

Programmation Avancée et Application

Introduction à Java

Jean-Guy Mailly

`jean-guy.mailly@u-paris.fr`

LIPADE - Université de Paris

<http://www.math-info.univ-paris5.fr/~jmailly/>

1. Au sujet du module
2. Les bases de Java
3. Types, variables, tableaux
4. Instructions, boucles, conditions
5. Entrées et sorties de base
6. Premières classes

Au sujet du module

- Jean-Guy Mailly : jean-guy.mailly@u-paris.fr, Bureau 814 I
- 18h de cours : lundi, 11h15–12h45, Polonovski
- 36h de TD : mercredi, 8h30–11h30, Cordier 523A (J. Delobelle)
mercredi, 15h45–18h45, Fourier A526 (J.-G. Mailly)
jeudi, 14h00–17h00, Fourier D529 (J.-G. Mailly)
jeudi, 17h00–20h00, Cordier 523A (J. Delobelle)

Au sujet du module

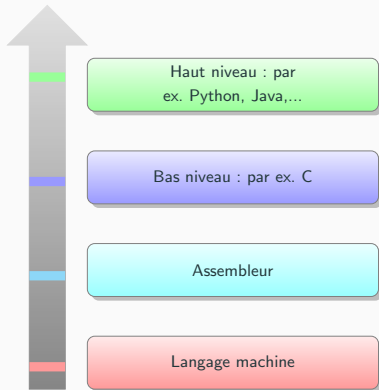
- Jean-Guy Mailly : jean-guy.mailly@u-paris.fr, Bureau 814 I
- 18h de cours : lundi, 11h15–12h45, Polonovski
- 36h de TD : mercredi, 8h30–11h30, Cordier 523A (J. Delobelle)
mercredi, 15h45–18h45, Fourier A526 (J.-G. Mailly)
jeudi, 14h00–17h00, Fourier D529 (J.-G. Mailly)
jeudi, 17h00–20h00, Cordier 523A (J. Delobelle)
- Modalités de contrôle de connaissances :
 - Contrôle continu : un contrôle durant le semestre (CC), un projet (P) et un contrôle terminal (CT)
 - Note finale : $\frac{CC}{4} + \frac{P}{4} + \frac{CT}{2}$

- Jean-Guy Mailly : jean-guy.mailly@u-paris.fr, Bureau 814 I
- 18h de cours : lundi, 11h15–12h45, Polonovski
- 36h de TD : mercredi, 8h30–11h30, Cordier 523A (J. Delobelle)
mercredi, 15h45–18h45, Fourier A526 (J.-G. Mailly)
jeudi, 14h00–17h00, Fourier D529 (J.-G. Mailly)
jeudi, 17h00–20h00, Cordier 523A (J. Delobelle)
- Modalités de contrôle de connaissances :
 - Contrôle continu : un contrôle durant le semestre (CC), un projet (P) et un contrôle terminal (CT)
 - Note finale : $\frac{CC}{4} + \frac{P}{4} + \frac{CT}{2}$
- Moodle : **IF05X030 Programmation Avancée et Application**

<https://moodle.u-paris.fr/course/view.php?id=12140>

Les bases de Java

Différents niveaux de langages

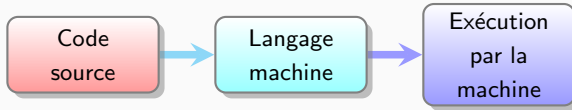


- Langage machine : séquence de 0 et de 1
- Assembleur : représentation du langage machine lisible pour un humain
- Bas niveau : proche de la machine, besoin de gérer soi-même la mémoire, etc
- Haut niveau : le programmeur se concentre sur le problème à résoudre plutôt que sur la machine

Avantages de Java

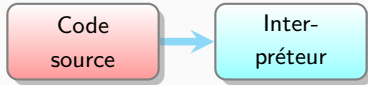
- Langage orienté objet
- Syntaxe « à la C » : similaire à C, C++, C#, Javascript, PHP
- Mécanismes complexes pris en charge :
 - Gestion automatisée de la mémoire
 - Sérialisation
 - Exceptions
- Librairie standard très développée (interfaces graphiques, réseau, bases de données,...)
- Portabilité
- À la base du développement d'applications Android

Rappel : langage compilé



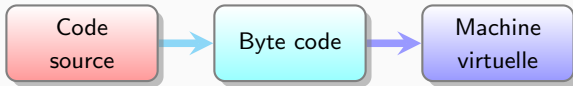
- La compilation est la transformation d'un code source en langage machine
- Le fichier produit par la compilation est directement exécuté par le système exploitation
- Exemple de langages compilés : C, C++, Cobol, Fortran, Pascal
- Avantage : généralement l'exécutable est optimisé pour le système d'exploitation
- Inconvénient : il faut compiler sur chaque système

Rappel : langage interprété



- L'interpréteur exécute directement le programme sans produire de code machine
- Exemple de langages interprétés : Bash, Python, Prolog
- Avantage : facilement portable, il suffit d'avoir un interpréteur disponible pour le système d'exploitation voulu
- Inconvénient : généralement moins efficace (pas d'optimisation)

Java : un langage intermédiaire



- Le code Java est compilé en byte code, un langage non compréhensible par le programmeur, mais différent du langage machine
- Le byte code n'est pas directement exécutable par le système d'exploitation : besoin d'une machine virtuelle
- Portabilité : il existe des machines virtuelles pour de nombreux systèmes d'exploitation ; le même byte code peut donc être utilisé directement sur différentes machines

- Java est régulièrement mis à jour.
- Version utilisée dans ce cours : Java SE 12 (sortie en mars 2019) ¹
- Distribution de Java SE sous deux formes :
 - Java Runtime Environment (JRE) : machine virtuelle + bibliothèque standard, nécessaire pour exécuter un programme Java
 - Java Development Kit (JDK) : JRE + compilateur, nécessaire pour développer un programme Java
- Bibliothèque standard = API standard (application programming interface)
- Machine virtuelle = JVM (Java Virtual Machine)

1. Pour info, Java 16 est sortie en mars 2021. Nous utiliserons la version disponible à l'Université.

Qu'est-ce qu'un programme Java ?

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

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

Qu'est-ce qu'une classe ? (grossièrement)

- structure de données
- fonctions pour manipuler ces données

Comment distribuer un programme Java ?

- Si une JVM est disponible pour un système d'exploitation, alors elle est forcément associée aux classes de l'API standard
- Pour distribuer un programme Java, il suffit donc de distribuer les classes écrites par le développeur (et éventuellement les classes des API tierces)
- Deux méthodes
 - Distribution des fichiers compilés en byte code (fichier `.class`)
 - Distribution d'une archive `jar` qui contient les classes

Comment distribuer un programme Java ?

- Si une JVM est disponible pour un système d'exploitation, alors elle est forcément associée aux classes de l'API standard
- Pour distribuer un programme Java, il suffit donc de distribuer les classes écrites par le développeur (et éventuellement les classes des API tierces)
- Deux méthodes
 - Distribution des fichiers compilés en byte code (fichier `.class`) ✗
 - Distribution d'une archive `jar` qui contient les classes ✓

Compilation d'une classe Java

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
```

Commande de base pour compiler une classe Java :

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

Exécution d'une classe 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
```

Compilation de plusieurs classes Java

On a maintenant plusieurs classes :

```
$ ls
```

```
ClasseA.java ClasseB.java
```

Compilation de plusieurs classes :

```
$ javac *.java
```

```
$ ls
```

```
ClasseA.class ClasseA.java ClasseB.class ClasseB.java
```

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
```

Compilation d'un projet

- 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.

Compilation d'un projet avec classes externes (1/2)

- On peut également indiquer au compilateur qu'il doit utiliser des classes prédéfinies
- 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
```

Compilation d'un projet avec classes externes (2/2)

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
```

Exécution d'un projet avec classes externes

Si toutes les classes ne sont pas dans le répertoire courant, il faut également indiquer 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 `classpath`, pas besoin de les indiquer

Hello, World !

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

- Nous reviendrons en détail sur les mots-clés par la suite
- `System.out.println` permet d'afficher un message sur la sortie standard.
- Ce programme 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`

La méthode main

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

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

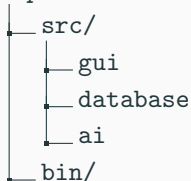
Sans rentrer dans les détails pour l'instant,

- `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
 - Plus de détails sur l'utilisation des tableaux plus tard
- `void` : la méthode ne renvoie aucune valeur
- `public` et `static` : plus tard...

Organisation du projet en packages (1/2)

- Le code source d'un programme Java est découpé en packages
- Chaque package réunit un ensemble de classes qui ont un lien « logique » entre elles
- Par exemple :
 - Un package pour l'interface graphique
 - Un package pour la gestion de la base de données
 - Un package pour l'intelligence artificielle
 - ...
- Cela se manifeste par la création de répertoires équivalents aux packages

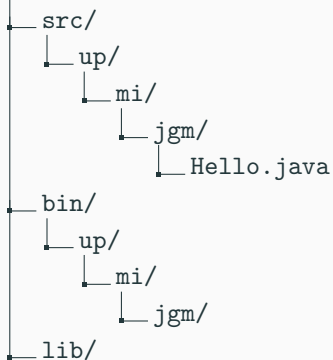
repertoirePrincipal



Organisation du projet en packages (2/2)

- 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



```
package up.mi.jgm ;

public class Hello {
    ...
}
```

Compilation du projet avec packages

Depuis le répertoire principal :

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

Compilation du projet avec packages

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).

Compilation du projet avec packages

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.

Exécution du projet avec packages

Depuis le répertoire principal :

```
java -cp bin/up/mi/jgm/:lib/ up.mi.jgm.Hello
```


Convention pour les noms de packages

- Les packages `java` et `javax` sont (par convention) réservés à l'API standard
- Par convention, l'arborescence des packages d'un projet doit correspondre au nom de domaine du site web associé au projet, dans l'ordre inversé
 - Par exemple, un projet développé par la société dont le site web est `exemple.fr` aura pour racine le package `fr.exemple.monprojet`
 - Un projet déposé sur `sourceforge.net` aura pour racine `net.sf.monprojet`
- Pour les exemples du cours, j'utiliserai `up.mi.jgm`, et je vous invite à respecter la même convention pour vos projets

Nom complet d'une classe

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

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

Import d'une classe

- Le mot clé `import` 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)

Classes homonymes

- Plusieurs classes définies dans différents packages peuvent avoir le même nom
- Une seule (au plus) peut être importée
- Il faut alors utiliser les noms complets

```
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();  
}
```

Attention aux classes homonymes !

- 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 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 ex., 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)

Compilation et utilisation d'archives jar

- Une archive jar 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)
- 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`
- Ne pas oublier d'ajouter le jar au classpath avec l'option `-cp`

Compilation et utilisation d'un jar exécutable

- On peut rendre un jar exécutable en indiquant la classe principale du programme (celle qui contient la méthode `main`)
- Création de l'archive :
`jar cvfe JARFILE MAINCLASS INPUTFILES`
- Exécution de l'archive :
`java -jar JARFILE`

- 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
 - Le créateur de jar
 - Une interface simplifiée pour la gestion de packages
 - Des outils de débogage
 - Plein d'autres choses...
- Nous verrons en TP comment utiliser Eclipse

Types, variables, tableaux

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

Types primitifs : les entiers

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

Types primitifs : les décimaux

- 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

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

Types primitifs : les booléens

Deux valeurs de vérité constantes boolean

- `true`
- `false`

Remarque : l'espace mémoire utilisé dépend de la JVM

Types primitifs : les booléens

Deux valeurs de vérité constantes boolean

- `true`
- `false`

Remarque : l'espace mémoire utilisé dépend de la JVM

⚠ Pas de conversion automatique des entiers en booléens (contrairement au langage C)

Types primitifs : les caractères

- 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

Caractères spéciaux

Retour à la ligne	<code>\n</code>
Tabulation	<code>\t</code>
Apostrophe	<code>\'</code>
Double apostrophe	<code>\"</code>
Backslash	<code>\\</code>
Caractère Unicode	<code>\uxxxx</code>

Déclaration de variables

Syntaxe générale : `type nomVariable ;`

- `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

Affectation en deux temps :

```
float x ;
```

```
x = 5.2 ;
```

Affectation en deux temps :

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

Affectation lors de la 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

- On peut déclarer et affecter plusieurs variables de même type en même temps :
 - `int a, b, c ;`
 - `int a = 2, b = 3, c = 4 ;`

Conversions implicites

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

1. byte
2. short
3. int
4. long
5. float
6. double

Exemples :

```
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
```

Conversions implicites

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

1. byte
2. short
3. int
4. long
5. float
6. double

Exemples :

```
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
```

Conversion implicite 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`

Exemples :

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

Conversions explicites

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

Exemples :

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

⚠ Pas de conversion entre nombres et booléens

Variantes de l'affectation

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

Rappel : 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
 - $\%$ est le modulo (reste de la division euclidienne). Par exemple, $5 \% 2$ donne 1

Rappel : comparaisons

- $x == y$ 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 ($<=$)
- Ces opérations retournent une valeur booléenne
- ⚠ 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

Rappel : opérateurs booléens

Symbole	Nom	Exemple	Court circuit
!	Négation	$!(x == y)$	
&	Et	$(x > y) \& (x > z)$	non
&&	Et	$(x > y) \&\& (x > z)$	oui
	Ou	$(x > y) (x > z)$	non
	Ou	$(x > y) (x > z)$	oui
^	Ou exclusif	$(x > y) ^ (x > z)$	

- Court circuit : évalue uniquement la première moitié de l'expression si c'est possible
- Ou exclusif : l'expression est vraie si **exactement une** des opérandes est vraie
- En logique, on parle généralement de conjonction (**et**) et de disjonction (**ou**)

Priorité des opérations

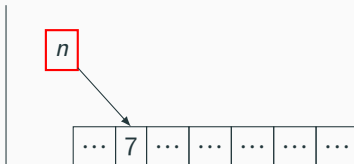
Priorité	Opérateurs
1	[] ()
2	++ -- ! ~ instanceof
3	* / %
4	+ -
5	<< >> >>>
6	< > <= >=
7	== !=
8	&
9	^
10	
11	&&
12	
13	? :
14	= += -= %= *= /=

- En cas de priorité identique, on exécute de gauche à droite
- En cas de doute, utiliser des parenthèses pour simplifier

Affectations et passages de paramètres pour les types primitifs

- Recopie de la valeur des variables pour les affectations et les paramètres des méthodes

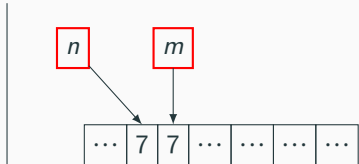
```
int n = 7 ;
```



Affectations et passages de paramètres pour les types primitifs

- Recopie de la valeur des variables pour les affectations et les paramètres des méthodes

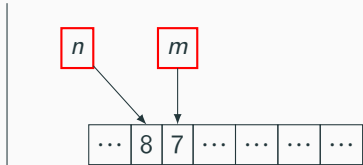
```
int n = 7 ;  
int m = n ;
```



Affectations et passages de paramètres pour les types primitifs

- Recopie de la valeur des variables pour les affectations et les paramètres des méthodes

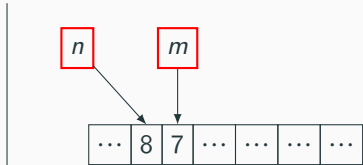
```
int n = 7 ;  
int m = n ;  
n = n + 1 ;
```



Affectations et passages de paramètres pour les types primitifs

- Recopie de la valeur des variables pour les affectations et les paramètres des méthodes

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

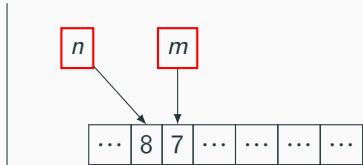


- Même fonctionnement pour les paramètres de méthodes : la valeur de la variable passée en paramètre n'est pas modifiée hors de l'exécution de la méthode

Affectations et passages de paramètres pour les types primitifs

- Recopie de la valeur des variables pour les affectations et les paramètres des méthodes

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



- 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

- Tableau : collection ordonnée de valeurs du même type
- Chaque valeur est identifiée par sa position dans le tableau : *indice*
- Pour un tableau à n éléments, les indices vont de 0 à $n - 1$
- Déclaration et initialisation :
`type [] nom = new type[longueur]`

Exemple :

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

initialise un tableau de 5 éléments de type `int`

Tableaux : initialisation des éléments

- 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 intialisation, les éléments du tableau prennent une valeur par défaut

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

Longueur d'un tableau

- Les tableaux sont des objets qui disposent d'un attribut `length` qui indique leur longueur, c'est-à-dire le nombre d'élément
- 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

Instructions, boucles, conditions

- Déclaration/affectation/incrémentation, ou appel d'une méthode
 - On reviendra plus tard sur les différents types de méthodes et leur utilisation
- Toute instruction simple se termine par un pont-virgule ;

Exemple : deux instructions simples

```
int n = 0 ;
```

```
System.out.println("n_=_ " + n) ;
```

Instructions composées : les blocs

- 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 : un bloc composé de deux instructions simples et d'un 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 jusqu'à la fin du bloc
- Elle est également accessible dans les blocs internes au bloc de sa déclaration

```
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

Variables homonymes (1/2)

- Une variable locale à une méthode peut porter le même nom qu'une variable de classe (c'est-à-dire un attribut)
 - C'est en général déconseillé pour éviter des erreurs !
 - Nous reparlerons de cela dans la partie dédiée aux classes
- Par contre, deux variables locales à la même méthode ne peuvent pas porter le même nom, même si elles ne sont pas définies dans le même bloc

Variables homonymes (2/2)

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

→ erreur de compilation :

Test.java:5: error: variable n is already defined in
method main(String[])

```
    int n = 1 ;  
        ^
```

1 error

Boucles while

- Répétition d'un bloc d'instructions tant qu'une condition est vraie
- La 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

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

Boucles while

⚠ 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, pas d'itération

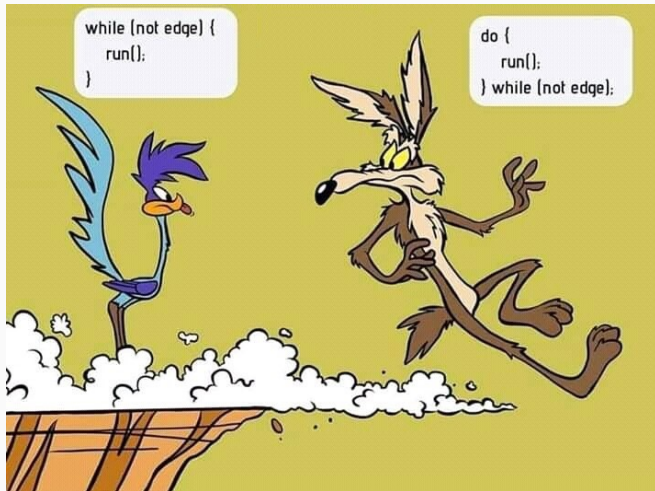
```
boolean b = false ;  
while(b){  
    // ces instructions ne sont pas executees  
}  
// ces instructions sont executees
```

- 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

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

⚠ Ne pas oublier le point-virgule après le `do-while`

Différence entre while et do-while



Boucles for

- Similaire au `while` : la boucle `for` teste une certaine condition, et répète un bloc d'instructions tant qu'elle est vraie
- 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  
}
```



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

est équivalent à

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

- Variante du `for` pour parcourir les éléments d'un tableau ou d'une collection
 - On reviendra plus tard sur les collections

⚠ Utilise le même mot clé que le `for` classique, mais la syntaxe est différente

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

Boucles for each

Exemple :

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

- Le mot clé `break` permet de sortir d'une boucle et de passer à l'exécution des instructions situées après la boucle
- Le mot clé `continue` permet de passer à l'itération suivante d'une boucle

- `if` permet d'exécuter un bloc d'instructions seulement si une condition est vérifiée
- `if` 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

Les conditions : 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.");
}
```

Les conditions : switch case

- Permet de choisir un bloc d'instructions à effectuer en fonction de la valeur d'une expression
- L'expression peut être :
 - de type primitif `byte`, `short`, `char`, `int`
 - de type énuméré (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!).

Les conditions : switch case

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

- Sans le mot clé `break`, 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

Les conditions : switch case

```
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

Les conditions : switch case

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

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

- 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

Les conditions : les ternaires

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

est équivalent à

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

Les conditions : les ternaires

```
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");
```

Entrées et sorties de base

La classe String

- La classe `String` est définie dans Java pour représenter les chaînes de caractères
- Elle est présente dans le package `java.lang` : 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
 - Nous reviendrons plus tard sur la création des objets et ce que cela implique du point de vue de la mémoire

La classe String : méthodes

- `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 classe String : concaténation

- L'opérateur `+` est utilisé pour la concaténation de `String`
- Par exemple :

`"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"`

La classe `String` : concaténation

- Les valeurs de types primitifs peuvent être converties automatiquement en `String` pour la concaténation. Par exemple :
 - l'entier 5 devient la chaîne "5"
 - le décimal 5.0 devient la chaîne "5.0"
 - le booléen (5 == 5) devient la chaîne "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 (ni pertinente en générale), il est donc recommandé de définir votre propre `toString` dans vos classes

- 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`

- `System.out` et `System.err` sont deux objets de même type (`java.io.PrintStream` sur lequel nous reviendrons plus tard)
- Pour débiter, 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

- `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` !

Entrées basiques : méthodes du Scanner

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

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

Par exemple :

```
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`

⚠ Toujours fermer le scanner avec `close()` après la fin de son utilisation

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
- Exemple : une calculatrice (très) simple

Exemple de programme simple : 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.in);  
        int choix, n1, n2;  
        ...  
    }  
}
```


Exemple de programme simple : la boucle

```
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);  
sc.close();
```

- On propose différents choix à l'utilisateur (0, 1 ou 2)
- 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

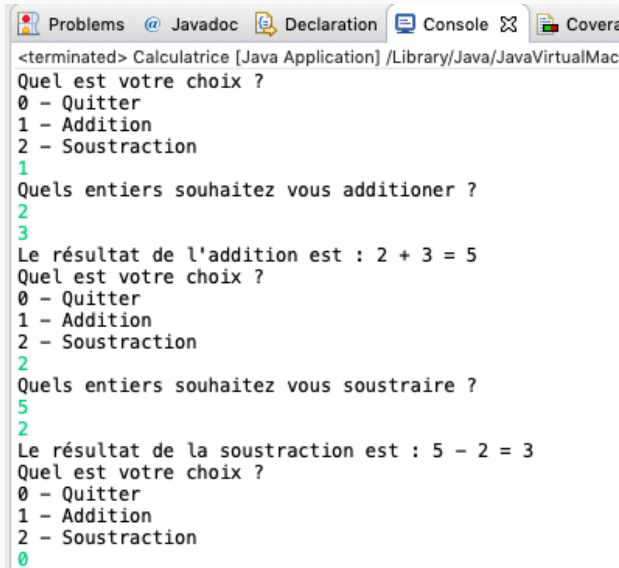
Exemple de programme simple : le switch-case (1/2)

```
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;  
}
```

Exemple de programme simple : le switch-case (2/2)

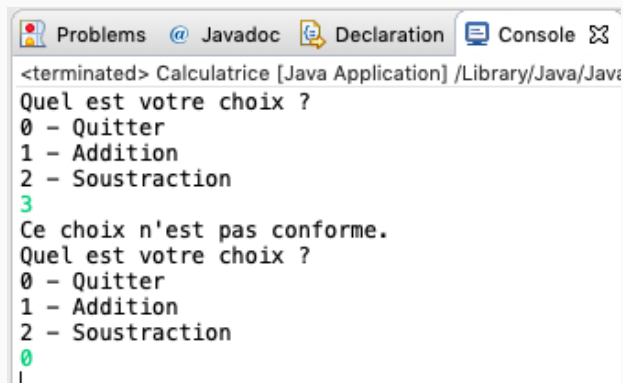
```
case 2:
    System.out.println("Quels_entiers_souhaitez_"
                        + "vous_soustraire?");
    n1 = sc.nextInt();
    n2 = sc.nextInt();
    System.out.println("Le_resultat_de_la_soustraction_"
                        + "est:_ " + n1 + "_-"
                        + n2 + "_=" + (n1 - n2));
    break;
default:
    System.out.println("Ce_choix_n'est_pas_conforme.");
}
```

Exemple de programme simple



```
<terminated> Calculatrice [Java Application] /Library/Java/JavaVirtualMac
Quel est votre choix ?
0 - Quitter
1 - Addition
2 - Soustraction
1
Quels entiers souhaitez vous additionner ?
2
3
Le résultat de l'addition est : 2 + 3 = 5
Quel est votre choix ?
0 - Quitter
1 - Addition
2 - Soustraction
2
Quels entiers souhaitez vous soustraire ?
5
2
Le résultat de la soustraction est : 5 - 2 = 3
Quel est votre choix ?
0 - Quitter
1 - Addition
2 - Soustraction
0
```

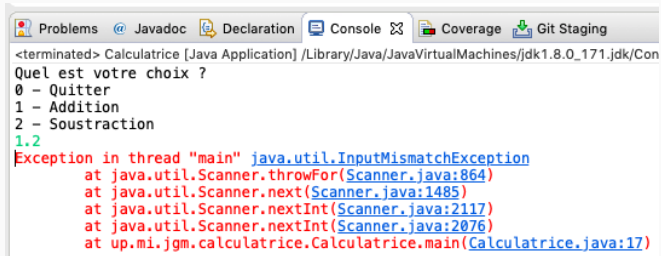
Exemple de programme simple



The screenshot shows an IDE window with a tab labeled 'Console'. The console output is as follows:

```
<terminated> Calculatrice [Java Application] /Library/Java/Java
Quel est votre choix ?
0 - Quitter
1 - Addition
2 - Soustraction
3
Ce choix n'est pas conforme.
Quel est votre choix ?
0 - Quitter
1 - Addition
2 - Soustraction
0
|
```

Exemple de programme simple



```
<terminated> Calculatrice [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_171.jdk/Con
Quel est votre choix ?
0 - Quitter
1 - Addition
2 - Soustraction
1.2
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:864)
    at java.util.Scanner.next(Scanner.java:1485)
    at java.util.Scanner.nextInt(Scanner.java:2117)
    at java.util.Scanner.nextInt(Scanner.java:2076)
    at up.mi.jgm.calculatrice.Calculatrice.main(Calculatrice.java:17)
```

Nous verrons plus tard la gestion des erreurs

Premières classes

Qu'est-ce qu'une classe ?

- Grossièrement, une classe est une structure de données accompagnée d'opérations pour manipuler les données
 - Données = attributs : variables définies dans la classe
 - Opérations = méthodes : fonctions définies dans la classe
 - Les attributs et les méthodes forment l'ensemble des membres de la classe

- 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
- Signature d'une méthode :
 - Liste de modificateurs
 - Type de retour
 - Nom de méthode
 - Liste de paramètres : type et nom de chaque paramètre (on parle aussi d'arguments)

Exemple :

```
public int somme(int n, int m){  
    return n + m ;  
}
```

- Modificateurs d'accès :
 - `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`,...)

Arguments variables

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

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

- Utilisation :

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

- 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`

Principe : mieux vaut de nombreuses méthodes courtes que peu de méthodes longues

- Factorisation : on peut réutiliser facilement un même morceau de code à plusieurs endroits
- Plus facile de lire et comprendre le code → plus facile de tester, déboguer, maintenir