code></td></tr>
<tr><td><code>*=</code></td><td>Multiplication et affectation</td><td><code>byte char short int long float double</code></td></tr>
<tr><td><code>/=</code></td><td>Division et affectation</td><td><code>byte char short int long float double</code></td></tr>
<tr><td><code>%=</code></td><td>Modulo et affectation</td><td><code>byte char short int long float double</code></td></tr>
<tr><td><code><<=</code></td><td>Décaler à gauche et affectation</td><td><code>byte char short int long</code></td></tr>
<tr><td><code>>>=</code></td><td>Décaler à droite (excepté signe) et affectation</td><td><code>byte char short int long</code></td></tr>
<tr><td><code>>>>=</code></td><td>Décaler à droite (signe aussi) et affectation</td><td><code>byte char short int long</code></td></tr>
<tr><td><code>&=</code></td><td>ET binaire et affectation</td><td><code>byte char short int long boolean</code></td></tr>
<tr><td><code>^=</code></td><td>OU exclusif binaire et affectation</td><td><code>byte char short int long boolean</code></td></tr>
<tr><td><code>|=</code></td><td>OU binaire et affectation</td><td><code>byte char short int long boolean</code></td></tr>
<tr><td><code>,</code></td><td>Enchaînement d'expressions</td><td>tous</td><td>de gauche à droite</td></tr>
</table>

Chaque case de la colonne « associativité » regroupe les opérateurs de même priorité.

Expressions

`short` et `byte`

Java effectue une conversion en valeur de type `int` de manière implicite sur les valeurs de type `short` et `byte` dès qu'une opération est effectuée, ce qui peut donner des résultats non conforme à ce que l'on pourrait attendre (détails (http://linformalibre.f2lt.fr/index.php?title=Particularit%C3%A9_des_Op%C3%A9rations_Binaires_sur_les_bytes_et_les_shorts_en_Java) [archive]).

Expressions booléennes

Les expressions booléennes employant les opérateurs ET logique (`&&`) et OU logique (`||`) sont évaluées de manière paresseuse. C'est à dire que l'évaluation s'arrête aussitôt que le résultat est déterminé.

Exemple avec l'opérateur ET logique (vrai si les deux opérandes sont vrais), si le premier opérande est faux le résultat est faux et le deuxième opérande n'est pas évalué. Si le deuxième opérande est le résultat de l'appel à une méthode,

cette méthode ne sera pas appelée :

```
String s = getText();
if ((s!=null) && (s.length()>0))
// si s vaut null, la longueur n'est pas testée sinon cela provoquerait une exception.
{ /*...*/ }
```

Parfois ce comportement n'est pas désirable. Dans ces cas-là, il faut utiliser les opérateurs binaires équivalents ET (`&`) et OU (`|`).

Le remplacement dans l'exemple précédent serait cependant une erreur :

```
String s = getText();
if ((s!=null) & (s.length()>0))
// si s vaut null, la longueur est tout de même testée et provoque donc une exception !
{ /*...*/ }
```

Savoir si une racine carrée est entière

La fonction calculant les racines carrées pouvant trouver des nombres à virgules, voici une courte technique pour savoir s'il s'agit d'un entier :

```
public static void main(String[] args) {
  int n = Integer.parseInt(args[0]);
  double r = (double)Math.sqrt(n);
  return (r*r == n);
}
```

Conditions

En Java, les séquences d'instructions *if...else* et *switch..case* permettent de formuler des traitements conditionnels.

Instructions if...else

L'instruction *if* permet d'exécuter une instruction (ou un bloc d'instructions) si une condition est remplie :

```
if (condition) {
    // instructions à exécuter
}
```

L'instruction *else* complète l'instruction *if*. Elle permet d'exécuter une instruction si la condition indiquée par le *if* n'est pas remplie :

```
if (condition) {
    // instructions à exécuter
} else {
    // instructions à exécuter si la condition n'est pas remplie
}
```

On peut aussi enchaîner les tests conditionnels les uns à la suite des autres :

```
if (condition1) {
    // bloc1
} else if (condition2) {
    // bloc2
} else {
    // bloc3
}
```

Instructions switch...case

La séquence d'instructions *switch* permet d'exécuter différentes instructions (ou blocs d'instructions) en fonction d'une liste de cas. L'instruction *default* permet de définir un cas par défaut.

```
switch (expression)
{
    case constante1:
        // instructions à exécuter
        break;
    case constante2:
        // instructions à exécuter
        break;
    // ...
    default:
        // instructions à exécuter
        break;
}
```

L'instruction *break* sert à sortir du test. Dans le cas contraire, les instructions du *case* suivant seraient également exécutées. Pour plus de précision, voir la section suivante (itération).

Remarque : le dernier *break* est facultatif (qu'il s'agisse de la clause *default* dans le cas précédent, ou d'une clause *case*). Cependant ajouter ce dernier *break* améliore la maintenabilité du code en cas d'ajout d'un nouveau cas (*case* ou *default*) ou de déplacement du dernier cas.

Attention : l'instruction *switch* ne fonctionne que sur des types simples : int, short, byte et char, ainsi que sur les wrappers correspondants (Integer, Short, Byte et Character).

Dans les versions récentes de Java, le *switch* accepte aussi le type énumération `enum` (depuis JDK 6) et le type `String` (depuis JDK 7).

Itérations

Java offre deux instructions permettant de réaliser des traitements itératifs : *while*, *for*. De plus les instructions *break* et *continue* permettent d'altérer le déroulement des boucles.

Instruction while

L'instruction *while* continue d'exécuter le traitement situé dans la boucle tant que la condition indiquée est vérifiée :

```
while (condition) {
    // instructions à exécuter
}
```

Une autre forme du *while* évalue la condition à la fin de chaque cycle, plutôt qu'au début. Le bloc d'instruction est donc toujours exécuté au moins une fois :

```
do {
    // instructions à exécuter
} while (condition);
```

Instruction for

L'instruction *for* permet d'exécuter un traitement de manière répétitive, mais en spécifiant différemment la condition d'arrêt.

À l'intérieur des parenthèses se trouvent trois blocs d'instructions séparés par un point-virgule. L'exécution de la boucle *for* commence par exécuter le premier bloc, qui sert généralement à initialiser une variable. Ensuite le second bloc est exécuté (la condition). Si la condition retourne la valeur booléenne *true*, le corps de la boucle *for* est exécuté (le bloc d'instruction entre accolade). Sinon l'exécution de la boucle *for* se termine. Si le corps de la boucle a été exécuté, le troisième bloc est exécuté (incrémenter une variable en général) et le processus recommence : la condition est évaluée, etc.

Voici un exemple typique d'une boucle *for* qui défini un compteur pour exécuter la boucle *i* fois. En d'autres termes, le traitement est effectué tant que le compteur n'a pas atteint une limite :

```
for (int i=0 ; i<limite ; i=i+increment) {
    // instructions à exécuter
}
```

Environnement concurrentiel

L'accès aux données par l'intermédiaire des itérateurs est par essence séquentiel, deux threads ne peuvent bien sur pas utiliser le même itérateur, un objet ne peut pas fournir un itérateur à plusieurs threads. Il est conseillé d'utiliser dans ce cas les classes spécialisées de java.util.concurrent

Sauts dans le code

Java permet de faire des sauts dans le code lors de l'exécution d'une boucle, c'est en général une mauvaise idée et il faut, autant que possible éviter de recourir à ces pratiques. Il s'agit des instructions *break* qui permet de sortir immédiatement de la boucle (c'est-à-dire de sauter immédiatement à la première instruction qui suit la boucle) ou de l'instruction *continue* qui permet de ne pas exécuter le reste de l'itération en cours, c'est-à-dire de sauter

immédiatement au bloc d'incrémentation et à une nouvelle évaluation de la condition.

Pour les collections

Le chapitre Collections montrera que ce type d'objet possède sa propre itération : *pour chaque objet d'une collection*.

Tableaux

Un tableau est un ensemble indexé de données d'un même type. L'utilisation d'un tableau se décompose en trois parties :

- Création du tableau
- Remplissage du tableau
- Lecture du tableau

Création d'un tableau

Un tableau se déclare et s'instancie comme une classe :

```
int monTableau[ ] = new int[10];
```

ou

```
int [ ] monTableau = new int[10];
```

L'opérateur [ ] permet d'indiquer qu'on est en train de déclarer un tableau.

Dans l'instruction précédente, nous déclarons un tableau d'entiers (int, integer) de taille 10, c'est-à-dire que nous pourrons stocker 10 entiers dans ce tableau.

Si [ ] suit le type, toutes les variables déclarées seront des tableaux, alors que si [ ] suit le nom de la variable, seule celle-ci est un tableau :

```
int [] premierTableau, deuxiemeTableau;
float troisiemeTableau[], variable;
```

Dans ces quatre déclarations, seule *variable* n'est pas un tableau.

Remplissage d'un tableau

Une fois le tableau déclaré et instancié, nous pouvons le remplir :

```
int [] monTableau = new int[10];

monTableau[5] = 23;
```

L'indexation démarre à partir de 0, ce qui veut dire que, pour un tableau de N éléments, la numérotation va de 0 à N-1.

Dans l'exemple ci-dessus, la 6ème case contient donc la valeur 23.

Nous pouvons également créer un tableau en énumérant son contenu :

```
int [] monTableau = {5,8,6,0,7};
```

Ce tableau contient 5 éléments.

Lorsque la variable est déjà déclarée, nous pouvons lui assigner d'autres valeurs en utilisant l'opérateur `new` :

```
monTableau = new int[]{11,13,17,19,23,29};
```

Lecture d'un tableau

Pour lire ou écrire les valeurs d'un tableau, il faut ajouter l'indice entre crochets (`[` et `]`) à la suite du nom du tableau :

```
int [] monTableau = {2,3,5,7,11,23,17};
int nb;

monTableau[5] = 23;  // ->  2 3 5 7 11 23 17
nb = monTableau[4];  // 11
```

L'indice 0 désigne le premier élément du tableau.

L'attribut `length` d'un tableau donne sa longueur (le nombre d'éléments). Donc pour un tableau nommé `monTableau` l'indice du dernier élément est `monTableau.length-1`.

Ceci est particulièrement utile lorsque nous voulons parcourir les éléments d'un tableau.

```
for (int i = 0; i < monTableau.length; i++) {
    int element = monTableau[i];
    // traitement
}
```

Les tableaux en Java 5

Java 5 fournit un moyen plus court de parcourir un tableau.

L'exemple suivant réalise le traitement sur monTableau :

```
for (int element : monTableau){
   // traitement
}
```

Attention néanmoins, la variable `element` contient une copie de `monTableau[i]`. Avec des tableaux contenant des variables primitives, toute modification de `element` n'aura aucun effet sur le contenu du tableau.

```
// Vaine tentative de remplir tous les éléments du tableau avec la valeur 10
for(int element : monTableau){
    element=10;
}

// La bonne méthode :
for(int i=0; i < monTableau.length; i++){
    monTableau[i]=10;
}
```

Tableaux à plusieurs dimensions

En Java, les tableaux à plusieurs dimensions sont en fait des tableaux de tableaux.

Exemple, pour allouer une matrice de 5 lignes de 6 colonnes :

```
int[][] matrice=new int[5][];
```

```
for (int i=0 ; i<matrice.length; i++)
    matrice[i]=new int[6];
```

Java permet de résumer l'opération précédente en :

```
int[][] matrice=new int[5][6];
```

La première version montre qu'il est possible de créer un tableau de tableaux n'ayant pas forcément tous la même dimension.

On peut également remplir le tableau à la déclaration et laisser le compilateur déterminer les dimensions des tableaux, en imbriquant les accolades :

```
int[][] matrice =
    {
        { 0, 1, 4, 3 } , // tableau [0] de int
        { 5, 7, 9, 11, 13, 15, 17 } // tableau [1] de int
    };
```

Pour déterminer la longueur des tableaux, on utilise également l'attribut `length` :

```
matrice.length     //  2
matrice[0].length  //  4
matrice[1].length  //  7
```

De la même manière que précédement, on peut facilement parcourir tous les éléments d'un tableau :

```
int i, j;
for(i=0; i<matrice.length; i++) {
    for(j=0; j<matrice[i].length; j++) {
        //Action sur matrice[i][j]
    }
}
```

La classe Arrays

La classe `Arrays` du package `java.util` possède plusieurs méthodes de gestion des tableaux de types de base, et d'objets :

- la méthode `asList` convertit un tableau en liste,
- la méthode `binarySearch` effectue la recherche binaire d'une valeur dans un tableau,
- la méthode `equals` compare deux tableaux (longueur et contenu),
- la méthode `fill` remplit un tableau avec la valeur donnée,
- la méthode `sort` trie un tableau dans l'ordre croissant de ses éléments.

Exemple de source :

```
import java.util.*;

public class ArrayExample {

  static Scanner input = new Scanner(System.in);

  public static void main(String[] args) {
    int numNoms = getInt("Nombre d'entrées du tableau ?");
    String[] noms = new String[numNoms];
    for (int i = 0; i < noms.length; i++) {
```

```
      noms[i] = getString("Entrée n°" + (i+1));
    }
    for (int i = 0; i < noms.length; ++i) {
      System.out.println(noms[i]);
    }
  }

  public static int getInt(String prompt) {
    System.out.print(prompt + " ");
    int entier = input.nextInt();
    input.nextLine(); // Get rid of this line
    // so that getString won't read it
    return entier;
  }

  public static String getString(String prompt) {
    System.out.print(prompt + " ");
    return input.nextLine();
  }
}
```

Copie d'un tableau

La copie d'un tableau implique la copie de ses éléments dans un autre tableau. Dans le cas d'un tableau d'objets, seules les références à ces objets sont copiées, aucun nouvel objet n'est créé.

La méthode `arraycopy` de la classe `System` permet de copier tout ou partie d'un tableau vers un autre tableau déjà alloué.

Comme toutes les classes, les tableaux dérivent de la classe `java.lang.Object`. Les méthodes de la classe Object sont donc utilisables :

```
int[] premiers = { 2, 3, 5, 7, 11 };
System.out.println( premiers.toString() ); // Par défaut <type>@<hashcode>, exemple : [I@a10829
```

La copie intégrale d'un tableau dans un nouveau tableau peut donc se faire en utilisant la méthode `clone()`. La valeur retournée par cette méthode étant de type `Object`, il faut la convertir dans le type concerné.

Exemple:

```
int[] nombres = { 2, 3, 5, 7, 11 };
int[] copie = (int[]) nombres.clone();
nombres[1]=4; // nombres contient 2 4 5 7 11
// tandis que copie contient toujours 2 3 5 7 11
```

Commentaires

Un commentaire permet d'insérer du texte qui ne sera pas compilé ni interprété. Il sert à ajouter du texte au code source.

Il est utile pour expliquer ce que fait le code source :

- expliquer le choix technique effectué : pourquoi tel algorithme et pas un autre, pourquoi appeler telle méthode, ...
- expliquer ce qui devra être fait par la suite (liste de chose à faire) : amélioration, problème à corriger, ...
- donner les explications nécessaires à la compréhension du code pour le reprendre soi-même plus tard, ou pour d'autres développeurs.

Il peut aussi être utilisé pour que le compilateur ignore une partie du code : code temporaire de débogage, code en cours de développement, ...

Il est également utile pour la documentation des classes.

Syntaxe

Les commentaires en Java utilisent la même syntaxe qu'en C++.

Un commentaire de fin de ligne commence par un double slash et se termine au retour à la ligne.

Exemple :

```
// Un commentaire pour donner l'exemple

int n = 10; // 10 articles
```

Un commentaire sur plusieurs lignes est encadré par slash + étoile, et étoile + slash.

Exemple :

```
/*
Ceci est un commentaire
sur plusieurs lignes.
*/

/*
Code de débogage désactivé :

int a=10;
while (a-->0) System.out.println("DEBUG: tab["+a+"]="+tab[a]);
*/
```

Documentation des classes

Java permet de documenter les classes et leurs membres en utilisant une syntaxe particulière des commentaires.

Syntaxe

Un commentaire de documentation est encadré par slash-étoile-étoile et étoile-slash (soit /** ... */). La documentation est au format HTML.

Exemple :

```
/**
```

```
    Une classe pour donner un <b>exemple</b> de documentation HTML.
*/
public class Exemple {
    /** ...Documentation du membre de type entier nommé exemple... */
    public int exemple;
}
```

Le commentaire de documentation se place juste avant l'entité commentée (classe, constructeur, méthode, champ).

Dans un commentaire de documentation, la première partie est un texte de description au format HTML. La seconde partie est une liste d'attributs spéciaux dont le nom commence par un arobase (@). Ces derniers sont interprétables par le compilateur et appelés *annotations*.

Exemple pour la méthode suivante :

```
/**
    Obtenir la somme de deux entiers.
    @param a Le premier nombre entier.
    @param b Le deuxième nombre entier.
    @return La valeur de la somme des deux entiers spécifiés.
*/
public int somme(int a, int b) {
    return a + b;
}
```

`Obtenir la somme de deux entiers.`
: Description de la méthode somme.

`@param a Le premier nombre entier.`
: Attribut de description du paramètre a de la méthode.

`@param b Le deuxième nombre entier.`
: Attribut de description du paramètre b de la méthode.

`@return La valeur de la somme des deux entiers spécifiés.`
: Attribut de description de la valeur retournée par la méthode.

Voici une liste non exhaustive des attributs spéciaux :

Attribut et syntaxe	Dans un commentaire de ...	Description
@author *auteur*	classe	Nom de l'auteur de la classe.
@version *version*	classe	Version de la classe.
@deprecated *description*	classe, constructeur, méthode, champ	Marquer l'entité comme obsolète (ancienne version), décrire pourquoi et par quoi la remplacer. Si l'entité marquée comme obsolète par cet attribut est utilisée, le compilateur donne un avertissement.
@see *référence*	classe, constructeur, méthode, champ	Ajouter un lien dans la section "Voir aussi".
@param *description de l'id*	constructeur et méthode	Décrire un paramètre de méthode.
@return *description*	méthode	Décrire la valeur retournée par une méthode.
@exception *description du type*	constructeur et méthode	Décrire les raisons de lancement d'une exception du type spécifié (clause `throws`).

Documentation

Le JDK fournit un outil nommé javadoc qui permet de générer la documentation des classes correctement commentées.

La commande javadoc sans argument donne la syntaxe complète de la commande.

Exemple : pour une classe nommée `Exemple` définie dans un *package* nommé `org.wikibooks.fr` dans le fichier `C:\ProgJava\org\wikibooks\fr\Exemple.java` :

```java
package org.wikibooks.fr;

/**
    Une classe d'exemple.
*/
public class Exemple {
    /**
        Obtenir la somme de deux entiers.
        @param a Le premier nombre entier.
        @param b Le deuxième nombre entier.
        @return La valeur de la somme des deux entiers spécifiés.
    */
    public int somme(int a, int b) {
        return a + b;
    }
}
```

La documentation peut être générée dans un répertoire spécifique (C:\ProgDoc par exemple) avec la commande suivante :

```
javadoc -locale fr_FR -use -classpath C:\ProgJava -sourcepath C:\ProgJava -d C:\ProgDoc org.wik
```

Les options de cette commande sont décrits ci-dessous :

`-locale fr_FR`
: La documentation est en français.

`-use`
: Créer les pages sur l'utilisation des classes et paquetages (*packages*).

`-classpath C:\ProgJava`
: Le chemin des classes compilées (*.class).

`-sourcepath C:\ProgJava`
: Le chemin des classes sources (*.java).

`-d C:\ProgDoc`
: Le chemin où la documentation doit être générée.

`org.wikibooks.fr`
: Le nom du paquetage (*package*) à documenter. Il est possible de spécifier plusieurs paquetages, ou un ou plusieurs noms de classe pour ne documenter que celles-ci.

La page de description d'un paquetage copie le texte de description à partir d'un fichier nommé `package.html` qui doit se situer dans le répertoire correspondant. Dans notre exemple, il faut documenter le paquetage dans le fichier `C:\ProgJava\org\wikibooks\fr\package.html`.

Dans les versions récentes de Java, le fichier `package.html` peut être remplacé par un fichier Java spécial nommé `package-info.java` contenant uniquement la déclaration du paquetage (*package*) précédée d'un commentaire de documentation.

Exemple (`C:\ProgJava\org\wikibooks\fr\package-info.java`) :

```java
/**
  Ce paquetage fictif sert à illustrer le livre sur Java
```

```
  de <i>fr.wikibooks.org</i>.
*/
package org.wikibooks.fr;
```

Les classes en Java

La notion de classe constitue le fondement de la programmation orientée objet. Une classe est la déclaration d'un type d'objet.

En Java, les classes sont déclarées à l'aide du mot-clef *class*, suivi du nom de la classe déclarée, suivi du corps de la classe entre accolades. Par convention, un nom de classe commence par une majuscule.

```
public class MaClasse
{
    // corps de la classe
}
```

Le fichier contenant cette déclaration doit avoir pour extension *.java*. Un fichier peut contenir plusieurs déclarations de classes (ce n'est pas recommandé, il faut partir du principe 1 classe = 1 fichier, pour des problèmes évidents de relecture du code, devoir modifier du code où plusieurs classes sont écrites dans un seul fichier est le meilleur moyen de faire n'importe quoi), mais il ne peut contenir qu'au plus une classe dite *publique* (dont le mot-clef *class* est précédé de *public*, comme dans l'exemple ci-dessus). Le fichier doit obligatoirement porter le même nom que cette classe publique : dans l'exemple ci-dessus, il faudrait donc sauver notre classe dans un fichier nommé *MaClasse.java*.

Un fichier *.java* peut commencer par une ou plusieurs déclarations d'*import*. Ces imports ne sont pas indispensables, mais autorisent en particulier l'accès aux classes prédéfinies sans avoir à spécifier leur chemin d'accès complet dans les collections de classes prédéfinies (organisées en *packages*). Dans le code ci-dessous, on souhaite par exemple utiliser la classe prédéfinie *Vector* (un type de données comparable à des tableaux dont la taille peut varier dynamiquement). Dans la sous-collection de classes prédéfinies "java", cette classe se trouve dans la sous-collection "util" (ou encore : cette classe est dans le package "java.util").

Sans import, il faut spécifier le nom complet de la classe (*packages* inclus) :

```
public class MaClasse
{
    // ...
    public static void main(String[] args)
    {
        // sans l'import :
        java.util.Vector v = new java.util.Vector();
        // ...
    }
}
```

Avec import, seul le nom de la classe (sans *packages*) utilisée est nécessaire :

```
import java.util.Vector;

public class MaClasse
{
    // ...
    public static void main(String[] args)
    {
        // avec l'import :
        Vector v = new Vector();
        // ...
    }
}
```

Quand plusieurs classes du même *package* sont utilisées, l'import peut utiliser le caractère étoile. Une classe peut donc avoir une longue liste d'import :

```
import java.util.Vector;
```

```
import java.util.ArrayList;
import java.util.HashMap;

public class MaClasse
{
    // ...
    public static void main(String[] args)
    {
        Vector    v = new Vector();
        ArrayList a = new ArrayList();
        HashMap   m = new HashMap();
    }
}
```

ou une liste plus courte :

```
import java.util.*;

public class MaClasse
{
    // ...
    public static void main(String[] args)
    {
        Vector    v = new Vector();
        ArrayList a = new ArrayList();
        HashMap   m = new HashMap();
    }
}
```

Enfin, la définition d'une classe peut aussi être précédée d'une (et une seule) déclaration de *package*, qui indique à quel emplacement se trouve le fichier dans l'arborescence des répertoires d'un projet Java. Par exemple, si la racine de notre projet Java est le répertoire /home/user/monProjet/ (ou c:\Mes documents\monProjet\ sous Windows) et que notre classe se trouve dans le sous-répertoire org/classes/ (soit, respectivement /home/user/monProjet/org/classes/ et c:\Mes documents\monProjet\org\classes\) nous aurons une déclaration du package de la forme :

```
package org.classes;

public class MaClasse
{
    // contenu de la classe
}
```

Important

Le package `java.lang` est automatiquement importé dans tous les programmes, il permet d'utiliser les types primitifs.

Instanciation d'une classe

Un *objet* peut être vu comme un ensemble de données regroupées à la manière des structures de C ou des enregistrements de Pascal. Une *classe* définit un modèle d'objet. Chaque objet créé à partir d'une classe est appelé *instance* de cette classe. L'ensemble de données internes d'un objet - les *champs* de cet objet - est spécifié par sa classe. Une classe peut en outre contenir : d'une part, des *constructeurs* - du code destiné à l'initialisation de ses instances au moment de leur création ; d'autre part des *méthodes* - du code destiné à la gestion des données internes de chacune de ses instances. On dit que ces méthodes sont *invocables* sur les objets.

Les objets se manipulent à l'aide de variables appelées *références*. Une référence vers un objet est l'analogue d'un pointeur vers celui-ci, au sens usuel de la notion de pointeur dans les langages impératifs tels que C - on dit que cette référence *pointe* sur l'objet, ou encore qu'elle *référence* cet objet. Notez que par abus de langage, mais par abus de langage seulement, on confond en pratique une référence vers un objet et l'objet lui-même. On parlera ainsi de "l'objet

`r`" plutôt que de "l'objet référencé par `r`" ou "l'objet pointé par `r`". Pour autant, une référence vers un objet **n'est pas** l'objet lui-même : une référence peut ne pointer vers aucun objet - en particulier si elle est déclarée sans être initialisée, ou si on lui affecte la *valeur de référence nulle*, notée `null` ; elle peut pointer successivement vers des objets différents au cours du temps ; d'autre part, deux références distinctes peuvent pointer vers le même objet - les champs de l'objet sont alors accessibles, et les méthodes de sa classe deviennent invocables via l'une ou l'autre de ces deux références, avec le même effet.

Le type d'une référence spécifie la (ou même les) classe(s) des objets sur lesquels elle est susceptible de pointer. Une référence déclarée de type "référence vers `MaClasse`", où `MaClasse` est le nom d'une classe, est susceptible de pointer vers n'importe quelle instance de `MaClasse`.

```
MaClasse r; // déclaration d'une référence de type "référence vers MaClasse"

/* instructions diverses */

r = new MaClasse(); // création d'une instance de MaClasse, puis référencement par r de l'objet
```

On peut déclarer une référence et lui faire immédiatement référencer une nouvelle instance, créée dans cette déclaration :

```
MaClasse r = new MaClasse(); // on crée une instance de MaClasse, que l'on référence par r
```

Membres

Introduction

Les membres d'une classe sont les méthodes (traitements) et attributs (données) qui en font partie.

Exemple :

```
public class Horloge
{
  // Définition des attributs
  int heures;
  int minutes;
  int secondes;

  // Définition des méthodes
  public void definitHeure()
  {
    heures = 12;
    minutes = 30;
    secondes = 30;
  }

  public void incrementeHeure()
  {
    secondes++;
    if (secondes==60)
    {
      secondes=0;
      minutes++;
      if (minutes==60)
      {
        minutes=0;
        heures++;
        if (heures==24)
        {
          heures=0;
        }
      }
    }
  }

  protected void afficheHeure()
  {
    System.out.println("Il est "+heures+":"+minutes+":"+secondes);
  }

  public static void main (String[] args)
  {
    Horloge montre = new Horloge(); // Nouvelle instance de la classe

    // Accès aux membres de la classe de l'objet avec le caractère point : <objet>.<membre>
    montre.definitHeure();
    for (int i=0 ; i<10 ; i=i+1)
    {
      montre.incrementeHeure();
      montre.afficheHeure();
    }
  }
}
```

Dans cet exemple, la classe *Horloge* contient trois attributs servant à représenter l'heure (*heures*, *minutes* et *secondes*) et trois méthodes (*definitHeure* qui initialise, *incrementeHeure*, qui augmente l'heure d'une seconde, et *afficheHeure*, qui affiche l'heure).

On distingue parmi les méthodes une catégorie particulière, les constructeurs.

Déclaration

Attribut

La déclaration d'un attribut se fait de la manière suivante :

```
modificateurs type nom;
```

Constructeur

La déclaration d'un constructeur se fait de la manière suivante :

```
modificateur d'accès nomDeLaClasse(type et nom des paramètres éventuels) { corps du constructeur }
```

Méthode

La déclaration d'une méthode se fait comme suit :

```
modificateurs typeDeRetour nom(type et nom des paramètres éventuels) { corps de la méthode }
```

Méthode à nombre de paramètres variable

Java 5 introduit un mécanisme permettant d'écrire des méthodes acceptant un nombre variable d'arguments ("varargs"), alors qu'il fallait passer par un tableau ou autre artifice similaire auparavant. La syntaxe est très similaire à la syntaxe utilisée pour la fonction printf en C, ce qui a permis d'ajouter une méthode `printf` dans la classe `PrintStream`, ce qui permet de faire `System.out.printf("what ever %d",5);`

Exemple :

```
public class Message
{

    public void message(String recette, String... arguments)
    {
      System.out.print("Ingrédients : \n");
      for (String s : arguments)
        System.out.println(s);
    }

    public static void main (String[] args)
    {
      Message menu = new Message();
      menu.message("déjeuner","ail","oignon","échalote");
    }
}
```

Bloc d'initialisation

Une classe peut comporter un ou plusieurs blocs d'instructions servant à initialiser toute instance de la classe, ou à initialiser des membres statiques si le bloc est déclaré statique (`static`).

Exemple :

```
public class ImageFile
```

```
{
    final int MAX_SIZE; // final -> variable initialisée une seule fois

    // Bloc d'initialisation
    {
        int maxsize;
        try
        {
            maxsize = Integer.parseInt(System.getProperty("file.maxsize"));
        }
        catch(Exception ex) // propriété "file.maxsize" non définie (NullPointerException) ou n
        {
            maxsize = 1000; // valeur par défaut
        }
        MAX_SIZE = maxsize; // variable initialisée une seule fois
    }

    ImageFile(File f){ /* ... */ }
    ImageFile(File f, int width){ /* ... */ }
    ImageFile(File f, int width, int height){ /* ... */ }
}
```

Les instructions contenues dans les blocs d'initialisation sont appelées avant les instructions du constructeur utilisé. Un bloc d'initialisation permet donc de :

- gérer les exceptions (contrairement à une affectation dans la déclaration de la variable membre),
- éviter la duplication du code dans les différents constructeurs.

La classe peut aussi comporter des blocs d'initialisation statiques, qui sont des blocs d'instructions précédés du modificateur `static`, et qui sont exécutés au chargement de la classe, juste après l'initialisation des attributs statiques.

Exemple :

```
public class Exemple
{
    static
    {
        System.out.println("La classe Exemple est chargée");
    }
}
```

Un bloc d'initialisation statique permet de :

- gérer les exceptions (contrairement à une affectation dans la déclaration de la variable membre),
- charger la bibliothèque native implémentant les méthode natives de la classe.

Modificateur

Modificateurs d'accès

En Java, la déclaration d'une classe, d'une méthode ou d'un membre peut être précédée par un modificateur d'accès.

Un modificateur indique si les autres classes de l'application pourront accéder ou non à la classe/méthode/membre (qualifié par la suite d'« item »).

Ces modificateurs sont au nombre de quatre :

- *public* : toutes les classes peuvent accéder à l'item
- *protected* : seules les classes dérivées et les classes du même package peuvent accéder à l'item
- *private* : l'item est seulement accessible depuis l'intérieur de la classe où il est défini.
- (par défaut) : sans modificateur d'accès, seules les classes du même package peuvent accéder à l'item.

L'utilisation des modificateurs permet au programmeur de contrôler la visibilité des différents items et permet d'empêcher que des actions illégales soient effectuées sur les items.

abstract

Le modificateur `abstract` indique qu'une classe ou méthode est abstraite.

final

Ajouté devant un attribut, il le rend immuable, dès lors qu'il est initialisé (autrement dit, il n'est pas obligatoire de l'initialiser dès la déclaration, contrairement à d'autres langages). Pour les types primitifs, `final` fige la valeur, pour les objets, `final` fige la **référence**, et non la **valeur** de la référence (i.e. seule l'instanciation est figée).

Devant une variable locale (dans une méthode, donc), il a le même comportement que pour un attribut.

Devant une méthode, il indique que cette méthode ne peut pas être modifiée dans une classe dérivée. Les méthodes `static` et `private` sont implicitement `final`.

Devant une classe, il indique que cette classe ne peut pas avoir de sous-classe.

static

Le modificateur static indique, pour une méthode, qu'elle peut être appelée sans instancier sa classe (syntaxe : Classe.methode()).

Pour un attribut, qu'il s'agit d'un attribut de classe, et que sa valeur est donc partagée entre les différentes instances de sa classe.

De plus, il est possible de déclarer dans une classe un bloc d'initialisation statique, qui est un bloc d'instruction précédé du modificateur `static`.

synchronized

Le modificateur `synchronized` indique que la méthode ne peut être exécutée que par un thread à la fois. Le verrou ne s'active que pour l'objet sur lequel la méthode a été appelée (une même méthode synchronized peut être exécutée en même temps par deux threads différents sur deux objets différents).

transient

Le modificateur `transient` indique que lors de la sérialisation de l'objet, cet attribut n'est pas sérialisé et donc il est ignoré. Cela signifie que lorsque l'on désérialise l'objet, l'attribut portant le modificateur `transient` n'est pas défini dans l'objet désérialisé. Il s'agit en général d'attributs qui peuvent être recalculés à partir des autres attributs de l'objet.

native

Ce modificateur permet d'indiquer que cet item est défini dans une bibliothèque externe écrite dans un autre langage de programmation, utilisant l'API JNI.

strictfp

Pour une méthode, une classe ou une interface, le modificateur `strictfp` (abréviation de *strict floating point*) force la JVM à évaluer les opérations à virgules flottantes (sur les `double` et `float`) conformément à la spécification Java, c'est-à-dire de la gauche vers la droite. Cela permet d'avoir un comportement identique d'une JVM à une autre et d'éviter certains dépassements de valeur limite pour les résultats intermédiaires.

volatile

Pour une variable, le modificateur `volatile` force la JVM, avant et après chaque utilisation de la variable, à la rafraîchir à partir de la mémoire principale au lieu d'utiliser un cache local. Cela permet de synchroniser la valeur de la variable entre plusieurs threads.

Héritage

L'héritage, l'un des mécanismes les plus puissants de la programmation orientée objet, permet de reprendre des membres d'une classe (appelée *superclasse* ou *classe mère*) dans une autre classe (appelée *sous-classe*, *classe fille* ou encore *classe dérivée*), qui en hérite. De cette façon, on peut par exemple construire une classe par héritage successif.

En Java, ce mécanisme est mis en œuvre au moyen du mot-clé *extends*

Exemple :

```
public class Vehicule
{
    public int vitesse;
    public int nombre_de_places;
}

public class Automobile extends Vehicule
{
    public Automobile()
    {
        this.vitesse = 90;
        this.nombre_de_places = 5;
    }
}
```

Dans cet exemple, la classe *Automobile* hérite de la classe *Vehicule*, ce qui veut dire que les attributs *vitesse* et *nombre_de_places*, bien qu'étant définis dans la classe *Vehicule*, sont présents dans la classe *Automobile*. Le constructeur défini dans la classe *Automobile* permet d'ailleurs d'initialiser ces attributs.

En Java, le mécanisme d'héritage permet de définir une hiérarchie de classes comprenant toutes les classes. En l'absence d'héritage indiqué explicitement, une classe hérite implicitement de la classe *Object*. Cette classe *Object* est la racine de la hiérarchie de classe.

La classe *Object*

Au moment de l'instanciation, la classe fille reçoit les caractéristiques qui peuvent être héritées de sa super-classe, qui elle-même reçoit les caractéristiques qui peuvent être héritées de sa propre superclasse, et ce récursivement jusqu'à la classe *Object*.

Ce mécanisme permet de définir des classes génériques réutilisables, dont l'utilisateur précise le comportement dans des classes dérivées plus spécifiques.

Il faut préciser que, contrairement à C++, Java ne permet pas l'héritage multiple, ce qui veut dire qu'une classe dérive toujours d'une et d'une seule classe.

Héritage d'interface

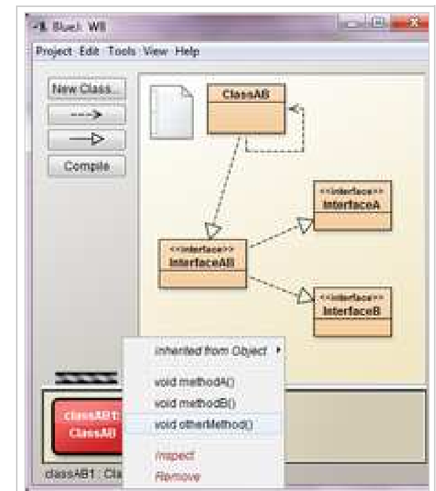
L'héritage d'interface est aussi possible en Java. À la différence des classes, l'héritage multiple est autorisé, ce qui veut dire qu'une interface peut hériter d'autant d'autres interfaces que désiré.

L'héritage d'interface se fait avec le mot clé **extends**, puisque c'est une interface qui "étend" une interface existante. Les différentes interfaces héritées sont séparées par une virgule.

L'héritage « multiple » pour une classe (via les interfaces) se fait avec le mot-clé **implements**. Exemple :

```
public interface InterfaceA {
   public void methodA();
}
```

```java
public interface InterfaceB {
   public void methodB();
}

public interface InterfaceAB extends InterfaceA, InterfaceB {
   public void otherMethod();
}

public class ClassAB implements InterfaceAB{

   public void methodA(){
       System.out.println("A");
   }

   public void methodB(){
       System.out.println("B");
   }
   public void otherMethod(){
       System.out.println("blablah");
   }

   public static void main(String[] args) {
       ClassAB classAb = new ClassAB();
       classAb.methodA();
       classAb.methodB();
       classAb.otherMethod();
   }
}
```



Exécution de cet exemple avec BlueJ.

Le mot-clé *super*

Le mot-clé *super* permet d'accéder aux membres de la super-classe d'une classe, de la même manière que l'on accède aux attributs de la classe elle-même à l'aide du mot-clé *this*.

Exemple :

```java
public class Avion extends Vehicule
{
    public Avion()
    {
        super();
    }
}
```

Dans cet exemple, le constructeur de la classe *Avion* fait appel au constructeur de la classe *Vehicule*.

Ce mot-clé permet également d'accéder explicitement aux membres de la classe de base, dans le cas, par exemple, où il existe déjà un membre portant ce nom dans la classe (surcharge de méthode, ...).

Exemple :

```java
public class Vehicule
{
    // ...
    public void rouler() throws Exception
    {
        position += vitesse;
    }
}

public class Avion extends Vehicule
{
    // ...
    public void rouler() throws Exception
    {
```

```
        if (altitude>0)
           throw new Exception("L'avion en vol ne peut rouler");
        else super.rouler();
    }
}
```

Lien externe

- L'héritage en Java (http://www.commentcamarche.net/java/javaherit.php3) [archive]

Encapsulation

En Java, comme dans beaucoup de langages orientés objet, les classes, les attributs et les méthodes bénéficient de niveaux d'accessibilité, qui indiquent dans quelles circonstances on peut accéder à ces éléments.

Ces niveaux sont au nombre de 4, correspondant à 3 mots-clés utilisés comme modificateurs : *private*, *protected* et *public*. La quatrième possibilité est de ne pas spécifier de modificateur (comportement par défaut).

Comportement par défaut

Si aucun modificateur n'est indiqué, l'élément n'est accessible que depuis les classes faisant partie du même *package*.

Exemple :

```
package com.moimeme.temps;

class Horloge
{
    // corps de la classe
}

public class Calendrier
{
    void ajouteJour()
    {
        // corps de la methode
    }

    int mois;

    // suite de la classe
}
```

La classe *Horloge*, la méthode *ajouteJour* et l'attribut *mois* ne sont accessibles que depuis les classes faisant partie du *package com.moimeme.temps*.

Modificateur "private"

Un attribut ou une méthode déclarée "private" n'est accessible que depuis l'intérieur même de la classe.

Modificateur "protected"

Un attribut ou une méthode déclarée "protected" est accessible uniquement aux classes d'un package et à ses sous-classes même si elles sont définies dans un package différent.

Modificateur "public"

Une classe, un attribut ou une méthode déclarée "public" est visible par toutes les classes et les méthodes.

En résumé

Le tableau résume les différents mode d'accès des membres d'une classe.

Modificateur du membre	**`private`**	***aucun***	**`protected`**	**`public`**
Accès depuis la classe	Oui	Oui	Oui	Oui
Accès depuis une classe du même `package`	Non	Oui	Oui	Oui
Accès depuis une sous-classe	Non	Non	Oui	Oui
Accès depuis toute autre classe	Non	Non	Non	Oui

Polymorphisme

Le polymorphisme veut dire que le même service, aussi appelé opération ou méthode, peut avoir un comportement différent selon les situations.

Polymorphisme paramétrable

Plusieurs signatures pour une même méthode

On peut donner à une même méthode, plusieurs signatures pour implémenter des comportements différents selon les types des paramètres passés. La signature d'une méthode est composée du nom de celle ci, de sa portée, du type de donnée qu'elle renvoie et enfin du nombre et du type de ses paramètres.

```
public class A {

  private int a;

  public A() { //constructeur 1
    System.out.println("Création de A");
  }

  public A(int a) { //constructeur 2 par surcharge
    this.a = a;
    System.out.println("Création de A");
  }

  public int getter() {
    return this.a;
  }

  public void setter(int a) {
    this.a = a;
  }

  public static void main(String[] args) {
    A premiera = new A(); //construction par 1
    A seconda = new A(1); //construction par 2
  }

}
```

Proposer le passage d'un nombre inconnu de paramètres

Dans la signature d'une méthode , on peut préciser qu'il est possible de passer plus de 1 paramètre du même type en suffixant le type du paramètre avec « `...` ».

```
// supposons la méthode suivante :
public String concatenation(String... elements) {
    // pour l'implementation, il faut considérer le paramètre comme un tableau
    String resultat = "";
    for (String element : elements) {
        resultat += element;
    }
    return resultat;
}

// elle peut être appelée ainsi
concatenation("abc", "de", "f"); // renvoie "abcdef"
```

Polymorphisme d'héritage

En redéfinissant une méthode dans une sous-classe, on peut spécialiser le comportement d'une méthode.

```
public class B extends A {

  private int b;

  public B() {
    super();
    System.out.println("Création de B");
  }

  public B(int a, int b){
    super(a);
    this.b = b;
    System.out.println("Création de B");
  }

  public int getter() {
    return this.b;
  }

  public void setter(int a, int b) {
    super.setter(a);
    this.b=b;
  }

  public static void main(String[] args) {
    B ob1 = new B(10,20);
    ((A) ob1).getter(); // par trans-typage, appel de la méthode getter de la classe A
  }

}
```

Classes abstraites

- Une classe abstraite se trouve à mi-chemin entre les interfaces et les classes.
- Les classes abstraites, comme les interfaces, ne sont pas instanciables.
- Les classes abstraites sont déclarées par le modificateur **abstract**.
- Il y a plusieurs intérêts à définir des classes abstraites :
  - pour interdire l'instanciation d'une classe ;
  - pour faire de la factorisation de code en ne donnant qu'une implémentation partielle.

Exemple

```
public interface Chien {

   void vieillir();

   void aboyer();
}
```

On sait que la méthode vieillir sera implémentée de la même manière quelle que soit l'implémentation de Chien. Plutôt que d'implémenter cette interface à chaque fois, on va factoriser le code dans une classe abstraite et étendre cette classe quand le besoin s'en fait sentir. On crée donc une classe AbstractChien qui n'implémente que la méthode vieillir de notre interface, les autres étant laissées **abstract**.

```
package cours;

public abstract class AbstractChien implements Chien {
    //Champs
    //On met les champs en protected pour que les classes filles
    //puissent les manipuler directement
    protected int age;
    protected String couleur;

    //Constructeur
    //Pourra être utilisé par les classes filles pour initialiser les champs.
    public AbstractChien(int age, String couleur) {
        this.age = age;
        this.couleur = couleur;
    }

    // Méthode

    //On donne l'implémentation qui est commune à tous les chiens
    public void vieillir() {
        age ++;
    }

    //Cette méthode n'est définie que par les classes filles
    //Elle est donc laissée abstract.
    public abstract void aboyer();

}
```

Interfaces

Présentation

- Liste de méthodes dont on donne seulement la signature
- Représente un "contrat", ce qu'on attend d'un objet
- Peut être implémentée par une ou plusieurs classes qui doivent donner une implémentation pour chacune des méthodes annoncées (et éventuellement d'autres).
- Une classe peut implémenter plusieurs interfaces (permettant un héritage multiple, en les séparant par des virgules après le mot *implements*).
- Toutes les méthodes d'une interface sont implicitement abstraites.
- Une interface n'a pas de constructeurs
- Une interface ne peut avoir de champs sauf si ceux-ci sont statiques.
- Une interface peut être étendue par une ou plusieurs autre(s) interface(s).

En fait, une interface est une classe abstraite dont toutes les méthodes sont abstraites et dont tous les attributs sont constants (des constantes, voir le mot-clé *final*).

Exemple

```
//Définition de l'interface

package cours;

public interface Vehicule {

    void rouler();

    void freiner();

}
```

On a défini ici ce qu'on attend d'un objet de type véhicule.

On peut maintenant donner une ou plusieurs implémentations de cette interface grâce à l'utilisation du mot clef **implements** :

```
package cours;

public class Velo implements Vehicule {
    //Champs

    private String marque;
    private int rayonRoue;

    //Constructeurs

    public Velo(String marque, int rayonRoue)
    {
        this.marque = marque;
        this.rayonRoue = rayonRoue;
    }

    //Methodes

    public void rouler() {
        //Coder ici la manière dont le vélo roule
    }

    public void freiner() {
```

```
        //Coder ici la manière dont le vélo freine
    }

        //Autres méthodes propres à Velo
}
```

```
package cours;

public class Auto implements Vehicule {
    //Champs

    private String marque;
    private int poids;

    //Constructeurs

    public Auto(String marque, int poids)
    {
        this.marque = marque;
        this.poids = poids;
    }

    //Methodes

    public void rouler() {
        //Coder ici la manière dont l'auto roule
    }

    public void freiner() {
        //Coder ici la manière dont l'auto freine
    }

        //Autres méthodes propres à Auto.
}
```

Dans cet exemple, nous avons donné deux implémentation de Vehicule.
Conséquences :

- Ces 2 objets peuvent être vus comme des véhicules, c'est ce qu'on appelle **le polymorphisme**.
- Partout où on attend un objet de type Vehicule, on peut mettre un de ces deux objets.
- Par ce biais, on introduit une couche d'abstraction dans notre programmation ce qui la rend beaucoup plus flexible.

Discussion

Si, par exemple, nous avions une classe Personne possédant une méthode conduire(Vehicule v), on peut alors écrire :

```
Personne p = new Personne();
p.conduire(new Velo()); //comme la méthode attend un Vehicule en argument, on peut passer tout
p.conduire(new Auto()); //idem
```

On peut "instancier" un Vehicule par le biais de ses implémentations :

```
Vehicule v = new Auto();
Vehicule t = new Velo();
```

Dans ce cas v et t sont vus comme des Vehicule et, par conséquent, on ne peut appeler sur ces objets que les méthodes définies dans l'interface Vehicule.

Classes internes

Une classe interne est déclarée à l'intérieur d'une autre classe. Elle peut donc accéder aux membres de la classe externe.

Classe interne statique

Une classe interne statique ne peut accéder qu'aux membres statiques de sa classe contenante, représentée par *ClassExterne* dans l'exemple suivant :

```
public class ClasseExterne
{
    private int compteur = 0;
    private static String nom = "Exemple";

    static class ClasseInterne
    {
        private int index = 0;
        public ClasseInterne()
        {
            System.out.println("Création d'un objet dans "+nom);
            // compteur ne peut être accedé
        }
    }
}
```

La compilation du fichier `ClasseExterne.java` produit deux fichiers compilés :

- `ClasseExterne.class` contient la classe `ClasseExterne` uniquement
- `ClasseExterne$ClasseInterne.class` contient la classe `ClasseInterne`

Classe interne non statique

Une classe interne non statique peut accéder aux membres statiques de la classe ainsi qu'aux membres de l'objet qui l'a créée. En fait, le compilateur crée un membre supplémentaire dans la classe interne référençant l'objet qui l'a créé.

Une telle classe interne peut-être déclarée de manière globale dans l'objet; elle sera accessible par l'ensemble des méthodes de l'objet. Elle peut aussi être déclarée de manière locale à une méthode de l'objet. Elle sera alors accessible depuis cette seule méthode.

Exemple (Classe non statique globale) :

```
public class ClasseExterne
{
    private int compteur = 0;

    class ClasseInterne
    {
        private int index = 0;
        public ClasseInterne()
        {
            compteur++;
        }
    }
}
```

Depuis la classe interne, dans le cas où plusieurs variables ou méthodes portent le même nom dans la classe interne et la classe externe, le pointeur `this` seul désigne l'instance de la classe interne, tandis que le pointeur `this` précédé du nom de la classe externe désigne l'instance de la classe externe.

```
public class ClasseExterne
{
    private int compteur = 10;

    class ClasseInterne
    {
        private int compteur = 0;
        public void count()
        {
            this.compteur++; // -> 1
            ClasseExterne.this.compteur--; // -> 9
        }
    }
}
```

Classe anonyme

Une classe peut être déclarée au moment de l'instanciation de sa classe parente. On parle alors de classe anonyme. Exemple :

```
public class ClasseExterne {
    Bissextile b = new Bissextile() {
        public boolean evaluer(annee) {
            if ((annee%4==0 && annee%100!=0) || annee%400==0) {
                return true;
            } else {
                return false;
            }
        }
    };
    public static void main(String args[]) {
        long an = Long.parseLong(args[0]);
        if (b.evaluer(an)) {
            System.out.println("L'année entrée est bissextile");
        } else {
            System.out.println("L'année entrée n'est pas bissextile");
        }
    }
}
```

La classe Bissextile est ici la classe mère d'une classe anonyme. Cette classe sera automatiquement nommée par le compilateur `ClasseExterne$1`. Il convient d'observer le point-virgule qui suit l'accolade fermante de sa déclaration.

Cette méthode de création de sous-classe peut également s'appliquer aux interfaces. En utilisant la syntaxe suivante, la classe anonyme créée implémente l'interface spécifiée :

```
new Interface()
{
    ...implémentation des méthodes de l'interface...
}
```

Remarques :

- Une classe anonyme est implicitement considérée comme *final* (et ne peut donc pas être *abstract*, abstraite).
- Une classe anonyme ne peut pas contenir de constructeur. Le constructeur appelé est celui qui correspond à l'instruction d'instanciation. Toutefois, elle peut avoir des blocs d'initialisation.
- Une classe anonyme ne possède pas d'identificateur et ne peut donc être instanciée qu'une seule fois (d'où son nom).

Transtypage

Le **transtypage** (ou *cast*) est la conversion d'une expression d'un certain type en une expression d'un autre type.

Transtypage implicite

On peut affecter à un champ ou une variable d'un type donné une expression de type moins élevé dans la hiérarchie des types. De même, une méthode ou un constructeur attendant un argument d'un type donné peut recevoir en argument effectif une expression de type moins élevé que celui indiqué dans sa déclaration. L'expression fournie sera dans ce cas automatiquement convertie en le type attendu, sans que l'utilisateur ait besoin d'expliciter cette conversion. Toute tentative de conversion implicite d'un type vers un type qui n'est pas plus haut dans la hiérarchie des types déclenchera une erreur au moins à l'exécution, ou dès la compilation si elle est détectable statiquement.

Cas des types primitifs

Dans le cas des types primitifs, la hiérarchie est la suivante : byte est plus bas que short, short plus bas que int, char plus bas que int, int plus bas que long, long plus bas que float, float plus bas que double. Le type boolean est incomparable avec les autres types de base. On peut par exemple assigner à une variable de type float une expression de type int égale à 3 : l'expression 3 sera, avant affectation, convertie en float (3.0). Cette forme de conversion est réversible : on peut, après passage de int à float, reconvertir l'expression de type float résultante en int par une conversion explicite (voir ci-dessous) et retrouver la même valeur.

```
int n;
float f;
n = 3;
f = n;  // 3 est converti en 3.0
```

Le type déterminé statiquement pour les arguments d'opérateurs arithmétiques est le premier type à partir de int dans lequel peuvent être convertis les types de tous les arguments. Les expressions constantes sont d'autre part typées statiquement en le premier type à partir de int permettant leur représentation.

```
short s;
s = 15;      // <-- erreur générée, 15 est typé statiquement de type int
             // s = (short) 15; fonctionne (conversion explicite)
s = s + s ; // <-- erreur générée, chaque sous-expression est convertie en int
             // et l'expression s + s est typée de type int.
             // s = (short) (s + s) fonctionne
int n;
long l = 10L;
n = s;         // correct : conversion implicite
n = n + l;     // <-- erreur générée, la sous-expression gauche est convertie en long
```

Typage par suffixe

Par défaut les entiers sont typés en `Integer` mais un suffixe peut les spécifier `Long` :

```
class Suffixes {
    public static void main(String[] args) {
        Object n = 1;
        System.out.println(n.getClass());  // Integer
        n = 1L;
        System.out.println(n.getClass());  // Long
        n = 1.1;
        System.out.println(n.getClass());  // Double
        n = 1.1F;
        System.out.println(n.getClass());  // Float
    }
}
```

Cas des types références

La *classe* d'un objet ne peut évidemment être convertie : durant toute sa durée de vie, il s'agit toujours de la classe dans laquelle est défini le constructeur employé lors de sa création. Le *type d'une référence* peut en revanche être converti selon les règles suivantes :

- Si A est ancêtre de la classe B, alors toute expression de type "référence vers B" peut être implicitement convertie en le type "référence vers A".
- Si I est une interface implémentée par la classe B, toute expression de type "référence vers B" peut être implicitement convertie en le type "référence vers I".
- Si J est une interface étendant l'interface I, toute expression de type "référence vers J" peut être implicitement convertie en le type "référence vers I".

Dans l'exemple ci-dessous, on fait pointer trois références a, i, j, b, c vers un même objet de classe C :

```
class A { ... }
interface I { ... }
interface J { ... }
class B extends A implements I { ... } // implémente I, descendante de A
class C extends B implements J { ... } // implémente I et J, descendante de A et B
...

A a;  // de type "référence vers A"
I i;  // de type "référence vers I"
J j;  // de type "référence vers J"
B b;  // de type "référence vers B"
C c;  // de type "référence vers C"

c = new C();  // l'opérateur new renvoie une référence de type "référence vers C"
               // vers l'objet créé, de classe C

// la suite d'affectations suivante est valide
i = c;  // C implémente I
j = c;  // C implémente J
b = c;  // B est ancêtre de C
i = b;  // B implémente I
a = b;  // A est ancêtre de B
a = c;  // A est ancêtre de C

// chaque affectation ci-dessous déclenchera statiquement une erreur
j = b // <-- B n'implémente pas J
i = j // <-- J n'est pas une extension de I
b = a // <-- B n'est pas ancêtre de A
```

La même règle permet de convertir toute référence en une référence de type Object :

```
class A extends ... implements ... { ... } // extends Object implicite
...
Object o = new A();
```

Visibilité des champs et méthodes après transtypage, liaison dynamique

Soit r une référence de type "référence vers X", pointant vers un certain objet de classe C. D'après les règles ci-dessus, X est donc soit une classe (concrète ou abstraite) ancêtre de C, soit une interface implémentée par C :

- Les seuls champs accessibles via r sont ceux visibles dans X (déclarés dans X ou hérités, et visibles dans le contexte de r). La valeur liée à r.*champ* est celle liée à this.*champ* dans X.

```
class A {
  int x = 0:
  int y = 1;
```

```
}
class B extends A {
  int x = 2;  // redéfinition du champ x
  int z = 3;  // nouveau champ
              // le champ y est hérité
}
...

B b = new B();
A a = b;

// les expressions suivantes sont valides
... a.x...   // la valeur est celle du champ a de A, soit 0
... b.x...   // la valeur est celle du champ a de B, soit 2
... a.y...   // valeur 1
... b.y...   // y est hérité, valeur 1
... b.z...   // valeur 3

//  l'expression suivante est invalide
... a.z...   // l'objet possède bien un champ z, mais il n'est
             // pas visible dans A
```

- Seules les méthodes dont le nom est visible (par déclaration ou par héritage) dans X sont invocables sur r. L'implémentation exécutée lors d'une invocation de la forme r.*méthode(..)* sera *l'implémentation de la méthode de même nom et de même signature dans la classe de l'objet*, et non l'implémentation vue dans X. Le nom de la méthode est dit *lié dynamiquement* à l'implémentation de cette méthode dans la classe de l'objet.

```
class A {
  void m() {
     System.out.println ("implémentation de m dans A");
  }
}
class B extends A {
  // la méthode m est héritée
  // nouvelle méthode :
  void n() {
     System.out.println ("implémentation de n dans B");
  }
}
class C extends B {
  // la méthode n est héritée
  // redéfinition de m
  void m() {
     System.out.println ("implémentation de m dans C");
  }
}
...
A a = new A();
a.m();  // affiche "implémentation de m dans A"
...
B b = new B();
b.m();  // affiche "implémentation de m dans A" (héritage)
b.n();  // affiche "implémentation de n dans B"
a = b;
a.m();  // affiche "implémentation de m dans A"
a.n();  // <--- erreur : la méthode n n'est pas visible dans A
...
C c = new C();
c.m();  // affiche "implémentation de m dans C"
c.n();  // affiche "implémentation de n dans B" (héritage)
b = c;
b.m();  // affiche "implémentation de m dans C" (liaison dynamique)
b.n();  // affiche "implémentation de n dans B"
a = c;
a.m();  // affiche "implémentation de m dans C" (liaison dynamique)
a.n();  // <--- erreur : la méthode n n'est pas visible dans A
```

Cas des conversions vers String

Toute expression peut être convertie implicitement (ou explicitement) dans le type "référence vers String". Dans le cas où cette expression n'est pas statiquement constante, il y a alors création dynamique d'un objet de classe String représentant la valeur de cette expression, et l'expression résultant devient une référence vers cet objet.

Transtypage explicite

Le type d'une expression peut également être explicitement converti avec la syntaxe suivante :

```
(nouveau_type)expression
```

Où `expression` est l'expression à convertir. S'il s'agit d'une expression composée, il faut l'encadrer par des parenthèses. La conversion explicite d'une expression doit être utilisée à chaque fois que l'on souhaite convertir une expression dans un type qui n'est pas plus haut dans la hiérarchie des types. Dans le cas des types numériques, cette conversion n'est sans pertes que si le type cible permet de représenter la même valeur. Dans le cas contraire, la valeur choisie dépend du type initial et du type cible. Dans le passage de float à int, la valeur choisie est par exemple la valeur entière de la valeur initiale :

```
int n;
float f;

n = 3;
f = n;         // f vaut 3.0
f = f + 1;     // conversion de 1 en 1.0 et somme : f vaut 4.0
n = (int) f;  // n vaut 4
f = f + 1.5;  // f passe à 5.5
n = (int) f;  // 5.5 est arrondi en 5 : n vaut 5.
```

Pour les types de références, la conversion est libre : une référence de type quelconque peut être explicitement convertie en toute référence dont le type permet de manipuler l'objet référencé, selon les règles ci-dessus. La non-validité de cette conversion n'est en général pas détectable avant l'exécution :

```
interface I { ... }
class A { ... }
class B extends A implements I { ... }
class C { ... }
 ...
Object o = new B(); // l'objet créé est de classe B
I i = (I) o;  // valide : B implémente I
A a = (A) o;  // valide : A est ancêtre de B
B b = (B) a;  // valide
C c = (C) o;  // invalide : C n'est pas ancêtre de B
```

Ces conversions "descendantes" sont bien sûr propices aux erreurs. L'opérateur `instanceof` permet de vérifier la validité d'une conversion avant de l'effectuer :

```
if (r instanceof C) {
  c = (C) r;
  // action sur les instances de C
  ...
}
else {
  // action sur les instances d'une autre classe
  ...
}
```

L'opérateur `instanceof` et les conversions supportent également les tableaux (à partir d'un objet de type `Object`) :

```
Object r = getObject();
```

```
if (r instanceof int[]) {
  int[] valeurs = (int[]) r;
  ...
}
else {
  ...
}
```

Autoboxing

Java 5 introduit un mécanisme permettant la simplification du transtypage, appelé *autoboxing*. Ce mécanisme permet d'utiliser indifféremment les types primitifs et les classes wrappers. Exemple :

Avant Java 5, il fallait écrire :

```
List integers = methodeRenvoyantDesIntegers();
for(i=0;i<integers.size();i++) {
    Integer integer = (Integer)integers.get(i);
    int actuel = Integer.parseInt(integer);
    methodNecessitantUnInt(actuel);
}
```

Alors qu'avec Java 5, il n'est plus nécessaire de passer par `parseInt()` :

```
List integers = methodeRenvoyantDesIntegers();
for(i=0;i<integers.size();i++) {
    int actuel = (Integer)integers.get(i);
    methodNecessitantUnInt(actuel);
}
```

On voit que les `int` et les `Integer` sont utilisés indifféremment.

Toutefois, il n'est pas possible de déclarer un type générique avec un type primitif. Il faut utiliser la classe englobante correspondante.

Exemple :

```
ArrayList<Integer> counters = new ArrayList<Integer>();
counters.add(500);
int n = counters.get(0);
```

Les limites de l'autoboxing est qu'il ne concerne que chaque type primitif et sa classe englobante respective. Par exemple le code suivant renvoie l'erreur `inconvertible types` en voulant passer du *String* au *Float* :

```
    public static void main(String[] args) {
        for(int i = 0; i < args.length; i++) {
            System.out.println((Float)args[i]);
        }
    }
```

Il faut donc écrire[1] :

```
public static void main(String[] args) {
    for(i=0;i<integers.size();i++) {
        System.out.println((Float.valueOf(args[i])).floatValue());
    }
```

Références

1. http://docs.oracle.com/javase/tutorial/java/data/converting.html

Types génériques

Définition

Les génériques (de l'anglais *generics*) sont des classes qui sont typés au moment de la compilation. Autrement dit, ce sont des classes qui utilisent des typages en paramètres. Ainsi une liste chainée, qui peut contenir des entiers, des chaines ou autres pourra être typée en liste de chaine ou liste d'entier et ceci permettra au programmeur de ne pas écrire systématiquement des transtypages, méthode qui pourrait s'avérer dangereuse, ce sera le compilateur qui vérifiera la cohérence des données.

Java 5 a introduit un principe de type générique, rappelant les *templates* (modèles) du C++, mais le code produit est unique pour tous les objets obtenus à partir de la même classe générique.

Avant java 5 :

```
public class MaListe
{
    private LinkedList liste;
    public setMembre(String s)
    {
       liste.add(s);
    }
    public int getMembre(int i)
    {
       return (Int)liste.get(i);
    }
}
```

Le transtypage est obligatoire, LinkedList manipule des objets Object, ici le compilateur ne peut détecter de problème, problème qui ne surviendra qu'à l'exécution (RunTimeError).

Dans la version avec génériques, on n'a plus besoin d'utiliser le transtypage donc le compilateur déclenchera deux erreurs durant la compilation, une sur la méthode, l'autre sur l'ajout d'un entier.

```
public class Famille < MaClasse >
{
    private LinkedList < MaClasse > liste;

    public setMembre(MaClasse m)
    {
        liste.add(m);
    }

    public MaClasse getMembre(int i)
    {
        return liste.get(i);
    }

    public Int getInt(int i)      //première erreur
    {
        return liste.get(i);
    }
}
```

Utilisation :

```
   Famille<String> famille = new Famille<String>();
   famille.setMembre("essai");
   famille.setMembre(210);          //seconde erreur
```

Il est important de comprendre que dans la déclaration de la classe le paramètre placé entre les caractères < et > représente bien une classe qui ne sera déterminée que lors de la déclaration de la création de l'objet. Aussi une erreur de typage sera produite à la compilation si les types utilisés par les méthodes ne sont le ou les types attendus. Dans cet exemple, l'erreur sera signalée sur le second ajout.

Dans la déclaration de la classe, la liste membre est déclarée ne pouvant contenir que des objets de classe MaClasse. L'identifiant MaClasse n'est pas une classe existante dans le packages et il est préférable qu'il ne le soit pas pour qu'aucune confusion ne soit faite, c'est à la déclaration de l'objet Famille que l'identifiant MaClasse sera résolu.

Il est évidemment possible d'utiliser un objet d'une classe héritant de celle utilisée pour paramétrer le type générique. Ceci permet de plus d'assurer la compatibilité ascendante avec les versions antérieures de Java : si aucune classe de paramétrage n'est indiquée, la classe par défaut est `java.lang.Object`.

Plusieurs paramètres

De la même façon que pour les classes basées sur les List, les déclarations de vos classes peuvent utiliser ces génériques. Cela permet de rendre le code plus souple et surtout réutilisable dans des contextes très différents. Plusieurs paramètres, séparés par des virgules, peuvent êtres utilisés entre les caractères < et >.

```
public class ListeDeTruc<Truc, Bidule>
{
    private LinkedList < Truc > liste;
    private ArrayList <Bidule> tableau;

    public void accumule(Truc m)
    {
        liste.add(m);
    }

    public Bidule recherche(int i)
    {
        return tableau.get(i);
    }
}
```

Déclaration possible :

```
ListeDeTruc<String,Integer> liste1 = new ListeDeTruc<String,Integer>();
ListeDeTruc<Thread,Date> liste2 = new ListeDeTruc<Thread,Date>();
```

Génériques et héritages

Lorsqu'un type de base doit répondre à des spécifications précises, il est possible d'écrire des choses du genre :

```
    public class ListeDeTruc<Truc extends Bidule, MaList<String>> implements Moninterface<Chose
```

En revanche, créer une classe qui hérite de ces objets est plus délicat. Ici Chose et Bidule sont des classes existantes, Truc ne sera résolu qu'au moment de la déclaration de l'objet ListeDeTruc.

L'utilisation du mot clef super est possible dans une classe héritant d'une classe générique.

Tableau de génériques

La déclaration d'un tableau d'objets dont le type est générique peut se faire sans déclencher ni erreur, ni avertissements et sans utiliser l'annotation `@SuppressWarnings("unchecked")`, en utilisant `<?>` :

```
ArrayList<?>[] namelists = new ArrayList<?>[5];
```

Conventions sur les noms des types

Bien qu'il soit tout à fait possible d'utiliser n'importe-quel identifiant suivant la convention de nommage des classes, il est plutôt recommandé d'utiliser un identifiant composé d'une seule lettre selon la convention[1] :

`<E>`
: « Element », utilisé abondamment pour le type des éléments d'une collection ;

`<K>` et `<V>`
: « Key », pour respectivement le type des clés et celui des valeurs d'une Map ou similaires ;

`<N>`
: « Number » ;

`<T>`
: « Type » ;

`<S>`, `<U>`, `<V>` etc.
: second, troisième, i^{ème} type.

Limitations

Malgré la similitude de syntaxe, les types génériques en Java sont différents des patrons (*templates* en anglais) du C++. Il faut plutôt les voir comme un moyen d'éviter de faire une conversion entre java.lang.Object et le type spécifié de manière implicite.

Parmi les limitations :

- il n'est pas possible d'implémenter plusieurs fois la même interface avec des paramètres différents,
- il n'est pas possible de créer deux versions surchargées d'une méthode (deux méthodes portant le même nom) l'une utilisant la classe Object et l'autre utilisant un type générique.

Références

1. « Type Parameter Naming Conventions » (http://download.oracle.com/javase/tutorial/java/generics/gentypes.html) [archive]

Instanciation et cycle de vie

L'instruction *new*

L'instruction `new` permet d'instancier une classe en utilisant l'un des constructeurs de la classe.

Par exemple pour créer un objet de type *MaClasse*, on écrit :

```
MaClasse cl = new MaClasse("hello");
```

Les constructeurs

Un constructeur est une méthode particulière de la classe appelée lors de la création d'une instance de la classe. Son rôle est d'initialiser les membres de l'objet juste créé. Un constructeur a le même nom que sa classe et n'a pas de valeur de retour.

Dans l'exemple suivant la classe *MaClasse* dispose de deux constructeurs, l'un ne prenant aucun paramètre et l'autre prenant un paramètre de type *String* :

```
public class MaClasse
{
  // Attributs
  private String name;

  // Constructeurs
  public MaClasse()
  {
    name = "defaut";
  }

  public MaClasse(String s)
  {
    name = s;
  }
}
```

Toute classe possède un constructeur. Cependant, il n'est pas obligatoire de déclarer un constructeur pour une classe. En effet, si aucun constructeur n'est déclaré dans une classe, un constructeur sans paramètre est ajouté de manière implicite. Celui-ci ne fait rien.

Nettoyage

Ramasse-miettes (Garbage Collector)

Le ramasse-miettes garde un compteur du nombre de référence pour chaque objet. Dès qu'un objet n'est plus référencé, celui-ci est marqué. Lorsque le ramasse-miettes s'exécute (en général quand l'application ne fait rien), les objets marqués sont libérés.

Son exécution se produit toujours à un moment qui ne peut être déterminé à l'avance. Il s'exécute lors des évènements suivants :

- périodiquement, si le processeur n'est pas occupé,
- quand la quantité de mémoire restante est insuffisante pour allouer un nouveau bloc de mémoire.

Il est donc recommandé de libérer toute référence (affecter `null`) à des objets encombrants (tableaux de grande taille,

collection d'objets, ...) dès que possible, ou au plus tard juste avant l'allocation d'une grande quantité de mémoire.

Exemple : Pour le code suivant, il faut 49 152 octets disponibles, le ramasse-miettes ne pouvant rien libérer durant l'allocation du deuxième tableau.

```
byte[] buffer = new byte[16384];
// -> 1. allocation de 16384 octets
//    2. affecter la référence à la variable buffer
// ...
buffer = new byte[32768];
// -> 1. allocation de 32768 octets
//    2. affecter la référence à la variable buffer
```

Une fois le code corrigé, il ne faut plus que 32 768 octets disponibles, le ramasse-miettes pouvant libérer le premier tableau avant d'allouer le deuxième.

```
byte[] buffer = new byte[16384];
// -> 1. allocation de 16384 octets
//    2. affecter la référence à la variable buffer
// ...
buffer = null; // Le ramasse-miettes peut libérer les 16384 octets du tableau si besoin.
buffer = new byte[32768];
// -> 1. allocation de 32768 octets
//    2. affecter la référence à la variable buffer
```

Finalisation

Lors de la libération des objets par le ramasse-miettes, celui-ci appelle la méthode `finalize()` afin que l'objet libère les ressources qu'il utilise.

Cette méthode peut être redéfinie afin de libérer des ressources systèmes non Java. Dans ce cas, la méthode doit se terminer en appelant la méthode de la classe parente :

```
super.finalize();
```

Représentation des objets dans la mémoire

La pile

Cette section est vide, pas assez détaillée ou incomplète.

Le tas

Objets dans la pile

Constantes et membres static

Attributs

this

Le mot-clé *this* désigne, dans une classe, l'instance courante de la classe elle-même. Il est utilisé à différentes fins décrites dans les sections suivantes.

Rendre Univoque

Il peut être utilisé pour rendre le code explicite et non ambigu.

Par exemple, si dans une méthode, on a un paramètre ayant le même nom qu'un attribut de la classe dont la méthode fait partie, on peut désigner explicitement l'attribut grâce à *this* :

```
public class Calculateur
{
    protected int valeur;

    public void calcule(int valeur)
    {
        this.valeur = this.valeur + valeur;
    }
}
```

Dans cet exemple, la méthode *calcule* additionne le paramètre *valeur* à l'attribut *valeur* et stocke le résultat dans l'attribut *valeur*. L'attribut a été désigné explicitement par le mot-clé *this*, désignant l'instance de la classe, préfixé au nom de l'attribut.

S'auto-désigner comme référence

Le mot-clé `this` peut être utilisé pour passer une référence à l'instance elle-même comme paramètre d'une méthode.

Par exemple : s'enregistrer comme écouteur d'évènement :

```
source.addListener(this);
```

Désigner l'instance de la classe qui encadre

Dans le cas de classes imbriquées, c'est-à-dire qu'une classe interne utilise l'instance de la classe externe, le mot-clé `this` préfixé du nom de la classe externe permet de désigner l'instance de la classe externe. S'il n'est pas préfixé, il désigne l'instance de la classe interne.

Exemple :

```
public class Livre
{
    String titre = "Le livre";

    class Chapitre
    {
        String titre = "Chapitre 1";

        public String toString()
        {
            return
                Livre.this.titre +     // "Le livre"
                "/" +
                this.titre ;           // "Chapitre 1"
        // ou   Chapitre.this.titre ; // "Chapitre 1"
```

```
        }
        // -> "Le livre/Chapitre 1"
    }
}
```

Objets comparables et clés

Certaines collections d'objets nécessitent l'utilisation de classes dont les instances sont comparables pour pouvoir trier la collection. De plus, pour utiliser une instance d'une classe comme clé dans les tables associatives, la classe doit posséder des propriétés supplémentaires.

Objets comparables

Pour utiliser un objet dans une collection triée, on peut :

- soit utiliser des éléments instances d'une classe comparable,
- soit spécifier un comparateur d'éléments au constructeur de la collection.

interface `Comparable`

Une classe implémente l'interface `java.lang.Comparable` en ajoutant la méthode suivante :

```
int compareTo(Object o)
```

ou (Java 5+) interface `Comparable<T>` :

```
int compareTo(T o)
```

Dans cette méthode l'objet `this` est comparé à l'objet `o`. Cette méthode doit retourner le signe de la soustraction virtuelle `this - o`. C'est à dire :

- -1 si `this < o`
- 0 si `this == o`
- +1 si `this > o`

Cette méthode doit avoir les propriétés suivantes, pour toutes instances de la classe nommées `o1`, `o2` et `o3` :

- `o1.compareTo(o1) == 0`,
- `o1.compareTo(o2) == - o2.compareTo(o1)`,
- `o1.compareTo(o2) > 0` et `o2.compareTo(o3) > 0` implique `o1.compareTo(o3) > 0`.

De plus, pour utiliser une instance de la classe dans une collection, le comportement de cette méthode doit être consistant avec celle de la méthode `equals` :

- Pour toute instance de la classe nommés `o1` et `o2`, les deux expressions booléennes `o1.compareTo(o2)==0` et `o1.equals(o2)` retournent la même valeur,
- La comparaison avec `null` doit lancer une exception `NullPointerException`.

Comparateur (`Comparator`)

L'interface `java.util.Comparator` permet de rendre comparable des instances dont la classe n'implémente pas l'interface `java.lang.Comparable` vu précédemment. Une classe implémentant cette interface doit déclarer deux méthodes :

```
int compare(Object o1, Object o2)
int equals(Object o)
```

ou (Java 5+) interface `Comparator<T>` :

```
int compare(T o1, T o2)
int equals(Object o)
```

La méthode `equals` compare les comparateurs (eux-mêmes) `this` et `o`.

La méthode `compare` compare les deux objets spécifiés et retourne le signe de la soustraction virtuelle `o1 - o2`, comme expliqué dans la section précédente. C'est-à-dire :

- -1 si `o1 < o2`
- 0 si `o1 == o2`
- +1 si `o1 > o2`

Clé de table associative

Pour utiliser une instance de classe comme clé d'une table associative (*Map* en anglais), la classe doit posséder les propriétés suivantes :

- La classe doit posséder une méthode `equals` cohérente,
- La classe doit posséder une méthode `hashCode` cohérente.

Dans le cas contraire (par défaut), il est toujours possible d'utiliser des instances de la classe comme clé, mais il faudra utiliser cette instance seulement. C'est à dire que pour retrouver une valeur dans une table associative, il ne sera pas possible d'utiliser une autre instance dont les attributs sont égaux à ceux de l'instance utilisée pour réaliser l'association.

Exemple : un tableau d'entiers utilisé comme clé.

```
int[] key1 = { 1, 2 };
int[] key2 = { 1, 2 }; // même contenu que key1

HashMap hmap = new HashMap();

// association key1 -> chaîne de caractère
hmap.put(key1, "Valeur pour la suite (1,2)");

// tentative pour retrouver la valeur
String s1 = (String)hmap.get(key1); // retourne "Valeur pour la suite (1,2)"

// tentative pour retrouver la valeur
String s2 = (String)hmap.get(key2); // retourne null
```

La tentative échoue car un tableau n'a pas toutes les propriétés nécessaires. La méthode `equals` est correcte, mais la méthode `hashCode` retourne deux valeurs différentes.

Méthode equals

Une classe doit implémenter la méthode `equals` de manière cohérente, c’est-à-dire, pour toutes instances de la classe nommées `o1`, `o2` et `o3` :

- `o1.equals(o1) == true`,
- `o1.equals(o2) == o2.equals(o1)`,
- `o1.equals(o2)` et `o2.equals(o3)` implique `o1.equals(o3)`.

Méthode hashCode

Pour utiliser une classe comme clé d'une table associative, il faut que la classe implémente la méthode hashCode de manière cohérente. Pour comprendre cela, il faut aborder le fonctionnement interne des tables associatives indexée par des objets :

1. Les méthodes de la table associative appellent la méthode hashCode de la clé pour obtenir un entier,
2. Cet entier est transformé en index dans un tableau par reste de la division par la taille du tableau,
3. Ce tableau contient une liste de clés comparées avec celle fournie à la méthode appelée en utilisant la méthode `equals` vue auparavant.

Il est donc essentiel que la méthode `hashCode` ait les propriétés suivantes :

- Cette méthode doit toujours retourner la même valeur si l'objet n'a pas été modifié,
- Pour deux objets égaux (méthode `equals` retourne `true`), la méthode `hashCode` doit retourner deux valeurs égales. C'est à dire, pour toute instance de la classe nommés `o1` et `o2`, `o1.equals(o2)` implique `o1.hashCode() == o2.hashCode()`,
- Il est recommandé que pour deux objets différents, la méthode `hashCode` retourne deux valeurs distinctes.

Objets utilisables comme clé

Comme expliqué dans la section d'introduction, il est possible d'utiliser n'importe quel type d'objet à condition d'utiliser exactement la même instance, ou bien que la classe possède des méthodes `equals` et `hashCode` cohérentes, comme la classe `java.lang.String` par exemple.

```
String key1 = "Clé de test";
String key2 = "Clé" + " de " + "test";

HashMap hmap = new HashMap();

// association key1 -> chaîne de caractère
hmap.put(key1, "Valeur pour la clé");

// tentative pour retrouver la valeur
String s1 = (String)hmap.get(key1); // retourne "Valeur pour la clé"

// tentative pour retrouver la valeur
String s2 = (String)hmap.get(key2); // retourne "Valeur pour la clé"
```

Énumérations

Java 5 introduit une nouvelle structure de données appelée *énumérations*. Cette structure permet de contenir une série de données constantes ayant un type sûr, ce qui veut dire que ni le type, ni la valeur réelle de chaque constante n'est précisé. Il est possible de comparer des valeurs d'une énumération entre elles à l'aide des opérateurs de comparaison et de l'instruction `switch`.

Exemple :

```
enum Animal { KANGOUROU, TIGRE, CHIEN, SERPENT, CHAT };

class Test
{
    public static void main(String[] args)
    {
        String aniMsg;
        Animal bebete = Animal.TIGRE;
        switch(bebete)
        {
            case KANGOUROU :
                aniMsg = "kangourou";
                break;
            case TIGRE :
                aniMsg = "tigre";
                break;
            case CHIEN :
                aniMsg = "chien";
                break;
            case SERPENT :
                aniMsg = "serpent";
                break;
            case CHAT :
                aniMsg = "chat";
                break;
        }
        System.out.println("L'animal est un "+aniMsg);
    }
}
```

Membres

Les énumérations sont en fait compilées sous forme de classes, éventuellement internes.

Une énumération peut donc avoir des constructeurs et méthodes. Ses constructeurs sont obligatoirement privés car aucune nouvelle instance ne peut être créée.

```
enum Animal
{
    // Il faut appeler l'un des constructeurs déclarés :
    KANGOUROU("kangourou", false),
    TIGRE("tigre", false),
    CHIEN("chien", true),
    SERPENT("serpent", false, "tropical"),
    CHAT("chat", true); // <- NB: le point-virgule pour mettre fin à la liste des constantes !

    // Membres :
    private final String environnement;
    private final String nom;
    private final boolean domestique;

    Animal(String nom, boolean domestique)
    { this(nom, domestique, null); }
```

```
    Animal(String nom, boolean domestique, String environnement)
    {
        this.nom = nom;
        this.domestique = domestique;
        this.environnement = environnement;
    }

    public String getNom(){ return this.nom; }
    public String getEnvironnement(){ return this.environnement; }
    public boolean isDomestique(){ return this.domestique; }
};

class Test
{
    public static void main(String[] args)
    {
        Animal bebete = Animal.TIGRE;
        System.out.print("L'animal est un "+bebete.getNom());
        System.out.print(bebete.isDomestique()?" (domestique)":" (sauvage)");
        String env = bebete.getEnvironnement();
        if (env!=null)
            System.out.print(" vivant dans un milieu "+env);
        System.out.println();
    }
}
```

Méthodes utiles

Les énumérations possèdent des méthodes communes.

`ordinal()`
: Obtenir l'index de la valeur selon l'ordre de déclaration (premier = 0).
: Exemple :

```
Animal.CHIEN.ordinal() /* -> 2 */
```

`name()`
: Obtenir le nom de la valeur.
: Exemple :

```
Animal.CHAT.name() /* -> "CHAT" */
```

`valueOf(String s)`
: (méthode statique) Obtenir la valeur dont le nom est spécifié en paramètre.
: Exemple :

```
Animal.valueOf("CHAT") /* -> Animal.CHAT */
```

`values()`
: (méthode statique) Obtenir un tableau contenant toutes les valeurs déclarées.
: Exemple :

```
Animal.values()[2] /* -> Animal.CHIEN */
```

Les énumérations implémentent l'interface `java.lang.Comparable` et possède donc les méthodes `equals` et `compare` pour comparer deux valeurs (ordonnées selon `ordinal()` par défaut).

Ensemble de valeurs énumérées

Les ensembles (*set* en anglais) font partie des classes de collection de l'API Java. Il existe une classe spécifique pour les ensembles de valeurs énumérées nommée `EnumSet`. Cette classe hérite de la classe abstraite `java.util.AbstractSet`.

Voir les ensembles.

Exceptions

Une **exception** est un signal qui se déclenche en cas de problème. Les exceptions permettent de gérer les cas d'erreur et de rétablir une situation stable (ce qui veut dire, dans certains cas, quitter l'application proprement). La gestion des exceptions se décompose en deux phases :

- La levée d'exceptions,
- Le traitement d'exceptions.

En Java, une exception est représentée par une classe. Toutes les exceptions dérivent de la classe *Exception* qui dérive de la classe *Throwable*.

Levée d'exception

Une exception est levée grâce à l'instruction *throw* :

```
if (k<0)
   throw new RuntimeException("k négatif");
```

Une exception peut être traitée directement par la méthode dans laquelle elle est levée, mais elle peut également être envoyée à la méthode appelante grâce à l'instruction *throws* (à ne pas confondre avec *throw*) :

```
import java.io.IOException;
public void maMethode(int entier) throws IOException
{
   //code de la methode
}
```

Dans cet exemple, si une exception de type *IOException* est levée durant l'exécution de *maMethode*, l'exception sera envoyée à la méthode appelant *maMethode*, qui devra la traiter.

Certaines exceptions sont levées implicitement par la machine virtuelle :

- `NullPointerException` quand une référence nulle est déréférencée (accès à un membre),
- `ArrayIndexOutOfBoundsException` quand l'indice d'un tableau dépasse sa capacité,
- `ArithmeticException` quand une division par zéro a lieu.

Celles-ci n'ont pas besoin d'être déclarées avec l'instruction *throws* car elles dérivent de la classe `RuntimeException`, une classe d'exceptions qui ne sont pas censées être lancées par une méthode codée et utilisée correctement.

Traitement d'exception

Le traitement des exceptions se fait à l'aide de la séquence d'instructions *try...catch...finally*.

- L'instruction *try* indique qu'une instruction (ou plus généralement un bloc d'instructions) susceptible de lever des exceptions débute.
- L'instruction *catch* indique le traitement pour un type particulier d'exceptions. Il peut y avoir plusieurs instructions *catch* pour une même instruction *try*.
- L'instruction *finally*, qui est optionnelle, sert à définir un bloc de code à exécuter dans tous les cas, exception levée ou non.

Il faut au moins une instruction *catch* ou *finally* pour chaque instruction *try*.

Exemple :

```
public String lire(String nomDeFichier) throws IOException
{
    try
    {
        // La ligne suivante est susceptible de lever une exception
        // de type FileNoFoundException
        FileReader lecteur = new FileReader(nomDeFichier);
        char[] buf = new char[100];
        // Cette ligne est susceptible de lever une exception
        // de type IOException
        lecteur.read(buf,0,100);
        return new String(buf);
    }
    catch (FileNotFoundException fnfe)
    {
        fnfe.printStackTrace(); // Indique l'exception sur le flux d'erreur standard
    }
    finally
    {
        System.err.println("Fin de méthode");
    }
}
```

Le bloc `catch (FileNotFoundException fnfe)` capture toute exception du type `FileNotFoundException` (cette classe dérive de la classe `IOException`).

Le bloc `finally` est exécuté quelque soit ce qui se passe (exception ou non).

Toute autre exception non capturée (telle `IOException`) est transmise à la méthode appelante, et doit **toujours** être déclarée pour la méthode, en utilisant le mot clé `throws`, sauf les exceptions dérivant de la classe `RuntimeException`. S'il n'y avait pas cette exception à la règle, il faudrait déclarer `throws ArrayIndexOutOfBoundsException` chaque fois qu'une méthode utilise un tableau, ou `throws ArithmeticException` chaque fois qu'une expression est utilisée, par exemple.

Classes et sous-classes d'exception

L'héritage entre les classes d'exceptions peut conduire à des erreurs de programmation. En effet, une instance d'une sous-classe est également considérée comme une instance de la classe de base.

Ordre des blocs catch

L'ordre des blocs `catch` est important : il faut placer les sous-classes avant leur classe de base. Dans le cas contraire le compilateur génère l'erreur `exception` *`classe_exception`* `has already been caught`.

Exemple d'ordre incorrect :

```
try{
    FileReader lecteur = new FileReader(nomDeFichier);
}
catch(IOException ioex) // capture IOException et ses sous-classes
{
    System.err.println("IOException catched :");
    ioex.printStackTrace();
}
catch(FileNotFoundException fnfex) // <-- erreur ici
// FileNotFoundException déjà capturé par catch(IOException ioex)
{
    System.err.println("FileNotFoundException catched :");
    fnfex.printStackTrace();
}
```

L'ordre correct est le suivant :

```
try{
    FileReader lecteur = new FileReader(nomDeFichier);
}
catch(FileNotFoundException fnfex)
{
    System.err.println("FileNotFoundException catched :");
    fnfex.printStackTrace();
}
catch(IOException ioex) // capture IOException et ses autres sous-classes
{
    System.err.println("IOException catched :");
    ioex.printStackTrace();
}
```

Sous-classes et clause throws

Une autre source de problèmes avec les sous-classes d'exception est la clause `throws`. Ce problème n'est pas détecté à la compilation.

Exemple :

```
public String lire(String nomDeFichier) throws FileNotFoundException
{
    try
    {
        FileReader lecteur = new FileReader(nomDeFichier);
        char[] buf = new char[100];
        lecteur.read(buf,0,100);
        return new String(buf);
    }
    catch (IOException ioe) // capture IOException et ses sous-classes
    {
        ioe.printStackTrace();
    }
}
```

Cette méthode ne lancera jamais d'exception de type `FileNotFoundException` car cette sous-classe de `IOException` est déjà capturée.

Relancer une exception

Une exception peut être partiellement traitée, puis relancée. On peut aussi relancer une exception d'un autre type, cette dernière ayant l'exception originale comme cause.

Dans le cas où l'exception est partiellement traitée avant propagation, la relancer consiste simplement à utiliser l'instruction `throw` avec l'objet exception que l'on a capturé.

Exemple:

```
public String lire(String nomDeFichier) throws IOException
{
    try
    {
        FileReader lecteur = new FileReader(nomDeFichier);
        char[] buf = new char[100];
        lecteur.read(buf,0,100);
        return new String(buf);
    }
    catch (IOException ioException) // capture IOException et ses sous-classes
    {
        // ... traitement partiel de l'exception ...
        throw ioException; //<-- relance l'exception
```

```
    }
}
```

Une exception d'un autre type peut être levée, par exemple pour ne pas propager une exception de type SQLException à la couche métier, tout en continuant à arrêter l'exécution normale du programme :

```
...
    catch (SQLException sqlException) // capture SQLException et ses sous-classes
    {
        throw new RuntimeException("Erreur (base de données)...", sqlException);
    }
...
```

La pile d'appel est remplie au moment de la création de l'objet exception. C'est à dire que les méthodes `printStackTrace()` affiche la localisation de la création de l'instance.

Pour mettre à jour la pile d'appel d'une exception pré-existante (réutilisation pour éviter une allocation mémoire, ou relancer une exception), la méthode `fillInStackTrace()` peut être utilisée :

```
...
    catch (IOException ioException) // capture IOException et ses sous-classes
    {
        // ... traitement partiel de l'exception ...
        ioException.fillInStackTrace(); // <-- pile d'appel mise à jour pour pointer ici
        throw ioException;              // <-- relance l'exception
    }
...
```

Catégorie d'objet lancé

Le chapitre traite des exceptions, mais en fait tout objet dont la classe est ou dérive de la classe `Throwable` peut être utilisé avec les mots-clés `throw`, `throws` et `catch`.

Classes dérivées de Throwable

Il existe deux principales sous-classes de la classe `Throwable` :

- `Exception` signale une erreur dans l'application,
- `Error` signale une erreur plus grave, souvent au niveau de la machine virtuelle (manque de ressource, mauvais format de classe, ...).

Créer une classe d'exception

Il est également possible d'étendre une classe d'exception pour spécialiser un type d'erreur, ajouter une information dans l'objet exception, ...

Exemple :

```
public class HttpException extends Exception
{
    private int code;
    public HttpException(int code,String message)
    {
        super(""+code+" "+message);
        this.code=code;
    }
    public int getHttpCode()
    {return code;}
```

```
}
```

Une instance de cette classe peut ensuite être lancée de la manière suivante :

```
public void download(URL url) throws HttpException
{
    ...
    throw new HttpException ( 404, "File not found" );
}
```

et capturée comme suit :

```
try
{
    download( ... );
}
catch(HttpException http_ex)
{
    System.err.println("Erreur "+http_ex.getHttpCode());
}
```

Voir aussi

- Liste des exceptions (https://docs.oracle.com/javase/6/docs/api/java/lang/Exception.html) [archive]

Extensions

Les extensions, ou *packages*, permettent de grouper ensemble des classes rattachées, à la manière des dossiers qui permettent de classer des fichiers.

Utilisation

Le fichier à inclure dans une extension doit contenir le mot-clé '`package`' suivi du nom de l'extension.

Ce nom peut être composé de plusieurs mots séparés par un point (`.`).

Exemple

pour inclure la classe Toto dans l'extension '`mesPackages.sousPackage1`', écrire au début du fichier **Toto.java** :

```
package mesPackages.sousPackage1;
// ne pas oublier le point-virgule en fin de ligne
```

La structure des répertoires doit suivre le nom de l'extension, c'est-à-dire que le fichier **Toto.java** doit se situer dans un sous-répertoire **mesPackages/sousPackage1/Toto.java**.

Lorsqu'ensuite on désire utiliser la classe Toto depuis une autre classe, il faudra au préalable écrire :

```
import mesPackages.sousPackage1.Toto;
```

ou

```
import mesPackages.sousPackage1.*;
// importation de toutes les classes
// de l'extension mesPackage.sousPackage1
```

ou utiliser directement une référence à l'extension :

```
mesPackages.sousPackage1.Toto toto
    = new mesPackages.sousPackage1.Toto();
```

Remarques

En *Java*, les programmeurs attribuent généralement un nom qui commence par une minuscule pour une extension, et un nom qui commence par une capitale pour une classe.

Les bibliothèques Java destinées à être distribuées regroupent leurs classes dans une ou plusieurs extensions dont le nom est normalement précédé par un nom de domaine (dans l'ordre inverse), par exemple :

```
package org.wikibooks.exemple;
```

Compilation

L'utilisation d'une extension nécessite une structure des répertoires correspondant au nom de l'extension.

Par exemple, le fichier `Toto.java` définit la classe `Toto` de l'extension `org.wikibooks.exemple` débute par :

```
package org.wikibooks.exemple;

class Toto ...
```

et **doit** se situer dans le répertoire `org/wikibooks/exemple`.

Supposons que le chemin du fichier soit `/home/me/javaprog/org/wikibooks/exemple/Toto.java`. La compilation se fait en spécifiant le chemin du package racine (répertoire parent de org) comme classpath, et en spécifiant ensuite le chemin relatif à ce répertoire :

```
javac -classpath /home/me/javaprog org/wikibooks/exemple/Toto.java
```

Quand un programme Java utilise cette classe, il doit être compilé et exécuté en spécifiant `/home/me/javaprog` (package racine) pour le paramètre *classpath*, et le nom de la classe doit inclure le nom du package :

```
java -classpath /home/me/javaprog org.wikibooks.exemple.Toto
```

Import statique

Pour utiliser les membres statiques publiques d'une classe, il faut nommer la classe où ils sont définis.

Exemple 1 :

```
double r = Math.cos(Math.PI * theta);
```

L'import statique permet d'importer les membres statiques d'une classe afin de ne pas nommer la classe en question.

Exemple 1 :

```
// importer le membre statique PI seulement
import static java.lang.Math.PI;
...
double r = Math.cos(PI * theta);
```

Exemple 2 :

```
// importer tous les membres statiques de la classe java.lang.Math
import static java.lang.Math.*;
...
double r = cos(PI * theta);
```

L'abus d'import statique n'est pas conseillé car le code ne contient plus de référence à la classe définissant le membre statique utilisé. Il ne faut l'utiliser que si les membres statiques d'un petit nombre de classes sont utilisés fréquemment.

Importation de packages de .jar

Pour importer un package d'un fichier `.jar`, il faut s'assurer que le fichier est dans le classpath courant (à *compile-* et *execution-time*). Ensuite, l'import se déroule comme si le .jar était décompressé.

Par exemple, pour compiler et lancer une classe d'un projet du dossier parent (contenant deux répertoires : `/source` et `/libraries`) compiler :

```
$ javac -classpath libraries/lib.jar source/MainClass.java
```

Puis le lancer :

```
$ java -classpath libraries/lib.jar source/MainClass
```

Cela nécessite que `MainClass` soit le package par défaut, ou un package appelé `source`, ce qui n'est pas très explicite.

Classes de base

Java est livré avec un ensemble de bibliothèques de base proposant des classes qui s'avèrent vite nécessaire. Tout programmeur Java doit avoir un bonne connaissance de toutes les possibilités offertes par ces composants. Il n'est pas nécessaire d'apprendre quoi que ce soit mais simplement d'avoir une idée générale de ce qui est disponible pour savoir quelles parties de la bibliothèque utiliser pour résoudre un problème donné.

Dorénavant, vous aurez toujours besoin d'avoir avoir la documentation complète (http://docs.oracle.com/javase/7/docs/api/index.html) [archive] sous le coude, via Internet ou sur votre ordinateur.

Extension java.lang

L'extension (le *package*) java.lang est l'extension de base de Java. Il n'est même pas nécessaire de faire un import de cette extension : il est implicite.

Cette extension regroupe les classes concernant les types (`Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float`, `Double`, `String`), les classes (`Object` classe de base de toutes les autres, `Class`), les threads (`Thread`, `ThreadGroup`), …

Les chaînes de caractères

Les chaînes de caractères ne sont pas des types primitifs et sont des instances de la classe `java.lang.String`. Les constantes sont entourées de guillemets.

Exemple :

```
String message = "Test des chaînes de caractères";
```

La concaténation se fait avec l'opérateur `+` :

```
String message = "Test des chaînes de caractères"
               + " et de la concaténation";
```

La concaténation d'une chaîne de caractères avec un type primitif ajoute sa représentation nominale (décimal pour les types entiers, le caractère pour `char`, `"true"` ou `"false"` pour `boolean`) :

```
int a = 100;
String message = "A vaut " + a;   // A vaut 100
```

La concaténation d'une chaîne de caractères avec un objet appelle sa méthode `toString()` retournant une chaîne de caractères :

```
Object o=new Object();
String message = "L'objet " + o;  // L'objet java.lang.Object@1fe081a3
```

Par défaut, la méthode `toString()` définie dans la classe `java.lang.Object` retourne le nom de la classe de l'objet, suivi du caractère arobase `@`, suivi de la valeur retournée par la méthode `hashCode()` en hexadécimal.

La méthode `length()` permet d'obtenir la longueur d'une chaîne de caractères :

```
String message = "Test des chaînes de caractères";
System.out.print  ( "Le message contient " );
```

```
System.out.print  ( message.length() );
System.out.println( " caractères" );
```

Equivaut à :

```
System.out.print  ( "Le message contient " );
System.out.print  ( "Test des chaînes de caractères".length() );
System.out.println( " caractères" );
```

Nous pouvons utiliser des guillemets à l'intérieur d'une chaine de caractères en utilisant le caractère d'échappement \

```
String ma_string = "le mot \"mot\" contient 3 lettres";
```

Nous pouvons également utiliser des caractères spéciaux tels que \n (nouvelle ligne), \t (tabulation) pour formater la sortie.

```
String ma_string_formatee = "colonne 1\tcolonne 2\nvaleur 1\tvaleur 2\n"
```

Enfin, nous pouvons afficher des caractères particuliers (tels les caractères accentués, par exemple) en utilisant le préfixe \u suivi du code Unicode du caractère sur 4 chiffres hexadécimaux :

```
String chaine_unicode = "\u002D"; // le signe moins -
```

Cependant, la séquence `\u` est également utilisable en dehors des chaînes de caractères pour encoder le code source. Exemple :

```
int a = 200 \u002D 50; // le signe moins - -> 150
```

équivaut à :

```
int a = 200 - 50; // 150
```

Extension java.io

Cette extension concerne les flux d'entrées et sorties en général (fichiers, réseau, buffer).

Les deux classes abstraites `InputStream` et `OutputStream` gèrent les entrées et sorties binaires, utilisant des tableaux d'octets.

Les sous-classes de celles-ci sont plus spécialisées :

- `FileInputStream` et `FileOutputStream` gèrent les fichiers,
- `ByteArrayInputStream` et `ByteArrayOutputStream` lisent/écrivent depuis/vers un tableau d'octets.

Les classes `Reader` et `Writer` utilisent les tableaux de caractères.

Leurs sous-classes sont également plus spécialisées :

- `FileReader` et `FileWriter` gèrent les fichiers,
- `StringReader` et `StringWriter` lisent/écrivent depuis/vers une chaîne de caractères.

Extension java.nio

Cette section est vide, pas assez détaillée ou incomplète.

Extension java.text

Cette section est vide, pas assez détaillée ou incomplète.

Extension java.util

Cette extension contient les classes permettant la gestion de collections (liste chaînées, table de hachage, tableau extensible, …). Elle contient également une classe de générateur aléatoire (classe `Random`), deux classes gérant le lancement de tâches périodiques ou à un moment donné (`Timer` et `TimerTask`), la gestion des dates (`Date`, `Calendar`, `GregorianCalandar`), …

Extension java.net

Cette extension contient les classes permettant de communiquer à travers un réseau en utilisant des sockets TCP ou UDP. Il existe également des classes pour gérer les adresses IP (version 4 et 6).

Cette section est vide, pas assez détaillée ou incomplète.

Extension java.awt

Cette extension regroupe les classes graphiques de base.

Les composants graphiques utilisent les composants natifs de la platforme de lancement. Ils sont nommés *Heavyweight*, par opposition à *Lightweight* qui désigne les sous-classes de la classe `Component`. Bien que beaucoup plus rapide, il tend à être remplacé par swing, bien plus fourni.

Extension javax.swing

C'est une extension de classe graphique très fournie.

Les composants sont entièrement développés en Java à partir de la classe `java.awt.Component`. Quelle que soit la plateforme de lancement de l'application, ils ont donc le même comportement et la même apparence, qui peut d'ailleurs être configurée.

Collections

Les collections en Java sont des classes permettant de manipuler les structures de données usuelles : listes, vecteurs, files, piles, etc. Une manière standard de représenter une structure de donnée en Java consiste à regrouper dans une interface l'ensemble des noms des opérations appliquables sur celle-ci (ajout, suppression, effacement, etc.), c'est-à-dire l'ensemble des opérations mentionnées dans le type de données abstrait implémenté par cette structure (sa spécification, indépendamment du choix d'implémentation). Cette interface sera implémentée par chaque classe représentant cette sorte de structure : par exemple, l'interface `List` regroupe un ensemble de noms de méthodes génériques permettant la manipulation de listes, et est implémentée à la fois par les classes concrètes `ArrayList` (implémentation des listes par tableaux extensibles) et `LinkedList` (implémentation des listes par chaînage).

Les interfaces et classes citées dans les exemples ci-dessous sont livrées dans le SDK standard. Elles se trouvent dans le package `java.util`.

Utilisation

```
List<String> ma_liste = new LinkedList<String>();
```

Dans cet exemple, *List* est une interface qui implémente les listes (collection d'éléments ordonnés, éventuellement répétés), on a choisi d'utiliser une liste chaînée pour représenter la liste en mémoire. *LinkedList* est la classe dont les éléments de la collection seront des instances.

Listes (List)

Cette interface est implémentée par un certain nombre de collections, et garantit que ces classes implémenteront l'ensemble des méthodes. Elle dérive de l'interface `Collection`. Les éléments sont indexés (i.e. numérotés de la même façon qu'un tableau est indicé).

Méthodes sur les listes

Type	Méthode[1]	Rôle
`boolean`	**`add`**`(int index, Object o)`	Ajouter un objet à l'index indiqué.
`boolean`	**`addAll`**`(int index, Collection c)`	Ajouter tous les objets d'une autre collection à l'index indiqué.
`Object`	**`get`**`(int index)`	Retourner l'objet à l'index indiqué.
`int`	**`indexOf`**`(Object o)`	Retourner le premier index de l'objet indiqué.
`int`	**`lastIndexOf`**`(Object o)`	Retourner le dernier index de l'objet indiqué.
`Object`	**`remove`**`(int index)`	Supprimer l'objet à l'index indiqué.
`Object`	**`set`**`(int index, Object o)`	Remplacer l'objet à l'index indiqué. L'objet précédent est retourné.
`int`	**`size`**`()`	Retourner le nombre d'éléments de la liste.
`List`	**`subList`**`(int fromIndex,int toIndex)`	Retourner une sous-liste de celle-ci.

Les différentes implémentations

On peut utiliser des tableaux redimensionnables (ArrayList) ou des listes chaînées (LinkedList)

Listes chaînées (`LinkedList`)

Cette classe implémente l'interface `List` en chaînant les éléments (liste doublement chaînée).

Les méthodes ajoutées sont :

`void addFirst(Object o)`
: Ajoute un élément en début de liste.

`void addLast(Object o)`
: Ajoute un élément en fin de liste.

`Object getFirst()`
: Retourne l'élément en début de liste.

`Object getLast()`
: Retourne l'élément en fin de liste.

`Object removeFirst()`
: Supprime et retourne l'élément en début de liste.

`Object removeLast()`
: Supprime et retourne l'élément en fin de liste.

Tableau redimensionnable (`ArrayList`)

Cette classe est un tableau dont la taille croit lorsque des éléments sont ajoutés.

Tableau redimensionnable (`Vector`)

Cette classe est un tableau dont la taille croit lorsque des éléments sont ajoutés. Cette classe implémente les méthodes de l'interface `List` et les suivantes :

`int indexOf(Object o,int index)`
: Retourne l'index de l'objet indiqué, en partant de l'index indiqué.

`int lastIndexOf(Object o,int index)`
: Retourne l'index de l'objet indiqué, en partant de l'index indiqué et en allant vers le début (index 0).

`void setSize(int newSize)`
: Tronquer/Agrandir le vecteur à la taille indiquée.

Cette classe a été créée avant la classe `ArrayList`. Elle est synchronisée : durant l'appel à une méthode de cette classe par un thread, un autre thread ne peut modifier le tableau.

Files (Queue)

Files avec priorités (PriorityQueue)

Pour utiliser une file avec priorités, les éléments doivent être des instances d'une classe qui implémente l'interface *Comparator*. Il faudra écrire la méthode *int compare(Object, Object)* qui compare les deux instances et renvoie un entier.

Si vous ne voulez ou ne pouvez pas modifier la classe qui décrit les éléments. Vous pouvez créer, dans la classe où vous utilisez votre file avec priorité, une classe interne qui hérite de la classe des éléments et implémente l'interface *Comparator*. Exemple, si vous voulez créer une file avec priorité de *Bidules* :

```java
import mon_package.Bidule;

public class MaClasseQuiUtiliseUneFileDeBidules
{
    /**
     * Classe permettant de créer une file avec priorité de Bidules
     *
     */
    public class ComparableBidule implements Comparator<Bidule>
```

```
    {
        public int compare(Bidule b1, Bidule b2)
        {
            // On retourne le signe de la soustraction abstraite b1 - b2
            // Si b1 <  b2  (ou b1 à classer avant b2) ---> retourner -1
            // Si b1 == b2  (ou b1 équivaut à b2)      ---> retourner  0
            // Si b1 >  b2  (ou b1 à classer après b2) ---> retourner +1
            ...
        }
    }

    public void une_methode()
    {
        PriorityQueue<ComparableBidule> f = new LinkedList<ComparableBidule>();
        Bidule b = new Bidule();
        f.add(b)
    }
}
```

Piles (`Stack`)

Une pile contient une liste d'objets. Il est possible d'ajouter un objet au sommet de la pile (empiler, ou *push* en anglais), et de retirer l'objet situé au sommet de la pile (dépiler, ou *pop* en anglais).

Exemple: En partant d'une pile vide, on effectue les opérations suivantes :

```
État de la pile     Opération
(vide)
                    empiler A
A
                    empiler B
A B
                    empiler C
A B C
                    dépiler -> C
A B
                    dépiler -> B
A
                    empiler D
A D
```

La classe `Stack` est une implémentation de pile qui dérive de la classe `Vector` et ajoute les méthodes suivantes pour gérer les objets comme une pile :

`boolean empty()`
: Retourne vrai si la pile est vide.

`Object peek()`
: Retourne l'objet au sommet de la pile sans l'enlever.

`Object pop()`
: Retourne et enlève l'objet au sommet de la pile.

`Object push(Object o)`
: Ajoute l'objet au sommet de la pile.

`int search(Object o)`
: Retourne l'index de l'objet depuis le sommet de la pile (1 = sommet de la pile, -1 = non trouvé).

Ensembles (Set)

Conformément à l'idée mathématique, les ensembles représentent plusieurs éléments non triés, sans répétitions. Les ajouts d'éléments déjà présents sont donc ignorés. Un élément est déjà présent si un tests equals sur un des éléments de l'ensemble renvoie vrai.

Cette interface est implémentée par un certain nombre de collections, et garantit que ces classes implémenteront l'ensemble des méthodes. Elle dérive de l'interface `Collection`, sans ajouter de nouvelles méthodes. Elle sert seulement à indiquer informellement que la collection implémentant cette interface ne contient aucun doublon d'objet (objets comparés par la méthode `equals`). Cette interface correspond donc aux ensembles mathématiques.

Les différentes implémentations

- La classe `HashSet` implémente l'interface `Set` en utilisant une table de hachage.
- `TreeSet` utilise un un arbre de recherche. Pour pouvoir utiliser un `TreeSet`, il faut que les éléments soit comparables. Cette fonction est plus lente que `HashSet`[2].
- `LinkedHashSet` diffère de `HashSet` car il maintient une liste doublement liée à travers toutes ses entrées, permettant de retrouver l'ordre d'insertion (mais pas pour les réinsertions).

Ensembles triés (SortedSet)

Les ensembles triés sont identiques aux ensembles simples excepté qu'ils peuvent être triés par défaut à leur création selon un tri dit *naturel* ou qu'un motif de tri a été appliqué. La méthode *comparator()* de cette interface permet de retourner le motif de tri utilisé ou retourne *null* si le tri est effectué de façon naturel en fonction du type des données.

`Range view`
: Permet des opérations sur les plages d'ensembles triés.

`Endpoints`
: Renvoie le premier ou dernier élément d'un ensemble trié.

`Comparator access`
: Renvoie le comparateur utilisé pour classer l'ensemble.

Tableaux associatifs (Map)

Cette interface est implémentée par les collections qui associent une clé à un objet. L'accès aux objets est donc effectué par une clé unique.

Il est possible d'utiliser n'importe quelle instance de classe comme clé. Cependant si cette classe ne possède pas les propriétés nécessaires, il faut utiliser exactement la même instance pour accéder à une valeur (voir Objets comparables et clés).

Les principales méthodes de cette interface sont :

`void clear()`
: Vider la collection.

`boolean containsKey(Object key)`
: Teste si la clé existe, c'est à dire associée à une valeur.

`boolean containsValue(Object value)`
: Teste si la valeur existe.

`Set entrySet()`
: Retourne l'ensemble des associations clés-valeurs.

`Set keySet()`
: Retourne l'ensemble des clés.

`Collection values()`
: Retourne la collection de valeurs.

`Object put(Object key, Object value)`
: Associe la clé à la valeur spécifiée, et retourne la valeur précédemment associée.

`boolean putAll(Map m)`
: Ajouter tous les objets d'une autre collection à celle-ci.

`Object get(Object key)`
: Retourne la valeur associée à la clé spécifiée, ou `null` si non trouvé.

`Object remove(Object key)`

Supprime l'objet associé à la clé, et retourne cet objet.

`boolean isEmpty()`
: Tester si la collection est vide.

`int size()`
: Retourne le nombre de clés.

Les implémentations

La classe `Hashtable` implémente l'interface `Map` de manière synchronisée.

La classe `HashMap` implémente l'interface `Map`, et permet d'utiliser la clé `null`.

Les tableaux triés (SortedMap)

Exactement comme SortedSet :

`Range view`
: Permet des opérations sur les plages de tableaux triés.

`Endpoints`
: Renvoie le premier ou dernier élément d'un tableau trié.

`Comparator access`
: Renvoie le comparateur utilisé pour classer le tableau.

Parcourir une collection

Les itérateurs

Les itérateurs sont des outils puissants pour parcourir le contenu d'une collection.

```
List<String> ma_liste = new LinkedList<String>();
ma_liste.add("Bonjour");
ma_liste.add(" le ");
ma_liste.add("monde");

Iterator<String> it = ma_liste.iterator(); // On paramètre Iterator par le type des éléments de
while (it.hasNext()) {
  System.out.println(it.next()); // Affiche "Bonjour le monde"
}
```

Attention, l'appel de next() renvoie l'élément courant dans le parcours et passe l'itérateur à l'élément suivant.

Pour pouvoir itérer sur une collection, il faut qu'elle soit itérable, c'est à dire qu'elle hérite de *Iterable*. C'est le cas la plupart du temps mais il y a des exceptions (HashMap).

L'utilisation des itérateurs requiert la classe Iterator définie dans le package *java.util*.

Boucles « for each »

```
// Parcours d'une collection
Collection<ObjetX> collection = ......;

for(ObjetX objetX : collection){
   objetX.methodeY(p1,p2);
}
```

Attention aux parcours des `Map`

Le code suivant convient tout a fait si vous avez besoin de parcourir les clés de la Map sans vous préoccuper des valeurs

```
Map<Key, Value> map;
for (Key key : map.keySet()) {
    // ...
}
```

Le code suivant est à proscrire si vous parcourrez l'ensemble des clés et des valeurs qui leurs sont associées

```
for (Key key : map.keySet()) {
    // l'appel de get engendre un temps coûteux en accès à chaque itération
    Value value = map.get(key);
}
```

Dans le cas précédent, préférez le code suivant

```
for (Map.Entry<Key, Value> entry : map.entrySet()) {
    Key key = entry.getKey();
    Value value = entry.getValue();
    // ...
}
```

Interface commune

La plupart des structures évoquées ci-dessus implémentent l'interface Collection et possèdent donc un ensemble de méthodes communes. Les principales méthodes sont :

`boolean add(Object o)`
: Ajouter un objet à la collection.

`boolean remove(Object o)`
: Retirer un objet de la collection.

`boolean contains(Object o)`
: Tester si la collection contient l'objet indiqué.

`boolean addAll(Collection c)`
: Ajouter tous les objets d'une autre collection à celle-ci.

`boolean removeAll(Collection c)`
: Retirer tous les objets d'une autre collection de celle-ci.

`boolean retainAll(Collection c)`
: Retirer tous les objets qui ne sont pas dans la collection spécifiée de celle-ci. À la fin les deux collections contiennent les mêmes objets.

`boolean containsAll(Collection c)`
: Tester si la collection contient tous les objets de la collection indiquée.

`void clear()`
: Vider la collection.

`boolean isEmpty()`
: Tester si la collection est vide.

`Iterator iterator()`
: Retourne un itérateur permettant de faire une boucle sur tous les objets contenus dans la collection.

`int size()`
: Retourne le nombre d'objets de la collection.

`Object[] toArray()`
: Convertit la collection en tableau d'objets.

`Object[] toArray(Object[] a)`
: Convertit la collection en tableau d'objets de classe spécifiée. Si le tableau fourni n'est pas assez grand, un nouveau tableau est alloué en utilisant la même classe d'élément.

Tableau récapitulatif

Interface	Implémentations			
	Tableau dynamique	Chaînage	Arborescence[3]	Table de hachage
Liste (List)	ArrayList, Vector	LinkedList		
Ensemble (Set)			TreeSet	HashSet
File (Queue)				
Tableau associatif (Map)				HashMap

Synchronisation

Si une collection est accédée par plusieurs threads, il faut utiliser la synchronisation afin que les modifications soient cohérentes, c'est à dire exécutées sans être interrompues par une modification effectuée par un autre thread.

Une collection synchronisée garantit que l'appel d'une méthode de la collection sera effectuée sans qu'aucun autre thread ne modifie cette collection entre-temps. Pour obtenir une collection synchronisée, vous devez appeler une méthode de la classe `Collections`, dont le nom dépend du type de collection :

```
type variable = Collections.synchronizedtype( new type(...) );
```

Exemple :

```
Set myset = Collections.synchronizedSet( new HashSet() );
```

Cependant, le thread peut être interrompu entre deux appels de méthode. Dans ce cas, vous devez synchroniser l'ensemble des instructions appelant les méthodes de la collection (mot-clé `synchronized`). Voir le chapitre "Threads et synchronisation". Autrement dit, les listes ne sont pas thread-safe, aucune méthode n'est atomique. Les sources de ces classes sont d'ailleurs consultables si vous avez accès au JDK.

La solution précédente peut résoudre tous les problèmes de synchronisation, par exemple celui ci :

```
Map<String,String> map=new HashMap<String,String>();
map.put("toto", "valeur exemple 1");
map.put("titi", "valeur exemple 1");
```

...

```
synchronized {
  if (!map.containsKey("titi"))
    bd.put("titi", "valeur exemple 3");
}
```

Une modification ne peut peut avoir lieu, par un autre thread, entre le containsKey et le put. Mais cette solution est peut efficace en réalité du fait car elle contraint à protéger tous les accès à la liste en lecture comme en écriture. Alors que les accès en lectures multiples n'ont pas à être bloquant, seulement ceux en écriture.

La solution la plus efficace est d'utiliser les listes de la bibliothèque java.util.concurrent qui possède un putIfAbsent. En outre, les méthodes de cette bibliothèque peuvent être considérées comme toujours plus rapide en mode multi-thread que celle d'une liste synchronisée,

Classes squelettes

Une interface possède souvent une longue série de méthodes à implémenter. Certaines sont implémentées généralement de la même manière par beaucoup de classes.

Plutôt que de répéter le code de ces méthodes dans les différentes implémentations, il est préférable de rassembler leur code dans une classe abstraite (car toutes les méthodes de l'interface ne sont pas implémentées).

Les classes de l'API Java utilisent de telles classes dont dérivent les différentes implémentations.

Ces classes sont :

- `AbstractCollection` pour l'interface `Collection`,
- `AbstractList` pour l'interface `List`,
- `AbstractSequentialList` pour l'interface `List`,
- `AbstractSet` pour l'interface `Set`,
- `AbstractMap` pour l'interface `Map`.

Elles ne sont pas instanciables directement car abstraites, mais servent de classes de base. Il est utile de connaître ces classes pour rechercher dans la documentation, ou réaliser une implémentation alternative à celle proposée par l'API.

Notes

1. http://docs.oracle.com/javase/7/docs/api/java/util/List.html
2. http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashSet.html
3. L'utilisation des arbres de recherche requiert que les éléments soient triables (*sortable*)

Annotations

Un nouveau principe introduit par Java 5 est la gestion des méta-données grâce à un mécanisme appelé *annotations*.

Ces annotations sont ajoutées dans le code devant les classes et leurs membres (méthodes et champs) pour :

- Ajouter des informations de documentation supplémentaires aux commentaires Javadoc,
- Ajouter des contraintes de compilation (voir @Override par exemple),
- Associer des méta-données qui peuvent être retrouvées par réflexion.

Malgré la similarité de syntaxe (arobase + nom) et de nom (@deprecated par exemple), il ne faut pas les confondre avec les commentaires Javadoc qui ne sont utilisés que par l'outil javadoc du JDK. De plus, une annotation correspond à un type de données (comme les classes, interfaces et énumérations). Ceci explique que la documentation d'un package possède une section supplémentaire nommée « *Annotation Types* » pour les annotations définies dans le package.

Fonctionnement

Toute entité (classe, interface, méthode ou champ) peut être précédée d'une ou plusieurs annotations contenant des méta-données renseignant le compilateur ou l'application elle-même. Ces annotations sont accessibles à l'exécution, en utilisant la réflexion, à partir de nouvelles méthodes de la classe `java.lang.Class`.

Syntaxe

La syntaxe d'une annotation est la suivante :

```
@annotation-type[ (name=value) * ]
```

La liste des paramètres entre parenthèse est optionnelle (vide par défaut). Elle contient une série de valeurs associées à un nom de champ définit par l'annotation.

Exemple 1 :

```
@Author(
    name = "Moi",
    date = "02/01/2009"
)
class MyClass() { }
```

Exemple 2 :

```
@Override
void uneMethode() { }
```

Annotations existantes

Beaucoup d'annotations complètent les balises spéciales pour la documentation Javadoc.

@Deprecated

```
java.lang.Deprecated
```

Cette annotation marque une entité obsolète. Son utilisation génère un avertissement à la compilation, contrairement

au tag @deprecated des commentaires Javadoc.

@Override

```
java.lang.Override
```

Cette annotation marque une méthode redéfinie.

Il ne s'agit pas d'une annotation de documentation mais d'un **ajout de contrainte** vérifiée à la compilation : une méthode marquée avec cette annotation **doit obligatoirement** être une méthode redéfinie de la classe mère. Dans le cas contraire (méthode non définie dans la classe mère), le compilateur génère une erreur *annotation type not applicable to this kind of declaration*.

À partir de Java 6, cette annotation peut aussi être utilisée pour les méthodes implémentant une interface.

@SuppressWarnings

```
java.lang.SuppressWarnings
```

Cette annotation signale au compilateur de supprimer certains avertissements à la compilation de l'entité.

Exemple 1 : pour supprimer les avertissements d'utilisations de méthodes obsolètes :

```
@SuppressWarnings("deprecation")
void uneMethode() { methodeObosolete(); }
```

Exemple 2 : pour supprimer les avertissements d'utilisations de méthodes obsolètes et d'utilisation de méthodes sans vérification de types (une version de la méthode avec types générique est préférable afin que le type d'élément soit vérifié) :

```
@SuppressWarnings({"unchecked", "deprecation"})
void uneMethode()
{
    Vector v=new Vector();
    methodeObosolete();
}
```

Créer de nouvelles annotations

La création est similaire à celle d'une interface.

Syntaxe

```
@interface identifiant {
   type champ() [ default valeur ];
}
```

Exemple de définition:

```
@interface InfoClasse
{
    String auteur();
    int revision() default 1;
    String[] references();
```

```
}
```

Exemple d'utilisation:

```
@InfoClasse( auteur="Moi", references={"Reference1","Reference2"} )
class UneClasse { }
```

Cette nouvelle annotation peut elle-même être taguée avec des annotations du package `java.lang.annotation` indiquant l'utilisation qui en est faite.

Documentation pour Javadoc

Si l'annotation définit des informations à afficher dans la documentation générée par Javadoc, la définition de l'annotation doit utiliser l'annotation `@Documented` (`java.lang.annotation.Documented`).

Correction de l'exemple précédent :

```
@Documented
@interface InfoClasse
{
    String auteur();
    int revision() default 1;
    String[] references();
}
```

Informations disponibles à l'exécution

Pour que les informations d'une annotation soient disponibles à l'exécution, il faut annoter sa définition avec `@Retention(RetentionPolicy.RUNTIME)`.

Exemple :

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@interface AnnotationPourExecution
{
   // Éléments d'information disponibles à l'exécution
}
```

Restreindre l'utilisation d'une annotation

La définition d'une annotation peut être annotée avec `@Target` pour spécifier avec quels types d'éléments l'annotation peut être utilisée.

La classe `java.lang.annotation.ElementType` définit les différents types qu'il est possible d'annoter.

Exemple :

```
import java.lang.annotation.*;

@Target(ElementType.ANNOTATION_TYPE)
@interface PourAutreAnnotation
{
}

@Target({ElementType.FIELD, ElementType.METHOD, ElementType.TYPE})
@interface InfoMembreOuType
```

```
{
// Type = class/interface/enum
}
```

Accès aux annotations d'une classe

L'API de réflexion de Java permet d'accéder aux annotations d'une classe, méthode ou champ.

Exemple :

```
import java.lang.annotation.*;

// ...

Class clas = objet.getClass(); // ou à partir d'une classe connue :   UneClasseConnue.class
Annotation[] annotations = clas.getAnnotations();

for (Annotation annotation : annotations)
{
    if (annotation instanceof InfoClasse)
    {
        InfoClasse info_classe = (InfoClasse) annotation;
        System.out.println("auteur : " + info_classe.auteur());
        System.out.println("revision : " + info_classe.revision());
    }
}
```

Entrées Sorties

Les opérations d'entrées-sorties concernent la lecture (entrée) et l'écriture (sortie) de données à travers différents types de flux...

En Java, les opérations d'entrées-sorties de base sont gérées par les classes du package `java.io`. Ces classes obéissent au patron de conception décorateur. Ces objets sont souvent créés au sein d'objets correspondant au patron de conception fabrique.

Flux d'entrée-sortie

Le package `java.io` possède deux classes principales :

- `InputStream` : cette classe abstraite définit les fonctions de lecture (entrée ou *input* en anglais),
- `OutputStream` : cette classe abstraite définit les fonctions d'écriture (sortie ou *output* en anglais).

Ces deux classes abstraites définissent des fonctions bas-niveau et sont implémentées dans différentes sous-classes concrètes. Trois types de flux sont possibles, les flux sur fichiers, les flux sur tubes, et les flux sur zones de mémoire. Par exemple on a :

- `FileInputStream` : lecture d'un fichier,
- `FileOutputStream` : écriture d'un fichier.

La classe `java.net.Socket` possèdent des méthodes retournant des instances concrètes des classes `InputStream` et `OutputStream` pour lire et écrire depuis/vers la socket TCP.

java.io.InputStream

La classe `java.io.InputStream` possèdent des méthodes lisant une série d'octets (byte).

- `InputStream.read()`: lit un byte
- `InputStream.read(byte[] b)`: lit `b.length` bytes.
- `InputStream.read(byte[] b, int off, int len)`: lit `len` bytes, et les dépose commencer à `b[off]`.

java.io.OutputStream

La classe `java.io.OutputStream` possède des méthodes écrivant une série d'octets (byte).

- `OutputStream.write()`: écrit un byte
- `OutputStream.write(byte[] b)`: écrit `b.length` bytes.
- `OutputStream.write(byte[] b, int off, int len)`: écrit `len` bytes, commencer à `b[off]`.

java.io.PrintStream

La classe `java.io.PrintStream` hérite de la classe `java.io.OutputStream`, et permet d'afficher tous les types de données sous forme textuelle.

La sortie standard et l'erreur standard impriment sur la console et sont des instances de la classe `java.io.PrintStream`.

Exemple :

```
system.out.println("Bonjour !"); // Affiche une chaîne avec retour à la ligne

int count = 100;
system.out.print(count); // Affiche un entier sans retour à la ligne
```

```
system.out.print(' ');    // Affiche un caractère
```

Tout objet peut être affiché car les méthodes `print` et `println` appellent la méthode `ToString()` définie dans la classe `java.lang.Object` racine de l'arbre hiérarchique de tous les types d'objets.

Exemple :

```
class NombreComplexe
{
    double n_real, n_img;

    public NombreComplexe(double r, double i)
    {
        this.n_real = r;
        this.n_img = i;
    }

    public String toString()
    {
        return n_real + " + i*"+n_img;
    }
}

NombreComplexe a = new NombreComplexe(1.0, 0.5);
system.out.println(a); // Appelle println(Object) pour afficher :
//                        1.0 + i*0.5
```

La méthode `toString()` est également appelée implicitement lors de la concaténation de chaînes de caractères :

```
String resultat = "Solution complexe :" + a;
// -> Solution complexe : 1.0 + i*0.5
```

Il est donc important que cette méthode n'ait aucun effet de bord (modification d'un objet, synchronisation, ...).

Lecture-écriture haut niveau

Le package `java.io` possède des classes permettant la lecture et l'écriture de différents types de données.

Reader, Writer, PrintStream

Cette section est vide, pas assez détaillée ou incomplète.

Exemple concret de flux

Fichiers textes

Cette section est vide, pas assez détaillée ou incomplète.

Images

```
import javax.imageio.ImageIO;
ImageIO.read(image);
ImageIO.write(imgage, "jpg", new File(filename));
```

Cette section est vide, pas assez détaillée ou incomplète.

Consoles

Cette section est vide, pas assez détaillée ou incomplète.

Réseaux

Cette section est vide, pas assez détaillée ou incomplète.

Processus légers et synchronisation

Les processus légers *(threads)*, ou fils d'exécution, permettent l'exécution de plusieurs tâches en même temps.

Qu'est ce qu'un processus léger ?

Un processus léger est un contexte d'exécution d'une application. Ce processus possède sa propre pile et pointeur d'exécution.

Une application en cours d'exécution (un processus) peut avoir plusieurs sous-processus léger. Tous les processus légers d'un même processus partagent la même zone de données. Ce qui veut dire que toute variable membre d'une classe est modifiable par n'importe quel processus léger. Il faut donc un moyen de synchroniser l'accès aux variables (voir paragraphe *Synchronisation*).

Par défaut, une application possède un seul processus léger, créé par le système. Cependant, en Java, d'autres processus légers sont créés quand l'application utilise une interface graphique, notamment un processus léger gérant la boucle de lecture des messages systèmes.

Processus léger courant

En Java, tout processus léger est représenté par un objet de classe `Thread`. Le processus léger courant est retourné par la méthode statique `currentThread` de la classe `Thread`.

La classe `Thread` possède quelques méthodes statiques agissant sur le processus léger courant :

- la méthode `sleep(long millis)` permet de suspendre le processus léger durant le temps donné en millisecondes;
- la méthode `yield()` permet de laisser les autres processus légers s'exécuter;
- la méthode `interrupted()` teste si le processus léger courant a été interrompu ;
- la méthode `dumpStack()` affiche la pile d'appel du processus léger courant (déverminage, ou débuggage en franglais).

Créer un processus léger

La classe `Thread` peut être dérivée pour créer un autre processus léger. Dans ce cas, il faut surcharger la méthode `run()` pour y mettre le code exécuté par le processus léger.

Exemple :

```
public class MyThread extends Thread
{
    public void run()
    {
        System.out.println("Un nouveau processus léger");
        Thread.sleep(1000); // suspendu pendant 1 seconde
        System.out.println("Fin du nouveau processus léger");
    }
}
```

Il est alors créé et démarré de la manière suivante :

```
MyThread myth=new MyThread();
System.err.println("Démarrer le processus léger ...");
myth.start();
System.err.println("Le processus léger est démarré.");
```

Il n'est pas toujours possible d'étendre la classe `Thread` car Java n'autorise qu'une classe de base. Mais il est permis

d'utiliser plusieurs interfaces. L'interface `Runnable` permet de résoudre le problème.

Par défaut, la méthode `run()` de la classe `Thread` appelle la méthode `run()` de l'interface `Runnable` passé en paramètre du constructeur.

Exemple :

```
public class MyClass extends AnotherClass
implements Runnable
{
    public void run()
    {
        System.out.println("Un nouveau processus léger");
        Thread.sleep(1000); // suspendu pendant 1 seconde
        System.out.println("Fin du nouveau processus léger");
    }
}
```

Le processus léger est alors créé et démarré de la manière suivante :

```
MyClass myclass=new MyClass ();
Thread th=new Thread(myclass); // <-- processus léger créé
System.err.println("Démarrer le processus léger ...");
th.start();
System.err.println("Le processus léger est démarré.");
```

Actions sur un processus léger

Cycle de vie d'un processus léger

Un processus léger possède différents états gérés par le système :

- état *prêt* : le processus est prêt à être exécuté,
- état *suspendu* : le processus est suspendu (attente d'une ressource),
- état *exécution* : le processus est en cours d'exécution,
- état *terminé* : le processus a achevé son exécution ou a été interrompu.

InterruptedException

Cette classe d'exception est lancée par les méthodes de la classe `Thread` et celle de la classe `Object` demandant la suspension pour un temps indéterminé du processus léger courant (attente en général).

Cette exception est lancée quand le processus léger en attente est interrompu. Capturer cette exception permet d'interrompre l'attente, et libérer des ressources pour terminer proprement.

Attendre la fin d'un processus léger

La méthode `join()` de la classe `Thread` peut être appelée pour attendre la fin d'un processus léger.

Exemple :

```
th.join(); // InterruptedException à capturer
```

Interrompre un processus léger

La méthode `interrupt()` de la classe `Thread` peut être appelée pour interrompre un processus léger. Cette méthode

provoque le lancement d'une exception de type `InterruptedException` quand le processus appelle une méthode d'attente.

Synchronisation

La synchronisation devient nécessaire quand plusieurs processus légers accèdent aux mêmes objets.

mot-clé synchronized

Le mot-clé `synchronized` permet un accès exclusif à un objet.

La syntaxe est la suivante :

```
... code non protégé ...
synchronized(objet) {
    ... code protégé ...
}
... code non protégé ...
```

Le code protégé n'est exécuté que par un seul processus léger à la fois, tant qu'il n'a pas terminé le bloc d'instruction.

Durant l'exécution de ce code protégé par un processus léger, un autre processus léger ne peut exécuter celui-ci, mais peut exécuter un autre bloc `synchronized` si celui-ci n'utilise pas le même objet et qu'il n'est pas déjà en cours d'exécution.

Le mot-clé `synchronized` peut également être utilisé dans la déclaration des méthodes :

```
public synchronized void codeProtege() {
    ... code protégé ...
}
```

est équivalent à :

```
public void codeProtege() {
    synchronized(this)
    {
        ... code protégé ...
    }
}
```

Pour une méthode statique (méthode de classe) :

```
public class MyClass {
    public synchronized static void codeProtege() {
        ... code protégé ...
    }
}
```

est équivalent à :

```
public class MyClass {
    public static void codeProtege() {
        synchronized(MyClass.class) {
            ... code protégé ...
        }
    }
}
```

Attente et signal

Quand le mot-clé `synchronized` ne suffit pas (par exemple, permettre l'accès à deux processus légers simultanément au lieu d'un seul), il est possible de suspendre un processus léger et le réveiller.

La classe `Object` possède les méthodes suivantes :

- `wait()` suspend le processus courant jusqu'à ce que la méthode `notify()` ou `notifyAll()` de cet objet soit appelée ;
- `wait(long timeout)` suspend le processus courant jusqu'à ce que la méthode `notify()` ou `notifyAll()` de cet objet soit appelée, ou bien que le temps indiqué soit écoulé ;
- `notify()` réveille l'un des processus en attente de cet objet,
- `notifyAll()` réveille tous les processus en attente de cet objet.

Pour appeler l'une de ces quatre méthodes, il faut *posséder* l'objet. Ce qui signifie utiliser l'instruction `synchronized`. Dans le cas contraire, l'exception suivante est levée :

```
java.lang.IllegalMonitorStateException: current thread not owner
```

Exemple :

```
synchronized(myobj)
{
    myobj.wait();
}
```

Comme ces méthodes sont définies dans la classe `Object`, il est possible de les utiliser avec n'importe quel type d'objet, et donc les chaînes de caractères et les tableaux.

Une bibliothèque spécialisée

Il est très courant d'être amené à protéger l'accès à une variable pour juste par exemple, une incrémentation sous condition, comparer et échanger deux valeurs, chercher et ajouter. Ces méthodes sont qualifiées d'atomiques. La solution la plus efficace consiste à utiliser les classes de la bibliothèque java.util.concurrent. Ces classes disposent de méthodes considérées comme toujours plus rapides en mode multi-thread que celles d'une liste synchronisée, prenons par exemple le cas des listes :

```
String str;
Map<String,String> map=new HashMap<String,String>();
map.put("toto", "valeur exemple 1");
map.put("titi", "valeur exemple 1");
...
synchronized { str=map.get("tutu");}
```

...

```
synchronized {
  if (!map.containsKey("titi"))
    bd.put("titi", "valeur exemple 3");
}
```

Un autre thread ne peut pas effectuer de modification entre containsKey et put, du fait du verrou posé par l'instruction synchronized. Mais cette solution est en réalité peu efficace car elle contraint le plus souvent à protéger tous les accès à la liste, en lecture comme en écriture, alors que les accès en lectures multiples n'ont pas à être bloquants, seuls ceux en écriture devant l'être. La classe ConccurentHashMap possède un putIfAbsent. Il est possible de réimplanter cette caractéristique via la synchronisation sur deux valeurs, mais les classes fournies par java.util.concurrent sont, elles,

exemptes de bugs.

Si la classe de liste ne vous satisfait pas, la pose de verrou via la classe ReentrantReadWriteLock est extrêmement simple.

Méthodes natives

Une classe peut posséder des méthodes natives. Une méthode native n'est pas implémentée en Java mais dans un autre langage de programmation (C ou C++ le plus souvent).

Le mot-clé `native` marque les méthodes natives d'une classe Java. Elles n'ont pas de corps d'implémentation (comme les méthodes abstraites).

L'outil javah du JDK permet de générer l'en-tête C/C++ (*.h) correspondant aux méthodes natives d'une classe.

Pour plus de détails, voir Développer en Java/Faire appel à du code natif.

Réflexion

La réflexion (*reflection* en anglais) permet l'introspection des classes, c'est-à-dire de charger une classe, d'en créer une instance et d'accéder aux membres statiques ou non (appel de méthodes, lire et écrire les attributs) sans connaître la classe par avance.

Java possède une API permettant la réflexion. Elle sert notamment à gérer des extensions (*plug-in* en anglais) pour une application.

L'API de réflexion

Le package `java.lang` possède trois classes utilisées pour l'utilisation dynamique des classes :

`java.lang.Class`
: Cette classe permet d'accéder aux caractéristiques d'une classe, à ses membres (méthodes et attributs), à la classe mère.

`java.lang.ClassLoader`
: Cette classe permet de gérer le chargement de classe. Il existe des sous-classes dont notamment `java.net.URLClassLoader` permettant de charger des classes en les cherchant dans une liste d'URLs (donc de fichiers JAR et répertoires également, en convertissant le chemin du fichier ou répertoire en URL).

`java.lang.Package`
: Cette classe permet d'accéder aux informations d'un package (informations de version, annotations, ...).

Les autres classes utiles sont définies dans le package `java.lang.reflect` et permettent d'accéder aux détails d'une classe. Les principales classes sont les suivantes :

`java.lang.reflect.Constructor`
: Référence à un constructeur d'une classe.

`java.lang.reflect.Method`
: Référence à une méthode d'une classe.

`java.lang.reflect.Field`
: Référence à un champs d'une classe.

`java.lang.reflect.Modifier`
: Attributs et méthodes statiques pour décoder les modificateurs des membres (public, private, protected, static, abstract, final, native, ...).

Les classes représentant des membres d'une classe (`Constructor`, `Method`, `Field`) implémentent toutes l'interface `java.lang.reflect.Member` comportant les méthodes suivantes :

`Class getDeclaringClass()`
: Retourne la classe définissant ce membre.

`String getName()`
: Retourne le nom.

`int getModifiers()`
: Retourne les modificateurs (public, protected, private, static, final, ...).

`boolean isSynthetic()`
: Teste si ce membre a été généré par le compilateur.

Charger une classe dynamiquement

La classe `java.lang.Class` possède deux méthodes statique pour obtenir une classe (après chargement si nécessaire) :

`static Class forName(String name)`
: Cette méthode équivaut à appeler la seconde méthode avec pour paramètres : `(name, true, this.getClass().getClassLoader())`.

static Class forName(String name, boolean initialize, ClassLoader loader)
: Charge la classe dont le nom complet (incluant les packages) est spécifié, en utilisant l'instance du chargeur de classe fourni. Le paramètre `initialize` vaut `true` pour initialiser la classe (appeler le bloc d'initialisation statique), ou `false` pour ne pas l'initialiser.

Il est également possible d'obtenir une `java.lang.Class` de manière statique :

- à partir d'un objet en appelant la méthode `getClass()`,
- à partir d'une référence à la classe en utilisant le champ `class`.

Exemple :

```java
package org.wikibooks.fr;

public class Exemple
{
    String nom;
    public String getNom()
    { return nom; }
}
...

Class c = Class.forName("org.wikibooks.fr.Exemple"); // sans référence statique à la classe
// ou
Class c = org.wikibooks.fr.Exemple.class; // référence statique à la classe
// ou
Class c = new Exemple().getClass(); // référence statique à la classe
```

Liste des membres d'une classe

Les méthodes suivantes permettent de lister les membres d'une classe :

getConstructors()
: Cette méthode retourne un tableau de `java.lang.reflect.Constructor` contenant tous les constructeurs définis par la classe.

getMethods()
: Cette méthode retourne un tableau de `java.lang.reflect.Method` contenant toutes les méthodes définies par la classe.

getFields()
: Cette méthode retourne un tableau de `java.lang.reflect.Field` contenant tous les attributs définis dans la classe.

Les méthodes ci-dessus retournent les membres publics de la classe, comprenant également ceux hérités des classes mères. Il existe une variante "Declared" de ces méthodes retournant tous les membres (publics, protégés, privés) déclarés par la classe uniquement (les membres hérités sont exclus).

Au lieu de lister tous les membres, puis en rechercher un en particulier, il est possible d'utiliser les méthodes spécifiques de recherche d'un membre précis d'une classe (publics et hérités, ou bien "Declared" pour tous ceux déclarés par la classe seule) :

getConstructor(Class... parameterTypes)
getDeclaredConstructor(Class... parameterTypes)
: Cette méthode retourne le constructeur déclaré avec les paramètres dont les types sont spécifiés.

getMethod(String name, Class... parameterTypes)
getDeclaredMethod(String name, Class... parameterTypes)
: Cette méthode retourne la méthode portant de nom spécifié et déclarée avec les paramètres dont les types sont spécifiés.

getField(String name)
getDeclaredField(String name)
: Cette méthode retourne l'attribut portant de nom spécifié.

Les membres retournés par toutes ces méthodes peuvent être d'instance ou statiques.

La méthode `getAnnotations()` retourne un tableau de `java.lang.Annotation` contenant toutes les annotations associées à la classe.

Instancier une classe et appel à un constructeur

La méthode `newInstance()` de la classe `java.lang.Class` permet de créer une nouvelle instance de la classe, en appelant le constructeur sans paramètre de la classe (qui doit donc en posséder un) :

```
Class c = Class.forName("org.wikibooks.fr.Exemple");

Object o = c.newInstance();  // équivaut à   new org.wikibooks.fr.Exemple();
```

Une classe comme celle ci-dessous peut ne pas avoir de constructeur sans paramètres :

```
package org.wikibooks.fr;

public class Livre
{
    String titre;
    int nb_pages;

    public Livre(String titre, int nb_pages)
    {
        this.titre = titre;
        this.nb_pages = nb_pages;
    }
}
```

Dans ce cas, il faut d'abord obtenir le constructeur, puis l'appeler :

```
Class c = Class.forName("org.wikibooks.fr.Livre"); // Accès à la classe Livre
Constructor constr = c.getConstructor(String.class, int.class); // Obtenir le constructeur (Str
Object o = constr.newInstance("Programmation Java", 120); // -> new Livre("Programmation Java",
```

Pour les versions de Java antérieures à 5.0 où l'auto-boxing n'existe pas, et où il faut explicitement utiliser des tableaux :

```
Class c = Class.forName("org.wikibooks.fr.Livre"); // Accès à la classe Livre
Constructor constr = c.getConstructor(new Class[]{ String.class, Integer.TYPE }); // Obtenir le
Object o = constr.newInstance(new Object{ "Programmation Java", Integer.valueOf(120) }); // ->
```

Appel à une méthode

L'appel à une méthode de la classe est basé sur le même principe que l'appel à un constructeur vu juste avant. Cependant, pour obtenir la référence à une méthode, il faut spécifier le nom. Lors de l'invocation de la méthode, il faut spécifier l'instance (l'objet) auquel s'applique la méthode (`null` pour une méthode statique).

Exemple :

```
package org.wikibooks.fr;

public class Livre
{
    String titre;
    int nb_pages;
```

```
    public Livre(String _titre, int _nb_pages) {
        this.titre = _titre;
        this.nb_pages = _nb_pages;
    }

    public int getNombreDeFeuilles(int pages_par_feuille)
    {
        return (nb_pages+pages_par_feuille-1)/pages_par_feuille;
    }
}
...
Class c = Class.forName("org.wikibooks.fr.Livre"); // Accès à la classe Livre

Constructor constr = c.getConstructor(String.class, int.class); // Obtenir le constructeur (Str
Object o = constr.newInstance("Programmation Java", 120); // -> new Livre("Programmation Java",

Method method = c.getMethod("getNombreDeFeuilles", int.class); // Obtenir la méthode getNombreD
int nb_feuilles = (int)method.invoke(o, 2); // -> o.getNombreDeFeuilles(2);
```

Accès à un attribut

L'accès à un attribut se fait en appelant les méthodes sur l'instance de `java.lang.reflect.Field` obtenu auprès de la classe.

Exemple :

```
package org.wikibooks.fr;

public class Livre
{
    public String titre;
    public int nb_pages;

    public Livre(String _titre, int _nb_pages) {
        this.titre = _titre;
        this.nb_pages = _nb_pages;
    }
}
...
Class c = Class.forName("org.wikibooks.fr.Livre"); // Accès à la classe Livre

Constructor constr = c.getConstructor(String.class, int.class); // Obtenir le constructeur (Str
Object o = constr.newInstance("Programmation Java", 120); // -> new Livre("Programmation Java",

Field f_titre = c.getField("titre"); // Obtenir l'attribut titre
String titre_du_livre = (String)f_titre.get(o); // -> o.titre
f_titre.set(o, "Java"); // -> o.titre = "Java";
```

Exemple concret : un gestionnaire d'extensions

Chaque extension d'une classe lui ajoute des méthodes.

Cette section est vide, pas assez détaillée ou incomplète.

Regex

Syntaxe

Expressions rationnelles courantes

Caractère	Type	Explication
.	Point	n'importe quel caractère
[...]	classe de caractères	tous les caractères énumérés dans la classe
[^...]	classe complémentée	Tous les caractères sauf ceux énumérés
^	circonflexe	marque le début de la chaine, la ligne...
$	dollar	marque la fin d'une chaine, ligne...
|	barre verticale	alternative - ou reconnaît l'un ou l'autre
(...)	parenthèse	utilisée pour limiter la portée d'un masque ou de l'alternative
*	astérisque	0, 1 ou plusieurs occurrences
+	le plus	1 ou plusieurs occurrence
?	interrogation	0 ou 1 occurrence

Les expressions rationnelles en Java nécessitent le package *java.util.regex*.

Elles proposent les mêmes constructeurs spéciaux que le C++, soit X une expression :

- *(?:X)* : groupe non capturant
- *(?>X)* : groupe non capturant indépendant
- *(?=X)* : avec lookahead positif
- *(?!X)* : avec lookahead négatif
- *(?<=X)* : avec lookbehind positif
- *(?<!X)* : avec lookbehind négatif

Recherche

La classe *Pattern* offre la fonction *matches* qui renvoie si une expression est trouvée dans une chaine.

Le mot précédant un autre :

```
import java.util.regex.Pattern;
public class Regex {
    public static void main(String[] args) {
        String chaine1 = "Test regex Java pour Wikibooks francophone.";
        System.out.println(Pattern.matches("[a-z]* Wikibooks",chaine1));
    }
}
// Affiche : "pour Wikibooks"
```

Remplacement

La classe *Matcher* permet de trouver tous les résultats des expressions (fonction *find*) et de les stocker dans un groupe de résultat.

Afficher ce qui est entre les balises HTML :

```
import java.util.regex.Pattern;
```

```
import java.util.regex.Matcher;

public class Regex {
    public static void main(String[] args) {
        String chaine = "Test regex Java pour <balise1>Wikibooks</balise1> francophone.";
        Pattern p = Pattern.compile("<.*>(.*)<.*>");
        Matcher m = p.matcher(chaine);
        while(m.find())
            System.out.println(m.group(1));
    }
}
// Affiche "Wikibooks"
```

Références

- http://docs.oracle.com/javase/1.4.2/docs/api/java/util/regex/Pattern.html

Conclusion

Conclusion

Vous avez appris, au travers de la lecture de ce livre, les fondements du langage Java. La maîtrise de ces notions est indispensable pour produire des applications ou des bibliothèques convenables. Néanmoins, si vous voulez pleinement profiter des nombreuses autres possibilités offertes par Java, il faut dès maintenant se pencher sur les nombreuses facettes de Java.

Elles sont notamment présentées dans le wikilivre « Développer en Java » qui propose de découvrir de nombreux outils, bibliothèques, techniques et pratiques spécifiques à Java.

JDK

Le JDK est un ensemble d'outil permettant de développer en Java.

Pour obtenir la liste des options d'un outil, il suffit de lancer l'outil sans aucun argument.

Compiler le code

javac est le compilateur qui convertit le code source .java en fichier .class (contenant le bytecode Java).

Supposons que vous avez

- un dossier « src » qui contient vos sources (tous vos fichiers *.java*).
- un dossier « bin » où vous placerez tous les fichiers compilés (les fichier *.class* correspondant)

```
# compile seulement la classe Exemple et place le résultat dans bin
javac -d bin src/Exemple.java

# compile toutes les sources trouvées dans src et les place dans bin
javac -d bin src/**/*.java
```

Si la compilation échoue parce que votre code utilise des classes que javac ne connait pas (erreur *Unable to find symbol*), vous devez préciser à javac un classpath comme expliqué plus loin.

Lancer l'application

`java` permet une dans lancer une application java en ligne de commande, il faut passer en paramètre le nom complet (*pleinement qualifié*) de la classe.

```
java org.wikibooks.fr.Exemple argument_1 argument_2
```

La ligne de commande ci-dessus appel la méthode public static void main(String[] args) de la classe Exemple du package org.wikibooks.fr avec args, un tableau à deux éléments : "argument_1" et argument_2".

javaw permet une application java sans console (interface graphique seule).

```
javaw org.wikibooks.fr.Exemple
```

Une archive java (*.jar) avec l'option `-jar`, un petit fichier (appelé « Manifest ») contenu dans le jar indique lui-même le nom de la classe principale à lancer.

```
java -jar chemin/vers/le/fichier.jar
```

Préciser le CLASSPATH

Dans toutes les commandes ci-dessus, il faut permettre à java de trouver tous les fichiers compilés nécessaire à l'exécution du code (les fichiers `*.class` générés avec `javac`). Pour cela, il faut préciser à `java` les répertoires ou celui-ci pourra trouver tout ça.

Pour cela, il faut définir ce qu'on appelle le « CLASS PATH », c'est une simple chaine qui définit plusieurs chemins pour trouver des .class séparés par ":". Les chemins peuvent être des chemins vers des fichiers .jar ou vers des répertoires contenant des fichiers `.class`. Il y a deux façon de préciser le CLASSPATH à java, la première est d'utiliser le paramètre -classpath de ligne de commande.

```
java -classpath chemin/vers/une/premiere/bibliotheque.jar:chemin/vers_une_autre/bibliotheque.ja
```

Ici, java essaiera de trouver le fichier Exemple.class dans les deux jars donnés puis dans le dossier bin. Si le fichier n'est trouvé dans aucun de ces éléments, il y aura une erreur (*Class not found exception*).

La seconde façon de préciser le CLASSPATH est de définir une variable d'environnement système. Sous linux,

```
export CLASSPATH="chemin/vers/une/premiere/bibliotheque.jar:chemin/vers_une_autre/bibliotheque.
```

Attention à ne pas confondre `java -jar fichier.jar` et `java -cp fichier.jar`, la première commande permet de lancer le programme qui se trouve dans *fichier.jar* et fonctionne ; la seconde précise que des classes peuvent être chargée depuis *fichier.jar* mais oublie de préciser quelle classe il faut lancer : java indiquera qu'un paramètre est manquant. Enfin, sachez que vous pouvez remplacer « -classpath » par « -cp ».

jar

L'outil `jar` permet de regrouper les classes (fichiers `*.class`) et ressources d'une application en une seule archive exécutable. Cette archive est au format ZIP mais possède l'extension `.jar`. Dans cette archive, un répertoire spécial situé à la racine nommé `META-INF` contient des fichiers d'information et de configuration pour la machine virtuelle Java, dont notamment le fichier `MANIFEST.MF` (fichier manifest, *MF* = *Meta File*).

Une archive JAR peut être exécutable si la classe principale (contenant une méthode statique nommée `main`) de l'application est spécifiée dans ce fichier `MANIFEST.MF`. Dans ce cas, l'application peut se lancer avec la commande suivante :

```
java -jar chemin_de_l_archive.jar ...arguments passés à l'application si besoin...
```

Ce fichier manifest est au format texte. Exemple :

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.1
Created-By: 19.1-b02 (Sun Microsystems Inc.)
Main-Class: org.wikibooks.fr.ApplicationExemple
Class-Path: .
```

Il possède différents champs. Chaque champ possède un nom, suivi de deux-points et de sa valeur. Si une valeur est longue, elle peut être répartie sur plusieurs lignes, chaque ligne additionnelle commençant alors par au moins un caractère espace.

Pour plus de détails sur les fichiers manifest voir http://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html

Attention !

Le champ `Class-Path` dans une archive JAR diffère du paramètre classpath passé aux outils Java :

- Il ne peut contenir que des chemins relatifs à d'autres archives Java (*.jar) ; les répertoires ne sont pas supportés ;
- Le séparateur est un caractère espace. Les fichiers référencés ne peuvent donc en contenir dans leur nom.

Pour créer une archive JAR à partir d'un répertoire contenant les classes et ressources (les sous-répertoires devant correspondre aux packages), et d'un fichier texte pour le fichier manifest :

```
jar cfm mon_archive.jar mon_manifest.txt -C repertoire .
```

Peu importe le nom du fichier manifest et son extension spécifiés, il sera renommé `MANIFEST.MF`.

Les arguments de la commande sont décrits ci-dessous :

cfm

Une série de caractères dont le premier spécifie l'action principale :

- **c** Créer une archive (**C**reate).
- **t** Afficher le contenu de l'archive (**T**able of content).
- **x** Extraire les fichiers de l'archive (e**X**tract files).
- **u** Mettre à jour l'archive existante (**U**pdate archive).

Les caractères suivants donne l'ordre des arguments qui suivent :

- **f** Le nom du fichier archive (archive **F**ile).
- **m** Le nom du fichier manifest à utiliser (**M**anifest file).

mon_archive.jar

Le nom du fichier archive.

mon_manifest.txt

Le nom du fichier manifest à utiliser.

-C *repertoire*

Précise le chemin du répertoire pour les chemins relatifs

.

Inclus le fichier spécifié (`.` désignant le répertoire courant sous la plupart des systèmes d'exploitation).

Eclipse possède une interface interactive pour générer un fichier JAR à partir des classes d'un projet Java, ou d'une configuration d'exécution.

javadoc

L'outil `javadoc` est le générateur de documentation, qui génère automatiquement de la documentation à partir des commentaires du code source.

Voir le chapitre sur les commentaires pour plus de détails sur l'outil et son utilisation.

jdb

Cette section est vide, pas assez détaillée ou incomplète.

le débogueur

javah

javah permet de générer un fichier d'en-tête C (*.h) contenant la déclaration des fonctions correspondantes aux méthodes natives de la classe compilée spécifiée.

```
javah -classpath directory-list classname
```

Voir Développer en Java/Faire appel à du code natif.

javap

javap permet de lister les membres et désassembler la classe compilée spécifiée.

```
javap -c -classpath directory-list classname
```

Voir Développer en Java/La machine virtuelle Java (JVM)#Le byte code.

Paramétrer la JVM

La machine virtuelle Java désigne l'environnement d'exécution d'un programme Java. Il est possible d'adapter cet environnement suivant les besoins : utilisation de la mémoire, configuration du réseau, etc.

Il existe 2 possibilités de paramétrage de cet environnement :

1. au moment du lancement du programme Java (et de la JVM)
2. au cours de l'exécution d'un programme Java

L'action de *paramétrer* la machine virtuelle représente le changement d'état d'un paramètre système. Dans un premier temps les paramètres les plus utilisés sont listés. Ensuite, les deux possibilités de paramétrage sont exposées.

Liste des paramètres de la machine virtuelle Java (JVM)

Paramètres du proxy

Nom du paramètre[1]	Description
http.proxyHost	le nom du serveur proxy
http.proxyPort	le port du proxy (valeur par défaut:80)
http.nonProxyHosts	liste des hôtes qui doivent être accédés directement, sans passer par le proxy. Il s'agît d'une liste d'expressions régulières séparées par '|'. Tous les hôtes correspondant à l'une de ces expressions seront accédés par une connexion directe sans passer par le proxy
http.proxyUser	le nom d'utilisateur utilisé pour le proxy
http.proxyPassword	le mot de passe utilisateur utilisé pour le proxy

Paramètres de la mémoire

Le paramétrage de la mémoire se fait au lancement d'un programme Java grâce aux arguments suivants :

Nom du paramètre	Description
-Xms	allocation mémoire initiale
-Xmx	allocation mémoire maximum
-Xmn	allocation mémoire pour les programmes fils (à confirmer?)

Exemple pour lancer un programme Java en allouant 256 Mo :

```
java -Xms256M -Xmx256M monProgrammeJava
```

Modification d'un paramètre de la JVM

Au lancement d'un programme Java, il est possible de positionner un ou plusieurs paramètres système de la manière suivante :

```
$ java -D<nom du paramètre1>=<valeur du paramètre1> -D<nom du paramètre2>=<valeur du paramètre2
```

Exemple pour positionner le proxy à utiliser

```
$ java -Dhttp.proxyHost=monproxy -Dhttp.proxyPort=3128 monProgrammeJava
```

dans cet exemple, le proxy utilisé sera l'hôte "monproxy" sur le port 3128

Au sein d'un programme Java, il est possible de changer l'état d'un paramètre système de la manière suivante :

```
System.getProperties().put("http.proxyHost", "monproxy");
System.getProperties().put("http.proxyPort", "3128");
System.getProperties().put("http.proxyUser", "toto");
System.getProperties().put("http.proxyPassword", "totoisback");
```

dans cet exemple, le proxy utilisé sera l'hôte "monproxy" sur le port 3128 en utilisant comme utilisateur "toto"

Références

1. (anglais) http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/Proxy.html

---

Cette page fait partie du livre Programmation

Liste des mots réservés

Voici la liste des mots réservés à la programmation en Java :

Mot réservé	Type	Description
`abstract`	Mot-clé	Déclaration d'une méthode ou d'une classe abstraite.
`assert`	Mot-clé	Assertion
`boolean`	Type de données	Valeur booléenne (vrai ou faux).
`break`	Mot-clé	Interrompre une boucle ou un choix multiple.
`byte`	Type de données	Entier signé de -128 à +127.
`case`	Mot-clé	Cas dans un choix multiple.
`catch`	Mot-clé	Capture d'un type d'exception.
`char`	Type de données	Caractères Unicode (UTF-16, donc sur 16 bits).
`class`	Mot-clé	Déclaration d'une classe.
`const`	Réservé[1]	Inutilisé actuellement.
`continue`	Mot-clé	Continuer une boucle en allant à l'itération suivante.
`default`	Mot-clé	Cas par défaut dans un choix multiple.
`do`	Mot-clé	Boucle itérative.
`double`	Type de données	Nombre à virgule flottante, double précision.
`else`	Mot-clé	Exécution conditionnelle.
`enum`	Mot-clé	Déclaration d'une énumération.
`extends`	Mot-clé	Héritage : déclaration de la classe mère, ou pour une interface de toutes les interfaces mères.
`false`	Valeur littérale	Valeur booléenne fausse.
`final`	Mot-clé	Déclarer un membre comme final.
`finally`	Mot-clé	Code exécuté quoi qu'il se passe dans un bloc de capture d'exception.
`float`	Type de données	Nombre à virgule flottante, simple précision.
`for`	Mot-clé	Boucle itérative.
`goto`	Réservé[1]	Inutilisé actuellement.
`if`	Mot-clé	Exécution conditionnelle.
`implements`	Mot-clé	Déclaration des interfaces implémentées par une classe.
`import`	Mot-clé	Déclaration des packages utilisés par une classe.
`instanceof`	Mot-clé	Tester si un objet est de la classe indiquée (voir Transtypage).
`int`	Type de données	Entier signé de -2 147 483 648 à +2 147 483 647.
`interface`	Mot-clé	Déclaration d'une interface.
`long`	Type de données	Entier signé de -9 223 372 036 854 775 808 à +9 223 372 036 854 775 807.
`native`	Mot-clé	Déclaration d'une méthode native.
`new`	Mot-clé	Allocation d'une instance de classe.

Mot réservé	Type	Description
`null`	Valeur littérale	Référence nulle.
`package`	Mot-clé	Déclaration du package de la classe.
`private`	Mot-clé	Déclaration d'un membre privé de la classe.
`protected`	Mot-clé	Déclaration d'un membre protégé de la classe.
`public`	Mot-clé	Déclaration d'un membre public de la classe.
`return`	Mot-clé	Retourner une valeur depuis une méthode.
`short`	Type de données	Entier signé de -32 768 à +32 767.
`static`	Mot-clé	Déclaration d'un membre statique de la classe.
`strictfp`	Mot-clé	Déclaration d'une méthode ou classe où les opérations en virgule flottante doivent être évalué strictement de gauche à droite selon la spécification Java.
`super`	Mot-clé	Référence à l'instance de la classe mère.
`switch`	Mot-clé	Début d'un choix multiple.
`synchronized`	Mot-clé	Voir Processus légers et synchronisation.
`this`	Mot-clé	Référence à l'instance de la classe englobante.
`throw`	Mot-clé	Lever une exception
`throws`	Mot-clé	Déclaration des exception levées par une méthode.
`transient`	Mot-clé	Déclaration d'un attribut à exclure de la sérialisation.
`true`	Valeur littérale	Valeur booléenne vraie.
`try`	Mot-clé	Capture d'un type d'exception.
`void`	Mot-clé	Déclaration d'une méthode ne retournant aucune valeur.
`volatile`	Mot-clé	Déclaration d'un attribut volatile, c'est à dire dont la valeur ne doit pas être mise en cache car elle est accédée par différents threads.
`while`	Mot-clé	Boucle itérative.

Suffixes

L

L est un suffixe pour déclarer une valeur litérale de type `long` au lieu de `int` (voir syntaxe des valeurs de type long).

F

Idem pour le suffixe F pour déclarer une valeur litérale de type `float` au lieu de `double` (voir syntaxe des valeurs de type float).

D

Idem pour le suffixe D pour déclarer une valeur litérale de type `double`. Cependant, le type par défaut des nombres à virgules étant `double`, ce suffixe n'est pas obligatoire (voir syntaxe des valeurs de type double).

Préfixes

0

0 est un préfixe utilisable pour une valeur entière exprimée en octal (base 8, chiffres de 0 à 7).

0x

0x est un préfixe utilisable pour une valeur entière exprimée en hexadécimal (base 16, chiffres de 0 à 9 et les

lettres de A à F).

Notes et références

1. Ces mots réservés ne sont pas utilisés, et ce, jusqu'à la version 7 de Java. Cependant, ils ne peuvent être utilisé comme nom de variable par exemple. Oracle se garde le droit de les utiliser plus tard.

Débogage

Erreurs à la compilation

Fenêtre de warning ...java uses unchecked or unsafe operations

Une déclaration ne tient pas compte des surcharges possibles, une conversion de variable pose problème. Il faut donc en prévoir les exceptions. Par exemple pour manipuler des entiers :

```
try {
  i = i/2;
} catch (NumberFormatException e) {
  return 0;
}
```

<identifier> expected

L'ordre des déclarations n'est pas chronologique.

annotation type not applicable to this kind of declaration

Voir @Override.

cannot be dereferenced

- Il suffit de retirer la conversion, le *.toString()* après la variable de type *int* sélectionnée.
- Ou en ajouter unecomme : *(float), Float.valueOf(), ou .floatValue()*.

cannot find symbol - *variable*

Soit :

- La variable mentionnée n'a pas été déclarée ;
- Sa déclaration trouve dans une condition (`if` ou `try`) fermée avant la ligne en erreur ;
- Elle se trouve dans un package non importé ;
- Si elle se trouve dans un autre fichier, les compiler depuis leur répertoire parent (qui doit être le même) :

```
javac Projet1/Classe1.java
javac Projet1/Classe2.java
```

cannot instantiate from arguments because actual and formal argument lists differ in length

Cela peut survenir suite à un Collections.sort(liste), alors même que le Collections.shuffle(liste) fonctionne. Il faut donc utiliser import `java.util.Comparator;` dans le trie :

```
import java.util.Collections;
import java.util.Comparator;
...
private List<T> list;
private Comparator<T> comparator = null;
if(this.comparator!= null) {
       Collections.sort(list, comparator);
    }
```

constructor *variable* in class *classe* cannot be applied to given types

Cette section est vide, pas assez détaillée ou incomplète.

exception IOException is never thrown in body of corresponding try statement

Un `try` inutile a été détecté.

impossible de trouver la classe principale

Lancer la commande "java" vers un .class (et non un .java).

incompatible types: possible lossy conversion from float to int

Problème de conversion.

Cette section est vide, pas assez détaillée ou incomplète.

java.lang.NumberFormatException

- Un nombre décimal contient un symbole qui n'est pas un chiffre (ex : €) ou bien une virgule à la placed'un point.

local variable *variable* is accessed from within inner class; needs to be declared final

Cette section est vide, pas assez détaillée ou incomplète.

no suitable method found for - *méthode*

La méthode mentionnée n'a pas été déclarée, ou du moins pas avec ce type de paramètre (ex : remplacer .addAll() par .add()).

no suitable constructor found for Integer(double)

Convertir plus explicitement :

```
MonEntier = ((Integer)MonDouble).intValue();
```

no suitable constructor for *classe* is abstract cannot be instanciated

S'il est possible de retirer le mot `abstract` de la déclaration de la classe, le faire.

non-static method/variable ... cannot be referenced from a static context

Cela peut se produire en invoquant une méthode ou une variable dans sa classe. Par exemple ci-dessous il fallait ajouter ou remplacer le "this." par "c." :

```
MaClasse c = New MaClasse();
c.MaMethode();
```

not a statement

Il manque sûrement les parenthèses après le nom d'une méthode.

reference to assertequals is ambiguous

Il convient d'assurer un typage plus fort des paramètres de `assertequals()`, par exemple avec `.intValue()`.

unclosed character literal

Problème d'encodage, remplacer les apostrophes par des guillemets.

unreachable statement

Une partie de code ne sera jamais exécutée, parce qu'elle est juste après un `break` ou un `return`.

unreported exception ...; must be caught or declared to be thrown

Une exception personnalisée est lancée (`throw new NomDelException("Message");`) sans que sa méthode soit déclarée avec `throws NomDelException`.

variable is already defined

Si la variable est redéfinie dans une condition différente, il faut les séparer avec des { }, ex :

```
switch (args[0].toString()) {
  case "1": {
    a = 1;
    break;
  }
  case "2": {
    a = 2;
    break;
  }
}
```

Erreurs à l'exécution

Si deux variables de type *String* identiques ne peuvent pas être comparées

Les forcer avec `.toString()`.

NullPointerException

Soulevée si l'on applique une méthode sur un objet *null*. On peut donc changer l'appel ou lever l'exception :

```
    try {
        ...
    } catch (NullPointerException npe) {
        npe.printStackTrace();
    }
```

aucun attribut manifest principal dans ...jar

Il n'y a pas de méthode main() dans le .jar, ce fichier ne peut donc pas être exécuté directement, mais uniquement décompressé (avec le freeware 7zip ou la commande `jar xvf`).

java.lang.ArithmeticException: / by zero

Une division par zéro s'évite généralement par un `if` sur le dénominateur, ou un `throw` d'une exception.

Class names, *classe*, are only accepted if annotation processing is explicitly requested

Les classes définies à la compilation (par exemple avec `javac -classpath`) sont introuvables.

Stream closed

Retirer le `.close()` de la boucle du `.read()` ou du `.write()`, même s'il est situé après. Si cela ne suffit pas, le retirer de la méthode.

Unable to load native library: Can't load IA 32-bit .dll on a AMD 64-bit platform

Réinstaller JRE (https://www.java.com/fr/download/faq/java_win64bit.xml) [archive] en changeant de version.

Voir aussi

Recommandations d'Oracle : http://search.oracle.com/search/search?start=1&search_p_main_operator=all&q=convention

Autres langages

Cette section sera utile aux personnes qui apprennent Java mais connaissant un ou plusieurs autres langages. Les confusions courantes sont évoquées et les équivalences listées.

C++

- En java on utilise `final` et non `const`. Le mot `const` étant tout de même réservé pour un usage ultérieur.
- Java ne définit pas d'espace de nom (*namespace*), mais des paquetages (`package`).
- On ne libère pas la mémoire. Cela est géré automatiquement par le ramasse-miette. À la place, il est recommandé de libérer les références aux objets qui ne sont plus utilisés en assignant `null` à la référence, dès que possible ou juste avant une potentielle allocation d'une grande quantité d'objets ou de grands tableaux.
- Il n'y a pas de pointeurs mais des références :
  - Les opérateurs `->` et `*` (déréférencement) n'existent pas. Il faut utiliser le point (`.`).
  - Les références (contrairement au C++) peuvent être modifiées pour référencer un autre objet en utilisant l'opérateur d'affectation (`=`).
- un `char` fait un octet en C++ mais 2 octets en Java car l'encodage Unicode utilisé est l'UTF-16.
- `unsigned` n'existe pas en Java : tous les types entiers sont signés.
- L'héritage multiple possible en C++ est interdit en Java, il faut utiliser des interfaces.
- L'opérateur de résolution de portée « `::` » n'existe pas en Java :
  - Pour utiliser un membre d'un package ou un membre statique d'une classe, utiliser le point (`.`).
  - Pour appeler une méthode telle qu'elle est définie dans la classe parente, utiliser la référence `super`.
- Les classes ou méthodes ne peuvent être déclarées *virtual* car elles le sont toutes : toutes les liaisons sont dynamiques.
- Les opérateurs ne sont pas redéfinissable en Java. Les opérateurs n'agissent pas sur les objets, excepté pour l'opérateur `+` permettant de concaténer deux chaînes de caractères.

PHP

- L'équivalent de PDO (PHP Data Objects) est JDBC (Java DataBase Connectivity).
- *Import* est remplacé par *require_once*.

Voir aussi

- Translinguisme/Programmation

Premier programme

Premier programme

Le fichier source

Ce programme doit être écrit dans le fichier **Exemple.java**.

```
public class Exemple {
	public static void main(String[] args) {
		System.out.println("Hello world!");
	}
}
```

Explications sur le langage

Ce programme est le classique Hello world. Comme son nom l'indique, ce programme va afficher la phrase "Hello world" à l'écran. Analysons-le ligne par ligne :

```
public class Exemple {
```

Cette ligne déclare une classe publique que nous appelons *Exemple*.

Un fichier `.java` ne peut contenir qu'une seule classe publique et le fichier doit porter le nom de cette classe. Ainsi, le fichier de ce premier exemple s'appellera obligatoirement `Exemple.java` (en respectant la classe).

Ce système de nommage permet au compilateur et à l'interpréteur de trouver les fichiers correspondant à une classe.

```
public static void main(String[] args) {
```

Cette ligne déclare une méthode appelée *`main`*. Cette méthode est le point d'entrée du programme (la méthode appelée lorsqu'il sera exécuté).

Elle prend en argument un tableau de chaînes de caractères (`String[] args`) et ne retourne rien (`void`).

Cette méthode est publique et statique, ce qui sera expliqué plus loin.

```
System.out.println("Hello world!");
```

Cette dernière instruction invoque la méthode `println` de l'objet `out` se trouvant dans la classe `System` en lui passant en argument la chaîne `Hello world!`. L'exécution de cette méthode aura pour résultat l'affichage de `Hello world!`.

Cette ligne peut sembler obscure pour l'instant. Les détails seront abordés par la suite.

Compilation du fichier source

Nous allons présenter la compilation de notre programme en utilisant le compilateur gratuit Java très répandu nommé **Javac**, disponible gratuitement auprès d'Oracle. Cependant, il faut savoir qu'il existe aussi de nombreux environnements de développement **Java** permettant de taper, compiler, exécuter ou débuguer des programmes dans ce langage.

Tapez le programme précédent et sauvegardez le dans un fichier *Exemple.java* (pour la raison expliquée précédemment) et tapez dans une fenêtre :

Invite de commande DOS	Terminal Unix

`> javac Exemple.java`
`> dir`
`Exemple.class`
`Exemple.java`
`>`
	`$ javac Exemple.java`
	`$ ls`
	`Exemple.class`
	`Exemple.java`
	`$`

Le compilateur Javac a produit le fichier *Exemple.class*, contenant le code intermédiaire. Ce fichier n'est normalement pas éditable[1], ce qui peut garantir le copyright.

En cas de problèmes...

Voici les points à vérifier selon le type de problème :

Le système indique que le compilateur est introuvable

1. Vérifiez que vous avez installé un kit de développement Java (le JDK) et pas simplement un environnement d'exécution (JRE) qui ne comporte pas les outils de compilation.
2. Vérifiez que le chemin du répertoire *bin* contenant le compilateur *javac* (*javac.exe* sous Windows) est dans la liste de la variable d'environnement `PATH`.

Le compilateur se lance mais affiche une erreur de classe non trouvée

1. Si la classe ne déclare aucun paquetage (*package*), vérifiez que vous lancez la commande depuis le répertoire où se trouve le fichier source (**.java*). Changez de répertoire si nécessaire avant de recommencer.
2. Sinon, vous devez lancer la commande depuis le répertoire parent du paquetage racine, en donnant le chemin relatif du fichier source depuis ce répertoire parent.
3. Dans les deux cas ci-dessus, en plus de changer de répertoire courant, il peut être nécessaire d'ajouter le chemin de ce répertoire dans le *classpath*. Cela peut être fait soit dans la ligne de commande avec l'option `-classpath` (ou `-cp`), soit dans la variable d'environnement `CLASS_PATH`.

Le compilateur se lance mais affiche une erreur de syntaxe

1. Vérifiez le contenu du fichier source. Pour compiler les exemples de ce livre, le mieux est de faire un copier-coller **complet** du code.
2. Assurez-vous de compiler le bon fichier source et pas un autre.

Exécution du programme

Java est une machine virtuelle java fournie par Oracle et disponible pour de nombreux environnements.

Pour exécuter notre code intermédiaire, il faut taper :

```
java Exemple
```

L'exécution du programme affiche dans une fenêtre console la fameuse phrase *Hello world!*.

En cas de problèmes...

Voici les points à vérifier selon le type de problème :

Le système indique que la commande java est introuvable

1. Vérifiez que vous avez installé un kit de développement Java (le JDK).
2. Vérifiez que le chemin du répertoire *bin* contenant l'interpréteur *java* (*java.exe* sous Windows) est dans la liste de la variable d'environnement `PATH`.

L'interpréteur se lance mais affiche une erreur de classe non trouvée

1. Vérifiez que vous avez compilé le fichier source **.java* (voir section précédente) sans erreur, c'est-à-dire que vous avez obtenu un fichier compilé **.class*.
2. Si la classe ne déclare aucun paquetage (*package*), vérifiez que vous lancez la commande depuis le répertoire où se trouve le fichier compilé (**.class*). Changez de répertoire si nécessaire avant de recommencer.

3. Sinon, vous devez lancer la commande depuis le répertoire parent du paquetage racine, en donnant le nom complet de la classe (incluant le nom du paquetage).
4. Dans les deux cas ci-dessus, en plus de changer de répertoire courant, il peut être nécessaire d'ajouter le chemin de ce répertoire dans le *classpath*. Cela peut être fait soit dans la ligne de commande avec l'option `-classpath` (ou `-cp`), soit dans la variable d'environnement `CLASS_PATH`.

L'interpréteur se lance mais rien ne s'affiche, ou le comportement n'est pas le même

1. Vérifiez le contenu du fichier source. Pour compiler les exemples de ce livre, le mieux est de faire un copier-coller **complet** du code.
2. Assurez-vous de lancer la bonne classe principale et pas une autre.

Voir aussi

- (anglais) http://docs.oracle.com/javase/tutorial/getStarted/index.html

Références

1. http://www.jmdoudoux.fr/java/dej/chap-decompil.htm

Cette page fait partie du livre Programmation

Générer un carré magique

Ce programme permet de générer des carrés magiques d'ordre impair.

```
import java.io.*;
/*
Nom : Carre.java
Rôle : Construction d'un carré magique d'ordre impair.
Compilation : javac Carre.java
Exécution : java Carre.java <ordre>
    avec <ordre> : entier impair > 1
Résultat : dans un fichier texte nommé carre_ordre.txt
*/
public class Carre
{
    public static void main(String args[])
    throws Exception
    {
        int ordre, ligne, colonne;
        int maxValue = (int)Math.sqrt((double)Integer.MAX_VALUE);

        System.out.println("Début du programme.");

        // test du paramètre obligatoire : > 1 et impair
        if (args == null || args.length == 0)
        {
            System.out.println("Usage : java Carre <ordre>\n"+
                "\tavec <ordre> : entier > 1 et impair.");
            System.exit(1) ;
        }
        ordre = Integer.parseInt(args[0]);
        if (ordre <= 1 || (ordre%2) == 0 || ordre > maxValue )
        {
            System.out.println(ordre + " n'est pas impair ou n'est pas supérieur a 1" +
                            " ou est trop grand : > " + maxValue) ;
            System.exit(1) ;
        }
        System.out.println("Le paramètre " + ordre + " est correct : > 1 et impair");

        // Création du tableau
        int carre[][] = new int[ordre][ordre] ;

        System.out.println("Début du calcul");
        // Rangement 1er nombre n au milieu de la première ligne
        // Puis rangement des autres nombres.
        ligne = 0 ;
        colonne = ordre/2 ;
        for (int n=1; n<=(ordre*ordre); n++)
        {
            // Écriture dans le tableau dans la case calculée
            carre[ligne][colonne] = n ;

            // Détermination de la position de la prochaine case à écrire
            if ((n%ordre) == 0)
            {
                // Si débordement à gauche du tableau
                // Écriture dans la case sous le dernier nombre
                ligne++ ;
            }
            else
            {
                // Écriture dans la case en haut à gauche
                ligne = ((ligne == 0) ? ordre-1 : ligne-1);
                colonne = ((colonne == 0) ? ordre-1 : colonne-1);
            }
        } // for (int n=1; n<=ordre*ordre; n++)
        int sommeMagique = ordre * ( ordre * ordre + 1) / 2;

        // Création du fichier résultat
```

```
        String nomFic = "carre_" + ordre + ".txt";
        System.out.println("Fin du calcul, écriture du fichier "+ nomFic + "...");
        PrintWriter hFic = new PrintWriter(new BufferedWriter(new FileWriter(nomFic)));

        // Impression du tableau
        hFic.println("Carre magique d'ordre " + ordre);
        for (ligne=0; ligne<ordre; ligne++)
        {
            for (colonne=0; colonne<ordre; colonne++)
            {
                hFic.print(carre[ligne][colonne] + " ");
            }
            hFic.println("") ;
        }
        hFic.println("La somme magique est " + sommeMagique);
        hFic.close();
        System.out.println("fin écriture, programme terminé.");
    } // public static void main(String args[])
} // public class Carre
```

Générer un triangle de Sierpiński

La méthode récursive `private void drawSierpinskiTriangle ( int[] x, int[] y, int d )` accepte des coordonnées de trois points (coins du triangle principal) plus ou moins arbitraires.

- Pour Compiler (vous devez disposez du JDK Java) : `javac SierpinskiTriangle.java`
- Pour exécuter :
  - Avec un éditeur de texte, créez le fichier *SierpinskiTriangle.html* dans le même répertoire que *SierpinskiTriangle.class* créé lors de la compilation. Copiez-y le code html ci-dessous.
  - Ouvrez le fichier *SierpinskiTriangle.html* avec un navigateur comme Firefox.

Source de *SierpinskiTriangle.html* :

```
<!DOCTYPE html PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
<title> SierpinskiTriangle </title>
<body>
<applet code="SierpinskiTriangle.class" width=1024 height=1024>
Java n'est pas installé !
</applet>
</body>
</html>
```

Source de *SierpinskiTriangle.java* :

```
/* Nom du fichier : SierpinskiTriangle.java */
import java.awt.*;
import java.applet.*;

public class SierpinskiTriangle extends Applet {
    private Graphics g;
    private int dMin=8;    // limite à la récursion en pixels

    public void paint(Graphics g) {
        this.g = g;
        int d  = 1024;    // base (largeur du triangle)
        int x0 =   50;    // distance de gauche
        int y0 =   50;    // distance du haut
        int h  = (int)(d*Math.sqrt(3)/2);    // hauteur
            // adapté à un triangle équilatéral

    //  spécification du triangle principal: points A, B, C
        int xA=x0,      yA=y0+h;    // (bas-gauche)
        int xB=x0+d,    yB=y0+h;    // (bas-droite)
    //  int xB=x0,      yB=y0;      // (haut-gauche)
    //  int xB=x0+d,    yB=y0;      // (haut-droite)
        int xC=x0+d/2,  yC=y0;    // triangle équilatéral (haut-milieu)
    //  int xC=x0,      yC=y0;    // (haut-gauche)
            // triangle perpendiculaire, angle droit près A
    //  int xC=x0+d,    yC=y0;    // (haut-droite)
            // triangle perpendiculaire, angle droit près B
        int[] x = { xA, xB, xC };
        int[] y = { yA, yB, yC };

        drawSierpinskiTriangle( x, y, d/2 );     // démarrer la récursion
    }

    private void drawSierpinskiTriangle ( int[] x, int[] y, int d ) {
        if (d<dMin) g.fillPolygon ( x, y, 3 );  // fond de la récursion
        else {
            // milieus des cotés du triangle:
            int xMc = (x[0]+x[1])/2,    yMc = (y[0]+y[1])/2;
            int xMb = (x[0]+x[2])/2,    yMb = (y[0]+y[2])/2;
            int xMa = (x[1]+x[2])/2,    yMa = (y[1]+y[2])/2;
```

```
            int[] xNouveau1 = { x[0], xMc, xMb };
            int[] yNouveau1 = { y[0], yMc, yMb };
            drawSierpinskiTriangle ( xNouveau1, yNouveau1, d/2 );    // récursion

            int[] xNouveau2 = { x[1], xMc, xMa };
            int[] yNouveau2 = { y[1], yMc, yMa };
            drawSierpinskiTriangle ( xNouveau2, yNouveau2, d/2 );    // récursion

            int[] xNouveau3 = { x[2], xMb, xMa };
            int[] yNouveau3 = { y[2], yMb, yMa };
            drawSierpinskiTriangle ( xNouveau3, yNouveau3, d/2 );    // récursion
        }
    }
}
```

XML

Il existe dans Java deux packages permettant de manipuler les fichiers XML : la manière SAX (Simple API for XML, navigation via un curseur) et DOM (Document Object Model, navigation via un arbre).

Soit le fichier XML suivant :

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Documentation>
    <Site Wiki="Wikibooks">
        <Livre Wikilivre="Java">
            <Page1>Introduction</Page1>
            <Page2>Bases du langage</Page2>
            <Exemple>XML</Exemple>
        </Livre>
    </Site>
</Documentation>
```

Remarque : les espaces sont interdits dans les noms des balises.

La méthode DOM

Le programme suivant génère ce fichier XML dans le même répertoire :

```
import java.io.File;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.*;

public class XML_Creation_DOM {

    public static void main(String argv[]) {

      try {
        DocumentBuilderFactory XML_Fabrique_Constructeur = DocumentBuilderFactory.newInstance()
        DocumentBuilder XML_Constructeur = XML_Fabrique_Constructeur.newDocumentBuilder();

        Document XML_Document = XML_Constructeur.newDocument();
        Element documentation = XML_Document.createElement("Documentation");
        XML_Document.appendChild(documentation);

        Element site = XML_Document.createElement("Site");
        documentation.appendChild(site);
        Attr attribut1 = XML_Document.createAttribute("Wiki");
        attribut1.setValue("Wikibooks");
        site.setAttributeNode(attribut1);

        Element livre = XML_Document.createElement("Livre");
        site.appendChild(livre);
        Attr attribut2 = XML_Document.createAttribute("Wikilivre");
        attribut2.setValue("Java");
        livre.setAttributeNode(attribut2);

        Element page1 = XML_Document.createElement("Page1");
        page1.appendChild(XML_Document.createTextNode("Introduction"));
        livre.appendChild(page1);

        Element page2 = XML_Document.createElement("Page2");
        page2.appendChild(XML_Document.createTextNode("Bases du langage"));
        livre.appendChild(page2);

        Element example = XML_Document.createElement("Exemple");
```

```
        example.appendChild(XML_Document.createTextNode("XML"));
        livre.appendChild(example);

        TransformerFactory XML_Fabrique_Transformeur = TransformerFactory.newInstance();
        Transformer XML_Transformeur = XML_Fabrique_Transformeur.newTransformer();
        DOMSource source = new DOMSource(XML_Document);
        StreamResult resultat = new StreamResult(new File("XML_résultat.xml"));
        XML_Transformeur.transform(source, resultat);
        System.out.println("Le fichier XML a été généré !");
      } catch (ParserConfigurationException pce) {
        pce.printStackTrace();
      } catch (TransformerException tfe) {
        tfe.printStackTrace();
      }
    }
}
```

La méthode SAX

Pour lire le fichier ci-dessus, il faut le parser avec un handler dont on redéfinit les méthodes :

```
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import org.xml.sax.helpers.DefaultHandler;

public class XML_Creation_SAX {
 public static void main(String[] args) throws Exception {
  try {
   SAXParserFactory factory = SAXParserFactory.newInstance();
   SAXParser parser = factory.newSAXParser();
   parser.parse("XML_résultat.xml", new DefaultHandler() {
    public void startDocument() throws SAXException { System.out.println("startDocument"); }
    public void endDocument() throws SAXException { System.out.println("endDocument"); }
    public void startElement(String uri, String localName, String qName, Attributes attributes)
    public void endElement(String uri, String localName, String qName) throws SAXException { Sy
   });
  } catch (Exception e) { System.err.println(e); System.exit(1); }
 }
}
```

La méthode JDOM

JDOM utilise les deux bibliothèques précédentes. Si l'erreur suivante survient : *error: package org.jdom does not exist*, il suffit de télécharger la bibliothèque (https://jdom.org/downloads/index.html) [archive]. Pour utilise la version 2, remplacer les `import org.jdom` par des `import org.jdom2`.

Cette section est vide, pas assez détaillée ou incomplète.

Voir aussi

- Gestion de fichiers sur Wikiversité

swing/Hello

Interfaces graphiques avec swing

Un premier exemple

Le premier programme de tout informaticien.

Le fichier Hello.java

```
import java.awt.BorderLayout;
import java.awt.Container;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class Hello extends JFrame {

    // Constructeur
    public Hello() {
        super("Titre de la JFrame");
        // Si on appuie sur la croix, le programme s'arrete
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                // On récupère le conteneur de la JFrame (this est
                // une JFrame car Hello hérite de JFrame)
        Container contentPane = this.getContentPane();
        // Choix du gestionnaire de disposition
        BorderLayout layout = new BorderLayout();
        contentPane.setLayout(layout);

        JPanel panel = new JPanel();

        JLabel label = new JLabel(
            "Bonjour, ceci est une JFrame qui contient"+
                        " un JPanel qui contient un JLabel");
        panel.add(label);
        // Ici ne sert pas car le panel est seul
        contentPane.add(panel, BorderLayout.CENTER);
        this.pack();
        this.setVisible(true);
    }

    // Méthode principale : démarrage du programme
    public static void main(String[] args) {
        new Hello();
    }
}
```

swing/Boutons sur image

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;
import javax.swing.*;
import java.io.*;
import java.net.*;

public class FormulaireImage {

   public static void main(String[] args) {
        JFrame f = new JFrame("Formulaire");
        JLayeredPane l = new JLayeredPane();
        l.setPreferredSize(new Dimension(150,150));
        JPanel pi = new JPanel() {
            public void paint(Graphics g) {
                try {
                    URL ImageWB = new URL("http://upload.wikimedia.org/wikipedia/commons/6/64/W
                    BufferedImage image1 = ImageIO.read(ImageWB);
                    g.drawImage(image1, 0, 0, null);
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        };
        pi.setBounds(10,10,150,150);    // Fenêtre 150*150 placée à aux coordonnées 10,10
        l.add(pi,new Integer(0));   // Déposée sur la couche zéro

        JPanel pb = new JPanel();
        JButton b1 = new JButton("Bouton 1");
        b1.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                System.out.println("Clic bouton 1");
            }
        });
        pb.add(b1);
        JButton b2 = new JButton("Bouton 2");
        b2.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                System.out.println("Clic bouton 2");
            }
        });
        pb.add(b2);
        pb.setBounds(30,30,100,60);
        l.add(pb,new Integer(1));
        f.add(l);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.pack();
        f.setVisible(true); // Ou show(); deprecated
   }
}
```

swing/Somme

Interfaces graphiques avec swing

Calculer le somme de 2 entiers

Le fichier Somme.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;

public class Somme extends JFrame implements ActionListener
{

    private JLabel textX,textY,textS;
    private JTextField editX,editY,editS;
    private JButton buCompute;
    double x,y,s;

    public static void main(String [] args)
    {
    Somme a=new Somme();
    }

    Somme()
    {
        x=0;
        y=0;
        setBounds(10,20,400,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        Container c;
        c=getContentPane();

        BorderLayout bl=new BorderLayout();
        c.setLayout(bl);


        JPanel p=new JPanel();
        p.setLayout(null);

        textX= new JLabel("Valeur de X : ");
        textX.setBounds(10,20,100,20);
        p.add(textX);

        textY= new JLabel("Valeur de Y : ");
        textY.setBounds(10,50,100,20);
        p.add(textY);

        textS= new JLabel("X+Y vaut : ");
        textS.setBounds(10,80,100,20);
        p.add(textS);

        editX=new JTextField(20);
        editX.setBounds(120,20,100,20);
        p.add(editX);

        editY=new JTextField(20);
        editY.setBounds(120,50,100,20);
        p.add(editY);

        editS=new JTextField(20);
        editS.setBounds(120,80,100,20);
        editS.setEditable(false);
        p.add(editS);
```

```
        Border bord=BorderFactory.createRaisedBevelBorder();
        buCompute=new JButton("CALCULER");
        buCompute.setActionCommand("CALCULER");
        buCompute.addActionListener(this);
        buCompute.setBorder(bord);
        buCompute.setBounds(240,50,100,20);

        p.add(buCompute);

        c.add(p, BorderLayout.CENTER);
        setVisible(true);
    }


    public void actionPerformed(ActionEvent e)
    {
    if(e.getActionCommand().equals("CALCULER"))
            {
        double vx,vy,d1;
        try{
             vx=Double.parseDouble(editX.getText());
            }catch(RuntimeException ex)
            {vx=x;editX.setText(String.valueOf(x));}
        try{
             vy=Double.parseDouble(editY.getText());
            }catch(RuntimeException ex)
            {vy=y;editY.setText(String.valueOf(y));}
            x=vx;
            y=vy;
            s=x+y;
            editS.setText(String.valueOf(s));
            }
    }
}
```

swing/Distance

Interfaces graphiques avec swing

Calculer la distance entre 2 points

Le fichier Distance.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;

public class Distance extends JFrame implements ActionListener
{

    private JLabel textXA,textYA,textXB,textYB,textDistance;
    private JTextField editXA,editXB,editYA,editYB,editDistance;
    private JButton editA,editB;
    private Point A,B;

    public static void main(String [] args)
    {
    Distance app=new Distance();
    }

    Distance()
    {
        setBounds(10,10,400,300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        Container c;
        c=getContentPane();

        BorderLayout bl=new BorderLayout();
        c.setLayout(bl);


        JPanel p=new JPanel();
        p.setLayout(null);

        textXA= new JLabel("Abscisse de A : ");
        textXA.setBounds(10,20,100,20);
        p.add(textXA);

        textYA= new JLabel("Ordonnée de A : ");
        textYA.setBounds(10,50,100,20);
        p.add(textYA);

        textXB= new JLabel("Abscisse de B : ");
        textXB.setBounds(10,90,100,20);
        p.add(textXB);

        textYB= new JLabel("Ordonnée de B : ");
        textYB.setBounds(10,120,100,20);
        p.add(textYB);

        textDistance= new JLabel("La distance AB vaut : ");
        textDistance.setBounds(10,170,150,20);
        p.add(textDistance);

        editXA=new JTextField(20);
        editXA.setBounds(120,20,100,20);
        editXA.setEditable(false);
        p.add(editXA);

        editYA=new JTextField(20);
```

```
        editYA.setBounds(120,50,100,20);
        editYA.setEditable(false);
        p.add(editYA);


        editXB=new JTextField(20);
        editXB.setBounds(120,90,100,20);
        editXB.setEditable(false);
        p.add(editXB);

        editYB=new JTextField(10);
        editYB.setBounds(120,120,100,20);
        editYB.setEditable(false);
        p.add(editYB);

        editDistance=new JTextField(20);
        editDistance.setBounds(150,170,150,20);
        editDistance.setEditable(false);
        p.add(editDistance);

        Border bord=BorderFactory.createRaisedBevelBorder();
        editA=new JButton("MODIFIER A");
        editA.setActionCommand("EDITA");
        editA.addActionListener(this);
        editA.setBorder(bord);
        editA.setBounds(240,35,100,20);
        p.add(editA);

        editB=new JButton("MODIFIER B");
        editB.setActionCommand("EDITB");
        editB.addActionListener(this);
        editB.setBorder(bord);
        editB.setBounds(240,105,100,20);
        p.add(editB);



        c.add(p, BorderLayout.CENTER);
        setVisible(true);

        A=new Point();
        B=new Point();
        ShowPoint();
    }


    void ShowPoint()
    {
    editXA.setText(String.valueOf(A.getX()));
    editYA.setText(String.valueOf(A.getY()));
    editXB.setText(String.valueOf(B.getX()));
    editYB.setText(String.valueOf(B.getY()));
    editDistance.setText(String.valueOf(A.distance(B)));
    }

    public void actionPerformed(ActionEvent e)
    {

        if(e.getActionCommand().equals("EDITA"))
        {
        PointDialog dialog=new PointDialog(this,"Modification du point A",A);
     dialog.setVisible(true);
     ShowPoint();
        }
        else
        if(e.getActionCommand().equals("EDITB"))
        {
        PointDialog dialog=new PointDialog(this,"Modification du point B",B);
        dialog.setVisible(true);
        ShowPoint();
        }
    }
```

```
}
```

Le fichier Point.java

```java
public class Point {

    private double x,y;

    Point()
    {
    x=0;
    y=0;
    }

    public double getX(){return x;}
    double getY(){return y;}

    void setXY(double x,double y)
    {
    this.x=x;
    this.y=y;
    }

    double distance(Point P)
    {
    double dx,dy;
    dx=x-P.x;
    dy=y-P.y;
    return java.lang.Math.sqrt(dx*dx+dy*dy);
    }
}
```

Le fichier PointDialog.java

```java
import javax.swing.*;
import javax.swing.border.Border;
import java.awt.event.*;
import java.awt.*;

public class PointDialog extends JDialog implements ActionListener
{
    private Point P;
    JButton OK,Cancel;
    JLabel textX,textY;
    JTextField editX,editY;

    PointDialog(JFrame f,String s,Point P)
    {
    super(f,s,true);
    this.P=P;
    setLocationRelativeTo(f);
    setBounds(500,100,400,200);
    setDefaultCloseOperation(JDialog.DO_NOTHING_ON_CLOSE);

    Container c=getContentPane();
    c.setLayout(null);

    textX=new JLabel("Abscisse du point :");
    textX.setBounds(10,20,120,20);
    c.add(textX);

    textY=new JLabel("Ordonnée du point :");
    textY.setBounds(10,50,120,20);
    c.add(textY);

    editX=new JTextField(20);
    editX.setBounds(140,20,100,20);
    c.add(editX);
```

```
    editY=new JTextField(20);
    editY.setBounds(140,50,100,20);
    c.add(editY);

    Border bord=BorderFactory.createRaisedBevelBorder();
    OK=new JButton("OK");
    OK.setActionCommand("OK");
    OK.addActionListener(this);
    OK.setBorder(bord);
    OK.setBounds(260,25,60,20);
    c.add(OK);

    Cancel=new JButton("Annuler");
    Cancel.setActionCommand("CANCEL");
    Cancel.addActionListener(this);
    Cancel.setBorder(bord);
    Cancel.setBounds(260,50,60,20);
    c.add(Cancel);

    ShowPoint();
    }

    private void ShowPoint()
    {
        editX.setText(String.valueOf(P.getX()));
        editY.setText(String.valueOf(P.getY()));
    }


    public void actionPerformed(ActionEvent e)
    {
    double x,y;
    if(e.getActionCommand().equals("OK"))
            {
        boolean ok=true;

        if(ok)  {
             try{
                 x=Double.parseDouble(editX.getText());
                }catch(RuntimeException ex)
                {x=P.getX();editX.setText(String.valueOf(x));ok=false;}

                try{
                     y=Double.parseDouble(editY.getText());
                    }catch(RuntimeException ex)
                    {y=P.getY();editY.setText(String.valueOf(y));ok=false;}
                if(ok)
                    {
                    P.setXY(x,y);
                    dispose();
                    }
                }
        }
        else if(e.getActionCommand().equals("CANCEL"))
            {
            dispose();
            }
    }

}
```

Vous avez la permission de copier, distribuer et/ou modifier ce document selon les termes de la **licence de documentation libre GNU**, version 1.2 ou plus récente publiée par la Free Software Foundation ; sans sections inaltérables, sans texte de première page de couverture et sans texte de dernière page de couverture.

Récupérée de « https://fr.wikibooks.org/w/index.php?title=Programmation_Java/Version_imprimable&oldid=483810 »

Dernière modification de cette page le 14 juillet 2015, à 01:33.
Les textes sont disponibles sous licence Creative Commons attribution partage à l'identique ; d'autres termes peuvent s'appliquer.
Voyez les termes d'utilisation pour plus de détails.
Développeurs