

# Introduction

---

Java est un langage de programmation originellement proposé par Sun Microsystems et maintenant par Oracle depuis son rachat de Sun Microsystems en 2010.

Java a été conçu avec deux objectifs principaux :

- Permettre aux développeurs d'écrire des logiciels indépendants de l'environnement *hardware* d'exécution.
- Offrir un langage orienté objet avec une bibliothèque standard riche

## L'environnement

---

L'indépendance par rapport à l'environnement d'exécution est garantie par la *machine virtuelle Java* (Java Virtual Machine ou **JVM**). En effet, Java est un langage compilé mais le compilateur ne produit pas de code natif pour la machine, il produit du [bytecode](#) : un jeu d'instructions compréhensibles par la JVM qu'elle va traduire en code exécutable par la machine au moment de l'exécution.

Pour qu'un programme Java fonctionne, il faut non seulement que les développeurs aient compilé le code source mais il faut également qu'un environnement d'exécution (comprenant la JVM) soit installé sur la machine cible.

Il existe ainsi deux environnements Java qui peuvent être téléchargés et installés depuis le [site d'Oracle](#) :

- JRE - Java Runtime Environment  
Cet environnement fournit uniquement les outils nécessaires à l'exécution de programmes Java. Il fournit entre-autres la machine virtuelle Java.
- JDK - Java Development Kit  
Cet environnement fournit tous les outils nécessaires à l'exécution mais aussi au développement de programmes Java. Il fournit entre-autres la machine virtuelle Java et le compilateur.

## Oracle JDK et Open JDK

---

Depuis 2006, le code source Java (et notamment le code source de la JVM) est progressivement passé sous licence libre GNU [GPL](#). Il existe une version de l'environnement Java incluant uniquement le code libre : [Open JDK](#). Depuis la version 11, Oracle distribue son propre JDK sous licence propriétaire (mais qui ne peut pas être utilisé en production) et une version sous licence GPL v2 + CPE (*Classpath Exception*).

## Un bref historique des versions

---

version	date	faits notables
1.0	janvier 1996	La naissance
1.1	février 1997	Ajout de JDBC et définition des JavaBeans

version	date	faits notables
1.2	décembre 1998	Ajout de Swing, des collections (JCF), de l'API de réflexion. La machine virtuelle inclut la compilation à la volée (Just In Time)
1.3	mai 2000	JVM HotSpot
1.4	février 2002	support des regexp et premier parser de XML
5	septembre 2004	évolutions majeures du langage : autoboxing, énumérations, varargs, imports statiques, foreach, types génériques, annotations. Nombreux ajout dans l'API standard
6	décembre 2006	
7	juillet 2011	Quelques évolutions du langage et l'introduction de java.nio
8	mars 2014	évolutions majeures du langage : les lambdas et les streams et une nouvelle API pour les dates
9	septembre 2017	les modules (projet Jigsaw) et jshell
10	mars 2018	inférence des types pour les variables locales (mot-clé <code>var</code> )
11	septembre 2018	Nouvelle licence : la version propriétaire de Oracle JDK n'est plus utilisable gratuitement en production et requière une souscription payante. Oracle fournit cependant une distribution libre sous licence GPL v2 + CPE ( <i>Classpath Exception</i> ) disponible sur le site <a href="http://jdk.java.net/">http://jdk.java.net/</a> . L'utilisation de cette distribution n'implique pas une distribution libre du code de l'application. Suppression de certains modules dépréciés en Java 9 (CORBA, JAXB, JAX-WS...). Nouvelle API de client HTTP.
12	mars 2019	Shenandoah : le nouvel algorithme de ramasse-miettes ( <i>Garbage collector</i> ). Extension de l'expression <code>switch</code> ( <i>preview feature</i> ) : support de la <i>arrow syntax</i> <code>-&gt;</code> , valeur de retour sur un <code>break</code> et possibilité de lister plusieurs constantes dans l'expression <code>case</code> .
13	septembre 2019	Nouvelle implémentation de la couche <i>socket</i> . Extension de l'expression <code>switch</code> ( <i>preview feature</i> ) : introduction du mot-clé <code>yield</code> pour affecter le résultat d'une expression <code>switch</code> . Support des blocs de texte ( <i>preview feature</i> ) en utilisant le tripe guillemet <code>"""</code> .
14	mars 2020	Extension de l'expression <code>switch</code> ( <i>standard feature</i> ) : introduction du mot-clé <code>yield</code> pour affecter le résultat d'une expression <code>switch</code> . Nouveau type <code>record</code> ( <i>preview feature</i> ). <i>Pattern matching</i> pour <i>instanceof</i> ( <i>preview feature</i> ).

version	date	faits notables
15	septembre 2020	Support des blocs de texte ( <i>standard feature</i> ) en utilisant le tripe guillemet <code>"""</code> . <code>Hidden classes</code> : création de classes à la volée qui ne peuvent pas être inspectées avec l'API de réflexion. Implémentation de l'algorithme de signature Edwards-Curve (EdDSA). <i>Sealed classes (preview feature)</i> : Les classes (ou les interfaces) scellées permettent de restreindre l'ensemble des classes pouvant les étendre (les implémenter).

À partir de la version 12, les nouvelles évolutions dans la grammaire du langage passent par un état *preview*. Une *preview feature* est une évolution complète disponible dans une version mais qui doit être activée explicitement à la compilation. Une *preview feature* n'est donc pas considérée comme définitive, ni même officiellement intégrée au langage. Elle permet à la communauté des développeurs d'évaluer cette fonctionnalité. Dans une version ultérieure, cette fonctionnalité peut acquérir le statut de *standard feature*. Cela signifie qu'elle est pleinement intégrée au langage et qu'elle est disponible par défaut.

Par exemple, pour activer les *preview features* disponibles dans la version 12, il faut passer les paramètres suivants à la compilation :

```
javac --enable-preview --release 12 MaClasse.java
```

Il faut également spécifier l'activation des *preview features* à l'exécution :

```
java --enable-preview MaClasse
```

## Installation

Pour suivre ce cours, vous aurez besoin d'un environnement de développement Java. L'installation du JDK dépend de votre plate-forme : il est distribué sous la forme d'un installateur pour Windows et MacOS, et sous la forme d'un package ou d'une archive sous Linux.

Attention

Votre machine dispose déjà très certainement d'un environnement d'exécution Java (**JRE**) qui ne contient pas les outils nécessaires pour développer en Java. Vous devez donc installer un JDK.

Téléchargement du JDK 11

<https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html>

Vous aurez également besoin d'un environnement de développement intégré (**IDE**) pour Java. Il en existe plusieurs. Dans le cadre de cette formation, nous utiliserons [Eclipse](#)

Il existe plusieurs *packages* d'Eclipse visant des publics différents. Si vous voulez uniquement faire du développement d'application Java, vous pouvez télécharger le package *Eclipse IDE for Java Developers*. Si par contre, vous voulez réaliser des développements d'application Web alors peut-être devriez-vous télécharger le package *Eclipse IDE for Java EE Developers*. Cette dernière version offre des fonctionnalités supplémentaires pour le développement d'applications serveur.

Eclipse est distribué sous la forme d'une archive tar.gz pour Linux, d'un package dmg pour MacOS et d'une archive zip pour Windows que vous pouvez décompresser où vous le souhaitez.

## Téléchargement de l'IDE Eclipse

- Pour Windows

[https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2019-09/R/eclipse-jee-2019-09-R-win32-x86\\_64.zip](https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2019-09/R/eclipse-jee-2019-09-R-win32-x86_64.zip)

- Pour MacOS

[https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2019-09/R/eclipse-jee-2019-09-R-macosx-cocoa-x86\\_64.dmg](https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2019-09/R/eclipse-jee-2019-09-R-macosx-cocoa-x86_64.dmg)

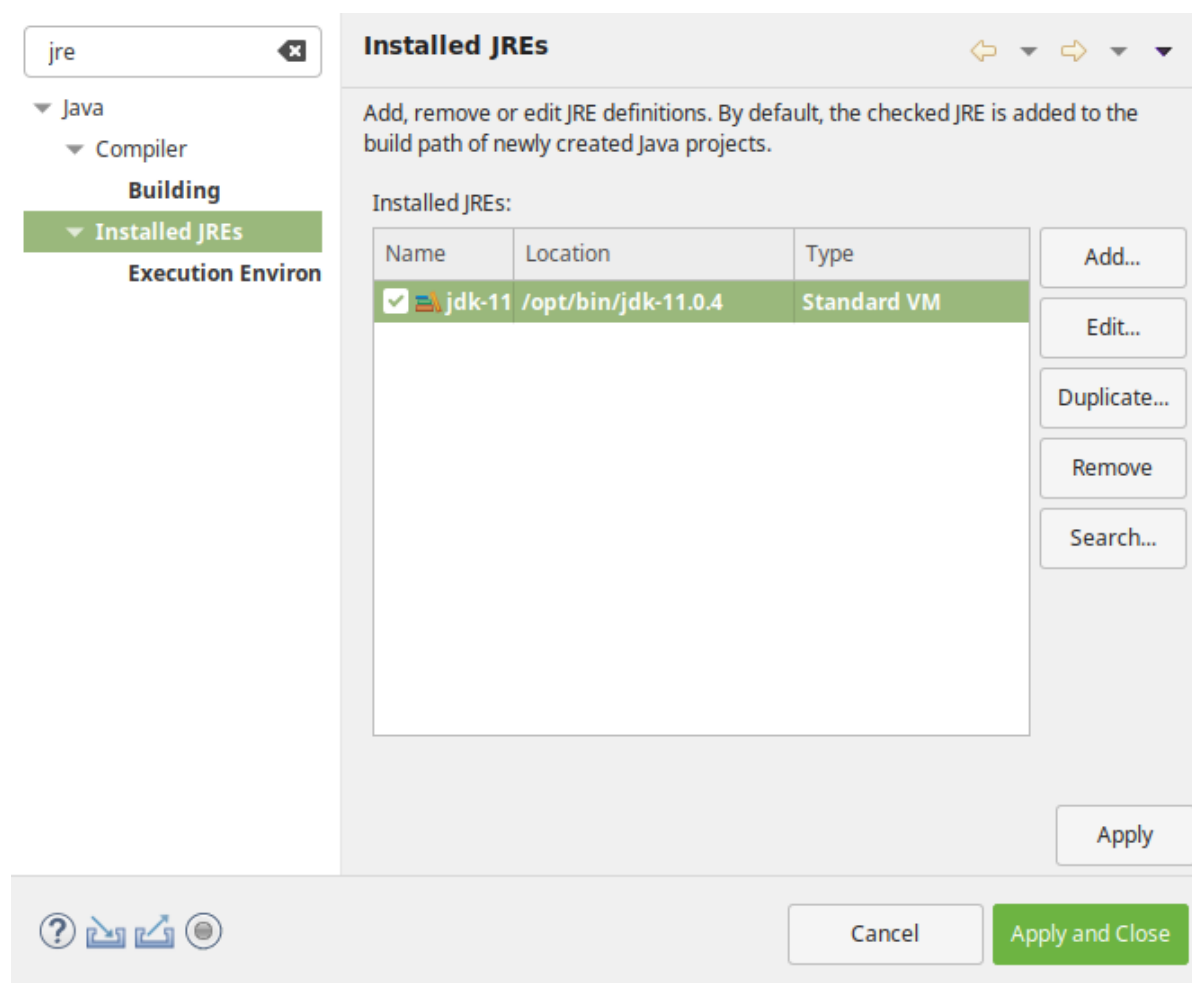
- Pour Linux

[https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2019-09/R/eclipse-jee-2019-09-R-linux-gtk-x86\\_64.tar.gz](https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2019-09/R/eclipse-jee-2019-09-R-linux-gtk-x86_64.tar.gz)

## Configuration d'Eclipse

Après avoir lancé Eclipse, il va falloir vérifier la version de Java utilisée par l'IDE et la modifier si nécessaire.

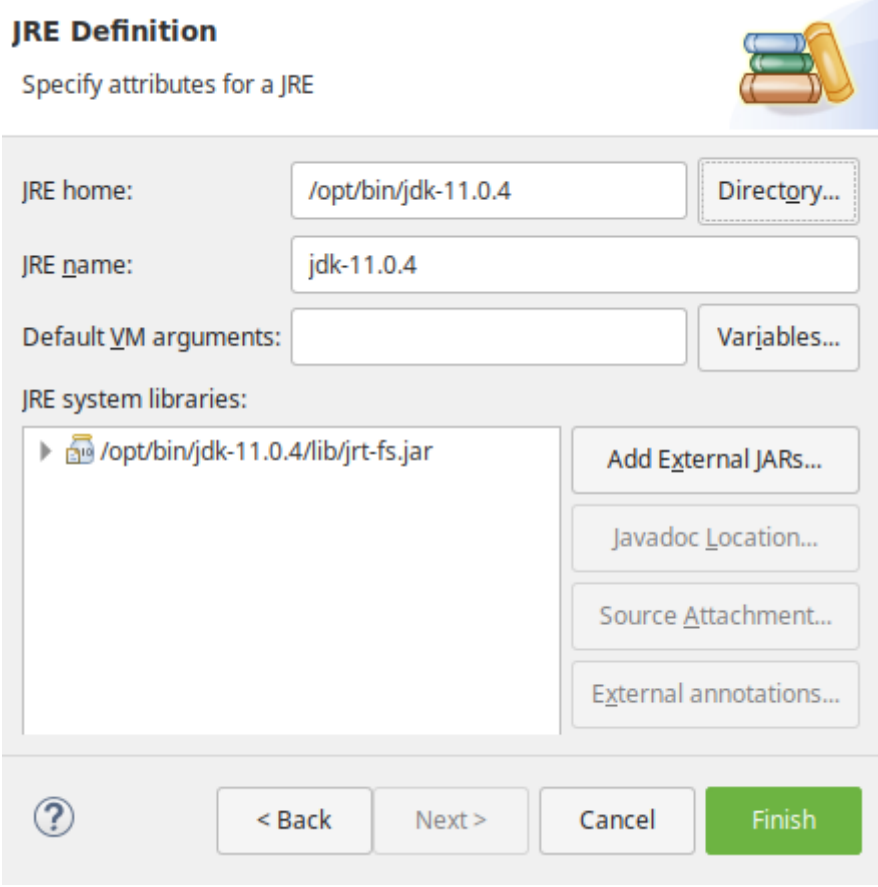
Pour vérifier les versions de Java disponibles dans Eclipse, ouvrez les préférences utilisateur, dans le menu Window • Preferences. Dans la zone de filtre en haut à gauche, saisissez « jre » (pour Java Runtime Environment) et sélectionnez dans l'arbre **Installed JREs** comme ci-dessous :



Vérifiez que le JDK que vous avez installé se trouve bien dans la liste des JRE détectés par Eclipse. De plus le JDK doit être coché pour indiquer à Eclipse qu'il s'agit de l'environnement d'exécution à utiliser par défaut pour tous les projets.

Si vous ne trouvez pas le JDK installé dans la liste, utilisez le bouton **Add...** pour l'ajouter manuellement :

1. Pour le choix du type de JRE, choisissez « Standard VM » et cliquez sur **Next**
2. Dans la boîte de dialogue **Add JRE**, cliquez sur le bouton **Directory...** pour sélectionner le répertoire d'installation du JDK
3. Eclipse s'occupe ensuite de remplir les champs nécessaires et vous n'avez plus qu'à cliquer sur **Finish**



**JRE Definition**  
Specify attributes for a JRE

JRE home: /opt/bin/jdk-11.0.4 Directory...

JRE name: jdk-11.0.4

Default VM arguments: Variables...

JRE system libraries:

- ▶ /opt/bin/jdk-11.0.4/lib/jrt-fs.jar

Add External JARs...

Javadoc Location...

Source Attachment...

External annotations...

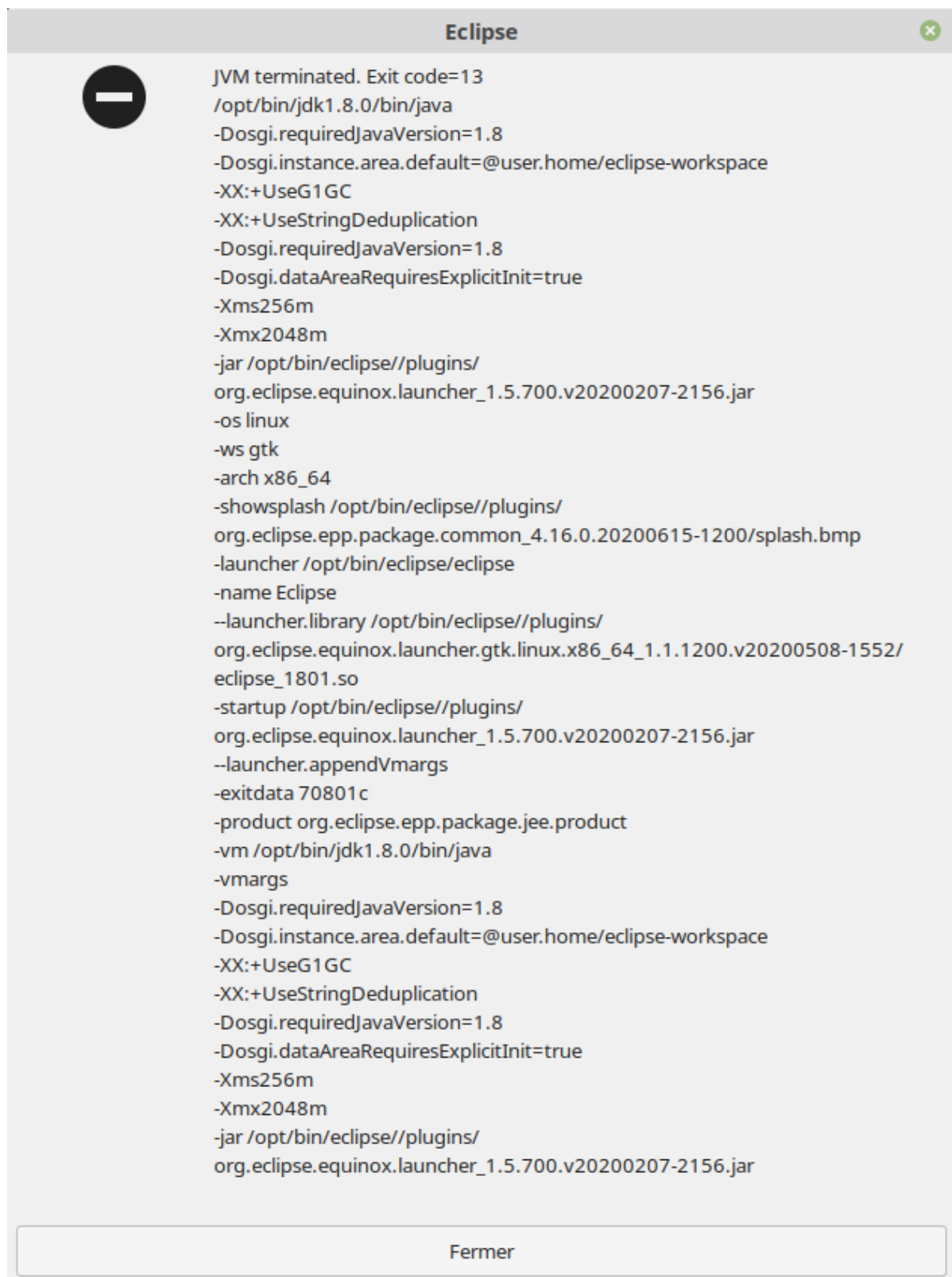
? < Back Next > Cancel Finish

Attention

N'oubliez pas de cocher la ligne de votre JDK dans l'écran **Installed JREs** pour qu'il devienne l'environnement d'exécution par défaut.

Astuce

**Eclipse ne se lance pas !**



Il est possible que Eclipse échoue à démarrer. Comme Eclipse est lui-même écrit en Java, il a besoin d'une machine virtuelle Java (JVM) pour s'exécuter. Il est possible que votre système n'arrive pas à détecter la présence d'une JVM. Cela peut se produire si vous avez réalisé l'installation manuellement à partir d'une archive. Vous devez alors préciser dans le fichier de configuration de lancement d'Eclipse, le chemin complet de l'exécutable `javaw.exe` (sous Windows) ou `java` (sous MacOS et Linux). Cet exécutable est présent dans le répertoire `bin` du JDK que vous avez installé. Sous Windows, il s'installe par défaut dans `C:\Program Files\Java\jdk-*version du JDK*\bin`

Pour modifier la configuration d'Eclipse, éditez le fichier `eclipse.ini` dans le répertoire de votre installation d'Eclipse. Pensez à faire une copie de ce fichier au cas où... Vous devez ajouter le paramètre `-vm` suivi à la ligne du chemin complet vers l'exécutable `javaw.exe` (ou `java`). **ATTENTION**, le paramètre `-vm` précède obligatoirement le paramètre `-vmargs` déjà présent dans le fichier :

Un exemple de fichier `eclipse.ini` modifié

```
-startup
plugins/org.eclipse.equinox.launcher_1.5.700.v20200207-2156.jar
--launcher.library
plugins/org.eclipse.equinox.launcher.gtk.linux.x86_64_1.1.1200.v20200508-1552
-product
org.eclipse.epp.package.jee.product
-showsplash
org.eclipse.epp.package.common
--launcher.defaultAction
openFile
--launcher.defaultAction
openFile
--launcher.appendVmargs
-vm
C:/Progra~1/Java/jdk11/bin/javaw.exe
-vmargs
-Dosgi.requiredJavaVersion=1.8
-Dosgi.instance.area.default=@user.home/eclipse-workspace
-XX:+UseG1GC
-XX:+UseStringDeduplication
--add-modules=ALL-SYSTEM
-Dosgi.requiredJavaVersion=1.8
-Dosgi.dataAreaRequiresExplicitInit=true
-Xms256m
-Xmx2048m
--add-modules=ALL-SYSTEM
```

Pour plus de détail, reportez-vous au [Wiki d'Eclipse](#).

## Compilation & exécution

Dans ce chapitre, nous n'allons pas directement nous intéresser au langage Java. Nous allons plutôt essayer de comprendre les mécanismes sous-jacents à la compilation et à l'exécution d'un programme Java. Nous verrons également comment créer un projet dans Eclipse.

## Un premier programme

Nous allons utiliser comme exemple la programme Java suivant :

```
/**
 * Ce programme n'est pas très intéressant
 */
public class PremierProgramme {

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

}
```

La syntaxe du langage nous importe peu pour l'instant. Ce programme Java va simplement écrire le message « Hello World! » sur la sortie standard.

Téléchargez le fichier [PremierProgramme.java](#) contenant ce code.

## La compilation

Pour pouvoir exécuter ce programme, nous allons devoir le compiler. Pour cela, nous devons utiliser le programme [javac](#) (Java Compiler) dont c'est la fonction. Dans un terminal, il suffit de se rendre dans le répertoire où se situe le fichier et de lancer la commande de compilation :

```
$ cd /home/david/workspace/workspace-java/exemple
$ javac PremierProgramme.java
```

Attention

Si votre système ne connaît pas la commande [javac](#) cela signifie simplement que le répertoire contenant ce programme n'est pas déclaré dans le chemin d'exécution du système. Il vous suffit de le rajouter ou de donner le chemin complet menant à ce programme. Par exemple :

```
$ /opt/bin/jdk1.8/bin/javac PremierProgramme.java
```

Le programme [javac](#) est installé dans le sous-répertoire **bin** du répertoire d'installation du JDK

La compilation devrait se passer sans problème et aboutir à la création du fichier **PremierProgramme.class** dans le même répertoire que le fichier java.

Note

Nous y reviendrons plus tard mais en Java, nous déclarons des classes (parfois un peu spéciales). Un fichier source porte l'extension **java** et contient le code source d'une classe. Un fichier résultant de la compilation porte le même nom que le fichier source mais avec l'extension **class**. Ce fichier n'est pas directement éditable car il contient des instructions en bytecode compréhensibles par la JVM.

## L'exécution

L'exécution d'un programme se fait par l'intermédiaire de la machine virtuelle. Pour invoquer cette dernière, on utilise tout simplement la commande [java](#) suivie du nom de la classe **sans l'extension** :

```
$ java PremierProgramme
```



## Attention

Si votre système ne connaît pas la commande [java](#) cela signifie simplement que le répertoire contenant ce programme n'est pas déclaré dans le chemin d'exécution du système. Il vous suffit de le rajouter ou de donner le chemin complet menant à ce programme. Par exemple :

```
$ /opt/bin/jdk-10/bin/java PremierProgramme
```

Le programme [java](#) est installé dans le sous-répertoire **bin** du répertoire d'installation du JDK

La commande [java](#) va chercher le fichier PremierProgramme.class pour l'exécuter. Cela signifie qu'à ce stade, vous n'êtes pas obligé de disposer du fichier source PremierProgramme.java.

## La liaison dynamique

Tous les langages de programmation évolués utilisent la notion de liaison (**link**). En effet, il est nécessaire à un moment donné de pouvoir créer un programme à partir de plusieurs fichiers source. Généralement, les fichiers source sont compilés un à un puis un mécanisme de liaison permet de gérer les dépendances entre chacun des fichiers. En programmation, on distingue la liaison **statique** et la liaison **dynamique**.

La liaison statique est une étape qui intervient après la compilation et qui permet de regrouper l'ensemble des fichiers compilés dans un fichier exécutable unique. Les langages tels que C et C++ supportent la liaison statique.

La liaison dynamique est une étape qui intervient au moment du lancement du programme. On vérifie que les fichiers compilés sont disponibles pour l'exécution.

Java ne supporte que la liaison dynamique. Cela signifie que chaque fichier compilé donnera un fichier class. Cela signifie également qu'un programme Java est en fait une collection de plusieurs fichiers class.

Si votre programme est dépendant d'une bibliothèque tierce en Java, vous devez également fournir les fichiers de cette bibliothèque au moment de l'exécution.

### Note

Il est impossible d'écrire un programme Java qui n'ait aucune dépendance avec d'autres fichiers class. Dans notre exemple, même simple, nous sommes dépendants de la classe **System**. Nous sommes même dépendants de la classe **Object** alors que ce mot n'est pas présent dans le fichier source. Heureusement, ces classes font partie de la bibliothèque standard de Java qui est disponible avec l'environnement d'exécution. Nous n'avons donc pas à nous préoccuper de comment la JVM va trouver le code pour ces classes. Mais elle le fera bel et bien en utilisant le mécanisme de liaison dynamique.

## Le classpath

La liaison dynamique implique qu'un programme Java est une collection de fichiers. Ces fichiers peuvent se trouver à différents endroits dans le système de fichiers. Il faut donc un mécanisme pour permettre de les localiser. En Java, on utilise le **classpath** : le chemin des classes. On peut par exemple spécifier un ou plusieurs chemins avec le paramètre **-classpath** aux commandes [java](#) et [javac](#) indiquant les répertoires à partir desquels il est possible de trouver des fichiers class.

```
$ java -classpath /home/david/workspace/workspace-java/exemple PremierProgramme
```

La commande ci-dessus peut être exécutée à partir de n'importe quel répertoire puisqu'elle précise un classpath. La JVM tentera de chercher un fichier `PremierProgramme.class` dans le répertoire `/home/david/Workspace/workspace-java/exemple`.

S'il existe des répertoires contenant des fichiers class que vous utilisez souvent, vous pouvez les inclure implicitement dans le classpath en déclarant ces répertoires dans la variable d'environnement **CLASSPATH**.

```
$ export CLASSPATH=/home/david/workspace/workspace-java/exemple
$ java PremierProgramme
```

Note

Même si le principe du classpath est simple, cela peut amener à des situations très complexes dans les projets. Si on indique plusieurs chemins, on peut avoir des répertoires utilisés comme classpath contenant des classes avec des noms identiques mais avec des comportements différents. On peut aussi exécuter à son insu du code malicieux. Depuis Java 9, un nouveau système baptisé *Jigsaw* et basé sur la notion de module a fait son apparition.

## Bibliothèques Java : les fichiers JAR

Si on se rappelle qu'un programme Java est une collection de fichiers class et qu'il n'est pas rare qu'un programme ait besoin de centaines voire de milliers de ces fichiers alors on se rend vite compte qu'il n'est pas très facile de distribuer un programme Java sous cette forme.

Pour palier à ce problème, on peut utiliser des fichiers jar. JAR signifie *Java ARchive* : il s'agit d'un fichier zip contenant un ensemble de fichiers class mais qui a l'extension **.jar**. Java fournit l'utilitaire [jar](#) pour créer une archive :

```
$ jar -cf monappli.jar PremierProgramme.class
```

L'utilitaire [jar](#) reprend la syntaxe de **tar** sous les systèmes \*NIX.

Un fichier JAR peut être ajouté au classpath rendant ainsi disponible l'ensemble des fichiers qu'il contient.

```
$ export CLASSPATH=/home/david/workspace/workspace-java/exemple/monappli.jar
$ java PremierProgramme
```

C'est un moyen simple de distribuer son code. Toutes les bibliothèques tierces Java sont disponibles sous la forme d'un fichier JAR.

## Création d'un projet dans Eclipse

Il est utile de comprendre le fonctionnement des outils tels que [java](#) ou [javac](#) mais ils ne sont pas d'une utilisation très aisée pour de vrais projets. On préférera utiliser un outil de build comme [Ant](#), [Maven](#) ou [Gradle](#) pour automatiser la compilation et un environnement de développement intégré comme [Eclipse](#) pour le développement.

Eclipse fournit des avantages précieux pour les développeurs. Notamment :

- Eclipse compile automatiquement les fichiers lorsqu'ils sont sauvés. Il est donc possible d'avoir immédiatement un retour sur les éventuelles erreurs de syntaxe ou autres.

- Eclipse offre un environnement riche pour manipuler et modifier les fichiers sources.

Quelques raccourcis clavier utiles dans Eclipse :

<b>CTRL + espace</b>	<b>Complétion de code</b>
CTRL + 1 (ou CTRL + SHIFT + 1)	Suggestions
SHIFT + ALT + R	Renommer dans tous les fichiers
MAJ + CTRL + F	Reformater le code
MAJ + CTRL + O	Organiser les imports
CTRL + SHIFT + T	Chercher le fichier d'une classe
CTRL + SHIFT + R	Chercher une ressource (un fichier)
F11	Exécuter la classe courante

On peut créer toutes sortes de projets différents dans Eclipse. Pour nous, le plus utile sera bien sûr le projet Java. Pour cela, il suffit d'aller dans le menu File ▸ New ▸ Java Project. On obtient alors la boîte de dialogue suivante :

**New Java Project** + x

**Create a Java Project**

Create a Java project in the workspace or in an external location.

Project name:

☒ Use default location

Location:

JRE

☒ Use an execution environment JRE:

☐ Use a project specific JRE:

☐ Use default JRE (currently 'jdk-10') [Configure JREs...](#)

Project layout

☐ Use project folder as root for sources and class files

☒ Create separate folders for sources and class files [Configure default...](#)

Working sets

☐ Add project to working sets

Working sets:

Il suffit de donner le nom du projet et de cliquer sur **Finish**.

Si la boîte de dialogue suivante s'affiche :

**New module-info.java** + x

**Create module-info.java**

Create a new module-info.java file.

Module name:

Cliquez sur **Don't create**

Note

Avec la boîte de dialogue ci-dessus, Eclipse nous demande si nous voulons créer un fichier `module-info.java`. Il s'agit d'un fichier spécial pour la déclaration des modules (c'est-à-dire des dépendances). Il s'agit d'une utilisation avancée de Java qui ne nous concerne pas pour l'instant. Si vous devez créer un nouveau projet plus tard, choisissez toujours le bouton **Don't create**.

Il se peut qu'Eclipse ouvre ensuite une boîte de dialogue pour vous demander si vous voulez changer de *perspective*. Dans Eclipse, une perspective est un agencement de l'espace de travail adapté pour certaines tâches. Il existe par exemple une perspective Java adaptée pour développer du code Java et une perspective Debug pour le debuggage du code.

Un fois le projet créé, on voit que Eclipse a ajouté automatiquement un répertoire `src` destiné à accueillir les sources du projet. Il suffit d'ajouter le fichier `PremierProgramme.java` à cet endroit.

Comme indiqué ci-dessus, toute modification dans ce fichier entraînera automatiquement sa compilation au moment de la sauvegarde.

Si l'on souhaite distribuer son projet, on peut, par exemple, produire un fichier JAR. Pour cela, il suffit de faire un clic droit sur le nom du projet dans le *Package Explorer* et de choisir **Export**. Dans la boîte de dialogue d'export, il faut chercher « jar » et sélectionner Java » JAR File. En cliquant sur **Next**, on spécifie le nom et l'emplacement du fichier JAR et il sera créé en cliquant sur **Finish**.

## Exercice

---

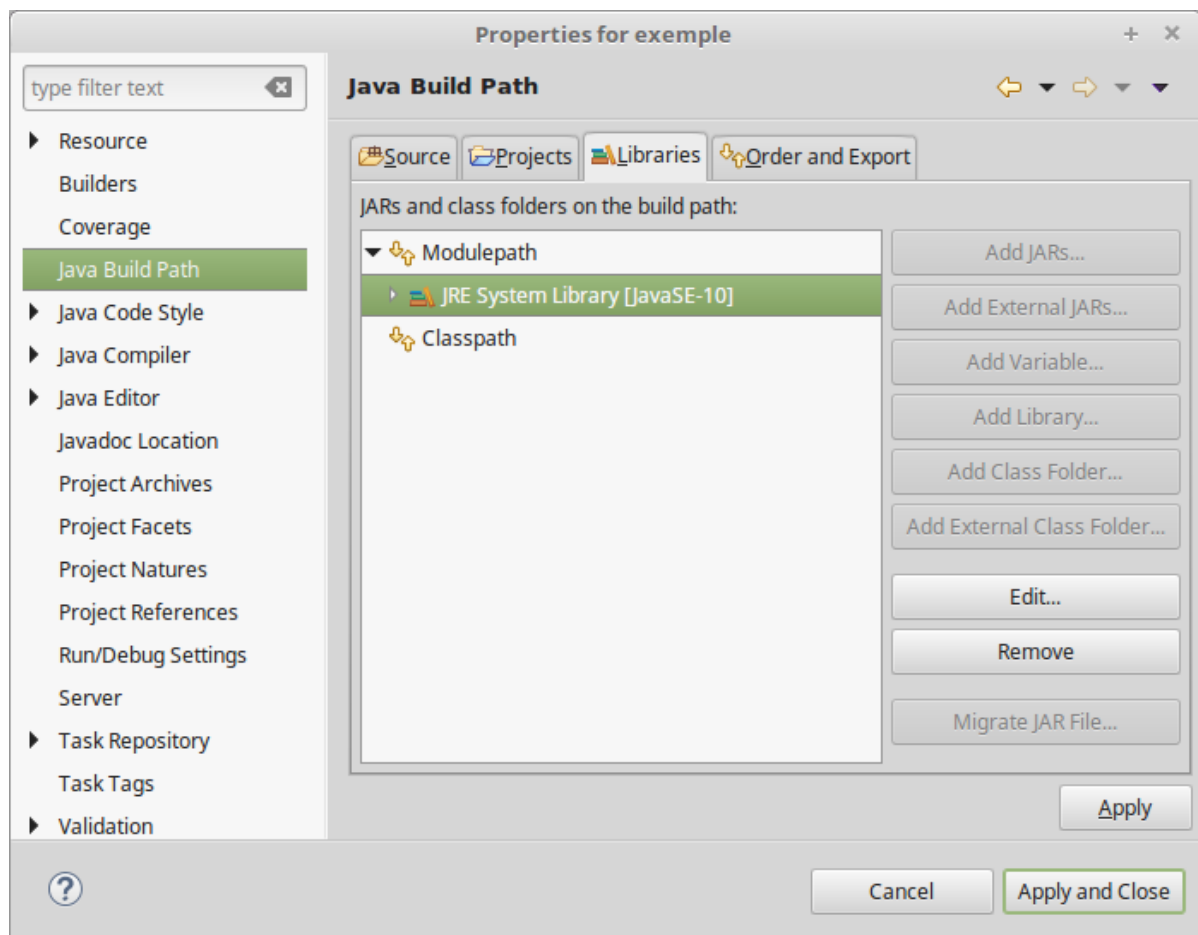
Utilisation d'une bibliothèque JAR

Pour cet exercice, nous allons créer un document PDF grâce à la bibliothèque [iText](#). Pour cela, téléchargez le fichier JAR suivant :

- <http://central.maven.org/maven2/com/itextpdf/itextpdf/5.5.13/itextpdf-5.5.13.jar>

Créez un projet Java dans Eclipse et copiez les fichiers JAR téléchargés dans un répertoire `lib` que vous créerez dans le projet Eclipse.

Ajoutez maintenant le fichier JAR comme bibliothèques du projet. Pour cela, faites un clic droit sur le nom du projet dans le *Package Explorer* et choisissez **Properties**. Dans la boîte de dialogue des propriétés du projet, choisissez **Java Build Path** et enfin, cliquez sur l'onglet **Libraries** :



Utilisez le bouton **Add JARs...** pour ajouter le fichier téléchargé comme une bibliothèque de votre projet.

Astuce

Dans le *Package Explorer*, vous pouvez directement faire un clic droit sur chaque fichier JAR et choisir Build Path > Add to Build Path.

Créer ensuite la classe *PremierPdf* et ajoutez le code suivant :

```
import java.io.FileOutputStream;

import com.itextpdf.text.Document;
import com.itextpdf.text.Paragraph;
import com.itextpdf.text.pdf.PdfWriter;

public class PremierPdf {

    public static void main(String[] args) throws Exception {
        Document document = new Document();
        PdfWriter.getInstance(document, new FileOutputStream("premier-pdf.pdf"));
        document.open();
        document.addTitle("Mon premier PDF");
        document.add(new Paragraph("Hello the world en PDF grâce à iText"));
        document.close();
    }
}
```

En exécutant cette classe, vous obtenez à la racine de votre projet le fichier *premier-pdf.pdf* créé grâce à la bibliothèque [iText](#).

## Astuce

Si vous ne voyez pas le fichier PDF apparaître dans le *Package Explorer*, faites un clic droit sur le nom du projet et choisissez **Refresh**.

## Note

Pour cet exercice, le fichier JAR est téléchargé depuis le site <http://central.maven.org> qui référence des milliers de bibliothèques Java. [Maven](#) est un outil de *build* qui permet de gérer les étapes de construction de votre projet et notamment de déclarer les dépendances logicielles.

# La structure fondamentale du langage

La syntaxe du langage Java est à l'origine très inspirée du C et du C++. Ces deux langages de programmation ont également servi de base au C#. Donc si vous connaissez C, C++ ou C# vous retrouverez en Java des structures de langage qui vous sont familières.

## Les instructions

Java est un [langage de programmation impératif](#). Cela signifie qu'un programme Java se compose d'instructions (**statements**) décrivant les opérations que la machine doit exécuter. En Java une instruction est délimitée par un **point-virgule**.

```
double i = 0.0;  
i = Math.sqrt(16);
```

## Le typage

Un type permet d'indiquer au programme les valeurs que peuvent prendre une variable, un paramètre ou un attribut. Java est un [langage fortement typé](#). Lorsque l'on déclare une variable un paramètre ou un attribut, il est **obligatoire de préciser son type**. Par exemple :

```
int i;
```

L'instruction précédente est la déclaration d'une variable *i*. Le type est toujours indiqué à gauche du nom de la variable (c'est également le cas pour les paramètres et les attributs). Dans l'exemple précédent, la variable *i* est déclarée comme étant du type `int`, c'est-à-dire qu'elle représente la valeur d'un nombre entier. En déclarant le type, le programmeur restreint le type de données que peut représenter cette variable. Ainsi, le compilateur va pouvoir signaler une erreur si jamais une instruction dans le reste du programme tente d'affecter une valeur qui n'est pas adéquate.

```
int i = 0;  
i = "bonjour"; // ERREUR : on tente d'affecter une chaîne de caractères  
               // à une variable qui représente un nombre
```

On dit que le typage en Java est fort car une fois que l'on a déclaré une variable (ou un paramètre ou un attribut) avec un type, il n'est plus possible de modifier cette déclaration.

Le type est également vérifié par la machine virtuelle au moment de l'exécution du programme. Le langage Java supporte donc le typage statique (vérification réalisée au moment de la compilation) et le typage dynamique (vérification réalisée au moment de l'exécution).

## Note

Depuis Java 10, il est possible d'utiliser l'identifiant `var` à la place du type :

```
var i = 0;
```

Lorsque que le compilateur rencontre une instruction comme la précédente, il détermine le type de la variable en fonction de la valeur d'initialisation (indiquée après le signe égal). Pour l'exemple précédent, comme on affecte 0 à la variable `i`, le compilateur déduit que le type de cette variable est `int` (on dit qu'il infère le type). Cette syntaxe permet d'éviter la répétition quand le type est évident.

Comme l'inférence de type repose sur la valeur d'initialisation, `var` est limité à la déclaration de variables précisant une valeur d'initialisation.

## Les blocs de code

Java permet de structurer le code en bloc. Un bloc est délimité par des **accolades**. Un bloc permet d'isoler par exemple le code conditionnel à la suite d'un **if** mais il est également possible de créer des blocs anonymes.

```
double i = 0.0;
i = Math.sqrt(16);

if (i > 1) {
    i = -i;
}

{
    double j = i * 2;
}
```

Un bloc de code n'a pas besoin de se terminer par un point-virgule. Certains outils émettent un avertissement si vous le faites.

## Les commentaires

Un commentaire sur une ligne commence par `//` et continue jusqu'à la fin de la ligne :

```
// ceci est un commentaire
double i = 0.0; // ceci est également un commentaire
```

Un commentaire sur plusieurs lignes commence par `/*` et se termine par `*/` :

```
/* ceci est un commentaire
   sur plusieurs lignes */
double i = 0.0;
```

Il existe un type spécial de commentaires utilisé par l'utilitaire [javadoc](#). Ces commentaires servent à générer la documentation au format HTML de son code. Ces commentaires, appelés **commentaires javadoc**, commencent par `/**` :



```

/**
 * Une classe d'exemple.
 *
 * Cette classe ne fait rien. Elle sert juste à donner un exemple de
 * commentaire javadoc.
 *
 * @author David Gayerie
 * @version 1.0
 */
public class MaClasse {

}

```

## Le formatage du code

Le compilateur Java n'impose pas de formatage particulier du code. Dans la mesure où une instruction se termine par un point-virgule et que les blocs sont délimités par des accolades, il est possible de présenter du code de façon différente. Ainsi, le code suivant :

```

double i = 0.0;
i = Math.sqrt(16);

if (i > 1) {
    i = -i;
}

```

est strictement identique pour le compilateur à celui-ci :

```

double i=0.0;i=Math.sqrt(16);if(i>1){i=-i;}

```

Cependant, le code source est très souvent relu par les développeurs, il faut donc en assurer la meilleure lisibilité. Les développeurs Java utilisent une convention de formatage qu'il **faut** respecter. Des outils comme Eclipse permettent d'ailleurs de reformater le code (sous Eclipse avec le raccourci clavier *MAJ + CTRL + F*). Rappelez-vous des conventions suivantes :

```

// On revient à la ligne après une accolade (mais pas avant)
if (i > 0) {
    // ...
}

// On revient systématiquement à la ligne après un point virgule
// (sauf) dans le cas de l'instruction for
int j = 10;
for (int i = 0; i < 10; ++i) {
    j = j + i;
}

// Dans un bloc de code, on utilise une tabulation ou des espaces
// pour mettre en valeur le bloc

if (i > 0) {
    if (i % 2 == 0) {
        // ...
    }
}

```

```

    } else {
        // ...
    }
}

// On sépare les variables des opérateurs par des espaces
i = i + 10; // plutôt que i=i+10

```

## Les conventions de nommage

Chaque langage de programmation et chaque communauté de développeurs définissent des conventions sur la façon de nommer les identifiants dans un programme. Comme pour le formatage de code, cela n'a pas d'impact sur le compilateur mais permet de garantir une bonne lisibilité et donc une bonne compréhension de son code par ses pairs. Les développeurs Java sont particulièrement attachés au respect des conventions de nommage.

Type	Convention	Exemple
Packages	Un nom de package s'écrit toujours en minuscule. L'utilisation d'un _ est tolérée pour représenter une séparation.	java.utils com.company.extra_utils
Classes et interfaces	Le nom des classes et des interfaces ne doivent pas être des verbes. La première lettre de chaque mot doit être en majuscule (écriture dromadaire).	MyClass SuppressionClientOperateur
Annotations	La première lettre de chaque mot doit être une majuscule (écriture dromadaire). Il est toléré d'écrire des sigles intégralement en majuscules.	@InjectIn @EJB
Méthodes	Le nom d'une méthode est le plus souvent un verbe. La première lettre doit être en minuscule et les mots sont séparés par l'utilisation d'une majuscule (écriture dromadaire).	run() runFast() getWidthInPixels()
Variables	La première lettre doit être en minuscule et les mots sont séparés par l'utilisation d'une majuscule (écriture dromadaire). Même si cela est autorisé par le compilateur, le nom d'une variable ne doit pas commencer par _ ou \$. En Java, les développeurs n'ont pas pour habitude d'utiliser une convention de nom pour différencier les variables locales des paramètres ou même des attributs d'une classe. Le nom des variables doit être explicite sans utiliser d'abréviation. Pour les variables « jetables », l'utilisation d'une lettre est d'usage (par exemple i, j ou k)	widthInPixels clientsInscrits total

Type	Convention	Exemple
Constantes	Le nom d'une constante s'écrit intégralement en lettres majuscules et les mots sont séparés par _.	LARGEUR_MAX INSCRIPTIONS_PAR_ANNEE

## Les mots-clés

Comme pour la plupart des langages de programmation, il n'est pas possible d'utiliser comme nom dans un programme un mot-clé du langage. La liste des mots-clés en Java est :

```

abstract  continue  for          new          switch
assert    default   if           package     synchronized
boolean   do         goto        private     this
break     double    implements protected  throw
byte      else      import      public      throws
case      enum     instanceof return      transient
catch     extends  int         short       try
char      final    interface   static      void
class     finally  long        strictfp   volatile
const     float     native      super       while
_ (underscore)

```

Note

**goto** et **const** sont des mots-clés réservés mais qui n'ont pas de signification dans le langage Java.

Il existe également des mots réservés qui ne sont pas strictement des mots-clés du langage :

```
true  false  null
```

## Une première classe

Java est langage orienté objet. Cela signifie que (presque) tout est un objet. La définition d'un objet s'appelle une classe. Donc programmer en Java revient à **déclarer** des classes, à **instancier** des objets à partir des classes déclarées ou fournies et à effectuer des opérations sur ces objets.

### Déclarer une classe

Dans ce chapitre, nous allons ébaucher l'implémentation d'une classe Voiture. La classe Voiture sera une représentation abstraite d'une voiture pour les besoins de notre application.

En Java, une classe est déclarée dans son propre fichier qui **doit** porter le même nom que la classe avec l'extension *.java*. Il nous faut donc créer le fichier **Voiture.java** :

```
/**
 * Une première classe représentant une voiture
 *
 * @author David Gayerie
 */
public class Voiture {

}
```

Vous pouvez [télécharger le fichier voiture.java](#). Vous pouvez également créer directement dans un projet sous Eclipse avec le menu File ▶ New ▶ Class.

#### Note

Si vous créez le fichier à partir d'Eclipse, celui-ci va vous proposer de créer le fichier dans un *package* (par défaut ayant le même nom que le projet). Vous pouvez effacer le contenu du champ package de la boîte de dialogue en attendant que nous abordions la notion de package.

## Anatomie d'une classe

En Java une classe est déclarée par le mot-clé **class** suivi du nom de la classe. Nous reviendrons plus tard sur le mot-clé **public** qui précède et qui permet de préciser la portée (**scope**) de la définition de cette classe. Ensuite, on ouvre un bloc avec des accolades pour déclarer le contenu de la classe.

La déclaration d'une classe peut contenir :

- des attributs

Les attributs représentent l'état interne d'un objet. Par exemple, notre voiture peut avoir un attribut pour mémoriser sa vitesse.

- des méthodes

Les méthodes représentent les opérations que l'on peut effectuer sur un objet de cette classe.

- des constantes

Les constantes sont un moyen de nommer des valeurs particulières utiles à la compréhension du code.

- des énumérations

Les énumérations sont des listes figées d'objets. Nous y reviendrons dans un chapitre ultérieur.

- des classes internes

Une classe peut contenir la déclaration d'autres classes que l'on appelle alors classes internes (**inner classes**). Nous y reviendrons dans un chapitre ultérieur.

L'ordre dans lequel apparaissent ces éléments dans la déclaration de la classe est sans importance en Java. Pour des raisons de commodité de lecture, les développeurs adoptent en général une convention : d'abord les constantes, puis les énumérations, puis les attributs et enfin les méthodes.

# Ajouter des méthodes

Ajoutons quelques méthodes à notre classe **Voiture**. Nous allons commencer par ajouter la méthode **getVitesse** qui permet de connaître la vitesse actuelle d'une voiture en km/h.

```
/**
 * Une première classe représentant une voiture
 *
 * @author David Gayerie
 */
public class Voiture {

    /**
     * @return La vitesse en km/h de la voiture
     */
    public float getVitesse() {

    }

}
```

Une méthode est identifiée par sa **signature**. La signature d'une méthode est de la forme :

```
[portée] [type de retour] [identifiant] ([liste des paramètres]) {
    [code]
}
```

Pour la méthode que nous venons de déclarer:

portée	public
type de retour	float (nombre à virgule flottante)
identifiant	getVitesse
liste des paramètres	aucun

Le code source précédent ne compilera pas, en effet, Java étant un langage fortement typé, nous sommes obligé d'indiquer le type de retour de la méthode *getVitesse* et donc le compilateur s'attend à ce que cette méthode retourne un nombre à virgule flottante. La vitesse de la voiture est typiquement une information qui correspond à l'état de la voiture à un moment donné. Il est donc intéressant de stocker cette information comme attribut de la classe :

```
/**
 * Une première classe représentant une voiture
 *
 * @author David Gayerie
 */
public class Voiture {

    private float vitesse;

    /**
```

```

    * @return La vitesse en km/h de la voiture
    */
    public float getvitesse() {
        return vitesse;
    }
}

```

Un attribut est identifié par :

```
[portée] [type] [identifiant];
```

Pour l'attribut *vitesse*, nous spécifions le type de portée **private**. En Java, un attribut a toujours une valeur par défaut qui dépend de son type. Pour le type **float**, la valeur par défaut est 0.

Nous pouvons maintenant enrichir notre classe avec des méthodes supplémentaires :

```

/**
 * Une première classe représentant une voiture
 *
 * @author David Gayerie
 */
public class Voiture {

    private float vitesse;

    /**
     * @return La vitesse en km/h de la voiture
     */
    public float getvitesse() {
        return vitesse;
    }

    /**
     * Pour accélérer la voiture
     * @param deltaVitesse Le vitesse supplémentaire
     */
    public void accelerer(float deltaVitesse) {
        vitesse = vitesse + deltaVitesse;
    }

    /**
     * Pour décélérer la voiture
     * @param deltaVitesse Le vitesse à soustraire
     */
    public void decelerer(float deltaVitesse) {
        vitesse = vitesse - deltaVitesse;
    }

    /**
     * Freiner la voiture.
     */
    public void freiner() {
        vitesse = 0;
    }
}

```

```
/**
 * Représentation de l'objet sous la forme
 * d'une chaîne de caractères.
 */
public String toString() {
    return "La voiture roule actuellement à " + vitesse + " km/h.";
}
}
```

Les méthodes *Voiture.accelerer(float)* et *Voiture.decelerer(float)* prennent toutes les deux un paramètre de type **float**. Comme ces méthodes ne retournent aucune valeur, nous sommes obligés de l'indiquer avec le mot-clé **void**.

La méthode *toString()* permet d'obtenir une représentation d'un objet sous la forme d'une chaîne de caractères. Cette méthode est notamment appelée lorsque l'on passe un objet en paramètre de la méthode `System.out.println()` pour afficher une représentation textuelle d'un objet. Notez, que l'opérateur **+** est utilisé en Java pour concaténer les chaînes de caractères et qu'il est possible de concaténer des chaînes de caractères avec d'autres types. Dans notre exemple, nous concaténons une chaîne de caractères avec le nombre à virgule flottante représentant la vitesse de la voiture.

Note

Java ne supporte pas la notion de fonction. Il n'est donc pas possible de déclarer des méthodes en dehors d'une classe.

## La méthode main

Si nous voulons utiliser notre classe dans un programme, il nous faut déterminer un point d'entrée pour l'exécution du programme. Un point d'entrée est représenté par la méthode **main** qui doit avoir la signature suivante :

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

Une classe ne peut déclarer qu'une seule méthode **main**. En revanche, toutes les classes peuvent déclarer une méthode **main**. Cela signifie qu'une application Java peut avoir plusieurs points d'entrée (ce qui peut se révéler très pratique). Voilà pourquoi la commande **java** attend comme paramètre le nom d'une classe qui doit déclarer une méthode **main**.

Ajoutons une méthode **main** à la classe *Voiture* pour réaliser un programme très simple :

1 2 3 4	
5 6 7 8	
9 10 11	
12 13 14	
15 16 17	
18 19 20	
21 22 23	
24 25 26	
27 28 29	
30 31 32	
33 34 35	
36 37 38	
39 40 41	
42 43 44	
45 46 47	
48 49 50	
51 52 53	
54 55 56	
57 58 59	
60 61 62	

```

/** * Une première classe représentant une voiture * * @author
David Gayerie */ public class Voiture { private float vitesse;
/** * @return La vitesse en km/h de la voiture */ public float
getVitesse() { return vitesse; } /** * Pour accélérer la voiture
* @param deltaVitesse Le vitesse supplémentaire */ public void
accelerer(float deltaVitesse) { vitesse = vitesse + deltaVitesse;
} /** * Pour décélérer la voiture * @param deltaVitesse Le
vitesse à soustraire */ public void decelerer(float deltaVitesse)
{ vitesse = vitesse - deltaVitesse; } /** * Freiner la voiture.
*/ public void freiner() { vitesse = 0; } /** * Représentation de
l'objet sous la forme * d'une chaîne de caractères. */ public
String toString() { return "La voiture roule actuellement à " +
vitesse + " km/h."; } public static void main(String[] args) {
Voiture voiture = new Voiture(); System.out.println(voiture);
voiture.accelerer(110); System.out.println(voiture);
voiture.decelerer(20); System.out.println(voiture);
voiture.freiner(); System.out.println(voiture); } }

```

À la ligne 49, le code commence par créer une **instance** de la classe Voiture. Une classe représente une abstraction ou, si vous préférez, un schéma de ce qu'est une Voiture pour notre programme. Évidemment, une voiture peut aussi être définie pas sa couleur, sa marque, son prix, les caractéristiques techniques de son moteur. Mais faisons l'hypothèse que, dans le cadre de notre programme, seule la vitesse aura un intérêt. Voilà pourquoi notre classe Voiture n'est qu'une abstraction du concept de Voiture.

Si dans notre programme, nous voulons interagir avec une voiture nous devons créer une instance de la classe Voiture. Cette instance (que l'on appelle plus simplement un objet) dispose de son propre espace mémoire qui contient son état, c'est-à-dire la liste de ses attributs. Créer une instance d'un objet se fait grâce au mot-clé **new**.

Note

Remarquez l'utilisation des parenthèses avec le mot-clé **new** :

```
voiture voiture = new Voiture();
```

Ces parenthèses sont obligatoires.

En Java, l'opérateur `.` sert à accéder aux attributs ou aux méthodes d'un objet. Donc si on dispose d'une variable *voiture* de type *Voiture*, on peut appeler sa méthode *accelerer* grâce à cet opérateur :

```
voiture.accelerer(90);
```



Aux lignes 51, 54, 57 et 60, nous utilisons la classe [System](#) pour afficher du texte sur la sortie standard. Notez que nous ne créons pas d'instance de la classe [System](#) avec l'opérateur **new**. Il s'agit d'un cas particulier sur lequel nous reviendrons lorsque nous aborderons les méthodes et les attributs de classe. Nous utilisons l'attribut de classe **out** de la classe [System](#) qui représente la sortie standard et nous appelons sa méthode [println](#) qui affiche le texte passé en paramètre suivi d'un saut de ligne. Cependant, nous ne passons pas une chaîne de caractères comme paramètre mais directement une instance de notre classe Voiture. Dans ce cas, la méthode [println](#) appellera la méthode *Voiture.toString()* pour obtenir une représentation textuelle de l'objet.

## Exécuter le programme en ligne de commandes

Dans un terminal, en se rendant dans le répertoire contenant le fichier Java, il est possible de le compiler

```
$ javac voiture.java
```

et de lancer le programme

```
$ java voiture
```

Ce qui affichera sur la sortie suivante :

```
La voiture roule actuellement à 0.0 km/h
La voiture roule actuellement à 110.0 km/h
La voiture roule actuellement à 90.0 km/h
La voiture roule actuellement à 0.0 km/h
```

## Exécuter le programme depuis Eclipse

Il suffit d'ajouter la classe *Voiture* dans le répertoire *src* d'un projet Java dans Eclipse. Ensuite, il faut faire un clic droit sur le nom du fichier dans le *Package Explorer* et choisir *Run as » Java Application*.

Note

Vous pouvez aussi appuyer sur la touche F11 lorsque vous êtes positionné dans le fichier source pour lancer l'exécution de la classe.

## Exercices

Exercice - La classe Ascenseur

On représente un ascenseur dans un système à l'aide d'une classe. La classe Ascenseur doit disposer de la méthode publique `getEtage()` qui retourne le numéro de l'étage auquel se trouve l'ascenseur. Si l'ascenseur est au rez-de-chaussée, la méthode retourne 0.

La classe Ascenseur a également la méthode `appeler(int etage)` qui permet d'appeler l'ascenseur à un étage donné. Cette méthode a pour conséquence d'amener l'ascenseur à l'étage donné.

Implémentez la classe Ascenseur. Ajoutez dans la classe une méthode :

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

Dans la méthode `main`, créez un objet de type `Ascenseur` et appelez ses méthodes. Après chaque appel, affichez sur la sortie standard l'étage où se trouve l'ascenseur.

#### Exercice - La classe `BoiteDeVitesses`

On souhaite créer une classe `BoiteDeVitesses` pour représenter une boîte de vitesses. Cette classe contient les méthodes :

- `changerVitesse()`  
Pour passer à la vitesse supérieure
- `retrograder()`  
Pour passer à la vitesse inférieure
- `getVitesse()`  
Pour donner la vitesse courante (0 représente le point mort)

Implémentez la classe `BoiteDeVitesses`. Ajoutez dans la classe une méthode :

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

Dans la méthode `main`, créez un objet de type `BoiteDeVitesses` pour créer un programme qui passe du point mort à la 5<sup>ème</sup> avant de revenir au point mort. Après chaque changement de vitesse, affichez sur la sortie standard le numéro de la vitesse courante.

## Les types primitifs

Java n'est pas complètement un langage orienté objet dans la mesure où il supporte ce que l'on nomme les *types primitifs*. Chaque type primitif est représenté par un mot-clé :

Français	Anglais	Mot-clé
Booléen	Boolean	<b>boolean</b>
Caractère	Character	<b>char</b>
Entier	Integer	<b>int</b>
Octet	Byte	<b>byte</b>
Entier court	Short integer	<b>short</b>
Entier long	Long integer	<b>long</b>
Nombre à virgule flottante	Float number	<b>float</b>
Nombre à virgule flottante en double précision	Double precision float number	<b>double</b>

Une variable de type primitif représente juste une valeur stockée dans un espace mémoire dont la taille dépend du type. À la différence des langages comme C ou C++, l'espace mémoire occupé par un primitif est fixé par la spécification du langage et non par la machine cible.

Type	Espace mémoire	Signé
<b>boolean</b>	indéterminé	non
<b>char</b>	2 octets (16 bits)	non
<b>int</b>	4 octets (32 bits)	oui
<b>byte</b>	1 octet (8 bits)	oui
<b>short</b>	2 octets (16 bits)	oui
<b>long</b>	8 octets (64 bits)	oui
<b>float</b>	4 octets (32 bits <a href="#">IEEE 754</a> floating point)	oui
<b>double</b>	8 octets (64 bits <a href="#">IEEE 754</a> floating point)	oui

## Le type booléen : boolean

Les variables de type booléen ne peuvent prendre que deux valeurs : **true** ou **false**. Par défaut, un attribut de type **boolean** vaut **false**.

On ne peut utiliser que des opérateurs booléens comme **==**, **!=** et **!** sur des variables de type booléen (pas d'opération arithmétique autorisée).

## Le type caractère : char

Les variables de type **char** sont codées sur 2 octets non signés car la représentation interne des caractères est l'UTF-16. Cela signifie que la valeur va de 0 à  $2^{16} - 1$ . Par défaut, un attribut de type **char** vaut **0** (c'est-à-dire le caractère de terminaison).

Pour représenter un littéral, on utilise l'apostrophe (**simple quote**) :

```
char c = 'a';
```

Même si les caractères ne sont pas des nombres, Java autorise les opérations arithmétiques sur les caractères en se basant sur le code caractère. Cela peut être pratique si l'on veut parcourir l'alphabet par exemple :

```
for (char i = 'a'; i <= 'z'; ++i) {
    // ...
}
```

On peut également affecter un nombre à une variable caractère. Ce nombre représente alors le code caractère :

```
char a = 97; // 97 est le code caractère de la lettre a en UTF-16
```

Affecter une variable de type entier à une variable de type **char** conduit à une erreur de compilation. En effet, le type **char** est un nombre signé sur 2 octets. Pour passer la compilation, il faut transtyper (**cast**) la variable :

```
int i = 97;
char a = (char) i; // cast vers char obligatoire pour la compilation
```

## Les types entiers : byte, short, int, long

Les types entiers diffèrent entre-eux uniquement par l'espace de stockage mémoire qui leur est alloué. Ils sont tous des types signés. Par défaut, un attribut de type **byte**, **short**, **int** ou **long** vaut 0.

La règle de conversion implicite est simple : on peut affecter une variable d'un type à une variable d'un autre type que si la taille mémoire est au moins assez grande.

```
byte b = 1;
short s = 2;
int i = 3;
long l = 4;

// conversion implicite ok
// car la variable à droite de l'expression
// est d'une taille mémoire inférieure
s = b;
i = s;
i = b;
l = b;
l = s;
l = i;
```

Dans tous les autres cas, il faut réaliser un transtypage avec un risque de perte de valeur :

```
b = (byte) s;
s = (short) i;
i = (int) l;
```

Lorsque vous affectez une valeur littérale à une variable, le compilateur contrôlera que la valeur est acceptable pour ce type :

```
byte b = 0;
b = 127; // ok
b = 128; // ko car le type byte accepte des valeurs entre -128 et 127
```

Les valeurs littérales peuvent s'écrire suivant plusieurs bases :

Base	Exemple
2 (binaire)	0b0010 ou 0B0010
8 (octal)	0174
10 (décimal)	129
16 (hexadécimal)	0x12af ou 0X12AF

On peut forcer une valeur littérale à être interprétée comme un entier long en suffixant la valeur par **L** ou **l** :

```
long l = 100L;
```

Pour plus de lisibilité, il est également possible de séparer les milliers par **\_** :

```
long l = 1_000_000;
```

Note

Les opérations arithmétiques entre des valeurs littérales sont effectuées à la compilation. Il est souvent plus lisible de faire apparaître l'opération plutôt que le résultat :

```
int hourInMilliseconds = 60 * 60 * 1000 // plutôt que 3_600_000
```

Danger

La représentation interne des nombres entiers fait qu'il est possible d'aboutir à un dépassement des valeurs maximales ou minimales (*buffer overflow* ou *buffer underflow*) . Il n'est donc pas judicieux d'utiliser ces types pour représenter des valeurs qui peuvent croître ou décroître sur une très grande échelle. Pour ces cas-là, on peut utiliser la classe [BigInteger](#) qui utilise une représentation interne plus complexe.

## Les types à virgule flottante : float, double

Les types **float** et **double** permettent de représenter les nombres à virgule selon le format [IEEE 754](#). Ce format stocke le signe sur un bit puis le nombre sous une forme entière (la mantisse) et l'exposant en base 2 pour positionner la virgule. Par défaut, un attribut de type **float** ou **double** vaut 0.

**float** est dit en simple précision et est codé sur 4 octets (32 bits) tandis que **double** est dit en double précision et est codé sur 8 octets (64 bits).

Il est possible d'ajouter une valeur entière à un type à virgule flottante mais l'inverse nécessite une transtypage (**cast**) avec une perte éventuelle de valeur.

```
int i = 2;
double d = 5.0;
d = d + i;
i = (int) (d + i);
```

Les valeurs littérales peuvent s'écrire avec un **.** pour signifier la virgule et/ou avec une notation scientifique en donnant l'exposant en base 10 :

```
double d1 = .0; // le 0 peut être omis à gauche de la virgule
double d2 = -1.5;
double d3 = 1.5E1; // 1.5 * 10, c'est-à-dire 15.0
double d4 = 0.1234E-15;
```

Une valeur littérale est toujours considérée en double précision. Pour l'affecter à une variable de type **float**, il faut suffixer la valeur par **F** ou **f** :

```
float f = 0.5f;
```

### Danger

La représentation interne des nombres à virgule flottante fait qu'il est possible d'aboutir à des imprécisions de calcul. Il n'est donc pas judicieux d'utiliser ces types pour représenter des valeurs pour lesquelles les approximations de calcul ne sont pas acceptables.

Par exemple, les applications qui réalisent des calculs sur des montants financiers ne devraient **jamais** utiliser des nombres à virgule flottante. Soit il faut représenter l'information en interne toujours en entier (par exemple en centimes d'euro) soit il faut utiliser la classe [BigDecimal](#) qui utilise une représentation interne plus complexe mais sans approximation.

## Les classes enveloppes

Comme les types primitifs ne sont pas des classes, l'API standard de Java fournit également des classes qui permettent d'envelopper la valeur d'un type primitif : on parle de **wrapper classes**.

Type	Classe associée
boolean	<a href="#">java.lang.Boolean</a>
char	<a href="#">java.lang.Character</a>
int	<a href="#">java.lang.Integer</a>
byte	<a href="#">java.lang.Byte</a>
short	<a href="#">java.lang.Short</a>
long	<a href="#">java.lang.Long</a>
float	<a href="#">java.lang.Float</a>
double	<a href="#">java.lang.Double</a>

### Note

Le tableau ci-dessus donne le nom complet des classes, c'est-à-dire en incluant le nom du package (*java.lang*).

Il est possible de créer une instance d'une classe enveloppe soit en utilisant son constructeur soit en utilisant la méthode de classe **valueOf** (il s'agit de la méthode recommandée).

```
Integer i = Integer.valueOf(2);
```

Pour obtenir la valeur enveloppée, on fait appel à la méthode *xxxValue()*, xxx étant le type sous-jacent :

```
Integer i = Integer.valueOf(2);  
int x = 1 + i.intValue();
```

Pourquoi avoir créé ces classes ? Cela permet d'offrir un emplacement facile à mémoriser à des méthodes utilitaires. Par exemple, toutes les classes enveloppes définissent une méthode de classe de la forme *parseXXX* qui permet de convertir une chaîne de caractères en un type primitif :

```
boolean b = Boolean.parseBoolean("true");
byte by = Byte.parseByte("1");
short s = Short.parseShort("1");
int i = Integer.parseInt("1");
long l = Long.parseLong("1");
float f = Float.parseFloat("1");
double d = Double.parseDouble("1");
// enfin presque toutes car Character n'a pas cette méthode
```

Une variable de type d'une des classes enveloppes référence un objet donc elle peut avoir la valeur spéciale **null**. Ce cas permet de signifier l'absence de valeur.

Les classes enveloppes contiennent des constantes pour donner des informations utiles. Par exemple, la classe [java.lang.Integer](#) déclare les constantes [MIN VALUE](#) et [MAX VALUE](#) qui donnent respectivement la plus petite valeur et la plus grande valeur représentables par la primitive associée.

Enfin les classes enveloppes sont conçues pour être non modifiables. Cela signifie que l'on ne peut pas modifier la valeur qu'elles enveloppent après leur création.

## L'autoboxing

Il n'est pas rare dans une application Java de devoir convertir des types primitifs vers des instances de leur classe enveloppe et réciproquement. Afin d'alléger la syntaxe, on peut se contenter d'affecter une variable à une autre et le compilateur se chargera d'ajouter le code manquant. L'opération qui permet de passer d'un type primitif à une instance de sa classe enveloppe s'appelle le **boxing** et l'opération inverse s'appelle **l'unboxing**.

Le code suivant

```
Integer i = 1;
```

est accepté par le compilateur et ce dernier lira à la place

```
Integer i = Integer.valueOf(1); // boxing
```

De même, le code suivant

```
Integer i = 1;
int j = i;
```

est également accepté par le compilateur et ce dernier lira à la place

```
Integer i = Integer.valueOf(1); // boxing
int j = i.intValue(); // unboxing
```

On peut ainsi réaliser des opérations arithmétiques sur des instances de classes enveloppes

```
Integer i = 1;
Integer j = 2;
Integer k = i + j;
```

Il faut bien comprendre que le code ci-dessus manipule en fait des objets et qu'il implique plusieurs opérations de boxing et de unboxing. Si cela n'est pas strictement nécessaire, alors il vaut mieux utiliser des types primitifs.

L'autoboxing fonctionne à chaque fois qu'une affectation a lieu. Il s'applique donc à la déclaration de variable, à l'affectation de variable et au passage de paramètre.

L'autoboxing est parfois difficile à utiliser car il conduit à des expressions qui peuvent être ambiguës. Par exemple, alors que le code suivant utilisant des primitives compile :

```
int i = 1;
float j = i;
```

Ce code faisant appelle à l'autoboxing ne compile pas en l'état :

```
Integer i = 1;
Float j = i; // ERREUR : i est de type Integer
```

Pire, l'autoboxing peut être source de bug. Le plus évident est l'unboxing d'une variable nulle :

```
Integer i = null;
int j = i; // ERREUR : unboxing de null !
```

Une variable de type **Integer** peut être **null**. Dans ce cas, l'unboxing n'est pas possible et aboutira à une erreur (NullPointerException). Si cet exemple est trivial, il peut être beaucoup plus subtil et difficile à comprendre pour un projet de plusieurs centaines (milliers) de lignes de code.

## Les opérateurs

Un opérateur prend un ou plusieurs opérandes et produit une nouvelle valeur. Les opérateurs en Java sont très proches de ceux des langages C et C++ qui les ont inspirés.

### L'opérateur d'affectation

L'affectation est réalisée grâce à l'opérateur `=`. Cet opérateur, copie la valeur du paramètre de droite (appelé *rvalue*) dans le paramètre de gauche (appelé *lvalue*). Java opère donc par copie. Cela signifie que si l'on change plus tard la valeur d'un des opérandes, la valeur de l'autre ne sera pas affectée.

```
int i = 1;
int j = i; // j reçoit la copie de la valeur de i

i = 10; // maintenant i vaut 10 mais j vaut toujours 1
```

Pour les variables de type objet, on appelle ces variables des **handlers** car la variable ne contient pas à proprement parler un objet mais la *référence d'un objet*. On peut dire aussi qu'elle pointe vers la zone mémoire de cet objet. Cela a plusieurs conséquences importantes.

```
Voiture v1 = new Voiture();
Voiture v2 = v1;
```



Dans, l'exemple ci-dessus, **v2** reçoit la copie de l'adresse de l'objet contenue dans **v1**. Donc ces deux variables référencent bien le même objet et nous pouvons le manipuler à travers l'une ou l'autre de ces variables. Si plus loin dans le programme, on écrit :

```
v1 = new voiture();
```

**v1** reçoit maintenant la référence d'un nouvel objet et les variables **v1** et **v2** référencent des instances différentes de **Voiture**. Si enfin, j'écris :

```
v2 = null;
```

Maintenant, la variable **v2** contient la valeur spéciale **null** qui indique qu'elle ne référence rien. Mais l'instance de *Voiture* que la variable **v2** référençait précédemment, n'a pas disparue pour autant. Elle existe toujours quelque part en mémoire. On dit que cette instance n'est plus référencée.

Important

Le passage par copie de la référence vaut également pour les paramètres des méthodes.

Note

= est plus précisément l'opérateur d'initialisation et d'affectation. Pour une variable, l'initialisation se fait au moment de sa déclaration et pour un attribut, au moment de la création de l'objet.

```
// Initialisation  
int a = 1;
```

L'affectation est une opération qui se fait, pour une variable, après sa déclaration et, pour un attribut, après la construction de l'objet.

```
int a;  
// Affectation  
a = 1;
```

## Les opérateurs arithmétiques

Les opérateurs arithmétiques à deux opérandes sont :

*	Multiplication
/	Division
%	Reste de la division
+	Addition
-	Soustraction

La liste ci-dessus est donnée par ordre de précedence. Cela signifie qu'une multiplication est effectuée avant une division.

```
int i = 2 * 3 + 4 * 5 / 2;  
int j = (2 * 3) + ((4 * 5) / 2);
```

Les deux expressions ci-dessus donne le même résultat en Java : 16. Il est tout de même recommandé d'utiliser les parenthèses qui rendent l'expression plus facile à lire.

## Les opérateurs arithmétiques unaires

Les opérateurs arithmétiques unaires ne prennent qu'un seul argument (comme l'indique leur nom), il s'agit de :

<code>expr++</code>	Incrément postfixé
<code>expr--</code>	Décrément postfixé
<code>++expr</code>	Incrément préfixé
<code>--expr</code>	Décrément préfixé
<code>+</code>	Positif
<code>-</code>	Négatif

```
int i = 0;  
i++; // i vaut 1  
++i; // i vaut 2  
--i; // i vaut 1  
  
int j = +i; // équivalent à int j = i;  
int k = -i;
```

### Note

Il y a une différence entre un opérateur postfixé et un opérateur préfixé lorsqu'ils sont utilisés conjointement à une affectation. Pour les opérateurs préfixés, l'incrément ou le décrétement se fait **avant** l'affectation. Pour les opérateurs postfixés, l'incrément ou le décrétement se fait **après** l'affectation.

```
int i = 10;  
j = i++; // j vaudra 10 et i vaudra 11  
  
int k = 10;  
l = ++k; // l vaudra 11 et k vaudra 11
```

## L'opérateur de concaténation de chaînes

Les chaînes de caractères peuvent être concaténées avec l'opérateur `+`. En Java, les chaînes de caractères sont des objets de type [String](#). Il est possible de concaténer un objet de type [String](#) avec un autre type. Pour cela, le compilateur insérera un appel à la méthode `toString` de l'objet ou de la classe enveloppe pour un type primitif.

```
String s1 = "Hello ";
String s2 = s1 + " world";
String s3 = " !";
String s4 = s2 + s3;
```

Note

L'opérateur de concaténation correspond plus à du sucre syntaxique qu'à un véritable opérateur. En effet, il existe la classe [StringBuilder](#) dont la tâche consiste justement à nous aider à construire des chaînes de caractères. Le compilateur remplacera en fait notre code précédent par quelque chose dans ce genre :

```
String s1 = "Hello ";

StringBuilder sb1 = new StringBuilder();
sb1.append(s1)
sb1.append(s2);

String s2 = sb1.toString();
String s3 = " !";

StringBuilder sb2 = new StringBuilder();
sb2.append(s2)
sb2.append(s3);

String s4 = sb2.toString();
```

Note

Concaténer une chaîne de caractères avec une variable nulle ajoute la chaîne « null » :

```
String s1 = "test ";
String s2 = null;
String s3 = s1 + s2; // "test null"
```

## Les opérateurs relationnels

Les opérateurs relationnels produisent un résultat booléen (**true** ou **false**) et permettent de comparer deux valeurs :

<	Inférieur
>	Supérieur
<=	Inférieur ou égal
>=	Supérieur ou égal
==	Égal
!=	Différent

La liste ci-dessus est donnée par ordre de précedence. Les opérateurs <, >, <=, >= ne peuvent s'employer que pour des nombres ou des caractères (**char**).

Les opérateurs `==` et `!=` servent à comparer les valeurs contenues dans les deux variables. Pour des variables de type objet, ces opérateurs **ne comparent pas** les objets entre-eux mais simplement les références contenues dans ces variables.

```
voiture v1 = new Voiture();
voiture v2 = v1;

// true car v1 et v2 contiennent la même référence
boolean resultat = (v1 == v2);
```

Prudence

Les chaînes de caractères en Java sont des **objets** de type [String](#). Cela signifie qu'il ne faut **JAMAIS** utiliser les opérateurs `==` et `!=` pour comparer des chaînes de caractères.

```
String s1 = "une chaîne";
String s2 = "une chaîne";

// sûrement un bug car le résultat est indéterminé
boolean resultat = (s1 == s2);
```

La bonne façon de faire est d'utiliser la méthode [equals](#) pour comparer des objets :

```
String s1 = "une chaîne";
String s2 = "une chaîne";

boolean resultat = s1.equals(s2); // OK
```

## Les opérateurs logiques

Les opérateurs logiques prennent des booléens comme opérandes et produisent un résultat booléen (**true** ou **false**) :

!	Négation
&&	Et logique
	Ou logique

```
boolean b = true;
boolean c = !b // c vaut false

boolean d = b && c; // d vaut false
boolean e = b || c; // e vaut true
```

Les opérateurs `&&` et `||` sont des opérateurs qui n'évaluent l'expression à droite que si cela est nécessaire.

```
ltest() && rtest()
```

Dans l'exemple ci-dessus, la méthode **ltest** est appelée et si elle retourne **true** alors la méthode **rtest()** sera appelée pour évaluer l'expression. Si la méthode **ltest** retourne **false** alors le résultat de l'expression sera **false** et la méthode **rtest** ne sera pas appelée.

```
ltest() || rtest()
```

Dans l'exemple ci-dessus, la méthode **ltest** est appelée et si elle retourne **false** alors la méthode **rtest()** sera appelée pour évaluer l'expression. Si la méthode **ltest** retourne **true** alors le résultat de l'expression sera **true** et la méthode **rtest** ne sera pas appelée.

Si les méthodes des exemples ci-dessus produisent des effets de bord, il est parfois difficile de comprendre le comportement du programme.

Astuce

Il existe en Java les opérateurs **&** et **|** qui forcent l'évaluation de tous les termes de l'expression quel que soit le résultat de chacun d'entre eux.

```
ltest() | ctest() & rtest()
```

Dans l'expression ci-dessus, peu importe la valeur booléenne retournée par l'appel à ces méthodes. Elles seront toutes appelées puis ensuite le résultat de l'expression sera évalué.

## L'opérateur ternaire

L'opérateur ternaire permet d'affecter une valeur suivant le résultat d'une condition.

```
exp booléenne ? valeur si vrai : valeur si faux
```

Par exemple :

```
string s = age >= 18 ? "majeur" : "mineur";  
int code = s.equals("majeur") ? 10 : 20;
```

## Les opérateurs *bitwise*

Les opérateurs *bitwise* permettent de manipuler la valeur des bits d'un entier.

~	Négation binaire
&	Et binaire
^	Ou exclusif (XOR)
	Ou binaire

```
int i = 0b1;

i = 0b10 | i; // i vaut 0b11

i = 0b10 & i; // i vaut 0b10

i = 0b10 ^ i; // i vaut 0b00

i = ~i; // i vaut -1
```

## Les opérateurs de décalage

Les opérateurs de décalage s'utilisent sur des entiers et permettent de déplacer les bits vers la gauche ou vers la droite. Par convention, Java place le bit de poids fort à gauche quelle que soit la représentation physique de l'information. Il est possible de conserver ou non la valeur du bit de poids fort qui représente le signe pour un décalage à droite.

<<	Décalage vers la gauche
>>	Décalage vers la droite avec préservation du signe
>>>	Décalage vers la droite sans préservation du signe

Puisque la machine stocke les nombres en base 2, un décalage vers la gauche équivaut à multiplier par 2 et un décalage vers la droite équivaut à diviser par 2 :

```
int i = 1;
i = i << 1 // i vaut 2
i = i << 3 // i vaut 16
i = i >> 2 // i vaut 4
```

## Le transtypage (cast)

Il est parfois nécessaire de signifier que l'on désire passer d'un type vers un autre au moment de l'affectation. Java étant un langage fortement typé, il autorise par défaut uniquement les opérations de transtypage qui sont sûres. Par exemple : passer d'un entier à un entier long puisqu'il n'y aura de perte de données.

Si on le désire, il est possible de forcer un transtypage en indiquant explicitement le type attendu entre parenthèses :

```
int i = 1;
long l = i; // ok
short s = (short) l; // cast obligatoire
```

L'opération doit avoir un sens. Par exemple, pour passer d'un type d'objet à un autre, il faut que les classes aient un lien d'héritage entre elles.

Prudence

Si Java impose de spécifier explicitement le transtypage dans certaines situations alors c'est qu'il s'agit de situations qui peuvent être problématiques (perte de données possible ou mauvais type d'objet). Il ne faut pas interpréter cela comme une limite du langage : il s'agit peut-être du symptôme d'une erreur de programmation ou d'une mauvaise conception.

Note

Le transtypage peut se faire également par un appel à la méthode [Class.cast](#). Il s'agit d'une utilisation avancée du langage puisqu'elle fait intervenir la notion de réflexivité.

## Opérateur et assignation

Il existe une forme compacte qui permet d'appliquer certains opérateurs et d'assigner le résultat directement à l'opérande de gauche.

Opérateur	Équivalent
<code>+=</code>	<code>a = a + b</code>
<code>-=</code>	<code>a = a - b</code>
<code>*=</code>	<code>a = a * b</code>
<code>/=</code>	<code>a = a / b</code>
<code>%=</code>	<code>a = a % b</code>
<code>&amp;=</code>	<code>a = a &amp; b</code>
<code>^=</code>	<code>a = a ^ b</code>
<code> =</code>	<code>a = a   b</code>
<code>&lt;&lt;=</code>	<code>a = a &lt;&lt; b</code>
<code>&gt;&gt;=</code>	<code>a = a &gt;&gt; b</code>
<code>&gt;&gt;&gt;=</code>	<code>a = a &gt;&gt;&gt; b</code>

À votre avis

```
int i = 100;
i += 1;
i >>= 1;
i /= 2;
i &= ~0;
i %= 20;
```

Quelle est la valeur de i ?

## L'opérateur .

L'opérateur . permet d'accéder aux attributs et aux méthodes d'une classe ou d'un objet à partir d'une référence.

```
String s = "Hello the world";
int length = s.length();
System.out.println("La chaîne de caractères contient " + length + "
caractères");
```

Note

On a l'habitude d'utiliser l'opérateur `.` en plaçant à gauche une variable ou un appel de fonction. Cependant comme une chaîne de caractères est une instance de [String](#), on peut aussi écrire :

```
int length = "Hello the world".length();
```

Lorsqu'on utilise la réflexivité en Java, on peut même utiliser le nom des types primitifs à gauche de l'opérateur `.` pour accéder à la classe associée :

```
String name = int.class.getName();
```

## L'opérateur ,

L'opérateur virgule est utilisé comme séparateur des paramètres dans la définition et l'appel des méthodes. Il peut également être utilisé en tant qu'opérateur pour évaluer séquentiellement une instruction.

```
int x = 0, y = 1, z = 2;
```

Cependant, la plupart des développeurs Java préfèrent déclarer une variable par ligne et l'utilisation de l'opérateur virgule dans ce contexte est donc très rare.

## Exercice

Exercice - La classe BoiteDeVitesses

On souhaite créer une classe BoiteDeVitesses pour représenter une boîte de vitesses. Cette classe contient les méthodes :

- `changerVitesse()`  
Pour passer à la vitesse supérieure
- `retrograder()`  
Pour passer à la vitesse inférieure
- `getVitesse()`  
Pour donner la vitesse courante (0 représente le point mort)
- `toString()`  
Qui retourne une chaîne de caractères (**String**). Si la vitesse vaut 0, la méthode retourne « point mort ». Pour les autres vitesses, la méthode retourne « Vitesse » suivi de la valeur (par exemple « Vitesse 1 »).

Implémentez la classe BoiteDeVitesses. Utilisez obligatoirement les opérateurs unaires pour les méthodes `changerVitesse()` et `retrograder()`. Pour la méthode `toString()`, utilisez l'opérateur ternaire.



Ajoutez une méthode `main` pour créer une boîte de vitesses et tester un appel aux méthodes.

# Les structures de contrôle

Comme la plupart des langages impératifs, Java propose un ensemble de structures de contrôle.

## if-else

L'expression **if** permet d'exécuter un bloc d'instructions uniquement si l'expression booléenne est évaluée à vrai :

```
if (i % 2 == 0) {  
    // instructions à exécuter si i est pair  
}
```

L'expression **if** peut être optionnellement suivie d'une expression **else** pour les cas où l'expression est évaluée à faux :

```
if (i % 2 == 0) {  
    // instructions à exécuter si i est pair  
} else {  
    // instructions à exécuter si i est impair  
}
```

L'expression **else** peut être suivie d'une nouvelle instruction **if** afin de réaliser des choix multiples :

```
if (i % 2 == 0) {  
    // instructions à exécuter si i pair  
} else if (i > 10) {  
    // instructions à exécuter si i est impair et supérieur à 10  
} else {  
    // instructions à exécuter dans tous les autres cas  
}
```

Note

Si le bloc d'instruction d'un **if** ne comporte qu'une seule instruction, alors les accolades peuvent être omises :

```
if (i % 2 == 0)  
    i++;
```

Cependant, beaucoup de développeurs Java préfèrent utiliser systématiquement les accolades.

## return

**return** est un mot clé permettant d'arrêter immédiatement le traitement d'une méthode et de retourner la valeur de l'expression spécifiée après ce mot-clé. Si la méthode ne retourne pas de valeur (**void**), alors on utilise le mot-clé **return** seul. L'exécution d'un **return** entraîne la fin d'une structure de contrôle.

```
if (i % 2 == 0) {  
    return 0;  
}
```

Écrire des instructions immédiatement après une instruction **return** n'a pas de sens puisqu'elles ne seront jamais exécutées. Le compilateur Java le signalera par une erreur *unreachable code*.

```
if (i % 2 == 0) {  
    return 0;  
    i++; // Erreur de compilation : unreachable code  
}
```

## while

L'expression *while* permet de définir un bloc d'instructions à répéter tant que l'expression booléenne est évaluée à vrai.

```
while (i % 2 == 0) {  
    // instructions à exécuter tant que i est pair  
}
```

L'expression booléenne est évaluée au départ et après chaque exécution du bloc d'instructions.

Note

Si le bloc d'instruction d'un **while** ne comporte qu'une seule instruction, alors les accolades peuvent être omises :

```
while (i % 2 == 0)  
    // instruction à exécuter tant que i est pair
```

Cependant, beaucoup de développeurs Java préfèrent utiliser systématiquement les accolades.

## do-while

Il existe une variante de la structure précédente, nommée **do-while** :

```
do {  
    // instructions à exécuter  
} while (i % 2 == 0);
```

Dans ce cas, le bloc d'instruction est exécuté une fois puis l'expression booléenne est évaluée. Cela signifie qu'avec un **do-while**, le bloc d'instruction est exécuté au moins une fois.

## for

Une expression **for** permet de réaliser une itération. Elle commence par réaliser une initialisation puis évalue une expression booléenne. Tant que cette expression booléenne est évaluée à vrai, le bloc d'instructions est exécuté et un incrément est appelé.

```
for (initialisation; expression booléenne; incrément) {
    bloc d'instructions
}
for (int i = 0; i < 10; ++i) {
    // instructions
}
```

Note

il n'est pas possible d'omettre l'initialisation, l'expression booléenne ou l'incrément dans la déclaration d'une expression *for*. Par contre, il est possible de les laisser vides.

```
int i = 0;
for (; i < 10; ++i) {
    // instructions
}
```

Il est ainsi possible d'écrire une expression *for* sans condition de sortie, la fameuse boucle infinie :

```
for (;;) {
    // instructions à exécuter à l'infini
}
```

Note

Si le bloc d'instruction d'un **for** ne comporte qu'une seule instruction, alors les accolades peuvent être omises :

```
for (int i = 0; i < 10; ++i)
    // instruction à exécuter
```

Cependant, beaucoup de développeurs Java préfèrent utiliser systématiquement les accolades.

## for amélioré

Il existe une forme améliorée de l'expression *for* (souvent appelée *for-each*) qui permet d'exprimer plus succinctement un parcours d'une collection d'éléments.

```
for (int i : maCollection) {
    // instructions à exécuter
}
```

Pour que cette expression compile, il faut que la variable désignant la collection à droite de : implémente le type [Iterable](#) ou qu'il s'agisse d'un tableau. Il faut également que la variable à gauche de : soit compatible pour l'assignation d'un élément de la collection.

```
short arrayOfShort[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

for (int k : arrayOfShort) {
    System.out.println(k);
}
```

# break-continue

Pour les expressions **while**, **do-while**, **for** permettant de réaliser des itérations, il est possible de contrôler le comportement à l'intérieur de la boucle grâce aux mots-clés **break** et **continue**.

**break** quitte la boucle sans exécuter le reste des instructions.

```
int k = 10;
for (int i = 1 ; i < 10; ++i) {
    k *= i
    if (k > 200) {
        break;
    }
}
```

**continue** arrête l'exécution de l'itération actuelle et commence l'exécution de l'itération suivante.

```
for (int i = 1 ; i < 10; ++i) {
    if (i % 2 == 0) {
        continue;
    }
    System.out.println(i);
}
```

## libellé

Il est possible de mettre un libellé avant une expression **for** ou **while**. La seule et unique raison d'utiliser un libellé est le cas d'une itération imbriquée dans une autre itération. Par défaut, **break** et **continue** n'agissent que sur le bloc d'itération dans lequel ils apparaissent. En utilisant un libellé, on peut arrêter ou continuer sur une itération de niveau supérieur :

```
int m = 0;

boucleDeCalcul:
for (int i = 0; i < 10; ++i) {
    for (int k = 0; k < 10; ++k) {
        m += i * k;
        if (m > 500) {
            break boucleDeCalcul;
        }
    }
}

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

Dans l'exemple ci-dessus, *boucleDeCalcul* est un libellé qui permet de signifier que l'instruction **break** porte sur la boucle de plus haut niveau. Son exécution stoppera donc l'itération des deux boucles et passera directement à l'affichage du résultat sur la sortie standard.

## switch

Une expression **switch** permet d'effectuer une sélection parmi plusieurs valeurs.

```

switch (s) {
  case "valeur 1":
    // instructions
    break;
  case "valeur 2":
    // instructions
    break;
  case "valeur 3":
    // instructions
    break;
  default:
    // instructions
}

```

**switch** évalue l'expression entre parenthèses et la compare dans l'ordre avec les valeurs des lignes **case**. Si une est identique alors il commence à exécuter la ligne d'instruction qui suit. Attention, un **case** représente un point à partir duquel l'exécution du code commencera. Si on veut isoler chaque cas, il faut utiliser une instruction **break**. Au contraire, l'omission de l'instruction **break** peut être pratique si on veut effectuer le même traitement pour un ensemble de cas :

```

switch (c) {
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
  case 'y':
    // instruction pour un voyelle
    break;
  default:
    // instructions pour une consonne
}

```

On peut ajouter une cas **default** qui servira de point d'exécution si aucun **case** ne correspond.

Note

Par convention, on place souvent le cas **default** à la fin. Cependant, il agit plus comme un libellé indiquant la ligne à laquelle doit commencer l'exécution du code. Il peut donc être placé n'importe où :

```

switch (c) {
  default:
    // instructions pour une consonne
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
  case 'y':
    // instructions pour les consonnes et les voyelles
}

```

## Prudence

En Java, le type d'expression accepté par un **switch** est limité. Un **switch** ne compile que pour un type primitif, une énumération ou une chaîne de caractères.

# Exercices

## Exercice - La classe BoiteDeVitesses

On souhaite créer une classe BoiteDeVitesses pour représenter une boîte de vitesses. Cette classe contient les méthodes :

- `changerVitesse()`  
Pour passer à la vitesse supérieure. Il n'est pas possible d'aller au delà de la cinquième vitesse (dans ce cas, un appel à la méthode doit être sans effet).
- `retrograder()`  
Pour passer à la vitesse inférieure. Il n'est pas possible de rétrograder en dessous de la première (dans ce cas, un appel à la méthode doit être sans effet).
- `setPointMort()`  
Pour aller au point mort.
- `getVitesse()`  
Pour donner la vitesse courante (0 représente le point mort).
- `toString()`  
Qui retourne une chaîne de caractères (**String**). Si la vitesse vaut 0, la méthode retourne « point mort ». Pour les autres vitesses, la méthode retourne respectivement : « première », « seconde », « troisième », « quatrième », « cinquième ».

Ajoutez une méthode `main` pour créer une boîte de vitesses et tester un appel aux méthodes.

Testez notamment que les contraintes pour les méthodes `changerVitesse()` et `retrograder()` sont bien implémentées.

## Exercice - Détecteur de température

On souhaite implémenter une classe `DetecteurTemperature`. Cette classe recevra des informations sur la température et sera capable de déduire si le niveau de température est normal, élevé ou critique :

Température	Niveau
$t \leq 30$	normal
$30 < t < 38$	élevé
$t \geq 38$	critique

La classe `DetecteurTemperature` possède les méthodes suivantes :

- `setTemperature(float t)`  
Pour positionner la température actuelle.
- `getNiveau()`

Qui retourne le niveau de la température actuelle sous la forme d'une chaîne de caractères :  
« normal », « élevé », « critique »

- `isNiveauAtteint(String niveau)`

Qui retourne `true` ou `false` si le niveau courant est au moins le niveau donné en paramètre.

Ajouter la méthode `main` suivante pour tester votre classe :

```
public static void main(String[] args) {
    DetecteurTemperature detecteurTemperature = new DetecteurTemperature();

    for(float temperature = 25; temperature < 45; temperature += 2) {
        detecteurTemperature.setTemperature(temperature);
        System.out.println("Température actuelle = " + temperature);
        System.out.println("Niveau actuel = " +
            detecteurTemperature.getNiveau());
        System.out.println("Niveau normal atteint = " +
            detecteurTemperature.isNiveauAtteint("normal"));
        System.out.println("Niveau élevé atteint = " +
            detecteurTemperature.isNiveauAtteint("élevé"));
        System.out.println("Niveau critique atteint = " +
            detecteurTemperature.isNiveauAtteint("critique"));
        System.out.println("#####");
    }
}
```

Exercice - Implémenter le code de César

Implémenter le [code de César](#) en utilisant le code ci-dessous. Cet algorithme de chiffrement classique, utilisé par Jules César dans ses correspondances secrètes, se base sur le principe du décalage. Une lettre est remplacée par son équivalent dans l'alphabet en effectuant un décalage. Le programme ci-dessous utilise un décalage de 23. Cela donne comme équivalence :

<b>a → xb → yc → zd → ae →</b> <b>bf → cg → dh → ei → f</b>	<b>j → gk → hl → im → jn → ko</b> <b>→ lp → mq → nr → o</b>	<b>s → pt → qu → rv → sw</b> <b>→ tx → uy → vz → w</b>

Ainsi la phrase :

In cryptography, a Caesar cipher is one of the simplest and most widely known encryption techniques.

sera encodée avec un décalage de 23 comme ceci :

Fk zovmqldoxmev, x Zxbpxo zfmebo fp lkb lc qeb pfjmibpq xka jlpq tfabiv hkltk bkbzovmqflk qbzekfnrbp.

Le programme suivant vérifie que le résultat du chiffrement est bien conforme au ce qui est attendu. Dans ce cas, il affichera `true` sur la sortie standard.

```
/**
 * Un exemple d'implémentation du Code César.
 */
```

```

* Pour plus d'info, vous pouvez consulter
* l'

```

Décommentez le code de chaque étape dans la méthode *main* au fur et à mesure que vous implémentez la méthode *encrypte* afin de vérifier que votre algorithme fonctionne correctement.



## Indication

On peut obtenir un tableau de caractères à partir d'une chaîne avec la méthode [toCharArray](#). S'il n'est pas possible de parcourir les éléments d'une chaîne de caractères avec un for amélioré, on peut facilement parcourir le tableau de caractères :

```
String helloWorld = "Hello world!";
for (char c : helloWorld.toCharArray()) {
    // ...
}
```

Il est possible de créer une nouvelle chaîne de caractères à partir d'un tableau :

```
char[] tableau = "Hello".toCharArray();
String chaine = new String(tableau);
```

# Les tableaux

Les tableaux représentent des collections de valeurs ou d'objets. En Java, les tableaux sont eux-mêmes des objets. Donc une variable de type tableau peut avoir la valeur **null**. Une variable de type tableau se déclare en ajoutant des crochets à la suite du type :

```
int[] tableau;
```

Il est également possible de placer les crochets après le nom de la variable :

```
int tableau[];
```

## Initialisation

Il est possible d'initialiser une variable de type tableau à partir d'une liste fixe délimitée par des accolades.

```
int[] tableauEntier = {1, 2, 3, 4, 5};
String[] tableauChaine = {"Bonjour", "le", "monde"};
```

## Création avec new

Les tableaux étant des objets, il est également possible de les créer avec le mot-clé **new**.

```
int[] tableauEntier = new int[] {1, 2, 3, 4};
String[] tableauChaine = new String[] {"Bonjour", "le", "monde"};
```

Si on ne souhaite pas donner de valeurs d'initialisation pour les éléments du tableau, il suffit d'indiquer uniquement le nombre d'éléments du tableau entre crochets.

```
int[] tableauEntier = new int[5];
String[] tableauChaine = new String[3];
```

Dans ce cas, les éléments d'un tableau sont tout de même initialisés avec une valeur par défaut (comme pour un attribut) :

Type	Valeur d'initialisation
boolean	false
char	'\0'
byte	0
short	0
int	0
long	0
float	0.0
double	0.0
référence d'objet	null

La taille du tableau peut être donnée par une constante, une expression ou une variable.

```
int t = 6;
int[] tableau = new int[t * t * 2];
```

Par contre, la taille d'un tableau est donnée à sa création et ne peut plus être modifiée. Il n'est donc pas possible d'ajouter ou d'enlever des éléments à un tableau. Dans ce cas, il faut créer un nouveau tableau avec la taille voulue et copier le contenu du tableau d'origine vers le nouveau tableau.

Un tableau dispose de l'attribut **length** permettant de connaître sa taille. L'attribut **length** ne peut pas être modifié.

```
int t = 6;
int[] tableau = new int[t * t * 2];
System.out.println(tableau.length); // 72
```

Note

Il est tout à fait possible de créer un tableau vide, c'est-à-dire avec une taille de zéro.

```
int[] tableau = new int[0];
```

Par contre, donner une taille négative est autorisé par le compilateur mais aboutira à une erreur d'exécution avec une exception de type [java.lang.NegativeArraySizeException](https://docs.oracle.com/javase/7/docs/api/java/lang/NegativeArraySizeException.html).

## Accès aux éléments d'un tableau

L'accès aux éléments d'un tableau se fait en donnant l'indice d'un élément entre crochets. Le premier élément d'un tableau a l'indice **0**. Le dernier élément d'un tableau a donc comme indice la taille du tableau moins un.

```
int[] tableau = {1, 2, 3, 4, 5};

int premierElement = tableau[0];
int dernierElement = tableau[tableau.length - 1];

System.out.println(premierElement); // 1
System.out.println(dernierElement); // 5

for (int i = 0, j = tableau.length - 1; i < j; ++i, --j) {
    int tmp = tableau[j];
    tableau[j] = tableau[i];
    tableau[i] = tmp;
}
```

Comme le montre l'exemple précédent, il est bien sûr possible de parcourir un tableau à partir d'un indice que l'on fait varier à l'aide d'une boucle **for**. Mais il est également possible de parcourir tous les éléments d'un tableau avec un **for** amélioré.

```
int[] tableau = {1, 2, 3, 4, 5};

for (int v : tableau) {
    System.out.println(v);
}
```

L'utilisation d'un **for** amélioré est préférable lorsque cela est possible. Par contre, il n'est pas possible avec un **for** amélioré de connaître l'indice de l'élément courant.

Si le programme tente d'accéder à un indice de tableau trop grand (ou un indice négatif), une erreur de type [java.lang.ArrayIndexOutOfBoundsException](#) survient.

```
int[] tableau = {1, 2, 3, 4, 5};
int value = tableau[1000]; // ERREUR À L'EXÉCUTION
```

## Tableau multi-dimensionnel

Il est possible d'initialiser un tableau à plusieurs dimensions.

```
int[][] tableauDeuxDimensions = {{1, 2}, {3, 4}};

int[][][] tableauTroisDimensions = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}};

System.out.println(tableauDeuxDimensions[0][1]);
System.out.println(tableauTroisDimensions[0][1][0]);
```

Il est également possible de créer un tableau multi-dimensionnel avec le mot-clé **new**.

```
int[][] tableauDeuxDimensions = new int[2][10];
int[][][] tableauTroisDimensions = new int[2][10][5];
```

Il n'existe pas réellement de type tableau multi-dimensionnel. Le compilateur le traite comme un tableau de tableaux. Il est donc autorisé de déclarer des tableaux sans préciser les dimensions au delà de la première et d'affecter ensuite des tableaux à chaque valeur. Ces tableaux peuvent d'ailleurs avoir des tailles différentes.

```
int[][] tableauDeuxDimensions = new int[2][];  
  
tableauDeuxDimensions[0] = new int[10];  
tableauDeuxDimensions[1] = new int[5];
```

## Conversion en chaîne de caractères

Si vous affichez un tableau sur la sortie standard, vous serez certainement surpris.

```
int[] tableau = {1, 2, 3, 4, 5};  
System.out.println(tableau);
```

La code précédent affichera sur la sortie standard quelque chose comme ceci :

```
[I@ee7d9f1
```

Cela peut sembler un bug mais il n'en est rien. En fait, la conversion d'un objet en chaîne de caractères affiche par défaut son type suivi du caractère @ suivi du code de hachage de l'objet. Normalement le type d'un objet correspond au nom de sa classe. Mais le type d'un tableau est noté `[` suivi du type des éléments du tableau (`I` indique le type primitif `int`).

Pour obtenir une chaîne de caractères donnant le contenu du tableau, il faut utiliser la classe outil [java.util.Arrays](#) qui contient des méthodes de classe `toString` adaptées pour les tableaux.

```
int[] tableau = {1, 2, 3, 4, 5};  
System.out.println(java.util.Arrays.toString(tableau));
```

Note

Pour les tableaux multi-dimensionnels, vous pouvez utiliser la méthode `[java.util.Arrays.deepToString(Object)]`.

## Égalité de deux tableaux

En Java, il n'est pas possible d'utiliser l'opérateur `==` pour comparer deux objets. En effet, cet opérateur compare la référence des variables. Cela signifie qu'il indique **true** uniquement si les deux variables référencent le même objet.

```
int[] tableau1 = {1, 2, 3, 4, 5};  
int[] tableau2 = {1, 2, 3, 4, 5};  
  
System.out.println(tableau1 == tableau1); // true  
System.out.println(tableau1 == tableau2); // false
```

Pour comparer deux objets, il faut utiliser la méthode **equals**. Les tableaux en Java disposent de la méthode **equals**, malheureusement, elle a exactement le même comportement que l'utilisation de l'opérateur `==`.

```
int[] tableau1 = {1, 2, 3, 4, 5};
int[] tableau2 = {1, 2, 3, 4, 5};

System.out.println(tableau1.equals(tableau1)); // true
System.out.println(tableau1.equals(tableau2)); // false
```

La classe util [java.util.Arrays](#) fournit des méthodes de classe **equals** pour comparer des tableaux en comparant un à un leurs éléments.

```
int[] tableau1 = {1, 2, 3, 4, 5};
int[] tableau2 = {1, 2, 3, 4, 5};

System.out.println(java.util.Arrays.equals(tableau1, tableau1)); // true
System.out.println(java.util.Arrays.equals(tableau1, tableau2)); // true
```

Il est également possible de comparer des tableaux d'objets. Dans ce cas, la comparaison des éléments se fait en appelant la méthode **equals** de chaque objet. La méthode **equals** possède la signature suivante :

```
public boolean equals(Object obj) {
    // ...
}
```

Par exemple, la classe [java.lang.String](#) fournit une implémentation de la méthode **equals**. Il est donc possible de comparer des tableaux de chaînes de caractères.

```
String[] tableau1 = {"premier", "deuxième", "troisième", "quatrième"};
String[] tableau2 = {"premier", "deuxième", "troisième", "quatrième"};

System.out.println(java.util.Arrays.equals(tableau1, tableau2)); // true
```

Note

Pour les tableaux multi-dimensionnels, vous pouvez utiliser la méthode [java.util.Arrays.deepEquals\(Object\[\], Object\[\]\)](#).

## Tri & recherche

Le tri et la recherche sont des opérations courantes sur des tableaux de valeurs. La classe util [java.util.Arrays](#) offrent un ensemble de méthodes de classe pour nous aider dans ces opérations.

Tout d'abord, [java.util.Arrays](#) fournit plusieurs méthodes **sort**. Celles prenant un tableau de primitives en paramètre trient selon l'ordre naturel des éléments.

```
int[] tableau = {1, 5, 4, 3, 2};
java.util.Arrays.sort(tableau);
System.out.println(java.util.Arrays.toString(tableau));
```

Il est également possible de trier certains tableaux d'objets. Par exemple, il est possible de trier des tableaux de chaînes de caractères.

```
String[] tableau = {"premier", "deuxième", "troisième", "quatrième"};
java.util.Arrays.sort(tableau);
System.out.println(java.util.Arrays.toString(tableau));
```

#### Note

La méthode `[java.util.Arrays.sort(Object)]` permet de trier des tableaux d'objets dont la classe implémente l'interface [java.lang.Comparable](#).

[java.util.Arrays](#) fournit des méthodes **binarySearch** qui implémentent l'algorithme de recherche binaire. Ces méthodes attendent comme paramètres un tableau et une valeur compatible avec le type des éléments du tableau. Ces méthodes retournent l'index de la valeur trouvée. Si la valeur n'est pas dans le tableau, alors ces méthodes retournent un nombre négatif. La valeur absolue de ce nombre correspond à l'index auquel la valeur aurait dû se trouver plus un.

```
int[] tableau = {10, 20, 30, 40, 50};
System.out.println(java.util.Arrays.binarySearch(tableau, 20)); // 1
System.out.println(java.util.Arrays.binarySearch(tableau, 45)); // -5
```

#### Avertissement

L'algorithme de recherche binaire ne fonctionne correctement que pour un tableau trié.

## Copie d'un tableau

Comme il n'est pas possible de modifier la taille d'un tableau, la copie peut s'avérer une opération utile. [java.util.Arrays](#) fournit des méthodes de classe *copyOf* et *copyOfRange* pour réaliser des copies de tableaux.

```
int[] tableau = {1, 2, 3, 4, 5};

int[] nouveauTableau = java.util.Arrays.copyOf(tableau, tableau.length - 1);
System.out.println(java.util.Arrays.toString(nouveauTableau)); // [1, 2, 3, 4]

nouveauTableau = java.util.Arrays.copyOf(tableau, tableau.length + 1);
System.out.println(java.util.Arrays.toString(nouveauTableau)); // [1, 2, 3, 4, 5, 0]

nouveauTableau = java.util.Arrays.copyOfRange(tableau, 2, tableau.length);
System.out.println(java.util.Arrays.toString(nouveauTableau)); // [3, 4, 5]

nouveauTableau = java.util.Arrays.copyOfRange(tableau, 2, 3);
System.out.println(java.util.Arrays.toString(nouveauTableau)); // [3]
```

Pour réaliser une copie, il existe également la méthode [java.lang.System.arraycopy](#). Contrairement aux précédentes, cette méthode ne crée pas de nouveau tableau, elle copie d'un tableau existant vers un autre tableau existant.

```
int[] tableau = {1, 2, 3, 4, 5};
int[] destination = new int[3];

/* Les paramètres attendus sont :
 * - le tableau source
 * - l'index de départ dans le tableau source
 * - le tableau destination
 * - l'index de départ dans le tableau destination
 * - le nombre d'éléments à copier
 */
System.arraycopy(tableau, 1, destination, 0, destination.length);
System.out.println(java.util.Arrays.toString(destination)); // [2, 3, 4]
```

## Typage d'un tableau

Un tableau est un objet. Cela implique qu'il respecte les règles de typage du langage. Ainsi on ne peut mettre dans un tableau que des valeurs qui peuvent être affectées au type des éléments

```
String[] tableau = new String[10];
tableau[9] = "Bonjour"; // OK
tableau[8] = new Voiture(); // ERREUR DE COMPILATION
```

De plus, les tableaux peuvent être affectés à des variables dont le type correspond à un tableau d'éléments de type parent.

```
Integer[] tableau = {1, 2, 3, 4};
Number[] tableauNumber = tableau;
```

Pour l'exemple précédent, il faut se rappeler la classe enveloppe [java.lang.Integer](#) hérite de la classe [java.lang.Number](#). Cependant, un tableau conserve son type d'origine : si on affecte une valeur dans un tableau, elle doit non seulement être compatible avec le type de la variable (pour passer la compilation) mais aussi être compatible avec le type de tableau à l'exécution. Si cette dernière condition n'est pas remplie, on obtiendra une erreur de type [java.lang.ArrayStoreException](#) au moment de l'exécution.

```
Integer[] tableau = {1};
Number[] tableauNumber = tableau;
tableauNumber[0] = Float.valueOf(2.3f); // ERREUR À L'EXÉCUTION
```

## Conversion d'un tableau en liste

La plupart des API Java utilisent des [collections](#) plutôt que des tableaux. Pour transformer un tableau d'objets en liste, on utilise la méthode [java.util.Arrays.asList](#). La liste obtenue possède une taille fixe. Par contre le contenu de la liste est modifiable, et toute modification des éléments de cette liste sera répercutée sur le tableau.

```
String[] tableau = {"Bonjour", "le", "monde"};
java.util.List<String> liste = java.util.Arrays.asList(tableau);

liste.set(0, "Hello");
liste.set(1, "the");
liste.set(2, "world");

// Le tableau a été modifié à travers la liste
system.out.println(java.util.Arrays.toString(tableau)); // [Hello, the, world]
```

# Attributs & méthodes

Dans ce chapitre, nous allons revenir sur la déclaration d'une classe en Java et détailler les notions d'attributs et de méthodes.

## Les attributs

Les attributs représentent l'état interne d'un objet. Nous avons vu précédemment qu'un attribut a une portée, un type et un identifiant. Il est déclaré de la façon suivante dans le corps de la classe :

```
[portée] [type] [identifiant];
public class Voiture {

    public String marque;
    public float vitesse;

}
```

La classe ci-dessus ne contient que des attributs, elle s'apparente à une simple structure de données. Il est possible de créer une instance de cette classe avec l'opérateur **new** et d'accéder aux attributs de l'objet créé avec l'opérateur . :

```
Voiture v = new Voiture();
v.marque = "DeLorean";
v.vitesse = 88.0f;
```

## La portée

Jusqu'à présent, nous avons vu qu'il existe deux portées différentes : **public** et **private**. Java est un langage qui supporte l'encapsulation de données. Cela signifie que lorsque nous créons une classe nous avons le choix de laisser accessible ou non les attributs et les méthodes au reste du programme.

Pour l'instant nous distinguerons les portées :

- **public**

Signale que l'attribut est visible de n'importe quelle partie de l'application.

- **private**

Signale que l'attribut n'est accessible que par l'objet lui-même ou par un objet du même type. Donc seules les méthodes de la classe déclarant cet attribut peuvent accéder à cet attribut.



Lorsque nous parlerons de l'encapsulation et du principe du *ouvert/fermé*, nous verrons qu'il est très souvent préférable qu'un attribut ait une portée *private*.

## L'initialisation

En Java, on peut indiquer la valeur d'initialisation d'un attribut pour chaque nouvel objet.

```
public class Voiture {  
  
    public String marque = "DeLorean";  
    public float vitesse = 88.0f;  
  
}
```

En fait, un attribut possède nécessairement une valeur par défaut qui dépend de son type :

Type	Valeur par défaut
boolean	false
char	'\0'
byte	0
short	0
int	0
long	0
float	0.0
double	0.0
référence d'objet	null

Donc, écrire ceci :

```
public class Voiture {  
  
    public String marque;  
    public float vitesse;  
  
}
```

ou ceci

```
public class Voiture {  
  
    public String marque = null;  
    public float vitesse = 0.0f;  
  
}
```

est strictement identique en Java.

## Attributs finaux

Un attribut peut être déclaré comme **final**. Cela signifie qu'il n'est plus possible d'affecter une valeur à cet attribut une fois qu'il a été initialisé. Dans ce cas, le compilateur exige que l'attribut soit initialisé *explicitement*.

```
public class Voiture {  
  
    public String marque;  
    public float vitesse;  
    public final int nombreDeRoues = 4;  
  
}
```

L'attribut `Voiture.nombreDeRoues` sera initialisé avec la valeur 4 pour chaque instance et ne pourra plus être modifié.

```
Voiture v = new Voiture();  
v.nombreDeRoues = 5; // ERREUR DE COMPILATION
```

Prudence

**final** porte sur l'attribut et empêche sa modification. Par contre si l'attribut est du type d'un objet, il est possible de modifier l'état de cet objet.

Pour une application d'un concessionnaire automobile, nous pouvons créer un objet *Facture* qui contient un attribut de type *Voiture* et le déclarer **final**.

```
public class Facture {  
  
    public final Voiture voiture = new Voiture();  
  
}
```

Sur une instance de *Facture*, on ne pourra plus modifier la référence de l'attribut *voiture* par contre, on pourra toujours modifier les attributs de l'objet référencé

```
Facture facture = new Facture();  
facture.voiture.marque = "DeLorean"; // OK  
facture.voiture = new Voiture() // ERREUR DE COMPILATION
```

## Attributs de classe

Jusqu'à présent, nous avons vu comment déclarer des attributs d'objet. C'est-à-dire que chaque instance d'une classe aura ses propres attributs avec ses propres valeurs représentant l'état interne de l'objet et qui peuvent évoluer au fur et à mesure de l'exécution de l'application.

Mais il est également possible de créer des *attributs de classe*. La valeur de ces attributs est partagée par l'ensemble des instances de cette classe. Cela signifie que si on modifie la valeur d'un attribut de classe dans un objet, la modification sera visible dans les autres objets. Cela signifie également que cet attribut existe au niveau de la classe et est donc accessible même si on ne crée aucune instance de cette classe.

Pour déclarer un attribut de classe, on utilise le mot-clé **static**.

```
public class Voiture {  
  
    public static int nombreDeRoues = 4;  
    public String marque;  
    public float vitesse;  
  
}
```

Dans l'exemple ci-dessus, l'attribut *nombreDeRoues* est maintenant un attribut de classe. C'est une façon de suggérer que toutes les voitures de notre application ont le même nombre de roues. Cette caractéristique appartient donc à la classe plutôt qu'à chacune de ses instances. Il est donc possible d'accéder directement à cet attribut depuis la classe :

```
System.out.println(Voiture.nombreDeRoues);
```

Notez que dans l'exemple précédent, [out](#) est également un attribut de la classe [System](#). Si vous vous rendez sur la documentation de cette classe, vous constaterez que [out](#) est déclaré comme **static** dans cette classe. Il s'agit d'une autre utilisation des attributs de classe : lorsqu'il n'existe qu'une seule instance d'un objet pour toute une application, cette instance est généralement accessible grâce à un attribut **static**. C'est une des façons d'implémenter le design pattern [singleton](#) en Java. Dans notre exemple, [out](#) est l'objet qui représente la sortie standard de notre application. Cet objet est unique pour toute l'application et nous n'avons pas à le créer car il existe dès le lancement.

Si le programme modifie un attribut de classe, alors la modification est visible depuis toutes les instances :

```
Voiture v1 = new Voiture();  
Voiture v2 = new Voiture();  
  
System.out.println(v1.nombreDeRoues); // 4  
System.out.println(v2.nombreDeRoues); // 4  
  
// modification d'un attribut de classe  
v1.nombreDeRoues = 5;  
  
Voiture v3 = new Voiture();  
  
System.out.println(v1.nombreDeRoues); // 5  
System.out.println(v2.nombreDeRoues); // 5  
System.out.println(v3.nombreDeRoues); // 5
```

Le code ci-dessus, même s'il est parfaitement correct, peut engendrer des difficultés de compréhension. Si on ne sait pas que *nombreDeRoues* est un attribut de classe, on peut le modifier en pensant que cela n'aura pas d'impact sur les autres instances. C'est notamment pour cela que Eclipse émet un avertissement si on accède ou si on modifie un attribut de classe à travers un objet. Même si l'effet est identique, il est plus lisible d'accéder à un tel attribut à travers le nom de la classe uniquement :

```
System.out.println(Voiture.nombreDeRoues); // 4

Voiture.nombreDeRoues = 5;

System.out.println(Voiture.nombreDeRoues); // 5
```

## Attributs de classe finaux

Il n'existe pas de mot-clé pour déclarer une constante en Java. Même si **const** est un mot-clé, il n'a aucune signification dans le langage. On utilise donc la combinaison des mots-clés **static** et **final** pour déclarer une constante. Par convention, pour les distinguer des autres attributs, on écrit leur nom en majuscules et les mots sont séparés par \_.

```
public class Voiture {

    public static final int NOMBRE_DE_ROUES = 4;
    public String marque;
    public float vitesse;

}
```

Prudence

Rappelez-vous que si l'attribut référence un objet, **final** n'empêche pas d'appeler des méthodes qui vont modifier l'état interne de l'objet. On ne peut vraiment parler de constantes que pour les attributs de type primitif.

## Les méthodes

Les méthodes permettent de définir le comportement des objets. nous avons vu précédemment qu'une méthode est définie par sa **signature** qui spécifie sa portée, son type de retour, son nom et ses paramètres entre parenthèses. La signature est suivie d'un bloc de code que l'on appelle le **corps** de méthode.

```
[portée] [type de retour] [identifiant] ([liste des paramètres]) {
    [code]
}
```

Dans ce corps de méthode, il est possible d'avoir accès au attribut de l'objet. Si la méthode modifie la valeur des attributs de l'objet, elle a un *effet de bord* qui change l'état interne de l'objet. C'est le cas dans l'exemple ci-dessous pour la méthode *accélérer* :

```
public class Voiture {

    private float vitesse;

    /**
     * @return La vitesse en km/h de la voiture
     */
    public float getvitesse() {
        return vitesse;
    }

}
```

```
/**
 * Pour accélérer la voiture
 * @param deltaVitesse Le vitesse supplémentaire
 */
public void accelerer(float deltaVitesse) {
    vitesse = vitesse + deltaVitesse;
}
}
```

Il est possible de créer une instance de la classe ci-dessus avec l'opérateur **new** et d'exécuter les méthodes de l'objet créé avec l'opérateur . :

```
Voiture v = new Voiture();
v.accelerer(88.0f);
```

## La portée

Comme pour les attributs, les méthodes ont une portée, c'est-à-dire que le développeur de la classe peut décider si une méthode est accessible ou non au reste du programme. Pour l'instant, nous distinguons les portées :

- **public**

Signale que la méthode est callable de n'importe quelle partie de l'application. Les méthodes publiques définissent le contrat de la classe, c'est-à-dire les opérations qui peuvent être demandées par son environnement.

- **private**

Signale que la méthode n'est callable que par l'objet lui-même ou par un objet du même type. Les méthodes privées sont des méthodes utilitaires pour un objet. Elles sont créées pour mutualiser du code ou pour simplifier un algorithme en le fractionnant en un ou plusieurs appels de méthodes.

## La valeur de retour

Une méthode peut avoir au plus un type de retour. Le compilateur signalera une erreur s'il existe un chemin d'exécution dans la méthode qui ne renvoie pas le bon type de valeur en retour. Pour retourner une valeur, on utilise le mot-clé **return**. Si le type de retour est un objet, la méthode peut toujours retourner la valeur spéciale **null**, c'est-à-dire l'absence d'objet. Une méthode qui ne retourne aucune valeur, le signale avec le mot-clé **void**.

```
public class Voiture {

    private String marque;
    private float vitesse;

    public float getVitesse() {
        return vitesse;
    }

    public void setMarque(String nouvelleMarque) {
        if (nouvelleMarque == null) {
            return;
        }
    }
}
```

```

    }
    marque = nouvelleMarque;
}

}

```

## Les paramètres

Une méthode peut éventuellement avoir des paramètres (ou arguments). Chaque paramètre est défini par son type et par son nom.

```

public class Voiture {

    public float getVitesse() {
        // implémentation ici
    }

    public void setVitesse(float deltaVitesse) {
        // implémentation ici
    }

    public void remplirReservoir(float quantite, TypeEssence typeEssence) {
        // implémentation ici
    }

}

```

Il est également possible de créer une méthode avec un nombre variable de paramètres (*varargs parameter*). On le signale avec trois points après le type du paramètre.

```

public class Calculatrice {

    public int additionner(int... valeurs) {
        int resultat = 0;
        for (int valeur : valeurs) {
            resultat += valeur;
        }
        return resultat;
    }

}

```

Le paramètre variable est vu comme un tableau dans le corps de la méthode. Par contre, il s'agit bien d'une liste de paramètres au moment de l'appel :

```

calculatrice calculatrice = new Calculatrice();

System.out.println(calculatrice.additionner(1)); // 1
System.out.println(calculatrice.additionner(1, 2, 3)); // 6
System.out.println(calculatrice.additionner(1, 2, 3, 4)); // 10

```

L'utilisation d'un paramètre variable obéit à certaines règles :

1. Le paramètre variable doit être le dernier paramètre

2. Il n'est pas possible de déclarer un paramètre variable acceptant plusieurs types

Au moment de l'appel, le paramètre variable peut être omis. Dans ce cas le tableau passé au corps de la méthode est un tableau vide. Un paramètre variable est donc également optionnel.

```
Calculatrice calculatrice = new Calculatrice();

system.out.println(calculatrice.additionner()); // 0
```

Il est possible d'utiliser un tableau pour passer des valeurs à un paramètre variable. Cela permet notamment d'utiliser un paramètre variable dans le corps d'une méthode comme paramètre variable à l'appel d'une autre méthode.

```
calculatrice calculatrice = new Calculatrice();

int[] valeurs = {1, 2, 3};
system.out.println(calculatrice.additionner(valeurs)); // 6
```

Pour l'exemple de la calculatrice, il peut sembler *naturel* d'obliger à passer au moins deux paramètres à la méthode *additionner*. Dans ce cas, il faut créer une méthode à trois paramètres :

```
public class Calculatrice {

    public int additionner(int valeur1, int valeur2, int... valeurs) {
        int resultat = valeur1 + valeur2;
        for (int valeur : valeurs) {
            resultat += valeur;
        }
        return resultat;
    }
}
```

## Paramètre final

Un paramètre peut être déclaré **final**. Cela signifie qu'il n'est pas possible d'assigner une nouvelle valeur à ce paramètre.

```
public class Voiture {

    public void accélérer(final float deltaVitesse) {
        deltaVitesse = 0.0f; // ERREUR DE COMPILATION

        // ...
    }
}
```

Rappelez-vous que **final** ne signifie pas réellement constant. En effet si le type d'un paramètre **final** est un objet, la méthode pourra tout de même appeler des méthodes sur cet objet qui modifient son état interne.

Note

Java n'autorise que le passage de paramètre par copie. Assigner une nouvelle valeur à un paramètre n'a donc un impact que dans les limites de la méthode. Cette pratique est généralement considérée comme mauvaise car cela peut rendre la compréhension du code de la méthode plus difficile. **final** est donc un moyen de nous aider à vérifier au moment de la compilation que nous n'assignons pas par erreur une nouvelle valeur à un paramètre. Cet usage reste tout de même très limité. Nous reviendrons plus tard sur l'intérêt principal de déclarer un paramètre **final** : la déclaration de classes anonymes.

## Les variables

Il est possible de déclarer des variables où l'on souhaite dans une méthode. Par contre, contrairement aux attributs, les variables de méthode n'ont pas de valeur par défaut. Cela signifie qu'il est obligatoire d'initialiser les variables. Il n'est pas nécessaire de les initialiser dès la déclaration, par contre, elles doivent être initialisées avant d'être lues.

## Méthodes de classe

Les méthodes définissent un comportement d'un objet et peuvent accéder aux attributs de l'instance. À l'instar des attributs, il est également possible de déclarer des *méthodes de classe*. Une méthode de classe ne peut pas accéder aux attributs d'un objet mais elle peut toujours accéder aux éventuels attributs de classe.

Pour déclarer une méthode de classe, on utilise le mot clé **static**.

```
public class Calculatrice {  
  
    public static int additionner(int... valeurs) {  
        int resultat = 0;  
        for (int valeur : valeurs) {  
            resultat += valeur;  
        }  
        return resultat;  
    }  
}
```

Comme pour l'exemple précédent, les méthodes de classe sont souvent des méthodes utilitaires qui peuvent s'exécuter sans nécessiter le contexte d'un objet. Dans un autre langage de programmation, il s'agirait de simples fonctions.

Les méthodes de classe peuvent être invoquées directement à partir de la classe. Donc il n'est pas nécessaire de créer une instance.

```
int resultat = Calculatrice.additionner(1, 2, 3, 4);
```

Note

Certaines classes de l'API Java ne contiennent que des méthodes de classe. On parle de classes utilitaires ou de classes outils puisqu'elles s'apparentent à une collection de fonctions. Parmi les plus utilisées, on trouve les classes [java.lang.Math](#), [java.lang.System](#), [java.util.Arrays](#) et [java.util.Collections](#).

Il est tout à fait possible d'invoquer une méthode de classe à travers une variable pointant sur une instance de cette classe :



```
Calculatrice c = new Calculatrice();
int resultat = c.additionner(1, 2, 3, 4);
```

Cependant, cela peut engendrer des difficultés de compréhension puisque l'on peut penser, à tort, que la méthode *additionner* peut avoir un effet sur l'objet. C'est notamment pour cela que Eclipse émet un avertissement si on invoque une méthode de classe à travers un objet. Même si l'effet est identique, il est plus lisible d'invoquer une méthode de classe à partir de la classe elle-même.

La méthode de classe la plus célèbre en Java est sans doute **main**. Elle permet de définir le point d'entrée d'une application dans une classe :

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

Les paramètres *args* correspondent aux paramètres passés en ligne de commande au programme **java** après le nom de la classe :

```
$ java MaClasse arg1 arg2 arg3
```

## Surcharge de méthode : overloading

Il est possible de déclarer dans une classe plusieurs méthodes ayant le même nom. Ces méthodes doivent obligatoirement avoir des paramètres différents (le type et/ou le nombre). Il est également possible de déclarer des types de retour différents pour ces méthodes. On parle de surcharge de méthode (**method overloading**). La surcharge de méthode n'a réellement de sens que si les méthodes portant le même nom ont un comportement que l'utilisateur de la classe jugera proche. Java permet également la surcharge de méthode de classe.

```
public class Calculatrice {

    public static int additionner(int... valeurs) {
        int resultat = 0;
        for (int valeur : valeurs) {
            resultat += valeur;
        }
        return resultat;
    }

    public static float additionner(float... valeurs) {
        float resultat = 0;
        for (float valeur : valeurs) {
            resultat += valeur;
        }
        return resultat;
    }
}
```

Dans l'exemple ci-dessus, la surcharge de méthode permet supporter l'addition pour le type entier et pour le type à virgule flottante. Selon le type de paramètre passé à l'appel, le compilateur déterminera laquelle des deux méthodes doit être appelée.

```
int resultatEntier = Calculatrice.additionner(1,2,3);
float resultat = Calculatrice.additionner(1f,2.3f);
```

Prudence

N'utilisez pas la surcharge de méthode pour implémenter des méthodes qui ont des comportements trop différents. Cela rendra vos objets difficiles à comprendre et donc à utiliser.

Si on surcharge une méthode avec un paramètre variable, cela peut créer une ambiguïté de choix. Par exemple :

```
public class Calculatrice {

    public static int additionner(int v1, int v2) {
        return v1 + v2;
    }

    public static int additionner(int... valeurs) {
        int resultat = 0;
        for (int valeur : valeurs) {
            resultat += valeur;
        }
        return resultat;
    }

}
```

Si on fait appel à la méthode *additionner* de cette façon :

```
calculatrice.additionner(2, 2);
```

Alors les deux méthodes *additionner* peuvent satisfaire cet appel. La règle appliquée par le compilateur est de chercher d'abord une correspondance parmi les méthodes qui n'ont pas de paramètre variable. Donc pour notre exemple ci-dessus, la méthode *additionner(int, int)* sera forcément choisie par le compilateur.

## Portée des noms et this

Lorsqu'on déclare un identifiant, qu'il s'agisse du nom d'une classe, d'un attribut, d'un paramètre, d'une variable..., il se pose toujours la question de sa portée : dans quel contexte ce nom sera-t-il compris par le compilateur ?

Pour les paramètres et les variables, la portée de leur nom est limitée à la méthode qui les déclare. Cela signifie que vous pouvez réutiliser les mêmes noms de paramètres et de variables dans deux méthodes différentes pour désigner des choses différentes.

Plus précisément, le nom d'une variable est limité au bloc de code (délimité par des accolades) dans lequel il a été déclaré. En dehors de ce bloc, le nom est inaccessible.

```
public int doSomething(int valeurMax) {
    int resultat = 0;

    // la variable i n'est accessible que dans la boucle for
    for (int i = 0; i < 10; ++i) {
```

```

// la variable k n'est accessible que dans la boucle for
for (int k = 0; k < 10; ++k) {
    // la variable m n'est accessible que dans ce bloc
    int m = resultat + i * k;
    if (m > valeurMax) {
        return valeurMax;
    }
    resultat = m;
}
}
return resultat;
}

```

En Java, le masquage de nom de variable ou de nom de paramètre est interdit. Cela signifie qu'il est impossible de déclarer une variable ayant le même nom qu'un paramètre ou qu'une autre variable accessible dans le bloc de code courant.

```

public int doSomething(int valeurMax) {
    int valeurMax = 2; // ERREUR DE COMPILATION
}
public int doSomething(int valeurMax) {
    int resultat = 0;
    for (int i = 0; i < 10; ++i) {
        resultat += i;
        if (resultat > 10) {
            int resultat = -1; // ERREUR DE COMPILATION
            return resultat;
        }
    }
}
return resultat;
}

```

Par contre, il est tout à fait possible de réutiliser un nom de variable dans deux blocs de code successifs. Cette pratique n'est vraiment utile que pour les variables temporaires (comme pour une boucle **for** contrôlée par un index). Sinon, cela gêne généralement la lecture.

```

public void doSomething(int valeurMin, int valeurMax) {
    for (int i = 0; i < valeurMax; ++i) {
        // implémentation
    }

    // on peut réutiliser le nom de variable i car il est déclaré
    // dans deux blocs for différents
    for (int i = 0; i < valeurMin; --i) {
        // implémentation
    }
}

```

En Java, le masquage du nom d'un attribut par un paramètre ou une variable est autorisé car les attributs sont toujours accessibles à travers le mot-clé **this**.

```
public class Voiture {  
    private String marque;  
  
    public void setMarque(String marque) {  
        this.marque = marque;  
    }  
}
```

**this** désigne l'instance courante de l'objet dans une méthode. On peut l'envisager comme une variable implicite accessible à un objet pour le désigner lui-même. Avec **this**, on peut accéder aux attributs et aux méthodes de l'objet. Il est même possible de retourner la valeur **this** ou la passer en paramètre pour indiquer une référence de l'objet courant :

```
public class Voiture {  
    private float vitesse;  
  
    public Voiture getPlusRapide(Voiture voiture) {  
        return this.vitesse >= voiture.vitesse ? this : voiture;  
    }  
}
```

S'il n'y a pas d'ambiguïté de nom, l'utilisation du mot-clé **this** est inutile. Cependant, certains développeurs préfèrent l'utiliser systématiquement pour indiquer explicitement l'accès à un attribut.

Prudence

**this** désignant l'objet courant, ce mot-clé n'est pas disponible dans une méthode de classe (méthode **static**). Pour résoudre le problème du masquage des attributs de classe dans ces méthodes, il suffit d'accéder au nom à travers le nom de la classe.

## Principe d'encapsulation

Un objet est constitué d'un état interne (l'ensemble de ses attributs) et d'une liste d'opérations disponibles pour ses clients (l'ensemble de ses méthodes publiques). En programmation objet, il est important que les clients d'un objet en connaissent le moins possible sur son état interne. Nous verrons plus tard avec les mécanismes d'héritage et d'interface qu'un client demande des services à un objet sans même parfois connaître le type exact de l'objet. La programmation objet introduit un niveau d'abstraction important et cette abstraction devient un atout pour la réutilisation et l'évolutivité.

Prenons l'exemple d'une classe permettant d'effectuer une connexion FTP et de récupérer un fichier distant. Les clients d'une telle classe n'ont sans doute aucun intérêt à comprendre les mécanismes compliqués du protocole FTP. Ils veulent simplement qu'on leur rende un service. Notre classe FTP pourrait très grossièrement ressembler à ceci :

```

public class ClientFtp {

    /**
     * @param uri l'adresse FTP du fichier
     *           par exemple ftp://monserveur/monfichier.txt
     * @return le fichier sous la forme d'un tableau d'octets
     */
    public byte[] getFile(String uri) {
        // implémentation
    }

}

```

Cette classe a peut-être des attributs pour connaître l'état du réseau et maintenir des connexions ouvertes vers des serveurs pour améliorer les performances. Mais tout ceci n'est pas de la responsabilité du client de cette classe qui veut simplement récupérer un fichier. Il est donc intéressant de cacher aux clients l'état interne de l'objet pour assurer un *couplage faible de l'implémentation*. Ainsi, si les développeurs de la classe *ClientFtp* veulent modifier son implémentation, ils doivent juste s'assurer que les méthodes publiques fonctionneront toujours comme attendues par les clients.

En programmation objet, le [principe d'encapsulation](#) nous incite à contrôler et limiter l'accès au contenu de nos classes au strict nécessaire afin de permettre le couplage le plus faible possible. L'encapsulation en Java est permise grâce à la portée **private**.

On considère que tous les attributs d'une classe **doivent** être déclarés **private** afin de satisfaire le [principe d'encapsulation](#).

Cependant, il est parfois utile pour le client d'une classe d'avoir accès à une information qui correspond à un attribut de l'état interne de l'objet. Plutôt que de déclarer cet attribut **public**, il existe en Java des méthodes dont la signature est facilement identifiable et que l'on nomme **getters** et **setters** (les accesseurs). Ces méthodes permettent d'accéder aux **propriétés** d'un objet ou d'une classe.

- **getter**

Permet l'accès en lecture à une propriété. La signature de la méthode se présente sous la forme : `public type getNomPropriete() { // ... }` Pour un type booléen, on peut aussi écrire : `public boolean isNomPropriete() { // ... }`

- **setter**

Permet l'accès en écriture à une propriété. La signature de la méthode se présente sous la forme : `public void setNomPropriete(type nouvelleValeur) { // ... }`

Ce qui donnera pour notre classe *Voiture* :

```

public class Voiture {

    // La vitesse en km/h
    private float vitesse;

    /**
     * @return La vitesse en km/h
     */
    public float getvitesse() {

```

```

        return vitesse;
    }

    /**
     * @param vitesse La vitesse en km/h
     */
    public void setVitesse(float vitesse) {
        this.vitesse = vitesse;
    }
}

```

Les *getters/setters* introduisent une abstraction supplémentaire : la **propriété**. Une propriété peut correspondre à un attribut ou à une expression. Du point de vue du client de la classe, cela n'a pas d'importance. Dans l'exemple ci-dessus, les développeurs de la classe *Voiture* peuvent très bien décider que l'état interne de la vitesse sera exprimé en mètres par seconde. Il devient possible de conserver la cohérence de notre classe en effectuant les conversions nécessaires pour passer de la propriété en km/s à l'attribut en m/s et inversement.

```

public class Voiture {

    // vitesse en m/s
    private float vitesse;

    private static float convertirEnMetresSeconde(float valeur) {
        return valeur * 1000f / 3600f;
    }

    private static float convertirEnKilometresHeure(float valeur) {
        return valeur / 1000f * 3600f;
    }

    /**
     * @return La vitesse en km/h
     */
    public float getVitesse() {
        return convertirEnKilometresHeure(vitesse);
    }

    /**
     * @param vitesse La vitesse en km/h
     */
    public void setVitesse(float vitesse) {
        this.vitesse = convertirEnMetresSeconde(vitesse);
    }
}

```

Avec les *getters/setters*, il est également possible de contrôler si une propriété est consultable et/ou modifiable. Si une propriété n'est pas consultable, il ne faut pas déclarer de *getter* pour cette propriété. Si une propriété n'est pas modifiable, il ne faut pas déclarer de *setter* pour cette propriété.

Astuce

Les *getters/setters* sont très utilisés en Java mais leur écriture peut être fastidieuse. Les IDE comme Eclipse introduisent un système de génération automatique. Dans Eclipse, faites un clic droit dans votre fichier de classe et choisissez *Source > Generate Getters and Setters...*

## Exercice

Exercice - La classe *Phrase*

Implémenter la classe *Phrase*. Cette classe permet de créer une chaîne de caractères en ajoutant des mots qui seront séparés par un séparateur. Par défaut, le séparateur est le caractère espace, mais il peut être changé.

La classe *Phrase* doit permettre de :

- ajouter un mot
- ajouter une série de mots
- ajouter un mot en le répétant n fois
- changer le séparateur qui peut être soit une chaîne de caractères soit un seul caractère
- générer la phrase terminée par un point lorsqu'on demande la représentation sous la forme d'une chaîne de caractères de l'objet
- connaître le nombre de lettres dans la phrase

Le code suivant :

```
Phrase phrase = new Phrase();
phrase.ajouter("Une");
phrase.ajouter("classe");
phrase.ajouter("pour");
phrase.ajouter("ajouter");
phrase.ajouter("des mots");
phrase.setSeparateur(" et encore ");
phrase.ajouter("des mots", 3);
phrase.setSeparateur(' ');
phrase.ajouter("toujours", "et", "encore");

System.out.println(phrase);
System.out.println(phrase.getNbLettres());
```

doit produire sur la sortie standard

```
Une classe pour ajouter des mots et encore des mots et encore des mots et encore
des mots toujours et encore.
88
```

Indication

On ne peut pas parcourir directement une chaîne de caractères, par contre on peut obtenir un tableau de caractères à partir d'une chaîne avec la méthode [toCharArray](#).

```
String helloWorld = "Hello world!";
for (char c : helloWorld.toCharArray()) {
    // ...
}
```

Pour savoir si un caractère est une lettre, on utilise la méthode **static** [Character.isAlphabetic](#).

Exercice - Méthode et attribut de classe

Ajoutez une méthode de classe à la classe *Phrase* qui retourne le nombre de lettres de la plus grande phrase créée.

## Cycle de vie d'un objet

Ce chapitre détaille la création d'un objet et son cycle de vie. Nous aborderons notamment les constructeurs et les mécanismes de gestion de la mémoire de la JVM.

### Le constructeur

Il est possible de déclarer des méthodes particulières dans une classe que l'on nomme **constructeurs**. Un constructeur a pour objectif d'initialiser un objet nouvellement créé afin de garantir qu'il est dans un état cohérent avant d'être utilisé.

Un constructeur a la signature suivante :

```
[portée] [nom de la classe]([paramètres]) {

}
```

Un constructeur se distingue d'une méthode car il n'a jamais de type de retour (pas même **void**). De plus un constructeur a obligatoirement **le même nom que la classe**.

```
public class Voiture {

    public Voiture() {
        // Le constructeur
    }

}
```

Lorsqu'une voiture est créée par l'application avec l'opérateur **new** comme avec l'instruction suivante :

```
Voiture voiture = new Voiture();
```

Alors, la JVM crée l'espace mémoire nécessaire pour le nouvel objet de type *Voiture*, puis elle appelle le constructeur et enfin elle assigne la référence de l'objet à la variable *voiture*. Donc le constructeur permet de réaliser une initialisation complète de l'objet selon les besoins des développeurs.



## Paramètres de constructeur

Comme les méthodes, les constructeurs peuvent accepter des paramètres et comme les méthodes, les constructeurs supportent la surcharge (*overloading*). Une classe peut ainsi déclarer plusieurs constructeurs à condition que la liste des paramètres diffère par le nombre et/ou le type.

```
public class Voiture {  
  
    private String marque;  
    private float vitesse;  
  
    public Voiture(String marque) {  
        this.marque = marque;  
    }  
  
    public Voiture(String marque, float vitesseInitiale) {  
        this.marque = marque;  
        this.vitesse = vitesseInitiale;  
    }  
  
}
```

Dans l'exemple ci-dessus, la classe *Voiture* déclare deux constructeurs avec des paramètres différents. Il est maintenant nécessaire de passer des paramètres au moment de la création d'une instance de cette classe. Pour cet exemple, on voit que le constructeur permet de forcer la création d'une instance de *Voiture* en fournissant au moins sa marque.

```
Voiture voiture = new Voiture("DeLorean");  
Voiture voiture2 = new Voiture("DeLorean", 88.0f);
```

## Valeur par défaut des attributs

Nous avons vu précédemment que les attributs d'une classe peuvent être initialisés explicitement à la déclaration. Dans le cas contraire, ils sont initialisés avec une valeur par défaut. Java garantit que cette initialisation a lieu avant l'appel au constructeur.

```
public class Vehicule {  
  
    private String marque;  
    private int nbRoues = 4;  
    private float vitesse;  
  
    public Vehicule(String marque) {  
        this.marque = marque;  
        // la vitesse vaudra 0 et nbRoues vaudra 4  
    }  
  
    public Vehicule(String marque, int nbRoues) {  
        this.marque = marque;  
        // On ne peut créer que des véhicules avec au plus 4 roues  
        if (nbRoues < this.nbRoues) {  
            this.nbRoues = nbRoues;  
        }  
    }  
  
}
```

```

    }
    // la vitesse vaudra 0
}

}

```

Les attributs déclarés **final** sont traités un peu différemment. Ces attributs doivent être obligatoirement et explicitement initialisés avant la fin de la création de l'objet. Donc, il est possible de les initialiser dans le constructeur. Par contre, le compilateur générera une erreur si :

- un constructeur tente d'accéder à un attribut **final** qui n'a pas encore été initialisé
- un constructeur se termine sans avoir initialisé explicitement tous les attributs **final**
- un constructeur tente d'affecter une valeur à un attribut **final** qui a déjà été initialisé au moment de sa déclaration.

```

public class Vehicule {

    private static final int DEFAULT_NBROUES = 4;

    private final String marque;
    private final int nbRoues;
    private float vitesse;

    public Vehicule(String marque) {
        this.marque = marque;
        this.nbRoues = DEFAULT_NBROUES;
        // la vitesse vaudra 0
    }

    public Vehicule(String marque, int nbRoues) {
        this.marque = marque;
        // On ne peut créer que des véhicules avec au plus 4 roues
        this.nbRoues = nbRoues < DEFAULT_NBROUES ? nbRoues : DEFAULT_NBROUES;
        // la vitesse vaudra 0
    }

}

```

## Constructeur par défaut

Le compilateur Java garantit que toutes les classes ont au moins un constructeur. Si vous créez la classe suivante :

```

public class Voiture {

}

```

Alors, le compilateur ajoutera le code nécessaire qui correspondrait à :

```
public class Voiture {

    public Voiture() {
    }

}
```

Ce constructeur est appelé le **constructeur par défaut**. Par contre si votre classe contient au moins un constructeur, quelle que soit sa signature, alors le compilateur n'ajoutera pas le **constructeur par défaut**.

```
public class Voiture {

    private final String marque;

    /* Le compilateur ne générera pas de constructeur par défaut.
     * Pour créer une voiture, je suis obligé de fournir sa marque en paramètre
     * de création.
     */
    public Voiture(String marque) {
        this.marque = marque;
    }

}
```

Astuce

Si votre classe ne contient qu'un seul constructeur sans paramètre dont le corps est vide, alors vous pouvez supprimer cette déclaration car le compilateur le générera automatiquement.

## Constructeur privé

Il est tout à fait possible d'interdire l'instantiation d'une classe en Java. Pour cela, il suffit de déclarer tous ses constructeurs avec une portée **private**.

```
public class Calculatrice {

    private Calculatrice() {
    }

    public static int additionner(int... valeurs) {
        int resultat = 0;
        for (int valeur : valeurs) {
            resultat += valeur;
        }
        return resultat;
    }

}
```

Comme montré dans l'exemple ci-dessus, un cas d'usage courant est la création d'une classe outil. Une classe outil ne contient que des méthodes de classe. Il n'y a donc aucun intérêt à instancier une telle classe. Donc, on déclare un constructeur privé pour éviter une utilisation incorrecte.

Note

On peut aussi considérer que la classe *Calculatrice* est simplement un espace de nom contenant un ensemble de fonctions. Même si les fonctions n'existent pas en Java, les classes outils sont un moyen de les simuler.

## Appel d'un constructeur dans un constructeur

Certaines classes peuvent offrir différents constructeurs à ses utilisateurs. Souvent ces constructeurs vont partiellement exécuter le même code. Pour simplifier la lecture et éviter la duplication de code, un constructeur peut appeler un autre constructeur en utilisant le mot-clé **this** comme nom du constructeur. Cependant, un constructeur ne peut appeler qu'un **seul** constructeur et, s'il le fait, cela doit être sa première instruction.

```
public class Vehicule {

    private static final int DEFAULT_NBROUES = 4;

    private final String marque;
    private final int nbRoues;
    private float vitesse;

    public Vehicule(String marque) {
        this(marque, DEFAULT_NBROUES, 0f);
    }

    public Vehicule(String marque, int nbRoues) {
        this(marque, nbRoues, 0f);
    }

    public Vehicule(String marque, int nbRoues, float vitesseInitiale) {
        this.marque = marque;
        this.nbRoues = nbRoues < DEFAULT_NBROUES ? nbRoues : DEFAULT_NBROUES;
        this.vitesse = vitesseInitiale;
    }

}
```

La classe *Vehicule* ci-dessus offre plusieurs possibilités d'initialisation, mais les développeurs de cette classe ont évité la duplication en plaçant le code d'initialisation dans le troisième constructeur.

## Appel d'une méthode dans un constructeur

Il est tout à fait possible d'appeler une méthode de l'objet dans un constructeur. Cela est même très utile pour éviter la duplication de code et favoriser la réutilisation. Attention cependant au statut particulier des constructeurs. Tant qu'un constructeur n'a pas achevé son exécution, l'objet n'est pas totalement initialisé. Il peut donc y avoir des cas où l'appel à une méthode peut avoir des comportements inattendus.

Prenons l'exemple suivant :

```
public class Vehicule {
    private static final int DEFAULT_NBROUES = 4;

    private final String marque;
```

```

private final int nbRoues;
private float vitesse;

public Vehicule(String marque, int nbRoues, float vitesseInitiale) {
    faireQuelqueChoseInattendue();
    this.marque = marque;
    this.nbRoues = nbRoues < DEFAULT_NBROUES ? nbRoues : DEFAULT_NBROUES;
    this.vitesse = vitesseInitiale;
}

private void faireQuelqueChoseInattendue() {
    System.out.println(this.nbRoues); // 0
}
}

```

Le constructeur appelle la méthode *faireQuelqueChoseInattendue* qui affiche la valeur de l'attribut *nbRoues*. Cet attribut est déclaré **final** donc il n'est pas modifiable durant la vie de l'objet et la tâche du constructeur va être, entre autres, de lui assigner une valeur. Mais comme la méthode *faireQuelqueChoseInattendue* est appelée avant l'initialisation, elle affichera 0. Il s'agit d'un comportement aberrant du point de vue de la définition de **final** mais qui compile et s'exécute sans erreur.

Plus généralement, si vous souhaitez appeler des méthodes de l'objet dans un constructeur, il faut prendre soin de s'assurer que l'état de l'objet nécessaire à l'exécution de ces méthodes est correctement initialisé avant par le constructeur.

## Injection de dépendances par le constructeur

L'état interne d'un objet (ses attributs) inclut souvent des références vers d'autres objets. Parfois, ces objets peuvent eux-même avoir une représentation interne complexe qui nécessite des références vers d'autres objets... Par exemple, une classe *Voiture* peut nécessiter une instance d'une classe *Moteur* :

```

public class Moteur {

    private int nbCylindres;
    private int nbSoupapesParCylindre;
    private float vitesseMax;

    public Moteur(int nbCylindres, int nbSoupapesParCylindre, float vitesseMax) {
        this.nbCylindres = nbCylindres;
        this.nbSoupapesParCylindre = nbSoupapesParCylindre;
        this.vitesseMax = vitesseMax;
    }

    // ...
}

```

À partir de la classe *Moteur* ci-dessus, nous pouvons fournir l'implémentation suivante de la classe *Voiture* :

```

public class Voiture {

    private String marque;
    private Moteur moteur;

    public Voiture(String marque, int nbCylindres, int nbSoupapesParCylindre, float
vitesseMax) {
        this.marque = marque;
        this.moteur = new Moteur(nbCylindres, nbSoupapesParCylindre, vitesseMax);
    }

    // ...
}

```

Et créer une instance de la classe *Voiture* :

```

voiture clio = new Voiture("Clio williams", 4, 4, 216);

```

Cependant, si nous considérons le type de relation qui unit la classe *Voiture* à la classe *Moteur*, nous constatons que non seulement la classe *Voiture* est dépendante de la classe *Moteur* mais qu'en plus la classe *Voiture* crée l'instance de la classe *Moteur* dont elle a besoin. Donc la classe *Voiture* a un couplage très fort avec la classe *Moteur*. Par exemple, si le constructeur de la classe *Moteur* évolue alors le constructeur de la classe *Voiture* doit également évoluer.

En programmation objet, créer une objet n'hésite souvent de disposer des informations nécessaires pour invoquer le constructeur de sa classe. La plupart du temps, les classes qui sont dépendantes d'autres classes n'ont pas vocation à les créer car il n'y a pas vraiment de raison à ce qu'elles connaissent les informations nécessaires à leur création. Dans le cas de notre classe *Voiture* nous pouvons proposer simplement l'implémentation :

```

public class Voiture {

    private String marque;
    private Moteur moteur;

    public Voiture(String marque, Moteur moteur) {
        this.marque = marque;
        this.moteur = moteur;
    }

    // ...
}

```

La création d'une instance de *Voiture* se fait maintenant en deux étapes :

```

Moteur moteur = new Moteur(4, 4, 216);
voiture clio = new Voiture("Clio williams", moteur);

```

On dit qu'une instance de la classe *Moteur* est **injectée** par le constructeur dans une instance de *Voiture*. En programmation objet, cela signifie que nous avons découplé l'utilisation de l'instance de la classe *Moteur* de sa création.

L'injection de dépendances est une technique de programmation qui permet à une classe de disposer des instances d'objet dont elle a besoin sans avoir à les créer directement.

Note

L'injection de dépendance est la technique qui est à la base de [l'inversion de dépendances](#) (appelée aussi parfois inversion de contrôle) qui est un des principes [SOLID](#) en programmation objet. Beaucoup de frameworks Java (comme le [Spring framework](#)) sont basés sur ce principe.

## Le bloc d'initialisation

Il est possible d'écrire un traitement d'initialisation s'effectuant avant l'appel au constructeur. Il suffit de déclarer un bloc anonyme dans la classe.

```
public class Voiture {  
  
    private final int nbRoues;  
  
    {  
        Configuration cfg = getConfiguration();  
        nbRoues = cfg.nbRouesParVoiture;  
    }  
  
    private Configuration getConfiguration() {  
        // le code ici pour consulter la configuration  
    }  
  
}
```

Dans l'exemple précédent, on suppose qu'il existe une classe *Configuration* et qu'il est possible de consulter la configuration de l'application pour connaître le nombre de roues par voiture. Le bloc d'initialisation accède à la configuration et affecte la bonne valeur à l'attribut **final** *nbRoues*.

Le bloc d'initialisation est très rarement employé en Java. On peut systématiquement obtenir le même comportement en déclarant un constructeur.

## Le bloc d'initialisation de classe

Il est possible d'écrire un traitement d'initialisation d'une classe. Ce traitement ne sera effectué qu'une seule fois : au moment du chargement de la définition de la classe dans la mémoire de la JVM. Une initialisation de classe se fait à l'aide d'un bloc d'instructions **static**.

```
public class Voiture {  
  
    private static final int NB_ROUES;  
  
    static {  
        Configuration cfg = getConfiguration();  
        NB_ROUES = cfg.nbRouesParVoiture;  
    }  
  
    private static Configuration getConfiguration() {  
        // le code ici pour consulter la configuration  
    }  
  
}
```

```
}
```

Dans l'exemple précédent, on suppose qu'il existe une classe *Configuration* et qu'il est possible de consulter la configuration de l'application pour connaître le nombre de roues par voiture. Le bloc **static** donne la possibilité d'initialiser une constante à partir d'un traitement plus complexe.

On peut obtenir un résultat similaire en initialisant la constante *NB\_ROUES* à partir d'un appel à une méthode de classe :

```
public class Voiture {  
  
    private static final int NB_ROUES = getConfiguration().nbRouesParVoiture;  
  
    private static Configuration getConfiguration() {  
        // le code ici pour consulter la configuration  
    }  
  
}
```

## Exercice

Exercice - Les constructeurs pour la classe *Phrase*

En reprenant la classe *Phrase* implémentée à l'exercice du chapitre précédent, ajoutez les constructeurs suivants :

```
// constructeur sans paramètre  
Phrase phrase = new Phrase();  
phrase.ajouter("Bonjour", "le", "monde");  
System.out.println(phrase); // Bonjour le monde.  
  
// constructeur pour indiquer quel caractère fini la phrase  
Phrase phraseInterrogative = new Phrase(Phrase.INTERROGATION);  
phraseInterrogative.ajouter("Ça", "va");  
System.out.println(phraseInterrogative); // Ça va ?  
  
Phrase phraseExclamative = new Phrase(Phrase.EXCLAMATION);  
phraseExclamative.ajouter("Oui");  
System.out.println(phraseExclamative); // Oui !  
  
Phrase phraseDeclarative = new Phrase(Phrase.DECLARATION);  
phraseDeclarative.ajouter("C'est", "bien");  
System.out.println(phraseDeclarative); // Assez bien.  
  
// constructeur pour indiquer quel caractère fini la phrase  
// et passer un nombre quelconque de mots à la phrase  
phrase = new Phrase(Phrase.INTERROGATION, "Et", "ensuite");  
System.out.println(phrase); // Et ensuite ?
```

Ajoutez également des méthodes de classe pour créer directement une *Phrase* selon son type :



```
phrase = Phrase.interrogation("Comment", "allez", "vous");
System.out.println(phrase); // Comment allez vous ?

phrase = Phrase.exclamation("Dormez");
System.out.println(phrase); // Dormez !

phrase = Phrase.declaration("Brian", "est", "dans", "la", "cuisine");
System.out.println(phrase); // Brian est dans la cuisine.
```

## Mémoire heap et stack

Comme pour la plupart des langages de programmation, Java utilise deux espaces mémoires : la **stack** (ou *call stack*, la pile d'appel) et le **heap** (le tas).

La **stack** correspond à l'espace alloué pour gérer la mémoire nécessaire à l'exécution des méthodes (d'un thread). C'est dans cet espace que les variables déclarées dans la méthode sont stockées. Cet espace a la structure d'un pile car lorsqu'une méthode appelle une autre méthode, l'espace mémoire nécessaire à cet appel s'empile au dessus de l'espace mémoire précédent. Lorsqu'une méthode se termine l'espace mémoire qui lui est alloué dans la stack est libéré. Cela signifie que lorsqu'une méthode se termine, il n'est plus possible d'accéder aux variables qu'elle a déclarées.

Le **heap** permet de stocker de l'information en allouant dynamiquement de l'espace mémoire lorsque cela est nécessaire et de le libérer lorsqu'il n'est plus utile. Le heap a une structuration plus complexe qui tient compte de la durée de vie présumée des éléments qui le composent. Dans le heap se trouve, la description des classes chargées par la JVM mais surtout tous les objets créés. En effet, le mot-clé **new** a pour fonction de créer un nouvel objet en stockant ses informations dans le heap.

Tous les objets Java étant créés dans le heap, leur durée de vie peut être plus longue que le temps d'exécution d'une méthode. Il n'est pas possible pour un développeur de demander explicitement la destruction d'un objet. Par contre il existe un procédé appelé le *ramasse-miettes* (**garbage collector**) qui se charge de libérer la mémoire lorsqu'il détecte qu'elle n'est plus utilisée.

Note

La machine virtuelle Java gère elle-même l'espace mémoire allouable à la stack et au heap (alors qu'il s'agit normalement d'une activité prise en charge par le système d'exploitation lui-même). Du coup, il est possible de paramétrer au lancement de la JVM la taille mémoire allouable si on souhaite introduire des quotas par processus avec les paramètres :

- -Xms  
Taille initiale du heap
- -Xmx  
Taille maximale du heap
- -Xss  
Taille de la stack (par thread)

Par exemple :

```
$ java -Xms512M -Xmx512M MonApplication
```

Dans l'exemple précédent, l'application est lancée avec un heap d'une taille fixe de 512 Mo.

## Le ramasse-miettes

---

Le ramasse-miettes (*garbage collector*) est un processus léger (*thread*) qui est créé par la JVM et qui s'exécute régulièrement pour contrôler l'état de la mémoire. S'il détecte que des portions de mémoire allouées ne sont plus utilisées, il les libère afin que l'application ne manque pas de ressource mémoire.

La présence du ramasse-miettes évite aux développeurs de devoir demander explicitement la libération de la mémoire. D'ailleurs il n'est pas possible en Java de demander explicitement la libération de la mémoire. Cependant, il est important que les développeurs comprennent le fonctionnement du ramasse-miettes.

Le ramasse-miettes vérifie périodiquement si les objets sont référencés. Un objet est référencé si :

- la méthode en cours d'exécution ou une méthode de la pile d'appel possède une variable référençant cet objet
- il existe un objet référencé qui possède un attribut référençant cet objet

Le ramasse-miettes gère également le problème de la référence circulaire. Un objet qui contiendrait un attribut qui le référence directement ou indirectement lui-même n'est pas réellement considéré comme une référence par le ramasse-miettes.

Donc, si un développeur souhaite qu'un objet soit détruit et son espace mémoire récupéré, il doit s'assurer que plus aucune référence n'existe vers cet objet. Par exemple, il peut affecter la valeur **null** aux variables et aux attributs qui référencent cet objet.

Il est également possible de forcer l'appel au ramasse-miettes grâce à la méthode [java.lang.System.gc\(\)](#). Cependant, cette méthode ne donne aucune garantie quant au résultat. Vous ne pouvez pas vous baser sur son appel pour garantir la suppression d'un objet non référencé. Le ramasse-miettes utilise un algorithme complexe qui rend son comportement difficilement prédictible.

### Note

Le ramasse-miettes est parfois la préoccupation des ingénieurs système. En effet, les serveurs implémentés en Java dépendent du ramasse-miettes pour gérer la désallocation de la mémoire. Même si l'exécution du ramasse-miettes est rapide, elle peut avoir des effets sur des serveurs très sollicités en entraînant des micro-interruptions du service. Java propose non pas un mais des algorithmes de ramasses-miettes configurables. Il est donc possible de choisir au lancement de la JVM le type de ramasse-miettes à utiliser.

Le ramasse-miettes fait l'objet de modification et d'évolution à toutes les versions de Java. Pour Java 9, vous pouvez vous reporter au [guide de tuning](#) du ramasse-miettes.

### Avertissement

Java propose un mécanisme de ramasse miettes mais ce dernier ne peut libérer l'espace mémoire que des objets non référencés. Si vous développez une application qui crée beaucoup d'objets sans donner la possibilité au ramasse-miettes de les collecter, votre application peut se retrouver à cours d'espace mémoire. Lors de la création d'un nouvel objet, vous obtiendrez alors une erreur du type [java.lang.OutOfMemoryError](#).

La mémoire n'est pas la seule ressource système avec laquelle les développeurs doivent composer. Si Java propose un mécanisme pour la gestion de la mémoire, il ne propose pas de mécanisme automatique pour réclamer les autres types de ressources, notamment les descripteurs de fichier et de socket.

## La méthode *finalize*

Si un objet souhaite effectuer un traitement avant sa destruction, il peut implémenter la méthode *finalize*. Cette méthode a la signature suivante :

```
protected void finalize() {  
}
```

Dans la pratique cette méthode n'est utilisée que pour des cas d'implémentation très avancés. En effet, la JVM ne donne strictement **aucune** garantie sur le moment où la méthode **finalize** est appelée. Elle peut même ne jamais être appelée si l'application se termine avant que le ramasse-miettes ne réclame l'espace mémoire de l'objet. Elle n'a donc pas le même statut ni la même importance qu'un destructeur dans le langage C++.

```
public class ObjetCurieux {  
  
    protected void finalize() {  
        System.out.print("je vais disparaître !");  
    }  
  
    public static void main(String[] args) {  
        ObjetCurieux objetCurieux = new ObjetCurieux();  
        objetCurieux = null;  
  
        System.gc();  
  
        for(int i = 0; i < 1000 ; ++i) {  
            System.out.print('.');  
        }  
    }  
}
```

Dans l'exemple ci-dessus, Un objet qui n'implémente que la méthode *finalize* est créé puis la variable qui le référence est mise à **null**. Ensuite, le programme appelle explicitement le ramasse-miettes avec la méthode [java.lang.System.gc\(\)](#). Enfin, une boucle se contente d'afficher mille points sur la sortie standard. Cette boucle **for** est utile car généralement le programme s'arrête trop vite et le ramasse-miettes n'a pas le temps d'appeler *finalize*. Si vous exécutez ce programme plusieurs fois, vous constaterez que le message « je vais disparaître ! » ne s'affiche pas au même moment. Cela traduit bien le fait que le comportement du ramasse-miettes varie d'une exécution à l'autre.

# Les packages

Un problème courant dans les langages de programmation est celui de la collision de noms. Si par exemple, je veux créer une classe *TextEditor* pour représenter une composant graphique complexe pour éditer un texte, un autre développeur peut également le faire. Si nous distribuons nos classes, cela signifie qu'une application peut se retrouver avec deux classes dans son *classpath* qui portent exactement le même nom mais qui ont des méthodes et des comportements différents.

Dans la pratique, la JVM chargera la première classe qu'elle peut trouver et ignorera la seconde. Ce comportement n'est pas acceptable. Pour cela, il faut pouvoir différencier ma classe *TextEditor* d'une autre.

Le moyen le plus efficace est d'introduire un espace de noms qui me soit réservé. Dans cet espace, *TextEditor* ne désignerait que ma classe. En Java, les **packages** servent à délimiter des espaces de noms.

## Déclaration d'un package

Pour qu'une classe appartienne à un package, il faut que son fichier source commence par l'instruction :

```
package [nom du package];
```

Une classe ne peut appartenir qu'à un seul package. Les packages sont également représentés sur le disque par des répertoires. Donc pour la classe suivante :

```
package monapplication;

public class TextEditor {

}
```

Cette classe doit se trouver dans le fichier *TextEditor.java* et ce fichier doit lui-même se trouver dans un répertoire nommé *monapplication*. Pour les fichiers *class* résultants de la compilation, l'organisation des répertoires doit être conservée (c'est d'ailleurs ce que fait le compilateur). Ainsi, si deux classes portent le même nom, elles se trouveront chacune dans un fichier avec le même nom mais dans des répertoires différents puisque ces classes appartiendront à des packages différents.

### Note

Quand on spécifie le **classpath** à la compilation ou au lancement d'un programme, on spécifie le ou les répertoires à partir desquels se trouvent les packages.

Si une classe ne déclare pas d'instruction **package** au début du fichier, on dit qu'elle appartient au package par défaut (qui n'a pas de nom). Même si le langage l'autorise, c'est quasiment toujours une mauvaise idée. Les IDE comme Eclipse signalent d'ailleurs un avertissement si vous voulez créer une classe dans le package par défaut. Jusqu'à présent, les exemples donnés ne mentionnaient pas de package. Mais maintenant que cette notion a été introduite, les exemples à venir préciseront toujours un package.

## Sous package

Comme pour les répertoires, les packages suivent une organisation arborescente. Un package contenu dans un autre package est appelé un **sous package** :

```
package monapplication.monsouspackage;
```

Sur le système de fichiers, on trouvera donc un répertoire *monapplication* avec à l'intérieur un sous répertoire *monsouspackage*.

## Nom d'un package

Comme le mécanisme des packages a été introduit pour éviter la collision de noms, il est conseillé de suivre une convention de nommage de ses packages. Pour une organisation, on utilise le nom de domaine inversé comme base de l'arborescence de packages : par exemple dev.gayerie. On ajoute généralement ensuite le nom de l'application ou de la bibliothèque.

Les noms de packages contenant le mot *java* sont réservés pour la bibliothèque standard. On trouve ainsi des packages *java* ou *javax* (pour indiquer une extension de Java) dans la bibliothèque standard fournie avec le JDK.

## Nom complet d'une classe

Une classe est normalement désignée par son *nom complet*, c'est-à-dire par le chemin de packages suivi d'un `.` suivi du nom de la classe.

Par exemple, la classe [String](#) s'appelle en fait [java.lang.String](#) car elle se trouve dans le package [java.lang](#). J'ai donc la possibilité, si je le souhaite, de créer ma propre classe String par exemple dans le package dev.gayerie :

```
package dev.gayerie;

public class String {

}
```

Il est possible d'accéder à une classe en spécifiant son nom complet. Par exemple, pour accéder à la classe [java.util.Arrays](#) :

```
package dev.gayerie;

public class MaClasse {

    public static void main(String[] args) {
        int[] tableau = {5, 6, 3, 4};
        java.util.Arrays.sort(tableau);
    }
}
```

Par défaut, une classe a accès à l'espace de nom de son propre package et du package [java.lang](#). Voilà pourquoi, il est possible d'utiliser directement les classes [String](#) ou [Math](#) sans avoir à donner leur nom complet : [java.lang.String](#), [java.lang.Math](#).

Si nous créons deux classes : *Voiture* et *Conducteur*, toutes deux dans le package `dev.gayerie` :

```
package dev.gayerie;

public class Conducteur {

    // ...

}

package dev.gayerie;

public class Voiture {

    private Conducteur conducteur;

    public void setConducteur(Conducteur conducteur) {
        this.conducteur = conducteur;
    }

    // ...

}
```

La classe *Voiture* et la classe *Conducteur* appartiennent toutes les deux au package `dev.gayerie`. La classe *Voiture* peut donc référencer la classe *Conducteur* sans préciser le package.

## Import de noms

Pour éviter de préfixer systématiquement une classe par son nom de package, il est possible d'importer son nom dans l'espace de noms courant grâce au mot-clé **import**. Une instruction **import** doit se situer juste après la déclaration de **package** (si cette dernière est présente). Donc, il n'est pas possible d'importer un nom en cours de déclaration d'une classe ou d'une méthode.

Le mot-clé **import** permet d'importer :

- Un nom de classe particulier

```
import java.util.Arrays;
```

- Un nom de méthode de classe ou d'attribut de classe

```
import static java.lang.Math.abs;
import static java.lang.System.out;
```

- Un nom de classe interne (inner class)

```
import java.util.Map.Entry;
```

- Tous les noms d'un package

```
import java.util.*;
```

- Tous les noms des méthodes et des attributs de classe

```
import static java.lang.Math.*;
```

Le caractère `*` permet d'importer tous les noms d'un package dans l'espace de nom courant. Même si cela peut sembler très pratique, il est pourtant déconseillé de le faire. Tous les IDE Java savent gérer automatiquement les importations. Dans Eclipse, lorsque l'on saisit le nom d'une classe qui ne fait pas partie de l'espace de nom, il suffit de demander la complétion de code (*CTRL* + *espace*) et de choisir dans la liste la classe appartenant au package voulu et Eclipse génère automatiquement l'instruction **import** pour ce nom de classe. De plus, on peut demander à Eclipse à tout moment de réorganiser les importations (*CTRL* + *MAJ* + *O*). Ainsi, la gestion des importations est grandement automatisée et le recours à `*` comme facilité d'écriture n'est plus vraiment utile.

```
package dev.gayerie;

import static java.lang.Math.random;
import static java.lang.System.out;
import static java.util.Arrays.sort;

import java.time.Duration;
import java.time.Instant;

public class BenchmarkTriTableau {

    public static void main(String[] args) {
        int[] tableau = new int[1_000_000];

        for (int i = 0; i < tableau.length; ++i) {
            tableau[i] = (int) random();
        }

        Instant start = Instant.now();
        sort(tableau);
        Duration duration = Duration.between(start, Instant.now());

        out.println("Durée de l'opération de tri du tableau : " + duration);
    }
}
```

#### Note

Si vous importez un nom qui est déjà défini dans l'espace courant, alors l'import n'aura aucun effet. Dans ce cas, vous serez obligé d'accéder à un nom de classe avec son nom long afin d'éviter toute ambiguïté.

## La portée de niveau package

Nous avons vu précédemment que les classes, les méthodes et les attributs peuvent avoir une portée **public** ou **private**. Il existe également une portée de niveau package. Une classe, une méthode ou un attribut avec cette portée n'est accessible qu'aux membres du même package. Cela permet notamment de créer des classes nécessaires au fonctionnement du package tout en les dissimulant aux éléments qui ne sont pas membres du package.

Il n'y a pas de mot-clé pour désigner la portée de niveau package. Il suffit simplement d'omettre l'information de portée.

Imaginons que nous voulions créer une bibliothèque de cryptographie. Nous pouvons créer une classe pour chaque algorithme. Par contre, pour simplifier l'utilisation, nous pouvons fournir une classe outil de chiffrement. Dans ce cas, il n'est pas nécessaire de rendre accessible à l'extérieur du package les classes représentant les algorithmes : on les déclare alors avec la portée package.

CypherAlgorithm.java

```
package dev.gayerie.cypher;

class CypherAlgorithm {

    public CypherAlgorithm() {
        // ...
    }

    public byte[] encrypt(byte[] msg) {
        // ...
    }
}
```

CypherLibrary.java

```
package dev.gayerie.cypher;

public class CypherLibrary {

    private CypherLibrary() {
    }

    public static byte[] cypher(byte[] msg) {
        CypherAlgorithm algo = new CypherAlgorithm();
        return algo.cypher(msg);
    }
}
```

La classe *CypherAlgorithm* est de portée package, elle est donc invisible pour les classes qui ne sont pas membres de son package. Par contre, elle est utilisée par la classe *CypherLibrary*.

La portée de niveau package est souvent utilisée pour dissimuler la complexité de l'implémentation en ne laissant voir que les classes et/ou les méthodes réellement utiles aux utilisateurs.

## Le fichier package-info.java

Il est possible de créer un fichier spécial dans un package nommé *package-info.java*. Au minimum, ce fichier doit contenir une instruction **package**. Ce fichier particulier permet d'ajouter un commentaire Javadoc pour documenter le package lui-même. Il peut également contenir des annotations pour le package.

contenu du fichier package-info.java pour dev.gayerie



```
package dev.gayerie;

/**
 * Ceci est le commentaire pour le package.
 */
```

# Héritage et composition

Une application Java est composée d'un ensemble d'objets. Un des intérêts de la programmation objet réside dans les relations que ces objets entretiennent les uns avec les autres. Ces relations sont construites par les développeurs et constituent ce que l'on appelle l'architecture d'une application. Il existe deux relations fondamentales en programmation objet :

- **est un** (*is-a*)

Cette relation permet de créer une chaîne de relation d'identité entre des classes. Elle indique qu'une classe peut être assimilée à une autre classe qui correspond à une notion plus abstraite ou plus générale. On parle **d'héritage** pour désigner le mécanisme qui permet d'implémenter ce type de relation.

- **a un** (*has-a*)

Cette relation permet de créer une relation de dépendance d'une classe envers une autre. Une classe a besoin des services d'une autre classe pour réaliser sa fonction. On parle également de relation de **composition** pour désigner ce type de relation.

## L'héritage (is-a)

Imaginons que nous voulions développer un simulateur de conduite. Nous pouvons concevoir une classe *Voiture* qui sera la représentation d'une voiture dans notre application.

```
package dev.gayerie.conduite;

public class Voiture {

    private final String marque;
    private float vitesse;

    public Voiture(String marque) {
        this.marque = marque;
    }

    // ...

}
```

Mais nous pouvons également rendre possible la simulation d'une moto. Dans ce cas, nous aurons également besoin d'une classe *Moto*.

```
package dev.gayerie.conduite;

public class Moto {

    private final String marque;
```

```

private float vitesse;

public Moto(String marque) {
    this.marque = marque;
}

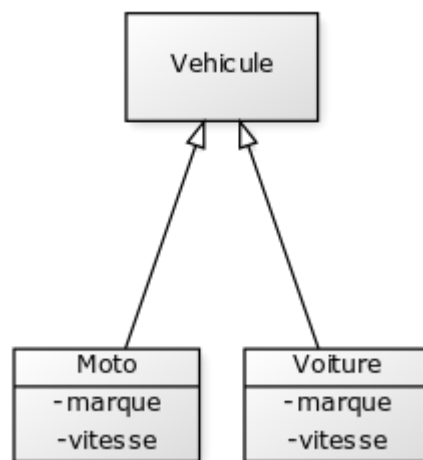
// ...

}

```

On se rend vite compte qu'au stade de notre développement, une voiture et une moto représentent la même chose. Faut-il alors créer deux classes différentes ? En programmation objet, il n'y a pas de réponse toute faite à cette question. Mais si notre application gère, par exemple, le type de permis de conduire, il serait judicieux d'avoir des représentations différentes pour ces types de véhicule car ils nécessitent des permis de conduire différents. Si mon application de simulation permet de faire se déplacer des objets de ces classes, alors il va peut-être falloir autoriser les objets de type *Voiture* à aller en marche arrière mais pas les objets de type *Moto*. Bref, comme souvent en programmation objet, on se retrouve avec des classes qui ont des notions en commun (dans notre exemple, la vitesse et la marque), tout en ayant leurs propres spécificités.

Pour ce type de relations, nous pouvons utiliser l'héritage pour faire apparaître une classe représentant une notion plus générale ou plus abstraite. Dans notre exemple, il pourrait s'agir de la classe *Vehicule*. Les classes *Voiture* et *Moto* peuvent *hériter* de cette nouvelle classe puisqu'une voiture **est un** véhicule et une moto **est un** véhicule.



En Java, l'héritage est indiqué par le mot clé **extends** après le nom de la classe. On dit donc qu'une classe *étend* une autre. La classe qui est étendue est appelée *classe mère* ou *classe parente* et la classe qui étend est appelée *classe fille* ou *classe enfant*.

```

package dev.gayerie.conduite;

public class Vehicule {

    // ...

}

package dev.gayerie.conduite;

public class Voiture extends Vehicule {

```

```

private final String marque;
private float vitesse;

public Voiture(String marque) {
    this.marque = marque;
}

// ...
}
package dev.gayerie.conduite;

public class Moto extends Vehicule {

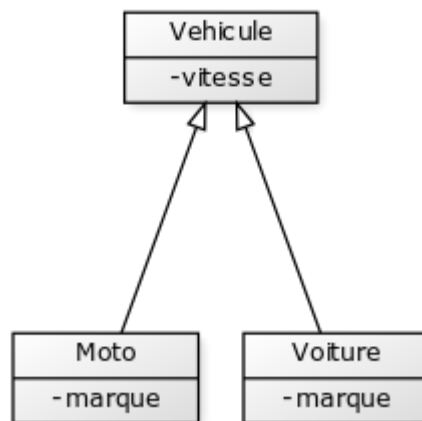
    private final String marque;
    private float vitesse;

    public Moto(String marque) {
        this.marque = marque;
    }

    // ...
}

```

Le terme d'héritage vient du fait qu'une classe qui en étend une autre *hérite* de la définition de sa classe parente et notamment de ses attributs et de ses méthodes. Par exemple, les classes *Voiture* et *Moto* ont en commun la déclaration de l'attribut *vitesse*. Cet attribut semble donc faire partie de la généralisation commune de *Vehicule*.



```

package dev.gayerie.conduite;

public class Vehicule {

    private float vitesse;

    // ...
}
package dev.gayerie.conduite;

public class Voiture extends Vehicule {

```

```

    private final String marque;

    public Voiture(String marque) {
        this.marque = marque;
    }

    // ...
}
package dev.gayerie.conduite;

public class Moto extends Vehicule {

    private final String marque;

    public Moto(String marque) {
        this.marque = marque;
    }

    // ...

}

```

Il est maintenant possible d'ajouter les méthodes *accélérer* et *décélérer* à la classe *Vehicule* et les classes *Voiture* et *Moto* en hériteront.

```

package dev.gayerie.conduite;

public class Vehicule {

    private float vitesse;

    public void accelerer(float deltavitesse) {
        this.vitesse += deltavitesse;
    }

    public void decelerer(float deltavitesse) {
        this.vitesse = Math.max(this.vitesse - deltavitesse, 0f);
    }

    // ...

}

```

Tous les véhicules de cette application peuvent maintenant accélérer et décélérer.

```

package dev.gayerie.conduite;

public class AppliSimple {

    public static void main(String[] args) {
        Voiture voiture = new Voiture("DeLorean");
        voiture.accelerer(88);

        Moto moto = new Moto("Kanedo");
        moto.accelerer(120);
    }

}

```

## Héritage et constructeur

Dans notre exemple précédent, l'attribut *marque* pourrait tout aussi bien être mutualisé dans la classe *Vehicule*. Cependant, il va falloir tenir compte des constructeurs de *Voiture* et *Moto* qui garantissent une initialisation de cet attribut à partir du paramètre.

En Java, nous avons vu qu'un constructeur peut appeler un autre constructeur déclaré dans la même classe grâce au mot-clé *this*. De la même manière, un constructeur peut appeler un constructeur de sa classe parente grâce au mot-clé **super**. Il doit respecter les mêmes contraintes :

- Un constructeur ne peut appeler qu'un constructeur.
- L'appel au constructeur doit être la première instruction du constructeur.

Il est donc possible de déclarer un constructeur dans la classe *Vehicule* et d'appeler ce constructeur depuis les constructeurs de *Voiture* et *Moto*.

```

package dev.gayerie.conduite;

public class Vehicule {

    private final String marque;
    private float vitesse;

    public Vehicule(String marque) {
        this.marque = marque;
    }

    public void accelérer(float deltaVitesse) {
        this.vitesse += deltaVitesse;
    }

    public void decelerer(float deltaVitesse) {
        this.vitesse = Math.max(this.vitesse - deltaVitesse, 0f);
    }

    // ...

}

package dev.gayerie.conduite;

```

```

public class Voiture extends Vehicule {

    public Voiture(String marque) {
        super(marque);
    }

    // ...

}

package dev.gayerie.conduite;

public class Moto extends Vehicule {

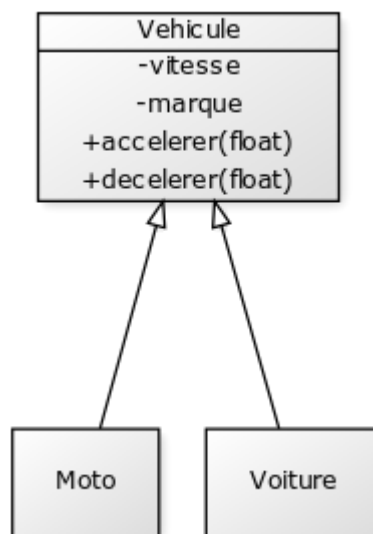
    public Moto(String marque) {
        super(marque);
    }

    // ...

}

```

*Voiture* et *Moto* peuvent maintenant proposer leurs propres méthodes et attributs tout en ayant en commun les mêmes méthodes et attributs que la classe *Vehicule*.



En java, si votre constructeur n'appelle aucun constructeur, alors le compilateur génèrera une instruction d'appel au constructeur sans paramètre de la classe parente.

Si vous créez la classe suivante :

```

package dev.gayerie.simple.test

public class MaClasse {

    public MaClasse() {
    }

}

```

Le compilateur génèrera le bytecode correspondant au code suivant :

```
package dev.gayerie.simple.test

public class MaClasse extends Object {

    public MaClasse() {
        super();
    }

}
```

Si vous omettez d'appeler un constructeur, alors le compilateur part du principe qu'il en existe un de disponible dans la classe parente et que ce constructeur ne prend pas de paramètre. Ainsi, Java garantit qu'un constructeur de la classe parente est toujours appelé avant l'exécution du constructeur courant. Cela signifie que, lors de la création d'un objet, on commence toujours par initialiser la classe la plus haute dans la hiérarchie d'héritage.

Ce qui peut sembler surprenant dans l'exemple précédent est que la classe *MaClasse* ne déclare pas de classe parente mais que le compilateur va forcer un héritage.

## Héritage simple : Object

Java ne supporte pas l'héritage multiple. Soit le développeur déclare avec le mot-clé **extends** une seule classe parente, soit le compilateur part du principe que la classe hérite de la classe [Object](#). Toutes les classes en Java ont une classe parente (hormis la classe [Object](#)). L'arbre d'héritage en Java ne possède qu'une seule classe racine : la classe [Object](#).

Note

C'est la classe [Object](#) qui déclare notamment les méthodes [toString](#) et [equals](#). Voilà pourquoi tous les objets Java peuvent avoir par défaut une représentation sous forme de chaîne de caractères et qu'ils peuvent être comparés aux autres.

## Héritage et affectation

L'héritage introduit la notion de *substituabilité* entre la classe enfant et la classe parente. Une classe enfant a son propre type mais partage également le même type que sa classe parente.

Pour notre exemple, cela signifie que l'on peut affecter à une variable de type *Vehicule*, une instance de *Voiture* ou une instance de *Moto* :

```
Vehicule vehicule = null;
vehicule = new Voiture("DeLorean");
vehicule = new Moto("Kameda");
```

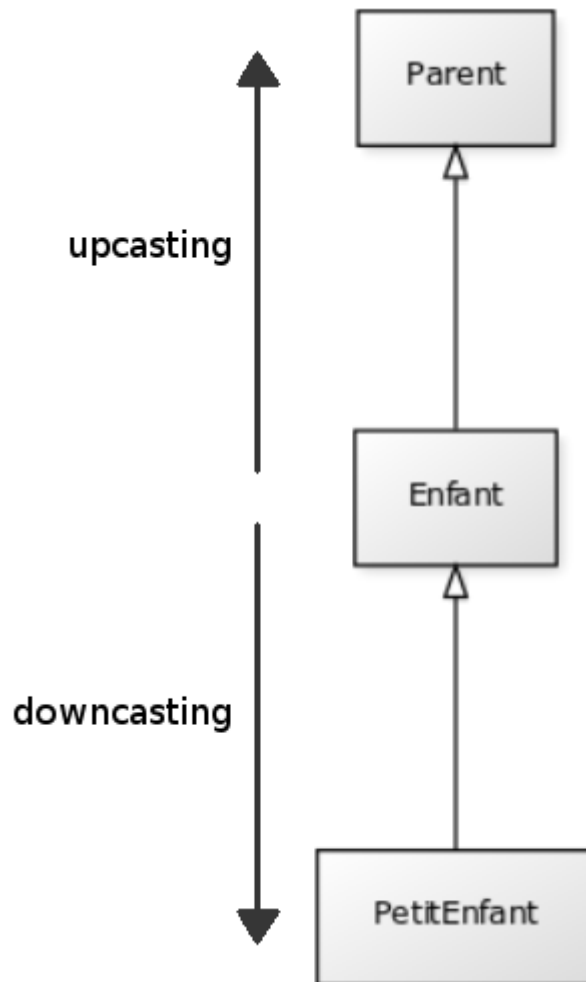
Cette possibilité introduit une abstraction importante dans la programmation objet. Si une partie d'un programme a besoin d'une instance de type *Vehicule* pour s'exécuter, alors cela signifie qu'une instance de n'importe quelle classe héritant directement ou indirectement de *Vehicule* peut être utilisée.

Lorsqu'on crée une classe par héritage, cela signifie qu'il faut faire attention à ne pas altérer le comportement attendu par les utilisateurs de la classe parente.

Note

L'acronyme [SOLID](#) proposé par Robert Martin regroupe 5 principes importants de la programmation objet. La lettre L désigne le [Liskov substitution principle](#) (principe de substitution de Liskov) qui décrit ce principe de substitution entre un type et son sous-type et les contraintes qui en découlent pour la conception objet.

Le principe de substituabilité est une application du transtypage (*casting*). Comme pour les types primitifs, il est possible d'affecter une référence d'un objet à une variable, attribut ou paramètre d'un type différent. Pour que cette affectation soit possible il faut que les deux types fassent partie de la même hiérarchie d'héritage.



Si le type d'arrivée correspond à un type parent, on parle d'*upcasting* (transtypage vers le haut). Si le type d'arrivée correspond à un type enfant, on parle de *downcasting* (transtypage vers le bas).

À partir du moment où l'implémentation des classes respectent le [principe de substitution de Liskov](#), l'upcasting est une opération sûre en programmation objet. Voilà pourquoi, il est possible d'affecter des instances de type *Voiture* à des variables de type *Vehicule*.

Note

Comme Java se base sur une hiérarchie à racine unique, toutes les classes héritent directement ou indirectement de [Object](#). Donc, toute instance peut être affectée à une variable, un attribut ou un paramètre de type [Object](#).

```
Object obj = null;
obj = new voiture("DeLorean");
obj = "ceci est une chaine de caractère";
obj = Integer.valueOf(1);
```



À l'inverse, le downcasting n'est pas une opération sûre en programmation objet. Prenons l'exemple trivial suivant :

```
vehicule vehicule = new voiture("DeLorean");  
Moto moto = vehicule; // ERREUR
```

La variable *vehicule* référence un objet de type *Voiture*, il n'est donc pas possible d'affecter cet objet à une variable de type *Moto*. Pour cette raison, le langage Java, n'autorise pas par défaut le downcasting : l'exemple ci-dessus ne compilera pas. Il est cependant possible de forcer le transtypage en utilisant la même syntaxe que pour les types primitifs.

```
vehicule vehicule = new voiture("DeLorean");  
voiture voiture = (voiture) vehicule;  
Moto moto = (Moto) vehicule; // ERREUR
```

Le code précédent compile puisque le développeur déclare explicitement le downcasting. Cependant, l'affectation à la ligne 3 est erronée puisque la variable *vehicule* référence une instance de *Voiture* que l'on veut affecter à une variable de type *Moto*. Pour les types primitifs, un transtypage invalide conduit à une possible perte de données. Par contre, pour des objets, un transtypage invalide génère à l'exécution une erreur de type [java.lang.ClassCastException](https://docs.oracle.com/javase/7/docs/api/java/lang/ClassCastException.html).

## Le mot-clé instanceof

Il est possible de découvrir à l'exécution si une variable, un attribut ou un paramètre est d'un type attendu, cela permet de contrôler les opérations de downcasting et d'éviter des erreurs d'exécution. Pour cela, le mot-clé **instanceof** retourne **true** si l'opérande à gauche est d'un type compatible avec l'opérande à droite.

1 2 3 4 5 6 7 8 9 10 11	<pre>vehicule vehicule = new voiture("DeLorean"); if (vehicule instanceof voiture) {     voiture voiture = (voiture) vehicule; // ... } if (vehicule instanceof Moto) {     Moto moto = (Moto) vehicule; // ... }</pre>

Le code ci-dessus s'exécutera sans erreur. À la ligne 8, comme la variable *vehicule* ne référence pas un objet compatible avec le type *Moto*, **instanceof** retournera **false**, empêchant ainsi le bloc de s'exécuter. Une opération de downcasting devrait toujours être contrôlée par une expression **instanceof** et le programme devrait être capable de se comporter correctement si l'instruction **instanceof** retourne **false**.

Prudence

Si le recours à **instanceof** permet de rendre les applications plus robustes, il n'en reste pas moins que les opérations de downcasting doivent rester l'exception dans un programme. Un recours systématique au downcasting est souvent le signe d'une mauvaise conception objet.

# La portée `protected`

Précédemment, nous avons introduit la classe *Vehicule* et nous avons pu l'utiliser pour mutualiser la déclaration des attributs *vitesse* et *marque*. Ces attributs ont été déclarés comme *private*. Donc ils ne sont accessibles que depuis la classe *Vehicule*. Même si la classe *Voiture* hérite des attributs et des méthodes de *Vehicule*, elles ne peuvent pas accéder aux attributs et aux méthodes privés des classes parentes. Imaginons maintenant que nous souhaitons ajouter la méthode *reculer*. Comme nous ne souhaitons pas fournir cette possibilité aux objets de la classe *Moto*, nous voulons ajouter cette méthode uniquement à la classe *Voiture*.

```
package dev.gayerie.conduite;

public class Voiture extends Vehicule {

    public Voiture(String marque) {
        super(marque);
    }

    public void reculer(float vitesse) {
        this.vitesse = -vitesse;
    }

    // ...

}
```

Le code précédent ne peut pas accéder à l'attribut *vitesse* déclaré dans la classe parente car il a été déclaré avec une portée **private**.

En Java, il existe une quatrième portée : la portée **protected**. Les attributs et les méthodes déclarés avec la portée **protected** sont accessibles par les membres du même package et par les classes filles. Ainsi en modifiant la déclaration de la classe *Vehicule* :

```
package dev.gayerie.conduite;

public class Vehicule {

    private final String marque;
    protected float vitesse;

    public Vehicule(String marque) {
        this.marque = marque;
    }

    public void accélérer(float deltaVitesse) {
        this.vitesse += deltaVitesse;
    }

    public void ralentir(float deltaVitesse) {
        this.vitesse = Math.max(this.vitesse - deltaVitesse, 0f);
    }

    // ...

}
```

```
}
```

La classe *Voiture* pourra compiler car elle a maintenant accès à l'attribut *vitesse*.

Le tableau ci-dessous résume toutes les portées en Java en les triant de la moins restrictive à la plus restrictive.

type	mot-clé	Description
Publique	<b>public</b>	Accessible depuis n'importe quel point de l'application
Protégée	<b>protected</b>	Accessible uniquement depuis les classes du même package et les classes filles
Package		Accessible uniquement depuis les classes du même package
Privée	<b>private</b>	Accessible uniquement dans la classe de déclaration et les classes internes

La portée **protected** pose parfois un soucis de conception. En effet, on pourrait considérer que les portées de type privé et package sont inutiles et que tous les attributs peuvent être déclarés avec la portée *protected*. Cependant, en programmation objet, le [principe du ouvert/fermé](#) stipule qu'une classe devrait être ouverte en extension mais fermée en modification. Cela signifie que par héritage, les développeurs doivent pouvoir étendre les fonctionnalités d'une classe en créant un sous-type mais ne doivent pas pouvoir modifier significativement le comportement de la classe parente. Empêcher une sous-classe de modifier l'état d'un attribut en le déclarant **private** est une bonne façon d'éviter aux développeurs d'une sous-classe de modifier involontairement le comportement d'une classe.

Un règle simple consiste à systématiquement déclarer **private** les attributs d'une classe sauf si une raison évidente nous suggère de déclarer la portée **protected**.

Note

Dans l'exemple précédent, la déclaration de l'attribut *vitesse* comme **protected** est peu satisfaisante car toutes les classes filles ont maintenant accès à cet attribut : cela n'est pas conforme au [principe du ouvert/fermé](#). Nous verrons au [chapitre suivant](#) qu'il existe une solution qui évite de modifier la portée de cet attribut.

Note

Le [principe du ouvert/fermé](#) (Open/close principle) représente le O dans l'acronyme [SOLID](#). Cet acronyme rassemble cinq notions fondamentales dans la conception objet.

## Héritage des méthodes et attributs de classe

Comme leur nom l'indique, les méthodes et les attributs de classe appartiennent à une classe. Il est possible d'accéder à une méthode de classe par la classe dans laquelle la méthode a été déclarée ou par n'importe quelle classe qui en hérite. Il en va de même pour les attributs de classe. Attention cependant, si l'attribut de classe est modifiable, sa valeur est partagée par l'ensemble des classes qui font partie de la hiérarchie d'héritage.

Un exemple classique est l'implémentation d'un compteur qui permet de savoir combien d'instances ont été créées. Il suffit de créer un compteur comme attribut de classe.

```

package dev.gayerie.conduite;

public class Vehicule {

    private static int nbInstances = 0;

    private final String marque;
    private float vitesse;

    public Vehicule(String marque) {
        ++Vehicule.nbInstances;
        this.marque = marque;
    }

    public static int getNbInstances() {
        return nbInstances;
    }

    // ...
}

```

Dans l'exemple précédent, nous avons ajouté l'attribut de classe *nbInstances* et la méthode de classe *getNbInstances*. L'attribut de classe est un compteur qui est incrémenté à chaque fois que le constructeur de *Vehicule* est appelé.

```

Voiture voiture = new Voiture("DeLorean");

Moto moto = new Moto("Kanedo");

System.out.println(Vehicule.getNbInstances()); // 2
System.out.println(Voiture.getNbInstances()); // 2
System.out.println(Moto.getNbInstances()); // 2

```

Dans l'exemple ci-dessus, la création d'une instance de *Voiture* et d'une instance de *Moto* incrémente le compteur *nbInstances*. L'appel à la méthode *getNbInstances* retournera le chiffre 2 quelle que soit la classe utilisée pour invoquer cette méthode. On voit ici, qu'il est parfois important, pour des raisons de lisibilité, d'utiliser la classe dans laquelle la méthode a été déclarée pour l'invoquer. En effet, une lecture rapide du code, pourrait nous faire croire que l'appel à *Voiture.getNbInstances* retourne le nombre d'instances de type *Voiture* créées alors qu'il s'agit du nombre d'instances de type *Vehicule* (donc incluant les instances de *Moto*).

## Héritage et final

En Java, il est possible de déclarer une classe **final**. Cela signifie qu'il est impossible d'étendre cette classe. Elle représente un élément terminal dans l'arborescence d'héritage.

```
package dev.gayerie.conduite;

public final class Moto extends Vehicule {

    public Moto(String marque) {
        super(marque);
    }

    // ...

}
```

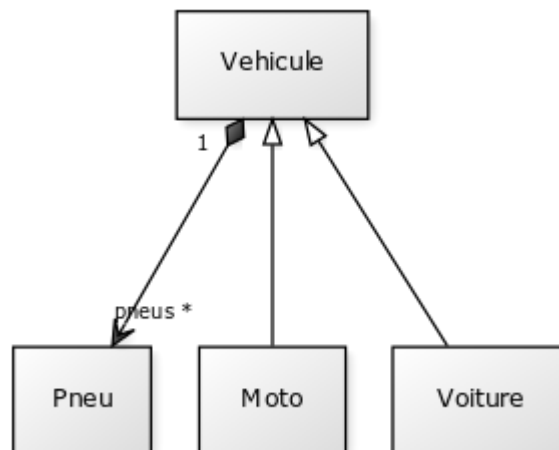
Dans l'exemple ci-dessus, la classe *Moto* est déclarée **final**. Donc il est maintenant impossible de déclarer une classe qui étende la classe *Moto*.

En raison de son impact très fort, la déclaration d'une classe comme **final** est réservée à des cas très particuliers. Un exemple est la classe [java.lang.String](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html). Cette classe est déclarée **final**. Il est donc impossible en Java de créer une classe qui hérite de [java.lang.String](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html). Les développeurs de l'API standard ont jugé qu'en raison de son importance, cette classe devait être fermée en extension afin d'éviter toute modification de comportement par héritage.

## La composition (has-a)

La composition est le type de relation le plus souvent utilisé en programmation objet. Elle indique une dépendance entre deux classes. L'une a besoin des services d'une autre pour réaliser sa fonction. La composition se fait en déclarant des attributs dans la classe.

Dans notre application de simulation de conduite, si nous introduisons une classe pour représenter des pneus.



```
package dev.gayerie.conduite;

public class Pneu {

    private float coefficientAdherence;

    // ..

}
```

Alors, nous pouvons indiquer que les véhicules **ont des** pneus.

```

package dev.gayerie.conduite;

public class Vehicule {

    private final String marque;
    protected float vitesse;
    protected Pneu[] pneus;

    public Vehicule(String marque) {
        this.marque = marque;
    }

    public Pneu[] getPneus() {
        return this.pneus;
    }

    // ...

}

package dev.gayerie.conduite;

public class Voiture extends Vehicule {

    public Voiture(String marque) {
        super(marque);
        this.pneus = new Pneu[] {new Pneu(), new Pneu(), new Pneu(), new Pneu()};
    }

    // ...

}

package dev.gayerie.conduite;

public class Moto extends Vehicule {

    public Moto(String marque) {
        super(marque);
        this.pneus = new Pneu[] {new Pneu(), new Pneu()};
    }

    // ...

}

```

## Le polymorphisme

Le polymorphisme est un mécanisme important dans la programmation objet. Il permet de modifier le comportement d'une classe fille par rapport à sa classe mère. Le polymorphisme permet d'utiliser l'héritage comme un mécanisme d'extension en adaptant le comportement des objets.

# Principe du polymorphisme

Prenons l'exemple de la classe *Animal*. Cette classe offre une méthode *crier*. Pour simplifier notre exemple, la méthode se contente d'écrire le cri de l'animal sur la sortie standard.

```
package dev.gayerie.animal;

public class Animal {

    public void crier() {
        System.out.println("un cri d'animal");
    }

}
```

Nous disposons également des classes *Chat* et *Chien* qui héritent de la classe *Animal*.

```
package dev.gayerie.animal;

public class Chat extends Animal {

}

package dev.gayerie.animal;

public class Chien extends Animal {

}
```

Ces deux classes sont une spécialisation de la classe *Animal*. À ce titre, elles peuvent **redéfinir** (*override*) la méthode *crier*.

```
package dev.gayerie.animal;

public class Chat extends Animal {

    public void crier() {
        System.out.println("Miaou !");
    }

}

package dev.gayerie.animal;

public class Chien extends Animal {

    public void crier() {
        System.out.println("Whouaf whouaf !");
    }

}
```

Chaque classe fille change le comportement de la méthode *crier*. Cela signifie qu'un objet de type *Chien* pour lequel on invoque la méthode *crier* ne fournira pas le même comportement qu'un objet de type *Chat*. Et cela, quel que soit le type de la variable qui référence ces objets.

```
Animal animal = new Animal();
animal.crier(); // affiche "un cri d'animal"

Chat chat = new Chat();
chat.crier(); // affiche "Miaou !"

Chien chien = new Chien();
chien.crier(); // affiche "whouaf whouaf !"

animal = chat;
animal.crier(); // affiche "Miaou !"

animal = chien;
animal.crier(); // affiche "whouaf whouaf !"
```

L'exemple de code ci-dessus montre que l'implémentation de la méthode *crier* dépend du type réel de l'objet et non pas du type de la variable qui le référence.

## Une exception : les méthodes privées

Les méthodes de portée **private** ne supportent pas le polymorphisme. En effet, une méthode de portée **private** n'est accessible uniquement que par la classe qui la déclare. Donc si une classe mère et une classe fille définissent une méthode **private** avec la même signature, le compilateur les traitera comme deux méthodes différentes.

## Redéfinition et signature de méthode

Le principe du polymorphisme repose en Java sur la redéfinition de méthodes. Pour que la redéfinition fonctionne, il faut que la méthode qui redéfinit possède une signature *correspondante* à celle de la méthode originale.

Le cas le plus simple est celui de l'exemple précédent. Les méthodes ont exactement la même signature : même portée, même type de retour, même nom et mêmes paramètres.

Cependant, la méthode qui redéfinit peut avoir une signature légèrement différente.

Une méthode qui redéfinit, peut avoir une portée différente si et seulement si, celle-ci est plus permissive que celle de la méthode d'origine. Il est donc possible d'augmenter la portée de la méthode mais jamais de la réduire :

- Une méthode de portée package peut être redéfinie avec la portée package mais aussi **protected** ou **public**.
- Une méthode de portée **protected** peut être redéfinie avec la portée **protected** ou **public**.
- Une méthode de portée **public** ne peut être redéfinie qu'avec la portée **public**.

Le changement de portée dans la redéfinition sert la plupart du temps à placer une implémentation dans la classe parente mais à laisser les classes filles qui le désirent offrir publiquement l'accès à cette méthode.



Au [chapitre précédent](#), nous avons introduit les classes *Vehicule*, *Voiture* et *Moto*. En partant du principe que seules les instances de *Voiture* peuvent offrir la méthode *reculer*, nous avons ajouté cette méthode dans la classe *Voiture*. Pour cela, nous avons dû modifier l'implémentation de la classe *Vehicule* en utilisant une portée **protected** pour l'attribut *vitesse*. Nous avons alors vu que cela n'était pas totalement conforme au [principe du ouvert/fermé](#).

```
package dev.gayerie.conduite;

public class Vehicule {

    private final String marque;
    protected float vitesse;

    public Vehicule(String marque) {
        this.marque = marque;
    }

    public void accelerer(float deltaVitesse) {
        this.vitesse += deltaVitesse;
    }

    public void decelerer(float deltaVitesse) {
        this.vitesse = Math.max(this.vitesse - deltaVitesse, 0f);
    }

    // ...
}

package dev.gayerie.conduite;

public class Voiture extends Vehicule {

    public Voiture(String marque) {
        super(marque);
    }

    public void reculer(float vitesse) {
        this.vitesse = -vitesse;
    }

    // ...
}
```

Nous pouvons maintenant revoir notre implémentation. En fait, c'est la méthode *reculer* qui doit être déclarée dans la classe *Vehicule* avec une portée **protected**. La classe *Voiture* peut se limiter à redéfinir cette méthode en la rendant **public**.

```
package dev.gayerie.conduite;

public class Vehicule {

    private final String marque;
    private float vitesse;
```

```

public Vehicule(String marque) {
    this.marque = marque;
}

public void accelerer(float deltaVitesse) {
    this.vitesse += deltaVitesse;
}

public void decelerer(float deltaVitesse) {
    this.vitesse = Math.max(this.vitesse - deltaVitesse, 0f);
}

protected void reculer(float vitesse) {
    this.vitesse = -vitesse;
}

// ...
}
package dev.gayerie.conduite;

public class Voiture extends Vehicule {

    public Voiture(String marque) {
        super(marque);
    }

    public void reculer(float vitesse) {
        super.reculer(vitesse);
    }

    // ...
}

```

Dans l'exemple ci-dessus, le mot-clé **super** permet d'appeler l'implémentation de la méthode fournie par la classe *Vehicule*. Ainsi l'attribut *vitesse* peut rester de portée **private** et les classes filles de *Vehicule* peuvent ou non donner publiquement l'accès à la méthode *reculer*.

Une méthode qui redéfinit peut avoir un type de retour différent de la méthode d'origine à condition qu'il s'agisse d'une classe qui hérite du type de retour. Si nous reprenons l'exemple d'héritage des classes *Animal*, *Chien* et *Chat*, nous pouvons définir une classe *EleveurAnimal* qui permet de créer une instance de *Animal* quand on appelle la méthode *elever* :

```

package dev.gayerie.animal;

public class EleveurAnimal {

    public Animal elever() {
        return new Animal();
    }

}

```

Si maintenant nous créons la classe *EleveurChien* qui hérite de la classe *EleveurAnimale*, la redéfinition de la méthode *elever* peut déclarer qu'elle retourne un type *Chien*. Comme *Chien* hérite de *Animal*, la méthode qui redéfinit se présente comme une spécialisation.

```
package dev.gayerie.animal;

public class EleveurChien extends EleveurAnimal {

    public Chien elever() {
        return new Chien();
    }

}
```

## Le mot-clé super

La redéfinition de méthode masque la méthode de la classe parente. Cependant, nous avons vu précédemment avec l'exemple de la méthode *reculer* que l'implémentation d'une classe fille a la possibilité d'appeler une méthode de la classe parente en utilisant le mot-clé **super**. L'appel à l'implémentation parente est très utile lorsque l'on veut effectuer une action avant et/ou après sans avoir besoin de dupliquer le code d'origine.

```
package dev.gayerie.conduite;

public class Voiture extends Vehicule {

    public Voiture(String marque) {
        super(marque);
    }

    public void accélérer(float deltaVitesse) {
        // faire quelque chose avant

        super.accélérer(deltaVitesse);

        // faire quelque chose après
    }

    // ...

}
```

Il existe tout de même une limitation : si une méthode a été redéfinie plusieurs fois dans l'arborescence d'héritage, le mot-clé **super** ne permet d'appeler que l'implémentation de la classe parente. Si la classe *Voiture* a redéfini la méthode *accélérer* et que l'on crée la classe *VoitureDeCourse* héritant de la classe *Voiture*.

```
package dev.gayerie.conduite;

public class VoitureDeCourse extends Voiture {

    public VoitureDeCourse(String marque) {
        super(marque);
    }

}
```

```

}

public void accélérer(float deltatvitesse) {
    // faire quelque chose avant

    super.accélérer(deltavitesse);

    // faire quelque chose après
}

// ...

}

```

La redéfinition de la méthode *accélérer* peut appeler l'implémentation de *Voiture* mais il est impossible d'appeler directement l'implémentation d'origine de la classe *Vehicule* depuis la classe *VoitureDeCourse*.

## L'annotation @Override

Les annotations sont des types spéciaux en Java qui commence par **@**. Les annotations servent à ajouter une information sur une classe, un attribut, une méthode, un paramètre ou une variable. Une annotation apporte une information au moment de la compilation, du chargement de la classe dans la JVM ou lors de l'exécution du code. Le langage Java proprement dit utilise relativement peu les annotations. On trouve cependant l'annotation [@Override](#) qui est très utile pour le polymorphisme. Cette annotation s'ajoute au début de la signature d'une méthode pour préciser que cette méthode est une redéfinition d'une méthode héritée. Cela permet au compilateur de vérifier que la signature de la méthode correspond bien à une méthode d'une classe parente. Dans le cas contraire, la compilation échoue.

```

package dev.gayerie.conduite;

public class Voiture extends Vehicule {

    public Voiture(String marque) {
        super(marque);
    }

    @Override
    public void reculer(float vitesse) {
        super.reculer(vitesse);
    }

    // ...

}

```

## Les méthodes de classe

Les méthodes de classe (déclarées avec le mot-clé **static**) ne supportent pas la redéfinition. Si une classe fille déclare une méthode **static** avec la même signature que dans la classe parente, ces méthodes seront simplement vues par le compilateur comme deux méthodes distinctes.

```

package dev.gayerie;

public class Parent {

    public static void methodeDeClasse() {
        System.out.println("appel à la méthode de la classe Parent");
    }

}

package dev.gayerie;

public class Enfant extends Parent {

    public static void methodeDeClasse() {
        System.out.println("appel à la méthode de la classe Enfant");
    }

}

```

Dans le code ci-dessus, la classe *Enfant* hérite de la classe *Parent* et toutes deux implémentent une méthode **static** appelée *methodeDeClasse*. Le code suivant peut être source d'incompréhension :

```

Parent a = new Enfant();
a.methodeDeClasse();

Enfant b = new Enfant();
b.methodeDeClasse();

```

Le résultat de l'exécution de ce code est :

```

appel à la méthode de la classe Parent
appel à la méthode de la classe Enfant

```

Comme les méthodes sont **static**, la redéfinition ne s'applique pas et la méthode appelée dépend du type de la variable et non du type de l'objet référencé par la variable. Cet exemple illustre pourquoi il est très fortement conseillé d'appeler les méthodes **static** à partir du nom de la classe et non pas d'une variable afin d'éviter toute ambiguïté.

```

Parent.methodeDeClasse();
Enfant.methodeDeClasse();

```

## Méthode finale

Une méthode peut avoir le mot-clé **final** dans sa signature. Cela signifie que cette méthode ne peut plus être redéfinie par les classes qui en hériteront. Tenter de redéfinir une méthode déclarée **final** conduit à une erreur de compilation. L'utilisation du mot-clé **final** pour une méthode est réservée à des cas très spécifiques (et très rares). Par exemple si on souhaite garantir qu'une méthode aura toujours le même comportement même dans les classes qui en héritent.

Note

Même si les méthodes **static** n'autorisent pas la redéfinition, elles peuvent être déclarées **final**. Dans ce cas, il n'est pas possible d'ajouter une méthode de classe qui a la même signature dans les classes qui en héritent.

## Constructeur et polymorphisme

Les constructeurs sont des méthodes particulières qu'il n'est pas possible de redéfinir. Les constructeurs créent une séquence d'appel qui garantit qu'ils seront exécutés en commençant par la classe la plus haute dans la hiérarchie d'héritage. Puisque toutes les classes Java héritent de la classe [Object](#), cela signifie que le constructeur de [Object](#) est toujours appelé en premier.

Cependant un constructeur peut appeler une méthode et dans ce cas le polymorphisme s'applique. Comme les constructeurs sont appelés dans l'ordre de la hiérarchie d'héritage, cela signifie qu'un constructeur invoque toujours une méthode redéfinie avant que la classe fille qui l'implémente n'ait pu être initialisée.

Par exemple, si nous disposons d'une classe *VehiculeMotorise* qui redéfinit la méthode *accelerer* pour prendre en compte la consommation d'essence :

```
package dev.gayerie.conduite;

public class VehiculeMotorise extends Vehicule {

    private Moteur moteur;

    public VehiculeMotorise(String marque) {
        super(marque);
        this.moteur = new Moteur();
    }

    @Override
    public void accelerer(float deltaVitesse) {
        moteur.consommer(deltaVitesse);
        super.accelerer(deltaVitesse);
    }

    // ...

}
```

Si maintenant nous faisons évoluer la classe *Vehicule* pour créer une instance avec une vitesse minimale :

```
package dev.gayerie.conduite;

public class Vehicule {

    private final String marque;
    protected float vitesse;

    public Vehicule(String marque) {
        this.marque = marque;
        this.accelerer(10);
    }

}
```

```

public void accelerer(float deltaVitesse) {
    this.vitesse += deltaVitesse;
}

// ...

}

```

Que va-t-il se passer à l'exécution de ce code :

```

vehiculeMotorise vehiculeMotorise = new vehiculeMotorise("DeLorean");

```

Le constructeur de *VehiculeMotorise* commence par appeler le constructeur de *Vehicule*. Ce dernier appelle implicitement le constructeur de [Object](#) (qui ne fait rien) puis il initialise l'attribut *marque* et il appelle la méthode *accelerer*. Comme cette dernière est redéfinie, c'est en fait l'implémentation fournie par *VehiculeMotorise* qui est appelée. Cette implémentation commence par appeler une méthode sur l'attribut *moteur* qui n'a pas encore été initialisé. Donc sa valeur est nulle et donc la création d'une instance de *VehiculeMotorise* échoue systématiquement avec une erreur du type [NullPointerException](#).

On voit par cet exemple que l'appel de méthode dans un constructeur peut amener à des situations complexes. Il est fortement recommandé d'appeler dans un constructeur des méthodes dont le comportement ne peut pas être modifié par la redéfinition : soit des méthodes déclarées **private** soit des méthodes déclarées **final**.

## Masquage des attributs par héritage

Il est possible de déclarer dans une classe fille un attribut portant le même nom que dans la classe parente. Cependant ceci ne correspond ni à une redéfinition ni au principe du polymorphisme. L'attribut de la classe fille se contente de masquer l'attribut de la classe parente.

Si l'attribut est de portée **private**, créer un attribut avec le même nom dans une classe fille n'a aucun impact particulier. Cela permet d'isoler l'état interne de la classe parente par rapport à sa classe fille.

Si l'attribut est de portée package, **protected** ou **public** alors l'attribut de la classe parente est simplement masqué dans la classe fille. Si une classe fille veut accéder à l'attribut de la classe parente, elle peut le faire à travers le mot-clé **super**.

```

package dev.gayerie;

public class Personne {

    protected String nom;

    public Personne(String nom) {
        this.nom = nom;
    }

    // ...

}

package dev.gayerie;

```

```

public class HerosMasque extends Personne {

    private String nom;

    public HerosMasque(String nom, String nomHeros) {
        super(nom);
        this.nom = nomHeros;
    }

    @Override
    public String toString() {
        // mmmmh ! Pas très clair
        return this.nom + " dont l'identité secrète est " + super.nom;
    }

    // ...

}

```

En raison de la difficulté de compréhension que cela peut entraîner, il est préférable de ne jamais créer dans une classe fille un attribut portant le même nom que celui d'un attribut de portée package, **protected** ou **public** d'une de ses classes parentes.

## Le principe du ouvert/fermé

Le [principe du ouvert/fermé](#) stipule qu'une classe doit être conçue pour être ouverte en extension mais fermée en modification.

D'un côté, si une classe hérite d'une autre classe, elle doit pouvoir ajouter des nouveaux comportements avec de nouvelles méthodes. Par contre la redéfinition de méthode ne doit pas être utilisée pour créer une implémentation qui a un comportement trop différent de celui de la classe parente.

D'une autre côté, si une classe hérite d'une autre classe, elle ne doit pas pouvoir modifier le fonctionnement décrit par la classe parente. En Java, pour interdire de modifier le comportement d'une classe, on peut déclarer ses attributs **private** et les méthodes jugées les plus critiques peuvent être déclarées **final**.

## Les classes abstraites

Nous avons vu que l'héritage est un moyen de mutualiser du code dans une classe parente. Parfois cette classe représente une abstraction pour laquelle il n'y a pas vraiment de sens de créer une instance. Dans ce cas, on peut considérer que la généralisation est *abstraite*.

Note

Par opposition, on appelle *classe concrète* une classe qui n'est pas abstraite.

## Déclarer une classe abstraite

Si nous reprenons notre exemple de la classe Vehicule :

```

package dev.gayerie.conduite;

```



```

public class Vehicule {

    private final String marque;
    protected float vitesse;

    public Vehicule(String marque) {
        this.marque = marque;
    }

    public void accélérer(float deltaVitesse) {
        this.vitesse += deltaVitesse;
    }

    public void ralentir(float deltaVitesse) {
        this.vitesse = Math.max(this.vitesse - deltaVitesse, 0f);
    }

    // ...

}

```

Cette classe peut avoir plusieurs classes filles comme *Voiture* ou *Moto*. Finalement, la classe *Vehicule* permet de faire apparaître un type à travers lequel il sera possible de manipuler des instances de *Voiture* ou de *Moto*. Il y a peu d'intérêt dans ce contexte à créer une instance de *Vehicule*. Nous pouvons très facilement l'empêcher en déclarant par exemple le constructeur avec une portée **protected**. En Java, nous avons également la possibilité de déclarer cette classe comme abstraite (**abstract**).

```

package dev.gayerie.conduite;

public abstract class Vehicule {

    private final String marque;
    protected float vitesse;

    public Vehicule(String marque) {
        this.marque = marque;
    }

    public void accélérer(float deltaVitesse) {
        this.vitesse += deltaVitesse;
    }

    public void ralentir(float deltaVitesse) {
        this.vitesse = Math.max(this.vitesse - deltaVitesse, 0f);
    }

    // ...

}

```

Le mot-clé **abstract** ajouté dans la déclaration de la classe indique maintenant que cette classe représente une abstraction pour laquelle il n'est pas possible de créer directement une instance de ce type.

```
vehicule v = new vehicule("X"); // ERREUR DE COMPILATION : LA CLASSE EST  
ABSTRAITE
```

Note

En Java, il n'est pas possible de combiner **abstract** et **final** dans la déclaration d'une classe car cela n'aurait aucun sens. Une classe abstraite ne pouvant être instanciée, il faut nécessairement qu'il existe une ou des classes filles.

## Déclarer une méthode abstraite

Une classe abstraite peut déclarer des méthodes abstraites. Une méthode abstraite possède une signature mais pas de corps. Cela signifie qu'une classe qui hérite de cette méthode doit la redéfinir pour en fournir une implémentation (sauf si cette classe est elle-même abstraite).

Par exemple, un véhicule peut donner son nombre de roues. Plutôt que d'utiliser un attribut pour stocker le nombre de roues, il est possible de faire du nombre de roues une propriété abstraite de la classe.

```
package dev.gayerie.conduite;

public abstract class Vehicule {

    private final String marque;
    protected float vitesse;

    public Vehicule(String marque) {
        this.marque = marque;
    }

    public abstract int getNbRoues();

    // ...

}
```

Toutes les classes concrètes héritant de *Vehicule* doivent maintenant fournir une implémentation de la méthode *getNbRoues* pour pouvoir compiler.

```
package dev.gayerie.conduite;

public class Voiture extends Vehicule {

    public Voiture(String marque) {
        super(marque);
    }

    @Override
    public int getNbRoues() {
        return 4;
    }

    // ...

}
```

```

}
package dev.gayerie.conduite;

public class Moto extends Vehicule {

    public Moto(String marque) {
        super(marque);
    }

    @Override
    public int getNbRoues() {
        return 2;
    }

    // ...

}

```

Une méthode abstraite peut avoir plusieurs utilités. Comme dans l'exemple précédent, elle peut servir à gagner en abstraction dans notre modèle. Mais elle peut aussi permettre à une classe fille d'adapter le comportement d'un algorithme ou d'un composant logiciel.

```

package dev.gayerie.tableur;

public abstract class Tableur {

    public void mettreAJour() {
        tracerLignesEtColonnes();
        int premiereLigne = getPremiereLigneAffichee();
        int premiereColonne = getPremierColonneAffichee();
        int derniereLigne = getDerniereLigneAffichee();
        int derniereColonne = getDerniereColonneAffichee();

        for (int ligne = premiereLigne; ligne <= derniereLigne; ++ligne) {
            for (int colonne = premiereColonne; colonne <= derniereColonne; ++colonne)
            {
                String contenu = getContenu(ligne, colonne);
                afficherContenu(ligne, colonne, contenu);
            }
        }
    }

    protected abstract String getContenu(int ligne, int colonne);

    private void afficherContenu(int ligne, int colonne, String contenu) {
        // ...
    }

    private int getDerniereColonneAffichee() {
        // ...
    }

    private int getDerniereLigneAffichee() {
        // ...
    }
}

```

```

}

private int getPremierColonneAffichee() {
    // ...
}

private int getPremiereLigneAffichee() {
    // ...
}

private void tracerLignesEtColonnes() {
    // ...
}
}

```

Dans l'exemple ci-dessus, on imagine une classe *Tableur* qui permet d'afficher un tableau à l'écran en fonction des lignes et des colonnes visibles. Il s'agit d'une classe abstraite et les classes qui spécialisent cette classe doivent fournir une implémentation de la méthode abstraite *getContenu* afin de fournir le contenu de chaque cellule affichée par le tableur.

## La classe Object

Java est un langage qui ne supporte que l'héritage simple. L'arborescence d'héritage est un arbre dont la racine est la classe [Object](#). Si le développeur ne précise pas de classe parente dans la déclaration d'une classe, alors la classe hérite implicitement de [Object](#).

La classe [Object](#) fournit des méthodes communes à toutes les classes. Certaines de ces méthodes doivent être redéfinies dans les classes filles pour fonctionner correctement.

Note

Rien ne vous interdit de créer une instance de [Object](#).

```
object myObj = new object();
```

## La méthode equals

En Java, l'opérateur `==` sert à comparer les références. Il ne faut donc **jamais** l'utiliser pour comparer des objets. La comparaison d'objets se fait grâce à la méthode [equals](#) héritée de la classe [Object](#).

```

vehicule v1 = new Voiture("DeLorean");
vehicule v2 = new Moto("Kawasaki");

if (v1.equals(v1)) {
    System.out.println("v1 est identique à lui-même.");
}

if (v1.equals(v2)) {
    System.out.println("v1 est identique à v2.");
}

```

L'implémentation par défaut de [equals](#) fournie par [Object](#) compare les références entre elles. L'implémentation par défaut est donc simplement :

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

#### Prudence

Il ne faut pas déduire de l'implémentation par défaut qu'il est possible d'utiliser `==` pour comparer des objets. N'importe quelle classe héritant de la classe [Object](#) peut modifier ce comportement : à commencer par une des classes les plus utilisées en Java, la classe [String](#).

Parfois, l'implémentation par défaut peut suffire. C'est notamment le cas lorsque l'unicité en mémoire suffit à identifier un objet. Cependant, si nous ajoutons la notion de plaque d'immatriculation à notre classe *Vehicule* :

```
public class Vehicule {  
  
    private String immatriculation;  
    private final String marque;  
  
    public Vehicule(String immatriculation, String marque) {  
        this.immatriculation = immatriculation;  
        this.marque = marque;  
    }  
  
    // ...  
}
```

Alors l'attribut immatriculation introduit l'idée d'identification du véhicule. Il est donc judicieux de considérer que deux véhicules sont égaux s'ils ont la même immatriculation. Dans ce cas, il faut redéfinir la méthode [equals](#).

```
package dev.gayerie.conduite;  
  
public class Vehicule {  
  
    private String immatriculation;  
    private final String marque;  
  
    public Vehicule(String immatriculation, String marque) {  
        this.immatriculation = immatriculation;  
        this.marque = marque;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        if (! (obj instanceof Vehicule)) {  
            return false;  
        }  
        Vehicule vehicule = (Vehicule) obj;  
        return this.immatriculation != null &&
```

```

        this.immatriculation.equals(vehicule.immatriculation);

    }

    // ...

}

```

Dans l'exemple précédent, notez l'utilisation de **instanceof** pour vérifier que l'objet en paramètre est bien compatible avec le type *Vehicule* (sinon la méthode retourne **false**). En effet, la signature de [equals](#) impose que le paramètre soit de type [Object](#). Il est donc important de commencer par vérifier que le paramètre est d'un type acceptable pour la comparaison. Notez également, que l'implémentation est telle que deux véhicules n'ayant pas de plaque d'immatriculation ne sont pas identiques.

L'implémentation de [equals](#) doit être conforme à certaines règles pour s'assurer qu'elle fonctionnera correctement, notamment lorsqu'elle est utilisée par l'API standard ou par des bibliothèques tierces.

- ◦ Son implémentation doit être réflexive :  
       Pour x non nul, x.equals(x) doit être vrai
- ◦ Son implémentation doit être symétrique :  
       Si x.equals(y) est vrai alors y.equals(x) doit être vrai
- ◦ Son implémentation doit être transitive :  
       Pour x, y et z non nuls  
       Si x.equals(y) est vrai  
       Et si y.equals(z) est vrai  
       Alors x.equals(z) doit être vrai
- ◦ Son implémentation doit être consistante  
       Pour x et y non nuls  
       Si x.equals(y) est vrai alors il doit rester vrai tant que l'état de x et de y est inchangé.
- Si x est non nul alors x.equals(null) doit être faux.

Note

Il est parfois facile d'introduire un bug en Java.

```

if (x.equals(y)) {
    // ...
}

```

Le code ci-dessus ne teste pas la possibilité pour la variable x de valoir **null**, entraînant ainsi une erreur de type [NullPointerException](#). Il ne faut donc pas oublier de tester la valeur **null** :

```

if (x != null && x.equals(y)) {
    // ...
}

```

Lorsque l'un des deux termes est une constante, alors il est plus simple de placer la constante à gauche de l'expression de façon à éviter le problème de la nullité. En effet, [equals](#) doit retourner **false** si le paramètre vaut **null**. Cela est notamment très pratique pour comparer une chaîne de caractères avec une constante :

```
if ("Message à comparer".equals(msg)) {  
    // ...  
}
```

On peut aussi utiliser la classe outil [java.util.Objects](#) qui fournit la méthode de classe [equals\(Object, Object\)](#) pour prendre en charge le cas de la valeur **null**. Notez toutefois que [equals\(Object, Object\)](#) retourne **true** si les deux paramètres valent **null**.

## La méthode hashCode

La méthode [hashCode](#) est fournie pour l'utilisation de certains algorithmes, notamment pour l'utilisation de table de hachage. Le principe d'un algorithme de hachage est d'associer un identifiant à un objet. Cet identifiant doit être le même pour la durée de vie de l'objet. De plus, deux objets égaux doivent avoir le même code de hachage.

L'implémentation de cette méthode peut se révéler assez technique. En général, on se basera sur les attributs utilisés dans l'implémentation de la méthode [equals](#) pour en déduire le code de hachage.

Cette méthode ne doit être redéfinie que si cela est réellement utile. Par exemple si une instance de cette classe doit servir de clé pour une instance de [HashMap](#).

```
package dev.gayerie.conduite;  
  
public class Vehicule {  
  
    private String immatriculation;  
    private final String marque;  
  
    public Vehicule(String immatriculation, String marque) {  
        this.immatriculation = immatriculation;  
        this.marque = marque;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        if (! (obj instanceof Vehicule)) {  
            return false;  
        }  
        Vehicule vehicule = (Vehicule) obj;  
        return this.immatriculation != null &&  
            this.immatriculation.equals(vehicule.immatriculation);  
    }  
  
    @Override  
    public int hashCode() {  
        return immatriculation == null ? 0 : immatriculation.hashCode();  
    }  
  
    // ...  
}
```

# La méthode toString

---

La méthode [toString](#) est une méthode très utile, notamment pour le débogage et la production de log. Elle permet d'obtenir une représentation sous forme de chaîne de caractères d'un objet. Elle est implicitement appelée par le compilateur lorsqu'on concatène une chaîne de caractères avec un objet.

Par défaut l'implémentation de la méthode [toString](#) dans la classe [Object](#) retourne le type de l'objet suivi de @ suivi du code de hachage de l'objet. Il suffit de redéfinir cette méthode pour obtenir la représentation souhaitée.

```
package dev.gayerie.conduite;

public class Vehicule {

    private final String marque;

    public Vehicule(String marque) {
        this.marque = marque;
    }

    @Override
    public String toString() {
        return "Véhicule de marque " + marque;
    }

    // ...

}

Vehicule v = new Vehicule("DeLorean");

String msg = "Objet créé : " + v;

System.out.println(msg); // "Objet créé : Véhicule de marque DeLorean"
```

# La méthode finalize

---

La méthode [finalize](#) est appelée par le ramasse-miettes avant que l'objet ne soit supprimé et la mémoire récupérée. Redéfinir cette méthode donne donc l'opportunité au développeur de déclencher un traitement avant que l'objet ne disparaisse. Cependant, nous avons déjà vu dans le chapitre sur le [cycle de vie](#) que le fonctionnement du ramasse-miettes ne donne aucune garantie sur le fait que cette méthode sera appelée.

# La méthode clone

---

La méthode [clone](#) est utilisée pour cloner une instance, c'est-à-dire obtenir une copie d'un objet. Par défaut, elle est déclarée **protected** car toutes les classes ne désirent pas permettre de cloner une instance.

Pour qu'un objet soit clonable, sa classe doit implémenter l'interface marqueur [Cloneable](#). L'implémentation par défaut de la méthode dans [Object](#) consiste à jeter une exception [CloneNotSupportedException](#) si l'interface [Cloneable](#) n'est pas implémentée. Si l'interface est implémentée, alors la méthode crée une nouvelle instance de la classe et affecte la même valeur



que l'instance d'origine aux attributs de la nouvelle instance. L'implémentation par défaut de [clone](#) n'appelle pas les constructeurs pour créer la nouvelle instance.

Prudence

L'implémentation par défaut de la méthode [clone](#) ne réalise pas un clonage en profondeur. Cela signifie que si les attributs de la classe d'origine référencent des objets, les attributs du clone référenceront les mêmes objets. Si ce comportement n'est pas celui désiré, alors il faut fournir une nouvelle implémentation de la méthode [clone](#) dans la classe.

Note

Par défaut, tous les tableaux implémentent l'interface [Cloneable](#) et redéfinissent la méthode [clone](#) afin de la rendre **public**. On peut donc directement cloner des tableaux en Java si on désire en obtenir une copie.

```
int[] tableau = {1, 2, 3, 4};
int[] tableauClone = tableau.clone();
```

## La méthode getClass

La méthode [getClass](#) permet d'accéder à l'objet représentant la classe de l'instance. Cela signifie qu'un programme Java peut accéder par programmation à la définition de la classe d'une instance. Cette méthode est notamment très utilisée dans des usages avancés impliquant la *réflexivité*.

L'exemple ci-dessous, affiche le nom complet (c'est-à-dire en incluant son package) de la classe d'un objet :

```
vehicule v = new vehicule("DeLorean");

System.out.println(v.getClass().getName());
```

## Les méthodes de concurrence

La classe [Object](#) fournit un ensemble de méthodes qui sont utilisées pour l'échange de signaux dans la programmation concurrente. Il s'agit des méthodes [notify](#), [notifyAll](#) et [wait](#).

## La classe String

En Java, les chaînes de caractères sont des instances de la classe [String](#). Les chaînes de caractères écrites littéralement sont toujours délimitées par des guillemets :

```
"Hello world"
```

## String et tableau de caractères

Contrairement à d'autres langages de programmation, une chaîne de caractères ne peut pas être traitée comme un tableau. Si on souhaite accéder à un caractère de la chaîne à partir de son index, il faut utiliser la méthode [String.charAt](#). On peut ainsi parcourir les caractères d'une chaîne :

```
String s = "Hello world";

for (int i = 0; i < s.length(); ++i) {
    char c = s.charAt(i);
    System.out.println(c);
}
```

La méthode [String.length](#) permet de connaître le nombre de caractères dans la chaîne. Il n'est malheureusement pas possible d'utiliser un for amélioré pour parcourir les caractères d'une chaîne car la classe [String](#) n'implémente pas l'interface [Iterable](#). Par contre, il est possible d'obtenir un tableau des caractères avec la méthode [String.toCharArray](#). On peut alors parcourir ce tableau avec un for amélioré.

```
String s = "Hello world";

for (char c : s.toCharArray()) {
    System.out.println(c);
}
```

Note

La méthode [String.toCharArray](#) a l'inconvénient de créer un tableau de la même longueur que la chaîne et de copier un à un les caractères. Si votre programme manipule intensivement des chaînes de caractères de taille importante, cela peut être pénalisant pour les performances. Depuis Java 8, il existe avec une nouvelle solution à ce problème avec un impact mémoire quasi nul : l'utilisation des streams et des lambdas.

```
String s = "Hello world";
s.chars().forEach(c -> System.out.println((char)c));
```

## Quelques méthodes utilitaires

Voici ci-dessous, quelques méthodes utiles fournies par la classe [String](#). Reportez-vous à la documentation de la classe pour consulter la liste complète des méthodes.

### [String.equals](#)

Compare la chaîne de caractères avec une autre chaînes de caractères.

```
System.out.println("a".equals("a"));    // true
System.out.println("a".equals("ab"));   // false
System.out.println("ab".equals("AB"));  // false
```

### [String.equalsIgnoreCase](#)

Comme la méthode précédente sauf que deux chaînes qui ne diffèrent que par la casse seront considérées comme identiques.

```
System.out.println("a".equalsIgnoreCase("a"));    // true
System.out.println("a".equalsIgnoreCase("ab"));   // false
System.out.println("ab".equalsIgnoreCase("AB"));  // true
```

### [String.compareTo](#)

Compare la chaîne de caractères avec une autre chaînes de caractères. La comparaison se fait suivant la taille des chaînes et l'ordre lexicographique des caractères. Cette méthode retourne 0 si les deux chaînes sont identiques, une valeur négative si la première est inférieure à la seconde et une valeur positive si la première est plus grande que la seconde.

```
System.out.println("a".compareTo("a"));    // 0
System.out.println("a".compareTo("ab"));   // < 0
System.out.println("ab".compareTo("a"));   // > 0
System.out.println("ab".compareTo("az"));  // < 0
System.out.println("ab".compareTo("AB"));  // > 0
```

### [String.compareToIgnoreCase](#)

Comme la méthode précédente sauf que deux chaînes qui ne diffèrent que par la casse seront considérées comme identiques.

```
System.out.println("a".compareToIgnoreCase("a"));    // 0
System.out.println("a".compareToIgnoreCase("ab"));   // < 0
System.out.println("ab".compareToIgnoreCase("a"));   // > 0
System.out.println("ab".compareToIgnoreCase("az"));  // < 0
System.out.println("ab".compareToIgnoreCase("AB"));  // 0
```

### [String.concat](#)

Concatène les deux chaînes dans une troisième. Cette méthode est équivalente à l'utilisation de l'opérateur +.

```
String s = "Hello".concat(" ").concat("world"); // "Hello world"
```

### [String.contains](#)

Retourne **true** si la chaîne contient une séquence de caractères donnée.

```
boolean b = "Hello world".contains("world"); // true
b = "Hello world".contains("Monde");          // false
```

### [String.endsWith](#)

Retourne **true** si la chaîne se termine par une chaîne de caractères donnée.

```
boolean b = "Hello world".endsWith("world"); // true
b = "Hello world".endsWith("Hello");          // false
```

### [String.startsWith](#)

Retourne **true** si la chaîne commence par une chaîne de caractères donnée.

```
boolean b = "Hello world".startsWith("Hello"); // true
b = "Hello world".startsWith("world");          // false
```

### [String.isEmpty](#)

Retourne **true** si la chaîne est la chaîne vide (*length()* vaut 0)

```
boolean b = "".isEmpty();    // true
b = "Hello world".isEmpty(); // false
```

### [String.length](#)

Retourne le nombre de caractères dans la chaîne.

```
int n = "Hello world".length(); // 11
```

### [String.replace](#)

Remplace un caractère par un autre dans une nouvelle chaîne de caractères.

```
String s = "Hello world".replace('l', 'x'); // "Hexxo worxd"
```

Cette méthode est surchargée pour accepter des chaînes de caractères comme paramètres.

```
String s = "Hello world".replace(" world", ""); // "Hello"
```

### [String.substring](#)

Crée une nouvelle sous-chaîne à partir de l'index de début et jusqu'à l'index de fin (non inclus).

```
String s = "Hello world".substring(2, 4); // "ll"
s = "Hello world".substring(0, 5);        // "Hello"
```

### [String.toLowerCase](#)

Crée une chaîne de caractères équivalente en minuscules.

```
String s = "Hello world".toLowerCase(); // "hello world"
```

### [String.toUpperCase](#)

Crée une chaîne de caractères équivalente en majuscules.

```
String s = "Hello world".toUpperCase(); // "HELLO WORLD"
```

### [String.trim](#)

Crée une nouvelle chaîne de caractères en supprimant les espaces au début et à la fin.

```
String s = "    Hello world    ".trim(); // "Hello world"
```

## Construction d'une instance de String

La classe [String](#) possède plusieurs constructeurs qui permettent de créer une chaîne de caractères avec l'opérateur **new**.

```
String s1 = new String(); // chaîne vide

String hello = "Hello world";
String s2 = new String(hello); // copie d'un chaîne

char[] tableau = {'H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd'};
String s3 = new String(tableau); // à partir d'un tableau de caractères.

byte[] tableauCode = {72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100};
String s4 = new String(tableauCode); // à partir d'un tableau de code UTF-16
```

## Immutabilité des chaînes de caractères

Les instances de la classe [String](#) sont immutables. Cela signifie qu'il est impossible d'altérer le contenu de la chaîne de caractères une fois qu'elle a été créée. Si vous reprenez la liste des méthodes ci-dessus, vous verrez que toutes les méthodes qui *modifient* le contenu de la chaîne de caractères crée une nouvelle chaîne de caractères et laissent intacte la chaîne d'origine. Cela signifie que des opérations intensives sur les chaînes de caractères peuvent être pénalisantes pour le temps d'exécution et l'occupation mémoire puisque toutes les opérations se font finalement par copie.

Nous avons vu qu'il n'existe pas réellement de constante en Java mais uniquement des attributs déclarés avec **static** et **final**. Cette immutabilité permet de garantir qu'une variable de [String](#) déclarée **static** et **final** ne peut plus être modifiée.

La JVM tire également partie de cette immutabilité afin de réaliser des optimisations de place mémoire. Si par exemple vous écrivez plusieurs fois dans le code source la même chaîne de caractères, la JVM ne créera pas un nouvel emplacement mémoire pour cette chaîne. Ainsi, il est possible d'avoir des comportements assez déroutants au premier abord en Java :

1 2	String s = "test"; System.out.println(s == "test"); // true
3 4	System.out.println(s == new String("test")); // false
5	System.out.println(new String("test") == "test"); // false

Dans le code ci-dessus, on utilise l'opérateur `==` donc on ne compare pas le contenu des chaînes de caractères mais la référence des objets. La chaîne de caractères « test » apparaît plusieurs fois dans le code. Donc quand la JVM va charger la classe qui contient ce code, elle ne créera qu'une et une seule fois l'instance de [String](#) pour « test ». Voilà pourquoi la ligne 3 affiche **true**. Le contenu de la variable `s` référence exactement la même instance de [String](#). Par contre, les lignes 4 et 5 créent explicitement une nouvelle instance de [String](#) avec l'opérateur **new**. Il s'agit donc de nouveaux objets avec de nouvelles références.

## La classe StringBuilder

La classe [StringBuilder](#) permet de construire une chaîne de caractères par ajout (concaténation) ou insertion d'éléments. Il est même possible de supprimer des portions. La quasi totalité des méthodes de la classe [StringBuilder](#) retourne l'instance courante du [StringBuilder](#) ce qui permet de chaîner les appels en une seule instruction. Pour obtenir la chaîne de caractères, il suffit d'appeler la méthode [StringBuilder.toString](#).

```

StringBuilder sb = new StringBuilder();
sb.append("Hello")
  .append(" ")
  .append("world")
  .insert(5, " the")    // On insère la chaîne à l'index 5
  .append('!');
System.out.println(sb); // "Hello the world!"

sb.reverse();
System.out.println(sb); // "!dlrow eht olleH"

sb.deleteCharAt(0).reverse();
System.out.println(sb); // "Hello the world"

```

La classe [StringBuilder](#) permet de pallier au fait que les instances de la classe [String](#) sont immutables. D'ailleurs, l'opérateur + de concaténation de chaînes n'est qu'un sucre syntaxique, le compilateur le remplace par une utilisation de la classe [StringBuilder](#).

```

String s1 = "Hello";
String s2 = "the";
String s3 = "world";
String message = s1 + " " + s2 + " " + s3; // "Hello the world"

```

Le code ci-dessus sera en fait interprété par le compilateur comme ceci :

```

String s1 = "Hello";
String s2 = "the";
String s3 = "world";
String message = new StringBuilder().append(s1).append(" ").append(s2).append(" ")
    .append(s3).toString();

```

## Formatage de texte

La méthode de classe [String.format](#) permet de passer une chaîne de caractères décrivant un formatage ainsi que plusieurs objets correspondant aux valeurs à mettre en forme.

Note

Cette section ne présente que les notions de base de la mise en forme des chaînes de caractères. Reportez-vous à la documentation de la classe [Formatter](#) qui est utilisée par la méthode [String.format](#).

```

String who = "the world";
String message = String.format("Hello %s!", who);

System.out.println(message); // "Hello the world!"

```

Dans l'exemple ci-dessus, la chaîne de formatage « Hello %s » contient un paramètre identifié par `%s` (s signifie que le paramètre attendu est de type [String](#)).

Un paramètre dans la chaîne de formatage peut contenir différentes informations :

```
%[index$][flags][taille]conversion
```

La conversion indique le type de donnée à convertir. L'information de conversion est obligatoire ! Attention, cela doit correspondre au type de la donnée effectivement passée en paramètre de [String.format](#). Quelques exemples de conversions acceptées :

conversion	type de données
s	Une chaîne de caractères (ou sinon la méthode <a href="#">toString</a> est appelée sur l'objet passé en paramètre)
c	Un caractère
b	Un booléen
d	Un nombre décimal ( <code>int</code> , <code>long</code> , <code>short</code> , <code>byte</code> )
f	Un nombre flottant ( <code>float</code> , <code>double</code> )
t	Une date ou un horaire. On peut préciser l'information spécifique que l'on souhaite extraire de la date : <code>te</code> pour le jour du mois, <code>tB</code> pour le nom du mois, <code>tY</code> pour l'année sur quatre décimales...
%	Le caractère %
n	Un retour à la ligne spécifique à la plate-forme sur laquelle le programme s'exécute

Les *flags* sont spécifiques au type de données et permettent de préciser le formatage. Par exemple pour les nombres, `0` permet de spécifier qu'il faut ajouter des zéros avant ou après si la donnée n'a pas la taille requise. Pour les nombres à virgule flottante, on peut utiliser `.` suivi d'un nombre pour indiquer le nombre maximum de chiffres après la virgule.

La taille donne le nombre minimum de caractères que la donnée doit occuper, une fois qu'elle est mise en forme.

L'index est la place du paramètre dans l'appel à la méthode [String.format](#). S'il n'est pas mentionné, on présume que les données sont dans l'ordre des paramètres passés à la méthode [String.format](#).

Quelques exemples d'appels à la méthode [String.format](#) :

```
String message = String.format("Nom : %10s%10s", "David", "Gayerie");

System.out.println(message); // "Nom :      David   Gayerie"
String message = String.format("Prix TTC : %08.2f %s", 12.5)

System.out.println(message); // "Prix TTC : 00012,50 euros"
int quantite = 12;
LocalDate now = LocalDate.now();

String message = String.format("quantité = %1$010d au %2$te %2$tB %2$tY",
quantite, now);

System.out.println(message); // "quantité = 0000000012 au 5 septembre 2017"
```

Il existe également une définition de la méthode [String.format](#) qui attend une instance de [Locale](#) en premier paramètre. La locale indique la langue du message et permet de formater les nombres, les dates, etc comme attendu.

```
int quantite = 12;
Locale now = Locale.now();

String message = String.format(Locale.ENGLISH, "quantity = %1$010d on %2$te %2$tB %2$tY", quantite, now);

System.out.println(message); // "quantity = 0000000012 on 5 september 2017"
```

Note

Il est également possible de formater des messages avec la classe [MessageFormat](#). Il s'agit d'une classe plus ancienne qui offre une syntaxe différente pour décrire les paramètres dans la chaîne de formatage.

## Les expressions régulières

Certaines méthodes de la classe [String](#) acceptent comme paramètre une [expression régulière](#) (*regular expression* ou *regex*). Une expression régulière permet d'exprimer avec des motifs un ensemble de chaînes de caractères possibles. Par exemple la méthode [String.matches](#) prend un paramètre de type [String](#) qui est interprété comme une expression régulière. Cette méthode retourne **true** si la chaîne de caractères est conforme à l'expression régulière passée en paramètre.

```
boolean match = "hello".matches("hello");
System.out.println(match); // true
```

L'intérêt des expressions régulières est qu'elles peuvent contenir des classes de caractères, c'est-à-dire des caractères qui sont interprétés comme représentant un ensemble de caractères.

.	N'importe quel caractère
[abc]	Soit le caractère a, soit le caractère b, soit le caractère c
[a-z]	N'importe quel caractère de a à z
[^a-z]	N'importe quel caractère qui n'est pas entre a et z
\s	Un caractère d'espacement (espace, tabulation, retour à la ligne, retour chariot, saut de ligne)
\S	Un caractère qui n'est pas un caractère d'espacement (équivalent à [^\s])
\d	Un caractère représentant un chiffre (équivalent à [0-9])
\D	Un caractère ne représentant pas un chiffre (équivalent à [^0-9])
\w	Un caractère composant un mot (équivalent à [a-zA-Z_0-9])
\W	Un caractère ne composant pas un mot (équivalent à [^\w])



```
String s = "hello";
System.out.println(s.matches("....."));           // true
System.out.println(s.matches("h[a-m]llo"));        // true
System.out.println(s.matches("\\w\\w\\w\\w\\w"));  // true
System.out.println(s.matches("h\\d\\s.o"));        // true
```

Une expression régulière peut contenir des quantificateurs qui permettent d'indiquer une séquence de caractères dans la chaîne.

X?	X est présent zéro ou une fois
X*	X est présent zéro ou n fois
X+	X est présent au moins une fois
X{n}	X est présent exactement n fois
X{n,}	X est présent au moins n fois
X{n,m}	X est présent entre n et m fois

```
String s = "hello";
System.out.println(s.matches(".*"));               // true
System.out.println(s.matches(".*+"));              // true
System.out.println(s.matches("x?hel+ow?"));        // true
System.out.println(s.matches(".*l{2}o"));           // true
System.out.println(s.matches("[eh]{0,2}l{1,100}o")); // true
```

Note

Il existe beaucoup d'autres motifs qui peuvent être utilisés dans une expression régulière. Reportez-vous à la [documentation Java](#).

Il est possible d'utiliser la méthode [String.replaceFirst](#) ou [String.replaceAll](#) pour remplacer respectivement la première ou toutes les occurrences d'une séquence de caractères définie par une expression régulière.

```
String s = "hello";
System.out.println(s.replaceAll("[aeiouy]", "^_")); // h^_a_l_l^_^
```

La méthode [String.split](#) permet de découper une chaîne de caractères en tableau de chaînes de caractère en utilisant une expression régulière pour identifier le séparateur.

```
String s = "hello the world";

// ["hello", "the", "world"]
String[] tab = s.split("\\w");

// ["hello", "world"]
tab = s.split(" the ");

// ["he", "", "", "the w", "r", "d"]
tab = s.split("[o]");
```

## Note

Les expressions régulières sont représentées en Java par la classe [Pattern](#). Il est possible de créer des instances de cette classe en compilant une expression régulière à l'aide de la méthode de classe [Pattern.compile](#).

# Les exceptions

---

La gestion des cas d'erreur représente un travail important dans la programmation. Les sources d'erreur peuvent être nombreuses dans un programme. Il peut s'agir :

- d'une défaillance physique ou logiciel de l'environnement d'exécution. Par exemple une erreur survient lors de l'accès à un fichier ou à la mémoire.
- d'un état atteint par un objet qui ne correspond pas à un cas prévu. Par exemple si une opération demande à positionner une valeur négative alors que cela n'est normalement pas permis par la spécification du logiciel.
- d'une erreur de programmation. Par exemple, un appel à une méthode est réalisé sur une variable dont la valeur est **null**.
- et bien d'autres cas encore...

La robustesse d'une application est souvent comprise comme sa capacité à continuer à rendre un service acceptable dans un environnement dégradé, c'est-à-dire quand toutes les conditions attendues normalement ne sont pas satisfaites.

En Java, la gestion des erreurs se confond avec la gestion des cas exceptionnels. On utilise alors le mécanisme des exceptions.

## Qu'est-ce qu'une exception ?

---

Une exception est une classe Java qui représente un état particulier et qui hérite directement ou indirectement de la classe [Exception](#). Par convention, le nom de la classe doit permettre de comprendre le type d'exception et doit se terminer par Exception.

Exemple de classes d'exception fournies par l'API standard :

- [NullPointerException](#)  
Signale qu'une référence **null** est utilisée pour invoquer une méthode ou accéder à un attribut.
- [NumberFormatException](#)  
Signale qu'il n'est pas possible de convertir une chaîne de caractères en nombre car la chaîne de caractère ne correspond pas à un nombre valide.
- [IndexOutOfBoundsException](#)  
Signale que l'on tente d'accéder à un indice de tableau en dehors des valeurs permises.

Pour créer sa propre exception, il suffit de créer une classe héritant de la classe [java.lang.Exception](#).

```
package dev.gayerie.heroes;

public class FinDuMondeException extends Exception {

    public FinDuMondeException() {
    }

    public FinDuMondeException(String message) {
        super(message);
    }
}
```

#### Note

La classe [Exception](#) fournit plusieurs constructeurs que l'on peut ou non appeler depuis la classe fille.

Une exception étant un objet, elle possède son propre état et peut ainsi stocker des informations utiles sur les raisons de son apparition.

```
package dev.gayerie.heroes;
import java.time.Instant;

public class FinDuMondeException extends Exception {

    private Instant date;

    public FinDuMondeException() {
        this(Instant.now());
    }

    public FinDuMondeException(Instant instant) {
        super("La fin du monde est survenue le " + instant);
        this.date = instant;
    }

    public Instant getDate() {
        return date;
    }
}
```

## Signaler une exception

Dans les langages de programmation qui ne supportent pas le mécanisme des exceptions, on utilise généralement un code retour ou une valeur booléenne pour savoir si une fonction ou une méthode s'est déroulée correctement. Cette mécanique se révèle assez fastidieuse dans son implémentation car cela signifie qu'un développeur doit tester dans son programme toutes les valeurs retournées par les fonctions ou les méthodes appelées

Les exceptions permettent d'isoler le code responsable du traitement de l'erreur. Cela permet d'améliorer la lisibilité du code source.

Lorsqu'un programme détecte un état exceptionnel, il peut le signaler en *jetant* une exception grâce au mot-clé **throw**.

```
if(isPlanDiaboliqueReussi()) {  
    throw new FinDuMondeException();  
}
```

Note

La classe [Exception](#) hérite de la classe [Throwable](#). Le mot-clé **throw** peut en fait être utilisé avec n'importe quelle instance qui hérite directement ou indirectement de [Throwable](#).

Jeter une exception signifie que le flot d'exécution normal de la méthode est interrompu jusqu'au point de traitement de l'exception. Si aucun point de traitement n'est trouvé, le programme s'interrompt.

## Traiter une exception

Pour traiter une exception, il faut d'abord délimiter un bloc de code avec le mot-clé **try**. Ce bloc de code correspond au flot d'exécution pour lequel on souhaite éventuellement *attraper* une exception qui serait jetée afin d'implémenter un traitement particulier. Le bloc **try** peut être suivi d'un ou plusieurs blocs **catch** pour intercepter une exception d'un type particulier.

1 2 3	try { if (heros == null) { throw new NullPointerException("Le heros
4 5 6	ne peut pas être nul !"); } boolean victoire =
7 8 9	heros.combattre(espritDuMal); boolean planDejoue =
10 11	heros.desamorcer(machineInfernale); if (!victoire    !planDejoue) {
12 13	throw new FinDuMondeException(); } heros.setPoseVictorieuse(); }
14 15	catch (FinDuMondeException fdme) { // ... }
16 17	

Dans l'exemple ci-dessus, si la variable *heros* vaut **null** alors le traitement du bloc **try** est interrompu à la ligne 3 par une [NullPointerException](#). Sinon le bloc continue à s'exécuter. La ligne 13 ne sera exécutée que si la condition à la ligne 9 est fausse. Par contre, si cette condition est vraie, le traitement du bloc est interrompu par le lancement d'une *FinDuMondeException* et le traitement reprend dans le bloc **catch** à partir de la ligne 16.

La bloc **catch** permet à la fois d'identifier le type d'exception concerné par le bloc de traitement et à la fois de déclarer une variable qui permet d'avoir accès à l'exception durant l'exécution du bloc **catch**. Un bloc **catch** sera exécuté si une exception du même type ou d'un sous-type que celui déclaré par le bloc est lancée à l'exécution. Attention, si une exception déclenche le traitement d'un bloc **catch**, le flot d'exécution reprend ensuite à la fin des blocs **catch**.

1 2 3	try { if (heros == null) { throw new NullPointerException("Le heros
4 5 6	ne peut pas être nul !"); } boolean victoire =
7 8 9	heros.combattre(espritDuMal); boolean planDejoue =
10 11	heros.desamorcer(machineInfernale); if (!victoire    !planDejoue) {
12 13	throw new FinDuMondeException(); } heros.setPoseVictorieuse(); }
14 15	catch (Exception e) { // ... }
16 17	

Dans le code ci-dessus, le bloc **catch** est associé aux exceptions de type [Exception](#). Comme toutes les exceptions en Java hérite directement ou indirectement de cette classe, ce bloc sera exécuté pour traité la [NullPointerException](#) à la ligne 3 ou la *FinDuMondeException* à la ligne 10.

Les blocs **catch** sont pris en compte à l'exécution dans l'ordre de leur déclaration. Déclarer un bloc **catch** pour une exception parente avant un bloc **catch** pour une exception enfant est considéré comme une erreur de compilation.

1 2 3	try { if (heros == null) { throw new NullPointerException("Le heros
4 5 6	ne peut pas être nul !"); } boolean victoire =
7 8 9	heros.combattre(espritDuMal); boolean planDejoue =
10 11	heros.desamorcer(machineInfernale); if (!victoire    !planDejoue) {
12 13	throw new FinDuMondeException(); } heros.setPoseVictorieuse(); }
14 15	catch (Exception e) { // ... } catch (FinDuMondeException fdme) { //
16 17	ERREUR DE COMPILATION }
18 19	

Dans, l'exemple précédent, il faut bien comprendre que [Exception](#) est la classe parente de *FinDuMondeException*. Donc si une exception de type *FinDuMondeException* est lancée, alors seul le premier bloc **catch** sera exécuté. Le second est donc simplement du code mort et générera une erreur de compilation. Pour que cela fonctionne, il faut inverser l'ordre des blocs **catch** :

1 2 3	try { if (heros == null) { throw new NullPointerException("Le heros
4 5 6	ne peut pas être nul !"); } boolean victoire =
7 8 9	heros.combattre(espritDuMal); boolean planDejoue =
10 11	heros.desamorcer(machineInfernale); if (!victoire    !planDejoue) {
12 13	throw new FinDuMondeException(); } heros.setPoseVictorieuse(); }
14 15	catch (FinDuMondeException fdme) { // ... } catch (Exception e) {
16 17	// ... }
18 19	

Maintenant, un premier bloc **catch** fournit un traitement particulier pour les exceptions de type *FinDuMondeException* ou de type enfant et un second bloc **catch** fournit un traitement pour les autres exceptions.

Parfois, le code du bloc **catch** est identique pour différents types d'exception. Si ces exceptions ont une classe parente commune, il est possible de déclarer un bloc **catch** simplement pour cette classe parente afin d'éviter la duplication de code. Dans notre exemple, la classe ancêtre commune entre [NullPointerException](#) et *FinDuMondeException* est la classe [Exception](#). Donc si nous déclarons un bloc **catch** pour le type [Exception](#), nous fournissons un bloc de traitement pour tous les types d'exception, ce qui n'est pas vraiment le but recherché. Dans cette situation, il est possible de préciser plusieurs types d'exception dans le bloc **catch** en les séparant par | :

1 2 3	<code>try { if (heros == null) { throw new NullPointerException("Le heros</code>
4 5 6	<code>ne peut pas être nul !"); } boolean victoire =</code>
7 8 9	<code>heros.combattre(espritDuMal); boolean planDejoue =</code>
10 11	<code>heros.desamorcer(machineInfernale); if (!victoire    !planDejoue) {</code>
12 13	<code>throw new FinDuMondeException(); } heros.setPoseVictorieuse(); }</code>
14 15	<code>catch (NullPointerException   FinDuMondeException ex) { //</code>
16 17	<code>traitement commun aux deux types d'exception... }</code>

Note

L'exécution d'un bloc **catch** peut très bien être interrompue par une exception. L'exécution d'un bloc **catch** peut même conduire à relancer l'exception qui vient d'être interceptée.

## Propagation d'une exception

Si une exception n'est pas interceptée par un bloc **catch**, alors elle remonte la pile d'appel, jusqu'à ce qu'un bloc **catch** prenne cette exception en charge. Si l'exception remonte tout en haut de la pile d'appel du thread, alors le thread s'interrompt. S'il s'agit du thread principal, alors l'application s'arrête en erreur.

Le mécanisme de propagation permet de séparer la partie de l'application qui génère l'exception de la partie qui traite cette exception.

Si nous reprenons notre exemple précédent, nous pouvons grandement l'améliorer. En effet, les méthodes *combattre* et *desamorcer* devraient s'interrompre par une exception plutôt que de retourner un booléen. L'exception jetée porte une information plus riche qu'un simple booléen car elle dispose d'un type et d'un état interne.

```
try {
    if (heros == null) {
        throw new NullPointerException("Le heros ne peut pas être nul !");
    }

    heros.combattre(espritDuMal);
    heros.desamorcer(machineInfernale);
    heros.setPoseVictorieuse();

} catch (FinDuMondeException ex) {
    // ...
}
```

Le code devient beaucoup plus lisible. On comprend que le bloc **try** peut être interrompu par une exception de type *FinDuMondeException* et le code du bloc n'est plus contaminé par des variables et des instructions **if** spécifiquement utilisées pour la gestion des erreurs.

La langage Java impose que les méthodes signalent les types d'exception qu'elles peuvent jeter. Ainsi, le code ci-dessus ne compilera que si au moins une des instructions du bloc **try** peut générer une *FinDuMondeException*. Cela permet au compilateur de détecter d'éventuel code mort. La déclaration des exceptions jetées par une méthode fait donc partie de sa signature et utilise le mot-clé **throws**.

```
package dev.gayerie.heroes;

public class Heros {

    public void combattre(Vilain vilain) throws FinDuMondeException {
        // ...
    }

    public void desamorcer(Piege piege) throws FinDuMondeException {
        // ...
    }

    public void setPoseVictorieuse() {
        // ...
    }
}
```

Grâce aux exceptions, il est maintenant possible d'interrompre une méthode. Il est même possible d'interrompre un constructeur. Cela aura pour effet de stopper la construction de l'objet et ainsi d'empêcher d'avoir une instance dans un état invalide.

```
package dev.gayerie.heroes;

public class Heros {

    public Heros(String classePerso) throws ClasseDePersoInvalideException {
        if (classePerso == null || "".equals(classePerso)) {
            throw new ClasseDePersoInvalideException();
        }
    }
}
```

La déclaration des exceptions dans la signature d'une méthode permet à la fois de documenter dans le code lui-même le comportement de la méthode tout en contrôlant à la compilation que les cas d'exception sont gérés par le code.

```
public Marchandise acheter(long montant, Currency devise)
    throws CreditInsuffisantException, DeviseRefuseeException,
        MarchandiseNonDisponibleException {
    // ...
}
```

Dans l'exemple ci-dessus, même sans avoir accès au code source, la signature suffit à renseigner sur les cas d'erreur que l'on va pouvoir rencontrer lorsqu'on appelle la méthode *acheter*.

## Exceptions et polymorphisme

Comme la déclaration des exceptions jetées par une méthode fait partie de sa signature, certaines règles doivent être respectées pour la redéfinition de méthode afin que le polymorphisme fonctionne correctement.

Selon le [principe de substitution de Liskov](#), dans la redéfinition d'une méthode, les préconditions ne peuvent pas être renforcées par la sous-classe et les postconditions ne peuvent pas être affaiblies par la sous-classe. Rapporté au mécanisme des exceptions, cela signifie qu'une méthode redéfinie ne peut pas lancer des exceptions supplémentaires. Par contre, elle peut lancer des exceptions plus spécifiques. Le langage Java ne permet pas de distinguer les exceptions qui signalent une violation des préconditions ou des postconditions. C'est donc aux développeurs de s'assurer que les postconditions ne sont pas affaiblies dans la sous-classe.

Ainsi, si la classe *SuperHeros* hérite de la classe *Heros*, elle peut redéfinir les méthodes en ne déclarant pas d'exception.

```
package dev.gayerie.heroes;

public class SuperHeros extends Heros {

    @Override
    public void combattre(Vilain vilain) {
        // ...
    }

    @Override
    public void desamorcer(Piege piege) {
        // ...
    }
}
```

Cette nouvelle classe peut aussi changer les types d'exception déclarés par les méthodes redéfinies à condition que ces types soient des classes filles des exceptions d'origine.

```
package dev.gayerie.heroes;

public class SuperHeros extends Heros {

    @Override
    public void desamorcer(Piege piege) throws PlanMachiaveliqueException {
        // ...
    }

}
```

Le code précédent ne compile que si l'exception *PlanMachiaveliqueException* hérite directement ou indirectement de *FinDuMondeException*.

```
package dev.gayerie.heroes;

public class PlanMachiaveliqueException extends FinDuMondeException {
    // ...
}
```

Note



Même si cela est maladroit, il est possible de conserver la déclaration des exceptions dans la signature même si la méthode ne jette pas ces types d'exception. Le compilateur ne vérifie pas si une méthode jette effectivement tous les types d'exception déclarés par sa signature.

## Le bloc finally

À la suite des blocs **catch** il est possible de déclarer un bloc **finally**. Un bloc **finally** est exécuté systématiquement, que le bloc **try** se soit terminé normalement ou par une exception.

Note

Si un bloc **try** se termine par une exception et qu'il n'existe pas de bloc **catch** approprié, alors le bloc **finally** est exécuté et ensuite l'exception est propagée.

```
try {
    if (heros == null) {
        throw new NullPointerException("Le heros ne peut pas être nul !");
    }

    heros.combattre(espritDuMal);
    heros.desamorcer(machineInfernale);
    heros.setPoseVictorieuse();
} catch (FinDuMondeException fdme) {
    // ...
} finally {
    // Ce bloc sera systématiquement exécuté
    jouerGeneriqueDeFin();
}
```

Note

Un bloc **finally** est exécuté même si bloc **try** exécute une instruction **return**. Dans ce cas, le bloc **finally** est d'abord exécuté puis ensuite l'instruction **return**.

Le bloc **finally** est le plus souvent utilisé pour gérer les ressources autre que la mémoire. Si le programme ouvre une connexion, un fichier..., le traitement est effectué dans le bloc **try** puis le bloc **finally** se charge de libérer la ressource.

```
java.io.FileReader reader = new java.io.FileReader(filename);
try {
    int nbCharRead = 0;
    char[] buffer = new char[1024];
    StringBuilder builder = new StringBuilder();
    // L'appel à reader.read peut lancer une java.io.IOException
    while ((nbCharRead = reader.read(buffer)) >= 0) {
        builder.append(buffer, 0, nbCharRead);
    }
    // le retour explicite n'empêche pas l'exécution du block finally.
    return builder.toString();
} finally {
    // Ce block est obligatoirement exécuté après le block try.
    // Ainsi le flux de lecture sur le fichier est fermé
    // avant le retour de la méthode.
    reader.close();
}
```

```
}
```

## Le try-with-resources

La gestion des ressources peut également être réalisée par la syntaxe du [try-with-resources](#).

```
try (java.io.FileReader reader = new java.io.FileReader(filename)) {
    int nbCharRead = 0;
    char[] buffer = new char[1024];
    StringBuilder builder = new StringBuilder();
    while ((nbCharRead = reader.read(buffer)) >= 0) {
        builder.append(buffer, 0, nbCharRead);
    }
    return builder.toString();
}
```

Après le mot-clé **try**, on déclare entre parenthèses une ou plusieurs initialisations de variable. Ces variables doivent être d'un type qui implémente l'interface [AutoCloseable](#) ou [Closeable](#). Ces interfaces ne déclarent qu'une seule méthode : **close**. Le compilateur ajoute automatiquement un bloc **finally** à la suite du bloc **try** pour appeler la méthode **close** sur chacune des variables qui ne valent pas **null**.

Ainsi pour ce code :

```
try (java.io.FileReader reader = new java.io.FileReader(filename)) {
    // ...
}
```

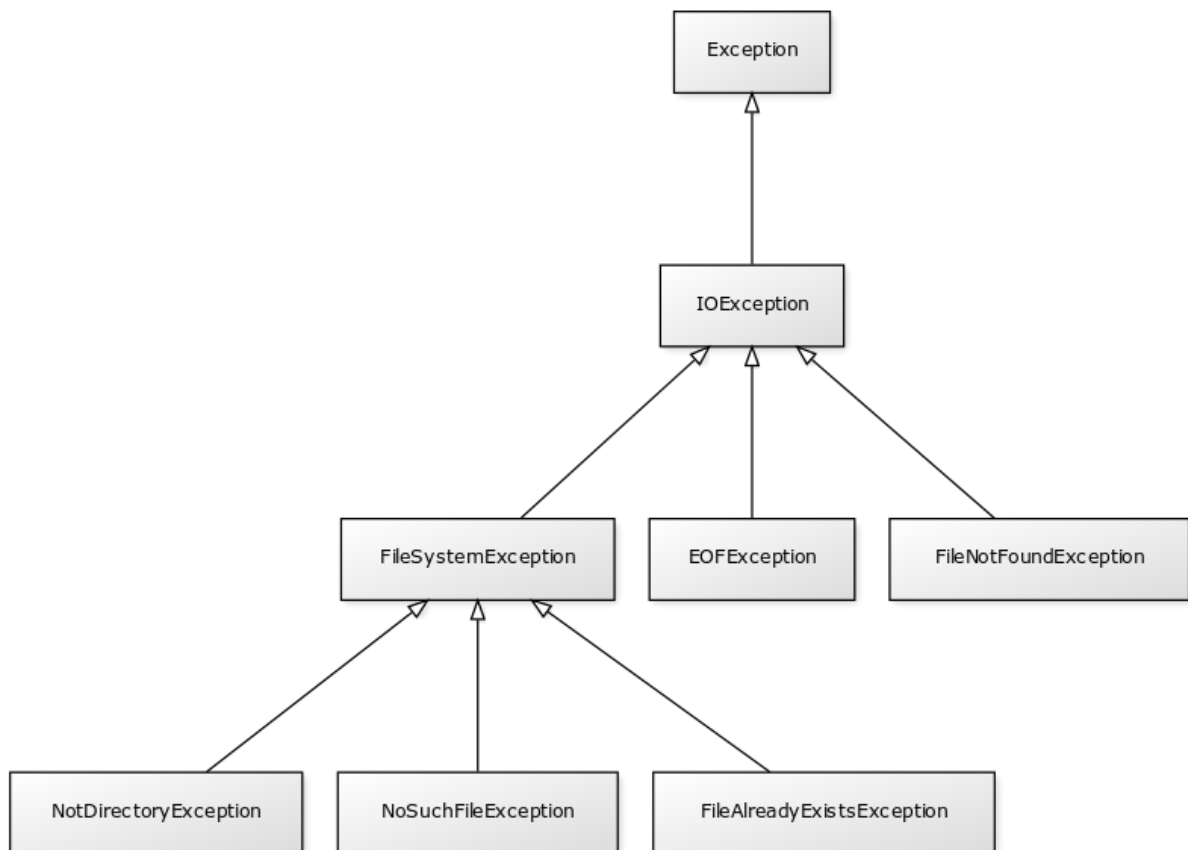
Le compilateur générera le bytecode correspondant à :

```
{
    java.io.FileReader reader = new java.io.FileReader(filename)
    try {
        // ...
    } finally {
        if (reader != null) {
            reader.close();
        }
    }
}
```

La syntaxe [try-with-resources](#) est à la fois simple à lire et évite d'oublier de libérer des ressources puisque le compilateur se charge d'introduire le code pour nous.

## Hiérarchie applicative d'exception

Comme les exceptions sont des objets, il est possible de créer une hiérarchie d'exception par héritage. C'est par exemple le cas pour les exceptions d'entrée/sortie en Java.



Un extrait de la hiérarchie de [java.io.IOException](https://docs.oracle.com/javase/7/docs/api/java/io/IOException.html)

La hiérarchie d'exception permet de grouper des erreurs en concevant des types d'exception de plus en plus généraux. Une application pourra donc traiter à sa convenance des exceptions générales comme [IOException](https://docs.oracle.com/javase/7/docs/api/java/io/IOException.html) mais pourra, au besoin, fournir un bloc **catch** pour traiter des exceptions plus spécifiques.

```
try {  
    // ... opérations sur des fichiers  
} catch (NoSuchFileException nsfe) {  
    // ...  
} catch (IOException ioe) {  
    // ...  
}
```

## Exception cause

Il est souvent utile d'encapsuler une exception dans une autre exception. Par exemple, imaginons une méthode qui souhaite réaliser une opération distante sur un serveur. Si le serveur distant n'est pas joignable, le programme devra intercepter une [IOException](https://docs.oracle.com/javase/7/docs/api/java/io/IOException.html). Mais cela n'a peut-être pas beaucoup de sens pour le reste du programme, la méthode peut décider de jeter à la place une exception définie par l'application comme une *OperationNonDisponibleException*.

```
package dev.gayerie;

public class OperationNonDisponibleException extends Exception {

    public OperationNonDisponibleException(Exception cause) {
        super(cause);
    }
}
```

Cette exception n'a pas de lien d'héritage avec une [IOException](#). Par contre, elle expose un constructeur qui accepte en paramètre une exception. Cela permet d'indiquer que l'exception a été causée par une autre exception.

```
try {

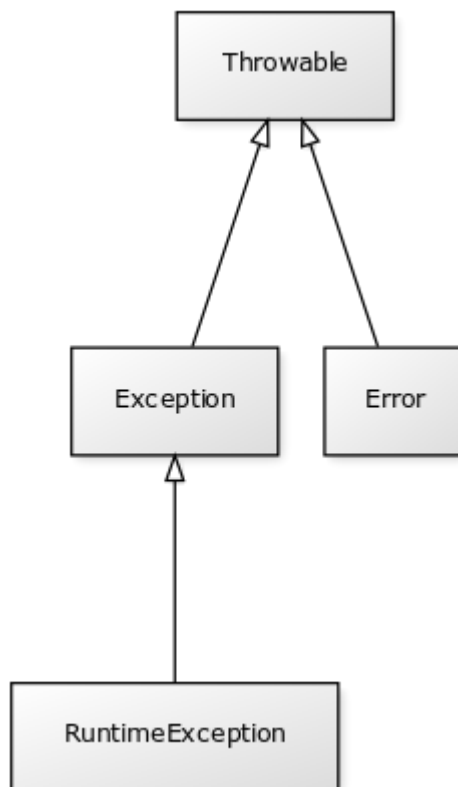
    // ... opérations d'entrée / sortie vers le serveur

} catch (IOException ioe) {
    throw new OperationNonDisponibleException(ioe);
}
```

La classe [Exception](#) fournit la méthode [getCause](#) (qu'elle hérite de [Throwable](#)) pour connaître l'exception qui est la cause du problème.

## Les erreurs et les exceptions runtime

En regardant plus en détail la hiérarchie à la base des exceptions, on découvre le modèle d'héritage suivant :



La classe [Throwable](#) est la classe indiquant qu'il est possible d'utiliser ce type avec le mot clé **throw**. De plus la classe [Throwable](#) fournit des méthodes utilitaires. Par exemple, la méthode [printStackTrace](#) permet d'afficher sur la sortie d'erreur standard la pile d'appel de l'application.

```
try {
    double d = 1/0; // produit une ArithmeticException
} catch (ArithmeticException e) {
    // Afficher la pile d'appel sur la sortie d'erreur standard
    e.printStackTrace();
}
```

La classe [Error](#) hérite de [Throwable](#) comme [Exception](#). [Error](#) est la classe de base pour représenter les erreurs sérieuses que l'application ne devrait pas intercepter. Lorsqu'une erreur survient cela signifie souvent que l'environnement d'exécution est dans un état instable. Par exemple, la classe [OutOfMemoryError](#) hérite indirectement de cette classe. Cette erreur signale que la JVM ne dispose plus d'assez de mémoire (généralement pour allouer de l'espace pour les nouveaux objets).

La classe [RuntimeException](#) représente des problèmes d'exécution qui proviennent la plupart du temps de bug dans l'application. Parmi les classes filles de cette classe, on trouve :

- [ArithmeticException](#)  
signale une opération arithmétique invalide comme une division par zéro.
- [NullPointerException](#)  
signale que l'on tente d'accéder à une méthode ou un attribut à travers une référence **null**.
- [ClassCastException](#)  
signale qu'un transtypage invalide a été réalisé.

Généralement, les exceptions qui héritent de [RuntimeException](#) ne sont pas interceptées ni traitées par l'application. Au mieux, elles sont interceptées au plus haut de la pile d'appel pour signaler une erreur à l'utilisateur ou dans les fichiers de log.

Les classes [Error](#), [RuntimeException](#) et toutes les classes qui en héritent sont appelées des *unchecked exceptions*. Cela signifie que le compilateur n'exige pas que ces exceptions apparaissent dans la signature des méthodes. En effet, elles représentent des problèmes internes graves de la JVM ou des bugs. Donc virtuellement toutes les méthodes en Java sont susceptibles de lancer de telles exceptions.

Si nous reprenons notre exemple des véhicules, les méthodes pour accélérer et décélérer devraient contrôler que le paramètre passé est bien un nombre positif. Si ce n'est pas le cas, elle peut jeter une [IllegalArgumentException](#) qui est une exception runtime fournie par l'API standard et qui sert à signaler qu'un paramètre est invalide. Cette exception ne doit pas être obligatoirement déclarée dans la signature de la méthode.

```
package dev.gayerie.conduite;

public class Vehicule {

    private final String marque;
    protected float vitesse;

    public Vehicule(String marque) {
        this.marque = marque;
    }

    public void accélérer(float deltaVitesse) {
```

```

    if (deltaVitesse < 0) {
        throw new IllegalArgumentException("deltaVitesse doit être positif");
    }
    this.vitesse += deltaVitesse;
}

public void decelerer(float deltaVitesse) {
    if (deltaVitesse < 0) {
        throw new IllegalArgumentException("deltaVitesse doit être positif");
    }
    this.vitesse = Math.max(this.vitesse - deltaVitesse, 0f);
}

// ...
}

```

#### Note

Il est tout de même intéressant de signaler les exceptions runtime qui sont engendrées par des violations de préconditions ou de postconditions. Cela permet de documenter explicitement ces préconditions et ces postconditions.

```

/**
 * Accélère le véhicule
 *
 * @param deltaVitesse la vitesse à ajouter à la vitesse courante.
 * @throws IllegalArgumentException si deltaVitesse est un nombre négatif.
 */
public void accélérer(float deltaVitesse) throws IllegalArgumentException {
    if (deltaVitesse < 0) {
        throw new IllegalArgumentException("deltaVitesse doit être positif");
    }
    this.vitesse += deltaVitesse;
}

```

Par opposition, toutes les autres exceptions sont appelées des *checked exception*. Une méthode qui est susceptible de laisser se propager une *checked exception* doit le signaler dans sa signature à l'aide du mot-clé **throws**.

## Choix entre checked et unchecked

En tant que développeurs, lorsque nous créons de nouvelles classes pour représenter des exceptions, nous avons le choix entre hériter de la classe [Exception](#) ou de la classe [RuntimeException](#). C'est-à-dire entre créer une *checked* ou une *unchecked* exception. La frontière entre les deux familles a évolué au cours des versions de Java.

#### Note

Il ne faut jamais créer une classe qui hérite de [Error](#). Les classes qui en héritent sont faites pour signaler un problème dans la JVM.

On considère généralement qu'il est préférable de créer une *unchecked exception* lorsque l'exception représente une erreur technique, un événement qui ne relève pas du domaine de l'application mais qui est plutôt lié à son contexte d'exécution. Généralement il s'agit d'exceptions que l'application ne pourra pas traiter correctement à part signaler un problème aux utilisateurs ou aux administrateurs. Par exemple, si votre application se connecte à un service distant, vous pouvez avoir besoin de créer une exception *RemoteServiceUnavailableException* pour signaler que le service ne répond pas. Ce type d'exception est probablement une *unchecked exception* et devrait hériter de [RuntimeException](#).

Par contre, les exceptions qui peuvent avoir une valeur pour le domaine applicatif devraient être des *checked exception*. Généralement, elles traduisent des états particuliers identifiés par les analystes du domaine.

Par exemple, si vous développez une application bancaire pour réaliser des transactions, certaines transactions peuvent échouer lorsqu'un compte bancaire n'est pas suffisamment approvisionné. Pour représenter cet état, on peut créer une classe *SoldInsuffisantException*. Il est probable que cette exception devrait être une *checked exception* afin que le compilateur puisse vérifier qu'elle est correctement traitée.

## Les énumérations

Dans une application, il est très utile de pouvoir représenter des listes finies d'éléments. Par exemple, si une application a besoin d'une liste de niveaux de criticité, elle peut créer des constantes dans une classe utilitaire quelconque.

```
package dev.gayerie;

public class ClasseUtilitaireQuelconque {

    public static final int CRITICITE_FAIBLE = 0;
    public static final int CRITICITE_NORMAL = 1;
    public static final int CRITICITE_HAUTE = 2;
}
```

Ce choix d'implémentation est très imparfait. Même si le choix du type **int** permet de retranscrire une notion d'ordre dans les niveaux de criticité, il ne permet pas de créer un vrai type représentant une criticité.

Il existe un type particulier en Java qui permet de fournir une meilleure implémentation dans ce cas : l'énumération. Un énumération se déclare avec le mot-clé **enum**.

```
package dev.gayerie;

public enum Criticite {
    FAIBLE,
    NORMAL,
    HAUTE
}
```

### Note

Les éléments d'une énumération sont des constantes. Donc, par convention, ils sont écrits en majuscules et les mots sont séparés par \_.

Comme une classe, une énumération a une portée et elle est définie dans un fichier qui porte son nom. Pour l'exemple ci-dessus, le code source sera dans le fichier *Criticite.java*.

Une énumération peut également être définie dans une classe, une interface et même dans une énumération.

```
package dev.gayerie;

public class Zone {

    public enum NiveauDeSecurite {FAIBLE, MOYEN, FORT}

    private NiveauDeSecurite niveau;

    // ...

}
```

L'énumération permet de définir un type avec un nombre fini de valeurs possibles. Il est possible de manipuler ce nouveau type comme une constante, comme un attribut, un paramètre et une variable.

```
package dev.gayerie;

public class RapportDebug {

    private Criticite criticite = Criticite.NORMAL;

    public RapportDebug() {
    }

    public RapportDebug(Criticite criticite) {
        this.criticite = criticite;
    }
}

RapportDebug rapport = new RapportDebug(Criticite.HAUTE);
```

Note

Une variable, un attribut ou un paramètre du type d'une énumération peut avoir la valeur **null**.

## Les méthodes d'une énumération

Nous verrons bientôt qu'une énumération est en fait une classe particulière. Donc une énumération fournit également des méthodes de classe et des méthodes pour chacun des éléments de l'énumération.

- **valueOf**

Une méthode de classe qui permet de convertir une chaîne de caractères en une énumération. Attention toutefois, si la chaîne de caractères ne correspond pas à un nom d'un élément de l'énumération, cette méthode produit une [IllegalArgumentException](#).  
`Criticite criticite = Criticite.valueOf("HAUTE");`

- **values**



Une méthode de classe qui retourne un tableau contenant tous les éléments de l'énumération dans l'ordre de leur déclaration. Cette méthode est très pratique pour connaître la liste des valeurs possibles par programmation. `for(Criticite c : Criticite.values()) { System.out.println(c); }`

Chaque élément d'une énumération possède les méthodes suivantes :

- **name**

Retourne le nom de l'élément sous la forme d'une chaîne de caractères. `String name = Criticite.NORMAL.name(); // "NORMAL"`

- **ordinal**

Retourne le numéro d'ordre d'un élément. Le numéro d'ordre est donné par l'ordre de la déclaration, le premier élément ayant le numéro 0. `int ordre = Criticite.HAUTE.ordinal(); // 2`

Note

Une énumération implémente également l'interface [Comparable](#). Donc, une énumération implémente la méthode [compareTo](#) qui réalise une comparaison en se basant sur le numéro d'ordre.

## Égalité entre énumérations

Par définition, chaque élément d'une énumération n'existe qu'une fois en mémoire. Une énumération garantit que l'unicité de la valeur est équivalente à l'unicité en mémoire. Cela signifie que l'on peut utiliser l'opérateur `==` pour comparer des variables, des attributs et des paramètres du type d'énumération. L'utilisation de l'opérateur `==` est même considérée comme la bonne façon de comparer les énumérations.

```
if (criticite == Criticite.HAUTE) {  
    // ...  
}
```

## Support de switch

Une énumération peut être utilisée dans une structure **switch** :

```
switch (criticite) {  
  
    case Criticite.FAIBLE:  
        // ...  
        break;  
  
    case Criticite.NORMAL:  
        // ...  
        break;  
  
    case Criticite.HAUTE:  
        // ...  
        break;  
  
}
```

# Génération d'une énumération

Le mot-clé **enum** est en fait un sucre syntaxique. Les énumérations en Java sont des classes comme les autres. Ainsi, l'énumération :

```
package dev.gayerie;

public enum Criticite {

    FAIBLE,
    NORMAL,
    HAUTE

}
```

est transcrite comme ceci par le compilateur :

```
package dev.gayerie;

public final class Criticite extends Enum<Criticite> {

    public static final Criticite FAIBLE = new Criticite("FAIBLE", 0);
    public static final Criticite NORMAL = new Criticite("NORMAL", 1);
    public static final Criticite HAUTE = new Criticite("HAUTE", 2);

    public static Criticite valueOf(String value) {
        return Enum.valueOf(Criticite.class, value);
    }

    public static Criticite[] values() {
        return new Criticite[] {FAIBLE, NORMAL, HAUTE};
    }

    private Criticite(String name, int ordinal) {
        super(name, ordinal);
    }
}
```

Note

Malheureusement le code ci-dessus ne compile pas car le compilateur Java n'autorise pas à créer soi-même une énumération.

Le code ci-dessus nous permet de remarquer que :

- Les valeurs d'une énumération sont en fait des attributs de classe du type de l'énumération elle-même.
- Une énumération est déclarée **final** donc il n'est pas possible d'hériter d'une énumération (sauf en créant une classe interne anonyme).
- Le constructeur d'une énumération est privé, empêchant ainsi de créer de nouvelle instance.
- Une énumération hérite de la classe [Enum](#).

## Ajout de méthodes et d'attributs

Lorsque l'on a bien compris qu'une énumération est une classe particulière, il devient évident qu'il est possible d'ajouter des attributs et des méthodes à une énumération.

```
package dev.gayerie;

public enum Couleur {

    ROUGE, ORANGE, JAUNE, VERT, BLEU, MAGENTA;

    private static final List<Couleur> COULEURS_CHAUDES = Arrays.asList(ROUGE,
    ORANGE, JAUNE);

    public boolean isChaud() {
        return COULEURS_CHAUDES.contains(this);
    }

    public boolean isFroide() {
        return !isChaud();
    }

    public Couleur getComplementaire() {
        Couleur[] values = Couleur.values();
        int index = this.ordinal() + (values.length / 2);
        return values[index % values.length];
    }
}
```

Note

Notez l'utilisation du point-virgule à la fin de la liste des couleurs. Ce point-virgule n'est obligatoire que lorsque l'on veut ajouter une déclaration dans l'énumération afin de séparer la liste des éléments du reste.

Les énumérations peuvent devenir des objets complexes qui fournissent de nombreux services.

```
Couleur couleur = Couleur.ROUGE;

System.out.println(couleur.isChaud()); // true
System.out.println(couleur.isFroide()); // false
System.out.println(couleur.getComplementaire()); // VERT
```

## Ajout de constructeurs

Il est également possible d'ajouter un ou plusieurs constructeurs dans une énumération.

Attention, ces constructeurs doivent impérativement être de portée **private** sous peine de faire échouer la compilation. L'appel au constructeur se fait au moment de la déclaration des éléments de l'énumération.

```
package dev.gayerie;

public enum Polygone {
```

```

    TRIANGLE(3), QUADRILATERE(4), PENTAGONE(5);

    private final int nbCotes;

    private Polygone(int nbCotes) {
        this.nbCotes = nbCotes;
    }

    public int getNbCotes() {
        return nbCotes;
    }
}

```

## Les interfaces

Une interface permet de définir un ensemble de services qu'un client peut obtenir d'un objet. Une interface introduit une abstraction pure qui permet un découplage maximal entre un service et son implémentation. On retrouve ainsi les interfaces au cœur de l'implémentation de beaucoup de bibliothèques et de frameworks. Le mécanisme des interfaces permet d'introduire également une forme simplifiée d'héritage multiple.

### Déclaration d'une interface

Une interface se déclare avec le mot-clé **interface**.

```

package dev.gayerie.compte;

public interface Compte {

}

```

Comme pour une classe, une interface a une portée, un nom et un bloc de déclaration. Une interface est déclarée dans son propre fichier qui porte le même nom que l'interface. Pour l'exemple ci-dessus, le fichier doit s'appeler *Compte.java*.

Une interface décrit un ensemble de méthodes en fournissant uniquement leur signature.

```

package dev.gayerie.compte;

public interface Compte {

    void deposter(int montant) throws OperationInterrompueException,
                                   CompteBlokueException;

    int retirer(int montant) throws OperationInterrompueException,
                                   CompteBlokueException;

    int getBalance() throws OperationInterrompueException;

}

```

Une interface introduit un nouveau type d'abstraction qui définit à travers ces méthodes un ensemble d'interactions autorisées. Une classe peut ensuite implémenter une ou plusieurs interfaces.

Note

Les méthodes d'une interface sont par défaut **public** et **abstract**. Il n'est pas possible de déclarer une autre portée que **public**.

```
package dev.gayerie;

public interface Mobile {

    public abstract void deplacer();

}
```

L'interface ci-dessus est strictement identique à celle-ci :

```
package dev.gayerie;

public interface Mobile {

    void deplacer();

}
```

## Implémentation d'une interface

Une classe signale les interfaces qu'elle implémente grâce au mot-clé **implements**. Une classe concrète doit fournir une implémentation pour toutes les méthodes d'une interface, soit dans sa déclaration, soit parce qu'elle en hérite.

```
package dev.gayerie.compte;

public class CompteBancaire implements Compte {

    private final String numero;
    private int balance;

    public CompteBancaire(String numero) {
        this.numero = numero;
    }

    @Override
    public void deposer(int montant) {
        this.balance += montant;
    }

    @Override
    public int retirer(int montant) throws OperationInterrompueException {
        if (balance < montant) {
            throw new OperationInterrompueException();
        }
    }
}
```

```

        return this.balance -= montant;
    }

    @Override
    public int getBalance() {
        return this.balance;
    }

    public String getNumero() {
        return numero;
    }
}

```

L'implémentation des méthodes d'une interface suit les mêmes règles que la redéfinition.

#### Note

Si la classe qui implémente l'interface est une classe abstraite, alors elle n'est pas obligée de fournir une implémentation pour les méthodes de l'interface.

Même si les mécanismes des interfaces sont proches de ceux des classes abstraites, ces deux notions sont clairement distinctes. Une classe abstraite permet de mutualiser une implémentation dans une hiérarchie d'héritage en introduisant un type plus abstrait. Une interface permet de définir les interactions possibles entre un objet et ses clients. Une interface agit comme un contrat que les deux parties doivent remplir. Comme l'interface n'impose pas de s'insérer dans une hiérarchie d'héritage, il est relativement simple d'adapter une classe pour qu'elle implémente une interface.

Une interface introduit un nouveau type de relation qui serait du type *est comme un (is-like-a)*.

Par exemple, il est possible de créer un système de gestion de comptes utilisant l'interface *Compte*. Il est facile ensuite de fournir une implémentation de cette interface pour un compte bancaire, un porte-monnaie électronique, un compte en ligne... t Une classe peut implémenter plusieurs interfaces si nécessaire. Pour cela, il suffit de donner les noms des interfaces séparés par une virgule.

```

package dev.gayerie.animal;

public interface Carnivore {

    void manger(Animal animal);

}

package dev.gayerie.animal;

public interface Herbivore {

    void manger(Vegetal vegetal);

}

package dev.gayerie.animal;

public class Humain extends Animal implements Carnivore, Herbivore {

```

```

@Override
public void manger(Animal animal) {
    // ...
}

@Override
public void manger(Vegetal vegetal) {
    // ...
}
}

```

Dans l'exemple précédent, la classe *Humain* implémente les interfaces *Carnivore* et *Herbivore*. Donc une instance de la classe *Humain* peut être utilisée dans une application partout où les types *Carnivore* et *Herbivore* sont attendus.

```

Humain humain = new Humain();

Carnivore carnivore = humain;
carnivore.manger(new Poulet()); // Poulet hérite de Animal

Herbivore herbivore = humain;
herbivore.manger(new Chou());   // Chou hérite de Vegetal

```

## Attributs et méthodes statiques

Une interface peut déclarer des attributs. Cependant tous les attributs d'une interface sont par défaut **public**, **static** et **final**. Il n'est pas possible de modifier la portée de ces attributs. Autrement dit, une interface ne peut déclarer que des constantes.

```

package dev.gayerie.compte;

public interface Compte {

    int PLAFOND_DEPOT = 1_000_000;

    void deposer(int montant) throws OperationInterrompueException,
    CompteBloqueException;

    int retirer(int montant) throws OperationInterrompueException,
    CompteBloqueException;

    int getBalance() throws OperationInterrompueException;

}

```

Note

On peut préciser **public**, **static** et **final** dans la déclaration d'un attribut d'interface :

```

public static final int PLAFOND_DEPOT = 1_000_000;

```

Ceci est strictement équivalent à

```
int PLAFOND_DEPOT = 1_000_000;
```

Une interface peut également déclarer des méthodes **static**. Dans ce cas, il s'agit de méthodes équivalentes aux méthodes de classe et l'interface doit fournir une implémentation pour ces méthodes. Ces méthodes doivent explicitement avoir le mot-clé **static** et elles ont une portée publique par défaut.

```
package dev.gayerie.compte;

public interface Compte {

    int PLAFOND_DEPOT = 1_000_000;

    static int getBalanceTotale(Compte... comptes) throws
    OperationInterrompueException {
        int total = 0;
        for (Compte c : comptes) {
            total += c.getBalance();
        }
        return total;
    }

    void deposer(int montant) throws OperationInterrompueException,
    CompteBlokueException;

    int retirer(int montant) throws OperationInterrompueException,
    CompteBlokueException;

    int getBalance() throws OperationInterrompueException;

}
```

## Héritage d'interface

Une interface peut hériter d'autres interfaces. Contrairement aux classes qui ne peuvent avoir qu'une classe parente, une interface peut avoir autant d'interfaces parentes que nécessaire. Pour déclarer un héritage, on utilise le mot-clé **extends**.

```
package dev.gayerie.animal;

public interface Omnivore extends Carnivore, Herbivore {

}
```

Une classe concrète qui implémente une interface doit donc disposer d'une implémentation pour les méthodes de cette interface mais également pour toutes les méthodes des interfaces dont cette dernière hérite.

```
package dev.gayerie.animal;

public class Humain extends Animal implements Omnivore {
```



```

@Override
public void manger(Animal animal) {
    // ...
}

@Override
public void manger(Vegetal vegetal) {
    // ...
}

}

```

L'héritage d'interface permet d'introduire de nouveaux types par agrégat. Dans l'exemple ci-dessus, nous faisons apparaître la notion d'omnivore simplement comme étant à la fois un carnivore et un herbivore.

## Les interfaces marqueurs

Comme chaque interface introduit un nouveau type, il est possible de contrôler grâce au mot-clé **instanceof** qu'une variable, un paramètre ou un attribut est bien une instance compatible avec cette interface.

```

Humain bob = new Humain();
if (bob instanceof Carnivore) {
    System.out.println("bob mange de la viande");
}

```

En Java, on utilise cette possibilité pour créer des interfaces marqueurs. Une interface marqueur n'a généralement pas de méthode, elle sert juste à introduire un nouveau type. Il est ensuite possible de changer le comportement d'une méthode si une variable, un paramètre ou un attribut implémente cette interface.

```

package dev.gayerie.animal;

public interface Cannibale {
}

package dev.gayerie.animal;

public class Humain extends Animal implements Omnivore {

    @Override
    public void manger(Animal animal) {
        if (!(animal instanceof Humain) || this instanceof Cannibale) {
            // ...
        }
    }

    @Override
    public void manger(Vegetal vegetal) {
        // ...
    }
}

```

```
}
```

Dans l'exemple ci-dessus, *Cannibale* agit comme une interface marqueur, elle permet à une classe héritant de *Humain* de manger une instance d'humain. Pour cela, il suffit de déclarer que cette nouvelle classe implémente *Cannibale* :

```
package dev.gayerie.animal;

public class Anthropophage extends Humain implements Cannibale {

}
```

Même si la classe *Anthropophage* ne redéfinit aucune méthode de sa classe parente, le fait de déclarer l'interface marqueur *Cannibale* suffit à modifier son comportement.

Le principe de l'interface marqueur est quelques fois utilisé dans l'API standard de Java. Par exemple, La méthode [clone](#) déclarée par [Object](#) jette une [CloneNotSupportedException](#) si elle est appelée sur une instance qui n'implémente pas l'interface [Cloneable](#). Cela permet de fournir une méthode par défaut pour créer une copie d'un objet mais sans activer la fonctionnalité. Il faut que la classe déclare son intention d'être clonable grâce à l'interface marqueur.

## Implémentation par défaut

Il est parfois difficile de faire évoluer une application qui utilise intensivement les interfaces. Reprenons notre exemple du *Compte*. Imaginons que nous souhaitions ajouter la méthode *transférer* qui consiste à transférer le solde d'un compte vers un autre.

```
package dev.gayerie.compte;

public interface Compte {

    void deposer(int montant) throws OperationInterrompueException,
        CompteBlokueException;

    int retirer(int montant) throws OperationInterrompueException,
        CompteBlokueException;

    int getBalance() throws OperationInterrompueException;

    void transferer(Compte destination) throws OperationInterrompueException,
        CompteBlokueException;

}
```

En ajoutant une nouvelle méthode à notre interface, nous devons fournir une implémentation pour cette méthode dans toutes les classes que nous avons créées pour qu'elles continuent à compiler. Mais si d'autres équipes de développement utilisent notre code et ont, elles-aussi, créé des implémentations pour l'interface *Compte*, alors elles devront adapter leur code au moment d'intégrer la dernière version de notre interface.

Comme les interfaces servent précisément à découpler deux implémentations, elles sont très souvent utilisées dans les bibliothèques et les frameworks. D'un côté, les interfaces introduisent une meilleure souplesse mais, d'un autre côté, elles entraînent une grande rigidité car il peut être difficile de les faire évoluer sans risquer de casser des implémentations existantes.

Pour palier partiellement à ce problème, une interface peut fournir une implémentation par défaut de ses méthodes. Ainsi, si une classe concrète qui implémente cette interface n'implémente pas une méthode par défaut, c'est le code de l'interface qui s'exécutera. Une méthode par défaut doit obligatoirement avoir le mot-clé **default** dans sa signature.

```
package dev.gayerie.compte;

public interface Compte {

    void deposer(int montant) throws OperationInterrompueException,
                                   CompteBlokueException;

    int retirer(int montant) throws OperationInterrompueException,
                                   CompteBlokueException;

    int getBalance() throws OperationInterrompueException;

    default void transferer(Compte destination) throws
                                   OperationInterrompueException,
                                   CompteBlokueException {

        if (destination == this) {
            return;
        }
        int montant = this.getBalance();
        if (montant <= 0) {
            return;
        }
        destination.deposer(montant);
        boolean retraitOk = false;
        try {
            this.retirer(montant);
            retraitOk = true;
        } finally {
            if (!retraitOk) {
                destination.retirer(montant);
            }
        }
    }
}
```

Une classe implémentant *Compte* n'a pas besoin de fournir une implémentation pour la méthode *transferer*. La classe *CompteBancaire* que nous avons implémentée au début de ce chapitre continuera de compiler et de fonctionner comme attendu tout en ayant une méthode supplémentaire.

Prudence

L'implémentation par défaut de méthode dans une interface la rapproche beaucoup du fonctionnement d'une classe abstraite. Cependant leurs usages sont différents. L'implémentation d'une méthode dans une classe abstraite est courant car la classe abstraite a cette notion de mutualisation de code. Par contre, l'implémentation par défaut de méthode dans une interface est très rare. Elle est réservée pour les types de situations décrits précédemment, afin d'éviter de casser les implémentations existantes.

## La ségrégation d'interface

En programmation objet, le [principe de ségrégation d'interface](#) stipule qu'un client ne devrait pas avoir accès à plus de méthodes d'un objet que ce dont il a vraiment besoin. L'objectif est de limiter au strict minimum les interactions possibles entre un objet et ces clients afin d'assurer un couplage minimal et faciliter ainsi les évolutions et le refactoring. En Java, le [principe de ségrégation d'interface](#) a deux conséquences :

1. Le type des variables, paramètres et attributs doit être choisi judicieusement pour restreindre au type minimum nécessaire par le code.
2. Une interface ne doit pas déclarer *trop* de méthodes.

Le premier point implique qu'il est préférable de manipuler les objets à travers leurs interfaces plutôt que d'utiliser le type réel de l'objet. Un exemple classique en Java concerne l'API des [collections](#). Il s'agit de classes permettant de gérer un ensemble d'objets. Elles apportent des fonctionnalités plus avancées que les tableaux. Par exemple la classe [java.util.ArrayList](#) permet de gérer une liste d'objets. Cette classe autorise l'ajout en fin de liste, l'insertion, la suppression et bien évidemment l'accès à un élément selon son index et le parcours complet des éléments.

Un programme qui crée une [ArrayList](#) pour stocker un ensemble d'éléments n'utilisera jamais une variable de type [ArrayList](#) mais plutôt une variable ayant le type d'une interface implémentée par cette classe.

```
// Utilisation de l'interface List
List maListe = new ArrayList();
// Utilisation de l'interface Collection
Collection maListe = new ArrayList();
// Utilisation de l'interface Iterable
Iterable maListe = new ArrayList();
```

Plus une partie d'une application a recours à des interfaces pour interagir avec les autres parties d'une application, plus il est simple d'introduire des nouvelles classes implémentant les interfaces attendues et qui pourront être directement utilisées.

Le second point est lié au principe [SOLID](#) de la [responsabilité unique](#). Une interface est conçue pour représenter un type de relation entre la classe qui l'implémente et ses clients. Plus le nombre de méthodes augmente, plus il y a de risque que l'interface représente en fait plusieurs types de relation. Dans ce cas, l'héritage entre interfaces et/ou l'implémentation de plusieurs interfaces deviennent une bonne solution pour isoler chaque relation.

Reprenons notre exemple de l'interface *Compte*. Si notre système est composé d'un sous-système de consultation, d'un sous-système de retrait et d'un sous-système de gestion de comptes alors cette interface devrait probablement être séparée en plusieurs interfaces afin d'isoler chaque responsabilité.

Une interface utilisée par le sous-système de consultation :

```
package dev.gayerie.compte;

public interface CompteConsultable {

    int getBalance() throws OperationInterrompueException;

}
```

Une interface utilisée par le sous-système de retrait :

```
package dev.gayerie.compte;

public interface OperationDeRetrait {

    int retirer(int montant) throws OperationInterrompueException,
        CompteBloqueException;

}
```

Une interface plus complexe utilisée par le système de gestion de comptes :

```
package dev.gayerie.compte;

public interface Compte extends CompteConsultable, OperationDeRetrait {

    void deposer(int montant) throws OperationInterrompueException,
        CompteBloqueException;

    default void transferer(Compte destination) throws
        OperationInterrompueException,
        CompteBloqueException {

        if (destination == this) {
            return;
        }
        int montant = this.getBalance();
        if (montant <= 0) {
            return;
        }
        destination.deposer(montant);
        boolean retraitOk = false;
        try {
            this.retirer(montant);
            retraitOk = true;
        } finally {
            if (!retraitOk) {
                destination.retirer(montant);
            }
        }
    }

}
```

# L'inversion de dépendance

Lorsque nous avons vu les constructeurs, nous avons vu que nous pouvions réaliser de [l'injection de dépendance](#) en passant comme paramètres de constructeur les objets nécessaires au fonctionnement d'une classe plutôt que de laisser la nouvelle instance créer ces objets elle-même. Grâce à la notion d'interface, nous pouvons réaliser une injection de dépendance en découplant totalement l'utilisation de l'objet passé par injection de son implémentation.

Si nous souhaitons créer une classe pour représenter une transaction bancaire, nous pouvons réaliser l'implémentation suivante :

```
package dev.gayerie.compte;

import java.time.Instant;

public class TransactionBancaire {

    private final Compte compte;
    private final int montant;
    private Instant date;

    public TransactionBancaire(Compte compte, int montant) {
        this.compte = compte;
        this.montant = montant;
    }

    public void effectuer() throws OperationInterrompueException,
    CompteBloqueException {
        if (isEffectuee()) {
            return;
        }
        compte.retirer(montant);
        date = Instant.now();
    }

    public void annuler() throws OperationInterrompueException,
    CompteBloqueException {
        if (! isEffectuee()) {
            return;
        }
        compte.deposer(montant);
        date = null;
    }

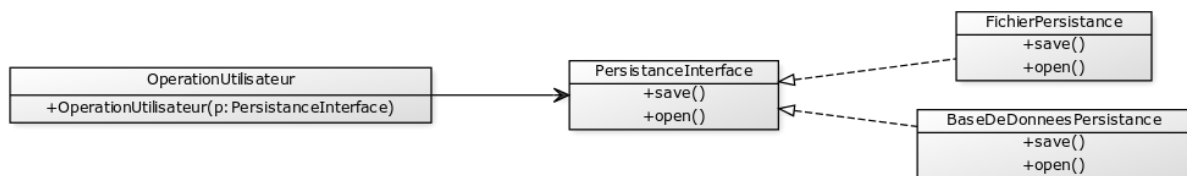
    public boolean isEffectuee() {
        return date != null;
    }

    public Instant getDate() {
        return date;
    }
}
```

L'implémentation précédente permet d'effectuer une transaction pour un compte donné et un montant donné et mémorise la date. Elle permet également d'annuler la transaction. Dans cette implémentation, nous avons réalisé une **inversion de dépendance**. La transaction ne connaît pas la nature exacte de l'objet *Compte* qu'elle manipule. La classe *TransactionBancaire* fonctionnera quelle que soit l'implémentation sous-jacente de l'interface *Compte*.

L'inversion de dépendance est un principe de programmation objet qui stipule que si une classe A est dépendante d'une classe B, alors il peut être souhaitable que, non seulement la classe A reçoive une instance de B par injection, mais également que B ne soit connue qu'à travers une interface.

L'inversion de dépendance est très souvent utilisée pour isoler les couches logicielles d'une architecture. Au sein d'une application, nous pouvons disposer d'un ensemble de classes pour gérer des opérations utilisateur et d'un ensemble de classes pour assurer la persistance des informations.



L'architecture logicielle peut utiliser l'inversion de dépendance pour assurer que les opérations utilisateur qui ont besoin de réaliser des opérations persistantes réalisent des appels à travers des interfaces qui sont injectées. D'un côté, on peut imaginer implémenter différentes classes gérant la persistance pour sauver les informations dans des fichiers, dans des bases de données ou sur des serveurs distants (et même nulle part si on souhaite exécuter le code dans un environnement de test). D'un autre côté on peut créer et faire évoluer un système de persistance en ayant une dépendance minimale aux opérations utilisateur puisque le système de persistance doit juste fournir des implémentations conformes aux interfaces.

## Méthodes et classes génériques

Parfois, on souhaite créer une classe mais on ne souhaite pas préciser le type exact de tel ou tel attribut. C'est souvent le cas quand la classe sert de conteneur à un autre type de classe. En Java, il est possible de créer des méthodes et des classes dont certains types sont des paramètres qui seront résolus au moment de l'invocation et de l'instanciation. On parle alors de méthodes et de classes génériques.

## L'exemple de la classe *ArrayList*

En Java, l'API standard fournit un ensemble de classes que l'on appelle couramment les [collections](#). Ces classes permettent de gérer un ensemble d'objets. Elles apportent des fonctionnalités plus avancées que les tableaux. Par exemple la classe [java.util.ArrayList](#) permet de gérer une liste d'objets. Cette classe autorise l'ajout en fin de liste, l'insertion, la suppression et bien évidemment l'accès à un élément selon son index et le parcours complet des éléments.

```
package dev.gayerie;

import java.util.ArrayList;
import java.util.List;

public class TestArrayList {
```

```

public static void main(String[] args) {
    List list = new ArrayList();

    list.add("bonjour le monde");
    list.add(1); // boxing ! list.add(Integer.valueOf(1));
    list.add(new Object());

    String s1 = (String) list.get(0);
    String s2 = (String) list.get(1); // ERREUR à l'exécution :
    ClassCastException
    String s3 = (String) list.get(2); // ERREUR à l'exécution :
    ClassCastException
}
}

```

Pour des instances de la classe [ArrayList](#), on peut ajouter des éléments avec la méthode [ArrayList.add](#) et accéder à un élément selon son index avec la méthode [ArrayList.get](#). Dans l'exemple précédent, on voit que cela n'est pas sans risque. En effet, un objet de type [ArrayList](#) peut contenir tout type d'objet. Donc quand le programme accède à un élément d'une instance de [ArrayList](#), il doit réaliser explicitement un transtypage (*cast*) avec le risque que cela suppose de se tromper de type. Ce type de classe exige donc beaucoup de rigueur d'utilisation pour les développeurs.

Une situation plus simple serait de pouvoir déclarer en tant que développeur qu'une instance de [ArrayList](#) se limite à un type d'éléments : par exemple au type [String](#). Ainsi le compilateur pourrait signaler une erreur si le programme tente d'ajouter un élément qui n'est pas compatible avec le type [String](#) ou s'il veut récupérer un élément dans une variable qui n'est pas d'un type compatible. Les classes et les méthodes génériques permettent de gérer ce type de situation. Elles sont une aide pour les développeurs afin d'écrire des programmes plus robustes.

## Création et assignation d'une classe générique

La classe [ArrayList](#) et l'interface [List](#) sont justement une classe générique et une interface générique supportant un type paramétré.

Note

[List](#) est une interface implémentée notamment par la classe [ArrayList](#).

Il est possible, par exemple, de déclarer qu'une instance est une liste de chaînes de caractères :

```

List<String> list = new ArrayList<String>();

```

On ajoute entre les signes < et > le paramètre de type géré par la liste. À partir de cette information, le compilateur va pouvoir nous aider à résoudre les ambiguïtés. Il peut maintenant déterminer si un élément peut être ajouté ou assigné à une variable sans nécessiter un transtypage explicite du développeur.

```

list.add("bonjour");
String s = list.get(0); // l'opération de transtypage n'est plus nécessaire

```

Par contre :



```
list.add(1); // Erreur de compilation : type String attendu

Object o = "je suis une chaîne affectée à une variable de type Object";
list.add(o); // Erreur de compilation : type String attendu

voiture v = (Voiture) list.get(0); // Erreur de compilation voiture n'hérite pas
de String
```

Pour les types paramétrés, le principe de substitution s'applique. Comme la classe [String](#) hérite de la classe [Object](#), il est possible de récupérer un élément de la liste dans une variable de type [Object](#) :

```
Object o = list.get(0); // OK
```

Une classe générique peut permettre de déclarer plusieurs types paramétrés. Par exemple, la classe [java.util.HashMap](#) permet de créer des tableaux associatifs (parfois appelés dictionnaires ou plus simplement *maps*) pour associer une clé à une valeur. La classe [HashMap](#) permet de spécifier le type de la clé et le type de la valeur. Pour créer un tableau associatif entre le nom d'une personne (type [String](#)) et une instance de la classe *Personne*, on peut écrire :

```
Map<String, Personne> tableauAssociatif = new HashMap<String, Personne>();
```

Note

[Map](#) est une interface implémentée notamment par la classe [HashMap](#).

## Notation en diamant

Lors de l'initialisation, il n'est pas nécessaire de préciser le type des paramètres à droite de l'expression. Le compilateur peut réaliser une inférence de types à partir de la variable à gauche de l'expression :

```
Map<String, Personne> tableauAssociatif = new HashMap<>();
List<Integer> listeDeNombres = new ArrayList<>();
```

Il s'agit d'un raccourci d'écriture qui évite de se répéter. On appelle la notation `<>`, la notation en diamant.

## Substitution et type générique

Avec l'héritage, nous avons vu que nous pouvons affecter à une variable (ou à un paramètre ou un attribut) une référence d'un objet du même type ou d'un type qui en hérite. On appelle cela le principe de substitution.

```
Object obj = new String();
```

Dans l'exemple ci-dessus, il est possible d'affecter un objet du type [String](#) à une variable de type [Object](#) car [String](#) hérite de [Object](#). Avec les types génériques, le principe de substitution est possible mais devient un peu plus complexe. Par exemple :

```
List<Object> listeString = new ArrayList<String>(); // ERREUR DE COMPILATION
```

Il n'est pas possible d'affecter une [ArrayList](#) de [String](#) à une variable de type [ArrayList](#) de [Object](#). En effet, si cela était autorisé, il serait alors possible d'ajouter avec la méthode [List.add](#) n'importe quel objet de type [Object](#) ou d'un type héritant de [Object](#). Donc un développeur pourrait ajouter à cette liste une instance d'une classe *Voiture* par exemple sans que le compilateur puisse détecter le problème :

```
listeString.add(new Voiture()); // Il vaut mieux ne pas pouvoir faire cela !
```

Pour les types génériques, il est nécessaire d'introduire la notion de type borné (*bounded type*) pour pouvoir gérer la substitution correctement. Mais avant d'aller plus loin, il est important de comprendre qu'il existe deux cas fondamentaux. Prenons une exemple de classes qui héritent les unes des autres : *Vehicule*, *Voiture*, *VoitureDeCourse*.

```
package dev.gayerie;

public class Vehicule {
    // ...
}

package dev.gayerie;

public class Voiture extends Vehicule {
    // ...
}

package dev.gayerie;

public class VoitureDeCourse extends Voiture {
    // ...
}
```

Si nous créons une instance de [ArrayList](#) pour le type *Voiture* :

```
ArrayList<Voiture> listeVoitures = new ArrayList<>();
```

Si on souhaite ajouter des objets dans cette liste, le principe de substitution nous assure que nous pouvons ajouter sans risque une instance de la classe *Voiture* ou une instance de la classe *VoitureDeCourse* (puisque une *VoitureDeCourse* est une *Voiture*).

```
listeVoitures.add(new Voiture());
listeVoitures.add(new VoitureDeCourse());
```

Si on souhaite accéder à un élément de cette liste, le principe de substitution nous dit que nous pouvons affecter sans risque un élément de cette liste à une variable de type *Voiture* ou de type *Vehicule* (puisque une *Voiture* est un *Vehicule*).

```
Voiture voiture = listeVoitures.get(0);
Vehicule vehicule = listeVoitures.get(0);
```

Il y a donc une différence selon que nous souhaitons ajouter un élément à cette liste ou que nous souhaitons consulter un élément de cette liste. L'ajout s'apparente à utiliser le type paramétré comme paramètre d'entrée et la consultation s'apparente à utiliser le type paramétré comme paramètre de sortie.

Une liste de *Voiture* peut donc aussi être considérée comme :

- une liste de quelque chose qui est au mieux de type *Voiture* dans le cas où l'on souhaite uniquement consulter les éléments de la liste.
- une liste de quelque chose qui est au moins de type *Voiture* dans le cas où on ne souhaite qu'ajouter de nouveaux éléments à la liste.

Il est possible d'exprimer cela en Java. Pour le premier cas, *Voiture* correspond à la borne supérieure (*upper bounded type*) et nous pouvons écrire l'expression suivante :

```
List<? extends Voiture> listePourConsultation = listeVoitures;  
Voiture voiture = listePourConsultation.get(0);
```

L'expression `<? extends Voiture>` désigne une **capture** et permet au compilateur de déterminer l'ensemble des classes acceptables.

Pour le second cas, *Voiture* correspond à la borne inférieure (*lower bounded type*) et nous pouvons écrire l'expression suivante :

```
List<? super Voiture> listePourAjout = listeVoitures;  
listePourAjout.add(new Voiture());  
listePourAjout.add(new VoitureDeCourse());
```

Il est également possible d'utiliser uniquement le caractère de substitution `?` dans la déclaration de la capture :

```
List<?> listePourAjout = listeVoitures;
```

Dans ce cas, on ne fournit aucune information au compilateur sur le type paramétré de l'instance de la classe.

Note

Pour une classe supportant plusieurs types génériques, on peut au besoin déclarer une capture pour chaque type :

```
Map<?, ? extends Personne> tableauAssociatif = new HashMap<String, Personne>();
```

La déclaration de capture est surtout utile pour la création de méthodes et classes supportant les types génériques.

## Écrire une méthode générique

L'utilisation des captures devient utile lorsque l'on veut écrire une méthode générique qui supporte les types paramétrés. Reprenons notre exemple ci-dessus des classes *Vehicule*, *Voiture* et *VoitureDeCourse*. La classe *Vehicule* définit la propriété *vitesse* accessible en lecture :

```
package dev.gayerie;

public class Vehicule {

    private int vitesse;

    public int getVitesse() {
        return vitesse;
    }

}
```

Nous voulons ajouter la méthode de classe *getPlusRapide* qui retourne le véhicule le plus rapide parmi une liste de véhicules :

```
package dev.gayerie;

import java.util.List;

public class Vehicule {

    private int vitesse;

    public int getVitesse() {
        return vitesse;
    }

    public static Vehicule getPlusRapide(List<Vehicule> vehicules) {
        Vehicule plusRapide = null;
        int vitesse = 0;
        for (Vehicule vehicule : vehicules) {
            if(vehicule.getVitesse() >= vitesse) {
                vitesse = vehicule.getVitesse();
                plusRapide = vehicule;
            }
        }
        return plusRapide;
    }

}
```

Si nous nous contentons de cette implémentation, nous allons certainement rencontrer quelques problèmes lors de l'utilisation de la méthode *Vehicule.getPlusRapide* :

```
List<Voiture> listeVoitures = new ArrayList<>();
listeVoitures.add(new Voiture());
listeVoitures.add(new VoitureDeCourse());

Vehicule plusRapide = Vehicule.getPlusRapide(listeVoitures); // ERREUR DE
COMPILATION
```

Le code ci-dessus ne compile pas. En effet, on tente de passer en paramètre à la méthode *Vehicule.getPlusRapide* une liste de type *Voiture* alors que la méthode est écrite pour une liste de type *Vehicule*. Nous pourrions utiliser la surcharge en fournissant une implémentation pour chaque type de liste, mais la bonne solution est de déclarer *Vehicule.getPlusRapide* comme une méthode générique :

```
package dev.gayerie;

import java.util.ArrayList;
import java.util.List;

public class Vehicule {

    private int vitesse;

    public int getVitesse() {
        return vitesse;
    }

    public static <T extends Vehicule> T getPlusRapide(List<T> vehicules) {
        T plusRapide = null;
        int vitesse = 0;
        for (T vehicule : vehicules) {
            if(vehicule.getVitesse() >= vitesse) {
                vitesse = vehicule.getVitesse();
                plusRapide = vehicule;
            }
        }
        return plusRapide;
    }
}
```

Pour déclarer une méthode générique, il faut décrire le type ou les types paramétrés supportés entre <>. Pour l'exemple ci-dessus, on utilise la capture *T*. *T* est le nom du type générique que l'on peut utiliser dans la signature et le code de la méthode. Dans notre exemple *T* représente donc le type *Vehicule* ou un type qui hérite de *Vehicule*. On peut donc parcourir les éléments de type *T* de la liste, lire leur propriété *vitesse* et retourner l'instance pour laquelle la vitesse est la plus élevée.

Maintenant nous pouvons utiliser cette méthode en passant une liste de type *Vehicule*, de *Voiture* ou de *VoitureDeCourse*

```
List<Voiture> listeVoitures = new ArrayList<>();
listeVoitures.add(new Voiture());
listeVoitures.add(new VoitureDeCourse());

Voiture plusRapide = Vehicule.getPlusRapide(listeVoitures);
```

Notez que la méthode *Voiture.getPlusRapide* retourne le type générique *T*. Donc le compilateur infère que si on appelle cette méthode avec une liste de type *Voiture* en paramètre alors cette méthode retourne une instance assignable à une variable de type *Voiture*.

Note

Par convention un type paramétré s'écrit avec une seule lettre en majuscule :

- T pour identifier un type générique en général
- E pour identifier un type générique qui représente un élément
- K pour identifier un type générique qui est utilisé comme clé (*key*)
- V pour identifier un type générique qui est utilisé comme une valeur
- U, V, W pour identifier une suite de types génériques si la méthode supporte plusieurs types génériques.

## Écrire une classe générique

Une classe peut également être générique et supporter un ou plusieurs types paramétrés. Par exemple, si nous voulons implémenter une classe *Paire* qui permet d'associer une instance d'une classe avec une instance d'une autre classe, il suffit d'utiliser des types paramétrés en les déclarant entre <> après le nom de la classe :

```
package dev.gayerie;

public class Paire<U, V> {

    private U valeurGauche;
    private V valeurDroite;

    public Paire(U valeurGauche, V valeurDroite) {
        this.valeurGauche = valeurGauche;
        this.valeurDroite = valeurDroite;
    }

    public U getValeurGauche() {
        return valeurGauche;
    }

    public V getValeurDroite() {
        return valeurDroite;
    }

    @Override
    public String toString() {
        return valeurGauche + " " + valeurDroite;
    }

}
```

La classe *Paire* peut maintenant être utilisée pour associer n'importe quel type d'instances :

```
Paire<String, Integer> paireStringInteger = new Paire<>("test", 1);

Paire<Voiture, Voiture> paireVoiturevoiture = new Paire<>(new Voiture(), new
Voiture());
```

Comme pour les méthodes, il est possible de préciser une capture pour les types paramétrés :

```
public class Paire<U extends Number, V> {
```

```

private U valeurGauche;
private V valeurDroite;

public Paire(U valeurGauche, V valeurDroite) {
    this.valeurGauche = valeurGauche;
    this.valeurDroite = valeurDroite;
}

public U getValeurGauche() {
    return valeurGauche;
}

public V getValeurDroite() {
    return valeurDroite;
}

@Override
public String toString() {
    return valeurGauche + " " + valeurDroite;
}
}

```

En précisant **<U extends Number, V>** dans la déclaration de la classe, nous limitons le premier type paramétré au type [Number](#) ou un type qui en hérite.

Note

La classe [Number](#) est la classe parente des classes enveloppes [Integer](#), [Long](#), [Short](#), [Byte](#), [Float](#) et [Double](#).

```

Paire<Integer, String> paireIntegerString = new Paire<>(1, "Test");
Paire<Float, String> paireFloatString = new Paire<>(1.3f, "Test");

```

## Limitations

Les méthodes et les classes génériques ont des limitations.

Les types paramétrés ne s'appliquent que pour des classes. On ne peut pas spécifier un type primitif. Si on désire créer une instance de [ArrayList](#) pour des nombres, alors on peut passer par la classe enveloppe [Integer](#) :

```

ArrayList<Integer> listeDeNombres = new ArrayList<Integer>();

```

La déclaration d'un type paramétré ne fait pas partie du nom d'une classe. Donc il n'est pas possible de spécifier un type paramétré avec le mot-clé **instanceof** :

```

if (listeVoiture instanceof List<Voiture>) { // ERREUR DE COMPILATION
    // ...
}

```

Il n'est pas possible d'instancier un type paramétré dans le corps d'une méthode générique :

```
public static <T> doSomething(List<T> l) {
    l.add(new T()); // ERREUR DE COMPILATION
}
```

Il n'est pas possible de déclarer un attribut de classe (**static**) en utilisant un type paramétré :

```
public class Test<T> {

    private static T attribut; // ERREUR DE COMPILATION

}
```

Il n'est pas possible de créer des tableaux en spécifiant des types paramétrés :

```
List<String>[] tableau = new List<String>[10]; // ERREUR DE COMPILATION
```

Il n'est pas possible d'utiliser un type paramétré dans une expression **catch** :

```
public static <T extends Exception> void doSomething() {
    try {
        // ...
    } catch (T t) { // ERREUR DE COMPILATION
        // ...
    }
}
```

Il n'est pas possible de surcharger (*overload*) une méthode en ne changeant que le type paramétré d'un paramètre :

```
public class Test {

    public void doSomething(List<String> l) {
        // ...
    }

    public void doSomething(List<Integer> l) { // ERREUR DE COMPILATION
        // ...
    }

}
```

#### Note

Beaucoup des limitations des classes et des méthodes génériques viennent de ce que l'on appelle *l'effacement du type* (*type erasure*). Les types paramétrés ne sont pas conservés dans le bytecode produit par le compilateur.

Pour l'exemple ci-dessus, la suppression du type par le compilateur conduit à la classe suivante :



```
public class Test {  
  
    public void doSomething(List l) {  
        // ...  
    }  
  
    public void doSomething(List l) {  
        // ...  
    }  
}
```

Donc, le résultat de la compilation amènerait à déclarer une classe avec deux méthodes strictement identiques. Voilà pourquoi il n'est pas possible de surcharger une méthode juste en changeant le type paramétré d'un paramètre.

## Les collections

Lors d'un [chapitre précédent](#), nous avons vu qu'il est possible de déclarer des tableaux en Java pour gérer un ensemble d'éléments. Cependant, ce type de structure reste limité : un tableau a une taille fixe (il est impossible d'ajouter ou d'enlever des éléments d'un tableau). De plus, il est souvent utile de disposer d'autres structures de données pour gérer des groupes d'éléments.

On appelle *collections* un ensemble de classes et d'interfaces fournies par l'API standard et disponibles pour la plupart dans le package [java.util](#). Parmi ces collections, on trouve les listes (*lists*), les ensembles (*sets*) et les tableaux associatifs (*maps*). Elles forment ce que l'on appelle le [Java Collections Framework](#).

Toutes ces classes et interfaces sont génériques. On ne peut donc créer que des collections d'objets. Si vous souhaitez créer une collection pour un type primitif, vous devez utiliser la classe enveloppe correspondante (par exemple [Integer](#) pour **int**).

## Les listes

Une liste est une collection ordonnée d'éléments. Il existe différentes façons d'implémenter des listes dont les performances sont optimisées soit pour les accès aléatoires aux éléments soit pour les opérations d'insertion et de suppression d'éléments.

Java propose plusieurs classes d'implémentation pour les listes selon les besoins de performance. Comme toutes ces classes implémentent des interfaces communes, il est conseillé de manipuler les instances de ces classes uniquement à travers des variables du type de l'interface adaptée : [Collection](#), [List](#), [Queue](#) ou [Deque](#).

L'interface [Collection](#) dont hérite toutes les autres interfaces pour les listes, hérite elle-même de [Iterable](#). Cela signifie que toutes les classes et toutes les interfaces servant à représenter des listes dans le [Java Collections Framework](#) peuvent être parcourues avec une structure de **for** amélioré (*foreach*).

## La classe ArrayList

La classe [java.util.ArrayList](#) est une implémentation de l'interface [List](#). Elle stocke les éléments de la liste sous la forme de blocs en mémoire. Cela signifie que la classe [ArrayList](#) est très performante pour les accès aléatoire en lecture aux éléments de la liste. Par contre, les opérations d'ajout et de suppression d'un élément se font en temps linéaire. Elle est donc moins performante

que la classe [LinkedList](#) sur ce point.

```
List<String> liste = new ArrayList<String>();

liste.add("une première chaîne");
liste.add("une troisième chaîne");

System.out.println(liste.size()); // 2

// insertion d'un élément
liste.add(1, "une seconde chaîne");

System.out.println(liste.size()); // 3

for (String s : liste) {
    System.out.println(s);
}

String premierElement = liste.get(0);

System.out.println(liste.contains("une première chaîne")); // true
System.out.println(liste.contains("quelque chose qui n'est pas dans la liste"));
// false

// suppression du dernier élément de la liste
liste.remove(liste.size() - 1);

// ajout de tous les éléments d'une autre liste à la fin de la liste
liste.addAll(Arrays.asList("une autre chaîne", "et encore une autre chaîne"));

System.out.println(liste.size()); // 4

// [une première chaîne, une seconde chaîne, une autre chaîne, et encore une
autre chaîne]
System.out.println(liste);
```

Il est possible de réserver de l'espace mémoire pour une liste pouvant contenir n éléments. Pour cela, on peut passer la taille voulue à la création d'une instance de [ArrayList](#) ou en appelant la méthode [ArrayList.ensureCapacity](#). La liste ne change pas de taille pour autant, un espace mémoire est simplement alloué en prévision.

```
// capacité de 10
ArrayList<String> liste = new ArrayList<String>(10);

// capacité d'au moins 100
liste.ensureCapacity(100);

System.out.println(liste.size()); // 0
```

# La classe LinkedList

La classe [java.util.LinkedList](#) est une implémentation de l'interface [List](#). Sa représentation interne est une liste doublement chaînée. Cela signifie que la classe [LinkedList](#) est très performante pour les opérations d'insertion et de suppression d'éléments. Par contre, l'accès aléatoire en lecture aux éléments se fait en temps linéaire. Elle est donc moins performante que la classe [ArrayList](#) sur ce point.

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

liste.add("une première chaîne");
liste.add("une troisième chaîne");

System.out.println(liste.size()); // 2

// insertion d'un élément
liste.add(1, "une seconde chaîne");

System.out.println(liste.size()); // 3

for (String s : liste) {
    System.out.println(s);
}

String premierElement = liste.get(0);

System.out.println(liste.contains("une première chaîne")); // true
System.out.println(liste.contains("quelque chose qui n'est pas dans la liste"));
// false

// suppression du dernier élément de la liste
liste.remove(liste.size() - 1);

// ajout de tous les éléments d'une autre liste à la fin de la liste
liste.addAll(Arrays.asList("une autre chaîne", "et encore une autre chaîne"));

System.out.println(liste.size()); // 4
System.out.println(liste);
```

La classe [LinkedList](#) implémente également les interfaces [Queue](#) et [Deque](#) (*double ended queue*), elle peut donc représenter des structures de type LIFO (*Last In First Out*) ou FIFO (*First In First Out*).

```
Queue<String> queue = new LinkedList<String>();

// insère un élément dans la file
queue.offer("un élément");

// lit l'élément en tête de la file sans l'enlever de la file
System.out.println(queue.peek()); // "un élément"
// lit l'élément en tête de la file et l'enleve de la file
System.out.println(queue.poll()); // "un élément"

System.out.println(queue.isEmpty()); // true
```

```
Deque<String> deque = new LinkedList<String>();

// empile deux éléments
deque.push("élément 1");
deque.push("élément 2");

// lit le premier élément de la file sans l'enlever
System.out.println(deque.peekFirst()); // élément 2
// lit le dernier élément de la file sans l'enlever
System.out.println(deque.peekLast()); // élément 1
// lit l'élément de tête de la file sans l'enlever
System.out.println(deque.peek()); // élément 2
// lit l'élément de tête de la file et l'enlève
System.out.println(deque.pop()); // élément 2
System.out.println(deque.pop()); // élément 1

System.out.println(deque.isEmpty()); // true
```

## La classe ArrayDeque

La classe [java.util.ArrayDeque](#) est une implémentation des interfaces [Queue](#) et [Deque](#) (mais elle **n'implémente pas** [List](#)). Elle est conçue pour être plus performante que [LinkedList](#) pour les opérations d'ajout et de suppression en tête et en fin de liste. Si vous voulez utiliser une collection uniquement pour représenter une file ou une pile de type LIFO (*Last In First Out*) ou FIFO (*First In First Out*), alors il est préférable de créer une instance de la classe [ArrayDeque](#).

```
Queue<String> queue = new ArrayDeque<String>();

// insère un élément dans la file
queue.offer("un élément");

// lit l'élément en tête de la file sans l'enlever de la file
System.out.println(queue.peek()); // "un élément"
// lit l'élément en tête de la file et l'enlève de la file
System.out.println(queue.poll()); // "un élément"

System.out.println(queue.isEmpty()); // true
Deque<String> deque = new ArrayDeque<String>();

// empile deux éléments
deque.push("élément 1");
deque.push("élément 2");

// lit le premier élément de la file sans l'enlever
System.out.println(deque.peekFirst()); // élément 2
// lit le dernier élément de la file sans l'enlever
System.out.println(deque.peekLast()); // élément 1
// lit l'élément de tête de la file sans l'enlever
System.out.println(deque.peek()); // élément 2
// lit l'élément de tête de la file et l'enlève
System.out.println(deque.pop()); // élément 2
System.out.println(deque.pop()); // élément 1

System.out.println(deque.isEmpty()); // true
```

Comme pour la classe [ArrayList](#), il est possible de réserver un espace mémoire pour n éléments au moment de la création d'une instance de [ArrayDeque](#).

```
// Assurer une capacité minimale de 100 éléments
ArrayDeque<String> arrayDeque = new ArrayDeque<>(100);

System.out.println(arrayDeque.size()); // 0
```

## La classe PriorityQueue

La classe [java.util.PriorityQueue](#) permet d'ajouter des éléments dans une file selon un ordre naturel : soit parce que les éléments de la file implémentent l'interface [Comparable](#), soit parce qu'une instance de [Comparator](#) a été fournie à la création de l'instance de [PriorityQueue](#). Quel que soit l'ordre d'insertion, les éléments seront extraits de la file selon l'ordre naturel.

```
Queue<String> queue = new PriorityQueue<>();

queue.add("i");
queue.add("e");
queue.add("u");
queue.add("o");
queue.add("a");
queue.add("y");

System.out.println(queue.poll()); // a
System.out.println(queue.poll()); // e
System.out.println(queue.poll()); // i
System.out.println(queue.poll()); // o
System.out.println(queue.poll()); // u
System.out.println(queue.poll()); // y
```

Prudence

La classe [PriorityQueue](#) ne garantit pas que l'ordre naturel sera respecté si on parcourt la file à l'aide d'un **for**.

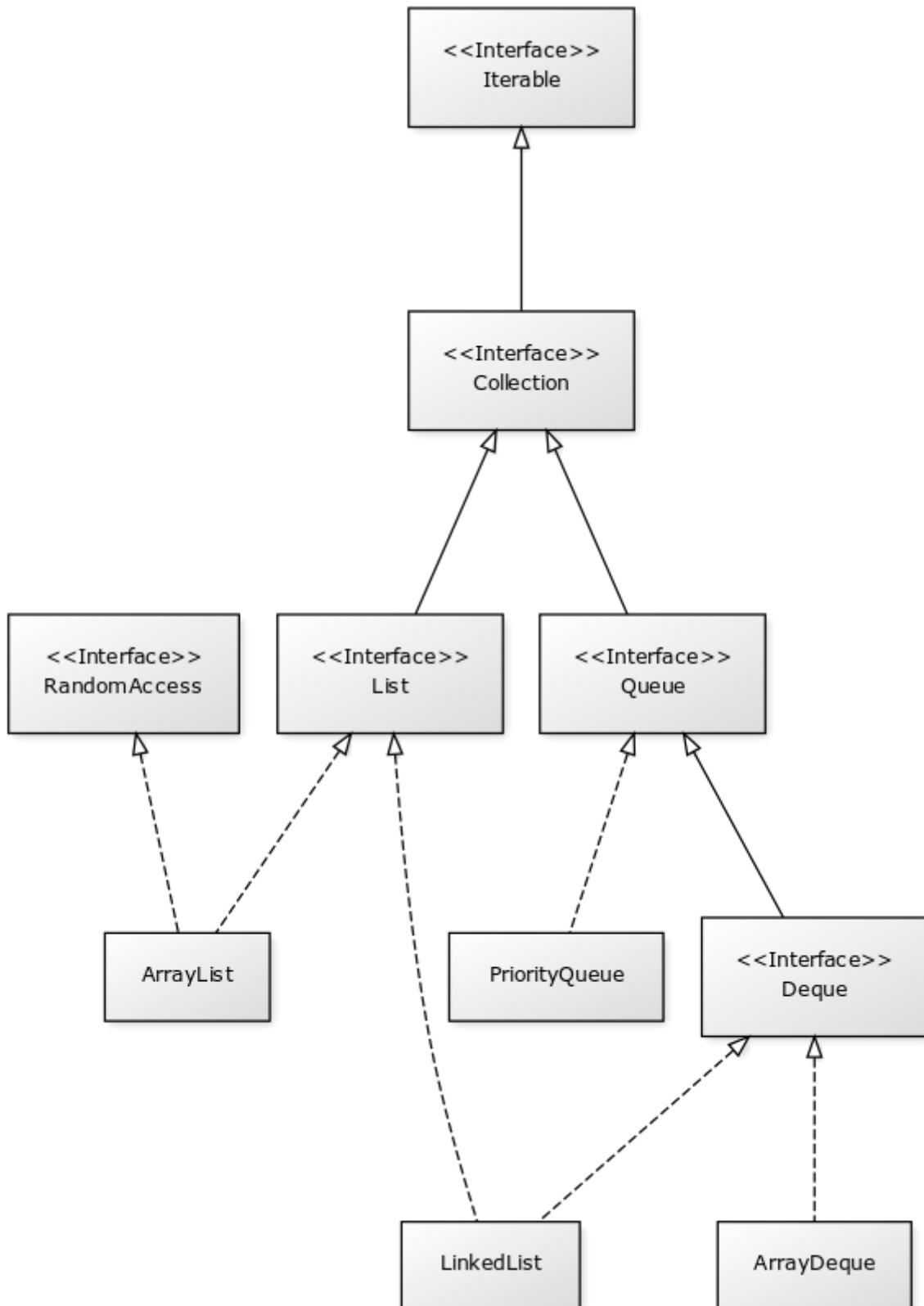
## Les classes Vector et Stack

La version 1.0 de Java a d'abord inclus les classes [java.util.Vector](#) et [java.util.Stack](#). La classe [Vector](#) permet de représenter une liste d'éléments comme la classe [ArrayList](#). La classe [Stack](#) qui hérite de [Vector](#) permet de représenter des piles de type LIFO (*Last In First Out*). Ces deux classes sont toujours présentes dans l'API pour des raisons de compatibilité ascendante mais il ne faut **surtout pas** s'en servir. En effet, ces classes utilisent des mécanismes de synchronisation internes dans le cas où elles sont utilisées pour des accès concurrents (programmation parallèle ou *multithread*). Or, non seulement ces mécanismes de synchronisation pénalisent les performances mais en plus, ils se révèlent largement inefficaces pour gérer les accès concurrents (il existe d'autres façons de faire en Java).

Les classes [ArrayList](#) et [ArrayDeque](#) se substituent très bien aux classes [Vector](#) et [Stack](#).

## Les interfaces pour les listes

Les listes du [Java Collections Framework](#) sont liées aux interfaces [Iterable](#), [Collection](#), [List](#), [Queue](#), [Deque](#) et [RandomAccess](#). Ci-dessous le diagramme de classes présentant les différents héritages et implémentations pour les quatre principales classes :



Comme proposé par le [principe de ségrégation d'interface](#), les variables, les paramètres et les attributs représentant des listes devraient avoir le type de l'interface adaptée. Par exemple, si vous utilisez une instance de [PriorityQueue](#), vous devriez y accéder à partir de l'interface [Queue](#) si vous n'effectuez que des opérations d'ajout, de suppression ou de consultation des éléments.

- [Iterable](#)

Cette interface permet d'obtenir un [Iterator](#) pour parcourir la liste. Elle permet également de parcourir la liste avec un **for** amélioré (*foreach*).

- [Collection](#)

Il s'agit de l'interface racine pour les collections. Elle déclare beaucoup de méthodes pour consulter ou modifier une collection. C'est également cette interface qui déclare la méthode [size](#) pour connaître la taille de la collection et les méthodes [toArray](#) pour obtenir un tableau à partir d'une collection. Par contre, cette interface ne permet pas d'accéder aléatoirement à un élément d'une collection (c'est-à-dire à partir de son index).

- [List](#)

Cette interface représente une collection ordonnée (une séquence) d'éléments. Elle déclare des méthodes pour accéder, pour modifier ou pour supprimer des éléments à partir de leur index (on parle aussi d'accès aléatoire). Cette interface déclare également la méthode [sort](#) pour permettre de trier la liste.

- [Queue](#)

Une file (*queue*) est une structure de données pour laquelle l'ordre des éléments est important mais les opérations de consultation, d'ajout et de suppression se font uniquement sur la tête de la file (le premier élément).

- [Deque](#)

[Deque](#) est la contraction de *double ended queue*. Cette interface représente une structure de données pour laquelle l'ordre des éléments est important mais les opération des consultation, d'ajout et de suppression se font soit sur le premier élément soit sur le dernier élément.

- [RandomAccess](#)

Il s'agit d'une [interface marqueur](#) qui signale que l'implémentation associée supporte les accès aléatoire en un temps constant. Par exemple, [ArrayList](#) implémente [RandomAccess](#) mais pas [LinkedList](#). Cette interface existe avant tout pour des raisons d'optimisation de parcours de liste.

## Les ensembles (set)

Les ensembles (*set*) sont des collections qui ne contiennent aucun doublon. Deux éléments *e1* et *e2* sont des doublons si :

```
e1.equals(e2) == true
```

ou si *e1* vaut **null** et *e2* vaut **null**. Pour contrôler l'unicité, le [Java Collections Framework](#) fournit trois implémentations : [TreeSet](#), [HashSet](#) et [LinkedHashSet](#).

Note

Il existe également un [EnumSet](#) qui représente un ensemble d'énumérations. Son implémentation est très compacte et très performante mais n'est utilisable que pour des [énumérations](#).

## La classe TreeSet

La classe [TreeSet](#) contrôle l'unicité de ces éléments en maintenant en interne une liste triée par ordre naturel des éléments. L'ordre peut être donné soit parce que les éléments implémentent l'interface [Comparable](#) soit parce qu'une implémentation de [Comparator](#) est passée en paramètre de constructeur au moment de la création de l'instance de [TreeSet](#).

```
Set<String> ensemble = new TreeSet<String>();

ensemble.add("élément");
ensemble.add("élément");
ensemble.add("élément");
ensemble.add("élément");

System.out.println(ensemble.size()); // 1

ensemble.remove("élément");

System.out.println(ensemble.isEmpty()); // true
```

La classe [TreeSet](#) a donc comme particularité de toujours conserver ses éléments triés.

## La classe HashSet

La classe [HashSet](#) utilise un code de hachage (hash code) pour contrôler l'unicité de ces éléments. Un code de hachage est une valeur associée à objet. Deux objets identiques doivent obligatoirement avoir le même code de hachage. Par contre deux objets distincts ont des codes de hachage qui peuvent être soit différents soit identiques. Un ensemble d'éléments différents mais qui ont néanmoins le même code de hachage forment un *bucket*. La classe [HashSet](#) maintient en interne un tableau associatif entre une valeur de hachage et un *bucket*. Lorsqu'un nouvel élément est ajouté au [HashSet](#), ce dernier calcule son code de hachage et vérifie si cette valeur a déjà été stockée. Si c'est le cas, alors les éléments du *bucket* associé sont parcourus un à un pour vérifier s'ils sont identiques ou non au nouvel élément.

Note

Le code de hachage d'un objet est donné par la méthode [Object.hashCode](#). L'implémentation par défaut de cette méthode ne convient généralement pas. En effet, elle retourne un code différent pour des objets différents en mémoire. Deux objets qui ont un état considéré comme identique mais qui existent de manière distincte en mémoire auront un code de hachage différent si on utilise l'implémentation par défaut. Beaucoup de classes redéfinissent donc cette méthode (c'est notamment le cas de la classe [String](#)).

```
Set<String> ensemble = new HashSet<String>();

ensemble.add("élément");
ensemble.add("élément");
ensemble.add("élément");
ensemble.add("élément");

System.out.println(ensemble.size()); // 1

ensemble.remove("élément");

System.out.println(ensemble.isEmpty()); // true
```

L'implémentation de la classe [HashSet](#) a des performances en temps très supérieures à [TreeSet](#) pour les opérations d'ajout et de suppression d'élément. Elle impose néanmoins que les éléments qu'elle contient génèrent correctement un code de hachage avec la méthode [hashCode](#). Contrairement à [TreeSet](#), elle ne garantit pas l'ordre dans lequel les éléments sont stockés et donc



l'ordre dans lequel ils peuvent être parcourus.

## La classe `LinkedHashSet`

La classe [LinkedHashSet](#), comme la classe [HashSet](#), utilise en interne un code de hachage mais elle garantit en plus que l'ordre de parcours des éléments sera le même que l'ordre d'insertion. Cette implémentation garantit également que si elle est créée à partir d'un autre [Set](#), l'ordre des éléments sera maintenu.

```
Set<String> ensemble = new LinkedHashSet<String>();

ensemble.add("premier élément");
ensemble.add("premier élément");
ensemble.add("premier élément");
ensemble.add("premier élément");

ensemble.add("deuxième élément");

ensemble.add("premier élément");

ensemble.add("troisième élément");

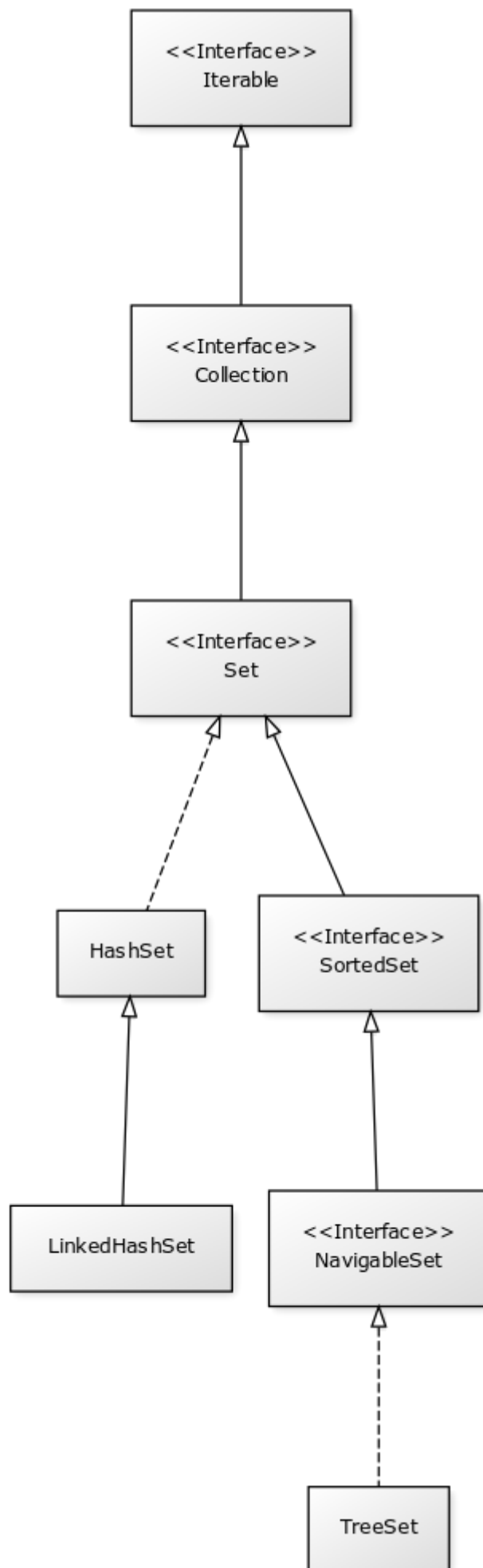
ensemble.add("premier élément");

// [premier élément, deuxième élément, troisième élément]
System.out.println(ensemble);
```

La classe [LinkedHashSet](#) a été créée pour réaliser un compromis entre la classe [HashSet](#) et la classe [TreeSet](#) afin d'avoir des performances proches de la première tout en offrant l'ordre de parcours pour ses éléments.

## Les interfaces pour les ensembles

Les ensembles du [Java Collections Framework](#) sont liés aux interfaces [Iterable](#), [Collection](#), [Set](#), [SortedSet](#) et [NavigableSet](#). Ci-dessous le diagramme de classes présentant les différents héritages et implémentations pour les trois principales classes :



Comme proposé par le [principe de ségrégation d'interface](#), les variables, les paramètres et les attributs représentant des ensemble devraient avoir le type de l'interface adaptée. Par exemple, si vous utilisez une instance de [HashSet](#), vous devriez y accéder à partir de l'interface [Set](#).

- [Iterable](#)

Cette interface permet d'obtenir un [Iterator](#) pour parcourir la liste. Elle permet également de parcourir l'ensemble avec un **for** amélioré (*foreach*).

- [Collection](#)

Il s'agit de l'interface racine pour les collections. Elle déclare beaucoup de méthodes pour consulter ou modifier une collection. C'est également cette interface qui déclare la méthode [size](#) pour connaître la taille de la collection et les méthodes [toArray](#) pour obtenir un tableau à partir d'une collection.

- [Set](#)

Il s'agit de l'interface qui définit la collection comme un ensemble, c'est-à-dire comme une liste d'éléments sans doublon.

- [SortedSet](#)

Cette interface indique que l'ensemble maintient en interne un ordre naturel de ses éléments. Elle offre notamment des méthodes pour accéder au premier et au dernier élément de l'ensemble.

- [NavigableSet](#)

Cette interface déclare des méthodes de navigation permettant par exemple de créer un sous ensemble à partir des éléments qui sont plus grands qu'un élément donné.

## Copie d'une collection dans un tableau

L'interface [Collection](#) commune aux listes et aux ensembles déclare deux méthodes qui permettent de copier les références des éléments d'une collection dans un tableau :

- [toArray\(\)](#)

Crée une nouvelle instance d'un tableau d'Object de la même taille que la collection et copie les références des éléments de la collection dans ce tableau.

- [toArray\(T\)](#)

Si le tableau passé en paramètre est suffisamment grand pour contenir les éléments de la collection, alors les références y sont copiées. Sinon un tableau du même type que celui passé en paramètre est créé et les références des éléments de la collection y sont copiées.

```
Collection<String> collection = new ArrayList<>();
collection.add("un");
collection.add("deux");
collection.add("trois");

Object[] tableauObjet = collection.toArray();

String[] tableauString = collection.toArray(new String[0]);

String[] autreTableauString = new String[collection.size()];
String[] memeTableauString = collection.toArray(autreTableauString);

// Tous les tableaux contiennent les mêmes éléments
```

```
System.out.println(Arrays.equals(tableauObjet, tableauString)); // true
System.out.println(Arrays.equals(tableauObjet, autreTableauString)); // true
System.out.println(Arrays.equals(tableauObjet, memeTableauString)); // true

// Les variables référencent le même tableau
System.out.println(autreTableauString == memeTableauString); // true
```

## Les tableaux associatifs (maps)

Un tableau associatif (parfois appelé dictionnaire) ou *map* permet d'associer une clé à une valeur. Un tableau associatif ne peut pas contenir de doublon de clés.

Les classes et les interfaces représentant des tableaux associatifs sont génériques et permettent de spécifier un type pour la clé et un type pour la valeur. Le [Java Collections Framework](#) fournit plusieurs implémentations de tableaux associatifs : [TreeMap](#), [HashMap](#), [LinkedHashMap](#).

Note

La classe [EnumMap](#) qui représente un tableau associatif dont les clés sont des énumérations. Son implémentation est très compacte et très performante mais n'est utilisable que pour des clés de type [énumération](#).

### La classe TreeMap

La classe [TreeMap](#) est basée sur l'implémentation d'un arbre bicolore pour déterminer si une clé existe ou non dans le tableau associatif. Elle dispose d'une bonne performance en temps pour les opérations d'accès, d'ajout et de suppression de la clé.

Cette classe contrôle l'unicité et l'accès à la clé en maintenant en interne une liste triée par ordre naturel des clés. L'ordre peut être donné soit parce que les éléments implémentent l'interface [Comparable](#) soit parce qu'une implémentation de [Comparator](#) est passée en paramètre de constructeur au moment de la création de l'instance de [TreeMap](#).

```
Map<String, Integer> tableauAssociatif = new TreeMap<>();
tableauAssociatif.put("un", 1);
tableauAssociatif.put("deux", 2);
tableauAssociatif.put("trois", 3);

System.out.println(tableauAssociatif.get("deux")); // 2

int resultat = 0;
for (String s : "un deux trois".split(" ")) {
    resultat += tableauAssociatif.get(s);
}

System.out.println(resultat); // 6

tableauAssociatif.remove("trois");
tableauAssociatif.put("deux", 1000);

System.out.println(tableauAssociatif.keySet()); // [deux, un]
System.out.println(tableauAssociatif.values()); // [1000, 1]
```

La classe [TreeMap](#) a donc comme particularité de conserver toujours ses clés triées.

# La classe HashMap

La classe [HashMap](#) utilise un code de hachage (hash code) pour contrôler l'unicité et l'accès aux clés. Un code de hachage est une valeur associée à un objet. Deux objets identiques doivent obligatoirement avoir le même code de hachage. Par contre deux objets distincts ont des codes de hachage qui peuvent être soit différents soit identiques. Un ensemble de clés différentes mais qui ont néanmoins le même code de hachage forment un *bucket*. La classe [HashMap](#) maintient en interne un tableau associatif entre une valeur de hachage et un *bucket*. Lorsqu'une nouvelle clé est ajoutée au [HashMap](#), ce dernier calcule son code de hachage et vérifie si ce code a déjà été stocké. Si c'est le cas, alors la valeur passée remplace l'ancienne valeur associée à cette clé. Sinon la nouvelle clé est ajoutée avec sa valeur.

Note

Le code de hachage d'un objet est donné par la méthode [Object.hashCode](#). L'implémentation par défaut de cette méthode ne convient généralement pas. En effet, elle retourne un code différent pour des objets différents en mémoire. Deux objets qui ont un état considéré comme identique mais qui existent de manière distincte en mémoire auront un code de hachage différent si on utilise l'implémentation par défaut. Beaucoup de classes redéfinissent donc cette méthode (c'est notamment le cas de la classe [String](#)).

```
Map<String, Integer> tableauAssociatif = new HashMap<>();
tableauAssociatif.put("un", 1);
tableauAssociatif.put("deux", 2);
tableauAssociatif.put("trois", 3);

System.out.println(tableauAssociatif.get("deux")); // 2

int resultat = 0;
for (String s : "un deux trois".split(" ")) {
    resultat += tableauAssociatif.get(s);
}

System.out.println(resultat); // 6

tableauAssociatif.remove("trois");
tableauAssociatif.put("deux", 1000);

System.out.println(tableauAssociatif.keySet()); // [deux, un]
System.out.println(tableauAssociatif.values()); // [1, 1000]
```

L'implémentation de la classe [HashSet](#) a des performances en temps supérieures à [TreeSet](#) pour les opérations d'ajout et d'accès. Elle impose néanmoins que les éléments qu'elle contient génèrent correctement un code de hachage avec la méthode [hashCode](#). Contrairement à la classe [TreeMap](#), elle ne garantit pas l'ordre dans lequel les clés sont stockées et donc l'ordre dans lequel elles peuvent être parcourues.

## La classe LinkedHashMap

La classe [LinkedHashMap](#), comme la classe [HashMap](#), utilise en interne un code de hachage mais elle garantit en plus que l'ordre de parcours des clés sera le même que l'ordre d'insertion. Cette implémentation garantit également que si elle est créée à partir d'une autre [Map](#), l'ordre des clés sera maintenu.

```
Map<String, Integer> tableauAssociatif = new LinkedHashMap<>();
tableauAssociatif.put("rouge", 0xff0000);
tableauAssociatif.put("vert", 0x00ff00);
tableauAssociatif.put("bleu", 0x0000ff);

// affichera : rouge puis vert puis bleu
for (String k: tableauAssociatif.keySet()) {
    System.out.println(k);
}
```

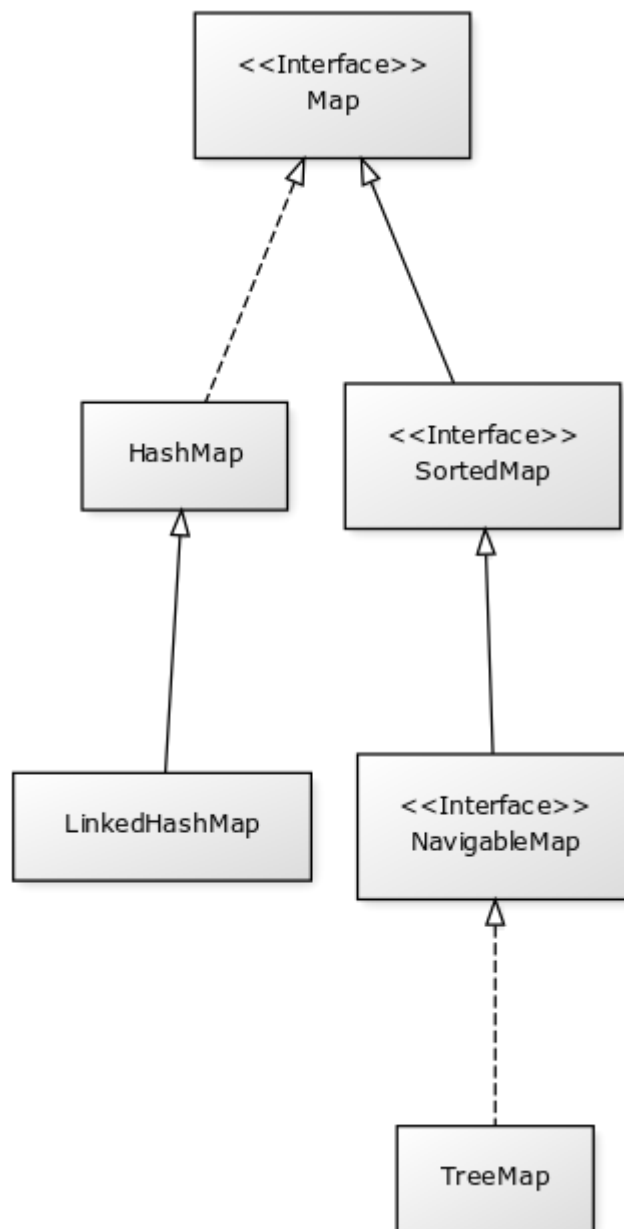
La classe [LinkedHashMap](#) a été créée pour réaliser un compromis entre la classe [HashMap](#) et la classe [TreeMap](#) afin d'avoir des performances proches de la première tout en offrant l'ordre de parcours pour ses clés.

## Les classes Dictionary et Hashtable

La version 1.0 de Java a d'abord inclus les classes [java.util.Dictionary](#) et [java.util.Hashtable](#) pour représenter des tableaux associatifs. Ces deux classes sont toujours présentes dans l'API pour des raisons de compatibilité ascendante mais il ne faut **surtout pas** s'en servir. En effet, ces classes utilisent des mécanismes de synchronisation internes dans le cas où elles sont utilisées pour des accès concurrents (programmation parallèle ou *multithread*). Or, non seulement ces mécanismes de synchronisation pénalisent les performances mais en plus, ils se révèlent largement inefficaces pour gérer les accès concurrents (il existe d'autres façons de faire en Java).

## Les interfaces pour les tableaux associatifs

Les tableaux associatifs du [Java Collections Framework](#) sont liés aux interfaces [Map](#), [SortedMap](#) et [NavigableMap](#). Ci-dessous le diagramme de classes présentant les différents héritages et implémentations pour les trois principales classes :



Comme proposé par le [principe de ségrégation d'interface](#), les variables, les paramètres et les attributs représentant des tableaux associatifs devraient avoir le type de l'interface adaptée. Par exemple, si vous utilisez une instance de [HashMap](#), vous devriez y accéder à partir de l'interface [Map](#).

- [Map](#)

Il s'agit de l'interface qui définit un tableau associatif. Elle déclare les méthodes d'ajout de clé et de valeur, de consultation et de suppression à partir de la clé. Il est également possible d'obtenir l'ensemble des clés ou la collection de toutes les valeurs. Cette interface permet également de connaître la taille du tableau associatif.

- [SortedMap](#)

Cette interface indique que le tableau associatif maintient en interne un ordre naturel de ses clés. Elle offre notamment des méthodes pour accéder à la première et à la dernière clé de l'ensemble.

- [NavigableMap](#)

Cette interface déclare des méthodes de navigation permettant par exemple de créer un sous ensemble à partir des clés qui sont plus grandes qu'une clé donnée.

# La classe outil Collections

La classe [java.util.Collections](#) est une classe outil qui contient de nombreuses méthodes pour les listes, les ensembles et les tableaux associatifs. Elle contient également des attributs de classes correspondant à une liste, un ensemble et un tableau associatif vides et immutables.

```
package dev.gayerie;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class TestCollections {

    public static void main(String[] args) {

        List<String> liste = new ArrayList<>();
        Collections.addAll(liste, "un", "deux", "trois", "quatre");

        // La chaîne a plus grande dans la liste : "un"
        String max = Collections.max(liste);
        System.out.println(max);

        // Inverse l'ordre de la liste
        Collections.reverse(liste);
        // [quatre, trois, deux, un]
        System.out.println(liste);

        // Trie la liste
        Collections.sort(liste);
        // [deux, quatre, trois, un]
        System.out.println(liste);

        // Recherche de l'index de la chaîne "deux" dans la liste triée : 0
        int index = Collections.binarySearch(liste, "deux");
        System.out.println(index);

        // Remplace tous les éléments par la même chaîne
        Collections.fill(liste, "même chaîne partout");
        // [même chaîne partout, même chaîne partout, même chaîne partout, même
chaîne partout]
        System.out.println(liste);

        // Enveloppe la liste dans une liste qui n'autorise plus à modifier son
contenu
        liste = Collections.unmodifiableList(liste);

        // On tente de modifier une liste qui n'est plus modifiable
        liste.add("Test"); // ERREUR à l'exécution : UnsupportedOperationException

    }

}
```



# Les entrées/sorties

En Java les entrées/sorties sont représentées par des objets de type [java.io.InputStream](#), [java.io.Reader](#), [java.io.OutputStream](#) et [java.io.Writer](#). Le package [java.io](#) définit un ensemble de classes qui vont pouvoir être utilisées conjointement avec ces quatre classes abstraites pour réaliser des traitements plus complexes.

## InputStream et classes concrètes

La classe [InputStream](#) est une [classe abstraite](#) qui représente un flux d'entrée de données binaires. Elle déclare des méthodes [read](#) qui permettent de lire des données octet par octet ou bien de les copier dans un tableau. Ces méthodes retournent le nombre de caractères lus ou -1 pour signaler la fin du flux. Il existe plusieurs classes qui en fournissent une implémentation concrète.

La classe [ByteArrayInputStream](#) permet d'ouvrir un flux de lecture binaire sur un tableau de **byte**.

```
package dev.gayerie.io;

import java.io.ByteArrayInputStream;

public class TestByteArrayInputStream {

    public static void main(String[] args) {
        byte[] tableau = "hello the world".getBytes();
        ByteArrayInputStream stream = new ByteArrayInputStream(tableau);

        int octet;
        while ((octet = stream.read()) != -1) {
            System.out.print((char) octet);
        }
    }
}
```

La classe [FileInputStream](#) permet d'ouvrir un flux de lecture binaire sur un fichier.

```
package dev.gayerie.io;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

public class TestFileInputStream {

    public static void main(String[] args) throws IOException {

        try (InputStream stream = new
        FileInputStream("/chemin/vers/mon/fichier.bin")) {
            byte[] buffer = new byte[1024];
            int nbRead;
            while ((nbRead = stream.read(buffer)) != -1) {
                // ...
            }
        }
    }
}
```

```

    }
}

}

}

```

Dans l'exemple ci-dessus, on utilise la méthode [InputStream.read](#) qui prend un tableau d'octets en paramètre. Cela est plus efficace que de lire le fichier octet par octet.

Prudence

À part s'ils représentent une zone mémoire, les flux de données sont généralement attachés à des ressources système (descripteurs de fichier ou de socket). Il est donc impératif de fermer ces flux en appelant leur méthode **close** lorsqu'ils ne sont plus nécessaires pour libérer les ressources système associées. Comme toutes les méthodes d'un flux sont susceptibles de jeter une [IOException](#), on utilise généralement le bloc **finally** pour appeler la méthode **close**.

```

InputStream stream = new FileInputStream("chemin/vers/mon/fichier.bin");
try {
    byte[] buffer = new byte[1024];
    int nbRead;
    while ((nbRead = stream.read(buffer)) != -1) {
        // ...
    }
} finally {
    stream.close();
}

```

Toutes les classes qui représentent des flux d'entrée ou de sortie implémentent l'interface [Closeable](#). Cela signifie qu'elles peuvent être utilisées avec la syntaxe *try-with-resources* et ainsi faciliter leur gestion en garantissant une fermeture automatique.

```

try (InputStream stream = new FileInputStream("/chemin/vers/mon/fichier.bin")) {
    byte[] buffer = new byte[1024];
    int nbRead;
    while ((nbRead = stream.read(buffer)) != -1) {
        // ...
    }
}

```

Les flux [System.in](#), [System.out](#) et [System.err](#) qui permettent de lire ou d'écrire sur la console sont des cas particuliers. Ils sont ouverts au lancement de l'application et seront automatiquement fermés à la fin. Il est néanmoins possible de fermer explicitement ces flux si on veut détacher l'application du *shell* à partir duquel elle a été lancée.

## OutputStream et classes concrètes

La classe [OutputStream](#) est une [classe abstraite](#) qui représente un flux de sortie de données binaires. Elle déclare des méthodes [write](#) qui permettent d'écrire des données octet par octet ou bien de les écrire depuis un tableau. La classe [OutputStream](#) fournit également la méthode [flush](#) pour forcer l'écriture de la zone tampon (s'il existe une zone tampon sinon un appel à cette méthode est sans effet).

Il existe plusieurs classes qui en fournissent une implémentation concrète.

La classe [ByteArrayOutputStream](#) permet d'ouvrir un flux d'écriture binaire en mémoire. Le contenu peut ensuite être récupéré sous la forme d'un tableau d'octets grâce à la méthode [toByteArray](#).

```
package dev.gayerie.io;

import java.io.ByteArrayOutputStream;
import java.util.Arrays;

public class TestByteArrayOutputStream {

    public static void main(String[] args) {
        ByteArrayOutputStream stream = new ByteArrayOutputStream();

        for (byte b : "Hello the world".getBytes()) {
            stream.write(b);
        }

        byte[] byteArray = stream.toByteArray();
        System.out.println(Arrays.toString(byteArray));
    }
}
```

La classe [FileOutputStream](#) permet d'ouvrir un flux d'écriture binaire sur un fichier.

```
package dev.gayerie.io;

import java.io.FileOutputStream;
import java.io.IOException;

public class TestFileOutputStream {

    public static void main(String[] args) throws IOException {

        try (FileOutputStream stream = new
FileOutputStream("chemin/vers/mon/fichierdesortie.bin")) {
            byte[] octets = "hello the world".getBytes();
            stream.write(octets);
        }

    }
}
```

Dans l'exemple ci-dessus, on utilise la méthode [OutputStream.write](#) qui prend un tableau d'octets en paramètre. Cela est plus efficace que d'écrire dans le fichier octet par octet.

Prudence

Comme cela a été signalé ci-dessus pour les [InputStream](#), les flux d'écriture qui ne correspondent pas à des zones mémoire (fichiers, sockets...) doivent impérativement être fermés lorsqu'ils ne sont plus utilisés pour libérer les ressources système associées.

# Flux orientés caractères

Le package [java.io](#) contient un ensemble de classes qui permettent de manipuler des flux caractères et donc du texte. Toutes les classes qui permettent d'écrire dans un flux de caractères héritent de la classe abstraite [Writer](#) et toutes les classes qui permettent de lire un flux de caractères héritent de la classe abstraite [Reader](#).

## Reader et classes concrètes

La classe [Reader](#) est une [classe abstraite](#) qui permet de lire des flux de caractères. Comme [InputStream](#), la classe [Reader](#) fournit des méthodes [read](#) mais qui acceptent en paramètre des caractères. Il existe plusieurs classes qui en fournissent une implémentation concrète.

La classe [StringReader](#) permet de parcourir une chaîne de caractères sous la forme d'un flux de caractères.

```
package dev.gayerie.io;

import java.io.IOException;
import java.io.Reader;
import java.io.StringReader;

public class TestStringReader {

    public static void main(String[] args) throws IOException {
        Reader reader = new StringReader("hello the world");

        int caractere;
        while ((caractere = reader.read()) != -1) {
            System.out.print((char) caractere);
        }
    }
}
```

Note

Il n'est pas nécessaire d'utiliser un [StringReader](#) pour parcourir une chaîne de caractères. Par contre, cette classe est très pratique si une partie d'un programme réalise des traitements en utilisant une instance de [Reader](#). Le principe de substitution peut s'appliquer en passant une instance de [StringReader](#).

La classe [FileReader](#) permet de lire le contenu d'un fichier texte.

```
package dev.gayerie.io;

import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;

public class TestFileReader {

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

```

try (Reader reader = new FileReader("/le/chemin/du/fichier.txt")) {
    char[] buffer = new char[1024];
    int nbRead;
    while ((nbRead = reader.read(buffer)) != -1) {
        // ...
    }
}

}

```

#### Note

La classe [FileReader](#) ne permet pas de positionner l'encodage de caractères (*charset*) utilisé dans le fichier. Elle utilise l'encodage par défaut de la JVM qui est dépendant du système. Dans la pratique l'usage de cette classe est donc assez limité.

## Writer et classes concrètes

La classe [Writer](#) est une [classe abstraite](#) qui permet d'écrire des flux de caractères. Comme [OutputStream](#), la classe [Writer](#) fournit des méthodes [write](#) mais qui acceptent en paramètre des caractères. Elle fournit également des méthodes [append](#) qui réalisent la même type d'opérations et qui retournent l'instance du [Writer](#) afin de pouvoir chaîner les appels. Il existe plusieurs classes qui en fournissent une implémentation concrète.

La classe [StringWriter](#) permet d'écrire dans un flux caractères pour ensuite produire une chaîne de caractères.

```

package dev.gayerie.io;

import java.io.IOException;
import java.io.StringWriter;

public class TestStringwriter {

    public static void main(String[] args) throws IOException {
        StringWriter writer = new StringWriter();

        writer.append("Hello")
            .append(' ')
            .append("the")
            .append(' ')
            .append("world");

        String resultat = writer.toString();

        System.out.println(resultat);
    }

}

```

La classe [FileWriter](#) permet d'écrire un flux de caractères dans un fichier.

```

package dev.gayerie.io;

import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;

public class TestFileWriter {

    public static void main(String[] args) throws IOException {

        try (Writer writer = new FileWriter("/chemin/vers/mon/fichier.txt", true)) {
            writer.append("Hello world!\n");
        }

    }

}

```

Note

Le booléen passé en second paramètre du constructeur de [FileWriter](#) permet de spécifier si le fichier doit être ouvert en ajout (*append*).

Note

La classe [FileWriter](#) ne permet pas de positionner l'encodage de caractères (*charset*) utilisé pour écrire dans le fichier. Elle utilise l'encodage par défaut de la JVM qui est dépendant du système. Dans la pratique l'usage de cette classe est donc assez limité.

## Les décorateurs de flux

Le package [java.io](#) fournit un ensemble de classes qui agissent comme des [décorateurs](#) pour des instances de type [InputStream](#), [Reader](#), [OutputStream](#) ou [Writer](#). Ces [décorateurs](#) permettent d'ajouter des fonctionnalités tout en présentant les mêmes méthodes. Il est donc très simple d'utiliser ces décorateurs dans du code initialement implémenté pour manipuler des instances des types décorés.

Les classes [BufferedInputStream](#), [BufferedReader](#), [BufferedOutputStream](#) et [BufferedWriter](#) permettent de créer un décorateur qui gère une zone tampon dont il est possible d'indiquer la taille à la construction de l'objet. Ces classes sont très utiles lorsque l'on veut lire ou écrire des données sur un disque ou sur un réseau afin de limiter les accès système et améliorer les performances.

```

package dev.gayerie.io;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;

public class TestFileWriter {

    public static void main(String[] args) throws IOException {

```

```

    try (Writer writer = new BufferedWriter(new FileWriter("monfichier.txt",
true), 1024)) {
        writer.append("Hello world!\n");
    }

}

}

```

Dans l'exemple ci-dessus, on crée un [BufferedWriter](#) avec une zone tampon de 1 Ko.

La classe [LineNumberReader](#) permet quant à elle, de compter les lignes lors de la lecture d'un flux caractères. Elle fournit également la méthode [readLine](#) pour lire une ligne complète.

```

package dev.gayerie.io;

import java.io.IOException;
import java.io.LineNumberReader;
import java.io.StringReader;

public class TestStringReader {

    public static void main(String[] args) throws IOException {
        StringReader stringReader = new StringReader("hello the world\nhello the
world");

        LineNumberReader reader = new LineNumberReader(stringReader);

        String line;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }

        System.out.println("Nombre de lignes lues : " + reader.getLineNumber());
    }

}

```

Les classes [InputStreamReader](#) et [OutputStreamWriter](#) permettent de manipuler un flux binaire sous la forme d'un flux caractères. La classe [InputStreamReader](#) hérite de [Reader](#) et prend comme paramètre de constructeur une instance de [InputStream](#). La classe [OutputStreamWriter](#) hérite de [Writer](#) et prend comme paramètre de constructeur une instance de [OutputStream](#). Ces classes sont particulièrement utiles car elles permettent de préciser l'encodage des caractères (*charset*) qui doit être utilisé pour passer d'un flux binaire au flux caractères et vice-versa.

```

package dev.gayerie.io;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.Reader;

public class TestFileReader {

```

```

public static void main(String[] args) throws IOException {

    String fichier = "/le/chemin/du/fichier.txt";
    try (Reader reader = new InputStreamReader(new FileInputStream(fichier),
"UTF-8")) {
        char[] buffer = new char[1024];
        int nbRead;
        while ((nbRead = reader.read(buffer)) != -1) {
            // ...
        }
    }

}

}

```

Dans l'exemple ci-dessus, le fichier est ouvert grâce à une instance de [FileInputStream](#) qui est passée à une instance de [InputStreamReader](#) qui lit les caractères au format UTF-8.

Il est possible de créer très facilement des chaînes de décorateurs.

```

package dev.gayerie.io;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.LineNumberReader;
import java.io.Reader;

public class TestFileReader {

    public static void main(String[] args) throws IOException {

        String fichier = "/le/chemin/du/fichier.txt";
        Reader inputStreamReader = new InputStreamReader(new
FileInputStream(fichier), "UTF-8");
        try (LineNumberReader reader = new LineNumberReader(inputStreamReader)) {
            String ligne;
            while ((ligne = reader.readLine()) != null) {
                // ...
            }
        }

    }

}

```

#### Note

Dans l'exemple ci-dessus, on utilise la syntaxe try-with-resources pour appeler automatiquement la méthode [close](#) à la fin du bloc **try**. Les décorateurs de flux implémentent la méthode [close](#) de manière à appeler la méthode [close](#) de l'objet qu'il décore. Ainsi quand on crée une chaîne de flux avec des décorateurs, un appel à la méthode [close](#) du décorateur le plus englobant appelle automatiquement toutes les méthodes [close](#) de la chaîne de flux.



Les objets statiques [System.in](#), [System.out](#) et [System.err](#) qui représentent respectivement le flux d'entrée de la console, le flux de sortie de la console et le flux de sortie d'erreur de la console sont des instances de [InputStream](#) ou de [PrintStream](#). [PrintStream](#) est un décorateur qui offre notamment les méthodes [print](#), [println](#) et [printf](#).

Note

Pour manipuler les flux de la console, il est également possible de récupérer une instance de [Console](#) en appelant la méthode [System.console\(\)](#).

## La classe Scanner

La classe [java.util.Scanner](#) agit comme un décorateur pour différents types d'instance qui représentent une entrée. Elle permet de réaliser des opérations de lecture et de validation de données plus complexes que les classes du packages [java.io](#).

```
package dev.gayerie.io;

import java.io.IOException;
import java.util.Scanner;

public class TestScanner {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Saisissez une chaîne de caractères : ");
        String chaine = scanner.nextLine();

        System.out.print("Saisissez un nombre : ");
        int nombre = scanner.nextInt();

        System.out.print("Saisissez les 8 caractères de votre identifiant : ");
        String identifiant = scanner.next("{8}");

        System.out.println("Vous avez saisi :");
        System.out.println(chaine);
        System.out.println(nombre);
        System.out.println(identifiant);
    }
}
```

On peut compléter l'implémentation précédente en effectuant une validation sur les données saisies par l'utilisateur :

```
package dev.gayerie.io;

import java.util.Scanner;

public class TestScanner {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
```

```

System.out.print("Saisissez une chaîne de caractères : ");
String chaine = scanner.nextLine();

Integer nombre = null;
do {
    System.out.print("Saisissez un nombre : ");
    if (!scanner.hasNextInt()) {
        scanner.next();
        System.err.println("Ceci n'est pas un nombre valide");
        continue;
    }
    nombre = scanner.nextInt();
} while (nombre == null);

String identifiant = null;
do {
    System.out.print("Saisissez les 8 caractères de votre identifiant : ");
    // On utilise une expression régulière pour vérifier le prochain token
    if (!scanner.hasNext(".{8}")) {
        scanner.next();
        System.err.println("Ceci n'est pas un identifiant valide");
        continue;
    }
    identifiant = scanner.next();
} while (identifiant == null);

System.out.println("Vous avez saisi :");
System.out.println(chaine);
System.out.println(nombre);
System.out.println(identifiant);
}
}

```

## Fichiers et chemins

En plus des flux de type fichier, le package [java.io](#) fournit la classe [File](#) qui représente un fichier. À travers, cette classe, il est possible de savoir si le fichier existe, s'il s'agit d'un répertoire... On peut également créer le fichier ou le supprimer.

```

package dev.gayerie.io;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class TestFile {

    public static void main(String[] args) throws IOException {
        File fichier = new File("unfichier.txt");

        if (!fichier.exists()) {
            fichier.createNewFile();
        }
    }
}

```

```

    if (fichier.canWrite()) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(fichier))) {
            writer.write("Hello world!");
        }
    }

    fichier.delete();
}
}

```

Pour représenter un chemin d'accès à un fichier, on peut utiliser une URL avec le schéma *file* :

```
file:///home/david/monfichier.txt
```

ou bien une chaîne de caractère représentant directement le chemin. L'inconvénient de cette dernière méthode est qu'elle n'est pas portable suivant les différents systèmes de fichiers et les différents systèmes d'exploitation. En Java, on utilise l'interface [Path](#) pour représenter un chemin de fichier de manière générique. Les classes [Paths](#) et [FileSystems](#) servent à construire des instances de type [Path](#). La classe [FileSystems](#) fournit également des méthodes pour obtenir des informations à propos du ou des systèmes de fichiers présents sur la machine. On peut accéder à une instance de [FileSystems](#) grâce à la méthode [FileSystems.getDefault\(\)](#).

```

package dev.gayerie.io;

import java.io.File;
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Path;
import java.nio.file.Paths;

public class TestPath {

    public static void main(String[] args) throws IOException {
        Path cheminFichier = Paths.get("home", "david", "fichier.txt");

        System.out.println(cheminFichier); // home/david/fichier.txt
        System.out.println(cheminFichier.getNameCount()); // 3
        System.out.println(cheminFichier.getParent()); // home/david
        System.out.println(cheminFichier.getFileName()); // fichier.txt

        cheminFichier = FileSystems.getDefault().getPath("home", "david",
"fichier.txt");
        File fichier = cheminFichier.toFile();

        // maintenant on peut utiliser le fichier
    }
}

```

Pour les opérations les plus courantes sur les fichiers, la classe outil [Files](#) fournit un ensemble de méthodes statiques qui permettent de créer, de consulter, de modifier ou de supprimer des fichiers et des répertoires en utilisant un minimum d'appel.

```

package dev.gayerie.io;

import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.util.List;

public class TestFiles {

    public static void main(String[] args) throws IOException {
        Path fichier1 = Paths.get("fichier.txt");

        // création du fichier
        fichier1 = Files.createFile(fichier1);

        System.out.println("Taille du fichier : " + Files.size(fichier1));

        try (BufferedWriter writer = Files.newBufferedWriter(fichier1,
            StandardOpenOption.WRITE)) {
            writer.append("Hello !\n");
            writer.append("Hello !\n");
            writer.append("Hello !\n");
        }

        System.out.println("Taille du fichier : " + Files.size(fichier1));

        // Copie vers un nouveau fichier
        Path fichier2 = Paths.get("fichier2.txt");
        Files.copy(fichier1, fichier2);

        // Lecture de l'intégralité du fichier
        List<String> lignes = Files.readAllLines(fichier2);

        // Suppression des fichiers créés
        Files.deleteIfExists(fichier1);
        Files.deleteIfExists(fichier2);

        System.out.println(lignes);
    }
}

```

#### Note

La classe [Files](#) se révèle très pratique d'utilisation notamment pour lire l'intégralité d'un fichier. Elle ne rend pas pour autant obsolète l'utilisation de [Reader](#) ou de [OutputStream](#). En effet, travailler à partir d'un flux peut avoir un impact important sur l'empreinte mémoire d'une application. Si une application doit parcourir un fichier pour trouver une information précise alors, si le fichier peut être de taille importante, l'utilisation de flux sera plus optimale car l'empreinte mémoire d'un flux est généralement celle de la taille de la zone tampon allouée pour la lecture ou l'écriture.

# Accès au réseau

La classe [URL](#), comme son nom l'indique, représente une URL. Elle déclare la méthode [openConnection](#) qui retourne une instance de [URLConnection](#). Une instance de [URLConnection](#) ouvre une connexion distante avec le serveur et permet de récupérer des informations du serveur distant. Elle permet surtout d'obtenir une instance de [OutputStream](#) si on désire envoyer des informations au serveur et une instance de [InputStream](#) si on désire récupérer les informations retournées par le serveur.

```
package dev.gayerie.io;

import java.io.IOException;
import java.io.InputStreamReader;
import java.io.LineNumberReader;
import java.io.Reader;
import java.net.URL;
import java.net.URLConnection;
import java.util.Objects;

public class HttpClient {

    public static void main(String[] args) throws IOException {
        URL url = new URL("https://www.ietf.org/rfc/rfc1738.txt");
        URLConnection connection = url.openConnection();

        String encodage = Objects.toString(connection.getContentEncoding(), "ISO-8859-1");
        Reader reader = new InputStreamReader(connection.getInputStream(), encodage);

        try (LineNumberReader lineNumberReader = new LineNumberReader(reader)) {
            String line;
            while ((line = lineNumberReader.readLine()) != null) {
                System.out.println(line);
            }

            System.out.println("Ce fichier contient " + lineNumberReader.getLineNumber()
+ " lignes.");
        }
    }
}
```

Le programme ci-dessus récupère, affiche sur la sortie standard et donne le nombre de lignes du document accessible à l'adresse <https://www.ietf.org/rfc/rfc1738.txt> (il s'agit du document de l'IETF qui décrit ce qu'est une URL).

L'API d'entrée/sortie de Java fournit une bonne abstraction. Généralement, une méthode qui manipule des flux fonctionnera pour des fichiers, des flux mémoire et des flux réseaux.

# La sérialisation d'objets

Les classes [ObjectOutputStream](#) et [ObjectInputStream](#) permettent de réaliser la sérialisation/désérialisation d'objets : un objet (et tous les objets qu'il référence) peut être écrit dans un flux ou lu depuis un flux. Cela peut permettre de sauvegarder dans un fichier un état de l'application ou bien d'échanger des données entre deux programmes Java à travers un réseau. La sérialisation d'objets a des limites :

- Seul l'état des objets est écrit ou lu, cela signifie que les fichiers *class* ne font pas partie de la sérialisation et doivent être disponibles pour la JVM au moment de la lecture (opération de désérialisation réalisée avec la classe [ObjectInputStream](#)).
- Le format des données sérialisées est propre à Java, ce mécanisme n'est donc pas adapté pour échanger des informations avec des applications qui ne seraient pas écrites en Java.
- Les données sérialisées sont très dépendantes de la structure des classes. Si des modifications sont apportées à ces dernières, une grappe d'objets préalablement sérialisée dans un fichier ne sera sans doute plus lisible.

Pour qu'un objet puisse être sérialisé, il faut que sa classe implémente [l'interface marqueur Serializable](#). Si un objet référence d'autres objets dans ses attributs alors il faut également que les classes de ces objets implémentent l'interface [Serializable](#). Beaucoup de classes de l'API standard de Java implémentent l'interface [Serializable](#), à commencer par la classe [String](#).

Note

Tenter de sérialiser un objet dont la classe n'implémente pas [Serializable](#) produit une exception de type [java.io.NotSerializableException](#).

Prenons comme exemple une classe *Personne* qui contient la liste de ses enfants (eux-mêmes de type *Personne*). Cette classe implémente l'interface [Serializable](#) :

```
package dev.gayerie;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Personne implements Serializable {

    private String prenom;
    private String nom;
    private List<Personne> enfants = new ArrayList<>();

    public Personne(String prenom, String nom) {
        this.prenom = prenom;
        this.nom = nom;
    }

    public String getNom() {
        return nom;
    }

    public String getPrenom() {
        return prenom;
    }
}
```

```

    }

    public void ajouterEnfants(Personne... enfants) {
        Collections.addAll(this.enfants, enfants);
    }

    public List<Personne> getEnfants() {
        return enfants;
    }

    @Override
    public String toString() {
        return this.prenom + " " + this.nom;
    }
}

```

Le code ci-dessous sérialise les données dans le fichier *arbre\_genialogique.bin*

```

package dev.gayerie.io;

import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.nio.file.Files;
import java.nio.file.Paths;

import dev.gayerie.Personne;

public class TestSerialisation {

    public static void main(String[] args) throws IOException {

        Personne personne = new Personne("Donald", "Duck");
        personne.ajouterEnfants(new Personne("Riri", "Duck"),
                                new Personne("Fifi", "Duck"),
                                new Personne("Loulou", "Duck"));

        OutputStream outputStream =
Files.newOutputStream(Paths.get("arbre_genialogique.bin"));
        try(ObjectOutputStream objectStream = new ObjectOutputStream(outputStream);)
        {
            objectStream.writeObject(personne);
        }
    }
}

```

Un autre code qui a accès à la même classe *Personne* peut ensuite lire le fichier *arbre\_genialogique.bin* pour retrouver les objets dans l'état attendu.

```

package dev.gayerie.io;

import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;

```

```

import java.nio.file.Files;
import java.nio.file.Paths;

import dev.gayerie.Personne;

public class TestDeserialisation {

    public static void main(String[] args) throws IOException,
    ClassNotFoundException {

        InputStream outputStream =
        Files.newInputStream(Paths.get("arbre_genalogique.bin"));
        try(ObjectInputStream objectStream = new ObjectInputStream(outputStream);) {
            Personne personne = (Personne) objectStream.readObject();

            System.out.println(personne);
            for (Personne enfant : personne.getEnfants()) {
                System.out.println(enfant);
            }
        }
    }
}

```

L'exécution du programme ci-dessus affichera :

```

Donald Duck
Riri Duck
Fifi Duck
Loulou Duck

```

## Donnée transient

Parfois une classe contient des informations que l'on ne souhaite pas sérialiser. Cela peut être dû à des limitations techniques (par exemple la classe associée n'implémente pas l'interface [Serializable](#)). Mais il peut aussi s'agir de données sensibles ou volatiles qui n'ont pas à être sérialisées. Pour que les processus de sérialisation/désérialisation ignorent ces attributs, il faut leur ajouter le mot-clé **transient**.

Pour la classe *Personne*, si on veut exclure la liste des enfants de la sérialisation/désérialisation, on peut modifier les attributs comme suit :

```

package dev.gayerie;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Personne implements Serializable {

    private String prenom;
    private String nom;
    private transient List<Personne> enfants = new ArrayList<>();
}

```



```
public Personne(String prenom, String nom) {  
    this.prenom = prenom;  
    this.nom = nom;  
}  
  
// ...  
}
```

Si nous exécutons à nouveau les programmes de sérialisation et de désérialisation du paragraphe précédent, la sortie standard affichera alors :

```
Donald Duck
```

Car l'état de la liste des enfants ne sera plus écrit dans le fichier *arbre\_genalogique.bin*.

## Identifiant de version de sérialisation

La principale difficulté dans la mise en pratique des mécanismes de sérialisation/désérialisation provient de leur extrême dépendance au format des classes.

Si la sérialisation est utilisée pour sauvegarder dans un fichier l'état des objets entre deux exécutions, alors il n'est pas possible de modifier significativement puis de recompiler les classes sérialisables (sinon l'opération de désérialisation échouera avec une erreur [InvalidClassException](#)). Si la sérialisation est utilisée pour échanger des informations entre deux applications sur un réseau, alors les deux applications doivent disposer dans leur *classpath* des mêmes définitions de classes.

En fait les classes qui implémentent l'interface [Serializable](#) possèdent un numéro de version interne qui change à la compilation si des modifications substantielles ont été apportées (ajout ou suppression d'attributs ou de méthodes par exemple). Lorsqu'un objet est sérialisé, le numéro de version de sa classe est également sérialisé. Ainsi, lors de la désérialisation, il est facile de comparer ce numéro avec celui de la classe disponible. Si ces numéros ne correspondent pas, alors le processus de désérialisation échoue en considérant que la classe disponible n'est pas compatible avec la classe qui a été utilisée pour créer l'objet sérialisé.

Si on ne souhaite pas utiliser ce mécanisme implicite de version, il est possible de spécifier un numéro de version de sérialisation pour ses classes. À charge du développeur de changer ce numéro lorsque les modifications de la classe sont trop importantes pour ne plus garantir la compatibilité ascendante avec des versions antérieures de cette classe. Le numéro de version est une constante de classe de type **long** qui doit s'appeler *serialVersionUID*.

```

package dev.gayerie;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Personne implements Serializable {

    private static final long serialVersionUID = 1775245980933452908L;

    // ...
}

```

#### Note

Eclipse produit un avertissement si une classe qui implémente [Serializable](#) ne déclare pas une constante *serialVersionUID*.

#### Astuce

Pour contourner le problème de dépendance entre le format de sérialisation et la déclaration de la classe, il est possible d'implémenter soi-même l'écriture et la lecture des données. Pour cela, il faut déclarer deux méthodes privées dans la classe : *writeObject* et *readObject*. Ces méthodes seront appelées (même si elles sont privées) en lieu et place de l'algorithme par défaut de sérialisation/désérialisation.

```

package dev.gayerie;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Personne implements Serializable {

    private static final long serialVersionUID = 1775245980933452908L;

    private String prenom;
    private String nom;
    private List<Personne> enfants = new ArrayList<>();

    private void writeObject(ObjectOutputStream s) throws IOException {
        // on ne sérialise que le prénom et le nom
        s.writeObject(prenom);
        s.writeObject(nom);
    }

    private void readObject(ObjectInputStream s) throws ClassNotFoundException,
    IOException {
        // on lit les données dans le même ordre qu'elles ont été écrites
        this.prenom = (String) s.readObject();
    }
}

```

```

        this.nom = (String) s.readObject();
        this.enfants = new ArrayList<>();
    }

    // ...

}

```

## Les dates

Les dates et le temps sont représentés en Java par des classes. Cependant, au fil des versions de l'API standard, de nouvelles classes ont été proposées pour représenter les dates et le temps. Pour des raisons de compatibilité ascendante, les anciennes classes sont demeurées même si une partie de leurs méthodes ont été dépréciées. Ainsi, il existe plusieurs façons de représenter les dates et le temps en Java.

### Date : la classe historique

La classe [java.util.Date](#) est la première classe à être apparue pour représenter une date. Elle comporte de nombreuses limitations :

- Il n'est pas possible de représenter des dates antérieures à 1900
- Elle ne supporte pas les fuseaux horaires
- Elle ne supporte que les dates pour le calendrier grégorien
- Elle ne permet pas d'effectuer des opérations (ajout d'un jour, d'une année...)

La quasi totalité des constructeurs et des méthodes de cette classe ont été dépréciés. Cela signifie qu'il ne faut pas les utiliser. Pourtant, la classe [Date](#) reste une classe largement utilisée en Java. Par exemple, la classe [Date](#) est la classe mère des classes [java.sql.Date](#), [java.sql.Time](#) et [java.sql.Timestamp](#) qui servent à représenter les types du même nom en SQL. Beaucoup de bibliothèques tierces utilisent directement ou indirectement la classe [Date](#).

```

package dev.gayerie;

import java.time.Instant;
import java.util.Date;

public class TestDate {

    public static void main(String[] args) {
        Date aujourd'hui = new Date(); // crée une date au jour et à l'heure
d'aujourd'hui
        Date dateAvant = new Date(0); // crée une date au 1 janvier 1970 00:00:00.000

        System.out.println(aujourd'hui.after(dateAvant)); // true
        System.out.println(dateAvant.before(aujourd'hui)); // true

        System.out.println(aujourd'hui.equals(dateAvant)); // false
        System.out.println(aujourd'hui.compareTo(dateAvant)); // 1

        Instant instant = aujourd'hui.toInstant();
    }
}

```

```
}
```

Actuellement, les méthodes permettant de comparer des instances de [Date](#) ne sont pas dépréciées. Pour construire une instance de [Date](#), on peut utiliser le constructeur sans paramètre pour créer une date au jour et à l'heure d'aujourd'hui. On peut également passer un nombre de type **long** représentant le nombre de millisecondes depuis le 1er janvier 1970 à 00:00:00.000 (*l'epoch*). Il est également possible de modifier une instance de [Date](#) avec la méthode [Date.setTime](#) en fournissant le nombre de millisecondes depuis le 1er janvier 1970 à 00:00:00.000.

Astuce

Pour connaître le nombre de millisecondes écoulées depuis le 1er janvier 1970 à 00:00:00.000, il faut appeler la méthode [System.currentTimeMillis](#).

```
long nbMilliSecondes = System.currentTimeMillis();
```

Note

Depuis l'introduction en Java 8 de la nouvelle API des dates, il est possible de convertir une instance de [Date](#) en une instance de [Instant](#) avec la méthode [Date.toInstant](#).

## Calendar

Depuis Java 1.1, la classe [java.util.Calendar](#) a été proposée pour remplacer la classe [java.util.Date](#). La classe [Calendar](#) pallie les nombreux désavantages de la classe [Date](#) :

- Il est possible de représenter toutes les dates (y compris avant notre ère)
- La classe [Calendar](#) supporte les fuseaux horaires (*timezone*)
- La classe [Calendar](#) est une classe abstraite pour laquelle il est possible de fournir des classes concrètes pour gérer différents calendriers (le JDK ne propose que la classe [GregorianCalendar](#) comme implémentation concrète pour le calendrier grégorien).

```
package dev.gayerie;

import java.util.Calendar;
import java.util.Locale;
import java.util.TimeZone;

public class TestCalendar {

    public static void main(String[] args) {
        // Date et heure d'aujourd'hui en utilisant le fuseau horaire du système
        Calendar date = Calendar.getInstance();
        // Date et heure d'aujourd'hui en utilisant le fuseau horaire de la France
        Calendar dateFrance = Calendar.getInstance(Locale.FRANCE);
        // Date et heure d'aujourd'hui en utilisant le fuseau horaire GMT
        Calendar dateGmt = Calendar.getInstance(TimeZone.getTimeZone("GMT"));

        // On positionne la date au 8 juin 2005 à 12:00:00
        date.set(2005, 6, 8, 12, 0, 0);

        System.out.println(date.toInstant());

        // On ajoute 24 heures à la date et on passe au jour suivant
```

```

        date.add(Calendar.HOUR, 24);
        // On décale la date de 12 mois sans passer à l'année suivante
        date.roll(Calendar.MONTH, 12);
        System.out.println(date.toInstant()); // 9 juin 2005 à 12:00:00

    }

}

```

Comme pour les instances de [Date](#), il est possible de comparer les instances de [Calendar](#) entre elles. Il est également possible de convertir une instance de [Calendar](#) en [Date](#) (mais alors on perd l'information du fuseau horaire puisque la classe [Date](#) ne contient pas cette information) grâce à la méthode [Calendar.getTime](#). Enfin, on utilise la méthode [Calendar.toInstant](#) pour convertir une instance de [Calendar](#) en une instance de [Instant](#).

Même si la classe [Calendar](#) est beaucoup plus complète que la classe [Date](#), son utilisation est restée limitée car elle est également plus difficile à manipuler. Son API la rend assez fastidieuse d'utilisation. Elle ne permet pas de représenter simplement la notion du durée. Et surtout, comme il s'agit d'une classe abstraite, il n'est pas possible construire une instance avec l'opérateur **new**. Il faut systématiquement utiliser une des méthodes de classes [Calendar.getInstance](#).

## L'API Date/Time

Depuis Java 8, une nouvelle API a été introduite pour représenter les dates, le temps et la durée. Toutes ces classes ont été regroupées dans la package [java.time](#).

## Les Dates

Les classes [LocalDate](#), [LocalTime](#) et [LocalDateTime](#) permettent de représenter respectivement une date, une heure, une date et une heure.

```

package dev.gayerie;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;
import java.time.temporal.ChronoUnit;

public class TestTime {

    public static void main(String[] args) {
        LocalDate date = LocalDate.of(2005, Month.JUNE, 5); // 05/06/2005
        date = date.plus(1, ChronoUnit.DAYS); // 06/06/2005
        LocalDateTime dateTime = date.atTime(12, 00); // 06/06/2005 12:00:00
        LocalTime time = dateTime.toLocalTime(); // 12:00:00

        time = time.minusHours(2); // 10:00:00
    }

}

```

On peut facilement passer d'un type à une autre. Par exemple la méthode [LocalDate.atTime](#) permet d'ajouter une heure à une date, créant ainsi une instance de [LocalDateTime](#). Toutes les instances de ces classes sont immutables.

Si on veut avoir l'information de la date ou de l'heure d'aujourd'hui, on peut créer une instance grâce à la méthode *now*.

```
LocalDate dateAujourd'hui = LocalDate.now();
LocalTime heureMaintenant = LocalTime.now();
LocalDateTime dateHeureMaintenant = LocalDateTime.now();
```

Une instance de ces classes ne contient pas d'information de fuseau horaire. On peut néanmoins passer en paramètre des méthodes *now* un [ZoneId](#) pour indiquer le fuseau horaire pour lequel on désire la date et/ou l'heure actuelle.

```
LocalDate dateAujourd'hui = LocalDate.now(ZoneId.of("GMT"));
LocalTime heureMaintenant = LocalTime.now(ZoneId.of("Europe/Paris"));
LocalDateTime dateHeureMaintenant =
    LocalDateTime.now(ZoneId.of("America/New_York"));
```

Note

Si vous avez besoin de représenter des dates avec le fuseau horaire, alors il faut utiliser la classe [ZonedDateTime](#).

Les classe [Year](#) et [YearMonth](#) permettent de manipuler les dates et d'obtenir des informations intéressantes à partir de l'année ou du mois et de l'année.

```
package dev.gayerie;

import java.time.LocalDate;
import java.time.Month;
import java.time.Year;
import java.time.YearMonth;

public class TestYear {

    public static void main(String[] args) {
        Year year = Year.of(2004);

        // année bissextile ?
        boolean isLeap = year.isLeap();

        // 08/2004
        YearMonth yearMonth = year.atMonth(Month.AUGUST);

        // 31/08/2004
        LocalDate localDate = yearMonth.atEndOfMonth();
    }
}
```

# La classe Instant

La classe [Instant](#) représente un point dans le temps. Contrairement aux classes précédentes qui permettent de représenter les dates pour les humains, la classe [Instant](#) est adaptée pour réaliser des traitements de données temporelles.

```
package dev.gayerie;

import java.time.Instant;

public class TestInstant {

    public static void main(String[] args) {
        Instant maintenant = Instant.now();
        Instant epoch = Instant.ofEpochSecond(0); // 01/01/1970 00:00:00.000

        Instant uneMinuteDansLeFuture = maintenant.plusSeconds(60);

        long unixTimestamp = uneMinuteDansLeFuture.getEpochSecond();
    }
}
```

Note

Les classes [LocalDate](#), [LocalTime](#), [LocalDateTime](#), [ZonedDateTime](#), [Year](#), [YearMonth](#), [Instant](#) implémentent toutes les interfaces [Temporal](#) et [TemporalAccessor](#). Cela permet d'utiliser facilement des instances de ces classes les unes avec les autres puisque beaucoup de leurs méthodes attendent en paramètres des instances de type [Temporal](#) ou [TemporalAccessor](#).

## Période et durée

Il est possible de définir des périodes grâce à des instances de la classe [Period](#). Une période peut être construite directement ou à partir de la différence entre deux instances de type [Temporal](#). Il est ensuite possible de modifier une date en ajoutant ou soustrayant une période.

```
package dev.gayerie;

import java.time.LocalDate;
import java.time.Month;
import java.time.Period;
import java.time.Year;
import java.time.YearMonth;

public class TestPeriode {

    public static void main(String[] args) {
        YearMonth moisAnnee = Year.of(2000).atMonth(Month.APRIL); // 04/2000

        // période de 1 an et deux mois
        Period periode = Period.ofYears(1).plusMonths(2);
    }
}
```

```

        YearMonth moisAnneePlusTard = moisAnnee.plus(periode); // 06/2001

        Period periode65Jours = Period.between(LocalDate.now(),
LocalDate.now().plusDays(65));
    }

}

```

La durée est représentée par une instance de la classe [Duration](#). Elle peut être obtenue à partir de deux instances de [Instant](#).

```

package dev.gayerie;

import java.time.Duration;
import java.time.Instant;

public class TestDuree {

    public static void main(String[] args) {
        Instant debut = Instant.now();

        // ... traitement à mesurer

        Duration duree = Duration.between(debut, Instant.now());
        System.out.println(duree.toMillis());
    }

}

```

## Formatage des dates

Pour formater une date pour l’affichage, il est possible d’utiliser la méthode `format` déclarée dans les classes [LocalDate](#), [LocalTime](#), [LocalDateTime](#), [ZonedDateTime](#), [Year](#) et [YearMonth](#).

Le format de représentation d’une date et/ou du temps est défini par la classe [DateTimeFormatter](#).

```

package dev.gayerie;

import java.time.LocalDateTime;
import java.time.Month;
import java.time.format.DateTimeFormatter;
import java.util.Locale;

public class TestDuree {

    public static void main(String[] args) {
        // 01/09/2010 16:30
        LocalDateTime dateTime = LocalDateTime.of(2010, Month.SEPTEMBER, 1, 16, 30);

        // En utilisant des formats ISO de dates
        System.out.println(dateTime.format(DateTimeFormatter.BASIC_ISO_DATE));
        System.out.println(dateTime.format(DateTimeFormatter.ISO_WEEK_DATE));
        System.out.println(dateTime.format(DateTimeFormatter.ISO_DATE_TIME));
    }

}

```



```

DateTimeFormatter datePattern = DateTimeFormatter.ofPattern("dd/MM/yyyy");
// 01/09/2010
System.out.println(dateTime.format(datePattern));

DateTimeFormatter dateTimePattern = DateTimeFormatter.ofPattern("dd/MM/yyyy
HH:mm");
// 01/09/2010 16:30
System.out.println(dateTime.format(dateTimePattern));

// 1 septembre 2010
DateTimeFormatter frenchDatePattern = DateTimeFormatter.ofPattern("d MMMM
yyyy", Locale.FRANCE);
System.out.println(dateTime.format(frenchDatePattern));
}

}

```

Note

Il est toujours possible d'utiliser la classe [SimpleDateFormat](#) pour formater une instance de la classe [java.util.Date](#).

## Lecture des dates

Pour transformer une chaîne de caractères en date (notamment pour obtenir une date à partir de la saisie d'un utilisateur ou de la lecture d'un fichier), il est possible d'utiliser la méthode `parse` déclarée dans les classes [LocalDate](#), [LocalTime](#), [LocalDateTime](#), [ZonedDateTime](#), [Year](#) et [YearMonth](#).

Le format d'une date et/ou du temps est défini par la classe [DateTimeFormatter](#).

```

package dev.gayerie;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.Year;
import java.time.YearMonth;
import java.time.format.DateTimeFormatter;

public class SaisieLocalDateTime {

    public static void main(String[] args) {

        String exemple = "02/06/2010 12:35";

        DateTimeFormatter dtf = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm");

        Year year = Year.parse(exemple, dtf);
        System.out.println(year);

        YearMonth yearMonth = YearMonth.parse(exemple, dtf);
        System.out.println(yearMonth);

        LocalDate localDate = LocalDate.parse(exemple, dtf);
    }
}

```

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

LocalDateTime localDateTime = LocalDateTime.parse(exemple, dtf);
System.out.println(localDateTime);
}

}
```

#### Note

Il est toujours possible d'utiliser la classe [SimpleDateFormat](#) pour transformer une chaîne de caractères en une instance de la classe [java.util.Date](#).

## Exercice

---

### Calcul de l'âge

Écrivez une application qui demande à l'utilisateur de saisir sa date de naissance au format jour/mois/année et qui affiche ensuite son âge. Pensez à gérer le cas où l'utilisateur ne saisit pas la date au format attendu.

#### Astuce

Pour calculer l'âge, il faut utiliser [la classe Periode](#).

## Programmation fonctionnelle et lambdas

---

La programmation fonctionnelle est un paradigme de programmation basé sur l'appel et la composition de fonctions. À l'origine, elle apparaît comme une approche alternative à la programmation impérative à laquelle appartient la programmation objet. Néanmoins, de plus en plus de langages de programmation impératifs incorporent dans leur grammaire des éléments de programmation fonctionnelle. C'est également le cas en Java depuis sa version 8.

Un trait important de la programmation fonctionnelle est de considérer une fonction comme une entité de première classe (*first class citizen*), c'est-à-dire qu'on peut lui appliquer les mêmes opérations que n'importe quel autre élément du langage. Ainsi une fonction doit posséder une identité intrinsèque lui permettant d'être passée en paramètre d'une autre fonction ou retournée par une autre fonction.

#### Note

Une fonction qui accepte une ou plusieurs autres fonctions en paramètres et qui retourne une fonction est appelée une fonction d'ordre supérieur (*higher-order function*).

Une approche fonctionnelle va favoriser la représentation d'une application comme un appel chaîné de fonctions pouvant prendre elles-mêmes des fonctions en paramètres. Ce type de représentation à plusieurs avantages. D'abord, il favorise le découpage du traitement en une suite de processus simples représentés chacun par une fonction. Cela améliore la lisibilité et la testabilité d'un programme. Pour les langages impératifs comme Java, cela offre une manière alternative d'implémenter certains traitements en limitant l'usage des structures de contrôle comme `if` ou les structures de boucle comme `while` et `for` qui sont alors implémentées sous la forme d'appels de fonctions. Là encore, l'objectif est d'améliorer la lisibilité du code en évitant l'imbrication de blocs de traitement et d'accolades.

L'introduction d'éléments de programmation fonctionnelle en Java a été un processus complexe car la grammaire du langage ne permet pas la déclaration de fonction. Comme nous le verrons, les fonctions anonymes ou les références de fonction en Java sont en fait un sucre syntaxique : il s'agit d'une simplification d'écriture qui mobilise en arrière-plan des notions purement objet comme l'interface fonctionnelle.

## Les fonctions anonymes : les lambdas

Une lambda est une fonction anonyme, c'est-à-dire une fonction qui est déclarée sans être associée à un nom. Le terme *lambda* est emprunté à la méthode formelle du [lambda-calcul](#). Les fonctions lambda (ou plus simplement les lambdas) sont couramment utilisées dans la programmation fonctionnelle. Elles permettent d'écrire des programmes plus concis et elles permettent de créer des [closures](#) (*fermetures*).

Depuis la version 8, il est possible de déclarer des lambdas en Java.

### Syntaxe des lambdas

En Java, les lambdas s'écrivent sous la forme :

```
(paramètres) -> { corps }
```

- Les parenthèses ne sont pas obligatoires si la lambda n'a qu'un seul paramètre.
- Si le compilateur peut inférer le type des paramètres alors il n'est pas obligatoire de déclarer ce type.
- Les accolades peuvent être omises si la lambda n'a qu'une instruction et si le contexte le permet.
- Si le corps de la lambda ne contient qu'une instruction, on peut omettre le point-virgule à la fin de l'instruction.
- Si le corps de la lambda ne contient qu'une instruction, on peut omettre le mot-clé `return` et le compilateur infère que la fonction retourne le résultat de l'instruction.

Exemple d'une lambda qui attend deux nombres entiers comme paramètres et qui retourne la somme de ces nombres :

```
(int a, int b) -> { return a + b; }
```

Notez qu'il n'est pas nécessaire de fournir le type de retour de la fonction. Le compilateur peut inférer qu'il s'agit d'un `int` correspondant à la somme des deux paramètres.

Cette fonction anonyme est équivalente à la déclaration d'une méthode statique dans la classe :

```
public static int somme(int a, int b) {  
    return a + b;  
}
```

La forme lambda évite d'avoir à fournir un nom à la fonction dont l'implémentation est évidente. On peut encore simplifier la lambda en supprimant les accolades, le mot-clé `return` et le point-virgule puisque cette fonction anonyme ne contient qu'une seule instruction.

```
(int a, int b) -> a + b
```

#### Note

Le corps d'une lambda se limite le plus souvent à une seule instruction très simple. Même s'il est possible de déclarer une lambda ayant plusieurs lignes d'instruction, cela doit rester l'exception dans un programme.

D'autres exemples de lambdas :

Une fonction qui attend une chaîne de caractères pour l'afficher sur la console :

```
String s -> System.out.println(s)
```

Une fonction qui ne prend aucun paramètre et qui retourne le nombre 42 :

```
() -> 42
```

## Usage des lambdas

Depuis Java 8, beaucoup de méthodes se comportent maintenant comme des fonctions de premier ordre en acceptant des fonctions en paramètre. Il est alors possible de déclarer une lambda au moment de l'appel.

Prenons l'exemple de la méthode [sort](#). Cette méthode déclarée par l'interface [List](#) permet de trier les éléments de cette liste. Mais pour cela, cette méthode a besoin d'appliquer une comparaison sur les éléments deux à deux pour connaître la relation d'ordre. L'algorithme de comparaison peut être donné par une lambda :

```
List<Integer> liste = new ArrayList<>();
liste.add(1);
liste.add(2);
liste.add(3);
liste.add(4);

// trie la liste en plaçant en premier les nombres pairs
liste.sort((e1, e2) -> (e1 % 2) - (e2 % 2));

// [2, 4, 1, 3]
System.out.println(liste);
```

Notez qu'il n'est pas nécessaire de préciser le type des paramètres `e1` et `e2` pour la déclaration de la lambda. Le compilateur peut les inférer car les éléments de la liste sont de type `Integer`.

Beaucoup de méthodes ont également été ajoutées à partir de Java 8 comme la méthode [forEach](#) déclarée par l'interface [Iterable](#). On peut ainsi effectuer un traitement sur chaque élément d'une collection en passant une lambda en paramètre.

```
Collection<String> collection = new ArrayList<>();
collection.add("un");
collection.add("deux");
collection.add("trois");

collection.forEach(e -> System.out.println(e));
```

On voit dans cet exemple simple que la programmation fonctionnelle implique un *style* différent de programmation. Nous pourrions écrire la même portion de code en utilisant une structure de boucle typique de la programmation impérative :

```
Collection<String> collection = new ArrayList<>();
collection.add("un");
collection.add("deux");
collection.add("trois");

for(String e : collection) {
    System.out.println(e);
}
```

## Lambda et closure

Une lambda définit une *closure* (fermeture), c'est-à-dire qu'elle définit un environnement lexical constitué de toutes les variables et de tous les attributs qu'elle capture au moment de sa déclaration. Le corps d'une lambda peut donc accéder au contenu d'une variable déclarée dans le code englobant.

1 2 3	List<String> prenoms = new ArrayList<>(); prenoms.add("Murielle");
4 5 6	prenoms.add("Jean"); prenoms.add("Michelle"); List<String> helloList
7 8 9	= new ArrayList<>(); prenoms.forEach(e -> helloList.add("Hello " +
10 11	e)); // [Hello Murielle, Hello Jean, Hello Michelle]
	system.out.println(helloList);

Dans l'exemple ci-dessus, on déclare à la ligne 6 la variable `helloList` correspondant à une nouvelle liste. À la ligne 8, la lambda a pour tâche d'ajouter un nouvel élément dans cette liste. Pour le compilateur, la variable `helloList` utilisée dans la lambda correspond bien à celle déclarée précédemment. Cette variable a été capturée par la *closure* définie par la lambda.

Comme pour la déclaration de classe anonyme, une lambda ne peut pas modifier le contenu d'un paramètre ou d'une variable issue d'une *closure* (le compilateur émettra une erreur dans ce cas). Par contre, il n'est pas nécessaire de déclarer comme **final** une variable ou un paramètre pour pouvoir y accéder dans une lambda.

```
List<Integer> liste = new ArrayList<>();
liste.add(1);
liste.add(2);
liste.add(3);
liste.add(4);

int i = 0;
liste.forEach(e -> i += e); // ERREUR DE COMPILATION : la variable i ne peut pas
                             // être modifiée car elle fait partie de la closure.
```

## Les interfaces fonctionnelles

La grammaire du langage Java ne supporte pas la notion de fonction. Il n'est donc pas possible de déclarer des fonctions en Java et encore moins des fonctions anonymes pour les passer en paramètre des méthodes. Pour incorporer des éléments de programmation fonctionnelle, les concepteurs du langage ont dû recourir à une astucieuse manipulation. Les lambdas correspondent en fait à une implémentation **d'une interface qui ne déclare qu'une seule méthode**. Le corps de la lambda correspond au code d'implémentation de cette unique méthode. Ainsi, on appelle **interface fonctionnelle** une interface déclarant une seule méthode abstraite et dont l'implémentation peut être fournie par une lambda.

Si nous déclarons l'interface ci-dessous :

```
package dev.gayerie;

public interface OperationSimple {

    int calculer(int i);

}
```

Alors, partout où le programme attend une implémentation de cette interface, il est possible de fournir une lambda :

```
OperationSimple os = i -> 2 * i;

int resultat = os.calculer(10);

System.out.println(resultat); // Affiche 20
```

Ainsi, il est très facile de déclarer des méthodes qui acceptent une lambda en paramètre. Il suffit de spécifier un paramètre correspondant à une interface fonctionnelle.

```
public class ValeurSimple {

    private int valeur;

    public ValeurSimple(int valeur) {
        this.valeur = valeur;
    }

}
```

```

    public void appliquer(OperationSimple os) {
        valeur = os.calculer(valeur);
    }

    public int getValeur() {
        return valeur;
    }
}

```

Dans la classe déclarée ci-dessus, la méthode `appliquer` attend en paramètre une instance d'un objet implémentant une interface fonctionnelle. Il est donc possible de fournir une lambda :

```

ValeurSimple vs = new ValeurSimple(10);

vs.appliquer(v -> v * v);

System.out.println(vs.getValeur()); // Affiche 100

```

L'annotation [@FunctionalInterface](#) peut être utilisée lors de la déclaration de l'interface. Elle signale au compilateur qu'il doit vérifier que cette interface peut être implémentée par des lambdas. Le compilateur contrôle que l'interface ne comporte qu'une seule méthode abstraite et signale une erreur dans le cas contraire.

```

package dev.gayerie;

@FunctionalInterface
public interface OperationSimple {

    int calculer(int i);

}

```

Il est donc très simple d'introduire des lambdas même pour des bibliothèques et des applications qui ont été développées avant Java 8.

Note

Concernant la méthode [sort](#) présentée précédemment, sa signature n'a pas évolué en Java 8. Cette méthode attend toujours en paramètre un objet qui implémente l'interface [Comparator](#). L'interface [Comparator](#) ne déclarant qu'une seule méthode abstraite, il s'agit donc d'une interface fonctionnelle.

Afin d'éviter aux développeurs de créer systématiquement leurs interfaces, le package [java.util.function](#) déclare les interfaces fonctionnelles les plus utiles. Par exemple, l'interface [java.util.function.IntUnaryOperator](#) déclare une méthode `applyAsInt` qui accepte un entier en paramètre et qui retourne un autre entier. Nous pouvons nous en servir pour définir un régulateur de vitesse dans une classe *Voiture*.

```

package dev.gayerie;

import java.util.function.IntUnaryOperator;

```

```

public class Voiture {

    private int vitesse;
    private IntUnaryOperator regulateurDeVitesse = v -> v;

    public void accelerer(int deltaVitesse) {
        this.vitesse = regulateurDeVitesse.applyAsInt(this.vitesse + deltaVitesse);
    }

    public void setRegulateurDeVitesse(IntUnaryOperator regulateur) {
        this.regulateurDeVitesse = regulateur;
    }

    public int getVitesse() {
        return vitesse;
    }

}

Voiture v = new Voiture();
v.setRegulateurDeVitesse(vitesse -> vitesse > 110 ? 110 : vitesse);

v.accelerer(90);
System.out.println(v.getVitesse()); // 90

v.accelerer(90);
System.out.println(v.getVitesse()); // 110

```

Le package [java.util.function](#) fournit beaucoup d'interfaces. Pour s'y retrouver, il suffit de garder à l'esprit la convention de nom utilisée :

Les interfaces fonctionnelles dont le nom se termine par `Function` déclarent une méthode qui prend un ou plusieurs paramètres et qui retourne un résultat. Par exemple, l'interface [BiFunction](#) déclare la méthode `apply` qui accepte deux paramètres de types différents et retourne un paramètre d'un autre type.

Les interfaces fonctionnelles dont le nom se termine par `Operator` déclarent une méthode qui prend un ou plusieurs paramètres du même type et retourne un paramètre du même type. Par exemple, l'interface [DoubleUnaryOperator](#) déclare la méthode `applyAsDouble` qui attend un paramètre de type `double` et retourne un résultat de type `double`.

Les interfaces fonctionnelles dont le nom se termine par `Consumer` déclarent une méthode qui prend un ou plusieurs paramètres mais qui ne retourne aucun résultat. Par exemple, l'interface [LongConsumer](#) déclare la méthode `accept` qui attend un paramètre de type `Long` et qui ne retourne rien.

Les interfaces fonctionnelles dont le nom se termine par `Supplier` déclarent une méthode qui ne prend aucun paramètre mais qui retourne un résultat. Par exemple, l'interface [Supplier](#) déclare la méthode `get` qui n'attend aucun paramètre et qui retourne un objet du type générique de l'interface.

Les interfaces fonctionnelles dont le nom se termine par `Predicate` déclarent une méthode qui prend un paramètre et qui retourne un résultat de type booléen.



## L'opérateur :: de référence de méthode

Un aspect important de la programmation fonctionnelle est de pouvoir référencer les fonctions, notamment pour pouvoir les passer comme paramètres dans un appel à une autre fonction.

En Java, la notion de fonction n'existe pas en tant que telle et une méthode n'est pas une entité de première classe. Néanmoins, il est possible de référencer une méthode en utilisant l'opérateur `::`. La référence de méthode rend le code plus lisible en évitant de déclarer une lambda.

Prenons un exemple : nous souhaitons écrire une lambda qui attend une chaîne de caractères en paramètre et qui retourne la valeur de la chaîne de caractères en lettres majuscules. Du point de vue des interfaces fonctionnelles, il s'agit d'un opérateur unaire. Nous pouvons donc écrire :

```
UnaryOperator<String> f = s -> s.toUpperCase();
```

Mais nous pouvons également considérer que la méthode `toUpperCase` agit comme un opérateur unaire : elle s'applique sur un objet de type `String` puisqu'elle est une méthode de la classe `String` et elle retourne une valeur de type `String`. Donc, il est possible de se passer complètement de la lambda pour référencer directement la méthode `toUpperCase` :

```
UnaryOperator<String> f = String::toUpperCase;
```

Notez que dans l'exemple ci-dessus, nous n'appelons pas la méthode `toUpperCase` mais nous la référençons avec l'opérateur `::`.

Il est également possible de référencer une méthode d'un objet particulier. Dans ce cas, la méthode est nécessairement invoquée sur l'objet à partir duquel on référence la méthode. En programmation, on appelle cela une *binding*.

Si nous reprenons un exemple vu précédemment :

```
Collection<String> collection = new ArrayList<>();  
collection.add("un");  
collection.add("deux");  
collection.add("trois");  
  
collection.forEach(e -> System.out.println(e));
```

La méthode `forEach` attend en paramètre une instance qui implémente l'interface fonctionnelle `Consumer`. L'interface `Consumer` déclare la méthode `accept` qui prend un type `T` en paramètre et ne retourne rien. Si maintenant nous comparons cette signature avec celle de la méthode `println` appelée sur l'objet `System.out`, cette dernière attend un objet en paramètre et ne retourne rien. La signature de `println` est compatible avec celle de la méthode de l'interface fonctionnelle `Consumer`. Donc, plutôt que de déclarer une lambda, il est possible d'utiliser l'opérateur `::` pour passer la référence de la méthode `println` :

```
Collection<String> collection = new ArrayList<>();  
collection.add("un");  
collection.add("deux");  
collection.add("trois");  
  
collection.forEach(System.out::println); // passage de la référence de la méthode
```

Il est également possible de référencer les constructeurs d'une classe. Cela aboutira à la création d'un nouvel objet à chaque appel. Les constructeurs sont référencés grâce à la syntaxe :

```
NomDeLaClasse::new
```

Par exemple, nous pouvons utiliser l'interface fonctionnelle [Supplier](#). Cette interface fonctionnelle peut être implémentée en utilisant un constructeur sans paramètre. Ainsi, si nous définissons une classe *Voiture* avec un constructeur sans paramètre :

```
package dev.gayerie;

public class Voiture {

    public Voiture() {
        // ...
    }

}
```

Nous pouvons utiliser la référence de ce constructeur pour créer une implémentation de l'interface fonctionnelle [Supplier](#) :

```
Supplier<Voiture> garage = Voiture::new;

Voiture v1 = garage.get(); // crée une nouvelle instance
Voiture v2 = garage.get(); // crée une nouvelle instance
```

De la même manière, nous pouvons référencer un constructeur avec des paramètres. Par exemple, la classe [SimpleDateFormat](#) déclare un constructeur qui attend une chaîne de caractères en paramètre pour définir le format d'une date. Un tel constructeur s'apparente à une interface fonctionnelle de type [Function](#) :

```
Function<String, SimpleDateFormat> f = SimpleDateFormat::new;

SimpleDateFormat sdf = f.apply("dd/MM/YYYY");
```

#### Astuce

La référence de méthode permet de simplifier l'écriture du code mais elle permet aussi d'étendre le support de la programmation fonctionnelle au-delà des fonctions anonymes. Si vous devez écrire un traitement complexe, alors l'utilisation d'une lambda risque de complexifier la lecture de votre code. Dans ce cas, il est recommandé d'écrire le traitement sous la forme d'une méthode et d'utiliser l'opérateur de référence de méthode pour passer cette méthode en paramètre.

## La classe Optional

Pour donner un aperçu du style de programmation fonctionnelle en Java, nous allons prendre l'exemple de la classe générique [Optional](#) introduite en Java 8.

Dans beaucoup de langages de programmation, nous avons la possibilité d'affecter à des variables, des paramètres ou des attributs une référence nulle.

```
String s = null;
```

Il faut bien reconnaître que cela n'a pas vraiment de sens puisque nous affectons ainsi à quelque chose, quelque chose qui n'est rien ! Et surtout cela peut conduire à toutes sortes de bugs et à l'apparition inopinée de [NullPointerException](#) en Java.

Java 8 introduit la classe [Optional](#) qui permet de représenter la possibilité qu'il existe ou non une valeur sans avoir besoin d'utiliser `null`. La classe [Optional](#) fournit les méthodes de construction statiques [of](#) et [ofNullable](#) ainsi que la méthode [isPresent](#) pour vérifier si une valeur est présente et la méthode [get](#) pour obtenir sa valeur.

```
Optional<String> v = Optional.ofNullable("test");

if (v.isPresent()) {
    System.out.println(v);
} else {
    System.out.println("valeur non trouvée");
}
```

Si on utilise une instance de la classe [Optional](#) à la manière des langages impératifs, on peut obtenir du code simple à comprendre mais assez inutilement long. L'intérêt de la classe [Optional](#) est de l'utiliser avec un style de programmation fonctionnelle.

```
Optional<String> v = Optional.ofNullable("test");

// retourne la valeur de l'optional si elle est définie ou la valeur par
// défaut passée en paramètre.
System.out.println(v.orElse("valeur non trouvée"));

// invoque la méthode passée en référence uniquement si une valeur est définie
// par l'optional
v.ifPresent(System.out::println);

// retourne la valeur de l'optional si elle est définie ou jette une exception
// construite à partir du constructeur fourni en paramètre.
String resultat = v.orElseThrow(ValeurNonTrouveeException::new);
```

Nous pourrions écrire le même code en style impératif mais au prix de l'utilisation de plusieurs structures de contrôle `if ... else`.

## Streams

L'API *streams* a été introduite avec Java 8 pour permettre la programmation fonctionnelle. Un *stream* (flux) est une représentation d'une séquence sur laquelle il est possible d'appliquer des opérations. Cette API a deux principaux intérêts :

1. Elle permet d'effectuer les opérations sur une séquence sans utiliser de structure de boucle. Cela permet de réaliser des traitements complexes tout en maintenant une bonne lisibilité du code.

2. Les opérations sur les *streams* sont réalisées en flux (d'où leur nom) ce qui limite l'empreinte mémoire nécessaire. Il est même possible de réaliser très simplement des traitements en parallèle pour tirer partie des possibilités d'une processeur multi-cœurs ou d'une machine multi-processeurs.

## Création d'un stream

Un *stream* est représenté par une instance de l'interface générique [Stream](#). On peut créer un [Stream](#) en utilisant un objet de type [builder](#)

```
Stream<String> stream = Stream.  
<String>builder().add("Hello").add("World").build();
```

Il existe également des interfaces filles de [Stream](#) pour certains types primitifs : [IntStream](#), [LongStream](#) et [DoubleStream](#). On peut créer des *streams* de ces types soit à partir d'une liste de valeurs soit en donnant les limites d'un intervalle.

```
IntStream intStream = IntStream.of(1, 20, 30, 579);  
  
IntStream rangeIntStream = IntStream.range(0, 1_000_000_000);
```

### Note

Comme mentionné à la section précédente, un des intérêts des *streams* vient de leur nature même de flux. Ainsi dans l'exemple précédent, la création d'un *stream* à partir d'un intervalle ne crée pas une valeur pour chaque élément. Ainsi la création d'un *stream* sur un intervalle d'un milliard est instantanée et ne prend presque aucune place en mémoire.

Il est même possible de créer un *stream* « infini » dont les valeurs sont calculées par une lambda.

```
// Un stream commençant à la valeur 1 et qui est représenté par la suite n = n + 1  
LongStream longStream = LongStream.iterate(1, n -> n + 1);
```

Il est également possible de créer un *stream* à partir d'un tableau grâce aux méthodes [Arrays.stream](#) :

```
int[] tableau = { 1, 2, 3, 4 };  
IntStream tableauStream = Arrays.stream(tableau);
```

Les collections peuvent également être utilisées sous la forme d'un *stream* car l'interface [Collection](#) définit la méthode [Collection.stream](#).

```
List<String> liste = new ArrayList<>();  
liste.add("Hello");  
liste.add("World");  
  
Stream<String> stream = liste.stream();
```

Le contenu d'un fichier texte peut aussi être parcouru sous la forme d'un *stream* de chacune de ses lignes grâce à la méthode [Files.lines](#) :

```
Path fichier = Paths.get("fichier.txt");
Stream<String> linesStream = Files.lines(fichier);
```

Ainsi, toutes les opérations qui impliquent une séquence d'éléments peuvent être traitées sous la forme d'un *stream*.

Il est possible de réaliser un traitement sur chaque élément du *stream* grâce à la méthode [Stream.forEach](#).

```
// Affiche les chiffres de 10 jusqu'à 0
IntStream.iterate(10, n -> n - 1).limit(11).forEach(System.out::println);
```

Un *stream* est également utilisé pour produire un résultat unique ou une collection. Dans le premier cas, on dit que l'on réduit, tandis que dans le second cas, on dit que l'on collecte.

## La réduction

La réduction consiste à obtenir un résultat unique à partir d'un *stream*. On peut par exemple compter le nombre d'éléments. Si le *stream* est composé de nombres, on peut réaliser une réduction mathématique en calculant la somme, la moyenne ou en demandant la valeur minimale ou maximale...

```
long resultat = LongStream.range(0, 50).sum();
System.out.println(resultat);

OptionalDouble moyenne = LongStream.range(0, 50).average();
if (moyenne.isPresent()) {
    System.out.println(moyenne.getAsDouble());
}
```

L'API *streams* introduit la notion de *Optional*. Certaines opérations de réduction peuvent ne pas être possibles. Par exemple, le calcul de la moyenne n'est pas possible si le *stream* ne contient aucun élément. La méthode [average](#) qui permet de calculer la moyenne d'un *stream* numérique retourne donc un [OptionalDouble](#) qui permet de représenter soit le résultat, soit le fait qu'il n'y a pas de résultat. On peut appeler la méthode [OptionalDouble.isPresent](#) pour s'assurer qu'il existe un résultat pour cette réduction.

Pour les streams de tout type, il est possible de réaliser une réduction à partir d'une lambda grâce à la méthode [Stream.reduce](#).

```
List<String> liste = Arrays.asList("une chaine", "une autre chaine", "encore une chaine");
Optional<String> chaineLaPlusLongue = liste.stream().reduce((s1, s2) ->
    s1.length() > s2.length() ? s1 : s2);

System.out.println(chaineLaPlusLongue.get()); // "encore une chaine"
```

# La collecte

La collecte permet de créer une nouvelle collection à partir d'un stream. Pour cela, il faut fournir une implémentation de l'interface [Collector](#). Cette interface est assez complexe, heureusement la classe util [Collectors](#) fournit des méthodes pour générer une instance de [Collector](#). Pour réaliser la collecte, il faut appeler la méthode [Stream.collect](#).

On peut ainsi collecter les éléments d'un *stream* sous la forme d'une [List](#), d'un [Set](#) ou de tout type de [Collection](#).

```
List<String> liste = Arrays.asList("une chaine", "une autre chaine", "encore une chaine");
List<String> autreListe = liste.stream().collect(Collectors.toList());
```

L'exemple précédent peut sembler trivial puisqu'au final, ce code crée une copie de la liste d'origine. Son intérêt deviendra évident lorsque nous appliquerons des opérations de filtre ou de mapping sur un *stream*.

Un [Collector](#) peut également réaliser une opération de regroupement pour créer des [Map](#). Si on dispose de la classe *Voiture* :

```
package dev.gayerie;

public class Voiture {

    private String marque;

    public Voiture(String marque) {
        this.marque = marque;
    }

    public String getMarque() {
        return marque;
    }
}
```

Alors il devient facile de grouper des instances d'une liste de *Voiture* selon leur marque.

```
List<Voiture> liste = Arrays.asList(new Voiture("citroen"),
                                   new Voiture("renault"),
                                   new Voiture("audi"),
                                   new Voiture("citroen"));

Map<String, List<Voiture>> map =
    liste.stream().collect(Collectors.groupingBy(Voiture::getMarque));

System.out.println(map.get("citroen").size()); // 2
System.out.println(map.get("renault").size()); // 1
System.out.println(map.get("audi").size());    // 1
```

On peut également créer une chaîne de caractères en joignant les éléments d'un *stream* :

```
List<String> list = Arrays.asList("un", "deux", "trois", "quatre", "cinq");
String resultat = list.stream().collect(Collectors.joining(", "));

System.out.println(resultat); // "un, deux, trois, quatre, cinq"
```

## Le filtrage

Une opération courante sur un *stream* consiste à appliquer un filtre pour éliminer une partie de ses éléments. Pour, cela on peut utiliser la méthode [Stream.filter](#).

```
List<Voiture> liste = Arrays.asList(new Voiture("citroen"),
                                   new Voiture("audi"),
                                   new Voiture("citroen"));

// on construit la liste des voitures qui ne sont pas de marque "citroen"
List<Voiture> sansCitroen = liste.stream()
                                .filter(v -> !v.getMarque().equals("citroen"))
                                .collect(Collectors.toList());

System.out.println(sansCitroen.size()); // 1
// On affiche les 500 premiers nombres qui ne sont pas divisibles par 7
IntStream.iterate(1, n -> n + 1)
    .filter(n -> n % 7 != 0)
    .limit(500)
    .forEach(System.out::println);
```

La méthode [Stream.filter](#) peut accepter une lambda qui reçoit en paramètre un élément du *stream* et qui retourne un **boolean** (**true** signifie que l'élément doit être conservé dans le *stream*). On peut bien évidemment chaîner les appels à la méthode [Stream.filter](#) :

```
// On affiche les 500 premiers nombres qui ne sont pas divisibles par 7
// et qui sont impairs
IntStream.iterate(1, n -> n + 1)
    .filter(n -> n % 7 != 0)
    .filter(n -> n % 2 != 0)
    .limit(500)
    .forEach(System.out::println);
```

## Le mapping

Le mapping est une opération qui permet de transformer la nature du *stream* afin de passer d'un type à un autre.

Par exemple, si nous voulons récupérer l'ensemble des marques distinctes d'une liste de *Voiture*, nous pouvons utiliser un mapping pour passer d'un *stream* de *Voiture* à un *stream* de [String](#) (représentant les marques des voitures).

```
List<Voiture> liste = Arrays.asList(new Voiture("citroen"),
                                   new Voiture("audi"),
                                   new Voiture("renault"),
                                   new Voiture("volkswagen"),
                                   new Voiture("citroen"));

// mapping du stream de voiture en stream de String
Set<String> marques = liste.stream()
                           .map(Voiture::getMarque)
                           .collect(Collectors.toSet());

system.out.println(marques); // ["audi", "citroen", "renault", "volkswagen"]
```

Pour réaliser un mapping vers un type primitif, il faut utiliser les méthodes [Stream.mapToInt](#), [Stream.mapToLong](#) ou [Stream.mapToDouble](#). On peut également utiliser ces méthodes pour convertir un *stream* contenant un type primitif vers un *stream* contenant un autre type primitif.

```
// Affichage de la racine carré des 100 premiers entiers
IntStream.range(1, 101)
          .mapToDouble(Math::sqrt)
          .forEach(System.out::println);
```

Pour la méthode [Stream.map](#), le type de retour de la lambda ou de la référence de méthode indique le nouveau type du *stream*.

## Le parallélisme

Afin de tirer profit des processeurs multi-cœurs et des machines multi-processeurs, les opérations sur les *streams* peuvent être exécutées en parallèle. À partir d'une [Collection](#), il suffit d'appeler la méthode [Collection.parallelStream](#) ou à partir d'un [Stream](#), il suffit d'appeler la méthode [BaseStream.parallel](#).

Un *stream* en parallèle découpe le flux pour assigner l'exécution à différents processeurs et recombine ensuite le résultat à la fin. Cela signifie que les traitements sur le *stream* ne doivent pas être dépendant de l'ordre d'exécution.

Par exemple, si vous utilisez un *stream* parallèle pour afficher les 100 premiers entiers, vous constaterez que la sortie du programme est imprédictible.

```
// affiche les 100 premiers entiers sur la console en utilisant un stream
// parallèle.
// Ceci n'est pas une bonne idée car l'opération d'affichage implique
// que le stream est parcouru séquentiellement. Or un stream parallèle
// est réparti sur plusieurs processeurs et donc l'ordre d'exécution
// n'est pas prédictible
IntStream.range(1, 101).parallel().forEach(System.out::println);
```

Par contre, les streams parallèles peuvent être utiles pour des réductions de type somme puisque le calcul peut être réparti en sommes intermédiaires avant de réaliser la somme totale.

## Exercice

Exercice - Chaîne de caractères et streams



Utilisez l'API des Streams pour compter le nombre de lettres dans une chaîne de caractères.

Astuce

- La méthode [chars](#) permet d'obtenir un stream de caractères depuis un objet de type [String](#).
- La méthode [Character.isAlphabetic](#) retourne **true** si le caractère passé en paramètre est une lettre.

Exercice - Lecture d'un fichier CSV

Étant donné un fichier CSV contenant une liste de produits. Pour chaque ligne, on a le nom du produit, le montant HT du produit et la taxe en pourcentage pour ce produit :

```
produit1;12.3;20
produit2;5.3;5.5
produit4;123.23;20
```

En utilisant les streams, écrivez deux programmes :

- Le premier programme doit retourner le prix TTC moyen de tous les produits
- Le second programme doit afficher la liste des produits taxés à 5,5% et qui coûtent moins de 100€ HT.

Astuce

Vous pouvez créer une classe *Produit* pour représenter en interne chacun des produits.

## Les classes internes

La plupart du temps, une classe en Java est déclarée dans un fichier portant le même nom que la classe avec l'extension *.java*. Cependant, il est également possible de déclarer des classes dans une classe. On parle alors de classes internes (*inner classes*). Cela est également possible, dans une certaine limite, pour les interfaces et les énumérations.

La déclaration des classes internes peut se faire dans l'ordre que l'on souhaite à l'intérieur du bloc de déclaration de la classe englobante. Les classes internes peuvent être ou non déclarées **static**. Ces deux cas correspondent à deux usages particuliers des classes internes.

```
package dev.gayerie;

public class ClasseEnglobante {

    public static class ClasseInterneStatic {
    }

    public class ClasseInterne {
    }

}
```

## Les classes internes static

Les classes internes déclarées **static** sont des classes pour lesquelles l'espace de noms est celui de la classe englobante.

```
package dev.gayerie;

public class ClasseEnglobante {

    public static class ClasseInterne {
    }

}
```

Pour une classe interne **static** :

- Son nom complet inclut le nom de la classe englobante (qui agit comme un package). Pour la classe ci-dessus, le nom complet de *ClasseInterne* est :

```
dev.gayerie.ClasseEnglobante.ClasseInterne
```

- La classe englobante et la classe interne partagent le même espace privé. Cela signifie que les attributs et les méthodes privés déclarés dans la classe englobante sont accessibles à la classe interne. Réciproquement, la classe englobante peut avoir accès aux éléments privés de la classe interne.
- Une instance de la classe interne n'a accès directement qu'aux attributs et aux méthodes de la classe englobante qui sont déclarés **static**.

Une classe interne **static** est souvent utilisée pour éviter de séparer dans des fichiers différents de petites classes utilitaires et ainsi de faciliter la lecture du code. Dans l'exemple ci-dessous, plutôt que de créer un fichier spécifique pour l'implémentation d'un comparateur, on ajoute son implémentation comme une classe interne.

```
package dev.gayerie;
import java.util.Comparator;

public class Individu {

    public static class Compareur implements Comparator<Individu> {
        @Override
        public int compare(Individu i1, Individu i2) {
            if (i1 == null) {
                return -1;
            }
            if (i2 == null) {
                return 1;
            }
            int cmp = i1.nom.compareTo(i2.nom);
            if (cmp == 0) {
                cmp = i1.prenom.compareTo(i2.prenom);
            }
            return cmp;
        }
    }

    private final String prenom;
    private final String nom;
```

```

public Individu(String prenom, String nom) {
    this.prenom = prenom;
    this.nom = nom;
}

@Override
public String toString() {
    return this.prenom + " " + this.nom;
}

}

Individu[] individus = {
    new Individu("John", "Eod"),
    new Individu("Annabel", "Doe"),
    new Individu("John", "Doe")
};

Arrays.sort(individus, new Individu.Comparateur());

System.out.println(Arrays.toString(individus));

```

Dans l'exemple ci-dessus, la classe *Individu* fournit publiquement une implémentation d'un [Comparator](#) qui permet de comparer deux instances en fonction de leur nom et de leur prénom. Notez que l'implémentation de la méthode *compare* peut accéder aux attributs privés *nom* et *prenom* des paramètres *i1* et *i2* car ils sont de type *Individu*.

## Les classes internes

Une classe interne qui n'est pas déclarée avec le mot-clé **static** est liée au contexte d'exécution d'une instance de la classe englobante.

Comme pour les classes internes **static**, le nom complet de classe interne inclut celui de la classe englobante et les deux classes partagent le même espace privé. Mais surtout, une classe interne maintient une référence implicite sur un objet de la classe englobante. Cela signifie que :

- une instance d'une classe interne ne peut être créée que par un objet de classe englobante : c'est-à-dire dans le corps d'une méthode ou dans le corps d'un constructeur de la classe englobante.
- une instance d'une classe interne a accès directement aux attributs de l'instance dans le contexte de laquelle elle a été créée.

Une classe interne est utilisée pour créer un objet qui a couplage très fort avec un objet du type de la classe englobante. On utilise fréquemment le mécanisme de classe interne lorsque l'on veut réaliser une interface graphique en Java avec l'API [Swing](#).

```
1 2 package dev.gayerie; import java.awt.FlowLayout; import
3 4 java.awt.event.ActionEvent; import javax.swing.AbstractAction; import
5 6 javax.swing.JButton; import javax.swing.JDialog; import
7 8 javax.swing.JLabel; public class BoiteDeDialogue extends JDialog {
9 private class IncrementerAction extends AbstractAction { public
10 IncrementerAction() { super("Incrémenter"); } @Override public void
11 actionPerformed(ActionEvent e) { incrementer(); } } private class
12 DecrementerAction extends AbstractAction { public DecrementerAction()
13 { super("Décrémenter"); } @Override public void
14 actionPerformed(ActionEvent e) { decrementer(); } } private JLabel
15 label; private int valeur; @Override protected void dialogInit() {
16 super.dialogInit(); this.setLayout(new FlowLayout()); this.label = new
17 JLabel(Integer.toString(this.valeur)); this.add(this.label);
18 this.add(new JButton(new IncrementerAction())); this.add(new
19 JButton(new DecrementerAction())); this.pack(); } private void
20 incrementer() { label.setText(Integer.toString(++this.valeur)); }
21 private void decrementer() { label.setText(Integer.toString(--
22 this.valeur)); } public static void main(String[] args) {
23 BoiteDeDialogue boiteDeDialogue = new BoiteDeDialogue();
24 boiteDeDialogue.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
25 boiteDeDialogue.setVisible(true); } }
```

52	
53	
54	
55	
56	
57	
58	
59	
60	
61	
62	
63	

L'exemple ci-dessus est un programme complet qui crée une boîte de dialogue contenant deux boutons qui permettent respectivement d'incrémenter et de décrémenter un nombre qui est affiché. La classe [JButton](#) qui représente un bouton attend comme paramètre de construction une instance implémentant l'interface [Action](#). Cette instance définit le libellé du bouton et l'action à réaliser lorsque l'utilisateur clique sur le bouton. Les boutons sont créés aux lignes 44 et 45. Les classes d'action utilisées pour chaque bouton sont définies aux lignes 13 et 24. Ces classes sont des classes internes. Dans leur méthode [actionPerformed](#), elles appellent soit la méthode *incrémenter* soit la méthode *décrémenter*. Ces deux méthodes sont définies par la classe englobante *BoiteDeDialogue*. Donc les instances de ces classes d'action appellent ces méthodes sur l'instance de l'objet englobant qui les a créées. Ainsi, les classes internes possèdent une référence sur l'objet *BoiteDeDialogue* qui les a créées.

Note

Notez dans l'exemple ci-dessus que les méthodes *BoiteDeDialogue.incrémenter* et *BoiteDeDialogue.décrémenter* sont privées. Comme une classe interne partage la même portée que sa classe englobante alors les classes internes *IncrémenterAction* et *DécrémenterAction* peuvent appeler ces méthodes.

## Les classes anonymes

Une classe anonyme est une classe qui n'a pas de nom. Elle est déclarée au moment de l'instanciation d'un objet. Comme une classe anonyme n'a pas de nom, il n'est pas possible de déclarer une variable qui serait un type de cette classe. Une classe anonyme est donc utilisée pour créer à la volée une classe qui spécialise une autre classe ou qui implémente une interface. Pour déclarer une classe anonyme, on déclare le bloc de la classe au moment de l'instanciation avec **new**.

Imaginons que nous souhaitions créer une interface pour représenter un système de log :

```
package dev.gayerie.logger;

public interface Logger {

    void log(String message);

}
```

On peut fournir une classe *GenerateurLogger* qui crée des instances implémentant l'interface *Logger*.

```
1
2
3
4
5
6
7
8
9
10
11 package dev.gayerie.logger; import java.time.LocalDateTime; public class GenerateurLogger {
12     private String application; /** * @param application Le nom de l'application */ public
13     GenerateurLogger(String application) { this.application = application; } public Logger
14     creerConsoleLogger() { return new Logger() { @Override public void log(String message) { //
15     Pour le format du message utilisé dans printf // Cf.
16     https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Formatter.html#syntax
17     System.out.println(String.format("%1$tY-%1$tb-%1$ta %1$tH:%1$tM %2$s - %3$s",
18     LocalDateTime.now(), application, message)); } }; } }
19
20
21
22
23
24
25
26
27
28
```

L'implémentation de la méthode *creerConsoleLogger* crée une instance implémentant l'interface *Logger* à partir d'une classe anonyme. L'implémentation de la méthode *log* affiche sur la sortie standard une chaîne de caractères formatée contenant la date et l'heure, le nom de l'application et le message passé en paramètre. Le nom de l'application correspond à l'attribut *application* de la classe *GenerateurLogger*. Comme pour les classes internes, les classes anonymes ont accès aux attributs et aux méthodes de l'objet englobant.

Il est possible de récupérer un objet implémentant *Logger* :

```
GenerateurLogger generateur = new GenerateurLogger("mon_appli");
Logger logger = generateur.creerConsoleLogger();
logger.log("un message de log");
```

Le code précédent affichera sur la sortie standard :

```
2017-nov.-jeu. 15:58 mon_appli - un message de log
```

Nous pouvons enrichir notre implémentation. Par exemple, la classe *GenerateurLogger* peut créer un logger qui ne fait rien ou encore un logger qui écrit les messages dans un fichier.

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22 package dev.gayerie.logger; import java.io.IOException; import java.io.Writer; import
23 java.nio.file.Files; import java.nio.file.Path; import java.nio.file.Paths; import
24 java.nio.file.StandardOpenOption; import java.time.LocalDateTime; public class
25 GenerateurLogger { private String application; /** * @param application Le nom de
26 l'application */ public GenerateurLogger(String application) { this.application =
27 application; } public Logger creerConsoleLogger() { return new Logger() { @Override public
28 void log(String message) { // Pour le format du message utilisé dans printf // Cf.
29 https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Formatter.html#syntax
30 System.out.println(String.format("%1$tY-%1$tb-%1$ta %1$tH:%1$tM %2$s - %3$s",
31 LocalDateTime.now(), application, message)); } }; } public Logger creerNoopLogger() { return
32 new Logger() { @Override public void log(String message) { } }; } public Logger
33 creerFileLogger(Path path) { return new Logger() { @Override public void log(String message)
34 { // Pour le format du message utilisé dans printf // Cf.
35 https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Formatter.html#syntax
36 String logMessage = String.format("%1$tY-%1$tb-%1$ta %1$tH:%1$tM %2$s - %3$s",
37 LocalDateTime.now(), application, message); try(Writer w = Files.newBufferedWriter(path,
38 StandardOpenOption.CREATE, StandardOpenOption.APPEND)) { w.append(logMessage).append('\n');
39 } catch (IOException e) { System.err.println(logMessage); } } }; }
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59

```

La classe ci-dessus définit maintenant trois classes anonymes qui implémentent toutes l'interface *Logger*. Notez à la ligne 50, que la classe anonyme qui écrit le message de log dans un fichier, ouvre le fichier à partir d'un paramètre *path* passé à la méthode *creerFileLogger*. Cela signifie qu'une classe anonyme a accès au paramètre de la méthode qui la déclare.

Note

Une classe anonyme peut utiliser les paramètres et les variables de la méthode qui la déclare uniquement à condition qu'ils ne soient modifiés ni par la méthode ni par la classe anonyme. Avant Java 8, le compilateur exigeait que ces paramètres et ces variables soient déclarés avec le mot-clé **final**. Même s'il n'est plus nécessaire de déclarer explicitement le statut **final**, le compilateur générera tout de même une erreur si on tente de modifier un paramètre ou une variable déclaré dans la méthode et utilisé par une classe anonyme.

```
// on déclare le paramètre final pour signaler explicitement qu'il n'est
// pas possible de modifier la référence de ce paramètre puisqu'il est
// utilisé par la classe anonyme.
public Logger creerFileLogger(final Path path) {
    return new Logger() {
        @Override
        public void log(String message) {
            // Pour le format du message utilisé dans printf
            // cf.
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Formatter.html#syntax
            String logMessage = String.format("%1$tY-%1$tb-%1$ta %1$tH:%1$tM %2$s -
            %3$s",
                                                    LocalDateTime.now(), application,
            message);
            try(Writer w = Files.newBufferedWriter(path, StandardOpenOption.CREATE,
            StandardOpenOption.APPEND)) {
                w.append(logMessage).append('\n');
            } catch (IOException e) {
                System.err.println(logMessage);
            }
        }
    };
}
```

## Accès aux éléments de l'objet englobant

Si nous reprenons notre code de la classe *GénérateurLogger*, nous nous rendons compte que le formatage du message a été dupliqué pour le logger qui écrit sur la sortie standard et pour celui qui écrit dans un fichier. Afin de mutualiser le code, nous pouvons créer une méthode *genererLogMessage* dans la classe englobante qui pourra être appelée par chaque classe anonyme.

```
package dev.gayerie.logger;

import java.io.IOException;
import java.io.Writer;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.time.LocalDateTime;

public class GénérateurLogger {

    private String application;
```



```

/**
 * @param application Le nom de l'application
 */
public GenerateurLogger(String application) {
    this.application = application;
}

public Logger creerConsoleLogger() {
    return new Logger() {
        @Override
        public void log(String message) {
            // Pour le format du message utilisé dans printf
            // cf.
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Formatter.html#syntax
            System.out.println(genererLogMessage(message));
        }
    };
}

public Logger creerFileLogger(Path path) {
    return new Logger() {
        @Override
        public void log(String message) {
            try(Writer w = Files.newBufferedWriter(path, StandardOpenOption.CREATE,
StandardOpenOption.APPEND)) {
                w.append(genererLogMessage(message)).append('\n');
            } catch (IOException e) {
                System.err.println(genererLogMessage(message));
            }
        }
    };
}

public Logger creerNoopLogger() {
    return new Logger() {
        @Override
        public void log(String message) {
        }
    };
}

private String genererLogMessage(String message) {
    return String.format("%1$tY-%1$tb-%1$ta %1$tH:%1$tM %2$s - %3$s",
        LocalDateTime.now(), application, message);
}
}

```

Mais nous voulons appeler cette nouvelle méthode *log*. Ce nom rentrera en collision avec le nom de la méthode *log* de l'interface *Logger*. Il existe une syntaxe particulière qui permet de référencer explicitement le contexte de la classe englobante en utilisant :

`NomDeLaClasse.this`

Ainsi nous pouvons renommer notre méthode *genererLogMessage* en *log* et nous pouvons l'invoquer explicitement dans les méthodes des classes anonymes avec la syntaxe :

```
GenerateurLogger.this.log(message);
```

#### Note

Cette syntaxe permet d'accéder aux attributs et aux méthodes de l'instance de la classe englobante.

```
package dev.gayerie.logger;

import java.io.IOException;
import java.io.Writer;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.time.LocalDateTime;

public class GenerateurLogger {

    private String application;

    /**
     * @param application Le nom de l'application
     */
    public GenerateurLogger(String application) {
        this.application = application;
    }

    public Logger creerConsoleLogger() {
        return new Logger() {
            @Override
            public void log(String message) {
                // Pour le format du message utilisé dans printf
                // Cf.
                https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Formatter.html#syntax
                System.out.println(GenerateurLogger.this.log(message));
            }
        };
    }

    public Logger creerFileLogger(Path path) {
        return new Logger() {
            @Override
            public void log(String message) {
                // Pour le format du message utilisé dans printf
                // Cf.
                https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Formatter.html#syntax
                try(Writer w = Files.newBufferedWriter(path, StandardOpenOption.CREATE,
                    StandardOpenOption.APPEND)) {
                    w.append(GenerateurLogger.this.log(message)).append('\n');
                }
            }
        };
    }
}
```

```

        } catch (IOException e) {
            System.err.println(GenerateurLogger.this.log(message));
        }
    }
};
}

public Logger creerNoopLogger() {
    return new Logger() {
        @Override
        public void log(String message) {
        }
    };
}

private String log(String message) {
    return String.format("%1$tY-%1$tb-%1$ta %1$tH:%1$tM %2$s - %3$s",
        LocalDateTime.now(), application, message);
}
}

```

#### Note

Il n'est pas possible de déclarer un constructeur dans une classe anonyme. En effet, un constructeur porte le même nom que sa classe et justement, par définition, les classes anonymes n'ont pas de nom. Le compilateur générera néanmoins un constructeur par défaut.

Cela entraîne une limitation : il n'est pas possible de déclarer une classe anonyme qui étendrait une classe ne possédant pas de constructeur sans paramètre.

## Classe interne à une méthode

Il est possible de déclarer une classe dans une méthode. Dans ce cas, il n'est pas possible de préciser la portée de la classe. La classe a automatiquement une portée très particulière puisqu'elle n'est visible que depuis la méthode dans laquelle elle est déclarée. Une classe déclarée dans une méthode peut fonctionner de la même manière qu'une classe anonyme : elle peut accéder aux paramètres et aux variables de la méthode qui la déclare (à condition qu'ils ne soient modifiés ni par la méthode ni par la classe).

```

package dev.gayerie.logger;

import java.time.LocalDateTime;

public class GenerateurLogger {

    private String application;

    /**
     * @param application Le nom de l'application
     */
    public GenerateurLogger(String application) {
        this.application = application;
    }
}

```

```

public Logger creerConsoleLogger() {
    class ConsoleLogger implements Logger {
        @Override
        public void log(String message) {
            // Pour le format du message utilisé dans printf
            // cf.
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Formatter.html#syntax
            System.out.println(GenerateurLogger.this.log(message));
        }
    }
    return new ConsoleLogger();
}

private String log(String message) {
    return String.format("%1$tY-%1$tb-%1$ta %1$tH:%1$tM %2$s - %3$s",
        LocalDateTime.now(), application, message);
}
}

```

Dans l'exemple ci-dessus, la méthode *creerConsoleLogger* déclare la classe interne *ConsoleLogger*.

Note

Contrairement aux classes anonymes, une classe interne à une méthode peut déclarer des constructeurs.

## Interface et énumération

Il est possible de déclarer des interfaces et des énumérations dans une classe. Il est même possible de déclarer des interfaces et des énumérations dans une interface. Dans ce cas, les interfaces et les énumérations sont traitées implicitement comme **static**. On peut ou non préciser le mot-clé.

```

package dev.gayerie;

public class ClasseEnglobante {

    public interface InterfaceInterne {

    }

    public enum EnumerationInterne{VALEUR1, VALEUR2}

}

```

## Plusieurs classes dans un même fichier

Même s'il ne s'agit pas de classes internes, il est possible de déclarer plusieurs classes dans un même fichier en Java. Mais les classes supplémentaires sont forcément de portée package.

En pratique cette possibilité n'est jamais utilisée par les développeurs qui préfèrent utiliser des classes internes **static** ou un fichier propre à chaque classe.

# Les annotations

---

Les annotations en Java sont des marqueurs qui permettent d'ajouter des méta-données aux classes, aux méthodes, aux attributs, aux paramètres, aux variables, aux paquets ou aux annotations elles-mêmes.

Les annotations sont utilisées dans des domaines divers. Leur intérêt principal est de fournir une méta-information qui pourra être exploitée par un programme.

## Utilisation des annotations

---

Une annotation est un type (comme une classe ou une interface) du langage Java : elle peut être référencée par son nom complet ou importée depuis un autre paquet grâce au mot-clé **import**.

Une annotation n'est pas instanciée, elle est simplement accolée à l'élément qu'elle vient enrichir :

```
package dev.gayerie;

public class Voiture {

    @Override
    public String toString() {
        return "une voiture";
    }

}
```

L'annotation [Override](#) est définie dans le package [java.lang](#) (c'est pour cela qu'il n'est pas nécessaire de l'importer explicitement). Cette annotation est utilisable uniquement sur les méthodes pour indiquer que la méthode est une redéfinition d'une méthode d'une classe parente (dans l'exemple précédent, la méthode redéfinit [Object.toString](#)). Cette annotation est exploitée par le compilateur pour réaliser des vérifications supplémentaires. C'est également le cas pour les autres annotations déclarées dans le même package :

- [Deprecated](#)

Permet de générer des warnings afin d'informer les autres développeurs que quelque chose (une classe, une méthode...) a été dépréciée et ne devrait plus être utilisée.

- [FunctionalInterface](#)

Permet au compilateur de s'assurer que l'interface qui porte cette annotation peut être implémentée par une lambda (Cf. [le chapitre sur les lambdas](#)).

- [Override](#)

Signale qu'une méthode est une redéfinition d'une méthode déclarée dans une classe parente. Cela permet au compilateur de signaler une erreur si ce n'est pas le cas.

- [SuppressWarnings](#)

Permet de forcer le compilateur à ne plus émettre d'avertissement à la compilation dans certains cas.

- [SafeVarargs](#)

Cette annotation s'ajoute à une méthode acceptant un paramètre variable (*varargs*) dont le type est un générique. En effet, le principe de l'effacement de type (*type erasure*) dans la gestion des classes génériques fait qu'il est possible de corrompre un type paramétré utilisé comme paramètre variable sans que le compilateur et la JVM ne puissent le détecter. Pour pallier à ce problème, le compilateur produit systématiquement un avertissement lorsqu'on utilise un type générique comme paramètre variable. Cette annotation permet de supprimer l'avertissement à la compilation et implique que le développeur s'est assuré que son implémentation est sûre.

L'API standard de Java (mais également des bibliothèques tierces) fournissent beaucoup d'autres annotations qui ne sont pas interprétées par le compilateur mais par le programme lui-même à l'exécution.

Certaines annotations déclarent des attributs. Il est possible de spécifier entre parenthèses la valeur de chaque attribut d'une annotation. Par exemple, l'annotation [XmlRootElement](#) permet d'indiquer qu'une classe peut être instanciée à partir d'un document XML et/ou qu'une de ses instances peut servir à générer un document XML. Cette annotation accepte deux attributs optionnels : *name* pour donner le nom de l'élément XML correspondant et *namespace* pour donner l'espace de nom XML auquel l'élément appartient.

```
package dev.gayerie;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "personne", namespace="http://xml.personne.com/ns")
public class Personne {

    private String prenom;
    private String nom;

    // ...

}
```

Si un attribut est de type tableau alors, il est possible de passer plusieurs valeurs entre accolades :

```
@SuppressWarnings(value = { "deprecation", "unused" })
public void doSomething() {
    // ...
}
```

Mais si un attribut est de type tableau et que l'on veut fournir une seule valeur alors, les accolades peuvent être omises :

```
@SuppressWarnings(value = "unused")
public void doSomething() {
    // ...
}
```

Enfin, si l'attribut porte le nom spécial **value** et qu'il est le seul dont la valeur est donnée alors, il est possible d'omettre le nom :

```
@SuppressWarnings("unused")
public void doSomething() {
    // ...
}
```

## Déclaration d'une annotation

Comme pour les classes, les interfaces et les énumérations, on crée une annotation dans un fichier portant le même nom que l'annotation avec l'extension *.java*. On déclare une annotation avec le mot-clé **@interface**.

```
package dev.gayerie;

public @interface MyAnnotation {

}
```

Note

Une annotation implémente implicitement l'interface [Annotation](#) et rien d'autre !

La déclaration des attributs d'une annotation a une syntaxe très particulière :

```
package dev.gayerie;

public @interface MyAnnotation {
    String name();
    boolean isOk();
    int[] range() default {1, 2, 3};
}
```

Les attributs d'une annotation peuvent être uniquement :

- un type primitif,
- une chaîne de caractères ([java.lang.String](#)),
- une référence de classe ([java.lang.Class](#)),
- une Annotation ([java.lang.annotation.Annotation](#)),
- une [énumération](#),
- un tableau à une dimension d'un de ces types.

Le mot-clé **default** permet de spécifier une valeur d'attribut par défaut si aucune valeur n'est donnée pour cet attribut lors de l'utilisation de cette annotation.

La déclaration d'une annotation peut elle-même être annotée par :

- [Documented](#)  
Pour indiquer si l'annotation doit apparaître dans la documentation générée par un outil comme *javadoc*.
- [Inherited](#)  
Pour indiquer que l'annotation doit être héritée par la classe fille.
- [Retention](#)

Pour préciser le niveau de rétention de l'annotation (Cf. ci-dessous).

- [Target](#)

Pour indiquer quels types d'éléments peuvent utiliser l'annotation : classe, méthode, attribut...

- [Repeatable](#)

Pour indiquer qu'une annotation peut être déclarée plusieurs fois sur un même élément.

```
package dev.gayerie;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Documented
@Target(ElementType.TYPE)
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
    String name();
    boolean isOk();
    int[] range() default {1, 2, 3};
}
```

L'annotation ci-dessus porte des méta-annotations qui indiquent que l'utilisation de cette annotation doit apparaître dans la documentation générée, qu'elle est utilisable sur les types Java (c'est-à-dire les classes, les interfaces) et que sa rétention est de type *RUNTIME*.

## Rétention d'une annotation

Une annotation est définie par sa rétention, c'est-à-dire la façon dont une annotation sera conservée. La rétention est définie grâce à la méta-annotation [Retention](#). Les différentes rétentions d'annotation sont :

- **SOURCE**

L'annotation est accessible durant la compilation mais n'est pas intégrée dans le fichier class généré.

- **CLASS**

L'annotation est accessible durant la compilation, elle est intégrée dans le fichier class généré mais elle n'est pas chargée dans la JVM à l'exécution.

- **RUNTIME**

L'annotation est accessible durant la compilation, elle est intégrée dans le fichier class généré et elle est chargée dans la JVM à l'exécution. Elle est accessible par introspection.



# Utilisation des annotations par introspection

Une annotation ne produit aucun traitement. Cela signifie que si on utilise des annotations dans son code, encore faut-il qu'un processus les interprète pour produire le comportement attendu. Hormis les quelques annotations interprétées par le compilateur, il faut donc s'assurer que les annotations seront traitées correctement.

Pour des annotations de rétentions **SOURCE** et **CLASS**, leur interprétation dépend de processeurs d'annotations qui sont des bibliothèques Java déclarées en paramètre du compilateur ou de la JVM. Il s'agit d'une utilisation assez avancée et relativement peu utilisée (en dehors des annotations directement prises en charge par le compilateur lui-même).

Note

[Lombok](#) est un exemple de projet open-source fournissant des annotations permettant de générer du code au moment de la compilation grâce à un processeur d'annotations.

L'utilisation la plus courante (notamment avec Java EE) est l'utilisation d'annotation de rétention **RUNTIME** car elles sont accessibles par introspection.

Java fournit une API standard appelée l'API de réflexion qui permet de réaliser à l'exécution une introspection des objets et des classes. Cela signifie qu'il est possible de connaître par programmation tout un ensemble de méta-informations. Par exemple, on peut connaître la liste des méthodes d'une classe et pour chacune le nombre et le type de ses paramètres. Mais surtout, on peut connaître les annotations utilisées et la valeur de leurs attributs.

Imaginons que nous souhaitons créer une framework de tests automatisés. Nous pouvons créer l'annotation `@Test` qui servira à indiquer quelles méthodes publiques d'une classe correspondent à des tests à exécuter par notre framework.

```
package dev.gayerie.framework.test;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Documented
@Inherited
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
}
```

Comme la rétention de cette annotation est **RUNTIME**, il est possible d'accéder à cette annotation par introspection. Le framework de test peut contenir une classe `TestFramework` qui accepte une instance de n'importe quel type d'objet et qui va exécuter une à une les méthodes publiques ayant l'annotation `@Test`.

```
package dev.gayerie.framework.test;

import java.lang.reflect.InvocationTargetException;
```

```

import java.lang.reflect.Method;

public class TestFramework {

    public static void run(Object o) {
        Method[] methods = o.getClass().getMethods();
        for (Method method : methods) {
            if (method.isAnnotationPresent(Test.class)) {
                runTest(o, method);
            }
        }
    }

    private static void runTest(Object o, Method method) {
        try {
            method.invoke(o);
            System.out.println("Test " + method.getName() + " ok");
        } catch (InvocationTargetException e) {
            System.err.println("Test " + method.getName() + " ko");
            e.getTargetException().printStackTrace();
        } catch (Exception e) {
            System.err.println("Test " + method.getName() + " ko");
            e.printStackTrace();
        }
    }
}

```

Grâce à l'API de réflexion, il est possible d'accéder à la représentation objet d'une classe avec la méthode [getClass](#).

Finalement, nous pouvons écrire une pseudo-classe de tests :

```

package dev.gayerie;

import dev.gayerie.framework.test.Test;
import dev.gayerie.framework.test.TestFramework;

public class MesTests {

    @Test
    public void doRight() {
        // ...
    }

    @Test
    public void dowrong() throws Exception {
        // ...
        throw new Exception("simule un test en échec");
    }

    public static void main(String[] args) {
        TestFramework.run(new MesTests());
    }
}

```

# Exercice

## Exercice - Java et XML

Java propose une API nommée JAXB qui permet de transformer un document XML en objets Java ou de transformer des objets Java en document XML. JAXB fournit des annotations qui indiquent la correspondance en XML des classes et de leurs attributs.

Parmi ces annotations, il y a :

- [XmlRootElement](#)
- [XmlElement](#)
- [XmlAttribute](#)

Par ailleurs, JAXB permet de créer des instances de *Marshaller* et de *Unmarshaller*. Le premier type permet de passer de Java à un document XML et le second type réalise l'opération inverse.

Utilisation d'un Marshaller JAXB

```
package dev.gayerie.xml;

import java.io.File;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;

public class XmlMarshaller {

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

        JAXBContext ctx = JAXBContext.newInstance(Personne.class, Adresse.class);

        Marshaller marshaller = ctx.createMarshaller();
        marshaller.marshal(createPersonne(), new File("personne.xml"));
    }

    private static Personne createPersonne() {
        // TODO à implémenter
    }

}
```

Utilisation d'un Unmarshaller JAXB

```
package dev.gayerie.xml;

import java.io.File;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.Unmarshaller;

public class XmlUnmarshaller {

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

```

JAXBContext ctx = JAXBContext.newInstance(Personne.class, Adresse.class);

Unmarshaller unmarshaller = ctx.createUnmarshaller();
Personne personne = (Personne) unmarshaller.unmarshal(new
File("personne.xml"));

System.out.println(personne);
}

}

```

Étant donné le document XML suivant :

```

<?xml version="1.0" encoding="UTF-8"?>
<personne id="0001">
  <nom>Doe</nom>
  <prenom>John</prenom>
  <age>31</age>
  <adresse type="personnelle">
    <rue>9 allée des vignes</rue>
    <codePostal>33000</codePostal>
    <ville>Bordeaux</ville>
  </adresse>
</personne>

```

Créez les classes *Personne* et *Adresse* et utilisez les annotations JAXB de manière à pouvoir passer d’une représentation Java à une représentation XML (et vice-versa).

## Les modules

Depuis sa version 9, le langage Java supporte la notion de module. Un module est un ensemble de classes et de packages qui forment un tout complet. Il peut se présenter sous la forme d’un répertoire ou d’un fichier JAR. Un projet utilise généralement plusieurs modules.

Il ne faut pas confondre un module et un package. Un package représente simplement un espace de nom pour ranger des types Java (classes, interfaces, énumérations, annotations ou sous-packages). Les modules ont été introduits pour améliorer la sécurité et pour permettre une meilleure modularité de la plate-forme Java.

## Des modules pour améliorer la sécurité

Java supporte la notion de portée pour les attributs, les méthodes et les types. Ainsi, il est possible de déclarer des classes de portée *package* pour qu’elles ne soient pas accessibles par du code extérieur au package de déclaration. Cependant, il ne s’agit pas d’un mécanisme de sécurité mais plus d’une manière de cacher de l’information qui n’est pas considérée comme pertinente pour le reste de l’application. Avec l’API de réflexion, il est tout à fait possible d’accéder aux types normalement inaccessibles à la compilation.

Java utilise la notion de chemin de classes (*classpath*) pour déterminer la localisation des packages et classes à charger dans la machine virtuelle. Ce mécanisme souffre de plusieurs problèmes de sécurité, notamment pour les serveurs qui peuvent potentiellement charger du code malveillant embarqué dans les applications qu’ils hébergent.

Pour ces raisons, le développeur d’un module doit préciser :

- la liste des modules requis par son module
- éventuellement la liste des packages que le module exporte. C'est-à-dire les packages qui seront accessibles par les autres modules qui peuvent requérir ce module.

## Des modules pour plus de modularité

Au fur et à mesure des années, l'API standard Java n'a fait que s'enrichir, rendant l'environnement Java de plus en plus volumineux. Cela peut être un handicap pour les applications ciblant un environnement à faible ressource ou pour faciliter la distribution d'une application. Beaucoup de fonctionnalités disponibles dans l'API standard ne sont pas ou peu utilisées par les applications (support CORBA, Web Services, accès aux bases de données...).

Avec les modules, il est possible de préciser les modules de l'API standard qui sont réellement utiles à l'application. Ainsi, il est possible de créer des applications plus légères.

L'introduction des modules a demandé un important travail au niveau de l'API standard. Cette dernière a été découpée en plus de 50 modules. Le module principal se nomme `java.base` et il contient les principaux packages de l'API standard comme `java.lang` et `java.util`.

## Le descripteur de module (module-info.java)

Un module est défini par un fichier descripteur. Le fichier s'appelle obligatoirement `module-info.java`. Il est placé à la racine des sources du module. Ce fichier sera donc compilé pour produire le fichier `module-info.class` qui doit être distribué avec les autres fichiers du module.

Un descripteur de module indique le nom du module :

```
module dev.gayerie {  
}
```

Comme pour les packages, le nom d'un module peut contenir plusieurs identifiants séparés par `.`. On utilise généralement la même convention de nommage que pour les packages en utilisant un nom de domaine inversé. Il n'est pas obligatoire que le nom du module ait un rapport avec le nom des packages qu'il contient.

Un descripteur de module permet de préciser :

- les modules dont dépend le module
- les packages exposés par le module
- les services provenant d'autres modules et utilisés par le module
- les services fournis par le module.

## Dépendance à d'autres modules

L'instruction `requires` permet de nommer un module dont dépend le module.

```
module dev.gayerie {  
    requires java.sql;  
}
```

Dans l'exemple ci-dessus, on déclare que le module `dev.gayerie` a besoin du module `java.sql` pour fonctionner correctement. Cela signifie que le module `dev.gayerie` utilise des types Java publics ou protégés exposés par le module `java.sql`. Si par exemple, on souhaite créer un module qui utilise la classe `java.sql.Connection`, il faut obligatoirement indiquer que ce module dépend de `java.sql` pour que la classe `java.sql.Connection` soit disponible à la compilation et à l'exécution.

L'instruction `requires` peut s'accompagner des mots-clés `static` et `transitive`.

- *static*

Signifie que la dépendance n'est nécessaire qu'à la compilation et qu'elle est optionnelle à l'exécution. Si le module n'est pas disponible à l'exécution, le programme n'échouera pas.

- *transitive*

Implique que si un autre module est dépendant de notre module, il sera également dépendant du module requis. On parle de dépendance transitive.

```
module dev.gayerie {
    requires transitive java.sql;
    requires static java.compiler
}
```

Dans l'exemple précédent, le module `dev.gayerie` indique qu'il utilise des types Java des modules `java.sql` et `java.compiler`. Les types publics et protégés rendus disponibles par le module `java.sql` seront aussi disponibles pour tout module qui requière `dev.gayerie` (car on utilise le mot-clé `transitive`). Les types publics et protégés du module `java.compiler` ne sont requis qu'au moment de la compilation et sont optionnels à l'exécution (car on utilise le mot-clé `static`).

Le module `java.base` est requis par défaut, il n'est pas nécessaire de le préciser dans la liste des dépendances.

## Exposer des packages

L'instruction `exports` permet de préciser le nom d'un package. Tous les types Java de portée `public` ou `protected` seront exposés pour tous les modules qui requièrent ce module. De cette manière, il est possible de contrôler les types qui sont exposés.

```
module dev.gayerie {
    exports dev.gayerie.data;
    exports dev.gayerie.service;
}
```

L'instruction `opens` permet de préciser le nom d'un package. Tous les types Java de portée `public` ou `protected` seront exposés à l'exécution et pas à la compilation. Cela permet d'autoriser l'accès pour le code qui utilise l'API de réflexion Java pour accéder dynamiquement à des types lors de l'exécution.

```
module dev.gayerie {
    exports dev.gayerie.data;
    exports dev.gayerie.service;
    opens dev.gayerie.handler;
}
```

## Note

Il est possible de déclarer l'ensemble du module `open`. Cela signifie que tous ses packages sont ouverts par défaut en plus de ceux qui sont déclarés comme exportés.

```
open module dev.gayerie {  
    exports dev.gayerie.data;  
    exports dev.gayerie.service;  
}
```

Il est possible d'exporter ou d'ouvrir certains packages uniquement pour un module donné. Cela peut être utile pour fournir des classes d'implémentation à un *framework* contenu dans un autre module. On précise le nom du module grâce au mot-clé `to`.

```
module dev.gayerie {  
    exports dev.gayerie.data to dev.gayerie.dao;  
}
```

## Utiliser un service

Les services désignent des implémentations d'interface ou des spécialisations de classes abstraites fournies par des bibliothèques tierces. On parle de *Service Provider Interface*, abrégé en *SPI*. Un service est accessible à travers la classe [ServiceLoader](#). Pour pouvoir utiliser un service issu d'un autre module, il faut déclarer son utilisation dans le descripteur de module grâce à l'instruction `uses`. Par exemple, le module *java.base* fournit des spécialisations du service [FileSystemProvider](#). Pour pouvoir les charger, il faut déclarer l'utilisation de ce service dans le module.

```
module dev.gayerie {  
    uses java.nio.file.spi.FileSystemProvider;  
}
```

On peut ensuite charger le service grâce à la classe [ServiceLoader](#) :

```
ServiceLoader<FileSystemProvider> loader =  
    ServiceLoader.load(FileSystemProvider.class);  
Optional<FileSystemProvider> fsp = loader.findFirst();
```

Si l'utilisation du service n'est pas déclarée dans le descripteur de module, le code précédent échoue avec une [ServiceConfigurationError](#).

## Fournir un service

Un module peut fournir une implémentation d'un service accessible à travers la classe [ServiceLoader](#). Pour cela, il faut déclarer l'implémentation du service avec l'instruction `provides ... with`.

```
import java.nio.file.spi.FileSystemProvider;
import dev.gayerie.service.RemoteFileSystemProvider;

module dev.gayerie {
    provides FileSystemProvider with RemoteFileSystemProvider;
}
```

## Le module anonyme

---

Pour permettre une rétro-compatibilité avec le code Java antérieur à la version 9, la machine virtuelle Java doit considérer qu'un fichier JAR ou un chemin de classes qui ne fournit pas de descripteur de module *module-info.class* appartient à un module anonyme. Un module anonyme exporte et ouvre tous les packages qu'il définit.

## La notion de module path

---

Avant l'apparition des modules, les classes Java et les packages pouvaient être localisés dans différents répertoires. L'ensemble des répertoires que la machine virtuelle Java doit inspecter pour trouver une classe constitue le chemin des classes ou *classpath*.

Avec l'apparition des modules, le *classpath* est remplacé par le chemin des modules ou *module-path*. Le chemin des modules correspond à l'ensemble des répertoires et/ou des fichiers JAR qui désignent un module, c'est-à-dire qui fournissent le descripteur de module *module-info.class*.