```java
/**
 * This class demonstrates how to use the standard output in Java.
 *
 * @author Jean-Guy Mailly
 *
 */
public class Hello {

    /**
     * Name of the user
     */
    public static String name = null;

    /**
     * Name of the user if not provided from command line
     */
    public static final String STRANGER_NAME = "Stranger";

    /**
     * Entry point of the program.
     *
     * @param args the commandline parameters of the program
     */
    public static void main(String[] args) {
        System.out.println("Hello, world!");

        if(args.length == 1)
            name = args[0];

        System.out.println("This programm is run by " + ((name != null)? name : STRANGER_NAME));

    }

}

```

Programmation Avancée et Application

Licence Informatique

Auteur: Jean-Guy Mailly

Institut: UFR Mathématiques - Informatique, Université Paris Cité

Date: 9 octobre 2023

Version: 0.2.1

Université Paris Cité

Table des matières

Chapitre 1. Introduction à Java

Contenu

- *Bases de Java*
- *Types, variables, tableaux*
- *Instructions, boucles, conditions*
- *Entrées et sorties de base*
- *Premières classes*

1.1 Les bases de Java

1.1.1 Programmation et langages

Avant d'aborder les bases de Java à proprement parler, il convient de rappeler quelques notions au sujet des langages de programmation de façon plus générale. On peut classer les langages en fonction d'un « niveau » (illustrés en Figure 1.1), qui indique à quel point ce langage est proche du langage utilisé par la machine. Les processeurs de nos machines ne fonctionnent que via des séquences de 0 et de 1, qui ne sont pas conçues pour être comprises facilement par un humain. Il est donc particulièrement ardu à un programmeur de « parler » directement à la machine. Une première étape pour faciliter la programmation est l'*assembleur*, qui peut être vu comme une représentation du langage machine lisible par un humain. L'assembleur nécessite de gérer finement la façon dont le programme utilise la mémoire de la machine, et n'offre pas de structures de contrôle complexes, comme certaines conditions ou boucles. Les langages de bas niveau, comme le C, offrent beaucoup plus de possibilité au programmeur, mais nécessitent quand même de gérer la mémoire du programme. Au contraire, un lange de haut niveau (comme Python ou Java) permet au programmeur de se concentrer sur le problème à résoudre plutôt que sur les détails liés au fonctionnement de la machine.

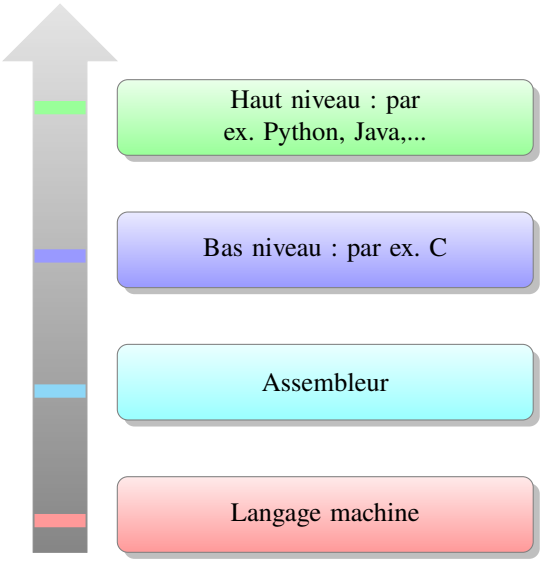


Figure 1.1 – Illustration des différents niveaux de langage.

Étant un langage de haut niveau, Java offre un certain nom d'avantages pour le programmeur. Tout d'abord, Java a été conçu pour réaliser des programmes en suivant le paradigme *orienté objet*. Nous reviendrons plus tard sur ce que cela signifie, mais on peut retenir que cela offre une possibilité de concevoir les programmes

de telle façon qu'ils seront plus faciles à écrire, à débuguer et à faire évoluer. De plus, un certain nombre de mécanisme complexes sont pris en charge directement par le langage : gestion automatisée de la mémoire, sérialisation, exceptions,. . . . La librairie standard de Java est très développée, elle fournit notamment les outils nécessaires à la gestion d'interfaces graphes, du réseau, des bases de données, . . . Java est également à la base du développement Android. Enfin, un dernier avantage intéressant est la syntaxe « à la C » : étant similaire sur un certain nombre de points aux langages C, C++, C#, Javascript, PHP,. . . , il est possible de passer aisément de Java à un de ces langages.

Une autre différence importante entre les langages de programmation est la façon dont on passe d'un code source écrit par le programmeur à l'exécution du programme. Comme l'ordinateur ne comprend que des séquences de 0 et de 1, il est nécessaire d'effectuer une « traduction ». Les deux moyens classiques sont la *compilation* et l'*interprétation*. La compilation est la transformation directe du code source en langage machine. Cela produit un fichier binaire, qui sera exécuté directement par le système d'exploitation de la machine (voir Figure 1.2). La compilation offre un certain nombre d'avantages, notamment le fait que le fichier exécutable est généralement optimisé pour le système d'exploitation et l'architecture de la machine. L'inconvénient associé est le fait qu'il est nécessaire de compiler sur chaque système, ce qui peut rendre le code plus difficile à porter sur différents systèmes. Parmi les langages compilés, on compte notamment C, C++, Cobol, Fortran et Pascal.

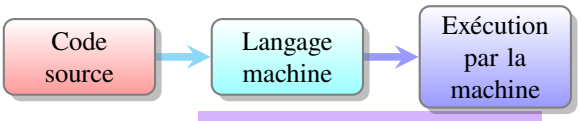


Figure 1.2 – Illustration de la compilation.

Au contraire, avec un langage interprété, un programme tier appelé *interpréteur* se charge directement de l'exécution du programme, sans passer par la production de code en langage machine (voir Figure 1.3). De tels langages auront généralement une efficacité moindre, en comparaison des langages compilés, mais offriront une meilleure portabilité des programmes (il suffit d'avoir un interpréteur disponible pour un système cible, pour que le programme puisse fonctionner sur ce système). Parmi les langages interprétés les plus classiques, on retrouve Bash, Python ou Prolog.

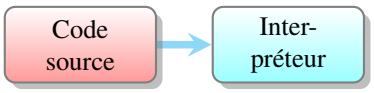


Figure 1.3 – Illustration de l'interprétation.

Java peut être vu comme un langage intermédiaire. Comme on peut le voir sur la Figure 1.4, le code Java est compilé en byte code, un langage non compréhensible par le programmeur, mais différent du langage machine. Ce byte code n'est pas directement exécutable par le système d'exploitation, on a besoin d'une machine virtuelle Java (c'est-à-dire, un interpréteur). Cela rend Java très portable, puisqu'il existe des machines virtuelles Java pour de nombreux systèmes d'exploitation, et donc le même byte code peut être utilisé directement sur différentes machines.

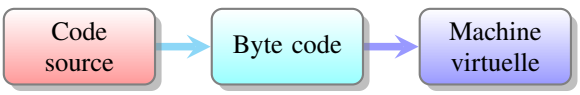


Figure 1.4 – Production et interprétation de bytecode.

1.1.2 Programme Java

Le langage Java est régulièrement mis à jour, ajoutant de nouvelles possibilités, tout en maintenant (autant que possible) une compatibilité avec le code plus ancien. La version utilisée dans ce cours sera Java SE version 17 (sortie en septembre 2021, qui est la version disponible sur les machines de l'UFR.) Java SE est distribué sous deux formes :

- Java Runtime Environment (JRE) : composé d'une machine virtuelle Java, et de l'ensemble de la bibliothèque standard, le JRE est nécessaire pour exécuter un programme Java
- Java Development Kit (JDK) : il s'agit du JRE auquel on ajoute (notamment) un compilateur, ce qui est nécessaire pour développer un programme Java.

Par bibliothèque standard, on entend l'ensemble des classes et fonctionnalités fournies par les développeurs de Java, et qui peuvent être utilisées pour développer vos propres applications. On parle aussi d'API standard de Java, pour *application programming interface*. Concernant la machine virtuelle Java, on parlera dans la suite de JVM (*Java Virtual Machine*).

Concrètement, un programme écrit en Java est un ensemble de *classes* qui sont :

- des classes fournies par l'API standard,
- des classes fournies par d'autres API,
- des classes écrites par le développeur du programme.

Grossièrement, une classe est une structure de données à laquelle on ajoute un ensemble de fonctions pour manipuler ces données. Nous verrons ce qu'est une classe plus en détail dans la suite.

Un programme Java peut être distribué assez facilement : dès lors qu'il existe une JVM pour un système d'exploitation, alors elle est associée aux classes de l'API standard. Il suffit donc de distribuer les classes écrites par le développeur (et éventuellement les classes des API tierces). Cela peut être fait via deux moyens, à savoir la distribution des fichiers compilés en byte code (des fichiers portant l'extension `.class`), ou la distribution d'une archive `.jar` qui contient les classes (cette méthode est préférable à la première).

1.1.3 Compilation et exécution d'un programme Java

Nous décrivons maintenant brièvement les bases de la compilation et de l'exécution d'une programma Java via la ligne de commande.

1.1.3.1 Cas de base : une ou plusieurs classes dans le même répertoire

On suppose ici que le terminal est situé dans le répertoire où se situe le fichier `ClasseA.java` ; sinon :

```
$ cd /chemin/vers/le/repertoire/du/code
$ ls
ClasseA.java
```

Voici la commande de base pour compiler une classe Java :

```
$ javac ClasseA.java
$ ls
ClasseA.class ClasseA.java
```

La classe `ClasseA` décrit un programme qui se contente d'afficher un message. Après la compilation, on l'exécute via :

```
$ java ClasseA
Hello, World!
```

La commande `java` s'appelle avec le nom de la classe, pas le nom d'un fichier (ici, juste `ClasseA` au lieu de `ClasseA.class`). Dans le cas contraire :

```
$ java ClasseA.class
Erreur :impossible de trouver ou de charger la classe
principale ClasseA.class
Cause par :java.lang.ClassNotFoundException:
ClasseA.class
```

Supposons qu'on a maintenant plusieurs classes :

```
$ ls
ClasseA.java ClasseB.java
```

On peut les compiler via :

```
$ javac *.java
$ ls
ClasseA.class ClasseA.java ClasseB.class ClasseB.java
```

1.1.3.2 Projets et arborescence des fichiers

Un projet Java est normalement subdivisé en plusieurs répertoires :

- Un répertoire pour les fichiers sources `.java`
- Un répertoire pour les fichiers compilés `.class`
- D'autres répertoires peuvent apparaître (on en reparlera plus tard. . .)

Dans notre exemple, on doit avoir :

```
repertoirePrincipal
├── src/
│   ├── ClasseA.java
│   └── ClasseB.java
└── bin/
    ├── ClasseA.class
    └── ClasseB.class
```

Pour compiler un projet complet, plutôt que de compiler dans le répertoire `src`, puis de déplacer les fichiers `.class` dans `bin`, on demande au compilateur de le faire depuis le répertoire principal :

```
$ ls
bin/ src/
$ javac -d bin src/*.java
$ ls -R
bin/ src/
./bin:
ClasseA.class ClasseB.class
./src:
ClasseA.java ClasseB.java
```

L'option `-d` de la commande `javac` permet d'indiquer le répertoire (en anglais *directory*) qui doit recevoir les fichiers compilés.

Il est possible d'utiliser des classes externes, on doit donc le préciser au compilateur. Par exemple, si `ClassA` a besoin de classes qui sont dans le répertoire `lib/` :

```
$ ls
bin/ lib/ src/
$ javac -d bin -classpath lib src/*.java
```

On peut utiliser la forme raccourcie de l'option :

```
$ javac -d bin -cp lib src/*.java
```

Dans le cas où il y a plusieurs sources externes, on utilise deux points (`:`) pour les séparer :

```
$ ls
bin/ lib/ src/
$ javac -d bin -cp lib1:lib2:lib3 src/*.java
```

On peut également utiliser une archive `.jar` :

```
$ javac -d bin -cp lib:MonArchive.jar src/*.java
```

De la même manière, il faut préciser à l'exécution qu'on utilise des classes externes si elles ne sont pas dans le répertoire courant. On indique à la commande `java` quel est le `classpath` :

```
$ ls -R
bin/ lib/ src/
./bin:
ClasseA.class ClasseB.class
./lib:
ClasseC.class
./src:
ClasseA.java ClasseB.java
$ java -cp bin:lib ClasseA
Hello, World!
Utilisation de la ClasseC.
```

Remarque : les classes de l'API standard sont automatiquement dans le `classepath`, il n'est pas nécessaire de les indiquer.

1.1.3.3 Premiers programmes simples

Le premier programme classique : **Hello, World !**

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

Nous reviendrons en détail sur les mots-clés utilisés par la suite. Précisons tout de même que `System.out.println` permet d'afficher un message sur la sortie standard. Le programme ci-dessus définit une classe qui s'appelle

`Hello`. Elle doit être définie dans un fichier `Hello.java`, dont la compilation provoque la création d'un fichier `Hello.class`.

Tout programme Java a un unique point d'entrée, qui est la méthode

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

Sans rentrer dans les détails pour l'instant, sachez que `String` est une classe de l'API qui permet de représenter des chaînes de caractères. `String[] args` est donc un tableau de `String` représentant les paramètres du programme sur la ligne de commande. [1] `void` signifie que la méthode ne renvoie aucune valeur. Concernant les mots clés `public` et `static` : nous verrons cela plus tard.

1.1.3.4 Organisation du projet en packages

Le code source d'un programma Java doit être découpé en packages. Chaque package réunit un ensemble de classes qui ont un lien « logique » entre elles. Par exemple, un package peut servir à réunir les classes liées au fonctionnement de l'interface graphique, un autre pour les classes qui permettent la gestion de la base de données, un troisième peut servir à gérer l'intelligence artificielle présente dans le programme,... D'un point de vue « physique », sur le disque dur, cela se manifeste par la création de répertoires équivalents aux packages dans le répertoire `src`.

```
repertoirePrincipal
├── src/
│   ├── gui
│   ├── database
│   └── ai
└── bin/
```

Les fichiers dans lesquels sont définies les classes sont placés dans les répertoires correspondant aux packages. Une instruction en début de fichier indique le package auquel appartient la classe.

```
repertoirePrincipal
├── src/
│   └── up/
│       └── mi/
│           └── jgm/
│               └── Hello.java
├── bin/
│   └── up/
│       └── mi/
│           └── jgm/
└── lib/
```

```
package up.mi.jgm ;

public class Hello {
   ...
```

1. Nous donnerons plus de détails sur l'utilisation des tableaux plus tard.

```
}
```

Il est nécessaire d'indiquer au compilateur comment tenir compte des packages lors de la compilation. Depuis le répertoire principal :

```
javac -d bin/up/mi/jgm/
      -cp src:lib
      src/up/mi/jgm/*.java
```

Ça peut être très fastidieux de procéder ainsi s'il y a de nombreux packages. La compilation de projets complexes peut cependant être automatisée via un `Makefile`, ou d'autres techniques (Maven, ant). Dans notre cas, l'utilisation d'Eclipse permettra de simplifier la compilation et l'exécution.

Pour distribuer un projet Java, la méthode habituelle consiste à utiliser une archive `jar` Une telle archive est un conteneur dédié au langage Java. Elle peut être utilisée pour transmettre des classes, sous forme de fichiers `.class` et/ou `.java`. Un `jar` peut aussi servir à transmettre un programme fonctionnel (on parle de `jar` exécutable, ou *runnable* `jar` *file* en anglais). Les commandes de bases pour la manipulation d'un `jar` sont les suivantes :

- Création de l'archive :
  `jar cf JARFILE INPUTFILES`
- Liste du contenu de l'archive :
  `jar tf JARFILE INPUTFILES`
- Extraction du contenu de l'archive :
  `jar xf JARFILE INPUTFILES`

Il ne pas oublier d'ajouter le `jar` au classpath avec l'option `-cp` lorsqu'il contient des classes externes utilisées comme ressources dans votre projet. Concernant les `jar` exécutables, on peut les créer en indiquant la classe principale du programme (celle qui contient la méthode `main`). Les commandes de base sont :

- Création de l'archive :
  `jar cvfe JARFILE MAINCLASS INPUTFILES`
- Exécution de l'archive :
  `java -jar JARFILE`

Cependant, la plupart des démarches que nous avons présentées peuvent être simplifiées grâce à l'utilisation d'un environnement de développement intégré (EDI, ou IDE en anglais) qui comprend :

- l'éditeur de texte « intelligent »,
- le compilateur,
- l'outil de création de `jar`,
- une interface simplifiée pour la gestion de packages,
- des outils de débugage,
- plein d'autres choses...

Nous verrons en TP comment utiliser l'IDE Eclipse.

1.1.4 Nommage des classes

Le nom complet d'une classe contient la spécification détaillée du package auquel elle appartient, par exemple `up.mi.jgm.Hello`. Il est normalement nécessaire de le préciser lors de l'utilisation de la classe, de la façon suivante :

```
public static void main(String[] args){
  up.mi.jgm.A monObjet = new up.mi.jgm.A();
}
```

Afin de simplifier l'utilisation des classes, on peut faire appel au mot-clé `import` qui permet de préciser une unique fois le nom complet d'une classe :

```
import up.mi.jgm.A ;

public static void main(String[] args){
  A monObjet = new A();
}
```

Une fois que l'`import` est fait, l'utilisation du nom simple `A` fera toujours référence à la même classe `up.mi.jgm.A`. Il n'est pas nécessaire d'importer les classes du package courant, ni celles du package `java.lang` de l'API standard (classes de base).

Il est possible que plusieurs classes définies dans différents packages aient le même nom. Dans ce cas, une seule (au plus) peut être importée, il faut alors utiliser les noms complets pour certaines d'entre elles :

```
import up.mi.jgm.A ;

public static void main(String[] args){
  A monObjet = new A();
  mon.autre.classe.A autre = new mon.autre.classe.A();
}
```

Il faut être particulièrement prudent avec les classes homonymes. En particulier, l'utilisation d'outils comme Eclipse, qui servent à faciliter et accélérer le travail du programmeur, peut induire des erreurs d'inattention. En effet, Eclipse permet d'importer automatique des classes qui sont utilisées dans votre code. S'il y a une homonymie, il vous demande de choisir laquelle des classes proposées est la bonne. Si vous n'êtes pas attentifs, vous risquez d'importer la mauvaise classe, et d'avoir un code incorrect sans vous en rendre compte ! Par exemple, de nombreuses classes de JavaFX portent le même nom que des classes d'AWT (une API dédiée aux interfaces graphiques que nous n'utiliserons pas dans ce cours). Cela peut donc être source d'une code qui semble correct à première vue, mais qui ne compilera pas.

1.2 Types, variables, tableaux

1.2.1 Types primitifs

En plus de types d'objets (nous y reviendrons plus tard), Java propose des types de données primitifs, pour représenter :

- les nombres entiers,
- les nombres décimaux,
- les caractères,
- les booléens.

Il existe plusieurs types primitifs permettant la représentation de nombres entiers. La principale différence est le nombre d'octets utilisés pour stocker le nombre dans la mémoire de l'ordinateur. En fonction de l'espace

utilisé en mémoire, l'ensemble de nombre qui peut être représenté est plus ou moins étendu (voir Table 1.1).

Type	Valeur min	Valeur max	Taille mémoire
`byte`	$-2^7 = -128$	$2^7 - 1 = 127$	1 octet
`short`	$-2^{15} = -32768$	$2^{15} - 1 = 32767$	2 octets
`int`	$-2^{31} \simeq -2 \times 10^9$	$2^{31} - 1 \simeq 2 \times 10^9$	4 octets
`long`	$-2^{63} \simeq 9 \times 10^{18}$	$2^{63} - 1 \simeq 9 \times 10^{18}$	8 octets

Table 1.1 – Types primitifs pour les nombres entiers

Pour les nombres décimaux, il existe deux types différents. Dans les deux cas, les nombres décimaux sont représentés sous la forme $x = s \times m \times 2^e$ où

- s est le signe, représenté par un bit,
- m est la mantisse,
- e est l'exposant.

Le nombre de bits pour représenter m et e en mémoire varient selon le type (voir Table 1.2).

Type	m	e	Taille mémoire
`float`	23 bits	8 bits	4 octets
`double`	52 bits	11 bits	8 octets

Table 1.2 – Types primitifs pour les nombres décimaux

Il existe un type qui permet de représenter deux valeurs de vérité constantes : `boolean`. Ces constantes sont `true` et `false`. L'espace mémoire utilisé pour représenter un booléen dépend de la JVM. Il est important de noter que contrairement à d'autres langages de programmation (comme le C), il n'y a pas de conversion automatique des entiers en booléens.

Enfin, le dernier type primitif sert à représenter des caractères. Ce type `char` utilise le standard unicode sur 2 octets :

- de 0 (`\u0000`) à 127 (`\u007f`) : identique aux codes ASCII,
- de 128 (`\u0080`) à 255 (`\uffff`) : codes Latin-1.

Il existe un certain nombre de caractères spéciaux (voir Table 1.3).

Signification	Caractère spécial
Retour à la ligne	`\n`
Tabulation	`\t`
Apostrophe	`\'`
Double apostrophe	`\"`
Backslash	`\\`
Caractère Unicode	`\uxxxx`

Table 1.3 – Liste des caractères spéciaux

1.2.2 Variables

Il est possible d'utiliser des variables pour stocker une valeur, par exemple le résultat d'un calcul, et la réutiliser plus tard durant l'exécution du programme.

1.2.2.1 Déclaration et affectation

De manière générale, la syntaxe pour déclarer une variable est très simple : `type nomVariable ;`
Par exemple,

- `float x ;`
- `int n ;`
- `boolean b ;`
- `char c ;`

Ces variables sont déclarées mais ne sont pas affectées : aucune valeur définie n'est stockée dans la mémoire allouée à ces variables. Mais maintenant que les variables sont déclarées, elles peuvent être utilisées dans la suite du programme, et donc recevoir une valeur.

Cela veut dire qu'on peut faire l'affectation en deux temps (déclaration, puis affectation) :

```
float x ;
x = 5.2 ;
```

On peut également affectater une valeur à la variable dès sa déclaration :

```
float x = 5.2 :
int n = 5 ;
float y = n ;
```

La dernière instruction fait une conversion automatique de l'entier 5 vers le décimal 5.0, c'est-à-dire du type plus restreint de la valeur, vers le type plus général de la variable qui doit convenir cette valeur.

On peut déclarer et affecter plusieurs variables de même type en même temps :

- `int a,b, c ;`
- `int a = 2,b = 3,c = 4 ;`

1.2.2.2 Conversions

Une conversion implicite (automatique) est faite lorsqu'on passe d'un type « plus particulier » à un type « plus général » :

1. `byte` (plus particulier),
2. `short`,
3. `int`,
4. `long`,
5. `float`,
6. `double` (plus général).

Exemple 1.1 Voici des exemples de conversions implicites.

```
short s = 6 ;
int i = 8 ;
double d = s ;// conversion implicite de 6 en 6.0
long l = s * i ;// convertit s et i en long pour faire la multiplication
```

Il y a également des conversions implicites de `char` vers `int`, `long`, `double` et `float`.

Si la conversion implicite est impossible, on utilise la syntaxe `type var1 = (type)var2` où `type` est le type de la variable `var1`.

Exemple 1.2 Voici des exemples de conversions explicites.

```
double d = -4.3 ;
float f = (float) d ;
int i = (int) d ;// i vaut -4
```

Rappelons qu'il n'y a pas de conversion entre nombres et booléens.

1.2.2.3 Opérations simples

1.2.2.3.1 Opérateurs mathématiques Rappelons la syntaxe pour effectuer des opérations mathématiques :

- `+`, `-` et `*` sont utilisés pour l'addition, la soustraction et la multiplication habituelles,
- division versus modulo :
  - `/` est utilisé pour la division ; si les deux opérandes sont des entiers, le résultat est un entier (division euclidienne). Par exemple, `5 / 2` donne 2. Dans le cas contraire, c'est la division en nombres réels qui est effectuée. Par exemple `5.2 / 2` donne 2.6.
  - `%` est le modulo (reste de la division euclidienne). Par exemple, `5 % 2` donne 1.

Ces opérations peuvent être utilisées dans des variantes de l'affectation :

- `=` est l'affectation « classique »,
- `x += y` est un raccourci pour `x = x + y`,
  - Cela fonctionne également avec `-=`, `*=`, `/=`, `%=`.
- `x++` est l'incrémentation (`x = x + 1`),
- `x--` est la décrémentation (`x = x - 1`).

1.2.2.3.2 Comparaisons et opérateurs booléens Les opérateurs de comparaison servent à comparer deux valeurs entre elles, et retournent une valeur booléennes qui correspond au résultat du test :

- `x == y` vérifie si la valeur de `x` est identique à celle de `y`.
  - Similaire pour la différence (`!=`), supérieur strict (`>`), supérieur ou égal (`>=`), inférieur strict (`<`) et inférieur ou égal (`<=`).
- Ne pas confondre affectation et égalité !

  `x = 5 ;` donne la valeur 5 à la variable x

  `(x == 5)` vérifie si la valeur de x est 5

Les opérateurs booleéns servent à combiner plusieurs valeurs booléennes en une seule (voir Table 1.4).

Symbole	Nom	Exemple	Effet
`!`	Négation	`!(x == y)`	Inverse la valeur booléenne
`&&`	Et	`(x > y) && (x > z)`	Indique si toutes les valeurs sont `true`
`||`	Ou	`(x > y) || (x > z)`	Indique si au moins une valeur est `true`

Table 1.4 – Opérateurs booléens en Java

En logique, on parle généralement de conjonction (**et**) et de disjonction (**ou**). Ces deux opérateurs sont « court-circuit », c'est-à-dire que lors de l'exécution, on évalue uniquement la première moitié de l'expression si c'est possible. Par exemple, pour que `x && y` soit vrai, il faut que les deux valeurs booléennes soient vraies. Si `x` est faux, il n'y a pas besoin d'évaluer `y` pour savoir que l'expression globale est fausse.

Notons l'existence d'opérateurs qui ressemblent aux opérateurs logiques, mais dont l'utilisation est légèrement différente : il s'agit des opérateurs bit à bit (voir Table 1.5). Il est probable que nous utiliserons assez peu

ces opérateurs, ainsi que les opérateurs de décalage de bits (ligne 5 de la Table 1.6).

Symbole	Nom	Exemple	Effet
`&`	Et	`4 & 7`	Retourne 1 si les bits de même poids sont à 1
`|`	Ou	`4 | 8`	Retourne 1 si au moins un des bits de même poids est à 1
`^`	Ou exclusif	`1 ^ 2`	Retourne 1 si les bits de même poids sont de valeurs différentes

Table 1.5 – Opérateurs bit à bit en Java

La Table 1.6 indique la priorité des opérateurs. En cas de priorité identique, on exécute de gauche à droite. En cas de doute, il vaut mieux utiliser des parenthèses pour clarifier les expressions.

Priorité	Opérateurs
1	`[] ()`
2	`++ -- ! ~ instanceof`
3	`* / %`
4	`+ -`
5	`<< >> >>>`
6	`< > <= >=`
7	`== !=`
8	`&`
9	`^`
10	`|`
11	`&&`
12	`||`
13	`?:`
14	`= += -= %= *= /=`

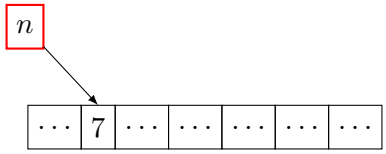
Table 1.6 – Priorité des opérateurs en Java

1.2.2.4 Affectations et passages de paramètres pour les types primitifs

En Java, les valeurs des types primitifs sont recopiées lors des affectations (et des passages de paramètres des méthodes). Voyons cela sur un exemple.

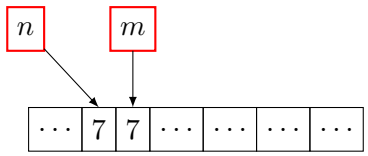
Exemple 1.3 Supposons qu'on initialise une variable `n` de type `int`, avec la valeur 7. Dans la mémoire du programme, cela correspond à une case qui contient l'encodage en binaire du nombre 7, et qui est lue à chaque fois que le programme utilise la valeur de `n`.

```
int n = 7 ;
```



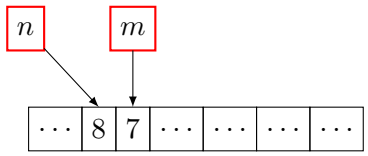
Supposons maintenant qu'on crée une deuxième variable `m` qui reçoit la valeur de `n` : cela crée une deuxième zone qui contiendra la valeur 7.

```
int n = 7 ;
int m = n ;
```



Ainsi, lorsqu'on modifie `n`, cela ne modifie pas `m`.

```
int n = 7 ;
int m = n ;
n = n + 1 ;
```



C'est le même fonctionnement pour les paramètres de méthodes : la valeur de la variable passée en paramètre n'est pas modifiée hors de l'exécution de la méthode. Nous parlerons plus tard du fonctionnement pour les objets.

1.2.3 Tableaux

Les tableaux permettent de manipuler des collections ordonnées de valeurs du même type. Cela signifie que chaque valeur est identifiée par sa position dans le tableau, on parle d'*indice*. Pour un tableau à n éléments, les indices vont de 0 à $n-1$. La déclaration et l'initialisation d'une variable de type tableau se font avec la syntaxe suivante : `type [] nom = new type[longueur]`. Par exemple :

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

initialise un tableau de 5 éléments de type `int`. Une fois le tableau déclaré, on peut affecter une valeur à chaque indice possible :

```
tab[0] = 1 ;
tab[1] = 2 ;
tab[2] = 3 ;
tab[3] = 4 ;
tab[4] = 5 ;
```

On peut également initialiser les éléments lors de l'initialisation du tableau :

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

Si des valeurs ne sont pas spécifiées lors de son initialisation, les éléments du tableau prennent une valeur par défaut, comme décrit par la Table 1.7. [2]

Les tableaux sont des objets qui disposent d'un attribut `length` qui indique leur longueur, c'est-à-dire le nombre d'élément. Par exemple, avec le tableau initialisé précédemment, `tab.length` vaut 5. Nous verrons plus tard quelques manipulations et algorithmes de bases sur les tableaux.

Il est impossible de modifier la longueur d'un tableau, `length` est un attribut constant.

2. Notons que ce sont les mêmes valeurs qui servent lors de la déclaration d'une variable non initialisée.

Type	Valeur
`boolean`	`false`
`char`	\u0000
`byte`, `short`, `int`, `long`	0
`float`, `double`	0.0
types objets	`null`

Table 1.7 – Valeurs par défaut

1.3 Instructions, boucles, conditions

1.3.1 Instructions et blocs

Une instruction « simple » est une déclaration, une affectation, une incrémentation ou l'appel d'une méthode (on reviendra plus tard sur les différents types de méthodes et leur utilisation). Toute instruction simple se termine par un point-virgule `;`.

Exemple 1.4 Deux instructions simples

```
int n = 0 ;
System.out.println("n␣=␣" + n) ;
```

Un bloc est une séquence d'instructions réunies entre accolades. Les instructions du bloc peuvent être

- des instructions simples,
- des blocs,
- des boucles,
- des conditions.

Exemple 1.5 Voici un exemple de bloc composé de deux instructions simples et d'un autre bloc :

```
{
int n = 0 ;
System.out.println("n␣=␣" + n) ;
  {
    // Contenu du bloc...
  }
}
```

Une variable est utilisable dans le bloc où se situe sa déclaration, depuis l'endroit de la déclaration jusque la fin du bloc en question. Elle est également accessible dans les blocs internes au bloc de sa déclaration.

Exemple 1.6

```
1 {
2 int n = 0 ;
3  {
4    System.out
5         .println("n␣=␣" + n);
6  }
7 }
```

La variable `n` est accessible pendant toute l'exécution du bloc : lignes 2 à 7.

Il existe des cas d'homonymie de variables : une variable locale à une méthode peut porter le même nom qu'une variable de classe (c'est-à-dire un attribut). Nous reparlerons de cela dans la partie dédiée aux classes, où nous verrons comment gérer les éventuelles ambiguités qui peuvent être liées à cela.

Bien entendu, deux variables locales contenues dans des méthodes différentes peuvent également porter le même nom. Par contre, deux variables locales à la même méthode ne peuvent pas porter le même nom, même si elles sont définies dans des blocs différents.

Exemple 1.7 Si on essaye de compiler le code suivant :

```
public class Test {
  public static void main(String[] args){
    int n = 0 ;
    {
      int n = 1 ;
      System.out.println("n␣=␣" + n);
    }
  }
}
```

on obtient cette erreur :

```
Test.java:5: error: variable n is already defined in
method main(String[])
    int n = 1 ;
        ^
1 error
```

1.3.2 Boucles

Une boucle est une instruction spéciale qui permet de répéter d'un bloc d'instructions. Il existe différents types de boucles, qui ont un fonctionnement légèrement différent.

1.3.2.1 Boucles while

Une boucle `while` (« tant que ») est la répétition d'un bloc d'instructions tant qu'une certaine condition est vraie. Cette condition peut être la valeur d'une variable booléenne, le résultat d'une comparaison, ou la valeur de retour d'une méthode. La condition est vérifiée avant l'exécution du bloc d'instructions. On appelle chaque répétition du bloc d'instructions une *itération*. Voici quelques exemples pour en illustrer la syntaxe et le fonctionnement :

Exemple 1.8

```
boolean b = true ;
while(b){
  // instructions a repeter
}
int n = 0 ;
while(n < 10){
  // instructions a repeter
}
```

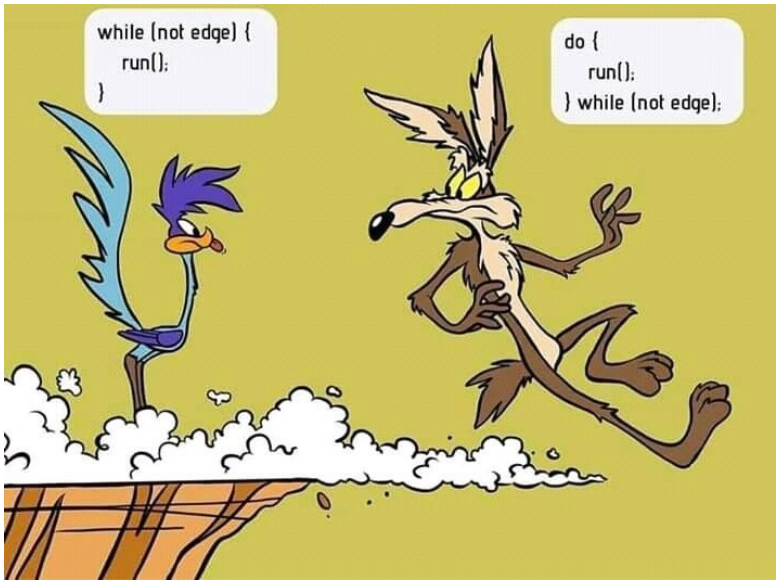


FIGURE 1.5 – Illustration humoristique de la différence entre while et do-while

Il n'y a pas de point-virgule après la condition de boucle !

```
boolean b = true ;
while(b);{
  // boucle infinie
  // ces instructions ne sont pas executees
}
```

Si la condition est fausse avant d'entrer dans la boucle, il n'y a pas d'itération.

```
boolean b = false ;
while(b){
  // ces instructions ne sont pas executees
}
// ces instructions sont executees
```

1.3.2.2 Boucles do-while

La boucle `do while` est une variante du `while` : la condition est vérifiée après l'exécution du bloc d'instructions. Le bloc d'instructions est donc exécuté au moins une fois, même si la condition est fausse.

Exemple 1.9

```
boolean b = true ;
do{
  // instructions a repeter
}while(b);
```

Attention de ne pas oublier le point-virgule après le `do-while`.

La Figure 1.5 présente une description humoristique de la différence entre les deux types de boucles que nous venons de voir.

1.3.2.3 Boucles for

La boucle `for` a globalement un fonctionnement similaire à la boucle `while` : elle teste une certaine condition, et répète un bloc d'instructions tant que cette condition est vraie. Cependant, elle a une syntaxe différente, avec trois parties distinctes :

- la partie `initialisation` est exécutée une seule fois, avant la boucle,
- la `condition` est vérifiée avant chaque exécution du bloc d'instructions,
- la `modification` est réalisée après chaque exécution du bloc d'instructions.

```
for(initialisation ;condition ;modification){
  // instructions a repeter
}
```

Exemple 1.10 Cette boucle `for` :

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

est équivalente à cette boucle `while` :

```
int i = 0 ;
while(i < 10){
  // instructions a repeter
  i++ ;
}
```

Dans les deux cas, les instructions seront répétées 10 fois.

1.3.2.4 Boucles for each

Le mot clé `for` peut être utilisé avec une autre syntaxe, afin de « simuler » une boucle `for each`. Cette variante du `for` est utilisée pour parcourir les éléments d'un tableau ou d'une collection. [3]

```
type[] tab = { ... } ;
for (type var :tab){
    // Instructions
}
```

Exemple 1.11

```
int[] tab = {1, 2, 3, 4} ;
int somme = 0 ;
for (int val :tab){
    somme += val ;
}
System.out.println("La somme des elements est " + somme);
```

3. On reviendra plus tard sur les collections.

1.3.2.5 Les débranchements : break et continue

Dans le cadre des boucles, le mot clé `break` permet de sortir d'une boucle et de passer à l'exécution des instructions situées après la boucle, tandis que le mot clé `continue` permet de passer à l'itération suivante d'une boucle.

1.3.3 Les conditions

1.3.3.1 if et else

Le mot-clé `if` permet d'exécuter un bloc d'instructions seulement si une condition est vérifiée. Il peut être suivi (de manière optionnelle) d'un bloc `else` qui précise les instructions à exécuter si la condition est fausse. Le bloc `else` peut lui-même commencer par un `if`, etc

Exemple 1.12 Voici un exemple d'utilisation des mots-clés `if` et `else`.

```
int age = ... ;
if (age < 10){
  System.out.println("Vous␣etes␣un␣enfant.");
} else if (age < 18){
  System.out.println("Vous␣etes␣un␣ado.");
} else if (age < 30){
  System.out.println("Vous␣etes␣un␣jeune␣adulte.");
} else if (age < 65){
  System.out.println("Vous␣etes␣un␣adulte.");
} else {
  System.out.println("C'est␣sympa,␣la␣retraite␣?");
}
```

1.3.3.2 switch case

On peut choisir un bloc d'instructions à effectuer en fonction de la valeur d'une expression avec le mot-clé `switch`. L'expression peut être :

- de type primitif `byte`, `short`, `char`, `int`,
- de type enuméré (nous reviendrons sur les `Enum` plus tard),
- de type `String` (nous reviendrons sur les `String` plus tard),
- de classes qui encapsulent certains types primitifs `Character`, `Byte`, `Short`, and `Integer` (...plus tard !).

Pour indiquer les valeurs possibles pour l'expression, on utilise le mot-clé `case`.

```
switch(expression) {
 case val1 :
  instruction1;
  instruction2;
  ...
  break;
 case val2 :
  ...
 default :
```

```
  ...
}
```

Sans le mot clé `break` à la fin d'un `case`, le cas `val2` sera effectué à la suite du cas `val1`. Le cas `default` (optionnel) permet de traiter toutes les valeurs qui ne sont pas gérées par un `case` spécifique. Par exemple :

```
switch(expression) {
 case val1 :
 case val2 :
  instruction1;
  instruction2;
  break;
 default :
  ...
}
```

Les instructions `instruction1` et `instruction2` sont effectuées dans le cas où `expression` vaut `val1` ou `val2`.

Enfin, il existe une deuxième syntaxe pour le `switch` :

```
switch(expression) {
 case val1, val2 -> {
  instruction1;
  instruction2;
  }
 default ->
  ...
}
```

Dans ce cas également, les instructions `instruction1` et `instruction2` sont effectuées dans le cas où `expression` vaut `val1` ou `val2`. Cette syntaxe « moderne » permet de faire la même chose depuis Java 14. Le `break` n'est plus obligatoire dans ce cas.

1.3.3.3 Les ternaires

Enfin, le dernier type de conditions que nous voyons est appelé une condition *ternaire*. Il est possible d'effectuer un *if-then-else* de manière concise grâce à la syntaxe ternaire : `(condition)?alors:sinon`. Si la `condition` est vraie, la partie `alors` est exécutée et retournée ; dans le cas contraire, la partie `sinon` est exécutée et retournée. Il est possible d'imbriquer des choses complexes dans la partie `alors` ou la partie `sinon` (y compris une imbrication de ternaires), mais pour améliorer la lisibilité du code, il est déconseillé d'en abuser.

Exemple 1.13

```
boolean b = ... ;
int n ;
if(b){
  n = 1 ;
}else{
  n = 2 ;
}
```

est équivalent à

```
boolean b = ... ;
int n = (b) ?1 :2 ;
```

```
boolean b = ... ;
if(b){
  System.out.println("Vrai");
}else{
  System.out.println("Faux");
}
```

est équivalent à

```
boolean b = ... ;
System.out.println(
          (b)?"Vrai":"Faux");
```

1.4 Entrées et sorties de base

Dans cette section, nous allons voir les éléments de base pour permettre la communication entre l'utilisateur et le programme.

1.4.1 La classe String

La classe `String` est définie dans Java pour représenter les chaînes de caractères. Elle est présente dans le package `java.lang`, ce qui signifie qu'il n'y a pas besoin de l'importer. Il est possible d'utiliser une constante de type `String` grâce aux guillemets :

```
String str = "Bonjour" ;
```

De nombreuses méthodes permettent de travailler avec des objets de cette classe (voir la Javadoc officielle pour les détails). Une `String` est un objet non modifiable : toute manipulation de la chaîne de caractères (concaténation, substitution d'un caractère par un autre,...) crée un nouvel objet. [4]

Voici quelques méthodes classiques de la classe `String` :

- `char charAt(int pos)` retourne le `char` une position donnée,
- `int compareTo(String str)` compare le `String` courant à celui passé en paramètre selon l'ordre lexicographique. La valeur retournée est un entier négatif si l'objet courant est situé avant `str` dans l'ordre lexicographique, un entier positif s'il est situé après, et 0 si les deux objets sont identiques,
- `boolean contains(CharSequence seq)` indique si `seq` est une sous-chaîne de l'objet courant,
- `boolean endsWith(String suffix)` détermine si la chaîne se termine par un certain suffixe,
- `int length()` retourne la longueur de la chaîne,
- `boolean startsWith(String prefix)` détermine si la chaîne commence par un certain préfixe.

La concaténation est l'opération qui consiste à « mettre bout à bout » plusieurs chaînes de caractères pour en créer une nouvelle. On utilise l'opérateur + pour effectuer cela.

Exemple 1.14

4. Nous reviendrons plus tard sur la création des objets et ce que cela implique du point de vue de la mémoire.

`"Bon" + "jour"` ⟶ `"Bonjour"`

`"Bon" + "jo" + "ur"` ⟶ `"Bonjour"`

L'opérateur + est une concaténation au lieu d'une addition si au moins un des opérandes est de type `String`. Par exemple :

`25 + "␣degres␣Celsius"` ⟶ `"25 degres Celsius"`

Les opérations sont effectuées de la gauche vers la droite :

`20 + 5 + "degres␣Celsius"` ⟶ `"25␣degres␣Celsius"`

Les valeurs de types primitifs peuvent être converties automatiquement en `String` pour la concaténation. Par exemple :

- l'entier 5 devient la chaine `"5"`,
- le décimal 5.0 devient la chaine `"5.0"`,
- le booléen `(5 == 5)` devient la chaine `"true"`.

Les valeurs de types non primitifs autres que `String` peuvent être converties *via* la méthode `toString()` qui est définie pour toutes les classes :

- si la méthode est définie explicitement, elle retourne une représentation sous forme de `String` de l'objet en question,
- si la méthode n'est pas définie explicitement, elle retourne le nom de la classe et une représentation du `hashCode` de la classe. Cette information n'est pas très lisible pour un utilisateur humain (ni pertinente en générale), il est donc recommandé de définir votre propre `toString` dans vos classes.

1.4.2 Entrées et sorties basiques

Java met à disposition plusieurs objets qui permettent de communiquer avec l'utilisateur via

- la sortie standard `System.out`,
- la sortie standard des erreurs `System.err`,
- l'entrée standard `System.in`.

1.4.2.1 Sorties basiques

`System.out` et `System.err` sont deux objets de même type (`java.io.PrintStream` sur lequel nous reviendrons plus tard). Pour débuter, on peut utiliser `System.out` et `System.err` principalement avec deux méthodes :

- `println(x)` affiche la valeur de `x` et provoque un retour à la ligne. La variante sans paramètre `println()` provoque un retour à la ligne,
- `print(x)` affiche la valeur de `x` sans retour à la ligne.

1.4.2.2 Entrées basiques

`System.in` est un objet de type `java.io.InputStream`. Pour simplifier son utilisation, nous l'appellerons à travers un objet de type `java.util.Scanner`.

```
Scanner sc = new Scanner(System.in) ;
```

Ne pas oublier d'importer `java.util.Scanner` !

Quelques méthodes de la classe `java.util.Scanner` :

- `boolean nextBoolean()`
- `double nextDouble()`
- `int nextInt()`
- `String nextLine()`
- ... (voir la Javadoc pour le reste)

Exemple 1.15

```
Scanner sc = new Scanner(System.in) ;
int n = sc.nextInt();
sc.close();
```

Ce code attend que l'utilisateur tape un entier au clavier ; cet entier est stocké dans la variable `n`.

Il faut toujours fermer le scanner avec `close()` après la *fin* de son utilisation.

1.4.3 Exemple de programme simple

Avec les éléments vus jusqu'ici, il est déjà possible de réaliser des petits programmes qui interagissent avec l'utilisateur et font des calculs en fonction de ses choix. Par exemple, une calculatrice (très) simple.

Nous découpons la description du programme en plusieurs étapes. D'abord, la méthode `main` dans laquelle on déclare un scanner pour lire l'entrée standard, et trois variables de type `int` qui serviront dans la suite. On remarque le scanner est fermé à la toute fin de la méthode `main`.

```
package up.mi.jgm.calculatrice;
import java.util.Scanner;

public class Calculatrice {
  public static void main(String[] args){
    Scanner sc = new Scanner(System.int);
    int choix, n1, n2;
    // ...
    sc.close();
  }
}
```

À la place laissée vide dans le code précédent, on introduit la boucle `do-while` suivante :

```
do {
  System.out.println("Quel␣est␣votre␣choix␣?");
  System.out.println("0␣-␣Quitter");
  System.out.println("1␣-␣Addition");
  System.out.println("2␣-␣Soustraction");

  choix = sc.nextInt();
  switch (choix) {
    ...
  }
} while (choix != 0);
```

On propose différents choix à l'utilisateur (0, 1 ou 2), et on récupère la valeur de son choix avec `sc.nextInt()`. Le `switch` permet d'adapter les actions au choix de l'utilisateur. On recommence tant que le choix de l'utilisateur est différent de 0. Les actions à effectuer dans les différents cas sont détaillées ci-dessous :

```
switch (choix) {
  case 0:
    break;
  case 1:
    System.out.println("Quels␣entiers␣souhaitez␣vous␣additioner␣?");
    n1 = sc.nextInt();
    n2 = sc.nextInt();
    System.out.println("Le␣resultat␣de␣l'addition␣est␣:␣"+n1+"␣+␣"+n2+"␣=␣"+(n1 + n2));
    break;
  case 2:
    System.out.println("Quels␣entiers␣souhaitez␣vous␣soustraire␣?");
    n1 = sc.nextInt();
    n2 = sc.nextInt();
    System.out.println("Le␣resultat␣de␣la␣soustraction␣est␣:␣"+n1+"␣-␣"+n2+"␣=␣"+(n1 - n2));
    break;
  default:
    System.out.println("Ce␣choix␣n'est␣pas␣conforme.");
}
```

La Figure 1.6 montre un exemple d'exécution de la calculatrice. Nous verrons plus tard la gestion des erreurs, comme celle qui se produit à la dernière étape.

1.5 Premières classes

Grossièrement, une classe est une structure de données accompagnée d'opérations pour manipuler les données :

- les données sont appelées les attributs, ce sont les variables définies dans la classe,
- les opérations sont appelées les méthodes, ce sont les fonctions définies dans la classe.

Les attributs et les méthodes forment l'ensemble des membres de la classe. [5]

1.5.1 Méthodes

Une méthode est définie par une signature suivie d'un bloc d'instructions (appelé le *corps* de la méthode). Contrairement à un `if` ou une boucle, il est obligatoire d'utiliser des accolades, même si le corps de la méthode ne contient qu'une instruction. La *signature* d'une méthode est composée de :

- la liste de modificateurs,
- le type de retour,
- le nom de méthode,
- la liste de paramètres : type et nom de chaque paramètre (on parle aussi d'arguments).

Exemple :

```
public int somme(int n, int m){
```

5. En fait, il peut y avoir d'autres membres, dont on parlera plus tard.

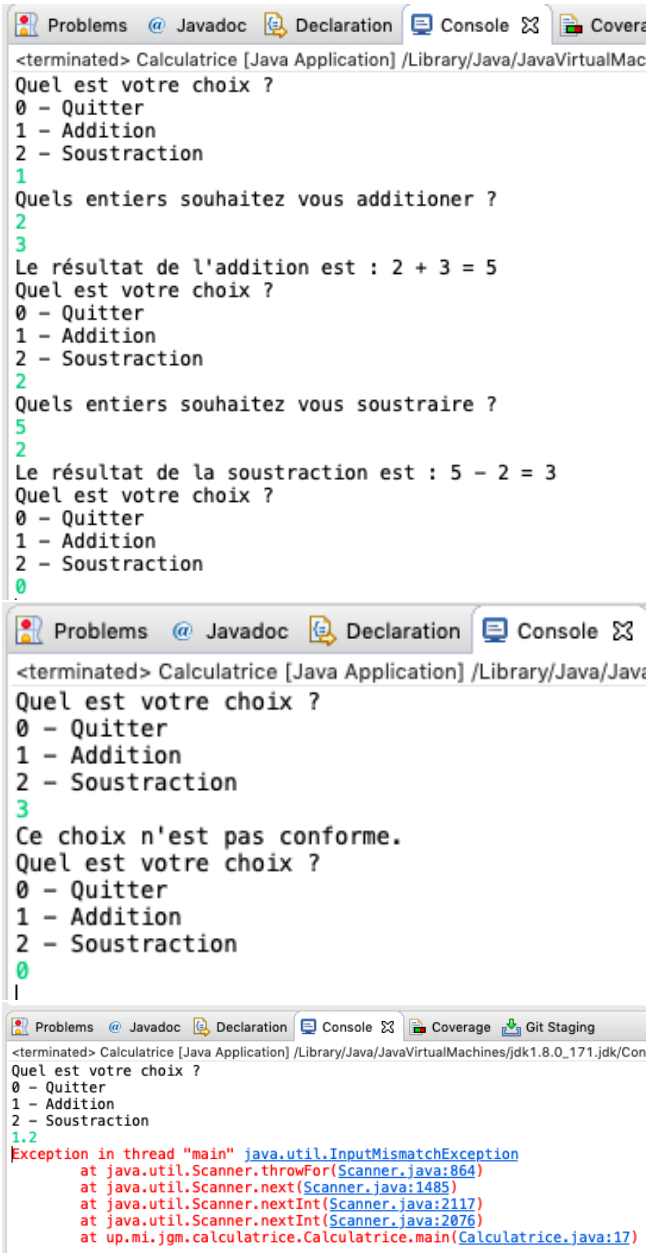


FIGURE 1.6 – Exécution de la calculatrice

```
  return n + m ;
}
```

1.5.1.1 Modificateurs

Il existe plusieurs types de modificateurs. Pour l'instant, on s'intéresse principalement aux modificateurs d'accès, qui indiquent la visibilité de la méthode dans le reste du code :

- `public` : on peut utiliser la méthode depuis n'importe quel endroit du programme,
- `protected` : on peut utiliser la méthode depuis les classes filles et les classes du même package,
- par défaut : sans modificateur d'accès, on peut utiliser la méthode depuis les classes du même package,
- `private` : on peut utiliser la méthode uniquement à l'intérieur de la classe où elle est définie.

Il existe d'autres modificateurs dont nous parlerons plus tard (`static`, `abstract`, `final`, `synchronized`,. . .).

1.5.1.2 Arguments variables

On peut définir des méthodes dont on ne connaît pas à l'avance le nombre d'arguments d'un certain type : on parle de *varargs*. Il ne peut y avoir qu'un seul *varargs* par méthode, en dernière position de la liste des arguments.

Exemple 1.16

```
public int somme(int... entiers){
  int resultat = 0 ;
  for(int n :entiers){
    resultat += n ;
  }
  return resultat ;
}
```

On peut utiliser cette méthode ainsi :

```
int n = somme(1,2,3);
int m = somme(1,2,3,4);
```

1.5.1.3 Retour d'une fonction

Le mot-clé `return` termine l'exécution d'une méthode, et retourne à la méthode appelante, généralement en transmettant une valeur. La valeur retournée doit correspondre au type de retour indiqué dans la signature. Si la méthode n'a pas besoin de retourner une valeur, on peut ne pas utiliser le mot-clé `return`, ou l'utiliser sans valeur associée : « `return ;` », dans ce cas le type de retour indiqué dans la signature est `void`.

1.5.1.4 Bonnes pratiques : méthodes et conventions de nommage

Il y a un principe à retenir en programmation : mieux vaut avoir de nombreuses méthodes courtes que peu de méthodes longues. Cela a plusieurs intérêts. Tout d'abord, cela facilite la factorisation : on peut réutiliser facilement un même morceau de code à plusieurs endroits juste en appelant la méthode correspondant. Ensuite, cela donne un code plus facile à lire et à comprendre, et donc un code plus facile à tester, débuguer, maintenir,. . .

Ensuite, il est important d'utiliser des noms significatifs pour les identificateurs, c'est-à-dire les noms des méthodes, variables,. . . On doit comprendre le rôle d'une méthode (ou d'une variable) grâce à son nom. Il faut donc limiter l'utiliser de variables dont le nom est très court à un usage local, comme par exemple un compteur de boucle :

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

ou un paramètre de méthode très courte :

```
int carre(int n){
  return n * n;
}
```

En Java, la casse est importante pour reconnaître la nature des identificateurs :

- un nom de variable ou de méthode commence par une minuscule,

- un nom de classe (ou interface, ou enum) commence par une majuscule,
- *Camel Case* : lorsque le nom est composé de plusieurs mots, ils sont indiqués par des majuscules : `int maVariableDeTypeInt`, `MaClasse`
- cas des constantes : tout en majuscules, les mots sont séparés par un underscore _ : `MA_CONSTANTE`.

Ce ne sont pas des contraintes techniques, c'est-à-dire que le code compile même s'il ne respecte pas ces consignes. Mais ce sont des conventions établies qui doivent être utilisées par tous les programmeurs en Java, afin que chacun puisse plus facilement relire le code d'autres développeurs.

1.5.1.5 Méthodes statiques

Le modificateur `static` permet d'indiquer qu'une méthode peut être appelée sans instancier sa classe [6] : Dans la classe, on peut appeler directement la méthode : `maMethode();`. Depuis une autre classe, on l'appelle précédée du nom de la classe dans laquelle est est définie : `MaClasse.maMethode();`. Pour un attribut, `static` indique que sa valeur est partagée entre les différentes instances de sa classe (on parle d'attribut de classe). Une méthode `static` peut faire appel à des méthodes et des attributs `static`, mais aucune méthode ou aucun attribut lié à une de la classe.

1.5.1.6 Réécriture de la Calculatrice

On peut réécrire le code de la Calculatrice en le découpant grâce à des méthodes statiques : on obtient un résultat plus lisible et plus facile à faire évoluer.

On créer une méthode par opération :

```
private static void addition(Scanner sc) {
  int n1, n2;
  System.out.println("Quels␣entiers␣souhaitez␣vous␣additioner␣?");
  n1 = sc.nextInt();
  n2 = sc.nextInt();
  System.out.println("Le␣resultat␣de␣l'addition␣est␣:␣"+n1+"␣+␣"+n2+"␣=␣"+(n1 + n2));
}
```

```
private static void soustraction(Scanner sc) {
  int n1, n2;
  System.out.println("Quels␣entiers␣souhaitez␣vous␣soustraire␣?");
  n1 = sc.nextInt();
  n2 = sc.nextInt();
  System.out.println("Le␣resultat␣de␣la␣soustraction␣est␣:␣"+n1+"␣-␣"+n2+"␣=␣"+(n1 - n2));
}
```

On pourrait facilement étendre la calculatrice avec des méthodes dédiées à d'autres opérations (division, multiplication,. . .).

On définit également une méthode qui va afficher le menu, et une autre pour l'exécution de la boucle principale du programme.

```
private static void menu() {
```

6. C'est-à-dire sans créer un objet qui appartient à cette classe. Les rappels sur la notion d'instance d'une classe viendront plus tard.

```
  System.out.println("Quel␣est␣votre␣choix␣?");
  System.out.println("0␣-␣Quitter");
  System.out.println("1␣-␣Addition");
  System.out.println("2␣-␣Soustraction");
}
```

```
private static void boucle(Scanner sc) {
  int choix;
  do {
    menu();
    choix = sc.nextInt();
    switch (choix) {
    case 0: break;
    case 1:
      addition(sc); break;
    case 2:
      soustraction(sc); break;
    default:
     System.out.println("Ce␣choix␣n'est␣pas␣conforme.");
    }
  } while (choix != 0);
}
```

Enfin, la méthode `main`, qui est beaucoup plus concise que dans le premier exemple.

```
public static void main(String[] args) {
  Scanner sc = new Scanner(System.in);
  boucle(sc);
  sc.close();
}
```

Pour que cela fonctionne, on suppose que toutes les méthodes de cet exemple sont définies dans une même classe, qu'on peut appeler par exemple `CalculatriceDecomposee`. Le fonctionnement du programme est exactement le même que le précédent, mais le code a l'avantage d'être plus clair et plus facile à maintenir.

1.5.2 Classes utilitaires

On appelle parfois une classe *utilitaire* un ensemble de méthodes qui réalisent des calculs indépendants d'une instance d'objet. Il n'est donc jamais utile d'instancier la classe. Une telle classe peut également définir des constantes qui sont utilisables soit dans les méthodes de la classe, soit partout ailleurs. Par exemple, `Math.PI` peut être utilisée pour calculer le périmètre d'un cercle, même sans créer d'instance de la classe `Math`. Une telle constante peut être déclarée ainsi : `public static final double PI = 3.14`.

La classe `Math` est un exemple classique de classe utilitaire. Elle propose notamment :

- deux constantes mathématiques :
  - `Math.E` : le nombre e (base des logarithmes néperiens),
  - `Math.PI` : le nombre π (périmètre d'un cercle divisé par son diamètre),
- de nombreuses méthodes :
  - `double abs(double a)` : valeur absolue d'un nombre,

- `double ceil(double a)` : l'entier le plus proche supérieur ou égal à `a` (la valeur retournée est de type `double`),
- `double floor(double a)` : comme `ceil` mais inférieur ou égal,
- `double pow(double a,double b)` : calcule a^b,
- Les fonctions trignonométriques habituelles ($\sin, \tan, \cos, \ldots$),
- ...

Pour plus d'informations sur cette classe, voir la documentation complète : `java.lang.Math`.

Nous présentons un exemple de classe utilitaire pour la manipulation de tableaux, dont voici la liste des méthodes :

```
package up.mi.jgm.util;
public class UtilTab {
  public static void affichageTableau(double[] tab, boolean enLigne) { ... }

  public static boolean appartient(double val, double[] tab) { ... }

  public static double min(double[] tab) { ... }

  public static double max(double[] tab) { ... }

  public static double somme(double[] tab) { ... }

  public static void triParSelection(double[] tab) { ... }
}
```

La première méthode sert à afficher le contenu d'un tableau de nombres, soit en ligne soit en colonne, selon la valeur du paramètre `enLigne`.

```
public static void affichageTableau(double[] tab, boolean enLigne) {
  for(int i = 0 ;i < tab.length ;i++) {
    System.out.print(tab[i]);
      if(enLigne) {
        System.out.print("␣");
      }else {
        System.out.println();
      }
    }
  System.out.println();
}
```

La deuxième méthode permet d'indiquer si un nombre donné appartient à un tableau.

```
public static boolean appartient(double val, double[] tab) {
  for(int i = 0 ;i < tab.length ;i++) {
    if(tab[i] == val) {
      return true ;
    }
  }
  return false ;
}
```

On peut également définir une méthode qui parcourt un tableau pour identifier son plus petit élément.

```
public static double min(double[] tab) {
  double tmpMin = tab[0] ;
  for(int i = 1 ;i < tab.length ;i++) {
    if(tab[i] < tmpMin) {
      tmpMin = tab[i];
    }
  }
  return tmpMin ;
}
```

La méthode `max` ressemble beaucoup à la méthode `min`, je vous laisse essayer. . .

La méthode qui calcule la somme des éléments d'un tableau est assez similaire.

```
public static double somme(double[] tab) {
  double tmpSomme = 0 ;
  for(int i = 0 ;i < tab.length ;i++) {
    tmpSomme += tab[i];
  }
  return tmpSomme;
}
```

Enfin, l'algorithme de tri par sélection peut être implémenté. Il consiste, en résumé, à échanger le plus petit élément du tableau avec celui qui est à la première position, puis à répéter cette opération entre la deuxième position et la fin du tableau, puis entre la troisième position et la fin du tableau,. . .

```
public static void triParSelection(double[] tab) {
  for(int i = 0 ;i < tab.length - 1 ;i++) {
    int indiceMin = rechercheIndicePlusPetit(tab, i);
    if(indiceMin != i) {
      echanger(tab,i, indiceMin);
    }
  }
}

private static int rechercheIndicePlusPetit(double[] tab, int indiceMin) {
  for(int j = indiceMin + 1 ;j < tab.length ;j++) {
    if(tab[j] < tab[indiceMin]) {
      indiceMin = j ;
    }
  }
  return indiceMin ;
}

private static void echanger(double[] tab, int i, int j) {
  double tmpVal = tab[i];
  tab[i] = tab[j];
  tab[j] = tmpVal ;
}
```

Voici un exemple d'utilisation des méthodes de la classe `UtilTab`, avec le résultat de l'exécution en Figure 1.7.

```
package up.mi.jgm.util;

public class TestUtilTab {
  public static void main(String[] args) {
    double[] tab = {18.2, 2.3, 5, 42, 23.7 };
    UtilTab.affichageTableau(tab, false);
    System.out.println(UtilTab.appartient(0, tab));
    System.out.println(UtilTab.appartient(2.3, tab));
    System.out.println(UtilTab.max(tab));
    System.out.println(UtilTab.min(tab));
    System.out.println(UtilTab.somme(tab));
    UtilTab.triParSelection(tab);
    UtilTab.affichageTableau(tab, true);
  }
}
```

Figure 1.7 – Exemple d'utilisation de la classe UtilTab

Chapitre 2. Bases de la Programmation Orientée Objet

Contenu

- ❑ *Objets et classes*

Une des principales particularités de Java est que c'est un langage *orienté objet*. Dans ce chapitre, on voit les bases de la *Programmation Orientée Objet* (POO) en Java.

2.1 Objets et classes

2.1.1 Créations de classes et d'objets en Java

2.1.1.1 Objets et classes : les bases

La première question à se poser est « Qu'est-ce qu'un objet ? ». Un objet, en programmation, est une entité créée et manipulée par un programme correspondant à une entité (concrète ou abstraite) du monde réel. Des exemples de modélisation d'entités du monde réel par des objets sont donnés dans la Table 2.1.

5 répertoires téléphoniques avec 100 personnes chacun	5 objets `Repertoire` 500 objets `Personne`
Un cinéma qui diffuse 20 films avec 73 acteurs différents	1 objet `Cinema` 20 objets `Film` 73 objets `Acteur`
Une promo de licence avec 200 étudiants qui ont 5 notes pour chacune des 8 unités d'enseignement qu'ils étudient	1 objet `Promotion` 200 objets `Etudiant` 8 objets `UniteEnseignement` 8000 objets `Note`

Table 2.1 – Exemples d'associations « monde réel - objets »

On peut ensuite se demander « Qu'est-ce qu'une classe ? ». Certains objets représentent des entités du monde réel qui appartiennent à une même catégorie, sont des instances d'un même concept. La *classe* est l'entité informatique qui correspond à ce concept. Des exemples sont donnés dans la Table 2.2.

Monde réel	Objets	Classe
Marlon Brando Robert De Niro Al Pacino Talia Shire ...	un objet `brando` un objet `deNiro` un objet `pacino` un objet `shire` ...	`Acteur`
Programmation avancée Génie logiciel Algorithmique avancée ...	un objet `progAv` un objet `genieLog` un objet `algoAv` ...	`UniteEnseignement`

Table 2.2 – Exemples d'associations « monde réel - objets - classes »

Pour créer une classe en Java, on place le code source dans des fichiers qui portent l'extension `.java`. Le préfixe du nom du fichier doit correspondre au nom de l'unique classe publique définie dans le fichier.

Exemple 2.1 Dans le fichier `MaClasse.java`, le code doit définir une classe publique `MaClasse` :

```
public class MaClasse {
 // ...
}
```

On verra dans la suite qu'on peut avoir plusieurs classes dans le même fichier (celles qui ne correspondent pas au nom du fichier ne peuvent pas être déclarées `public`). Pour le début du cours, on fait l'hypothèse simplificatrice « 1 fichier = 1 classe ».

On a parlé des membres *de classe* précédemment (`static`), qui ne nécessitent pas de créer d'instances de la classe pour les utiliser. Les membres (attributs ou méthodes) non `static` sont les membres *d'instance* : on doit créer un objet (= une instance) de cette classe pour les utiliser.

Exemple 2.2 Dans cette classe qui sert à représenter des acteurs, les attributs `nom`, `prenom` et `nationalite` n'ont pas de sens s'ils ne sont pas associés à un acteur précis.

```
public class Acteur {
  private String nom ;
  private String prenom ;
  private String nationalite ;

  // ...
}
```

On a déjà parlé des signatures des méthodes précédemment. Plusieurs méthodes qui servent globalement à la même chose peuvent porter le même nom, à condition d'avoir des paramètres différents (types et/ou nombre de paramètres) : c'est la surcharge.

Exemple 2.3

```
public class Film {
  public void ajoutActeur(Acteur acteur){
    // ...
  }

  public void ajoutActeur(String nom, String prenom, String nationalite){
    // ...
  }
}
```

2.1.1.2 Constructeurs

Les constructeurs sont des méthodes particulières qui servent à créer des instances de la classe. La définition d'un constructeur est différente de la définition d'une méthode « classique ». Contrairement à une méthode ordinaire, un constructeur n'a pas de type de retour. Un constructeur doit obligatoirement avoir le même nom que la classe (y compris la majuscule au début !).

La principale utilité d'un constructeur est d'initialiser les attributs de la classe. D'un point de vue pratique, c'est également au moment de l'appel du constructeur qu'une zone dans la mémoire du programme est allouée

à l'objet en cours de création.

Exemple 2.4 Voici un exemple de constructeur qui initialise un objet `Acteur` en fonction de son nom, son prénom et sa nationalité.

```
public class Acteur {
  private String nom ;
  private String prenom ;
  private String nationalite ;

  public Acteur(String n, String p, String nat){
    nom = n ;
    prenom = p ;
    nationalite = nat ;
  }
}
```

Les constructeurs peuvent aussi être surchargés :

Exemple 2.5 Avec le constructeur surchargé, on initialise un acteur sans connaître sa nationalité.

```
public class Acteur {
  // ...

  public Acteur(String n, String p, String nat){
    nom = n ;
    prenom = p ;
    nationalite = nat ;
  }
  public Acteur(String n, String p){
    nom = n ;
    prenom = p ;
    nationalite = "" ;
  }
}
```

Le mot-clé `this` a plusieurs rôles. Le premier est de lever les ambiguités si un attribut porte le même nom qu'une variable locale, c'est-à-dire distinguer un attribut d'une variable locale (ou paramètre de méthode) qui porte le même nom. Le second est de faire appel à un constructeur depuis un autre constructeur.

Exemple 2.6 Dans cet exemple, `this.nom` fait référence à l'attribut `nom` de la classe `Acteur`, tandis que lorsque `nom` est écrit seul, il fait référence au paramètre du constructeur.

```
public class Acteur {
  // ...

  public Acteur(String nom, String prenom,
                      String nationalite){
    this.nom = nom ;
    this.prenom = prenom ;
    this.nationalite = nationalite ;
  }
}
```

Exemple 2.7 Dans cet exemple, l'utilisation de `this(...)` dans le second constructeur (celui qui possède deux paramètres) permet d'utiliser le premier constructeur (celui qui possède trois paramètres).

```
public class Acteur {
  // ...

  public Acteur(String nom, String prenom,
                      String nationalite){
    this.nom = nom ;
    this.prenom = prenom ;
    this.nationalite = nationalite ;
  }

  public Acteur(String nom, String prenom){
    this(nom,prenom,"") ;
  }
}
```

L'nstanciation est la création d'un objet appartenant à une classe, via un appel au constructeur avec le mot-clé `new`.

Exemple 2.8 Cet exemple montre comment initialiser une zone de la mémoire qui contiendra les informations concernant Robert De Niro. [1]

```
Acteur deNiro = new Acteur("De␣Niro", "Robert", "Americain");
```

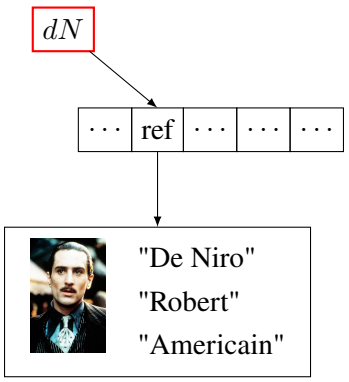
Ce code crée une zone de la mémoire dédiée aux données de l'objet créé, retourne une référence vers cet objet.

2.1.1.3 Affectation et passage de paramètres

On a déjà vu qu'en Java, l'affectation se fait en recopie et le passage de paramètres se fait par valeur... pour les types simples. Concernant les objets, c'est un peu plus complexe. En Java, on ne manipule jamais directement les objets, mais seulement des références vers ces objets ($\simeq$ des pointeurs). Lors d'une affectation ou d'un passage de paramètre, c'est donc la référence qui est copiée, pas l'objet lui même.

Voici un exemple d'affectation d'un objet correspondant à l'acteur Robert De Niro.

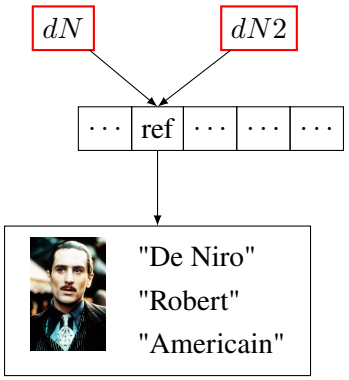
```
Acteur dN = new Acteur("De␣Niro", "Robert", "Americain");
```



1. Je triche un peu, Robert De Niro a également la nationalité italienne depuis 2006...

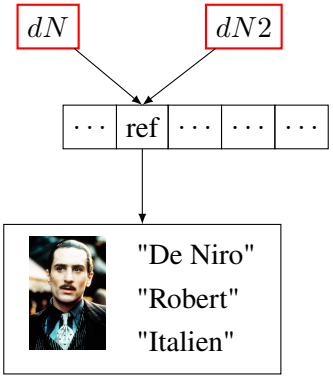
Lorsqu'on crée une nouvelle variable `dN2` à laquelle on affecte `dN`, cela crée une référence qui correspond à la même zone dans la mémoire du programme.

```
Acteur dN = new Acteur("De␣Niro", "Robert", "Americain");
Acteur dN2 = dN ;
```



Par conséquent, si on modifie ces informations via les méthodes de l'objet `dN2`, cela revient au même que de modifier directement `dN`.

```
Acteur dN = new Acteur("De␣Niro", "Robert", "Americain");
Acteur dN2 = dN ;
dN2.setNationalite("Italien") ;
```

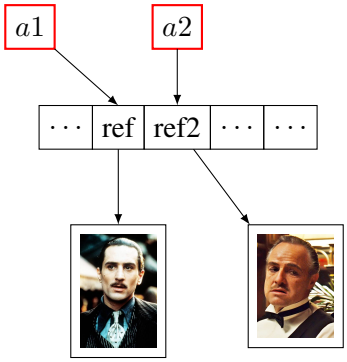



```
Acteur dN = new Acteur("De␣Niro", "Robert", "Americain");
Acteur dN2 = dN ;
dN2.setNationalite("Italien") ;
System.out.println(dN.getNationalite()) ;
```

Ce petit programme affiche donc `"Italien"`.

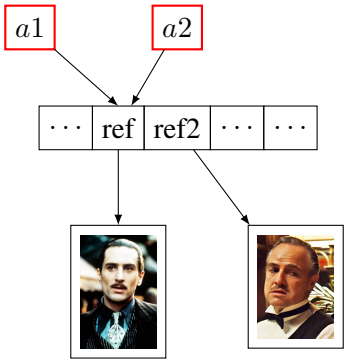
Supposons maintenant qu'on crée deux objets de type `Acteur`.

```
Acteur a1 = new Acteur("De␣Niro", "Robert", "Americain");
Acteur a2 = new Acteur("Brando", "Marlon", "Americain");
```



Lorsqu'on affecte à la variable `a2` le contenu de la variable `a1`, cela signifie que maintenant `a2` correspond à la même référence que `a1`.

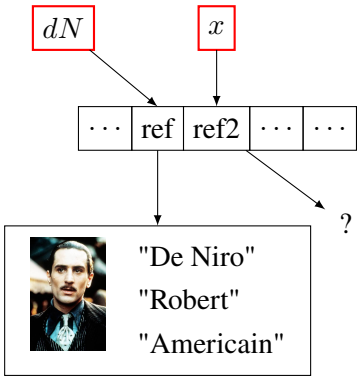
```
Acteur a1 = new Acteur("De␣Niro", "Robert", "Americain");
Acteur a2 = new Acteur("Brando", "Marlon", "Americain");
a2 = a1 ;
```



Le *garbage collector* (GC, ou ramasse-miettes en français) est le sous-système de la JVM en charge de la mémoire. Il libère l'espace correspondant à Marlon Brando dès que plus aucune référence à cet objet n'est faite dans le programme.

Il existe le mot-clé `null`, qui signifie qu'on ne fait référence à aucun objet.

```
Acteur dN = new Acteur("De␣Niro", "Robert", "Americain");
Acteur x = null ;
System.out.println(dN.getNom());
System.out.println(x.getNom());
```



Le premier appel à `System.out.println` affiche `"De Niro"`. Par contre, le deuxième appel pose problème : si on essaye d'accéder à un membre d'une référence `null`, on optient une erreur : `java.lang.NullPointerException`.

Voyons maintenant le passage de paramètres des méthodes.

```
public class Test {
  private static void f(Acteur a){
   a.setNationalite("Italien") ;
  }
  public static void main(String[] args){
    Acteur dN = new Acteur("De Niro", "Robert", "Americain");
    f(dN);
    System.out.println(dN.getNationalite());
  }
}
```

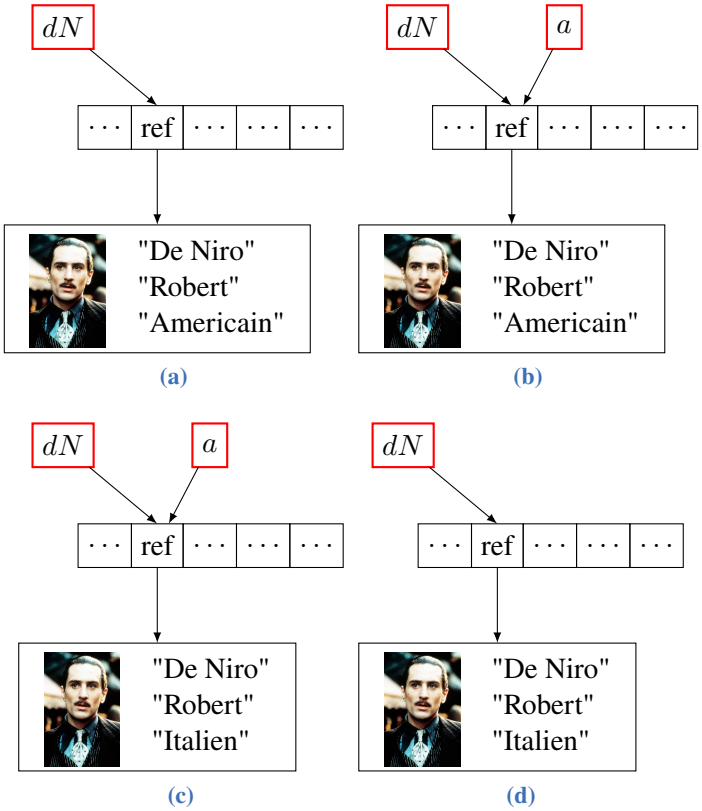


FIGURE 2.1

1. Début de l'exécution de la méthode `main` : initialisation de `dN` (Figure 2.1a).
2. Appel de la méthode `f` une variable locale `a` (le paramètre) est créée, correspondant à la même référence que `dN` (Figure 2.1b).
3. Exécution de `f` : modification de la nationalité de De Niro via la variable `a` (Figure 2.1c).
4. Après l'exécution de `f`, la variable locale `a` n'existe plus. Affichage : `"Italien"` (Figure 2.1d).

On peut modifier le contenu d'un objet dans une méthode, via les méthodes de cet objet. Mais on ne peut pas affecter complètement une nouvelle référence à cet objet :

```
public class Test {
  private static void f(Acteur a){
    a = new Acteur("Brando","Marlon", "Americain");
  }
  public static void main(String[] args){
    Acteur dN = new Acteur("De␣Niro", "Robert", "Americain");
    f(dN);
    System.out.println(dN.getNom());
  }
}
```

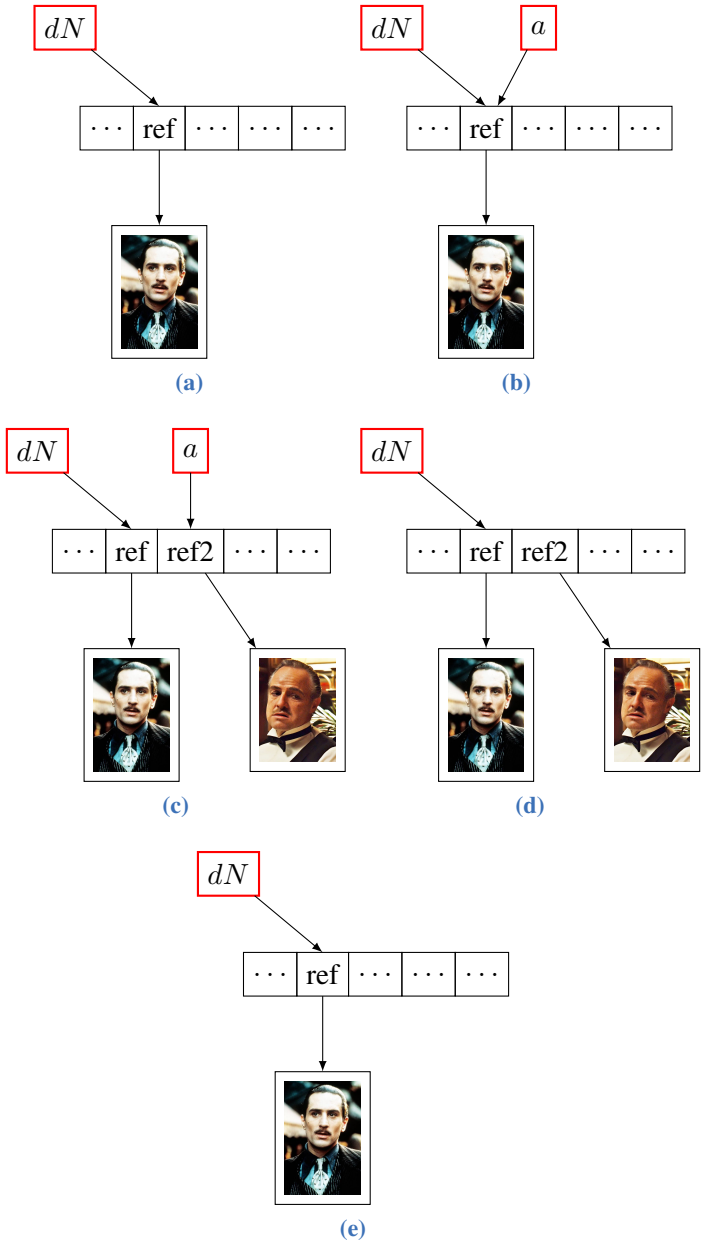


FIGURE 2.2

1. Au début de l'exécution de la méthode `main`, la variable `dN` est initialisée (Figure 2.2a).
2. Appel de la méthode `f` une variable locale `a` (le paramètre) est créée, correspondant à la même référence

que dN (Figure 2.2b).

3. Pendant l'exécution de f, création d'un nouvel objet et d'une nouvelle référence pour Marlon Brando (Figure 2.2c).
4. Sortie de f : suppression de la variable locale a (Figure 2.2d).
5. Le Garbage Collector libère l'espace occupé par l'objet correspondant à Marlon Brando (Figure 2.2e).
6. Après l'exécution de f, il ne reste qu'un objet en mémoire. Affichage : "De Niro" (Figure 2.2e).

Pour finir avec les informations sur l'opérateur « new » et les initialisations d'objets, cet opérateur retourne un objet qui peut être directement utilisé dans une expression plus complexe, par exemple l'appel d'une méthode directement sur le résultat du new :

```
System.out.println(new Acteur("De␣Niro","Robert", "Americain").getNom());
```

ou l'utilisation comme paramètre d'une méthode :

```
Film film = new Film();
film.ajoutActeur(new Acteur("De␣Niro","Robert", "Americain"));
```

Nous avons brièvement évoqué le Garbage Collector (GC) précédemment. Le GC détecte les objets inutilisables et les supprime de la mémoire. Il fonctionne en tâche de fond et ne supprime pas forcément les objets *immédiatement* après qu'ils soient devenus inutilisables. Par « objet inutilisable », on entend un objet qui n'est associé à aucune référence. Le programmeur n'a pas à gérer manuellement la mémoire (pas de free() comment en C), mais en cas de besoin, on peut activer le GC manuellement avec « System.gc(); », même si ce n'est pas courant.

Enfin, pour conclure cette partie, il est important de noter que les tableaux sont des objets : pour chaque type d'éléments de tableaux, une classe correspondante est créée. Par exemple,

```
public class Test {
  public static void main(String[] args){
    String[] tab = {"Hello␣", "World!"};
    System.out.println(tab[0].getClass());
    System.out.println(tab.getClass());
  }
}
```

Dans ce cas l'affichage obtenu est class java.lang.String
class [Ljava.lang.String;

Comme les tableaux sont des objets, les affectations et passages de paramètres se font par référence. La longueur du tableau correspond donc à un attribut public et constant : tab.length. [2]

De plus, les éléments d'un tableaux peuvent être de n'importe quel type, y compris des tableaux ! On peut utiliser ce genre de structure, par exemple, pour représenter une matrice d'entiers de dimension 3×3 :

```
int[][] t = new int[3][] ;
t[0] = {1,2,3} ;
t[1] = {4,5,6} ;
t[2] = {7,8,9} ;
```

2. Nous verrons plus tard la définition d'attributs constants.

2.1.2 Encapsulation

2.1.2.1 Le principe d'encapsulation

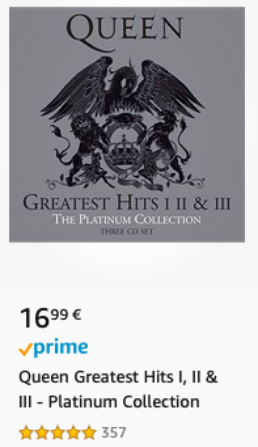
Le *principe d'encapsulation* dit que les données (c'est-à-dire les attributs) ne doivent pas être accessibles et modifiables librement par n'importe qui. Pour cela, il est recommandé que les attributs soient `private`, et ne puissent être utilisés ou modifiés que via des méthodes. Cela permet au développeur de s'assurer que ses classes seront utilisées d'une manière cohérente par la suite.

Exemple 2.9 Voici un exemple de classe sans encapsulation

```
public class Produit {
  public String nom ;
  public double prix ;
  public Produit(String nom, double prix){
    this.nom = nom ;
    this.prix = prix ;
  }
}
```

L'utilisation de cette classe peut mener à des incohérences :

```
Produit prod = new Produit("CD Greatest Hits", 16.99);
// Ailleurs dans le code...
prod.prix = -10 ;
```



Exemple 2.10 On ré-écrit la même classe avec encapsulation :

```
public class Produit {
  private String nom ;
  private double prix ;

  public Produit(String nom, double prix){
    this.nom = nom ;
    this.prix = prix ;
  }
  public void setPrix(double prix){
    if(prix>=0){
```

```
      this.prix = prix ;
    }
  }
}
```

Il est impossible de mettre un prix incohérent maintenant ! Si nécessaire, on peut faire le même genre de contrôle de cohérence dans le constructeur.

L'encapsulation a plusieurs buts. D'abord, on assure dans les méthodes le respect de certaines contraintes (voir l'exemple du prix d'un produit). Ensuite, une conséquence importante de l'encapsulation est que les éventuelles modifications de la classe d'une version à l'autre sont transparentes pour les utilisateurs, qui ne voient que son interface (c'est-à-dire les méthodes publiques) sans voir les détails d'implémentation.

Pour résumer, on peut dire qu'un attribut doit être `private` dans la majorité des cas, et on y accède ou on le modifie via des méthodes. Les seules exceptions sont (certaines) constantes, et l'utilisation de `protected` pour l'héritage (on reviendra plus tard sur le sujet de l'héritage).

2.1.2.2 Les constantes

Le modificateur `final` permet de définir des constantes : un attribut `final` ne peut plus être modifié après son initialisation. Pour un attribut de type primitif, la valeur est fixée. Par exemple, `final int N = 5 ;` → `N` vaudra toujours 5, il est impossible par la suite de lui affecter une autre valeur.

Pour un attribut de type objet, la référence est fixée, mais l'état de l'objet peut être modifié via des méthodes. Par exemple, si on a initialisé l'objet `final Acteur DE_NIRO = new Acteur(...);`, on peut modifier sa nationalité avec `DE_NIRO.setNationalite("Italien");`.

C'est le même fonctionnement pour une variable locale ; on parlera plus tard de l'utilisation de `final` pour les méthodes et classes. Par convention, les noms des constantes sont écrits en majuscules, avec les mots séparés par un underscore _.

Par défaut, une constante est relative à une instance de la classe, elle peut (par exemple) être initialisée dans le constructeur :

```
public class Personne {
  private final String NUM_SECU ;
  public Personne(String numSecu){
    NUM_SECU = numSecu ;
  }
}
```

On peut également définir une constante « absolue » :

```
public class Math {
  public static final double PI = 3.14 ;
}
```

Dans ce cas, `PI` est accessible depuis n'importe quel endroit du code (`public`), sans devoir créer d'instance de `Math` (`static`), et sans possibilité de modifier sa valeur (`final`).

2.1.2.3 Accès et modification des attributs

Par convention, on accède à un attribut `XXX` via une méthode `getXXX` et on le modifie avec `setXXX` (on parle de *getters* et *setters*) :

```
public class Acteur {
  private String nationalite ;

  public void setNationalite(String nationalite){
    this.nationalite = nationalite ;
  }

  public String getNationalite(){
    return nationalite ;
  }
}
```

Annexe A Documentation

A.1 Commentaires

La documentation du code est essentielle : cela facilite la réutilisation et les modifications du code (par soi-même, ou par d'autres développeurs). Il existe deux formes classiques de commentaires en Java :

```
// Un commentaire en une ligne

/*
 Un commentaire sur
 plusieurs lignes
*/
```

Ces formes de commentaires doivent être utilisées avec parcimonie, pour expliquer uniquement des points précis du code qui peuvent être complexes. Mais abuser de ces commentaires risque, au contraire, de rendre le code plus lourd, il faut donc éviter de commenter les parties triviales du code ! Un exemple de ce qu'il ne faut pas faire :

```
int i = 0 ;// on initialise le compteur
while(i < 100) { // on boucle 100 fois
  if((i % 2) == 0){ // on verifie si i est pair
    System.out.println(i + "␣est␣pair.");
    /*
    On affiche un message si
    i est pair
    */
  }
  i++ ;// on incremente le compteur
}
```

Dans cet exemple, les commentaires présents à chaque ligne de code le rendent plus lourd et moins lisible. D'autant plus que lorsque les conventions de nommage sont respectées, et que les identificateurs choisis sont significatifs (c'est-à-dire que les noms de variables, méthodes ou classes expliquent à quoi servent ces variables, méthodes ou classes), la lecture du code peut se faire de manière « transparente », sans explication additionnelle. Par exemple :

```
public Integer plusPetitElement(List<Integer> listeEntiers){
  if(listeEntiers.isEmpty())
    return null ;
  Integer minTemporaire = listeEntiers.get(0) ;
  for(int i = 1 ;i < listeEntiers.size() ;i++){
    if(listeEntiers.get(i) < minTemporaire)
      minTemporaire = listeEntiers.get(i) ;
  }
  return minTemporaire ;
}
```

Chaque identifiant a un nom pertinent, ce qui rend le code lisible sans avoir besoin de commentaire chaque ligne.

A.2 Javadoc

Au lieu de commenter directement le code dans le corps des méthodes, on privilégie la documentation des classes, méthodes et attributs via des commentaires Javadoc. Un commentaire Javadoc commence par `/**` et se termine par `*/`, et commence par une description textuelle de l'élément documenté, et comporte un certain nombre d'annotations (*tags* en anglais) qui permettent d'indiquer le rôle des éléments du code. L'outil `javadoc` du JDK génère un ensemble de pages web correspondant à la documentation du code, en se basant sur les commentaires Javadoc. L'intérêt de la Javadoc est de pouvoir utiliser les méthodes d'une classe juste en connaissant leur signature et leur documentation, mais sans avoir besoin de regarder les détails du code pour comprendre ce que font les méthodes et comment elles s'utilisent.

A.2.1 Les tags Javadoc

La Table A.1 décrit quelques annotations habituelles pour les commentaires Javadoc.

Tag	Localisation	Rôle
`@author`	Classe	Nom de l'auteur
`@version`	Classe	Version de la classe
`@deprecated`	Classe Constructeur Méthode Attribut	Marquer comme obsolète, [1] décrire pourquoi, et par quoi remplacer
`@see`	Classe Constructeur Méthode Attribut	Ajoute un lien « Voir aussi »
`@param`	Constructeur Méthode	Décrire un paramètre
`@return`	Méthode	Décrire la valeur retournée

Table A.1 – Quelques annotations Javadoc usuelles

Cette liste n'est pas exhaustive. Les commentaires Javadoc peuvent utiliser HTML pour leur mise en forme.

A.2.2 Les bons commentaires Javadoc

Idéalement, tous les attributs, toutes les méthodes et tous les constructeurs doivent être commentés : cela permet de comprendre le fonctionnement de la classe sans même avoir besoin de regarder le code, la Javadoc suffit. C'est d'autant plus utile quand l'utilisateur du code n'a a disposition que les fichiers compilés, mais pas le code source. Une phrase doit expliquer clairement le rôle de l'entité commentée. Le rôle de chaque paramètre doit être expliqué, ainsi que la signification de la valeur de retour des méthodes. Les éventuelles restrictions et contraintes doivent être données (par exemple, le prix d'un produit ne peut pas être négatif). Par défaut, seule la

Javadoc des membres `public` est générée ; cela peut être modifié via la ligne de commande de l'outil `javadoc` ou l'interface de l'IDE.

A.2.3 Exemple de Javadoc

```
package up.mi.jgm.util;

/**
 * Classe utilitaire qui permet la manipulation de tableaux
 *
 * @author Jean-Guy Mailly
 * @version 1.0
 *
 */
public class UtilTab {

/**
 * Permet l'affichage des elements d'un tableau, en ligne ou en colonne
 *
 * @param tab    le tableau a afficher
 * @param enLigne booleen qui vaut true si le tableau doit etre affiche en ligne
 */
 public static void affichageTableau(double[] tab, boolean enLigne) {
  // ...
 }

/**
 * Methode qui permet de determine si une valeur appartient au tableau
 *
 * @param val la valeur dont on teste l'appartenance
 * @param tab le tableau
 * @return true si et seulement si val appartient a tab
 */
public static boolean appartient(double val, double[] tab) {
  // ...
}

/**
 * Methode qui determine l'element minimal d'un tableau
 *
 * @param tab le tableau dont on cherche le minimum
 * @return le minimum de tab
 */
public static double min(double[] tab) {
  // ...
}

/**
```

```
 * Methode qui determine l'element maximal d'un tableau
 *
 * @param tab le tableau dont on cherche le maximum
 * @return le maximum de tab
 */
public static double max(double[] tab) {
 // ...
}

/**
 * Methode qui calcule la somme des elements d'un tableau
 *
 * @param tab le tableau dont on calcule la somme
 * @return la somme des elements de tab
 */
public static double somme(double[] tab) {
  // ...
}

/**
 * Methode qui trie un tableau par selection. Le tri par selection consiste a chercher
 * l'element le plus petit du tableau, et a l'echanger avec le premier element.
 * Le processus est reitere pour le deuxieme plus petit qui est echange avec le deuxieme
 * element du tableau, etc, jusqu'a ce que le tableau soit trie.
 * Cette methode modifie le tableau.
 *
 * @param tab le tableau a trier.
 */
public static void triParSelection(double[] tab) {
  // ...
}

/**
 * Methode qui permet d'obtenir l'indice du plus petit element d'un tableau a partir
 * d'une position donnee
 *
 * @param tab     le tableau dont on cherche le plus petit element
 * @param indiceMin la position a partir de laquelle on recherche le plus petit element
 * @return l'indice du plus petit element de tab situe apres la position indiceMin
 */
private static int rechercheIndicePlusPetit(double[] tab, int indiceMin) {
  // ...
}

/**
 * Methode qui echange deux elements d'un tableau, donnes par leur position.
 * Modifie le tableau.
 * @param tab le tableau dans lequel on echange deux elements
```

```
 * @param i l'indice du premier element a echanger
 * @param j l'indice du second element a echanger
 */
private static void echanger(double[] tab, int i, int j) {
  // ...
}

}
```

Voici un aperçu de la Javadoc générée.

Method Summary

All Methods	Static Methods	Concrete Methods

Modifier and Type	Method and Description
`static void`	**`affichageTableau`**`(double[] tab, boolean enLigne)` Permet l'affichage des elements d'un tableau, en ligne ou en colonne
`static boolean`	**`appartient`**`(double val, double[] tab)` Methode qui permet de determine si une valeur appartient au tableau
`private static void`	**`echanger`**`(double[] tab, int i, int j)` Methode qui echange deux elements d'un tableau, donnes par leur position
`static double`	**`max`**`(double[] tab)` Methode qui determine l'element maximal d'un tableau
`static double`	**`min`**`(double[] tab)` Methode qui determine l'element minimal d'un tableau
`private static int`	**`rechercheIndicePlusPetit`**`(double[] tab, int indiceMin)` Methode qui permet d'obtenir l'indice du plus petit element d'un tableau a partir d'une position donnee
`static double`	**`somme`**`(double[] tab)` Methode qui calcule la somme des elements d'un tableau
`static void`	**`triParSelection`**`(double[] tab)` Methode qui trie un tableau par selection.

Method Detail

affichageTableau

```
public static void affichageTableau(double[] tab,
                                    boolean enLigne)
```

Permet l'affichage des elements d'un tableau, en ligne ou en colonne

Parameters:

`tab - le tableau a afficher`

`enLigne - booleen qui vaut true si et seulement si le tableau doit etre affiche en ligne`

appartient

```
public static boolean appartient(double val,
                                 double[] tab)
```

Methode qui permet de determine si une valeur appartient au tableau

Parameters:

`val - la valeur dont on teste l'appartenance`

`tab - le tableau`

Returns:

`true si et seulement si val appartient a tab`

triParSelection

```
public static void triParSelection(double[] tab)
```

Methode qui trie un tableau par selection. Le tri par selection consiste a chercher l'element le plus petit du tableau, et a l'echanger avec le premier element. Le processus est reitere pour le deuxieme plus petit qui est echange avec le deuxieme element du tableau, etc, jusqu'a ce que le tableau soit trie.

Parameters:

`tab - le tableau a trier.`

rechercheIndicePlusPetit

```
private static int rechercheIndicePlusPetit(double[] tab,
                                            int indiceMin)
```

Methode qui permet d'obtenir l'indice du plus petit element d'un tableau a partir d'une position donnee

Parameters:

`tab - le tableau dont on cherche le plus petit element`

`indiceMin - la position a partir de laquelle on recherche le plus petit element`

Returns:

`l'indice du plus petit element de tab situe apres la position indiceMin`

echanger

Annexe B Manipulation de chaînes de caractères

On a vu dans ce cours les bases de la manipulation de chaînes de caractères. On a alors évoqué un « soucis » : les `String` ne sont pas des objets modifiables, ce qui signifie que lorsqu'on croit modifier des chaînes de caractères, ce qui se passe en réalité (et que Java nous cache) c'est la création de nombreux objets (possiblement, des objets temporaires qui seront rapidement détruits). Ces créations et destructions d'objets peuvent avoir un impact sur le temps d'exécution du programme. Nous allons donc voir un moyen de limiter l'impact des manipulations de chaînes de caractères, grâce à deux classes : `StringBuilder` et `StringBuffer`.

B.1 Les classes StringBuilder et StringBuffer

Conceptuellement, les classes `java.lang.StringBuffer` et `java.lang.StringBuilder` représentent des choses très similaires : des chaînes de caractères modifiables avec différentes méthodes pour les manipuler :

- `append(...)` pour ajouter quelque chose à la fin de la chaîne,
- `delete(...)` pour retirer une partie de la chaîne,
- `insert(...)` pour insérer quelque chose dans la chaîne,
- `replace(...)` pour remplacer une partie de la chaîne,
- `reverse(...)` pour retourner la chaine.

On peut donc éviter de nombreuses créations (et destructions) inutiles d'objets `String` en utilisant ces méthodes. Si on doit récupérer un objet `String` à la fin des manipulations de la chaine (par exemple, pour respecter la signature d'une méthode `toString()`), il suffit d'utiliser la méthode `toString()` du `StringBuffer` ou `StringBuilder` à la fin des manipulations.

La différence entre les deux est simple : `StringBuffer` est une classe synchronisée, ce qui signifie qu'elle est sécurisée dans un contexte multi-threads, contrairement à `StringBuilder`. Donc, on préfère

- `StringBuilder` dans une situation mono-thread pour gagner en efficacité,
- `StringBuffer` dans une situation multi-threads pour gagner en sécurité.

Comme pour l'instant on ne fait que de la programmation mono-thread, on utilisera donc principalement la classe `StringBuilder`. On pourra être amenés à utiliser `StringBuffer` une fois qu'on aura étudié les threads.

Comme c'est toujours le cas, vous êtes invités à voir la Javadoc de ces classes pour plus de détails à leur sujet :

- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/StringBuilder.html
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/StringBuffer.html

B.2 Exemple

Pour constater la différence d'efficacité des manipulations de chaînes via la classe `String` ou les classes `StringBuilder` et `StringBuffer`, on considère un petit exemple simple.

```
package testbuilder;

public class TestStringBuilder {
```

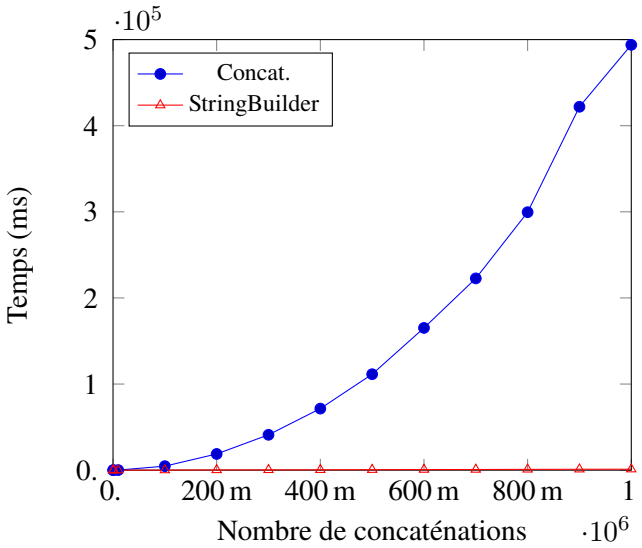
```
  public static void main(String[] args) {
    int upperBound = 100000;

    long startingTime = System.currentTimeMillis();
    String str = "" ;
    for(int i = 0 ;i < upperBound ;i++) {
      str += "test " + i + "\n" ;
    }
    System.out.println(str);
    long endingTime = System.currentTimeMillis() ;

    long stringTime = endingTime - startingTime;

    startingTime = System.currentTimeMillis();
    StringBuilder builder = new StringBuilder();
    for(int i = 0 ;i < upperBound ;i++) {
      builder.append("test") ;
      builder.append(Integer.toString(i));
      builder.append("\n");
    }
    System.out.println(builder.toString());
    endingTime = System.currentTimeMillis();

    long builderTime = endingTime - startingTime;

    System.out.println("stringTime = " + stringTime + " - builderTime = " + builderTime);

  }
}
```

Ce programme concatène 100000 fois des chaînes de caractères avant d'afficher le résultat de la concaténation, et réalise la même opération via une instance de `StringBuilder`. Dans les deux cas, le temps de calcul est mesuré, avant d'être affiché. Pour s'éviter l'affichage de deux fois 100000 lignes, nous allons ici nous contenter de donner l'affichage obtenu avec le dernier appel à `System.out.println` :

```
stringTime = 4787 - builderTime = 168
```

Cela signifie que l'utilisation d'un `StringBuilder` permet d'obtenir le résultat presque 30 fois plus vite qu'avec des concaténations de `String` ! Bien entendu, les temps obtenus peuvent varier un peu selon la machine utilisée, mais l'idée générale est la même : le `StringBuilder` permet d'éviter une perte de temps importante causée par la création et la destruction d'objets temporaires.

Pour mieux visualiser l'impact de l'utilisation du `StringBuilder`, voici le résultat d'une petite expérience pour laquelle on a fait varier la valeur de la variable `upperBound` :



La courbe bleue représente le temps nécessaire pour faire ces concaténations avec la classe `String` (en fonction du nombre de concaténations qu'on réalise, entre 100 et 1000000), alors que la courbe rouge représente le temps nécessaire lorsqu'on utilise la classe `StringBuilder` : on voit que même avec 1000000 de « concaténations », il faut au plus une seconde (au lieu de 493 secondes avec la classe `String`).