

Programmation et Algorithmique II

Ch.1 – Introduction à Java

Bruno Quoitin
(bruno.quoitin@umons.ac.be)

Chapitre I

- **Objectifs du Chapitre**

- Premier aperçu d'un nouveau langage de programmation : **Java**.
- Ecrire et compiler un premier programme Java.
- Déclaration et utilisation de **variables**
- **Types de données** primitifs et chaînes de caractères
- Expressions et opérateurs
- **Structures de contrôle** du flux d'exécution du programme
- Définitions et appels de sous-programmes (**méthodes**)

Table des Matières

- 1. Le Langage Java**
2. Compilation et exécution
3. Variables, types et littéraux
4. Opérateurs et expressions
5. Chaînes de caractères
6. Méthodes
7. Structures de contrôle
8. Ecrire des commentaires

Avantages et Objectifs de Java

- **Simplicité**
 - langage *relativement* simple ; syntaxe proche de C, C++.
- **Portabilité**
 - indépendant de la plateforme (architecture et système d'exploitation).
- **Bibliothèque (« API Java »)**
 - grand nombre de méthodes / classes / structures de données.
 - Impossible d'en connaître l'entièreté.
- **Sécurité (*safety*)**
 - abstraction du matériel, typage fort et restrictions d'accès.
- **Gratuité**
 - outils de développement disponibles gratuitement.

Langage Java

- Langage en perpétuelle évolution

Version	Année	Améliorations
1.0	1996	
1.1	1997	Classes internes
1.2	1998	API Swing, Collections
1.3	2000	Amélioration des performances
1.4	2002	Assertions, XML
5.0	2004	Classes génériques, boucles for améliorées, énumérations, ...
6	2006	Amélioration des bibliothèques
7	2011	try-with-resource; littéraux binaires, ...
8 (LTS)	2014	lambda expressions
9	2017	jshell (REPL)
10	mars 2018	inférence type variable locale
11 (LTS)	sept. 2018	plus d'inférence type variable locale
12	Mars 2019	...
...
17 (LTS)	sept 2021	...

1 release
tous les 6
mois

Langage Java

- **Premier aperçu**

- Cet exemple donne un premier aperçu de la structure d'un programme Java. Il s'agit du « plus petit » programme Java.

```
public class HelloPrinter
{
    public static void main(String[] args)
    {
        // Affiche un message dans la console
        System.out.println("Hello, World!");
    }
}
```

- Lorsque ce programme est exécuté (en console), il affiche le résultat suivant

```
Hello, World!
```

Langage Java

- Premier aperçu : mots réservés

```
public class HelloPrinter
{
    public static void main(String[] args)
    {
        // Affiche un message dans la console
        System.out.println("Hello, World!");
    }
}
```

`public`, `class`, `static` et `void` sont des mots réservés du langage.

Langage Java

- Premier aperçu : identifiants

```
public class HelloPrinter
{
    public static void main(String[] args)
    {
        // Affiche un message dans la console
        System.out.println("Hello, World!");
    }
}
```



- HelloPrinter est le nom de la classe.
- main est un nom de méthode.
- args est un nom de variable (argument de méthode).

HelloPrinter, main et args sont des identifiants. Il existe des restrictions sur les caractères qui peuvent être utilisés pour former un identifiant.

Langage Java

- Premier aperçu : classe

```
public class HelloPrinter
{
    public static void main(String[] args)
    {
        // Affiche un message dans la console
        System.out.println("Hello, World!");
    }
}
```

Le bloc de base d'un programme Java est la **classe**, un concept important de la programmation orienté-objet.

Une classe

- a un **identifiant unique** : HelloPrinter
- est placée dans un fichier de même nom (ici : HelloPrinter.java)

Un fichier contient une seule classe.

Langage Java

- Premier aperçu : méthode

Une **méthode** est un sous-programme (similaire à une fonction en Python).

```
public class HelloPrinter
{
    public static void main(String[] args)
    {
        // Affiche un message dans la console
        System.out.println("Hello, World!");
    }
}
```

Note : La méthode `main` est le **point d'entrée** du programme. Seules les classes qui définissent une telle méthode sont exécutables directement.

- L'argument `args` contiendra les paramètres passés au programme en ligne de commande.

Langage Java

- Premier aperçu : blocs

```
public class HelloPrinter
{
    public static void main(String[] args)
    {
        // Affiche un message dans la console
        System.out.println("Hello, World!");
    }
}
```



Les accolades ouvrantes « { » et fermantes « } » délimitent un **bloc** du programme.

Ici, une paire d'accolades délimite le **bloc « classe »** et une autre paire délimite le **bloc « méthode »**.

Note : en Python, les blocs sont identifiés grâce à l'indentation.

Langage Java

- Premier aperçu : commentaires

```
public class HelloPrinter
{
    public static void main(String[] args)
    {
        // Affiche un message dans la console
        System.out.println("Hello, World!");
    }
}
```

Le langage Java permet l'ajout de **commentaires** sous 2 formes différentes :

- sur 1 ligne, commençant par //
- sur plusieurs lignes, commençant par /* et finissant par */

Langage Java

- Premier aperçu : instructions

```
public class HelloPrinter
{
    public static void main(String[] args)
    {
        // Affiche un message dans la console
        System.out.println("Hello, World!");
    }
}
```

Le **bloc** d'une méthode peut contenir

- des déclarations de variables
- des instructions (affectations, appels de méthode, tests de condition, boucles, ...)

Appel de méthode : la méthode `println` définie dans l'object `System.out` de type `PrintStream` de la bibliothèque (API) Java.

Langage Java

- **Documentation**

- Les méthodes/classes de l'API Java sont bien documentées. La documentation est disponible en ligne. Pour Java 11, à l'adresse <https://docs.oracle.com/en/java/javase/11/docs/api>

println

```
public void println(String x)
```

Prints a String and then terminate the line. This method behaves as though it invokes [print\(String\)](#) and then [println\(\)](#).

Parameters:

x - The String to be printed.



Vous devez avoir le réflexe de consulter la documentation !!!

Langage Java

- **Organisation du code source**

- Contrairement au langage Python, Java n'impose pas d'indentation particulière du code source
- Ainsi, l'exemple `HelloPrinter` pourrait aussi s'écrire comme ceci.

```
public
class
HelloPrinter
{
    public
    static
    void
    main (String[]
    args)
    {
        // Affiche un message dans la console
        System.out.println("Hello, World!"); }
}
```

Langage Java

- **Organisation du code source**

- ... ou encore comme ceci.

```
public class HelloPrinter { public static void  
main(String[] args) {  
    // Affiche un message dans la console  
    System.out.println("Hello, World!"); } }
```



Ces deux exemples sont bien moins lisibles. Ils illustrent CE QU'IL NE FAUT PAS FAIRE.

Le programmeur (vous!) est responsable de la lisibilité du programme \Rightarrow de son indentation correcte.

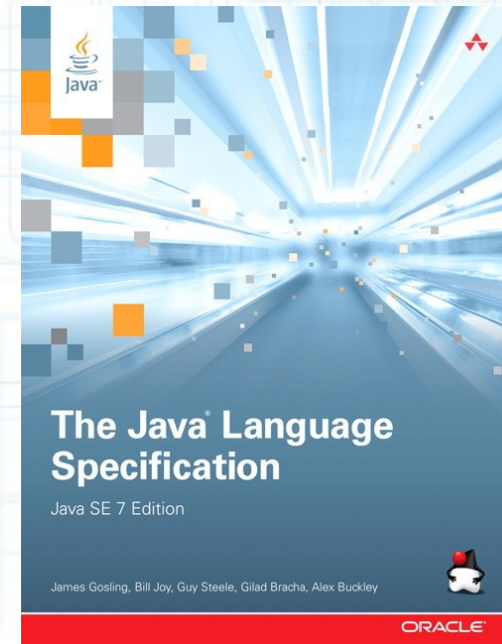
Plus l'organisation de vos travaux s'approchera de celle de ces exemples, plus la note obtenue s'approchera de 0. A bon entendeur... :-)

Langage Java

- **Spécification**

- Ce cours est une introduction au langage Java. Tous les détails ne seront pas couverts. En cas de question ou par curiosité, il est possible de consulter la spécification du langage Java.

- ***The Java Language Specification***,
Java SE 7 Edition, J. Gosling et al
February 2013
<http://docs.oracle.com/javase/specs>



Organisation d'un fichier Java

```
/* Exemple montrant l'organisation générale  
 * d'une classe Java  
 * © 1997, J. Gosling */
```

```
import java.util.Scanner;
```

Importer des classes ou groupes de classes (typiquement de la bibliothèque Java).

```
public class ClasseExemple {
```

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

Déclaration de **variable "globale"**.

```
    public static int fact(int n) {  
        if (n > 1)  
            return n * fact(n-1);  
        return 1;  
    }
```

Définition de **méthodes** (fonctions)

```
    public static void main(String [] args) {  
        System.out.print("Entrez un entier positif : ");  
        int reponse= fact(input.nextInt());  
        System.out.println(reponse);  
    }
```

Déclaration de **variable "locale"**.

```
}
```

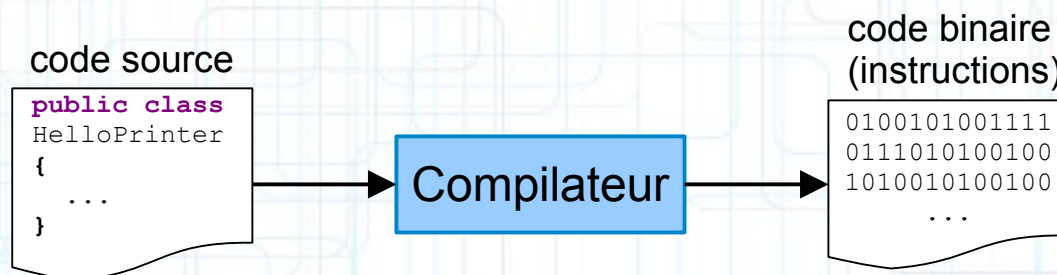
Table des Matières

1. Le Langage Java
- 2. Compilation et exécution**
3. Variables, types et littéraux
4. Opérateurs et expressions
5. Chaînes de caractères
6. Méthodes
7. Structures de contrôle
8. Ecrire des commentaires

Compilation et exécution

- **Compilation**

- La **compilation** est le processus qui va produire à partir du code source d'un programme une suite d'instructions exécutables.
- Le **compilateur** est l'outil qui se charge de la compilation. Il est utilisé par le développeur du programme.

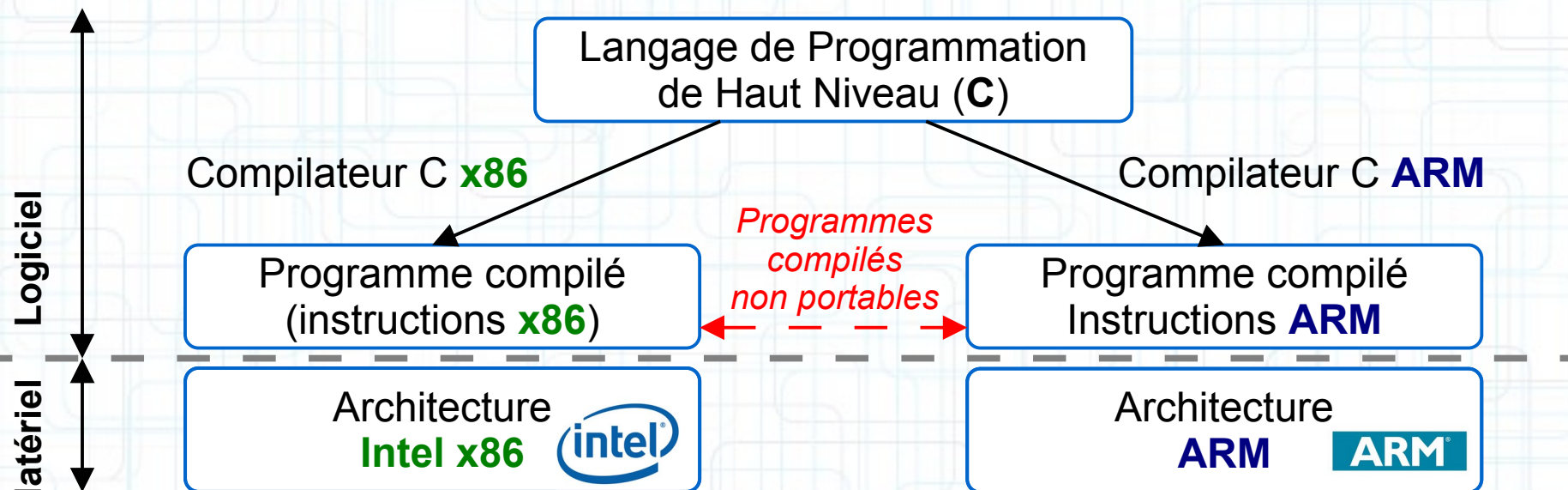


- Tâches principales du compilateur
 - **Vérification** de la conformité du code source au langage.
 - **Transformation** du code source en une série d'instructions élémentaires

Compilation et exécution

- **Compilation**

- Un langage tel que C/C++ doit être compilé pour fonctionner sur une **plateforme particulière**.
 - Le code compilé pour une plateforme particulière ne pourra fonctionner que sur cette plateforme. La plateforme dépend typiquement du **processeur** (x86, ARM, MIPS, ...) et du **système d'exploitation** (Windows, Linux, Mac OS X, ...).

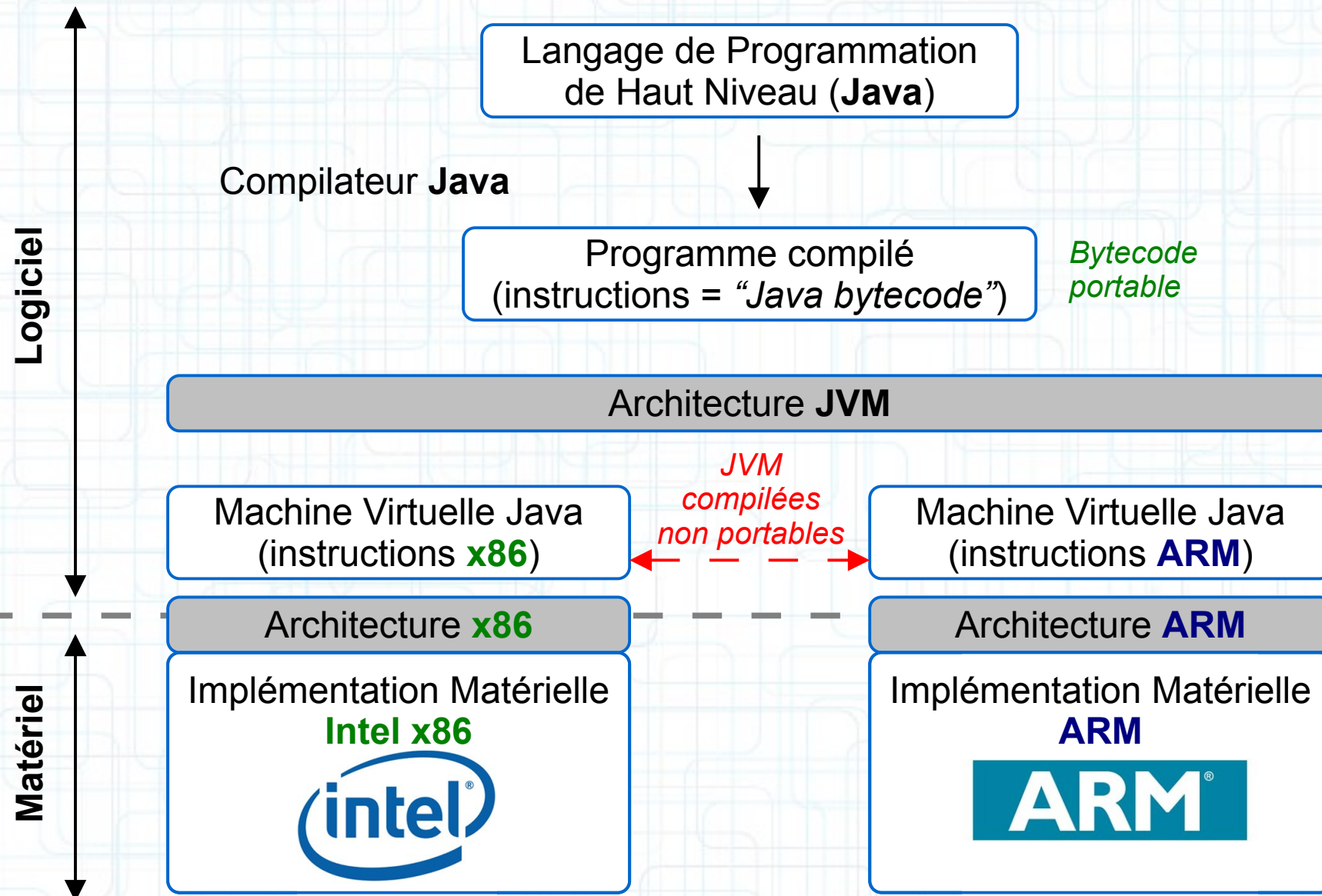


Compilation et exécution

- **Compilation**

- Java est un langage qui est **compilé** pour une architecture virtuelle et **interprété** par une **machine virtuelle**.
 - Le **compilateur** Java produit des instructions destinées à une machine virtuelle (processeur virtuel). Cette suite d'instructions est appelée **bytecode**.
 - La **machine virtuelle** est un programme qui exécute (interprète) sur une architecture particulière le *bytecode* d'une classe.
 - La machine virtuelle java est nommée JVM (*Java Virtual Machine*)
 - Certaines JVM recompilent à la volée le *bytecode* pour l'architecture native afin de gagner en performance (JIT – *Just In Time compiler*)

Compilation et exécution



Compilation et exécution

- **Plateforme Java**

- La **plateforme Java** est l'ensemble JVM + API + compilateur. Plusieurs plateformes existent
 - **Oracle Java SE (Standard Edition)**, Java EE (*Enterprise Edition*) et Java ME (*Mobile Edition*)
 - **Dalvik** VM de Google (utilisé sous Android)
 - **OpenJDK** (supporté par Oracle et IBM)
 - **gcj** (*GNU compiler for java*) de GNU
 - **jikes** d'IBM et **MSJVM** de Microsoft (plus maintenus), ...
- La plateforme comprend deux parties
 - **JRE** (*Java Run-time Environment*) : machine virtuelle, librairie (compilée), ...
 - **JDK** (*Java Development Kit*) : compilateur, sources de la librairie, documentation, divers outils, ...

Ce cours
utilise
OpenJDK

Nécessaire
pour créer
des
programmes
en Java

Compilation et exécution

- **Plateforme Java SE**

- Outils importants

- Compilateur : **javac**
 - Machine virtuelle : **java**
 - Désassembleur : **javap**
 - Générateur de documentation : **javadoc**
 - Le *bytecode* est stocké dans un fichier binaire de même nom que la classe, avec l'extension **.class**

- Exemple d'utilisation :

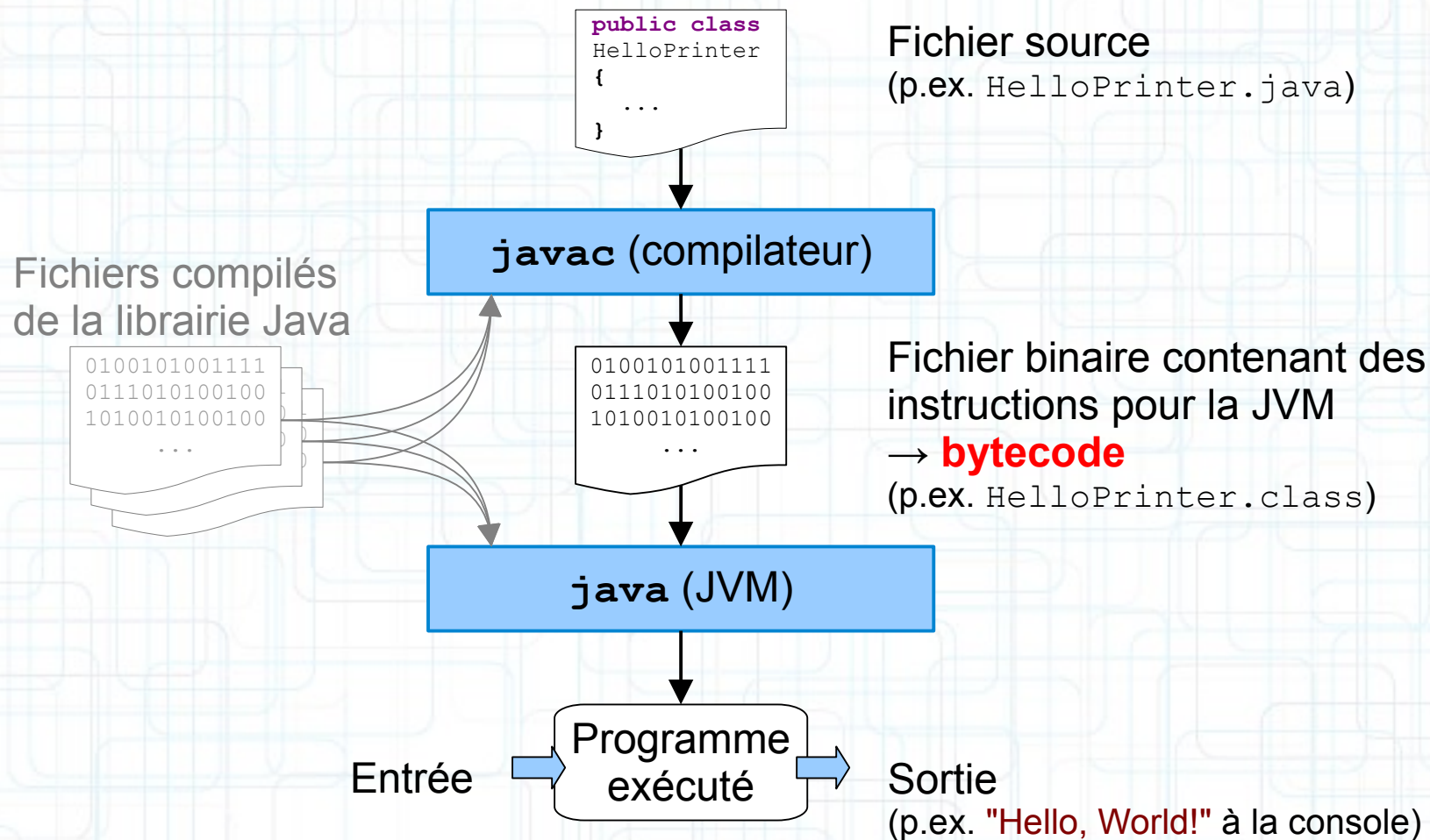
```
bash-3.2$ javac HelloPrinter.java
bash-3.2$ ls
HelloPrinter.java  HelloPrinter.class
bash-3.2$ java HelloPrinter
Hello, World!
bash-3.2$
```

Le compilateur produit un fichier **.class**

La machine virtuelle est invoquée avec le nom de classe (sans l'extension **.class**)

Compilation et exécution

- **Compilation et Exécution**



Compilation et exécution

- **Fichiers .class**

- Les fichiers **.class** contiennent des instructions pour la machine virtuelle Java, encodées sous forme « binaire ».
- Ces fichiers ont une structure qu'il ne nous est pas nécessaire de connaître.

```
bash-3.2$ hexdump -C HelloWorld.class
00000000  ca fe ba be 00 00 00 32 00 1d 0a 00 06 00 0f 09 |.....2.....|
00000010  00 10 00 11 08 00 12 0a 00 13 00 14 07 00 15 07 |.....|
00000020  00 16 01 00 06 3c 69 6e 69 74 3e 01 00 03 28 29 |.....<init>...()|
00000030  56 01 00 04 43 6f 64 65 01 00 0f 4c 69 6e 65 4e |V...Code...LineN|
00000040  75 6d 62 65 72 54 61 62 6c 65 01 00 04 6d 61 69 |umberTable...mai|
00000050  6e 01 00 16 28 5b 4c 6a 61 76 61 2f 6c 61 6e 67 |n...([Ljava/lang|
00000060  2f 53 74 72 69 6e 67 3b 29 56 01 00 0a 53 6f 75 |/String;)V...Sou|
00000070  72 63 65 46 69 6c 65 01 00 11 48 65 6c 6c 6f 50 |rceFile...HelloP|
00000080  72 69 6e 74 65 72 2e 6a 61 76 61 0c 00 07 00 08 |rinter.java.....|
00000090  07 00 17 0c 00 18 00 19 01 00 0d 48 65 6c 6c 6f |.....Hello|
000000a0  2c 20 57 6f 72 6c 64 21 07 00 1a 0c 00 1b 00 1c |, World!.....|
...
```

« magic number »
(identifie un fichier
de type .class)

Numéro de version
(0x32 = J2SE1.6)

Compilation et exécution

- **Fichiers .class**

- L'outil **javap** permet d'obtenir de l'information à propos d'un fichier compilé (**.class**).

```
bash-3.2$ javap HelloPrinter
Compiled from "HelloPrinter.java"
public class HelloPrinter extends java.lang.Object{
    public HelloPrinter();
    public static void main(java.lang.String[]);
}@
```

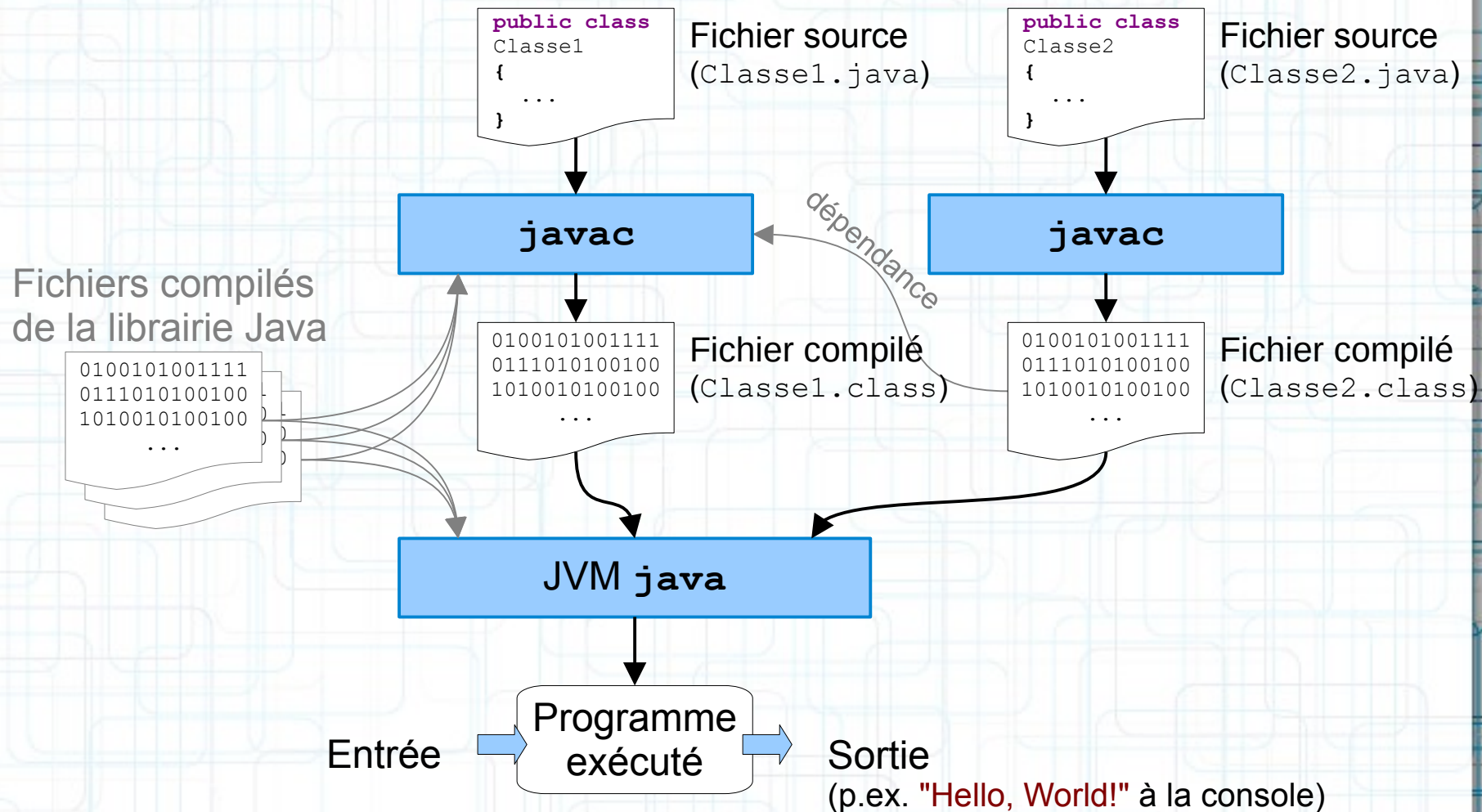
→ Liste des méthodes de la classe ainsi que leur signature.

```
bash-3.2$ javap -c HelloPrinter
...
public static void main(java.lang.String[]);
  Code:
    0:  getstatic      #2; //Field java/lang/System.out:Ljava/io/PrintStream;
    3:  ldc           #3; //String Hello, World!
    5:  invokevirtual #4; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
    8:  return
...
```

→ Instructions JVM implémentant la méthode main

Compilation et exécution

- **Compilation de plusieurs fichiers**

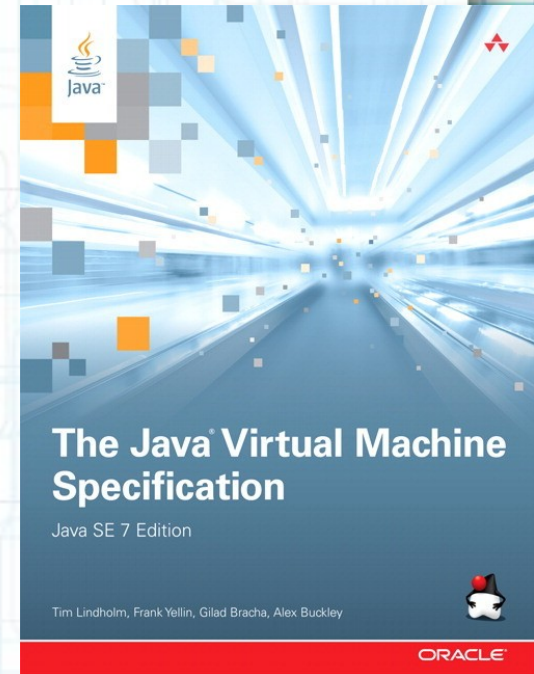


Compilation et exécution

- **Spécification de la JVM**

- Pour les étudiants curieux, il existe un ouvrage qui décrit en détail la machine virtuelle Java

- ***The Java Virtual Machine Specification***,
Java SE 7 Edition, de T. Lindholm et al
February 2013
<http://docs.oracle.com/javase/specs>



Langage Java

- **Erreurs à la compilation / à l'exécution**
 - Certaines erreurs sont détectées par le compilateur, d'autres sont détectées par la machine virtuelle.
 - **Erreurs à la compilation** (*compile-time errors*)
 - détectées par le compilateur (`javac`) et affichées à l'écran
 - erreurs de syntaxe, de type, symbole inconnu, etc.
 - **Erreurs à l'exécution** (*run-time errors*)
 - détectées par la machine virtuelle.
 - erreurs de programmation / de logique non détectables par le compilateur (p.ex. référence nulle)
 - erreurs dues à l'environnement (p. ex. fichier inaccessible)
 - Dans la suite du cours, Il sera important de bien comprendre quelles erreurs sont détectées par le compilateur ou la JVM.

Langage Java

- Erreurs à la compilation / à l'exécution

- Exemple d'erreur détectée à la compilation

```
System.ouch.println("Hello, World!");  
System.out.println("Hello, World!");
```

```
bash-3.2$ javac HelloPrinter.java  
HelloPrinter.java:7: cannot find symbol  
symbol : variable ouch  
location: class java.lang.System  
    System.ouch.println("Hello, World!");
```

Localisation de l'erreur :
Nom du fichier + ligne

Description de l'erreur :
symbole non trouvé



Durant les TPs, si vous rencontrez un problème, **assurez-vous d'avoir lu et compris le(s) message(s) d'erreur** avant d'appeler un assistant à l'aide !!!

Table des Matières

1. Le Langage Java
2. Compilation et exécution
- 3. Variables, types et littéraux**
4. Opérateurs et expressions
5. Chaînes de caractères
6. Méthodes
7. Structures de contrôle
8. Ecrire des commentaires

Variables

- **Typage**

- Java est un langage « **typé** »
 - Un **type** est associé aux variables, valeurs et fonctions. Cela permet de détecter certaines erreurs de programmation.
 - P.ex. compatibilité entre variable et valeur affectée ; compatibilité entre fonction et arguments
- Typage **explicite**
 - Le type de chaque variable doit être **déclaré** avant que la variable puisse être utilisée.
 - Au contraire, Python infère le type des variables durant l'exécution.
- Typage **statique** et **dynamique**
 - Les vérifications de types se font **statiquement** à la compilation (par le compilateur) et **dynamiquement** durant l'exécution (par la JVM).

Variables

- **Déclaration**

- Une **déclaration de variable** comporte

- le **nom de la variable**
- le nom du **type de la variable** (nombre, chaîne de caractères, objet, ...). Le type d'une variable dépend de l'usage qui en sera fait.
- une **affectation** (optionnelle) d'une valeur à la variable.

- Une déclaration de variable suit la syntaxe suivante

nomType *nomVariable* [= *expression*] ;

*Le symbole = et l'expression qui le suit sont optionnels
Ils permettent d'initialiser la variable lors de sa déclaration.
Dans la suite du cours, un élément de syntaxe optionnel sera
présenté entre []*

Variables

- **Déclaration**

- Exemple

```
public class MaClasse {  
  
    public static void main() {  
        int x= 13;  
        int y;  
        String message= "Hello, World!";  
  
        /* ... */  
    }  
}
```

Déclaration de 3 variables dans la méthode `main`.

- La variable `x` de type entier (`int`) est initialisée avec la valeur `13`
- La variable `y` n'est pas initialisée.
- La variable `message`, de type chaîne de caractères (`String`), est initialisée avec la valeur « `Hello, World!` ».

Variables

- **Déclaration**

- La syntaxe de Java permet la déclaration simultanée de plusieurs variables d'un même type.

```
nomType nomVariable1 [= expression1 ], nomVariable2 [= expression2 ], ... ;
```

Chaque variable peut recevoir individuellement une valeur initiale optionnelle.

- Exemples

```
int a, b, c;  
int x= 2, y= 3, z= 4;  
String msg1= "Hello, World!", msg2;
```

Variables

- **Portée (scope)**

- Une variable a une **portée** limitée au bloc dans lequel elle est déclarée. Elle ne peut être utilisée en dehors de ce bloc.
- Une variable déclarée dans une méthode est appelée **variable locale**.

```
public class MaClasse {  
    public static void fonction() {  
        int x= 13;  
        /* ... */  
    }  
  
    public static void main() {  
        System.out.println(x);  
    }  
}
```

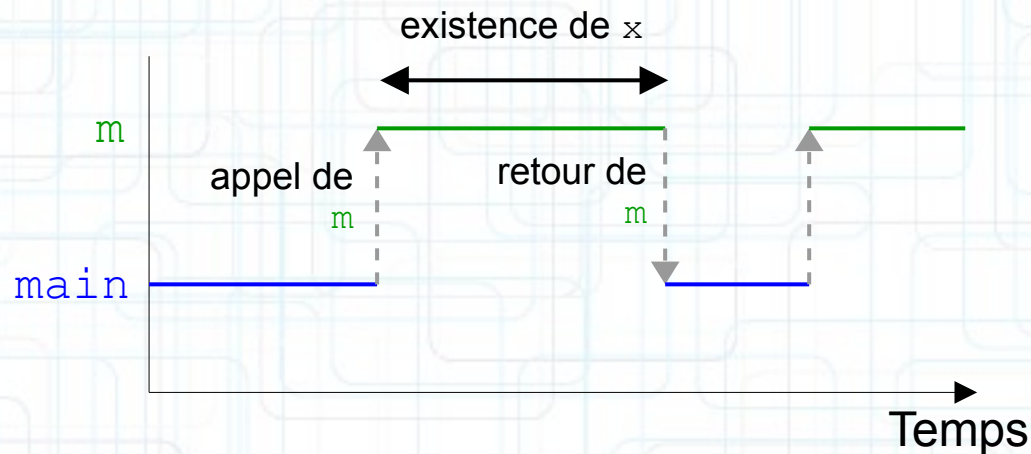
La portée de la variable `x` est limitée au **bloc délimité par les accolades bleues** (ici, le corps de la méthode `fonction`)

La méthode `main` **n'a pas accès** à la variable `x` (hors de portée).

Variables

- **Durée de vie**

- Une variable locale a une **durée de vie** limitée à l'exécution de la méthode. Elle « perd » sa valeur lorsque l'exécution de la méthode se termine.



```
public class MaClasse {  
    public static void m() {  
        int x;  
        /* ... */  
        x= 5;  
    }  
  
    public static void main() {  
        m();  
    }  
}
```

- Note : les *variables de classe* et *d'instance* que nous verrons par la suite ont des portées et durées de vie différentes de celles des variables locales.

Variables

- **Identifiants**

- Les noms de variables sont des identifiants et sont soumis aux règles suivantes.
- Règle de formation d'un **identifiant**
 - le premier caractère est une lettre ("a–zA–Z") ou un *underscore* ("_")
 - les caractères suivants sont des lettres ("a–zA–Z"), chiffres ("0–9") ou *underscore* ("_").
 - un identifiant ne peut être un mot-clé du langage
- Attention ! Les identifiants sont **sensibles à la casse** : il faut respecter les minuscules/majuscules.
 - Par exemple, les identifiants « toto » et « TotO » sont différents.

Variables

- **Mots-clés (réservés)**

- Le langage Java réserve de nombreux mots-clés (*keywords*).
 - `byte`, `short`, `int`, `long`, `float`, `double`, `char`,
`boolean`, `class`, `interface`, `package`, `void`, `public`,
`private`, `protected`, `static`, `final`, `for`, `while`, `if`,
`else`, `new`, `try`, `catch`, `finally`, `throw`, `return`, `null`,
`synchronized`, `super`, `native`, `goto`, `abstract`,
`break`, `case`, `switch`, `const`, `default`, `do`, `extends`,
`implements`, `import`, `instanceof`, `this`, `throw`,
`throws`, ...
- Aucun de ces mots-clés ne peut être utilisé comme identifiant !

Variables

- **Convention de nommage**

- Par convention, les noms de variables suivent les règles suivantes
 - le nom **commence par une lettre minuscule**.
 - le nom suit la **forme « chameau »** lorsqu'il est composé de plusieurs mots : la première lettre de chaque mot concaténé est en majuscule.
- Exemples

```
int index= 13;
```

```
String monBeauMessage= "Hello, World!";
```

```
int nombreEtudiants= 43;
```

Bonne pratique : le nom d'une variable DOIT être suffisamment descriptif pour que le lecteur du programme en comprenne la signification !

Variables

- **Vérification : compatibilité de types**

- Lorsqu'une valeur est assignée à une variable, le compilateur vérifie que **leurs types sont compatibles**.
- Exemple

```
int varEntiere;  
varEntiere= "Hello, World!";
```

```
bash-3.2$ javac BadVarAssign.java  
BadVarAssign.java:7: incompatible types  
found    : java.lang.String  
required: int  
           varEntiere= "Hello, World!";  
                        ^  
1 error  
bash-3.2$
```

Le compilateur détecte qu'un littéral de type « chaîne de caractères » doit être affecté à une variable de type **int**, ce qui n'est pas possible.

Variables

- **Vérification : initialisation des variables**

- Lorsqu'une variable est utilisée, le compilateur vérifie que la variable a été préalablement **initialisée**.

- Exemple

```
int x;  
int y= 5;  
int z= x+y;
```

```
bash-3.2$ javac NotAssignedVar.java  
NotAssignedVar.java:7: variable x might not  
have been initialized  
                int z= x+y;  
                    ^
```

```
1 error  
bash-3.2$
```

Le compilateur détecte que la variable `x` utilisée dans l'expression `x+y` n'est pas initialisée.

Constantes

- **Définition**

- Les constantes sont définies de façon similaire aux variables.
 - Elles ne sont pas locales aux méthodes mais définies au niveau de la classe.
 - Leur déclaration comporte les mots-clés **public**, **static** et **final**.
- Exemple

```
public class Comptabilite
{
    public static final double TAUX_TVA = 21;

    public static void main(String [] args) {
        System.out.print("TVA belge : ");
        System.out.println(TAUX_TVA);
    }
}
```

Bonne pratique :
utiliser des constantes
permet d'associer un
nom (descriptif) à une
valeur.
Dans le cas contraire,
on parle parfois de
"valeurs magiques"...

Types

- **Types de Variables**

- **Types primitifs**

- Permettent de représenter des nombres, des caractères et des booléens.
 - Par exemple: `int`, `char`, ...

- **Types énumérés**

- Données prenant un nombre restreint de valeurs.

- **Types objets**⁽¹⁾

- Permettent de représenter des données ayant une structure et un comportement plus complexes.
 - sera couvert au [Chapitre II](#)

- **Tableaux**

- sera couvert au [Chapitre III](#)

⁽¹⁾ En fait des références à des objets.

Types

- **Types Primitifs**

- Le langage Java supporte 8 types primitifs.

Nom du type	Taille	Domaine	Représentation
byte	8 bits	$[-128, 127]$	complément à 2
short	16 bits	$[-32768, 32767]$	complément à 2
int	32 bits	$[-2^{31}, 2^{31}-1]$	complément à 2
long	64 bits	$[-2^{63}, 2^{63}-1]$	complément à 2
float	32 bits	$\approx [-10^{38}, 10^{38}]$	IEEE-754 simple précision
double	64 bits	$\approx [-10^{308}, 10^{308}]$	IEEE-754 double précision
boolean	non spécifiée	{false, true}	
char	16 bits	$['\u0000', '\uffff']$	Unicode

- Pour les distinguer aisément, les noms de ces types sont en minuscules.

Types

- **Types énumérés**

- Souvent, une variable ne doit pouvoir prendre qu'un **nombre restreint de valeurs**.
- Exemple
 - une année d'étude ne peut prendre que les valeurs BAB1, BAB2, BAB3, MAB1 et MAB2.
 - Idée : représentation sous forme d'entiers (type **byte**) avec 1=BAB1, 2=BAB2, 3=BAB3, 4=MAB1 et 5=MAB2.

```
public class ValeursRestreintes {  
    public static void main(String [] args) {  
        byte annee= 2;          /* correspond à BAB2 */  
        System.out.println(annee);  
        annee= 8;               /* ne correspond à rien */  
    }  
}
```

Erreur, mais le compilateur ne peut la détecter :-)

Types

- **Types énumérés**

- Un **type énuméré** permet de définir un ensemble limité de valeurs et d'associer un nom à chaque valeur.

- Syntaxe

```
enum nomEnum { nom1 , ... , nomN }
```

- Exemple

```
public class TypeEnumere {  
    enum AnneeEtude { BAB1, BAB2, BAB3, MAB1, MAB2 };  
  
    public static void main(String [] args) {  
        AnneeEtude annee= AnneeEtude.BAB2;  
        System.out.println(annee);  
    }  
}
```

Valeur préfixée par
nom du type énuméré.

Contrainte

type énuméré doit être
déclaré globalement
dans une classe et pas
localement à une
méthode.

Vérification par le compilateur :
seuls les valeurs déclarées
peuvent être assignées.

Types

- **Types Objets**

- **String** (`java.lang.String`)
 - permet de représenter des chaînes de caractères
- **Number** (`java.lang.Number`)
 - `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double` permettent de représenter des nombres de la même façon que les types primitifs
 - `BigInteger` et `BigDecimal` permettent de représenter des nombres de n'importe quelle précision.
- Plein d'autres types
 - définis dans la bibliothèque Java ou dans d'autres librairies
 - définis par le programmeur (**vous !**)

Littéraux

- **Littéraux**

- Un **littéral** est l'expression d'une valeur fixe dans le langage de programmation.
- Il existe différents types de littéraux en Java
 - Littéraux numériques (entiers ou flottants)
 - Littéraux booléens (**true** et **false**)
 - Littéraux caractères
 - Littéraux chaînes de caractères

Littéraux

- **Littéraux numériques entiers**

- Par défaut, les littéraux entiers sont de type `int`. Il est possible de créer des littéraux entiers long (de type `long`) en ajoutant le suffixe `L` ou `l`.
- Les littéraux entiers peuvent être exprimés en décimal, mais aussi en hexadécimal, en octal et en binaire⁽¹⁾ en utilisant respectivement les préfixes « `0x` », « `0` » et « `0b` ».
- Exemple
 - les 4 expressions suivantes déclarent la variable `x` et lui affectent la même valeur (123).

```
int x= 123;           // représentation décimale
int x= 0x7b;          // représentation hexadécimale
int x= 0173;          // représentation octale
int x= 0b1111011;     // représentation binaire (java 7)
```

(1) littéraux binaires seulement depuis java 7.

Littéraux

- **Littéraux numériques flottants**

- Par défaut, les littéraux flottants sont de type **double** (double précision). Il est possible d'exprimer des littéraux flottants de type **float** (simple précision) en ajoutant le suffixe **F** ou **f**.
- Les littéraux flottants peuvent être exprimés en utilisant la notation scientifique.
- Exemples

```
double x= 12.345;           // type double
float   y= 12.345F;         // type float
double z= 1.2345e1;         // notation scientifique
                                // z=1,2345x101
```


Littéraux

- **Littéraux caractères**

- Les littéraux caractères sont représentés sous la forme d'un caractère entouré d'apostrophes.

- Exemples

```
char car1= 'a';  
char car2= 't';
```

- Les littéraux caractères peuvent également être exprimés avec des codes de caractère Unicode.

- Exemple

```
char car1= '\u0061';    // code de 'a'  
char car2= '\u0074';    // code de 't'  
char car3= '\u000A';    // code du caractère de  
                        // contrôle  
                        // « passer à la ligne »
```

Littéraux

- **Littéraux chaînes de caractères**

- Les littéraux chaînes de caractères (`String`) sont exprimés sous forme de suites de caractères entourées de guillemets (").

- Exemple

```
String msg= "Hello, World!";
```

- Des caractères spéciaux peuvent être exprimés à l'aide de **séquences d'échappement**. Par exemple, pour insérer un guillemet dans un littéral chaîne de caractères, il faut le précéder d'un « *backslash* » (\).

- Exemple

```
String msg= "Le message est \"Hello, World!\"";
```


Littéraux

- **Littéraux chaînes de caractères**

- Le tableau ci-dessous présente quelques séquences d'échappement utiles.

Séquence	Signification
\ "	Le caractère guillemet (<i>double quote</i>)
\ '	Le caractère apostrophe (<i>single quote</i>)
\\	Le caractère « backslash »
\n	Le caractère de contrôle « passer à la ligne »
\r	Le caractère de contrôle « retourner en début de ligne »
\t	Le caractère de contrôle « tabulation »

- Note : ces séquences d'échappement sont aussi utilisables dans les littéraux caractères.

Table des Matières

1. Le Langage Java
2. Compilation et exécution
3. Variables, types et littéraux
4. **Opérateurs et expressions**
5. Chaînes de caractères
6. Méthodes
7. Structures de contrôle
8. Ecrire des commentaires

Expression

- **Expression**

- Une **expression** est une combinaison de littéraux, de variables et d'opérateurs visant à produire une nouvelle valeur.
- Une expression a un type qui dépend des types des littéraux, variables et opérateurs qui la composent.
- Exemple

```
// expression composée d'un littéral
int x= 5;
/* expression composée d'un littéral, d'une
   variable et d'un opérateur */
int y= x+6;
/* expression dont le type diffère des
   variables qui la composent */
boolean test= y > x;
```


Opérateurs

- **Opérateurs arithmétiques binaires**

- Les opérateurs arithmétiques binaires suivants sont supportés. Ces opérateurs ne peuvent être appliqués qu'à des termes numériques.

Opérateur	Opération
$a + b$	Additionne a et b
$a - b$	Soustrait b de a
$a * b$	Multiplie a par b
a / b	Divise a par b
$a \% b$	Reste de la division entière de a par b

- Exemple

```
int x= 5, y= 6, z= x+y;           // z=11
int a= 17, b= 4, c= a/b, d= a%b;  // c=4 et d=1
float f= 12.23, g= 12, h= f%g;    // h=0.23
```


Opérateurs

- **Opérateurs arithmétiques unaires**

- Les opérateurs arithmétiques unaires suivants sont supportés. Ces opérateurs ne peuvent être appliqués qu'à des termes numériques.

Opérateur	Opération
$a++$	Post-incrémentation de a ⁽¹⁾
$++a$	Pré-incrémentation de a
$a--$	Post-décrément de a
$--a$	Pré-décrément de a
$-a$	Négation de a

Peuvent être utilisés en dehors d'expressions.
Forme d'affectation : a est modifié.

- Exemple

```
int a= 5, b= 6;  
int c= a--;  
int d= ++b;
```

```
// a=4 et c=5  
// b=7 et d=7
```

(1) les opérateurs de pré-/post-incrémentation et décrémentation n'existent pas en Python.

Opérateurs

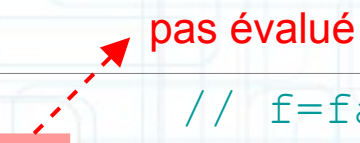
- **Opérateurs logiques / conditionnels**

- Les opérateurs conditionnels ne peuvent être appliqués qu'à des termes booléens (**true** ou **false**).

Opérateur	Opération	
<code>a && b</code>	ET logique entre <code>a</code> et <code>b</code> ⁽¹⁾	
<code>a b</code>	OU logique entre <code>a</code> et <code>b</code> ⁽¹⁾	
<code>a ? b : c</code>	La valeur de l'expression est <code>b</code> si <code>a</code> est vrai, <code>c</code> sinon	opérateur "ternaire"
<code>!a</code>	Complément de <code>a</code>	

- Exemple

```
boolean t= true, f= !t;           // f=false
boolean lazy= true || (5 > 10);   // true
String unit= (price > 1) ? "euros" : "euro";
```

 pas évalué

(1) Note : l'opérande de droite des opérateurs `&&` et `||` n'est pas évalué si celui de gauche est respectivement **false** ou **true**.

Opérateurs

- **Opérateurs de comparaison**

- Le langage Java supporte les opérateurs de comparaison suivants.

Opérateur	Opération
$a == b$	Teste l'égalité de a et b
$a != b$	Teste l'inégalité de a et b
$a < b$	Teste si a est strictement inférieur à b
$a <= b$	Teste si a est inférieur à b
$a > b$	Teste si a est strictement supérieur à b
$a >= b$	Teste si a est supérieur à b

```
int x= 1000;  
float y= 1e3f;  
System.out.println(x == y); // true  
String z= "1000";  
System.out.println(x == z);
```

Attention, certains types ne peuvent être comparés entre eux.

Opérateurs

- **Opérateurs bit-à-bit (*bitwise*)**

- Les opérateurs bit-à-bit permettent de manipuler la représentation binaire des variables de types primitifs numériques et caractères.

Opérateur	Opération
$a \& b$	Effectue un ET bit-à-bit de a et b
$a b$	Effectue un OU bit-à-bit de a et b
$a \wedge b$	Effectue un OU-exclusif bit-à-bit de a et b
$a \ll b$	Décale les bits de a de b positions vers la gauche
$a \gg b$	Décale les bits de a de b positions vers la droite
$a \ggg b$	Décale les bits de a de b positions vers la droite (version non signée)
$\sim a$	Complémente les bits de a

Note : Ces opérateurs seront discutés au cours de Fonctionnement des Ordinateurs.

Opérateurs

- Opérateurs bit-à-bit (*bitwise*)

- Exemple

```
byte a= -12;  
byte b= ~a;  
byte c= a >> 2;  
byte d= a << 2;  
byte e= a & (1 << 4);
```

- Que valent les variables *b*, *c*, *d* et *e* ?

11110100

(a=-12)

11110100

(a=-12)

11110100

(a=-12)

11110100

(a=-12)

00001011

(b=11)

11111101

(c=-3)

11010000

(d=-48)

00010000

(1 << 4)

00010000

(e=16)

Note : Ces opérateurs seront discutés au cours de Fonctionnement des Ordinateurs.

Expression

- **Ordre des opérations**

- Une expression arithmétique ou logique peut contenir plusieurs opérateurs.
- Par exemple, l'expression $a+b/c$ correspond-t-elle à $(a+b)/c$ ou à $a+(b/c)$?
- Il est important de comprendre comment une expression est évaluée !

Expression

- Règle d'évaluation

- Précédence des opérateurs : les opérateurs de **plus haute priorité** sont considérés d'abord.

$$a + \underbrace{b * c}$$

* a une priorité plus élevée que +

- Règle d'associativité : à priorité égale, l'associativité de l'opérateur est utilisée.

$$\underbrace{a + b} + c$$

+ associatif à gauche

$$a = \underbrace{b = c}$$

= associatif à droite

- Ordre d'évaluation des sous-expressions

- Indépendant de priorité et associativité.
- S'il n'y a pas de dépendances, une expression est évaluée **de gauche à droite** indépendamment des règles de priorité et d'associativité.

$$a() + b() * c() \\ \text{ordre : } a, b, c$$

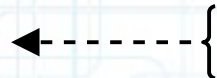
$$a() + b(c(), d()) \\ \text{ordre : } a, c, d, b$$

résultats de c() et d() nécessaires pour b()

Opérateurs

Précédence et Associativité

Les opérateurs d'une même ligne ont la même priorité.



Opérateurs	Associativité
++ -- (<i>post</i>)	à gauche
++ -- (<i>pré</i>)	à droite
+ - ~ ! (<i>unaires</i>)	à droite
* / %	à gauche
+ -	à gauche
<< >> >>>	à gauche
< > <= >=	à gauche
== !=	à gauche
&	à gauche
^	à gauche
	à gauche
&&	à gauche
	à gauche
? :	à droite
=	à droite

Précédence croissante

Opérateurs

- **Précédence des Opérateurs**

- Exemple

```
double a= 5, b= 12, c= 19;  
double y= a + b / c;           // y=5.631...
```

L'opérateur de division a une précedence supérieure à celle de l'opérateur d'addition. L'expression est donc équivalente à $a+(b/c)$.

- Exemple

```
int a= 123, b= 456, c= 78;  
int y= a + b - c;           // y=501
```

Les opérateurs d'addition et de soustraction ont la même précedence, c'est la règle du *gauche-à-droite* qui s'applique.

Opérateurs

- **Précédence des Opérateurs**

- Exemple

```
int a= 10;  
a= ++a * ( ++a + 5 );
```

- Quelle est la valeur finale de **a** ?
 - Réponse : 187

Bonne pratique :

- évitez les expressions qui ne peuvent être comprises facilement
- évitez l'usage de multiples opérateurs de pré-/post incrémentation/décrémentation

Note : cet exemple est inspiré d'une question posée sur le forum *stackoverflow* (29 janvier 2015).

Table des Matières

1. Le Langage Java
2. Compilation et exécution
3. Variables, types et littéraux
4. Opérateurs et expressions
- 5. Chaînes de caractères**
6. Méthodes
7. Structures de contrôle
8. Ecrire des commentaires

Chaînes de Caractères

- **Opérateur de Concaténation**

- L'**opérateur de concaténation**, noté +, crée une nouvelle String dont le contenu est la concaténation de deux autres String.

- Exemple

```
String msg1= "Hello, ";  
String msg2= "World!";  
String msg3= msg1 + msg2;
```

- L'opérateur de concaténation s'applique également au cas où seul la première opérande est une String. Dans ce cas, la seconde opérande est **automatiquement convertie** en String.

- Exemple

```
int x= 17;  
System.out.println("Le résultat est " + x);  
                                            
                        "Le résultat est 17"
```


Chaînes de Caractères

• Opérateur de Concaténation

- Les règles de précedence et d'associativité des opérateurs continuent à s'appliquer.



Exemple

```
int x= 17, y= 23;  
System.out.println("x + y = "+x+y); // Incorrect
```

"x + y = 17"
"x + y = 1723"

Les opérateurs de concaténation et d'addition (+) ont la même précédence. L'opérateur + est associatif à gauche.

```
System.out.println("x * y = "+x*y); // Correct
```

"x * y = 391"

Ici, l'opérateur de multiplication (*) a une précédence plus grande que l'opérateur de concaténation (+).

Chaînes de Caractères

- **Méthodes spécifiques aux String**
 - Les chaînes de caractères étant des objets, elles offrent des méthodes (fonctions) qui leurs sont propres.
 - La méthode **length()** permet de déterminer la longueur d'une chaîne de caractères.
 - La méthode **substring(int b, int e)** permet d'extraire la sous-chaîne commençant à la position **b** et se terminant à la position **e-1**.
 - La méthode **charAt(int i)** permet d'extraire le caractère à la position **i**.

```
String s= "Le monde est plat";  
int longueurChaine= s.length();  
System.out.println("Hello".length());           // 5  
System.out.println(s.substring(3, 8));           // monde  
System.out.println(s.charAt(s.length()-1));      // t
```


Chaînes de Caractères

• Opérateurs de Comparaison



- La comparaison de `String`, **ne doit pas être effectuée** avec les opérateurs de comparaison vu précédemment

== != > >= < <=

- Il est nécessaire d'utiliser des méthodes spécialement prévues à cet effet par le type `String` :
 - Test d'égalité : la méthode **`equals`** permet de tester si une `String` est égale à une autre.
 - Comparaison : la méthode **`compareTo`** permet de comparer lexicographiquement une `String` à une autre.

Chaînes de Caractères

- Opérateurs de Comparaison



- Exemple incorrect

```
String s1= "Hello";  
String s2= "Hel";  
String s3= "lo";  
System.out.println(s1 == (s2+s3));
```

false

Bien que le compilateur accepte l'utilisation de l'opérateur `==`, le test d'égalité ne porte pas sur le contenu des chaînes !!!

Note : les optimisations effectuées par java pourraient faire réussir ce test (il est donc non déterministe).

- Exemple correct

```
String s1= "Hello";  
String s2= "Hel";  
String s3= "lo";  
System.out.println(s1.equals(s2+s3));
```

true

L'utilisation de la méthode spécifique `equals` teste le contenu des chaînes.

Chaînes de Caractères

- **Opérateurs de Comparaison**

- La méthode `compareTo` compare une chaîne avec une autre chaîne, selon l'ordre lexicographique.
- La méthode retourne
 - < 0 si l'autre chaîne est « plus grande »
 - 0 si les chaînes sont égales
 - > 0 si l'autre chaîne est « plus petite »

```
String s1= "contrainte";  
String s2= "covid";  
String s3= "albert";  
System.out.println(s1.compareTo(s2));  
System.out.println(s1.compareTo(s3));  
System.out.println("contrainte".compareTo(s1));
```

```
-8  
2  
0
```


Table des Matières

1. Le Langage Java
2. Compilation et exécution
3. Variables, types et littéraux
4. Opérateurs et expressions
5. Chaînes de caractères
- 6. Méthodes**
7. Structures de contrôle
8. Ecrire des commentaires

Méthodes

- **Définition d'une Méthode**

- Une **méthode** est un sous-programme⁽¹⁾, i.e. une suite d'instructions, qui peut être appelée depuis un autre endroit dans le programme.
- La définition d'une méthode comprend 4 parties.
 - un **nom** (identifiant)
 - optionnellement des **arguments et leurs types**
 - un **type de valeur de retour**
(le mot-clé **void** indique qu'aucune valeur n'est retournée)
 - les **instructions** de la méthode
- La **signature** d'une méthode est l'ensemble des types de ses arguments.

⁽¹⁾ fonction en Python

Méthodes

- Définition d'une Méthode

- Syntaxe

```
public static typeRetour nomMethode ( arguments )  
{  
    instructions  
}
```

- Les noms et types des arguments sont définis selon la syntaxe suivante

```
typeArg1 nomArg1 , typeArg2 nomArg2 , ...
```

- Exemple

```
// retourne la somme des arguments 'x' et 'y'  
public static int somme(int x, int y)  
{  
    return x + y;  
}
```

Note : cette méthode doit être définie à l'intérieur d'une classe.

Méthodes

- **Convention de Nommage**

- La convention de nommage des méthodes est similaire à celle des variables.
 - commencer par une **minuscule**
 - notation **chameau**
- Les noms de méthodes sont des **actions**.
 - Exemples : `computeArea`, `getNumElements`, `runTask`, `chargerDonnees`
- Cas particulier, méthodes retournant un **boolean**
 - Exemples : `isReady`, `hasChild`, `estTermine`, `contientElement`

Bonne pratique : le nom d'une méthode DOIT être suffisamment descriptif pour que le lecteur du programme en comprenne la signification !

Méthodes

- **Mot-clé `return`**

- Le mot-clé `return` est utilisé dans une méthode
 - pour retourner la résultat de la méthode. Celle-ci est fournie sous la forme d'une expression.
 - pour terminer la méthode.
- Règles d'utilisation
 - une méthode qui retourne une valeur doit toujours se terminer par un `return`.
 - une méthode qui ne retourne pas de valeur (type `void`) peut être terminée par un `return` (sans valeur)
 - une méthode peut contenir plusieurs `return`.

Méthodes

- Appel de Méthode

- L'**appel de méthode** s'effectue en donnant le nom de la méthode et en lui fournissant les valeurs des arguments, selon la syntaxe suivante.

```
nomMethode ( valArg1 , valArg2 , ... )
```

- Les valeurs des arguments (*valArg_i*) sont des expressions. Leurs types doivent correspondre à ceux de la définition de la méthode.
- Si la méthode retourne une valeur, celle-ci devrait être utilisée dans une expression (par exemple une affectation). Il est néanmoins permis d'ignorer le résultat.

Méthodes

- Appel de Méthode

- Exemple

```
public class MaClasse
{
    public static int somme(int x, int y) {
        return x + y;
    }

    public static void main(String [] args) {
        int laSomme = somme(5, 7);
        somme(8, 9);
    }
}
```

Le résultat de l'appel est utilisé dans une expression.

Il est permis d'ignorer le résultat d'une méthode. Attention, ça n'est pas détecté si fait involontairement.

Méthodes

- **Méthode sans résultat**

- Une méthode sans résultat est déclarée en utilisant **void** comme type de retour. L'appel d'une telle fonction ne peut pas être utilisé dans une expression

```
public class MaClasse
{
    /* La méthode 'printSmurf' affiche un
       message à la console */
    public static void printSmurf() {
        System.out.println("Smurf");
        return;
    }

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

Note : dans une méthode sans résultat, le rôle de **return** est de terminer la méthode. Dans cet exemple, **return** devrait être omis car il n'apporte rien.

L'appel n'est pas utilisé dans une expression (pas de valeur).

Méthodes

- **Longueur des méthodes**

- Il est difficile de rester concentré lors de la lecture d'une méthode qui s'étale sur plusieurs pages

Bonne pratique :

- les méthodes devraient être gardées **courtes** (typiquement $\leq \sim 1$ page ou ~ 25 lignes)

- Afin de limiter la longueur d'une méthode

Bonne pratique :

- éviter le code redondant

- éventuellement, factoriser le code redondant (p.ex. calculer plusieurs fois la même chose) vers des méthodes "de support" (*helper methods*).

Bonne pratique :

- une méthode devrait ne remplir qu'**une seule tâche**

Méthodes

- Mots-clés **public** et **static**
 - A ce stade, les méthodes que nous avons définies sont des méthodes dites « *de classe* » ou méthodes « *statiques* ».
 - La déclaration de méthodes de classe contient le
 - *modificateur* **static**.
 - le *spécificateur d'accès* **public**. Ce dernier permet l'utilisation de la méthode à l'extérieur de la classe dans laquelle elle est définie.
 - Nous regarderons la définition générale des méthodes au **Chapitre II**.

Méthodes

- **Surcharge de méthodes**

- Au sein d'une même classe, toutes les méthodes devraient avoir un nom différent.
- Il est possible de déclarer plusieurs méthodes de même nom dans une classe si elles ont des signatures différentes. On parle alors de **surcharge de méthode** (« *overloading* »).
- Condition : deux signatures de méthodes sont différentes si au moins l'une des conditions suivantes est vérifiée
 - les signatures ont un **nombre d'arguments différents**
 - les signatures ont des **arguments de types différents**.

Méthodes

- **Surcharge de méthodes**

- Il est possible de surcharger la méthode `add(int, int)` avec `add(double, double)` car elles ont des signatures différentes.

```
public static int add(int x, int y) {  
    return x + y;  
}  
  
public static double add(double x, double y) {  
    return x + y;  
}
```

- En revanche, il n'est pas possible d'ajouter la méthode suivante car sa signature est identique à la première.

```
public static long add(int x, int y) {  
    return x + y;  
}
```


Méthodes

- **Surcharge de méthodes**

- La surcharge de méthodes est utilisée p.ex. pour les méthodes **print** / **println** fournies par la classe **PrintStream** de la bibliothèque Java (souvenez-vous de `System.out`).

```
public class PrintStream
{
    /* ... */
    public void println() ;
    public void println(boolean x) ;
    public void println(char x) ;
    public void println(char [] x) ;
    public void println(double x) ;
    public void println(float x) ;
    public void println(int x) ;
    public void println(long x) ;
    public void println(Object x) ;
    public void println(String x) ;
    /* ... */
}
```


Table des Matières

1. Le Langage Java
2. Compilation et exécution
3. Variables, types et littéraux
4. Opérateurs et expressions
5. Chaînes de caractères
6. Méthodes
- 7. Structures de contrôle**
8. Ecrire des commentaires

Structures de contrôle

- **Introduction**

- Le langage Java supporte des structures de contrôle classiques telles que
- branchements conditionnels
 - **if / if ... else**
 - **switch**
- boucles
 - **for**
 - **while**
 - **do ... while**

Table des Matières

1. Le Langage Java
2. Compilation et exécution
3. Variables, types et littéraux
4. Opérateurs et expressions
5. Chaînes de caractères
6. Méthodes
- 7. Structures de contrôle**
 - 1. Branchements conditionnels**
 - 2. Boucles**

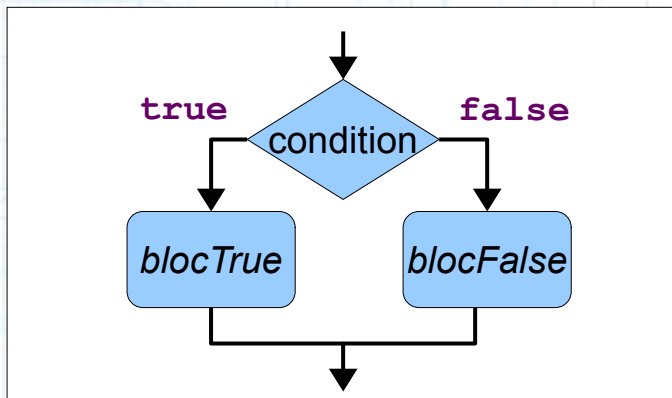
Branchements conditionnels

- Structure **if** / **if-else**

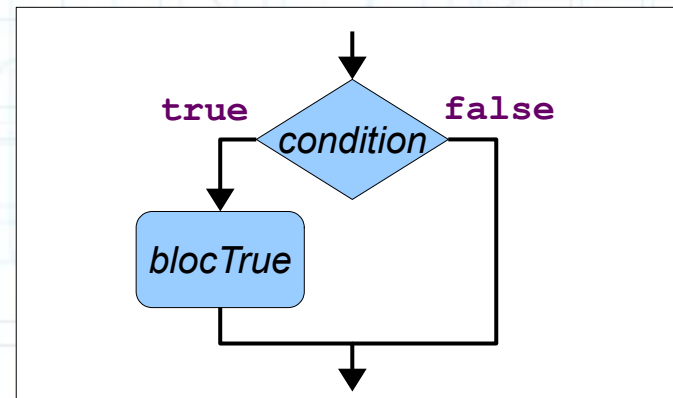
- La structure de contrôle **if** ou **if-else** permet de tester une *condition* exprimée sous forme d'une expression de type booléen. Selon la valeur de l'expression (**true** ou **false**), des blocs d'instructions différents sont exécutés.

- Syntaxe

```
if ( condition )  
    blocTrue  
else  
    blocFalse
```



```
if ( condition )  
    blocTrue
```



Branchements conditionnels

- **Bloc d'instructions**

- Un **blocs d'instructions** est une séquence d'instructions, délimitées par une *accolade ouvrante* ({) et une *accolade fermante* (}). Les accolades peuvent être omises dans le cas où il n'y a qu'une seule instruction.
- Il est possible de déclarer des *variables locales* à un bloc. Leur portée est limitée au bloc.
- Exemple : bloc de plusieurs instructions

```
{  
    int resultat = a*x*x + b*x + c;  
    System.out.println("Le résultat est "+  
                        resultat);  
}
```

Variable locale.

- Exemple : bloc d'une instruction unique

```
System.out.println("You're not alone");
```

Les accolades peuvent être omises lorsqu'il n'y a qu'une instruction dans le bloc.

Branchements conditionnels

- **Structure `if` / `if-else`**

- Exemple

```
public class TestPaysage {  
  
    public static final double RESULTAT_MIN      = 10;  
    public static final double RESULTAT_DISPENSE = 12;  
  
    public static void main(String [] args) {  
  
        double resultatEtudiant= 13;  
        if (resultatEtudiant >= RESULTAT_MIN)  
        {  
            System.out.println("Bravo, c'est réussi");  
            if (resultatEtudiant >= RESULTAT_DISPENSE)  
                System.out.println("Dispense accordée");  
        }  
        else  
            System.out.println("Oups. Courage...");  
    }  
}
```


Branchements conditionnels

- **Ambiguïté possible**



- Lorsque plusieurs structures **if-else** sont imbriquées, il peut exister une ambiguïté: il peut être difficile de déterminer à quel **if** un **else** se rapporte.

- Exemple

```
if ( condition1 )  
    if ( condition2 )  
        bloc1  
else  
    bloc2
```

Le second **else** est-il lié au test de la première ou de la seconde condition ?

L'indentation semble indiquer qu'il est lié à la première condition. Cependant, le compilateur ne se préoccupe pas de l'indentation (contrairement à Python).

- Règle : le **else** est lié au **if** le plus proche.

Branchements conditionnels

- **Ambiguïté possible**

- Pour lever l'ambiguïté de l'exemple précédent, il est nécessaire de délimiter plus précisément les blocs d'instructions.

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

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



```
if ( condition1 )  
    if ( condition2 )  
        bloc1  
else  
    bloc2
```

Bonne pratique : parfois les accolades ne sont pas strictement utiles, mais rendent le code plus lisibles.

Utilisez-les dans ces cas !

Branchements conditionnels

- Lisibilité

Bonne pratique :

1) utilisez des conditions claires.

2) employez "**else-if**" pour des décisions multiples

```
if ( cond1 ) {  
    if ( cond2 ) {  
        if ( cond3 ) {  
            doSomething();  
        } else  
            error(3);  
    } else  
        error(2);  
} else {  
    error(1);  
}
```

limiter profondeur
d'imbrication
(si possible)

```
if ( !cond1 ) {  
    error(1);  
} else if ( !cond2 ) {  
    error(2);  
} else if ( !cond3 ) {  
    error(3);  
} else {  
    doSomething();  
}
```

D'après un exemple tiré de "*The Practice of Programming*" de Brian W. Kernighan et Rob Pike, Addison-Presley, 1999.

Branchements conditionnels

- **Structure switch**

- La structure de contrôle **switch** est utilisée lorsqu'un choix parmi plusieurs possibilités doit être effectué. Il s'agit d'une alternative à une suite de multiples tests **if/else**.
- Le bloc d'instructions à exécuter dépend d'une valeur variable. Celle-ci est comparée à plusieurs valeurs constantes, chacune associée à un bloc d'instructions différent.
- Limitation :
 - La structure de contrôle **switch** ne fonctionne qu'avec les types primitifs **byte**, **short**, **char**, **int**, avec les chaînes de caractères (**String**)⁽¹⁾ et avec les types énumérés (**enum**).
- Note : il n'existe pas d'équivalent à **switch** en **Python**.

(1) depuis Java7.0

Branchements conditionnels

- **Structure switch**

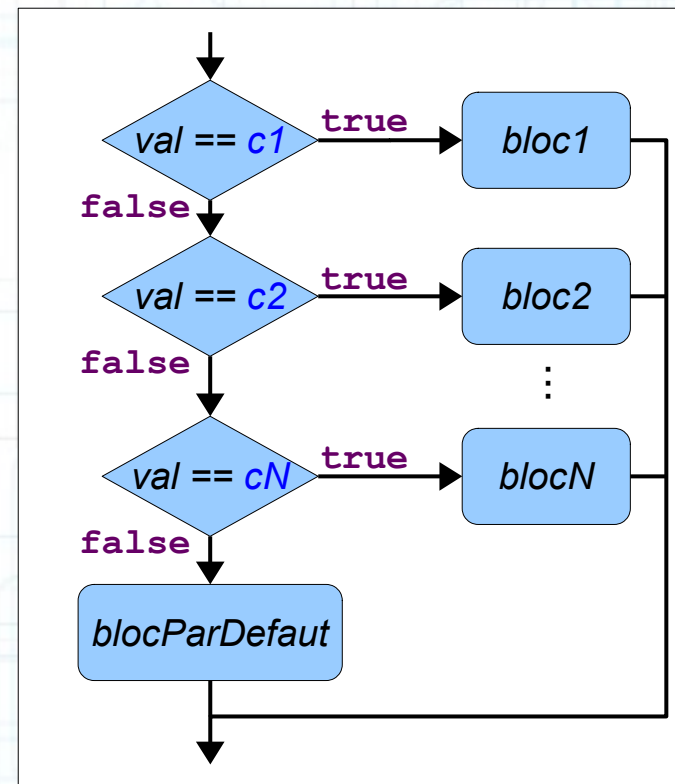
- Syntaxe

```
switch ( val ) {  
  case c1 :  
    bloc1  
    break;  
  case c2 :  
    bloc2  
    break;  
  ...  
  case cN :  
    blocN  
    break;  
  [ default :  
    blocParDefault ]  
}
```

valeurs
constantes
différentes

Optionnel :
cas « par
défaut »
(ne peut
apparaître
qu'une fois)

les crochets indiquent
un bloc optionnel.
Ils ne font pas partie de
la syntaxe.



Branchements conditionnels

- **Contraintes**

- Chaque cas est associé à une valeur constante.
 - Les valeurs constantes sont spécifiées en utilisant des constantes entières, caractères ou chaînes de caractères (Java 1.7).
 - Les valeurs sont spécifiées en utilisant des littéraux ou mieux en utilisant des constantes (**static final**).
- Au sein d'une même structure **switch**
 - Les valeurs correspondant aux différents cas doivent être de même type.
 - Il ne peut y avoir deux cas associés à la même valeur.
 - Il ne peut y avoir qu'un cas par défaut.

Branchements conditionnels

- **Structure switch – Exemple**

- Soit un programme de lecture de fichiers audio. Le programme est muni d'une interface utilisateur en console.
- Le programme affiche le menu suivant :

```
Actions possibles:  
(1) Lecture  
(2) Pause  
(3) Stop  
(4) Plage précédente  
(5) Plage suivante  
(6) Quitter
```

```
Quelle est votre sélection ?
```

- Suivant le code d'action (entier) sélectionné par l'utilisateur, des actions différentes doivent être effectuées par le programme.

Branchements conditionnels

- **Structure switch – Application**
 - Une implémentation possible repose sur l'utilisation de multiples structures **if/else**.

```
Scanner input= new Scanner(System.in);
while (true) {
    int action= input.nextInt();
    if (action == 1)
        doPlay();
    else if (action == 2)
        doPause();
    else if (action == 3)
        doStop();
    else if (action == 4)
        doPrevTrack();
    else if (action == 5)
        doNextTrack();
    else if (action == 6)
        System.exit(0);
    else
        System.err.println("Action non supportée");
}
```

Branchements conditionnels

- **Structure switch – Application**

- La structure **switch** permet de montrer explicitement qu'il s'agit d'une sélection parmi de multiples alternatives.

```
Scanner input= new Scanner(System.in);  
while (true) {  
    int action= input.nextInt();  
    switch (action) {  
        case 1: doPlay(); break;  
        case 2: doPause(); break;  
        case 3: doStop(); break;  
        case 4: doPrevTrack(); break;  
        case 5: doNextTrack(); break;  
        case 6: System.exit(0);  
        default:  
            System.err.println("Action non supportée");  
    }  
}
```

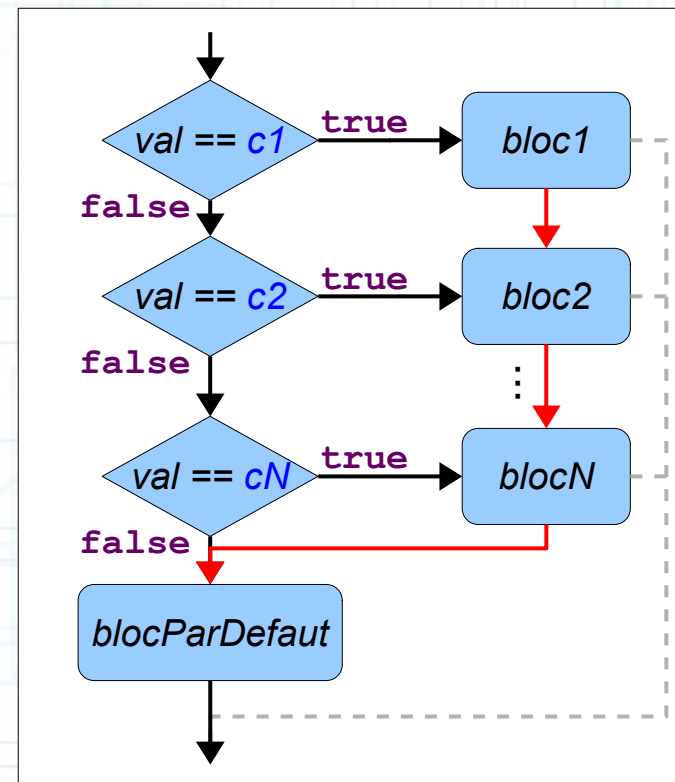

Branchements conditionnels

- Omission du mot-clé **break**

- Le mot-clé **break** peut être omis pour un ou plusieurs cas. La conséquence est que l'exécution se poursuit avec le ou les blocs suivants jusqu'à un **break** ou la fin du **switch**.

```
switch ( val ) {  
  case c1 :  
    bloc1  
    break;  
  case c2 :  
    bloc2  
    break;  
  ...  
  case cN :  
    blocN  
    break;  
  [ default:  
    blocParDefault ]  
}
```

il est autorisé d'omettre **break** pour un ou plusieurs cas.



Branchements conditionnels

- Omission du mot-clé **break**

- Omettre le mot-clé **break** permet par exemple d'exécuter une action commune à un ensemble de valeurs constantes différentes.

```
...  
switch (action) {  
    case 1:  
    case 2:  
    case 5:  
        faitCeci();  
        break;  
    case 6:  
    case 19:  
        faitCela();  
        break;  
    default:  
        System.err.println("Action non supportée");  
}
```

Explication :

- si action vaut 1, 2 ou 5, le **bloc bleu** est exécuté.
- si action vaut 6 ou 19, le **bloc vert** est exécuté.

Branchements conditionnels

- **Types énumérés et switch**

- Les types énumérés sont souvent utilisés en combinaison avec la structure de contrôle **switch**.
- Exemple : cas du menu en ligne de commande vu précédemment.

```
enum PlayerAction { PLAY, PAUSE, STOP, NEXT, PREVIOUS, QUIT };

printMenu();
PlayerAction action= getUserAction();
switch (action) {
    case PLAY      : doPlay(); break;
    case PAUSE     : doPause(); break;
    case STOP      : doStop(); break;
    case PREVIOUS  : doPrevTrack(); break;
    case NEXT      : doNextTrack(); break;
    case QUIT      : System.exit(0);
}
```

Dans une structure **switch**, les valeurs du type énuméré ne doivent pas être préfixées par le nom du type énuméré.

Le compilateur sait qu'il s'agit du type utilisé dans l'expression testée par **switch** (ici variable `action`).

Table des Matières

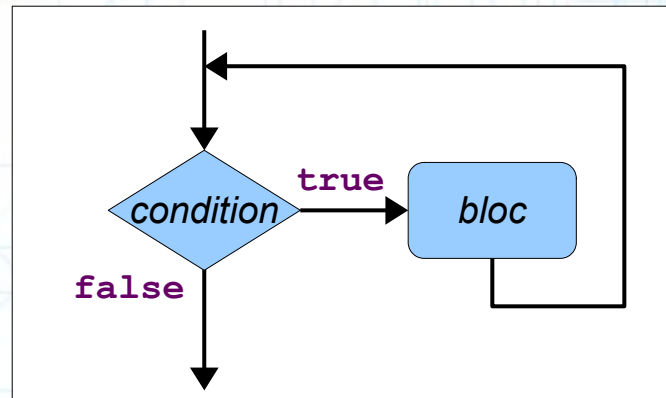
1. Le Langage Java
2. Compilation et exécution
3. Variables, types et littéraux
4. Opérateurs et expressions
5. Chaînes de caractères
6. Méthodes
- 7. Structures de contrôle**
 1. Branchements conditionnels
 - 2. Boucles**

Boucles

- **Boucle while**

- La boucle **while** permet d'exécuter un bloc d'instructions tant qu'une condition est vraie.
- Syntaxe

```
while ( condition )  
    bloc
```



Les instructions de *bloc* sont exécutées tant que la valeur de l'expression *condition* est évaluée à **true**.

Boucles

- **Boucle while**

- Exemple

```
public class TestBoucle {  
    public static void main(String [] args) {  
        byte octet= -13;  
        int index= 7;  
        while (index >= 0)  
        {  
            if ((octet & (1 << index)) != 0)  
                System.out.print("1");  
            else  
                System.out.print("0");  
            index--;  
        }  
        System.out.println();  
    }  
}
```

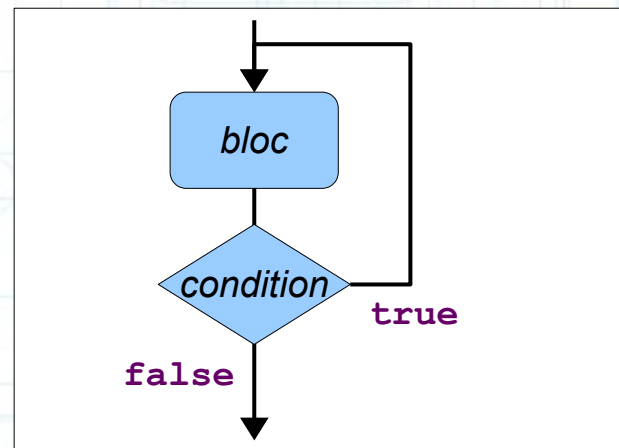
- Question subsidiaire : que fait ce programme ?
(exécutez-le pour vous assurer de votre compréhension)

Boucles

- **Boucle do-while**

- La boucle **do-while** est une variante de la boucle **while** qui teste la condition en fin de boucle. La boucle est donc toujours exécutée au moins une fois.
- Syntaxe

```
do  
    bloc  
while ( condition )
```



Boucles

- Boucle do-while

```
import java.util.Scanner;

public class BoucleDoWhile {

    public static void printMenu() {
        System.out.println("MENU: choix de l'opération");
        System.out.println(" (1) Addition");
        System.out.println(" (2) Multiplication");
        System.out.println(" (0) Quitter");
    }

    public static void main(String [] args) {
        printMenu();
        Scanner input= new Scanner(System.in);
        int select;
        // Boucle tant que le choix est invalide
        do {
            select= input.nextInt();
        } while ((select < 0) || (select > 2));
        System.out.println("Choix = " + select);
    }
}
```

Pourquoi utiliser **do...while** ?

La condition de continuation nécessite d'avoir obtenu la sélection de l'utilisateur.

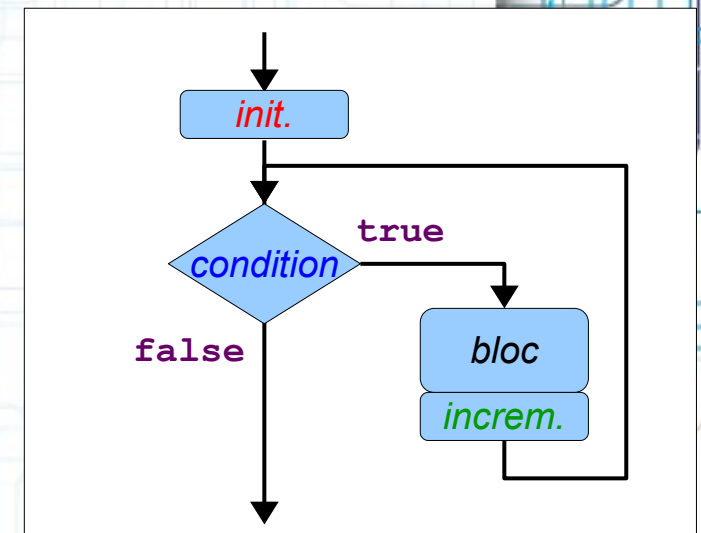
Boucles

- **Boucle for**

- La boucle **for** est adaptée aux boucles qui maintiennent un compteur/index incrémenté à chaque itération. la boucle permet d'effectuer l'initialisation du compteur, d'exprimer une condition de continuation et d'effectuer l'incrément du compteur.
- Syntaxe

```
for ( initialisation ; condition ; incrementation )  
    bloc
```

- *initialisation* effectuée une seule fois avant la boucle ;
- *condition* testée avant chaque itération ;
- *incrémentation* effectuée à la fin de chaque itération.
- Note : *initialisation* et *incrémentation* sont des séquences d'instructions séparées par des virgules.



Boucles

- **Boucle for**

- Exemple

```
for ( int index= 0 ; index < 10 ; index++ )  
    System.out.println(index);
```

- Exemple équivalent avec **while**

```
int index= 0;  
while ( index < 10 ) {  
    System.out.println(index);  
    index++;  
}
```

Note : dans cet exemple, la variable `index` est déclarée dans le bloc `for`. Sa portée est donc limitée à ce bloc \Rightarrow elle n'est pas utilisable en dehors de la boucle `for`.

Boucles

- **Sortie prématurée : break**

- Il est possible de forcer la sortie d'une boucle sans que la condition de sortie de boucle soit remplie, en utilisant le mot-clé **break**. Ce mot-clé peut être utilisé avec toutes les boucles du langage Java.

- Exemple

```
int number;  
System.out.println("Entrez des nombres (0=fin)");  
while (true) {  
    number= readNumber();  
    if (number == 0)  
        break;  
    // fait quelquechose avec 'number'  
}
```

- Dans le cas de boucles imbriquées, **break** sort uniquement de la boucle qui le contient.

Boucles

- **Itération suivante : `continue`**

- Il est possible de passer directement à l'itération suivante d'une boucle en utilisant le mot-clé `continue`. Ce mot-clé peut être utilisé avec toutes les boucles du langage Java.
- Exemple

```
int somme= 0;
for (int i= 1; i <= 100; i++) {
    if (i % 2 == 0)
        continue;
    somme+= i;
}
System.out.println(somme);
```

- Question subsidiaire
 - que fait cette boucle ?
 - quelle valeur est affichée ?

Table des Matières

1. Le Langage Java
2. Compilation et exécution
3. Variables, types et littéraux
4. Opérateurs et expressions
5. Chaînes de caractères
6. Méthodes
7. Structures de contrôle
- 8. Ecrire des commentaires**

Commentaires

- **Comment écrire des commentaires ?**
 - On demande aux étudiants en informatique (et aux professionnels) de commenter leur code. C'est une pratique parfois difficile à mettre en œuvre. Voici quelques règles de bonne pratique.
 - Question : **A quoi servent les commentaires ?**
 - Réponse : **A faciliter la lecture du code.**
 - Aider à la compréhension (p.ex. détails d'implémentation)
 - Aider à l'utilisation (p.ex. informer de pré-/post-conditions)
 - Donner une vue de haut-niveau (architecture du programme).
 - Indiquer une référence vers une explication plus détaillée.
 - Indiquer la source du code, s'il est repris d'ailleurs.

Commentaires

- **Comment écrire des commentaires ?**
 - **Règle 1 : Ecrire du code simple à comprendre.** Le bon code nécessite moins de commentaires !!!
 - **Règle 2 : Ne pas commenter ce qui est évident.** Les commentaires ci-dessous ajoutent du bruit inutile au code.

```
/* retourne un succès */  
return SUCCESS;
```

```
count++; /* incrémente le compteur */
```

```
int anneeEtudiant; /* variable entière */
```


Commentaires

- Comment écrire des commentaires ?

- Règle 3 : Commenter les fonctions et les données globales

- une ligne peut suffire

```
// retourne la factorielle d'un nombre dans [0..12]  
public static int fact(int n) { /* ... */ }
```

- si le code est difficile (algo ou structures de données complexes), un commentaire plus long peut être nécessaire.

commentaires
"javadoc"

```
/**  
 * Prédit le nombre de vaches folles par exploitation  
 * Utilise l'algorithme de Krutzfeld-Jakobs, ACM SIGCOW, 1993  
 *  
 * @param frac fraction de farines animales dans  
 *           l'alimentation. frac est dans [0..1].  
 * @param epsilon facteur de risque lié à la race  
 *           epsilon est dans [-1..1]. */  
public static int estimerVachesFolles(float frac, float  
epsilon) { /* ... */ }
```

Commentaires

- **Comment écrire des commentaires ?**
 - **Règle 4 : Ne pas contredire le code.**
 - exemple : le commentaire indique quelque chose que le code ne fait pas
 - **Règle 5 : Rester cohérent dans le choix de la langue.**
 - Anglais ou français.
 - Garder la même langue dans tout le programme.
 - **Règle 6 : Clarifier, ne pas ajouter de la confusion**
 - Eviter un commentaire aussi long voire plus long que le code.
 - **Règle 7 : Ne pas commenter le code *puant*, le ré-écrire.**



Devoir



- **Avant le premier TP**

- Apprendre à utiliser le compilateur et la machine virtuelle en ligne de commande : être capable de compiler et exécuter le petit exemple introductif `HelloPrinter`.
- S'assurer de savoir accéder à la documentation de l'API Java.
- Reproduire certains exemples du chapitre (p.ex. ceux illustrant les *boucles*).
- Eventuellement installer et essayer un environnement de développement intégré (IDE) tel que *Eclipse* ou *NetBeans*.