

# **Licence Professionnelle**

## **Ingénierie des Systèmes Informatiques et Logiciels**

### **(ISIL)**

# **PROGRAMMATION ORIENTÉE OBJET**

## **(JAVA)**

*Pr. Said BENKIRANE*  
*2017/2018*

# Programmation Orientée Objet (Java)

Période : 1<sup>er</sup> semestre (S5)

| Cours | TD  | TP  | Evaluation | VH Global |
|-------|-----|-----|------------|-----------|
| 20h   | 10h | 16h | 4h         | 50h       |

**DS (Théoriques et pratiques)**

+

**Mini projet (exposé)**

+

**Examen de fin de semestre**

# Programme

Partie 1: Le langage Java

partie 2: La Programmation Orientée  
Objet (Java).

Partie 3: Les Exceptions, Entrées/Sorties,  
Collections, Threads et Enums

Partie 4: Connexion aux Bases de  
données.

Partie 5: Les interfaces Graphiques

# Partie 1: Le langage Java

- Les plates-formes Java
- Variables et opérateurs
- Structures conditionnelles et itératives
- Les Méthodes
- Les Tableaux
- Les Chaines de caractères

# Partie 2: POO(Java)

- Classes et Objets
- Packages, Contrôle d'accès et Encapsulation
- Membres statiques
- Héritage et polymorphisme
- Classes abstraites et interfaces

# Partie 3: Les Exceptions, Entrées/Sorties, Collections, threads et Enumérations

- Les Exceptions
- Les Entrées/Sorties
- Les Collections
- Les Threads
- Les Enumérations

# Partie 4:Connexion aux Bases de données.

- ❑ Connexion au base de données
- ❑ Exécution des requêtes
- ❑ Les Exceptions appropriées
- ❑ Jointures et transactions

# Partie 5: Les interfaces Graphiques

- ❑ Les Composants graphiques
- ❑ Les Gestionnaires de mise en forme
- ❑ Les Contrôleurs d'évènements
- ❑ Application avec plusieurs interfaces



# **Les plates-formes Java**

- Java SE
- Java EE
- Java ME
- Java SE Advanced & Suite
- Java Embedded
- Java DB
- Web Tier
- Java Card
- Java TV
- New to Java
- Community
- Java Magazine

[Overview](#)[Downloads](#)[Documentation](#)[Community](#)[Technologies](#)[Training](#)

## Java SE Downloads

[DOWNLOAD](#)

Java Platform (JDK) 9

[DOWNLOAD](#)

NetBeans with JDK 8

### Java Platform, Standard Edition

#### Java SE 9

Java SE 9 is the latest update to the Java Platform. This release includes much awaited new features like the modularization of the Java Platform, better performance, support for new standards, and many other improvements.

[Learn more](#)

- [Installation Instructions](#)
- [Release Notes](#)
- [Oracle License](#)
- [Java SE Licensing Information User Manual](#)
- [Third Party Licenses](#)
- [Certified System Configurations](#)
- [Roadmap](#)

JDK

[DOWNLOAD](#)

Server JRE

[DOWNLOAD](#)

JRE

[DOWNLOAD](#)

# Bref historique du langage Java

---

- Le langage **Java** a été conçu au sein de l'entreprise *Sun Microsystems* (par une équipe dirigée par James Gosling).
- Initialement baptisé Oak (chêne), il a été officiellement lancé en 1995 sous le nom **Java**
  - Java est synonyme de "café" en argot américain
- Le langage a été popularisé par :
  - Sa portabilité (indépendance des plates-formes)
  - Le slogan "*Write Once Run Anywhere*"
  - Une syntaxe de base très proche de C/C++
  - Une orientation Web native
  - Des éléments de sécurité intégrés
  - La gratuité de son kit de développement (JDK)
  - Son adoption dans la formation (écoles, universités)
  - Son nom ?

# Caractéristiques principales

---

- Orientation Objet
- (C++) --
- Sécurité intégrée
- Portabilité
- Robustesse (typage fort, gestion de la mémoire, ...)
- Richesse des librairies de classes (*plate-forme Java*)
- Multitâche intégré au langage (*Multithreading*)
- Bonne intégration des communications réseau (*Sockets, RMI, ...*)
- Évolutivité (*SDK 1.0, 1.1, 1.2, 1.3, 1.4, 5.0, 6.0, 7.0, 8.0, ...*)
- Java ≠ JavaScript
- C# est fortement inspiré de Java

# Premier programme Java

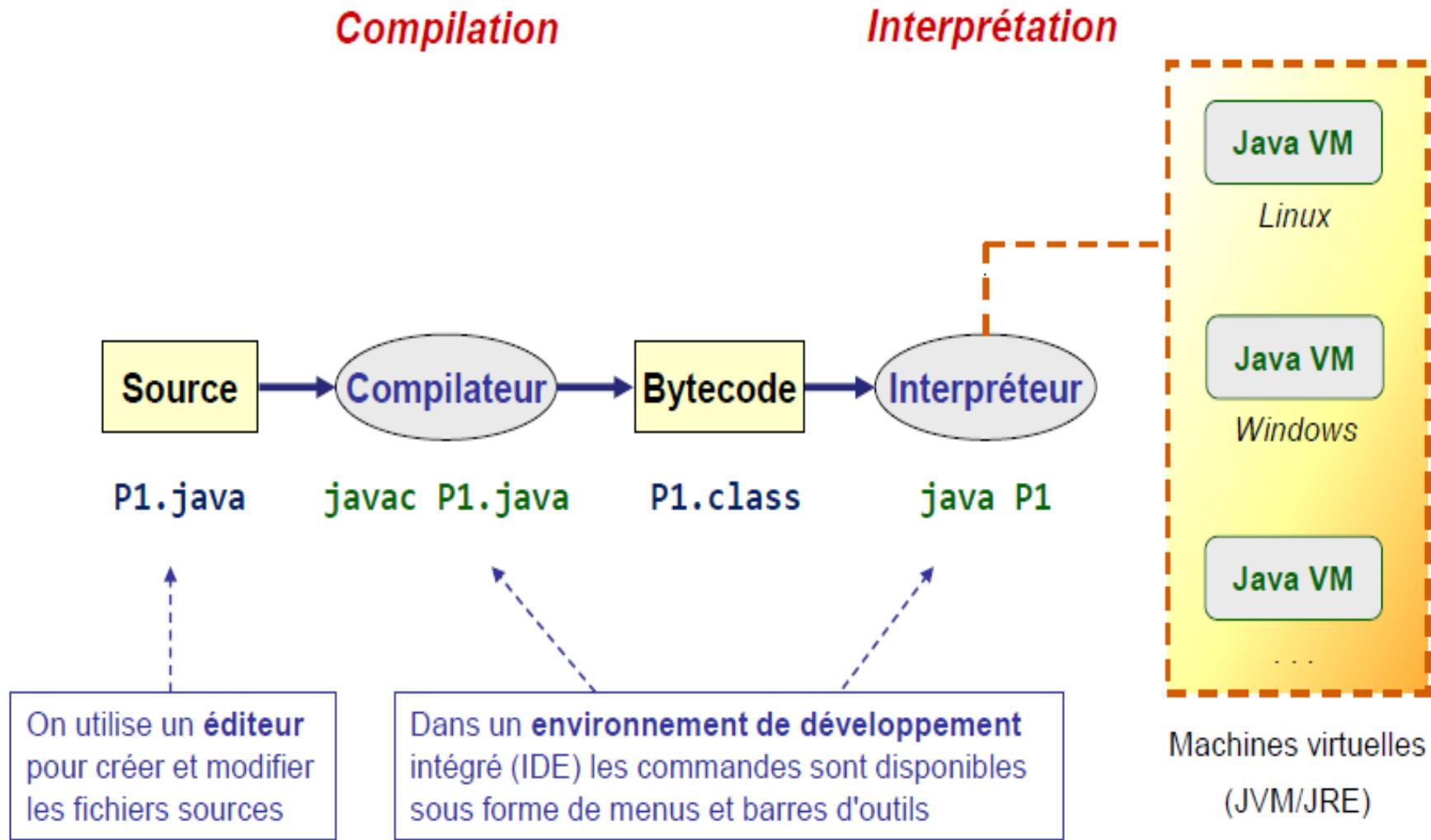
- Un premier programme (très poli) enregistré dans le fichier "Bonjour.java" :

**Nom de la classe** : il doit correspondre au nom du fichier source (sans l'extension)

```
//-----  
// Un premier programme Java  
//-----  
public class Bonjour {  
    public static void main(String[] args) {  
        System.out.println("Bonjour à tous !");  
    }  
}
```

**Point d'entrée du programme**

# Compilation et Interprétation





# **Variables et opérateurs**

# Commentaires

- Trois formes de commentaires :

// ...*Texte*...

- ✓ Commence dès // et se termine à la fin de la ligne
- ✓ Sur une seule ligne
- ✓ A utiliser de préférence pour les commentaires généraux

/\* ...*Texte*... \*/

- ✓ Le texte entre /\* et \*/ est ignoré par le compilateur
- ✓ Peuvent s'étendre sur plusieurs lignes
- ✓ Ne peuvent pas être imbriqués
- ✓ Peuvent être utiles pour inactiver (temporairement) une zone de code

/\*\* ...*Texte*... \*/

- ✓ Commentaire de documentation (comme /\* ... \*/ mais avec fonction spéciale)
- ✓ Interprétés par l'utilitaire javadoc pour créer une documentation au format HTML
- ✓ Peuvent contenir des balises de documentation (@author, @param, ...)
- ✓ Peuvent contenir des balises HTML (Tags)

# Identificateurs

---

- Les **identificateurs** sont des **noms symboliques** permettant de référencer les éléments des programmes Java (variables, fonctions, ...).
- Règles pour les identificateurs :
  - Doivent commencer par une lettre ou un souligné (ou, à éviter, un caractère monétaire)
  - Suivi (éventuellement) d'un nombre quelconque de lettres, chiffres ou soulignés (ou, à éviter, de caractères monétaires)
  - Distinction entre les majuscules et les minuscules (éviter de créer des identificateurs qui ne se distinguent que par la casse)
  - Utilisent le jeu de caractères *Unicode* (16 bits)
  - Selon les plates-formes, les caractères alphabétiques spéciaux (par exemple les caractères accentués, les lettres grecques, ...) peuvent encore poser des problèmes => A éviter
  - Les mots réservés du langage sont exclus (voir liste à la page suivante)

# Mots réservés (*Keywords*)

- Mots réservés du langage Java (mots-clé) :

|                             |                            |                           |                  |                           |
|-----------------------------|----------------------------|---------------------------|------------------|---------------------------|
| <b>abstract</b>             | <b>default</b>             | <b>goto</b> <sup>1)</sup> | <b>package</b>   | <b>synchronized</b>       |
| <b>assert</b> <sup>3)</sup> | <b>do</b>                  | <b>if</b>                 | <b>private</b>   | <b>this</b>               |
| <b>boolean</b>              | <b>double</b>              | <b>implements</b>         | <b>protected</b> | <b>throw</b>              |
| <b>break</b>                | <b>else</b>                | <b>import</b>             | <b>public</b>    | <b>throws</b>             |
| <b>byte</b>                 | <b>enum</b> <sup>4)</sup>  | <b>instanceof</b>         | <b>return</b>    | <b>transient</b>          |
| <b>case</b>                 | <b>extends</b>             | <b>int</b>                | <b>short</b>     | <b>true</b> <sup>2)</sup> |
| <b>catch</b>                | <b>false</b> <sup>2)</sup> | <b>interface</b>          | <b>static</b>    | <b>try</b>                |
| <b>char</b>                 | <b>final</b>               | <b>long</b>               | <b>strictfp</b>  | <b>void</b>               |
| <b>class</b>                | <b>finally</b>             | <b>native</b>             | <b>super</b>     | <b>volatile</b>           |
| <b>const</b> <sup>1)</sup>  | <b>float</b>               | <b>new</b>                | <b>switch</b>    | <b>while</b>              |
| <b>continue</b>             | <b>for</b>                 | <b>null</b> <sup>2)</sup> |                  |                           |

<sup>1)</sup> Mots réservés du langage mais non utilisés actuellement

<sup>2)</sup> Littéraux prédéfinis

<sup>3)</sup> Depuis la version 1.4 du JDK

<sup>4)</sup> Depuis la version 1.5 du JDK

# Notion de variable [1]

- En informatique, la notion de **variable** est une **notion fondamentale** (un concept essentiel).
- Une **variable** définit une **case mémoire nommée et typée**.
  - Le **nom** est représenté par un identificateur
  - Le **type** définit le genre d'informations qui sera enregistré ainsi que les opérations qui pourront être effectuées sur ces informations
  - Il existe un certain nombre de types prédéfinis mais il est également possible de créer ses propres types
- Avant de pouvoir être utilisée, une variable doit préalablement être **déclarée** (définition de l'identificateur et du type).
- Syntaxe :

*Type Identificateur ;*

```
float    note;  
Polygone triangle;
```

note

4.75

## Notion de variable [2]

- La **valeur de la variable** peut être définie (initialisée), consultée, modifiée en utilisant le nom de la variable.
- Les **variables** ont une **visibilité** et une **durée de vie** qui dépendent du contexte dans lequel elles sont déclarées.
- Le **type d'une variable** peut être soit un **type primitif**, soit un **type référence** (c'est-à-dire le nom d'une classe ou d'un tableau)
- Exemples :

```
int      unNombreEntier;    // Type primitif int
String   motDePasse;        // Classe String (prédéfinie)
Point    position;          // Classe Point
char[]   voyelles;         // Tableau de caractères
Point[]  nuage;            // Tableau de Point(s)
```

# Représentation en mémoire

- De manière interne, la zone mémoire allouée à une variable est identifiée par une adresse.
- Cette adresse est complètement cachée au programmeur : l'accès à la zone mémoire s'effectue en utilisant le nom de la variable.

*Code Java*

```
...  
float note;  
...
```

*Représentation interne*

| Variable | Adresse | Contenu |
|----------|---------|---------|
|          | 1215    |         |
| note     | 1216    |         |
|          | 1217    |         |

*Notation abstraite  
(à utiliser)*

note

```
...  
note = 5.2;  
...
```

*Représentation interne*

| Variable | Adresse | Contenu |
|----------|---------|---------|
|          | 1215    |         |
| note     | 1216    | 5.2     |
|          | 1217    |         |

note  5.2

# Types primitifs [1]

| Type           | Contient                     | Taille                 | Valeurs                                |
|----------------|------------------------------|------------------------|--|
| <b>boolean</b> | Booléen                      | [1 bit <sup>1)</sup> ] | true, false                            |
| <b>char</b>    | Caractère (entier non-signé) | 16 bits                | \u0000..\uFFFF                         |
| <b>byte</b>    | Entier signé                 | 8 bits                 | -128..127                              |
| <b>short</b>   | Entier signé                 | 16 bits                | -2 <sup>15</sup> .. 2 <sup>15</sup> -1 |
| <b>int</b>     | Entier signé                 | 32 bits                | -2 <sup>31</sup> .. 2 <sup>31</sup> -1 |
| <b>long</b>    | Entier signé                 | 64 bits                | -2 <sup>63</sup> .. 2 <sup>63</sup> -1 |
| <b>float</b>   | Nombre en virgule flottante  | 32 bits                | $\pm 3.4 \times 10^{38}$               |
| <b>double</b>  | Nombre en virgule flottante  | 64 bits                | $\pm 1.7 \times 10^{308}$              |

<sup>1)</sup> Dépend de l'implémentation de la machine virtuelle

# Types primitifs [2]

## ▪ Booléen (**boolean**)

- Ne peuvent prendre que deux valeurs : *Vrai* ou *Faux*
- Ne peuvent pas être interprétés comme des valeurs numériques [0, 1]
- Valeurs littérales (valeurs qui figurent directement dans le code)
  - ✓ `true`    *Vrai*
  - ✓ `false`    *Faux*

## ▪ Caractère (**char**)

- Caractères *Unicode* (codage normalisé sur 16 bits)
- Site de référence de la norme : [www.unicode.org](http://www.unicode.org)
- Peuvent être traités comme des entiers (non-signés !)
- Valeurs littérales
  - ✓ Entre apostrophes : '`A`' (attention : "`A`" n'est pas de type **char**)
  - ✓ Séquence d'échappement pour certains caractères spéciaux  
(voir table page suivante)

# Types primitifs [3]

- Séquences d'échappement (`char`)

| Code   | Signification   |
|--------|---|
| \b     | Retour en arrière ( <i>Backspace</i> )                              |
| \t     | Tabulateur horizontal   |
| \n     | Saut de ligne ( <i>Line-feed</i> )                                  |
| \f     | Saut de page ( <i>Form-feed</i> )                                   |
| \r     | Retour de chariot ( <i>Carriage-Return</i> )                        |
| \"     | Guillemet   |
| \'     | Apostrophe  |
| \\"    | Barre oblique arrière ( <i>Backslash</i> )                          |
| \xxx   | Caractère <i>Latin-1</i> (xxx : valeur octale 000..377)             |
| \xxxxx | Caractère <i>Unicode</i><br>(xxxx : valeur hexadécimale 0000..FFFF) |

# Types primitifs [4]

- Entiers signés (**byte**, **short**, **int**, **long**)
  - Valeurs littérales (**int** par défaut) : notation habituelle
  - Suffixe **l** ou **L** pour type **long** (**L** est préférable)
  - Préfixe **0** (zéro) pour valeur octale (base 8)
  - Préfixe **0b** ou **0B** pour valeur binaire (base 2) Java 7
  - Préfixe **0x** ou **0X** pour valeur hexadécimale (base 16)
  - Exemples :
    - `0` // valeur de type int
    - `123` // valeur de type int
    - `-56` // valeur de type int
    - `0377` // 377 [octal] = 255 [décimal]
    - `433L` // valeur de type long
    - `0b1011` // valeur binaire de type int
    - `0xff` // valeur hexadécimale de type int
    - `0xA0B3L` // valeur hexadécimale de type long

# Types primitifs [5]

## ▪ Nombres en virgule flottante (**float**, **double**)

- Norme IEEE 754-1985
- Précision
  - ✓ **float** : env. 6 chiffres significatifs
  - ✓ **double** : env. 15 chiffres significatifs
- Valeurs littérales (**double** par défaut) : notation habituelle (*n.m*)
- Suffixe **f** ou **F** pour type **float**
- Suffixe **d** ou **D** pour type **double** (rarement nécessaire)
- Notation exponentielle avec **e** ou **E** suivi de l'exposant
- Valeurs spéciales : *infini*, *-infini*, zéro négatif, *NaN* (*Not a Number*)

**Attention** : Les nombres en virgule flottante sont des approximations de nombres réels (tous les nombres réels ne peuvent pas être représentés de manière exacte).

Il faut en tenir compte dans les comparaisons et notamment dans les tests d'égalité (prendre en compte un "epsilon").

# Types primitifs [6]

- Nombres en virgule flottante (**float**, **double**)

```
123.45          // valeur de type double (par défaut)  
17d             // valeur de type double  
-4.032F         // valeur de type float  
6.02e23         // 6.02 x 1023 de type double  
-5.076E-2f      // -5.076 x 10-2 de type float
```

- Le caractère '\_' peut être inséré dans les littéraux numériques Java 7

```
long creditCardNr = 1234_5678_9012_3456L;  
int tenMillions = 10_000_000;  
int bytePattern = 0b11000011_00001111_11110000_00110011;  
float pi           = 3.14_15_93f;
```

- Cas particuliers (problèmes numériques) :

|          |                       |                          |
|----------|-----------------------|--------------------------|
| 1.2/0.0  | // Infini             | Double.POSITIVE_INFINITY |
| -5.1/0.0 | // Moins l'infini     | Double.NEGATIVE_INFINITY |
| 0.0/0.0  | // Not a Number (NaN) | Double.NaN               |

# Affichage sur la console

- Pour afficher une valeur littérale ou le contenu d'une variable sur la console de sortie, on utilise les lignes de code suivantes :

- `System.out.print(...);` // Affichage (reste sur la même ligne)
- `System.out.println(...);` // Affichage et retour à la ligne

- Exemples :

```
System.out.println("Résultats");
System.out.println("-----");
```

Résultats  
-----

```
int size = 123;
char unit = 'm';
System.out.print("Longueur : ");
System.out.print(size);
System.out.print(" ");
System.out.println(unit);
```

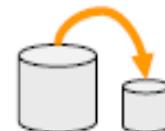
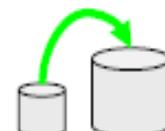
Longueur : 123 m

Même affichage mais en utilisant l'opérateur de concaténation (+)

```
int size = 123;
char unit = 'm';
System.out.println("Longueur : " + size + " " + unit);
```

# Conversions de types [1]

- Mis à part le type booléen, tous les types primitifs peuvent être convertis entre eux.
- L'encodage *Unicode* permet (si nécessaire) de considérer les caractères comme des nombres.  
Le type **char** peut donc agir comme un entier **short** mais il est non signé (contrairement aux autres entiers).
- Conversion élargissante / Promotion** (automatique)  
(avec perte éventuelle de précision lors du passage en virgule flottante)
- Conversion restrictive** explicite (par transtypage / casting)  
(sous la responsabilité du programmeur)



Remarque : Certaines conversions restrictives implicites sont automatiquement effectuées lorsque le résultat d'une **expression constante** est assignée à une variable (**byte**, **short** ou **char**) pour autant que la valeur de l'expression puisse être représentée (sans perte) par la variable.

```
short compteur = 12*34; // Conversion implicite int -> short
```

# Conversions de types [2]

- Conversions des types primitifs



|               |         | Conversion vers |      |       |      |     |      |       |        |
|---------------|---------|-----------------|------|-------|------|-----|------|-------|--------|
|               |         | boolean         | byte | short | char | int | long | float | double |
| Conversion de | boolean | -               | N    | N     | N    | N   | N    | N     | N      |
|               | byte    | N               | -    | E     | R    | E   | E    | E     | E      |
|               | short   | N               | R    | -     | R    | E   | E    | E     | E      |
|               | char    | N               | R    | R     | -    | E   | E    | E     | E      |
|               | int     | N               | R    | R     | R    | -   | E    | E*    | E      |
|               | long    | N               | R    | R     | R    | R   | -    | E*    | E*     |
|               | float   | N               | R!   | R!    | R!   | R!  | R!   | -     | E      |
|               | double  | N               | R!   | R!    | R!   | R!  | R!   | R*    | -      |

N : pas de conversion possible

R : conversion restrictive

E : conversion élargissante  
(promotion)

\* : perte de précision

! : valeur tronquée

# Transtypage (Casting)

- Conversion de type explicite (au risque du programmeur)
- Syntaxe : **( Type\_de\_destination ) Valeur\_à\_convertir**
- Exemples :

```
int i = 17;
byte b = 4;           // Conversion implicite : littéral int -> byte
b = i;                // Erreur à la compilation
b = (byte)i;          // Conversion explicite (int -> byte)
```

```
float f = 23.86f;
i = (int)f;           // Valeur tronquée (i==23)
```

# Types "Référence"

- En plus des huit types primitifs, Java définit deux autres catégories de types de données :

- les **classes** et
- les **tableaux**

qui seront étudiés en détail plus tard.

- Les classes et les tableaux sont des types généralement composites (représentant des valeurs multiples).
- Ils sont collectivement nommés : **Types "Référence"** (on parle également parfois de types **non-primitifs**).
- Pour une variable de type référence, la case mémoire ne contient pas directement les valeurs de la variable mais une *référence* (une sorte de pointeur) vers une autre zone mémoire contenant les valeurs.

```
String nom = "Dupont";
```



# Chaînes de caractères

- Les chaînes de caractères ne font pas partie des types primitifs
- Les chaînes de caractères font partie des **types référence** (ce sont des objets de la classe **String**)
- D'usage courant  $\Rightarrow$  Syntaxe particulière (simplifiée) pour les valeurs littérales :
  - Texte entre guillemets ("...")
  - Peuvent contenir des séquences d'échappement identiques à celles définies pour les types **char**
  - Ne peuvent pas s'étendre sur plusieurs lignes
  - Concaténation (mise bout à bout) avec l'opérateur **+**
  - Conversions possibles : *types primitifs*  $\Leftrightarrow$  **String**

"Ceci est une chaîne de caractères"

"Le caractère \" se nomme 'guillemet'"

"La concaténation peut s'étendre " +  
"sur plusieurs lignes"

# Conversions types primitifs ⇔ String

- Il est possible de convertir des valeurs de types primitifs en chaînes de caractères (`String`) et inversement en utilisant des fonctions (méthodes) disponibles dans des classes appelées *Wrappers*.

|               |         | Conversions                     |                                    |
|---------------|---------|---------------------------------|------------------------------------|
| Type primitif | Wrapper | en String                       | de String                          |
| byte          | Byte    | <code>toString(byte b)</code>   | <code>parseByte(String s)</code>   |
| short         | Short   | <code>toString(short s)</code>  | <code>parseShort(String s)</code>  |
| int           | Integer | <code>toString(int i)</code>    | <code>parseInt(String s)</code>    |
| long          | Long    | <code>toString(long l)</code>   | <code>parseLong(String s)</code>   |
| float         | Float   | <code>toString(float f)</code>  | <code>parseFloat(String s)</code>  |
| double        | Double  | <code>toString(double d)</code> | <code>parseDouble(String s)</code> |

D'autres fonctions de conversion sont également disponibles dans la classe `String`, notamment la méthode `valueOf()` qui permet de convertir en `String` les valeurs de tous les types primitifs.

# Exemples de conversions

```
int    i;                      // Variable de type int
float  f;                      // Variable de type float
double d;                      // Variable de type double
String s;                      // Variable de type String

s = "412";
i = Integer.parseInt(s);        // Conversion String -> int

s = "1.234";
d = Double.parseDouble(s);      // Conversion String -> double

f = 123.8F;
s = Float.toString(f);         // Conversion float -> String
s = String.valueOf(f);         // Conversion float -> String
```

**Attention** : Ne pas confondre les littéraux de type caractère ou chaîne de caractères avec les littéraux numériques. Par exemple '4', "4" et 4.

Le premier est un caractère (type `char`), le deuxième une chaîne de caractères (type `String`) et le troisième est une valeur numérique entière (type `int`).

# Opérateurs

- Les **opérateurs** sont des éléments syntaxiques qui effectuent certaines opérations en utilisant les valeurs de leurs **opérandes** (paramètres de l'opération).



- ## ▪ Opérateurs unaires (monadiques)      1 opérande

-b Opérateur monadique moins

a++ Opérateur monadique de post-incrémentation

- **Opérateurs binaires** (dyadiques)      *2 opérandes*

**a \* b** L'opérateur de multiplication est binaire

`y = z` L'affectation est également binaire

(int)y Transtypage (casting)



- Un seul opérateur ternaire en Java

`x > y ? x : y`      Retourne la valeur maximale entre x et y

# Liste des opérateurs [1]

| Opérateur              | Type(s) des opérandes                           | Opération   |
|------------------------|---|---|
| .                      | objet, membre                                   | accès au membre d'un objet  |
| []                     | tableau, int                                    | accès à l'élément d'un tableau  |
| ( params )             | méthode, liste de paramètres                    | invocation (appel) de méthode   |
| ++, --                 | variable  | post-incrémantation, décrémentation   |
| ++, --                 | variable  | pré-incrémantation, décrémentation  |
| +, -                   | nombre  | plus monadique, moins monadique   |
| ~                      | entier  | complément binaire  |
| !                      | booléen   | NON logique   |
| <b>new</b><br>( type ) | classe, liste de paramètres<br>type, quelconque | création (instanciation) d'objet<br><b>transtypage</b> (conversion de type) |
| *, /, %                | nombre, nombre                                  | multiplication, division, modulo  |
| +, -                   | nombre, nombre                                  | addition, soustraction  |
| +                      | chaîne de caract., quelconque                   | concaténation   |
| <<, >>, >>>            | entier, entier                                  | décalage à gauche, droite, non-signé  |
| <, <=                  | nombre, nombre                                  | inférieur, inférieur ou égal  |
| >, >=                  | nombre, nombre                                  | supérieur, supérieur ou égal  |
| <b>instanceof</b>      | référence, type                                 | comparaison de types  |
| ==                     | primitif, primitif                              | égalité (valeurs égales)  |
| !=                     | primitif, primitif                              | non-égalité   |
| ==                     | référence, référence                            | égalité (référence au même objet)   |
| !=                     | référence, référence                            | non-égalité   |

# Liste des opérateurs [2]

| Opérateur   | Type(s) des opérandes           | Opération                             |
|---|---------------------------------|---------------------------------------|
| &   | entier, entier                  | ET binaire (bit à bit)                |
| &   | booléen, booléen                | ET logique                            |
| ^   | entier, entier                  | OU exclusif binaire (bit à bit)       |
| ^   | booléen, booléen                | OU exclusif logique                   |
|   | entier, entier                  | OU binaire (bit à bit)                |
|   | booléen, booléen                | OU logique                            |
| &&  | booléen, booléen                | ET logique conditionnel <sup>1)</sup> |
|   | booléen, booléen                | OU logique conditionnel <sup>1)</sup> |
| ? :   | booléen, quelconque, quelconque | opérateur conditionnel (ternaire)     |
| =   | variable, quelconque            | affectation                           |
| *=, /=, %=,<br>+=, -=, <<=,<br>>>=, >>>=,<br>&=, ^=,  = | variable, quelconque            | affectation avec opération            |

# Affectation / Assignation

- Enregistre dans l'opérande de gauche (variable) la valeur de l'opérande de droite (valeur de l'expression)
- Le type de la variable (opérande de gauche) doit être compatible avec le type de l'expression de droite (si nécessaire, conversion élargissante automatique)
- Le type et valeur de retour d'une expression d'affectation correspondent au type de la variable et, respectivement, à sa valeur après affectation
- **Attention** : Ne pas confondre l'opérateur d'affectation (**=**) avec celui d'égalité (**==**)  
*"prend pour valeur"* et non pas *"est égal à"*

```
a = b + c; // La variable a prend pour valeur le résultat de la somme (b+c)
```

```
a = b = c; // Associativité à droite => a = (b = c)
```

```
t[i] = circle.center();
```

# Affectation avec opérateur

- Combinaison de l'affectation avec un opérateur (arithmétique ou orienté bits)
- Le type de l'opérande de gauche (variable) doit être compatible avec le type de l'expression de droite

`var op= expr` est équivalent à `var = var op ( expr )`

- Opérateurs combinés :    `+=`    `-=`    `*=`    `/=`    `%=`  
                               `&=`    `|=`    `^=`  
                               `<<=`    `>>=`    `>>>=`

```
i += 2;           // i = i + 2;  
a *= z + 4;      // a = a * (z + 4);  
flag |= mask;    // flag = flag | mask;  
                  // set some bits according to a mask
```

# Opérateur instanceof

- Permet de tester si une expression (de type référence) est compatible (convertible) avec un type donné.
- Retourne `true` si l'opérande de gauche (qui doit être une expression de type objet ou tableau) est une instance du type spécifié par l'opérande de droite (qui doit être un type référence)

```
"Texte" instanceof String;          // true
"Texte" instanceof Object;          // true
null    instanceof String;          // false
new int[] {1} instanceof Object;    // true
new int[] {1} instanceof byte[];   // false tableau d'int ⇔ tableau de byte

// Utile pour prévenir des erreurs de transtypage
if (forme instanceof Polygone) {
    Polygone p1 = (Polygone)forme;
}
```

# Opérateurs particuliers

- Ces opérateurs particuliers sont parfois considérés comme des **éléments syntaxiques**, parfois comme **opérateurs** :
  - Accès à un membre d'un objet (.)  
`obj.x`  
`obj.f()`
  - Accès à un élément d'un tableau ([ ])  
`t[2]`
  - Invocation de méthode (())  
`rectangle.move(x, y)`
  - Création d'un objet (new)  
`new Point(4.0, -2.5);`  
`new ArrayList();`
  - Conversion de type / Transtypage (())  
`(float)position`  
`(int)(a*1.414f)`  
`(Circle)shape`

# Instructions [1]

---

- Une **instruction** définit une opération qui sera exécutée par la machine virtuelle Java.
- Par défaut, les instruction sont exécutées séquentiellement (les unes après les autres), dans l'ordre dans lequel elles ont été rédigées (ordre d'apparition dans le fichier source).
- Cependant, certaines instructions de **contrôle de flux** permettent de modifier l'ordre d'exécution d'une manière bien définie (instructions exécutées conditionnellement, répétition d'instructions, etc.).
- Les **expressions avec effets de bord** (affectation, incrémentation, décrémentation et invocation de méthodes) peuvent être utilisées comme instructions; elles sont alors suivies d'un point-virgule :

```
j = 2;  
count++;  
move(dx, dy);
```

## Instructions [2]

- Une **instruction vide** est constituée d'un simple point-virgule (`;`)  
Une instruction vide ne fait rien, mais son expression syntaxique peut parfois être utile.
- Une instruction peut s'étendre sur plusieurs lignes du fichier source  
Dans ce cas, les retours à la ligne doivent être placés entre les éléments syntaxiques.
- Plusieurs instructions peuvent être placées sur une même ligne du fichier source.  
Cette pratique est généralement à éviter et doit être réservée aux cas où les instructions sont fortement corrélées et assez courtes.
- Le point-virgule marque la fin d'une instruction (et non la fin d'une ligne).

```
screen.draw(x, y);  
x += dx; y += dy;
```

# Bloc d'instructions

- Un **bloc d'instructions** (appelé également **instruction composée**) est constitué d'un certain nombre d'instructions quelconques placées entre accolades ( `{ ... }` ).
- Un bloc d'instructions peut être utilisé pratiquement partout où une instruction est requise par la syntaxe Java.

```
if (a<b)
{
    // Début du bloc d'instructions
    t = b;
    b = a;
    a = t;
}
// Fin du bloc d'instructions
```

- Pour mettre en évidence la structure du code, on **indente** les instructions à l'intérieur du bloc.

La forme suivante est fréquemment utilisée :

```
if (a<b) {
    t = b;
    b = a;
    a = t;
}
```

*( Notez bien l'indentation )*

# Déclaration de variables locales [1]

- Une déclaration de **variable locale** (souvent simplement appelée *variable*) définit une zone mémoire en lui associant un **nom** (identificateur) et un **type**.  
La déclaration permet en outre de spécifier (optionnellement) la **valeur initiale** de la variable.
- Syntaxe :

**Type Identificateur [= Expression];**
- Toutes les variables locales doivent être **déclarées et initialisées avant d'être utilisées**.
- En Java, les déclarations de variables peuvent être placées n'importe où dans les blocs d'instructions.

**Conseil** : Pour faciliter la lecture du code, grouvez de préférence les déclarations de variables **au début des structures** (méthodes, blocs d'instructions)

## Déclaration de variables locales [2]

---

- Une seule instruction de déclaration de variable permet de **déclarer** et d'**initialiser plusieurs variables**, pour autant qu'elles soient toutes du même type.
- Lors de déclarations multiples, les noms des variables (avec leurs éventuelles valeurs initiales) sont séparés par des virgules.

```
int counter;  
Point p;  
  
int number= 0;           // Déclaration et initialisation  
String s = getName();   // Valeur connue à l'exécution seulement  
int i, j, k;             // Déclarations multiples  
int i=2, j=5, k;         // i et j sont initialisés, k ne l'est pas
```

# Représentation en mémoire

- Une déclaration de variable locale définit (dans la mémoire vive) une zone mémoire dont la taille et le codage dépend du type de la variable.
- Les détails d'implémentation ne sont pas importants (mais il faut retenir le principe).

```
{  
    int price;  
    price = 3;  
    {  
        int cost=2*price;  
    }  
}
```

| Variable | Adresse | Contenu |
|----------|---------|---------|
|          | 1215    |         |
| price    | 1216    | 3       |
|          | 1217    |         |
| cost     | 1218    | 6       |
|          | 1219    |         |

*Représentation abstraite usuelle (notation à utiliser)*

price 3

cost 6

# Portée des variables

- Les variables locales ne peuvent être utilisées que dans la méthode ou le bloc d'instruction au sein duquel elles ont été déclarées
- On définit la **portée** (scope) ou **portée lexicale** d'une variable la zone de code où la variable est accessible (cette zone va de sa déclaration jusqu'à la fin du bloc dans lequel elle est déclarée).
- A la fin de leurs portées lexicales, les variables ne sont plus visibles et le système peut récupérer l'espace mémoire qui leur était alloué

```
public static void m() {  
    int i = 0;                      // i  
    while (i<10) {                  // i  
        double d = 1.2 * i;          // i d  
        System.out.println(d);       // i d  
        i++;                        // i d  
    }                                // i  
    System.out.println(i);           // i  
}
```



# **Structures conditionnelles et itératives**

# La sélection

---

- On appelle **sélection** la possibilité de **choisir d'exécuter** une ou plusieurs instructions **en fonction de la situation** (exprimée sous la forme d'une condition) prévalant à l'instant considéré.
- On trouve **trois formes** de réalisation en Java :
  - L'**instruction conditionnelle** (**if (cond) ...**) qui permet d'exécuter ou non un bloc d'instructions.
  - L'**instruction d'alternative** (**if (cond) ... else ...**) qui permet d'exécuter un bloc d'instructions ou, alternativement, un autre bloc d'instructions
  - L'**instruction sélective** (**switch (expr) case ...**) qui permet (dans sa variante habituelle) de choisir d'exécuter un parmi plusieurs blocs d'instructions

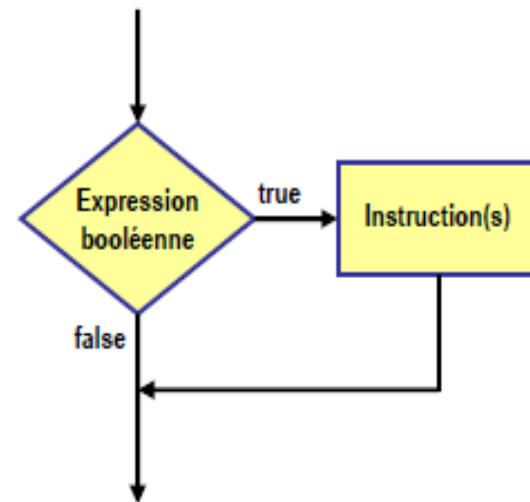
# Instruction if

```
if ( expression_booléenne ) instruction
```

- Exécution d'une instruction de manière conditionnelle
- La **condition** est exprimée par une **expression booléenne**
- L'instruction est exécutée si la condition est vraie

```
if (age >= 18) status = "Adult";
```

```
if ((age<16) || isStudent) {  
    canHaveAllocations = true;  
    amount = 400;  
}
```

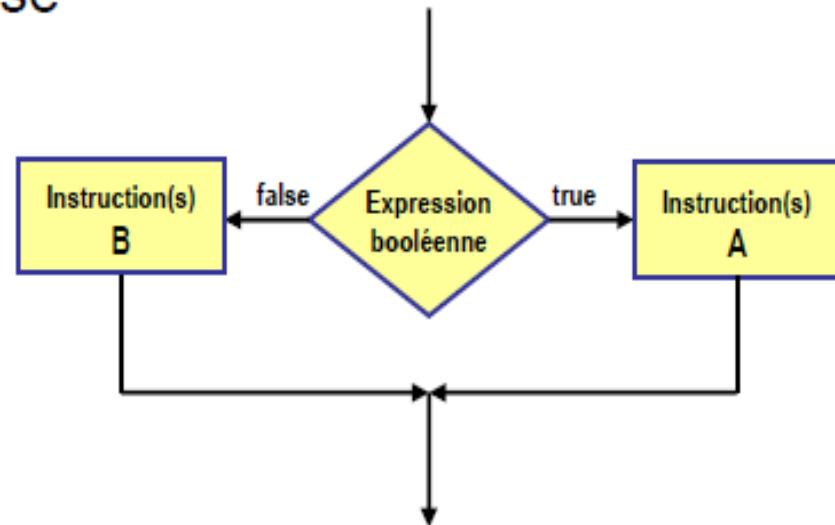


# Instruction if ... else

```
if ( expression_booléenne ) instruction_A else instruction_B
```

- Une clause **else** peut (optionnellement) être mentionnée dans l'instruction **if**
- L'instruction qui suit la clause **else** est exécutée si l'expression booléenne (la condition) est fausse

```
if (average >= 4)
    System.out.println("Pass");
else {
    System.out.println("Fail");
    repeat = true;
}
```



# Emboîtement d'instructions if ... else

- L'instruction **if / else** peut contenir dans chacune de ses deux branches une autre instruction **if / else** qui peut elle-même contenir une autre instruction ...
- On peut ainsi avoir plusieurs niveaux d'emboîtement formant une logique plus ou moins complexe.
- En l'absence de blocs d'instructions (`{...}`), **la clause else se rapporte toujours au if précédent (sans else) le plus proche** (l'indentation ne change pas la logique du code !).

```
if (x >= 2)
    if (x <= 20)
        status = 1;
else
    if (x <= 10)
        status = 2;
else
    status = 3;
```

```
if (x >= 2)
    if (x <= 20)
        status = 1;
else
    if (x <= 10)
        status = 2;
else
    status = 3;
```

```
if (x >= 2) {
    if (x <= 20)
        status = 1;
}
else {
    if (x < 0)
        status = 0;
else
    status = 2;
}
```

L'indentation est importante pour la lisibilité du code !

## Imbrication if ... else if ... [1]

- Lorsque l'on doit choisir une instruction (ou un bloc d'instructions) parmi plusieurs (groupes d'instructions mutuellement exclusifs) il faut imbriquer les clauses **if** **else if** sur plusieurs niveaux.

```
if (n==1) {  
    ...                                // Code executed if n is 1  
}  
else {  
    if (n==2) {  
        ...                                // Code executed if n is 2  
    }  
    else {  
        if (n==3) {  
            ...                                // Code executed if n is 3  
        }  
        else {  
            if (n==4) {  
                ...                                // Code executed if n is 4  
            }  
            else {  
                ...                                // Code executed otherwise  
            }  
        }  
    }  
}
```

## Imbrication if ... else if ... [2]

- Si le nombre d'alternatives est grande, l'indentation provoque un décalage du code vers la droite qui peut devenir gênant.
- C'est pourquoi l'on préférera, dans ce cas, la disposition suivante :

```
if      (n==1) {  
    ...          // Code executed if n is 1  
}  
else if (n==2) {  
    ...          // Code executed if n is 2  
}  
else if (n==3) {  
    ...          // Code executed if n is 3  
}  
else {  
    ...          // Code executed otherwise  
}
```

Remarque : Il ne s'agit pas d'une nouvelle syntaxe mais simplement d'une mise en page plus lisible (sans cumul de l'indentation).

# Instruction switch [1]

```
switch ( expression ) {  
    case const1 : suite_instructions_1  
    case const2 : suite_instructions_2  
    ...  
    default : suite_instructions_default  
}
```

- Instruction de sélection multiple parmi une liste d'instructions (indexées) et dont le **point d'entrée** est déterminé par la valeur d'une expression (constante) de type `byte`, `short`, `int`, `char` d'un type énuméré (`enum`) ou `String` Java 7
- A partir du point d'entrée, l'exécution se poursuit ensuite dans les instructions suivantes (celles des autres clauses `case`) jusqu'à la fin de l'instruction `switch` ou jusqu'à l'instruction `break` (ou `return`) qui interrompt l'instruction `switch`

# Instruction switch [2]

```
n = ...          // Variable of type byte, short, int, char, enum or String  
switch (n) {  
    case 1 :  
        Instruction; // Start here if n is 1  
        Instruction;  
        ...  
    case 5 :  
    case 6 :  
        Instruction; // Start here if n is 5 or 6  
        Instruction;  
        ...  
    case 8 :  
        Instruction; // Start here if n is 8  
        Instruction;  
        ...  
    default :  
        Instruction; // Start here if n ≠ {1, 5, 6, 8}  
        Instruction;  
        ...  
}
```

Java 7

- Point d'entrée

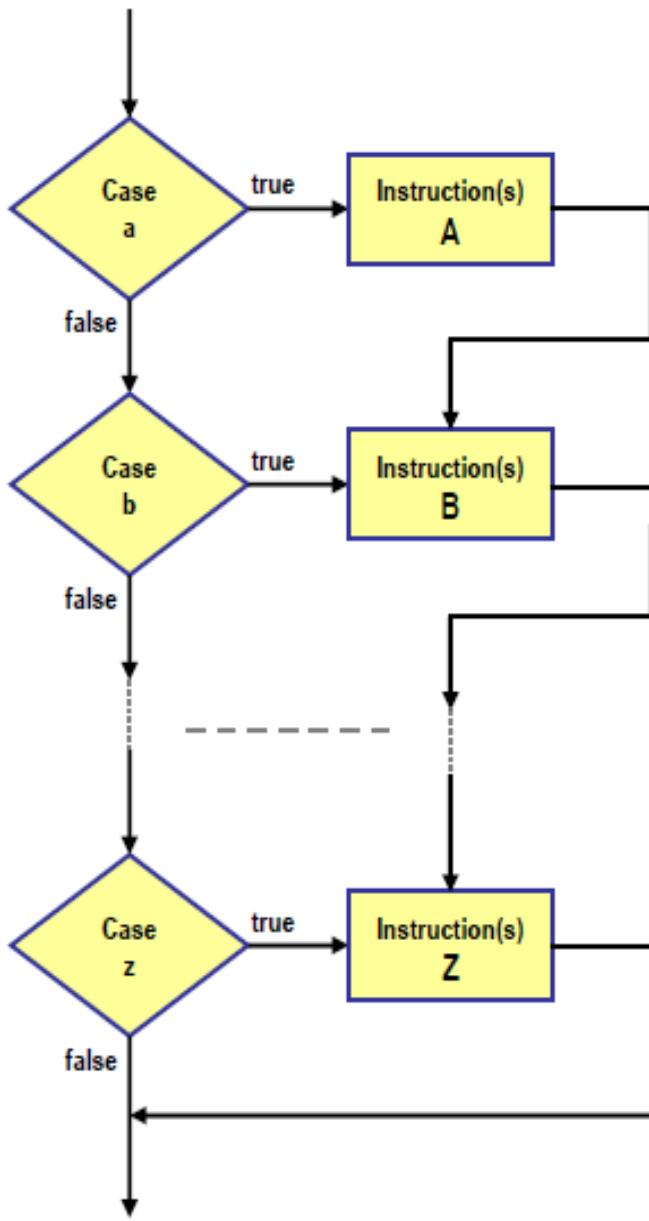
A partir du point d'entrée, toutes les instructions qui suivent sont exécutées jusqu'à la fin de l'instruction switch ou jusqu'à l'instruction break (ou autre instruction d'interruption comme return, throw, ...)

## Instruction switch [3]

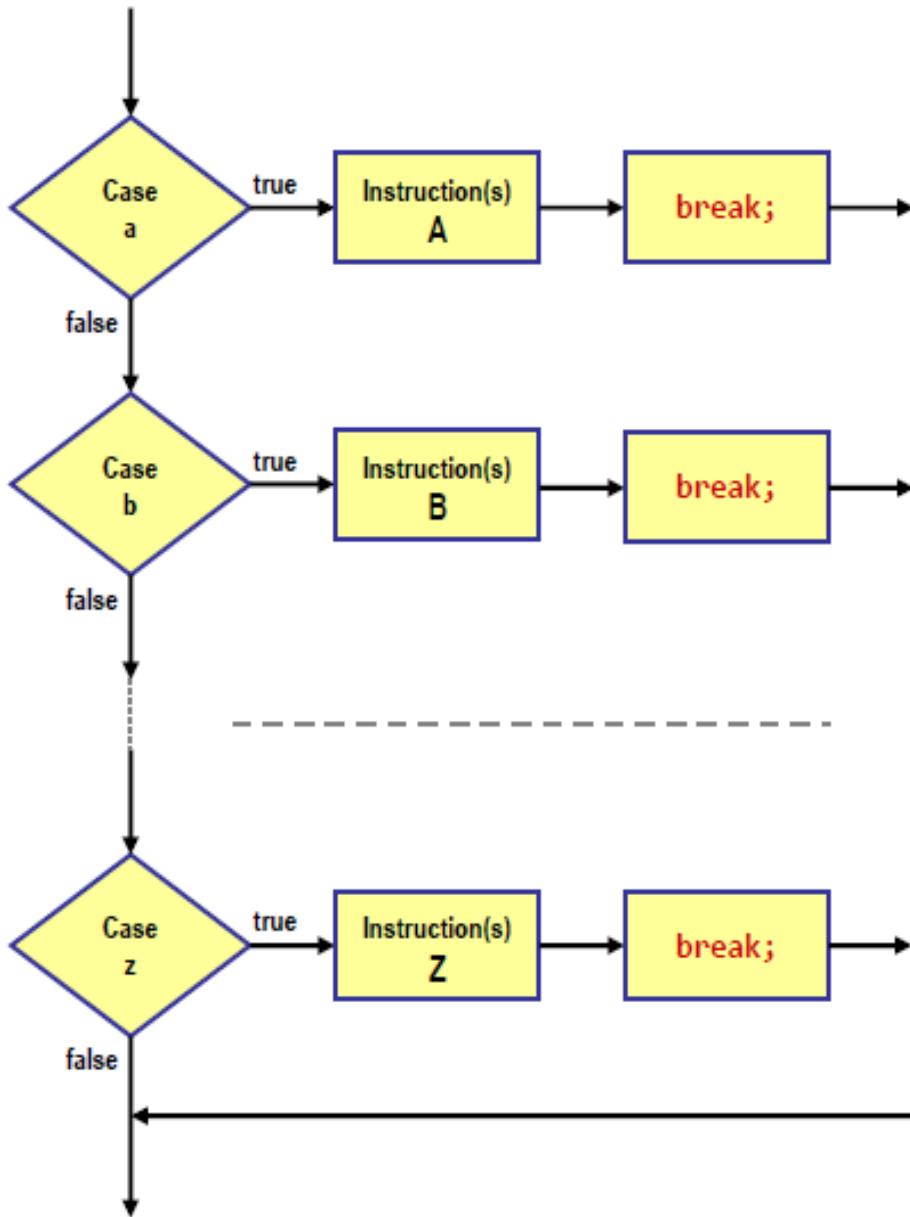
- Le mot clé **default** définit un point d'entrée qui est sélectionné si la valeur de l'expression ne correspond à aucune valeur dans la liste des clauses **case**. Le mot clé **default** est optionnel et est généralement placé comme dernière clause (ce n'est pas imposé par le langage mais fortement recommandé).
- Plusieurs clauses **case** peuvent étiqueter la même suite d'instructions (même point d'entrée pour plusieurs valeurs différentes).
- Les valeurs associées aux clauses **case** doivent être des valeurs ou expressions constantes (évaluables par le compilateur).
- Toutes les valeurs des clauses **case** doivent être différentes.

L'instruction **switch** est une instruction de bas niveau qui peut être remplacée par un enchaînement d'instructions **if...else**. Les instructions **if...else** offrent, en outre, davantage de flexibilité (expressions de types quelconques, utilisation de variables et d'expressions relationnelles et logiques dans les clauses de sélection) et n'imposent pas l'utilisation quasi systématique d'instructions d'interruption (**break**, **return**, ...).

# Instruction switch [4]



# Instruction switch [5]



# Instruction switch [6]

```
int month, nbDays;  
month = . . .  
switch (month) {  
    case 4 :  
    case 6 :  
    case 9 :  
    case 11 :  
        nbDays = 30; break;  
    case 2 :  
        nbDays = 28; break; // Années bissextiles non considérées  
    default :  
        nbDays = 31; break;  
}
```

**Conseil :** Sauf cas exceptionnels, placez systématiquement une instruction **break** (ou **return**) à la fin de chaque clause **case** car il est rare que l'on souhaite l'exécution des instructions des clauses **case** qui suivent celles du point d'entrée.

# Instruction switch [7]

```
boolean positiveAnswer = false;  
char answer = readKeyboardChar();  
  
switch (answer) {  
    case 'y' :  
    case 'Y' :  
    case 'j' :  
    case 'J' :  
    case 'o' :  
    case 'O' : positiveAnswer = true;  
                break;  
  
    case 'n' :  
    case 'N' : positiveAnswer = false;  
                break;  
  
    default : System.out.println("Caractère incorrect");  
}
```

# Instruction switch [8]

```
String typeOfDay;  
switch (dayOfWeek) {  
    case "Monday":  
        typeOfDay = "Start of work week";      break;  
  
    case "Tuesday":  
    case "Wednesday":  
    case "Thursday":  
        typeOfDay = "Midweek";                  break;  
  
    case "Friday":  
        typeOfDay = "End of work week";       break;  
  
    case "Saturday":  
    case "Sunday":  
        typeOfDay = "Weekend";                 break;  
  
    default:  
        typeOfDay = "Invalid day of the week"; break;  
}
```

Java 7

**Attention :** La comparaison prend en compte les minuscules/majuscules

# Itérations / Boucles

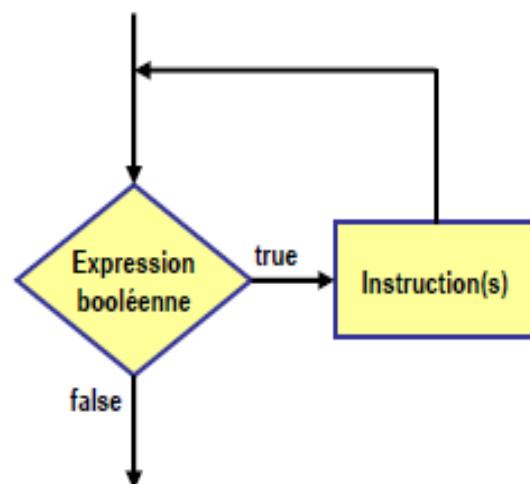
- On appelle **boucle** ou **itération** l'exécution multiple d'une ou de plusieurs instructions **en fonction de la situation** (exprimée sous la forme d'une condition) prévalant à l'instant considéré.
- On trouve **trois formes** de réalisation en Java :
  - L'instruction **while** qui permet de répéter un nombre quelconque de fois (0 ou plus) l'exécution d'une instruction ou d'un bloc d'instructions. Le nombre de répétitions n'est pas forcément connu au moment d'entrer dans la boucle.
  - L'instruction **do ... while** qui permet de répéter un nombre quelconque de fois (1 ou plus) l'exécution d'une instruction ou d'un bloc d'instructions. Le nombre de répétitions n'est pas forcément connu au moment d'entrer dans la boucle.
  - L'instruction **for** qui permet de répéter un nombre quelconque de fois (0 ou plus) l'exécution d'une instruction ou d'un bloc d'instructions. Le nombre de répétitions est généralement connu au moment d'entrer dans la boucle (mais ce n'est pas toujours le cas).

# Instruction while

**while ( expression\_booléenne ) instruction**

- Instruction itérative
  - L'expression booléenne est évaluée
  - Si elle est vraie, l'instruction est exécutée
  - Puis l'expression booléenne est évaluée à nouveau
  - etc...
  - L'instruction **while** se termine si l'expression booléenne est fausse
- L'instruction est exécutée 0, 1 ou  $n$  fois

```
// Affiche les nombres de 0 à 9
int count = 0;
while (count <= 9) {
    System.out.println(count);
    count++;
}
```

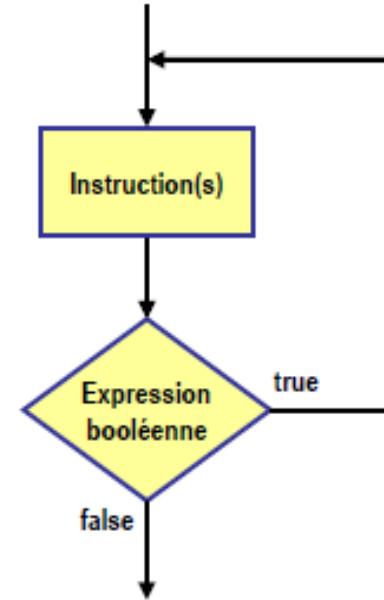


# Instruction do ... while

**do instruction while ( expression\_booléenne );**

- Instruction itérative
  - Exécute l'instruction
  - Évalue l'expression booléenne
  - Si elle est vraie, l'instruction est exécutée à nouveau
  - Évalue à nouveau l'expression booléenne
  - etc...
  - L'instruction **do** se termine si l'expression booléenne est fausse
- L'instruction est exécutée 1 ou  $n$  fois

```
do {  
    display("Nombre positif : ");  
    number = keyboard.readNumber();  
} while (number < 0);
```



# Instruction for [1]

**for** ( *initialisation* ; *expression\_booléenne* ; *conclusion* ) *instruction*

- Instruction itérative
- Les expressions *d'initialisation* et de *conclusion* peuvent être formées par zéro, une ou plusieurs instructions (expressions) séparées par des virgules
- A deux exceptions près<sup>1)</sup>, l'instruction **for** est équivalente à la boucle **while** suivante :

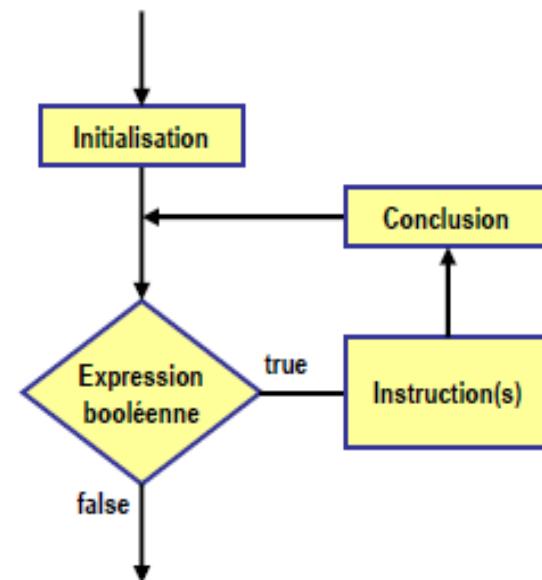
```
initialisation;
while ( expression_booléenne ) {
    instruction;
    conclusion;
}
```

1) - S'il y a un **continue**, *conclusion* sera exécuté dans l'instruction **for** mais pas dans **while**  
- *Initialisation* et *conclusion* ne peuvent pas être des instructions séparées par des virgules dans la boucle **while**

# Instruction for [2]

- En lieu et place d'instructions, *l'initialisation* peut contenir la déclaration (et éventuellement l'initialisation) d'une ou plusieurs variables locales (mais toutes du même type) dont la portée est limitée au corps de la boucle
- La *conclusion* est fréquemment constituée par une instruction de mise à jour d'un compteur de boucle (incrémentation par exemple)
- L'initialisation*, *l'expression booléenne* et *la conclusion* sont facultatives mais le point-virgule est obligatoire (par défaut, l'expression booléenne est vraie)

```
// Affiche les nombres de 0 à 9
for (int count=0; count<=9; count++) {
    System.out.println(count);
}
```



# Imbrication de boucles [1]

- On peut placer n'importe quelle(s) instruction(s) à l'intérieur d'une instruction d'itération `while`, `do` ou `for`, y compris une nouvelle instruction d'itération.
- Les instructions itératives peuvent donc être imbriquées.
- Une telle construction est très fréquente.

```
for (int i=1; i<=5; i++) {  
    System.out.println("i> " + i);  
    for (int k=0; k<=2 ; k++) {  
        System.out.println("  k> " + k);  
    }  
    System.out.println("-----");  
}
```



i> 1  
k> 0  
k> 1  
k> 2

i> 2  
k> 0  
k> 1  
k> 2

i> 3  
k> 0  
k> 1  
k> 2

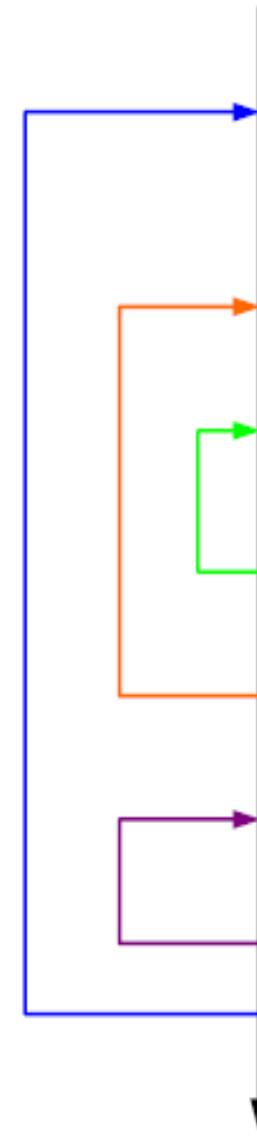
i> 4  
k> 0  
k> 1  
k> 2

i> 5  
k> 0  
k> 1  
k> 2

# Imbrication de boucles [2]

- Autre exemple d'imbrication de boucles :

```
for (int i=0; i<3; i++) {  
    System.out.println("i> " + i);  
    int j = 2*i;  
    while (j>0) {  
        System.out.println("  j> " + j);  
        for (int k=6; k>0 ; k=k-2) {  
            System.out.println("    k> " + k);  
        }  
        j = j/2;  
    }  
    int m = 4-i;  
    do {  
        System.out.println("  m> " + m);  
    } while (--m > 0);  
}
```



```
i> 0  
m> 4  
m> 3  
m> 2  
m> 1  
i> 1  
j> 2  
  k> 6  
  k> 4  
  k> 2  
j> 1  
  k> 6  
  k> 4  
  k> 2  
m> 3  
m> 2  
m> 1  
i> 2  
j> 4  
  k> 6  
  k> 4  
  k> 2  
j> 2  
  k> 6  
  k> 4  
  k> 2  
j> 1  
  k> 6  
  k> 4  
  k> 2  
m> 2  
m> 1
```

# Instructions étiquetées

- On peut **donner un nom** (étiquette, label) à une instruction en la précédant d'un identificateur et d'un caractère deux points (':') :

*nom : instruction ;*

- Les étiquettes ne sont utilisées qu'en relation avec les instructions d'interruption **break** et **continue** (voir pages suivantes).
- En Java, les étiquettes ne peuvent pas être utilisées pour effectuer un branchement direct (**goto label**).

Remarque : même si **goto** est un mot réservé du langage, il n'est pas utilisé actuellement.

# Instruction break

**break [ étiquette ] ;**

- L'instruction **break** force l'interpréteur à passer immédiatement à la fin de l'instruction englobante **switch**, **while**, **do** ou **for** la plus interne
- L'instruction **break** peut être suivie du nom d'une étiquette (label). Dans ce cas, elle a pour effet de quitter immédiatement l'instruction englobante (qui peut être quelconque dans ce cas) portant l'étiquette correspondante

```
testBreak:  
for (int i=1; i<=5; i++) {  
    for (int j=1; j<=5; j++) {  
        if (j == 3) break;                      // Termine la boucle interne  
        if (i == 4) break testBreak;             // Termine la boucle externe  
        System.out.println("i*j=" + i*j);  
    }  
}
```

# Instruction continue [1]

**continue** [ étiquette ] ;

- L'instruction **continue** force la boucle englobante la plus interne à démarrer une **nouvelle itération**
- L'instruction **continue** ne peut être utilisée qu'au sein d'une boucle **while**, **do** ou **for**
- Avec une étiquette, l'instruction **continue** force la boucle englobante identifiée par l'étiquette à démarrer une nouvelle itération
- Effets :
  - **Avec while et do** : évalue l'expression booléenne et, si elle est vraie, exécute une nouvelle fois le corps de la boucle
  - **Avec for** : effectue les instructions de conclusion puis évalue l'expression booléenne et, si elle est vraie, exécute une nouvelle fois le corps de la boucle

## Instruction continue [2]

- Un exemple d'utilisation de l'instruction **continue** :

```
testContinue:  
for (int i=1; i<=5; i++) {  
    for (int j=1; j<=5; j++) {  
        if (j == 3) continue; // Prochaine itération  
                            // de la boucle interne  
  
        if (i == 4) continue testContinue; // Prochaine itération  
                                         // de la boucle externe  
        System.out.println("i*j=" + i*j);  
    }  
}
```

# Instruction return

```
return [ expression ] ;
```

- L'instruction **return** termine l'exécution d'une méthode en cours d'exécution et rend le contrôle à la méthode appelante en retournant (restituant) éventuellement une valeur (expression) qui doit être compatible avec le type de la méthode.
- Si une méthode déclare un type de retour, l'instruction **return** (avec une expression compatible) doit obligatoirement figurer dans le corps de la méthode (dernière instruction exécutable).

```
public static double square(double x) {  
    return x * x;  
}
```



# Les Méthodes

# Pourquoi créer des méthodes

- La taille des problèmes à résoudre à l'aide d'un programme peut être très complexe, d'où la nécessité de :
  - **Subdiviser** les problèmes en sous-problèmes (plus simples) que l'on traite indépendamment : [ approche *Top/Down* ]
    - ➔ Abstraction, modularisation, raffinements successifs ➔
  - **Réutiliser** des parties de code existantes (exemple : entrée/sorties, fonctions mathématiques, graphiques, etc) : [ approche *Bottom/Up* ]
    - ➔ Rapidité de développement, efficacité, fiabilité ➔
- Une **méthode** permet de regrouper, sous un **nom**, une suite d'instructions **exécutable à répétition**.

# Une méthode c'est ...

---

- On peut dire, en quelques mots, qu'une **méthode**...
  - c'est le regroupement d'une **suite d'instructions** auquel on attribue un **nom** (identificateur)
  - peut être **invoquée** (appelée) par une instruction (dans une méthode appelante)
  - permet la déclaration (optionnelle) de **variables locales**
  - peut être optionnellement **paramétrée** au moyen d'une liste **d'arguments (paramètres)**
  - peut optionnellement **retourner une valeur** (un résultat) à l'appelant
  - peut être liée à l'existence d'un **objet** (méthode d'instance) ou à l'existence d'une **classe** (méthode statique)
  - peut optionnellement générer une **exception** qui indique qu'un événement exceptionnel s'est produit durant son exécution (par exemple une erreur)

# Méthodes [1]

---

- Une **méthode** est une **séquence nommée d'instructions**.
- Ces instructions peuvent être exécutées en **invoquant** (appelant) la méthode.
- Lorsqu'une méthode est invoquée, elle peut recevoir un certain nombre de valeurs appelées **arguments** (ou **paramètres**).
- Une méthode peut facultativement **retourner une valeur** d'un certain type (elle se comporte alors comme une expression complexe). On utilise également le terme **fondction** pour désigner une méthode qui retourne une valeur.
- Une invocation de méthode est une expression (avec effet de bord) qui peut également être utilisée en tant qu'instruction simple (sans exploitation d'une éventuelle de valeur de retour).
- Après l'exécution de la dernière instruction de la méthode, le contrôle retourne à l'instruction qui a invoqué la méthode.

## Méthodes [2]

---

- Une méthode est caractérisée par sa **signature** (ou **en-tête**) qui comprend :
  - le **nom de la méthode**
  - le type et le nom de chacun des **paramètres formels** (arguments)
  - le **type de la valeur renvoyée** par la méthode (ou sinon **void**)
  - les types des **exceptions** que la méthode peut générer
  - divers **modificateurs** de méthode qui fournissent des informations supplémentaires sur certaines propriétés de la méthode
- La signature d'une méthode définit tout ce qu'il est nécessaire de savoir pour l'utiliser (pour l'utiliser à bon escient, une description du comportement est naturellement indispensable).
- La signature constitue une partie importante de la spécification de la méthode appelée aussi **API** (*Application Programming Interface*)

# Méthodes [3]

Modificateurs

Type de retour de la méthode

Nom de la méthode

Liste des paramètres formels

Variable locale

`public static int max(int a, int b)`

```
int r = a;  
if (b > a) r = b;  
return r;
```

}

Valeur de retour  
(Expression)

En-tête de la méthode (signature)

Corps de la méthode

...

`int v = max(k, 3);`

...

Invocation (appel) de la méthode avec les paramètres effectifs

# Déclaration de méthode

*En\_tête\_de\_méthode { Corps\_de\_méthode }*

- L'**en-tête** de la méthode définit la **signature** de la méthode.
- Le **corps** de la méthode comprend les **instructions** qui seront exécutées lors de l'invocation de la méthode.

Remarque : Contrairement à d'autres langages (Ada, C++, ...) qui imposent ou permettent de placer dans des unités de compilation séparées les spécifications (en-têtes) et implémentations (corps), le langage Java groupe ces deux éléments dans une unique structure.

# En-tête de méthode

- La syntaxe pour déclarer l'en-tête de la méthode est la suivante :

```
[ modificateurs ] type nom ( liste_param ) [ throws exceptions ]
```

**modificateurs** : Un ou plusieurs mots-clés séparés par des espaces. Ils définissent certaines propriétés de la méthode (visibilité, etc). *Optionnel*  
[public, private, protected, final, static, native, abstract, synchronized]

**type** : Définit le type de retour de la méthode (fonction).  
Si la méthode ne retourne pas de valeur, le type doit être **void** (vide)

**nom** : Nom de la méthode  
(même règles que pour les identificateurs).

**liste\_param** : Liste de paramètres formels séparés par des virgules.  
Chaque paramètre est défini comme une variable locale (*type nom*).  
La liste de paramètres peut être vide (les parenthèses demeurent).

**exceptions** : Liste des types d'exceptions pouvant être générées par la méthode  
(les exceptions sont séparées par des virgules). *Optionnel*

# Corps de méthode

{ *Instructions* }

- Le **corps de la méthode** est constitué d'une séquence d'instructions délimitées par des accolades.
- Des **variables locales** peuvent être déclarées, leur portée sera limitée au corps de la méthode (à partir de la déclaration).
- Si la signature de la méthode déclare un **type de retour** (autre que **void**), le corps de la méthode doit comprendre une **instruction return** suivie d'une expression compatible avec le type de retour déclaré dans la signature.  
Cette expression définit la **valeur de retour** de la méthode (la valeur restituée).

# Exemples de méthodes [1]

```
//-----  
// Method without parameter  
// Prints a separator line  
//-----  
public static void separate() {  
    System.out.println("-----");  
}  
  
//-----  
// Returns the square of the parameter val  
//-----  
public static double square(double val) {  
    return val * val;  
}
```

## Exemples de méthodes [2]

```
//-----
// Returns the max value between v1 and v2
//-----
public static int max(int v1, int v2) {
    if (v1 > v2) return v1;
    else          return v2;
}

//-----
// Method that calls other methods
//-----
public static double fctA(double a, int b, int c) {
    int temp = max(b, c);
    return square(a) / temp;
}
```

# Méthode main

- La **méthode main()** joue un rôle particulier.
- C'est le **point d'entrée d'une application** Java (ce n'est pas le cas pour les applets qui doivent disposer des méthodes `init()` et `start()`).
- Elle est **invoquée par la machine virtuelle** lors du lancement de l'application. La méthode reçoit en argument un tableau de `String` contenant les éventuelles valeurs des paramètres de lancement (pour plus de détail consulter le slide intitulé *Arguments sur la ligne de commande* du chapitre consacré aux Entrées/Sorties).
- Sa **signature est imposée**.

```
//-----
// Main method (displays "Hello")
//-----
public static void main(String[] args) {
    System.out.println("Hello");
}
```

## Utilité des méthodes

---

- En Java, c'est pratiquement la seule **construction permettant d'exécuter** des instructions.
- Permet le **découpage** d'un problème en sous-problèmes plus simples.
- Améliore la **lisibilité** du code source (si le nom des méthodes est bien choisi, leur invocation donnera déjà au lecteur du code des informations précieuses sur le rôle de ces méthodes).
- Favorise le principe d'**abstraction** et d'**encapsulation** : l'utilisateur d'une méthode n'a pas à connaître tous les détails de l'implémentation; seule la spécification est nécessaire (notion de "*boîte noire*").
- Permet la **réutilisation** des instructions (évite de dupliquer le code avec tous les inconvénients que cela comporte).

# Emplacement des méthodes

- En Java, toutes les méthodes doivent être **déclarées au sein d'une classe**. Il n'y a pas de méthodes globales.
- Contrairement à d'autres langages, Java n'autorise pas de créer directement une méthode à l'intérieur d'une autre méthode. Il n'y a **pas d'imbrication de méthodes** (à moins de créer une classe locale).
- Dans l'ordre des lignes du code source, les méthodes peuvent être invoquées (appelées) avant d'avoir été déclarées.  
Il est ainsi possible à deux méthodes de s'appeler mutuellement.

```
.....
public static void f(int i) {
    System.out.println("f(" + i + ")");
    if (i>0) g(--i);
}
.....
public static void g(int j) {
    System.out.println("g(" + j + ")");
    f(j);
}
.....
```

# Invocation de méthode [1]

- L'**invocation** (l'appel) d'une méthode provoque l'exécution de ses instructions.
- Pour invoquer une méthode, on utilise son nom suivi de zéro, une ou plusieurs expressions séparées par des virgules et placées entre parenthèses.
- Les valeurs de ces expressions constituent les **arguments** (ou **paramètres effectifs**) de la méthode. Ils doivent correspondre en types (compatibilité) et en nombre à la liste définie dans la signature de la méthode.
- Syntaxe :

*Nom\_de\_la\_méthode ( expr1, expr2, ... )*

Remarque : Les expressions constituant les paramètres effectifs sont évaluées de gauche à droite. C'est important dans des expressions du genre :

*f(f(a, f(b, c)), f(d, e));*

## Invocation de méthode [2]

- Les méthodes définies **sans type de retour** (déclarées avec **void**) sont invoquées à la manière d'une instruction simple (donc suivies d'un point-virgule).

```
separate(); // Les parenthèses sont obligatoires
```

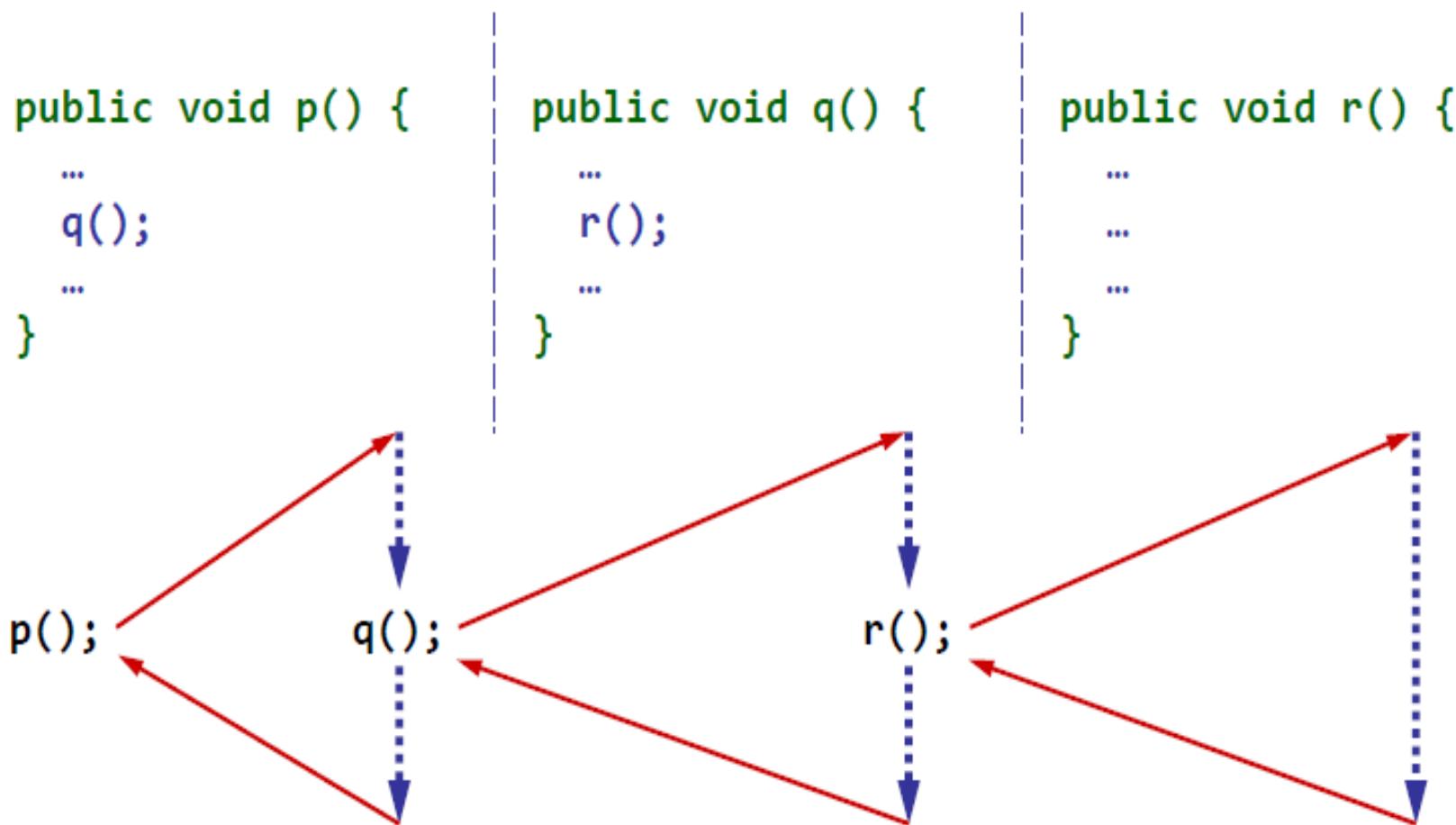
- Les méthodes définies **avec un type de retour** sont généralement invoquées dans des expressions (affectation, paramètre effectif de méthode, ...) mais peuvent également être invoquées comme des instructions simples (dans ce cas la valeur de retour n'est pas utilisée).

```
double PI = 3.1416;
double area, radius = 1.5;
int error;
String f = "config.ini";

area = PI * square(radius); // Invocation dans une expression
error = readfile(f); // Invocation dans une expression
readfile(f); // Invocation comme instruction simple
```

# Séquencement

- Lors de l'invocation de méthodes, une **pile d'appels** est constituée de manière à pouvoir revenir au point d'invocation après l'exécution de la méthode (chemin de retour).



## Passage de paramètres [1]

- **Lors de la définition** d'une méthode, on déclare une liste (éventuellement vide) de **paramètres formels** qui peuvent être considérés comme des variables locales dans le corps de la méthode.
- **Lors de l'invocation** d'une méthode, on doit transmettre, pour chaque paramètre formel, une expression dont le type est compatible avec celui qui est déclaré dans la signature de la méthode. Ces expressions constituent les **paramètres effectifs**.
- Le **passage des valeurs des paramètres** peut s'assimiler à une **assignation** entre les variables constituées par les paramètres formels et les expressions qui représentent les paramètres effectifs.
- Cette assignation a lieu à l'entrée de la méthode (avant l'exécution de la première instruction).  
Ce mécanisme est appelé "**Passage par valeur**" (ou également mode "**Copy In**").

## Passage de paramètres [2]

- La méthode agit donc sur des **copies locales** des valeurs des expressions transmises en paramètres lors de l'invocation.
- A la sortie de la méthode (après l'exécution de la dernière instruction), les variables contenant les copies locales ne sont pas réassignées en retour (**pas de "Copy Out"**).
- A chaque invocation de la méthode, de l'espace mémoire est alloué pour enregistrer les valeurs des paramètres effectifs (ainsi que pour les autres variables locales déclarées dans la méthode).  
Cet espace mémoire est libéré à la sortie de la méthode.
- Les variables contenant les copies locales des paramètres effectifs ainsi que les variables locales déclarées dans la méthode **ne peuvent donc pas être utilisées en dehors du corps de la méthode**.

# Passage de paramètres [3]

- Illustration du mécanisme de passage de paramètres lors d'un appel de fonction.
- Dans l'exemple, le paramètre effectif (**nb**) est copié dans la variable locale représentant le paramètre formel (**i**) de la méthode **triple()**.

```
public static void main(String[] args) {  
    int nb = 4;  
  
    int r = triple(nb); -----  
  
    System.out.println("> " + nb + ", " + r);  
}
```

nb 4      r 12

**i = nb;**

```
public static int triple(int i) { <-----  
  
    return 3*i;  
}
```

i 4 ----- nb 4  
i 4                    nb 4

Console

> 4, 12

## Passage de paramètres [4]

---

- Le mécanisme qui a été vu pour le passage de paramètres de types primitifs s'applique également si les paramètres passés sont de **types référence** (des objets ou des tableaux).
- Dans ce cas, on aura donc une copie locale de la référence mais pas des valeurs référencées (c'est efficace à l'exécution mais peut induire des effets de bord dont il faut être conscient).
- Le détail du passage de paramètres de type référence ainsi que les implications que peut induire ce mode de transfert seront décrits dans le chapitre suivant consacré aux tableaux.
- Il est possible de définir des méthodes comportant un nombre variable de paramètres. Comme cette syntaxe fait appel à des tableaux, elle sera décrite à la fin du chapitre suivant.

# Surcharge de méthodes (Overloading)

- Le langage Java permet de définir **plusieurs méthodes avec le même nom** pour autant que la **liste des paramètres** soit **différente**.
- Deux listes de paramètres sont différentes si :
  - le nombre de paramètres est différent *ou alors*
  - la liste ordonnée des types des paramètres est différente  
*(Remarque : la deuxième règle est suffisante)*
- On remarquera que :
  - le type de retour de la méthode n'entre pas en considération
  - le nom des paramètres formels non plus
- Exemples :

```
public static int triple(int i) {  
    return (i*3);  
}  
  
public static float triple(float f) {  
    return (f*3);  
}
```

## Surcharge de méthodes [2]

- La technique consistant à définir **plusieurs méthodes avec le même nom** s'appelle la **surcharge de méthode** (*Overloading*).
- Lors de l'invocation d'une méthode surchargée, c'est le compilateur qui détermine (sur la base du profil des paramètres) la méthode qui doit être appelée.

Conseil : *Dans un contexte donné, plusieurs méthodes ne doivent porter le même nom que si les opérations effectuées ont une forte similitude sémantique.*

Remarque : *Contrairement à d'autres langages (Ada, C++, ...), Java ne permet pas de surcharger les opérateurs (+, -, \*, /, ==, ...).*

# Réursivité [1]

- Une **méthode** est dite **récursive** si elle s'invoque elle-même.

```
public static long factorial(long x) {  
    if (x<0) { ... };          // Error if x is negative  
    if (x==0) return 1;  
    else      return x * factorial(x-1);  
}
```

- On parle de **réursivité indirecte** si la méthode  $m_1$ , en cours de déclaration invoque une autre méthode  $m_2$  qui, elle, invoque  $m_1$ , dans ses instructions (directement ou indirectement).

## Récursivité [2]

---

- Les méthodes récursives doivent toujours contenir une **instruction d'arrêt** (généralement sous la forme d'une instruction `if (...)`) correspondant à un **cas de base** qui termine la récursivité (sous peine d'épuiser les ressources du système, notamment la mémoire).
- L'algorithme doit d'autre part garantir (dans toutes les situations) une **convergence vers le cas de base** (c'est-à-dire se rapprocher de l'instruction d'arrêt de la récursivité).
- Certains algorithmes se prêtent particulièrement bien à l'écriture de code récursif (parcours d'arbres, de graphes, fonctions mathématiques récurrentes, ...).



# Les Tableaux

# Tableaux

- Un **tableau** est une séquence indexée d'éléments (valeurs) de **même type**.
- Il est caractérisé par sa **taille** (nombre d'éléments qu'il peut contenir) et par le **type de ses éléments**.
- Le **type tableau** est un type référence (comme les objets).
- Les **éléments d'un tableau** peuvent être des valeurs de types primitifs, des objets ou des tableaux.
- Il faut bien faire la distinction entre les deux étapes :
  - la **déclaration du tableau** (`[]`) qui crée une variable (référence) représentant un objet de type tableau
  - et la **création du tableau** (`new`) qui alloue l'espace mémoire pour enregistrer les éléments du tableau et initialise le contenu

tab

|   |   |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |

# Déclaration de tableaux [1]

*Type\_des\_eléments [] Identificateur ;*

- **Déclaration** de la variable identifiant l'objet de type tableau.
- La variable déclarée constitue une **référence** (sorte de pointeur vers une adresse mémoire) qui permet d'accéder au tableau.
- Le littéral **null** est une valeur particulière qui indique l'absence de référence (ou la référence vers rien).
- La taille du tableau n'est pas définie lors de sa déclaration mais seulement lors de sa création.

```
int[]    intArray;      // Déclaration d'un tableau de valeurs primitives int
Point[]  constellation; // Déclaration d'un tableau d'objets Point
byte[][] matrix;        // Déclaration d'un tableau de tableaux de byte
```

## Déclaration de tableaux [2]

*Type\_des\_eléments [ ] Identificateur = Initialisation ;*

- Une **expression d'initialisation** peut être ajoutée à la déclaration; dans ce cas, le tableau est créé et la taille est déterminée par le résultat de l'expression d'initialisation.
- L'expression d'initialisation doit retourner un objet de type tableau compatible avec le type de la variable que l'on déclare.
- L'initialisation peut consister en une expression de création de tableau (avec l'opérateur **new** [voir pages suivantes] ).
- Une syntaxe particulière permet de spécifier un **littéral tableau** comme expression d'initialisation : {expr1, expr2, expr3, ...} (cette syntaxe, sans new, n'est autorisée que lors de la déclaration)

```
char[] vowels= {'a','e','i','o','u','y'};
```

# Création de tableaux [1]

**new Type\_des\_eléments [ Taille ]**

- La **taille du tableau** (nombre d'éléments) est déterminée par l'expression entre crochets qui doit avoir une valeur entière positive (ou zéro); la taille ne doit pas obligatoirement être connue à la compilation.
- Une fois qu'un tableau est créé, sa **taille ne peut plus varier**.
- Les **éléments du tableau** sont **initialisés à leurs valeurs par défaut** (`false` pour les booléens, `0` pour tous les autres types primitifs, `null` pour les objets et les tableaux).
- L'opérateur **new retourne une référence** à l'objet tableau créé.

```
int[] topTen = new int[10];      // Tableau de 10 entiers (int)
String[] text = new String[50];   // Déclaration et création simultanées
byte[][] matrix = new byte[25][80];
Point[] constellation = new Point[astro.getDimension()];
```

## Création de tableaux [2]

```
new Type_des_eléments [] { élém1, élém2, ...}
```

- **Combinaison de la création** d'un tableau **avec l'initialisation** de ses éléments (qui doivent être compatibles avec le type déclaré).
- La **taille du tableau** n'est pas spécifiée explicitement. Elle est déterminée par le **nombre d'éléments énumérés** entre les accolades (agrégat).
- Cette notation permet de créer des **tableaux anonymes** (non affectés à une variable) qui peuvent, par exemple, être passés en paramètre lors de l'invocation d'une méthode.

```
// Exemple de tableau anonyme passé à la méthode askQuestion
```

```
String answer= askQuestion("Continuer?", new String[] {"Yes", "No"});
```

# Utilisation des tableaux [1]

- Chaque **élément d'un tableau** est assimilé à une variable à laquelle on peut accéder individuellement. Par exemple : `t[k]++`
- Chaque élément est identifié par un **indice entier** compris **entre 0 et (*n*-1)**, *n* étant la taille du tableau
- L'**accès aux éléments** d'un tableau s'effectue de la manière suivante :

*Nom\_tableau [ indice ]*

```
int[] tab;           // Déclaration d'un tableau d'entiers (int)
tab    = new int[3]; // Création du tableau avec 3 éléments
tab[0] = 7;          // Assignation du premier élément
tab[1] = 4;
tab[2] = tab[0] + tab[1]; // Assignation et accès aux éléments
```

## Utilisation des tableaux [2]

- Le **nombre d'éléments d'un tableau** peut être obtenu à l'aide de l'attribut **length** qui est prédéfini pour tous les objets de type tableau (accessible en lecture uniquement).

Exemple :    `nbElements = tab.length;`

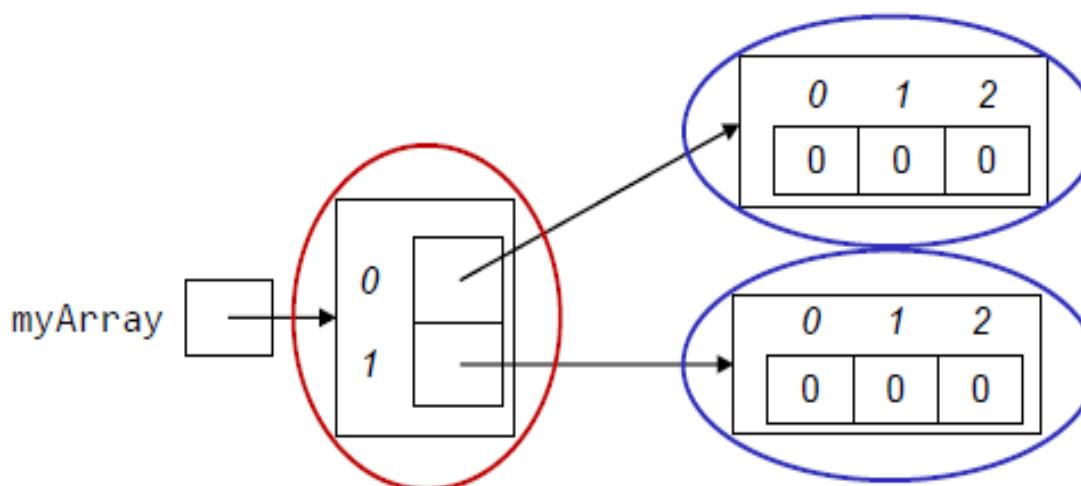
- Si, lors de l'accès à un élément d'un tableau, l'indice spécifié est négatif ou supérieur à l'indice du dernier élément (`length -1`), l'exception `ArrayIndexOutOfBoundsException` sera générée.
- L'expression qui détermine l'**indice** doit être de type `byte`, `short`, `char` ou `int` (le type `long` n'est pas autorisé).
- Le littéral `null` peut être utilisé pour représenter l'absence de tableau (comme pour tout autre objet)

Exemple :    `char[] word = null;`

# Tableaux multidimensionnels [1]

- Les tableaux peuvent avoir plusieurs dimensions. On parle alors de **tableaux multidimensionnels** (attention à ne pas confondre nombre de dimensions et taille du tableau).
- Un **tableau multidimensionnel** est un tableau de tableaux (de tableaux, de tableaux, ...).
- Chaque **paire de crochets ([ ])** représente une **dimension** (aussi bien pour la déclaration, la création que l'accès aux éléments)

```
int[][] myArray = new int[2][3]; // Éléments initialisés à 0
```



# Tableaux multidimensionnels [1]

- Exemple d'utilisation

```
//-----
// Multiplication table 0x0 ... 9x9
//-----
int[][] multTable = new int[10][10];      // Éléments initialisés à 0
for (int i=0; i<multTable.length; i++) {
    for (int j=0; j<multTable[i].length; j++) {
        multTable[i][j] = i*j;
    }
}
```

## Tableaux multidimensionnels [2]

- L'attribut **length** peut être consulté pour chacune des dimensions

```
char[][] crossword = new char[5][10];
int nbRows = crossword.length;           // nbRows=5
int nbCols = crossword[0].length;        // nbCols=10
```

- Lors de la création d'un tableau multidimensionnel, il n'est **pas obligatoire de définir la taille de toutes les dimensions** (cependant, la première dimension doit obligatoirement être définie).
- Par contre, si l'on définit les tailles, il faut le faire dans l'ordre, **de gauche à droite**.

```
// Ok, déclaration de la première dimension
float[][][] colorPalette = new float[10][][];

// Ok, déclaration des deux premières dimensions
float[][][] colorPalette = new float[10][20][];

// Erreur à la compilation !
float[][][] colorPalette = new float[][20][];
```

## Tableaux multidimensionnels [3]

- Pour les **tableaux multidimensionnels**, les **valeurs littérales** (initialiseurs) sont constituées par l'**emboîtement d'accolades** contenant les éléments de chacune des dimensions (reflet de la structure : tableau de tableaux).

```
//-----
// Tableau de 7 éléments dont chacun représente
// un tableau de 5 éléments
//-----

int[][] multTable = { {0, 0, 0, 0, 0},
                      {0, 1, 2, 3, 4},
                      {0, 2, 4, 6, 8},
                      {0, 3, 6, 9, 12},
                      {0, 4, 8, 12, 16},
                      {0, 5, 10, 15, 20},
                      {0, 6, 12, 18, 24} };
```

# Tableaux multidimensionnels [4]

- Les **tableaux multidimensionnels** étant des tableaux de tableaux, ils ne doivent **pas obligatoirement être rectangulaires** (ou cubiques ou ...).

```
//-----
// Tableau de 7 éléments dont chacun représente
// un tableau de taille variable
// (tableau triangulaire)
//-----
```

```
int[][] multTable = { {0},
                      {0,  1},
                      {0,  2,  4},
                      {0,  3,  6,  9},
                      {0,  4,  8, 12, 16},
                      {0,  5, 10, 15, 20, 25},
                      {0,  6, 12, 18, 24, 30, 36} };
```

# Tableaux multidimensionnels [5]

- Les **éléments tableaux** des tableaux multidimensionnels peuvent être **définis dynamiquement** y compris leur taille si elle n'a pas été déterminée lors de la déclaration.

```
//-----
// Tableau de 100 éléments représentant chacun un
// tableau de taille variable (créé dynamiquement).
// Chaque élément du tableau est une chaîne de caractères.
//-----

String[][] s = new String[100][];
for (int i=0; i<s.length; i++) {
    s[i] = new String[(i%16)+1]; // Création dynamique du sous-tableau
    for (int j=0; j<s[i].length; j++) {
        s[i][j] = i + "/" + j;
    }
}
```

# Représentation en mémoire [1]

- Chaque variable est associée à l'adresse d'une case mémoire
- Pour les **types primitifs**, la case mémoire contient la **valeur**
- Pour les **types référence**, la case mémoire contient une **référence** à la zone mémoire contenant l'objet ou le tableau
- Les illustrations qui suivent sont schématiques et servent à mettre en lumière le principe de fonctionnement  
(l'implémentation réelle peut être sensiblement différente)

## Variable de type primitif

|                   | Variable | Adresse | Contenu |
|-------------------|----------|---------|---------|
|                   |          | 1215    |         |
|                   |          | 1216    |         |
| int number = 122; | number   | 1217    | 122     |
|                   |          | 1218    |         |
|                   |          | 1219    |         |

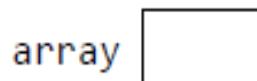
Notation abstraite

number 122

# Notation abstraite [1]

## Déclaration de tableau

```
int[] array;
```



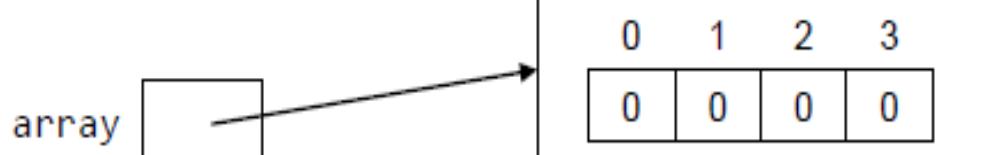
```
array= null;
```



La valeur null représente une référence vers rien

## Création de tableau

```
array= new int[4];
```

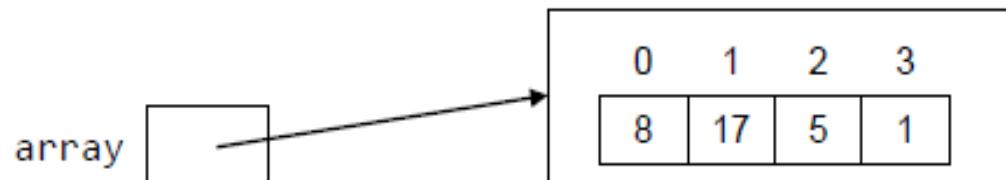


Initialisation automatique des éléments à leurs valeurs par défaut

# Notation abstraite [2]

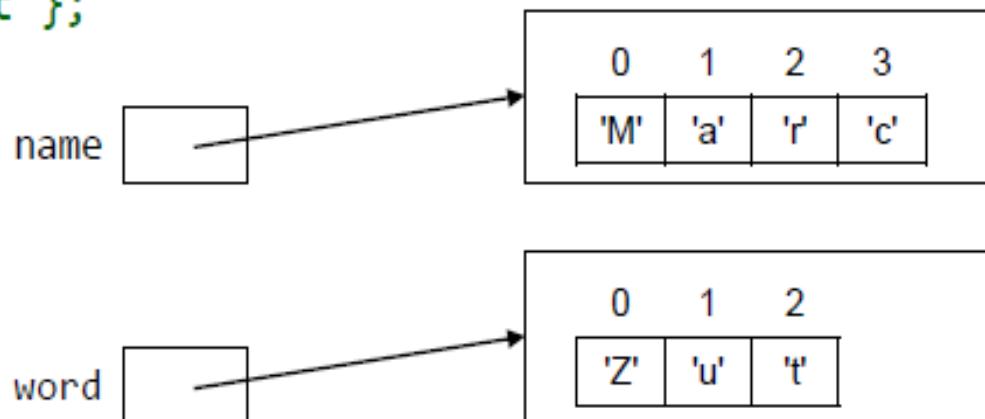
## Affectation des éléments du tableau

```
array[0] = 8;  
array[1] = 17;  
array[2] = 5;  
array[3] = 1;
```



## Déclaration et affectation combinées

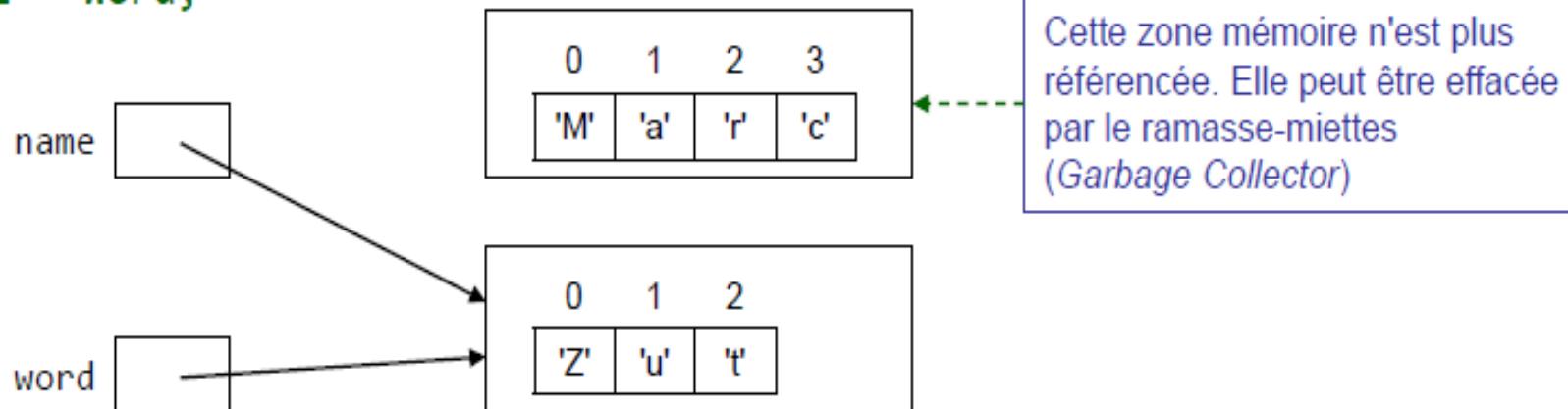
```
char[] name= {'M','a','r','c'};  
char[] word= {'Z', 'u', 't'};
```



# Notation abstraite [3]

## Affectation de tableaux

```
name = word;
```



## Attention aux effets d'alias (couplage entre variables)

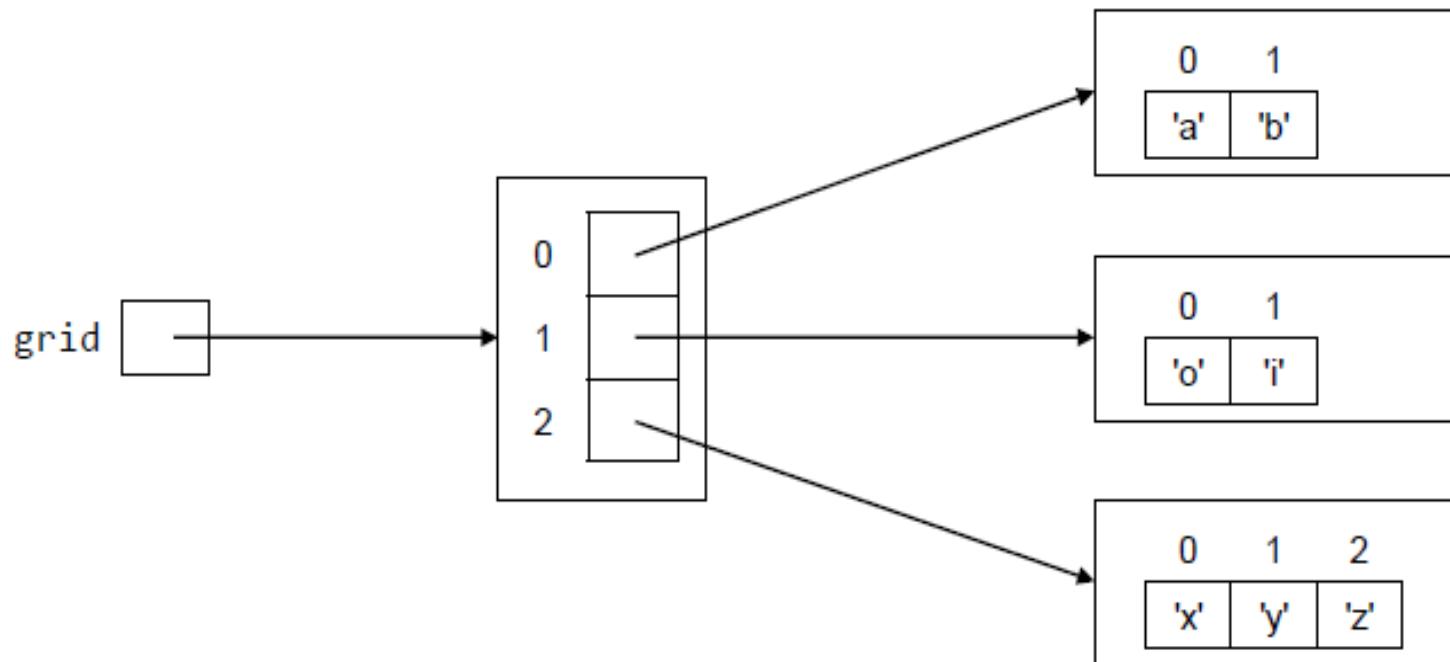
```
System.out.println(name[1]); // 'u'  
word[1] = 'o';  
System.out.println(name[1]); // 'o'
```

En modifiant un élément de `word`, on modifie également le contenu de `name`!

# Notation abstraite [3]

## Tableaux multidimensionnels

```
char[][] grid = { {'a', 'b'},  
                  {'o', 'i'},  
                  {'x', 'y', 'z'} };
```



Par extension, le même mécanisme s'applique aux tableaux comportant plus de deux dimensions

## Référence / Objet référencé [1]

---

- Une **référence** peut être **assimilée** à un **pointeur vers une adresse en mémoire** (mais Java ne connaît pas de réels pointeurs).
- Les références sont **opaques** (contenu invisible) et **ne peuvent pas être manipulées** par le programmeur.
- De ce fait, les détails du mécanisme sous-jacent (détails d'implémentation) ne sont pas très importants car ils n'affectent pas le principe général de fonctionnement.
- Avec les types référence, il faut toujours **bien distinguer** si l'on traite la **référence** (**adresse**) ou l'**objet référencé** (**contenu**). Cette distinction est **très importante**, notamment avec les opérateurs d'égalité (`==`), d'inégalité (`!=`) et d'affectation (`=`) ainsi que lors des passages en paramètre (invocation de méthodes).
- Ces remarques s'appliquent à tous les types référence, c'est-à-dire aux tableaux et aux objets (que nous verrons plus tard).

## Référence / Objet référencé [2]

- Comparaison et assignation de tableau et de contenu.

```
char[] c1 = {'a', 'b', 'c'};  
char[] c2 = {'a', 'b', 'c'};  
char[] c3 = c1;  
  
if (c1 == c2) System.out.println("c1 égal à c2");  
if (c1 == c3) System.out.println("c1 égal à c3");  
if (c2 == c3) System.out.println("c2 égal à c3");  
  
if (c1[0] == c2[0]) System.out.println("c1[0] == c2[0]");  
if (c1[0] == c3[0]) System.out.println("c1[0] == c3[0]");  
if (c2[0] == c3[0]) System.out.println("c2[0] == c3[0]");  
  
c3[0] = 'z';  
if (c1[0] == c3[0]) System.out.println("c1[0] == c3[0]");  
if (c2[0] == c3[0]) System.out.println("c2[0] == c3[0]");
```

## Passage de tableaux en paramètre [1]

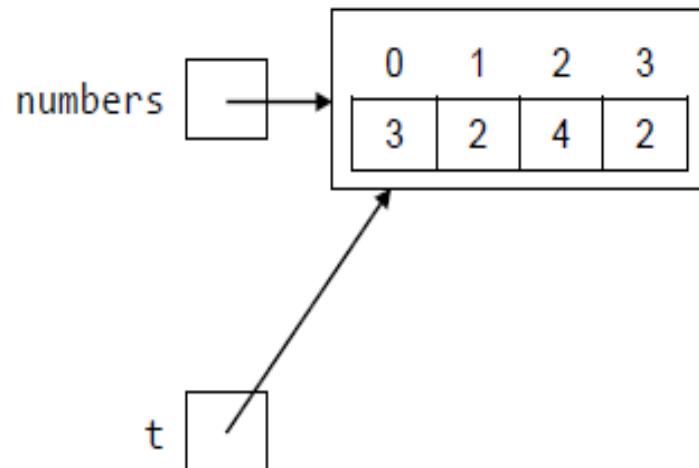
- Le mécanisme de passage de paramètres qui a été vu dans le chapitre consacré aux méthodes s'applique également aux **types référence** et notamment aux tableaux.
- Le passage par valeur de la référence (avec copie locale de cette dernière) a cependant des implications qu'il est important de bien comprendre.
- Dans ce cas, la méthode peut accéder et modifier les valeurs référencées car c'est la référence à l'objet qui est assignée à la copie locale (paramètre effectif) lors de l'invocation.
- On a donc une copie locale de la référence mais pas des valeurs référencées (c'est efficace à l'exécution mais peut induire des effets de bord non souhaités).

Remarque : Une méthode qui modifie l'état de certains objets ou tableaux passés en paramètres doit clairement l'indiquer dans sa spécification (description).

# Passage de tableaux en paramètre [2]

- Dans l'exemple ci-dessous, la méthode `max()` reçoit, au moment de son l'appel, une copie de la référence du tableau `numbers` (paramètre effectif) dans la variable `t` (paramètre formel).

```
public static void main(String[] args) {  
    int[] numbers = {3, 2, 4, 2};  
    int nMax = max(numbers);  
    System.out.println("Greatest: " + nMax);  
}  
  
public static int max(int[] t) {  
    int max = t[0];  
    for (int i=0; i<t.length; i++) {  
        if (t[i]>max) max = t[i];  
    }  
    return max;  
}
```



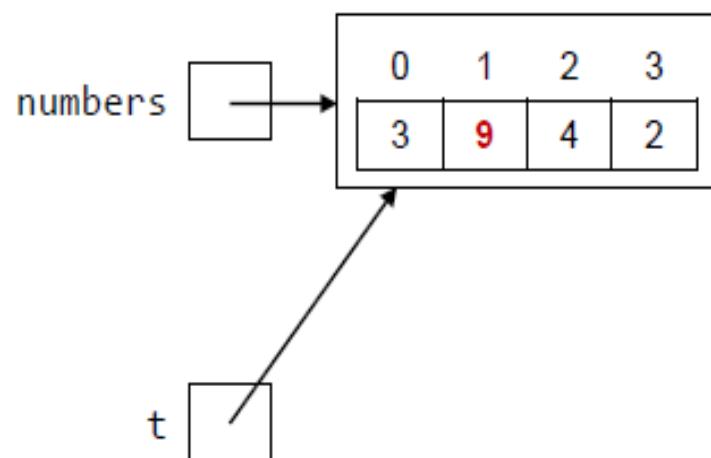
Console

Greatest: 4

# Passage de tableaux en paramètre [3]

- Le mécanisme a pour conséquence que la méthode `max()` peut, si elle le souhaite, modifier le contenu du tableau `t` et donc celui de `numbers` car c'est la même zone mémoire (alias sur les références) !

```
public static void main(String[] args) {  
    int[] numbers = {3, 2, 4, 2};  
  
    int nMax = max(numbers);  
  
    System.out.println("Greatest: " + nMax);  
}  
  
  
public static int max(int[] t) {  
    t[1] = 9;          // Caution: side effect !  
    int max = t[0];  
    for (int i=0; i<t.length; i++) {  
        if (t[i]>max) max = t[i];  
    }  
    return max;  
}
```



Console

Greatest: 9

# Méthodes avec paramètre variable [1]

- Il est possible de définir des méthodes comportant un **nombre variable de paramètres**.
- Le mécanisme de passage des paramètres variables repose sur une mise en tableau automatique. Il s'agit seulement d'une simplification de la syntaxe de déclaration et d'invocation de la méthode.
- Syntaxe de déclaration d'un paramètre variable :

*Type... NomDuParamètre*

- Exemple :

```
public static void dispErrors(int cause, String... errText)
```

- Une méthode ne peut comporter qu'**un seul paramètre variable** et il doit être **le dernier** dans la liste des paramètres formels.
- Dans l'exemple donné, les paramètres seront transmis dans un tableau de chaînes de caractères **String[]** (le mécanisme est simplement masqué).

## Méthodes avec paramètre variable [2]

- Dans le corps de la méthode, les valeurs du paramètre variable sont lues comme les éléments d'un tableau (selon le type déclaré).
- Exemple :

```
public static void dispErrors(int cause, String... errText) {  
    ...  
    for (int k=0; k<errText.length; k++) {  
        System.out.println(errText[k]);  
    }  
    ...  
}
```

- Cette méthode `dispErrors()` pourra être invoquée avec un nombre variable d'arguments :

```
dispErrors(2, "Erreur fatale");  
  
dispErrors(7, "Erreur de lecture", "Le fichier n'existe pas");  
  
dispErrors(5, "Erreur", "Accès bloqué", "Compte expiré");  
  
dispErrors(0);
```

## Boucle for revisitée (for-each) [1]

- A partir de la version 1.5 du langage, une syntaxe complémentaire a été introduite pour l'instruction **for**. Elle permet de parcourir tous les éléments d'un tableau ou d'une collection de données (itérateur) en utilisant une syntaxe allégée :

```
for ( type elem : tableau ) instruction
```

- Cette forme de l'instruction **for** est parfois appelée **for-each**
- Exemple :

```
boolean[] message = new boolean[200];
...
for (boolean elem : message) {
    if (elem) System.out.println("-");
    else      System.out.println(".");
}
...
```

## Boucle for revisitée (for-each) [2]

- En utilisant cette nouvelle syntaxe, l'exemple donné précédemment pour illustrer les méthodes avec paramètres variables pourrait être codé ainsi :

```
public static void dispErrors(int cause, String... errText) {  
    ...  
    for (String text : errText) {  
        System.out.println(text);  
    }  
    ...  
}
```

- Le symbole ":" se lit "*in*" ("*dans*").
- Cette nouvelle syntaxe ne remplace pas la syntaxe de base (elle la simplifie dans certaines situations particulières).
- Chaque fois que l'on doit accéder à l'itérateur (par exemple l'indice du tableau) dans le corps de la boucle cette syntaxe ne peut pas être utilisée.

# Méthodes utilitaires

- La classe **Arrays** (du package `java.util`) définit une série de méthodes statiques utilitaires permettant de manipuler des tableaux :
  - Assignation d'une valeur à certains éléments d'un tableau ... `fill()`
  - Test d'égalité du contenu de deux tableaux ..... `equals()`
  - Tri du contenu d'un tableau ..... `sort()`
  - Conversion d'un tableau en liste (`List`) ..... `asList()`
  - Recherche binaire dans un tableau trié ..... `binarySearch()`
- La classe **System** (du package `java.lang`) contient une méthode statique `arraycopy()` permettant de copier les éléments spécifiés d'un tableau dans un autre tableau, à une position donnée (attention : copie au premier niveau seulement [*Shallow copy*] ).  
Le second tableau doit être du même type que le premier. Il peut même s'agir du même tableau.
- La méthode `clone()` de la classe **Object** peut également être utilisée pour copier le contenu d'un tableau (*Shallow copy*) :

```
int[] b = (int[])a.clone();
```



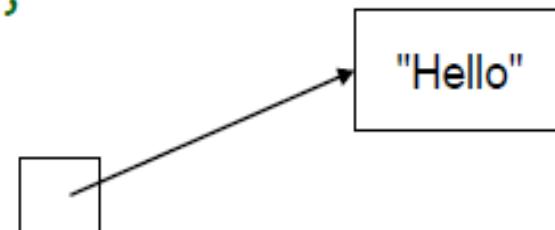
# Les Chaines de caractères

# Chaînes de caractères (String)

- En Java les chaînes de caractères sont représentées par des objets de type **String** (une classe prédefinie).
- Les chaînes de caractères ne font donc pas partie des types primitifs mais sont des types référence.
- Une syntaxe particulière est utilisée pour définir des littéraux de type **String** : on entoure le texte avec des guillemets ("...").
- On peut insérer des séquences d'échappement (identiques à celles définies pour le type **char**) dans les littéraux de type **String**.
- La déclaration et création d'une variable de type **String** s'effectue de la manière suivante :

```
String someText = "Hello";
```

someTexte



# Comparaison de chaînes

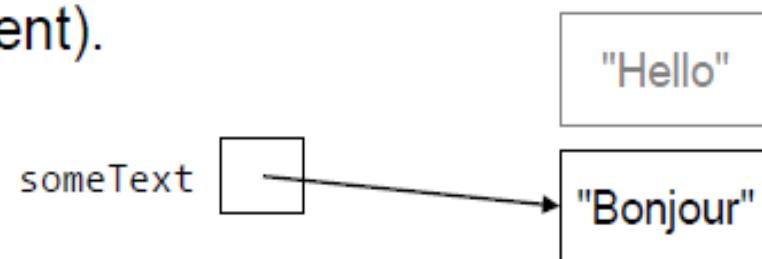
- Comme pour tous les types référence, l'opérateur "**==**" compare les références et non pas les objets référencés (les chaînes de caractères).
- La méthode **equals()** permet, elle, de comparer deux chaînes de caractères (objets référencés) et retourne **true** si elles sont égales.
- La méthode **compareTo()** remplace les opérateurs "plus petit", "égal", "plus grand" en utilisant l'ordre lexicographique (ordre du dictionnaire) et en retournant respectivement une valeur (**int**) négative, 0 ou positive.

```
String myString = "abc";
String otherStr = "abcdef";
if (myString == otherStr)
    System.out.println("Same string object");
if (myString.equals(otherStr))
    System.out.println("Same contents");
if (myString.compareTo(otherStr) < 0)
    System.out.println(myString + " smaller than " + otherStr);
```

# Objets String immuables

- Les objets de type **String** sont **immuables** : une fois créés ils ne peuvent plus être modifiés. Naturellement, la variable de type référence peut changer de valeur, et pointer vers une nouvelle chaîne (créeée à un autre emplacement).

```
String someText = "Hello";  
someText = "Bonjour";
```



- Les objets immuables ont l'avantage de pouvoir être partagés sans risque et l'on évite les problèmes d'alias ( $\Rightarrow$  optimisations possibles).
- Par contre, si l'on manipule fréquemment des objets de type **String** (à l'intérieur de boucles par exemple) cela peut conduire à la création d'un nombre considérable d'objets (temporaires) avec un coût non négligeable (ressources mémoires et temps d'exécution).
- Il est préférable dans ce cas de déclarer et utiliser des objets de type **StringBuffer** ou **StringBuilder** qui sont eux modifiables (voir pages suivantes).

# Opérations sur les chaînes [1]

- L'opérateur "+" permet de concaténer (mettre bout à bout) des chaînes de caractères.
- Si un des deux opérandes de l'opérateur "+" est un **String**, l'autre opérande sera, si nécessaire, automatiquement converti en un objet de type **String** (par invocation de la méthode **toString()**).

```
String myText = "Hello";
String      s1 = myText + " Mark";
String      rec = "Hello" + s1 + 3 + ')';
```

**Remarque :** Les objets de type **String** étant immuables, la concaténation est implémentée en créant un objet temporaire de type **StringBuffer** qui est utilisé ensuite pour créer un nouvel objet de type **String**.

C'est donc une opération assez lourde.

## Opérations sur les chaînes [2]

- Différentes méthodes permettent de convertir des valeurs de types primitifs en objets de type **String** et inversement (voir la table qui figure à la fin du premier chapitre du document de cours).
- Ces méthodes se trouvent soit dans la classe **String**, soit dans les classes *Wrapper* associées aux types primitifs (**Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, **Character**, **Boolean**).
- Dans les librairies de base, il existe souvent plusieurs méthodes pour effectuer un même type de conversion.

|   |          |
|---|----------|
| int i = Integer.parseInt("43");         | → 43     |
| double d = Double.parseDouble("3.14");  | → 3.14   |
| String s = String.valueOf('Z');         | → "Z"    |
| boolean b = (i > 10);                   | → true   |
| String t = (new Boolean(b)).toString(); | → "true" |
| String t = Boolean.toString(b); // Idem | → "true" |

## Opérations sur les chaînes [3]

- La classe **String** dispose d'un grand nombre de méthodes pour traiter les chaînes de caractères. Les exemples qui suivent n'en sont qu'une illustration sommaire.
- La méthode **length()** permet de connaître la longueur d'une chaîne de caractères (nombre de caractères).
- La méthode **charAt()** permet de retrouver le caractère situé à la position  $n$  d'une chaîne ( $0 \leq n \leq \text{length}() - 1$ ).
- La méthode **substring()** retourne une sous-chaîne d'une chaîne donnée. Les paramètres déterminent la position initiale et la position finale (+1) de la sous-chaîne dans la chaîne originelle.

```
String name = "James Bond";
int    i = name.length();           // i = 10
char   c = name.charAt(1);         // c = 'a'
String s = name.substring(2, 4);   // s = "me"
String t = "Yes".toUpperCase();   // t = "YES"
```

# Opérations sur les chaînes [4]

- La méthode **split()** permet de découper une chaîne de caractères sur la base d'un séparateur qui est exprimé sous la forme d'une *expression régulière (regex)*. La méthode retourne le résultat dans un tableau de **String** qui contient les fragments découpés.
- La syntaxe à utiliser pour les expressions régulières est définie dans la classe **Pattern (java.util.regex)**.

```
public static void main(String[] args) {  
    String path = "usr/lib/ruby/test.rb";  
    String[] elems;  
  
    String sepPattern1 = "/";      // Slash separator  
    elems = path.split(sepPattern1);  elems → { "usr", "lib", "ruby", "test.rb" }  
  
    String sepPattern2 = "/|\\.>"; // Slash or dot separator  
    elems = path.split(sepPattern2);  elems → { "usr", "lib", "ruby", "test", "rb" }  
}
```

# Formatage de chaînes [1]

- A partir de la version 1.5 du langage, une méthode de formatage assez puissante a été introduite dans la classe **String**.
- Cette méthode statique nommée **format()** retourne une chaîne de caractères formatée et prend en paramètre une chaîne de formatage (qui doit respecter une certaine syntaxe), ainsi qu'un nombre variable d'arguments (de type **Object** ou de types primitifs qui seront automatiquement convertis par le nouveau mécanisme d'*autoboxing*).
- Exemple :

```
String s = String.format("%.2f", v);
```

dans ce cas, si **v** = 1.56789f, alors **s** = "1.57"

- La syntaxe de la chaîne de formatage est relativement complexe et sa description se trouve dans l'API de la classe **Formatter** (du package **java.util**).

## Formatage de chaînes [2]

- Cette nouvelle méthode de formatage est largement inspirée de celle de la méthode `printf()` que l'on trouve dans le langage C (elle n'est pas totalement équivalente mais très fortement compatible).
- Une méthode `printf()` a d'ailleurs été ajoutée dans la classe `PrintStream` et peut donc être invoquée sur les flux de sortie `System.out` et `System.err`.
- Exemples :

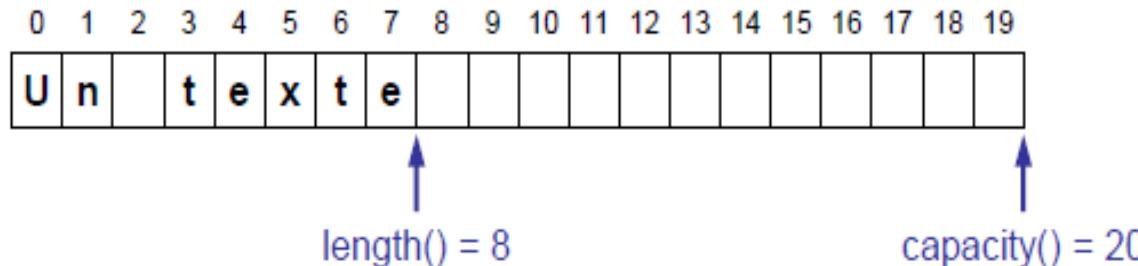
```
int i = 250;
float v = 1.23456f;

System.out.printf("v with 3 digits %.3f %n", v);
System.out.printf("Hex/Octal %1$x/%1$o %n", i);
System.out.printf("%5d%8.3f %n", i, v);
```

|                       |
|-----------------------|
| v with 3 digits 1.235 |
| Hex/Octal fa/372      |
| 250 1.235             |

# Classe StringBuffer [1]

- Contrairement à la classe **String**, la classe **StringBuffer** permet de créer des chaînes de caractères modifiables (la taille et le contenu peuvent varier durant l'exécution de l'application).
- La notion de **capacité (Capacity)** représente la taille du *buffer* interne qui mémorise la chaîne de caractères (nombre total de caractères disponibles avant de devoir agrandir la taille du buffer interne).
- La capacité est automatiquement augmentée lorsque c'est nécessaire. La capacité initiale peut être définie lors de la création d'un objet **StringBuffer** (paramètre du constructeur, 16 par défaut).
- Ne pas confondre **capacité** et **longueur** d'un objet **StringBuffer**.



# Classe StringBuffer [2]

- Pour créer un objet de type **StringBuffer**, on doit utiliser l'opérateur **new(...)** (pas de syntaxe simplifiée).

```
StringBuffer someText = new StringBuffer("Hello");
```

- Création en définissant une capacité initiale :

```
// Création d'une chaîne vide avec une capacité de 2000 caractères
StringBuffer sb = new StringBuffer(2000);

// Ajout d'une chaîne littérale (String)
sb.append("Bonjour");

System.out.println(sb.length());          //    7
System.out.println(sb.capacity());        // 2000

sb.append(" à tous !");

System.out.println(sb.length());          //   16
System.out.println(sb.capacity());
System.out.println(sb.capacity());        // 2000
```

# Opérations sur les StringBuffer

- Différentes méthodes permettent de manipuler les objets de type **StringBuffer** :

|                             |   |
|-----------------------------|---|
| <code>append()</code>       | Ajoute un élément à la fin de la chaîne de caractères (surcharge pour différents types d'éléments)              |
| <code>insert()</code>       | Insère un élément à une position donnée de la chaîne de caractères (surcharge pour différents types d'éléments) |
| <code>replace()</code>      | Remplace une partie d'une chaîne de caractères par une autre  |
| <code>delete()</code>       | Efface une portion donnée d'une chaîne de caractères  |
| <code>setCharAt()</code>    | Remplace un caractère à une position donnée   |
| <code>deleteCharAt()</code> | Efface (supprime) un caractère à une position donnée  |
| <code>substring()</code>    | Extrait une sous-chaîne   |
| <code>toString()</code>     | Convertit le contenu en type <b>String</b>  |

# Conversions String ⇔ StringBuffer

- Pour convertir un objet de type **String** en un objet de type **StringBuffer** on utilise le constructeur (opérateur **new(...)**).

```
String      s1    = "Hello";
StringBuffer sBuf = new StringBuffer(s1);
```

- Pour convertir un objet de type **StringBuffer** en un objet de type **String** on utilise la méthode **toString(...)** ou **substring(...)**.

```
StringBuffer vText = new StringBuffer();
vText.append("Un texte");

String s2 = vText.toString();
String s3 = vText.substring(3, 6); // s3 <-- "tex"
```

# Classe **StringBuilder**

---

- Dans la librairie standard de la plateforme Java, on trouve également la classe **StringBuilder**.
- Son API est compatible avec celle de la classe **StringBuffer** qu'elle peut remplacer dans toutes les situations où le code s'exécute dans un seul *thread* (c'est-à-dire dans tous les cas où il n'est pas indispensable que le code soit *thread-safe*).
- La plupart des méthodes de la classe **StringBuilder** s'exécutent plus rapidement que celles de la classe **StringBuffer**.
- Toutes les indications données dans les pages précédentes pour la classe **StringBuffer** s'appliquent également à la classe **StringBuilder**.



**FIN DE LA PARTIE I**