# **Java** : fondamentaux

**Achref El Mouelhi**

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

# Plan

# Plan

# Plan

# Java

## Une variable ?

- Pointeur vers une zone mémoire
- Permettant de stocker une ou plusieurs données
- Pouvant avoir plusieurs valeurs différentes dans un programme

# Java

## Une variable ?

- Pointeur vers une zone mémoire
- Permettant de stocker une ou plusieurs données
- Pouvant avoir plusieurs valeurs différentes dans un programme

## **Java** : langage de programmation fortement typé

- Il faut préciser le type de chaque variable
- Une variable peut avoir de valeurs différentes mais ne peut changer de type.

# Java

## Deux modes de typage en **Java**

- Statique : type précisé à la déclaration de la variable
- Dynamique : variable déclarée avec le mot-clé `var` $\Rightarrow$ type déterminé dynamiquement à partir de la valeur initiale. [disponible depuis **Java 11**]

# Java

## Nom de variables en **Java**

- Utiliser le **CamelCase**
- Premier caractère : lettre (ni `_` ni `$`)
- Caractère espace non autorisé
- Underscore `_` non autorisé comme nom de variable depuis **Java 9**
- Privilégier les noms en relation avec le contexte
- Ne pas choisir un mot-clé du langage
- Ne pas choisir un ancien mot-clé du langage comme `goto` ou `const` (liste de mots-clés : `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html`)
- Sensible à la casse

# Java

## Déclarer une variable

```
type nomVariable;
```

# Java

### Déclarer une variable

```
type nomVariable;
```



## Deux choix possibles pour typer les variables

- Types simples (primitifs) : commencent par une minuscule et ne peuvent pas contenir la valeur `null`.
- Types objets : commencent par une majuscule (respectant le **Pascal case**)

# Java

## Principaux types primitifs en **Java**

- `byte` : entier codé sur 1 octet (entre -128 et 127)
- `short` : entier codé sur 2 octets (entre -32 768 et 32 767)
- `int` : entier codé sur 4 octets (entre -2 147 483 648 et 2 147 483 647)
- `long` : entier codé sur 8 octets (entre -9 223 372 036 854 775 808 et +9 223 372 036 854 775 807)
- `float` : nombre à virgule codé sur 4 octets
- `double` : nombre à virgule codé sur 8 octets
- `boolean` : soit `true` soit `false` (1 octet)
- `char` : caractère codé sur 2 octet situé entre deux `'`

# Java

## Principaux types primitifs en **Java**

- `byte` : entier codé sur 1 octet (entre -128 et 127)
- `short` : entier codé sur 2 octets (entre -32 768 et 32 767)
- `int` : entier codé sur 4 octets (entre -2 147 483 648 et 2 147 483 647)
- `long` : entier codé sur 8 octets (entre -9 223 372 036 854 775 808 et +9 223 372 036 854 775 807)
- `float` : nombre à virgule codé sur 4 octets
- `double` : nombre à virgule codé sur 8 octets
- `boolean` : soit `true` soit `false` (1 octet)
- `char` : caractère codé sur 2 octet situé entre deux `'`

Pas de type primitif pour les chaînes de caractère.

# Java

## Exemple

```
int x;
```

# Java

## Exemple

```
int x;
```

## Déclarer et initialiser une variable

```
int x = 5;
```

# Java

## Exemple

```
int x;
```

## Déclarer et initialiser une variable

```
int x = 5;
```



## Ceci est une erreur

```
byte x = 130;
```

# Java

**Depuis Java 7, on peut déclarer un entier avec des séparateurs de chiffres (underscore _) pour améliorer la lisibilité des grands nombres.**

```
int x = 1_000_000;
```

# Java

**Depuis Java 7, on peut déclarer un entier avec des séparateurs de chiffres (underscore _) pour améliorer la lisibilité des grands nombres.**

```
int x = 1_000_000;
```

## Remarques

- Les underscores ne changent pas la valeur du nombre : ils sont purement décoratifs pour améliorer la lisibilité.
- Ils peuvent être utilisés entre les chiffres, mais pas au début, à la fin ou à côté d'un point décimal.

# Java

**Les déclarations suivantes sont valides**

```
int x = 1_000_000;
long y = 2_147_483_647L;
double z = 3.14_159_265;
```

# Java

**Les déclarations suivantes sont valides**

```
int x = 1_000_000;
long y = 2_147_483_647L;
double z = 3.14_159_265;
```

**Cependant, les suivantes sont invalides**

```
int x = _1000;
// Erreur : ne peut pas commencer par un underscore.

int y = 1000_;
// Erreur : ne peut pas terminer par un underscore.

double z = 3._14159;
// Erreur : ne peut pas précéder ou suivre un point.
```

# Java

## Déclarer une variable avec le mot-clé `var`

```
var nomVariable = valeurInitiale;
```

# Java

## Déclarer une variable avec le mot-clé `var`

```
var nomVariable = valeurInitiale;
```



### Remarque

Une variable déclarée avec le mot-clé `var`

- doit être initialisée
- a un type déterminé en fonction de sa valeur initiale
- ne peut changer de type

# Java

## Exemple

```
var x = 10;
// le type est par défaut int
```

# Java

**Exemple**

```
var x = 10;
// le type est par défaut int
```

**Pour le type** `long`

```
var z = 10l;
```

# Java

**Exemple**

```
var x = 10;
// le type est par défaut int
```

**Pour le type** `long`

```
var z = 10l;
```

**Pour le type** `double`

```
var y = 10.0;
```

# Java

**Exemple**

```
var x = 10;
// le type est par défaut int
```

**Pour le type** `long`

```
var z = 10l;
```

**Pour le type** `double`

```
var y = 10.0;
```

**Pour le type** `float`

```
var y = 10.0f;
```

# Java

**Pour le type** `short`

```
var y = (short) 10;
```

# Java

**Pour le type** `short`

```
var y = (short) 10;
```

**Pour le type** `byte`

```
var y = (byte) 10;
```

# Java

**Pour le type** `short`

```
var y = (short) 10;
```

**Pour le type** `byte`

```
var y = (byte) 10;
```

**Ceci génère une erreur**

```
var x = 2;
x = "bonjour";
```

# Java

## Pour les variables numériques (`int`, `float`...)

- `=` : affectation
- `+` : addition
- `-` : soustraction
- `*` : multiplication
- `/` : division
- `%` : reste de la division

# Java

## Quelques raccourcis

- `i = i + 1` ⇒ `i++;`
- `i = i - 1` ⇒ `i--;`
- `i = i + 2` ⇒ `i += 2;`
- `i = i - 2` ⇒ `i -= 2;`
- ...

# Java

## Exemple de post-incrémentation

```
int i = 2;
int j = i++;
System.out.println(i); // affiche 3
System.out.println(j); // affiche 2
```

# Java

## Exemple de post-incrémentation

```
int i = 2;
int j = i++;
System.out.println(i); // affiche 3
System.out.println(j); // affiche 2
```

## Exemple de pre-incrémentation

```
int i = 2;
int j = ++i;
System.out.println(i); // affiche 3
System.out.println(j); // affiche 3
```

# Java

**Pour convertir le contenu d'une variable (le `cast` pour les types compatibles)**

```
int x = 100;
byte z = (byte) x;
System.out.println(z); // affiche 100
```

# Java

**Pour convertir le contenu d'une variable (le `cast` pour les types compatibles)**

```
int x = 100;
byte z = (byte) x;
System.out.println(z); // affiche 100
```

**Attention aux valeurs qui dépassent l'intervalle**

```
int x = 200;
byte z = (byte) x;
System.out.println(z); // affiche -56
```

# Java

## Les emballages (wrapper) de types primitifs (types objets) en **Java**

- `Byte` pour `byte`
- `Short` pour `short`
- `Long` pour `long`
- `Integer` pour `int`
- `Float` pour `float`
- `Double` pour `double`
- `Boolean` pour `boolean`
- `Character` pour `char`

# Java

## Les emballages (wrapper) de types primitifs (types objets) en **Java**

- `Byte` pour `byte`
- `Short` pour `short`
- `Long` pour `long`
- `Integer` pour `int`
- `Float` pour `float`
- `Double` pour `double`
- `Boolean` pour `boolean`
- `Character` pour `char`

Il existe aussi plusieurs autres types objets : `String`, `Date`...

# Java

## Pourquoi les types primitifs et les types objets ?

- Les types primitifs sont moins coûteux en mémoire
- Les types objets offrent plusieurs méthodes à appliquer sur les valeurs : conversion, nombre de caractère...
- Toutes les classes wrappers contiennent une méthode (`TypeObjet.parseTypeSimple(string)`) qui permet de convertir une chaine de caractère en type primitif de cette classe

# Java

**Pas besoin de convertir d'un type simple vers son wrapper (ou inversement)**

```
Integer j = 2;
int k = j; // unboxing

System.out.println(k);
// affiche 2

int l = 3;
Integer m = l; // autoboxing

System.out.println(m);
// affiche 3
```

# Java

**Pas besoin de convertir d'un type simple vers son wrapper (ou inversement)**

```
Integer j = 2;
int k = j; // unboxing

System.out.println(k);
// affiche 2

int l = 3;
Integer m = l; // autoboxing

System.out.println(m);
// affiche 3
```

**Lorsqu'une classe wrapper est `null`, une tentative d'opération peut provoquer une `NullPointerException`**

```
Integer number = null;
int result = number + 5;
```

# Java

## Pour connaître la valeur max autorisée par un type numérique

```
System.out.println(Integer.MAX_VALUE);
// affiche 2147483647
```

# Java

## Pour connaître la valeur max autorisée par un type numérique

```
System.out.println(Integer.MAX_VALUE);
// affiche 2147483647
```

## La valeur min

```
System.out.println(Integer.MIN_VALUE);
// affiche -2147483648
```

# Java

## Pour connaître la valeur max autorisée par un type numérique

```
System.out.println(Integer.MAX_VALUE);
// affiche 2147483647
```

## La valeur min

```
System.out.println(Integer.MIN_VALUE);
// affiche -2147483648
```

## Le nombre d'octets

```
System.out.println(Integer.SIZE);
// affiche 32
```

# Java

## Exemple d'utilisation d'une chaîne de caractère

```
String string = new String();
string = "bonjour";
System.out.println(string); // affiche bonjour
```

# Java

### Exemple d'utilisation d'une chaîne de caractère

```
String string = new String();
string = "bonjour";
System.out.println(string); // affiche bonjour
```

### On aussi

```
String string = new String("bonjour");
System.out.println(string); // affiche bonjour
```

# Java

### Exemple d'utilisation d'une chaîne de caractère

```
String string = new String();
string = "bonjour";
System.out.println(string); // affiche bonjour
```

### On aussi

```
String string = new String("bonjour");
System.out.println(string); // affiche bonjour
```

### Ou encore

```
String string = "bonjour";
System.out.println(string);  // affiche bonjour
```

# Java

### Exemple d'utilisation d'une chaîne de caractère

```
String string = new String();
string = "bonjour";
System.out.println(string); // affiche bonjour
```

### On aussi

```
String string = new String("bonjour");
System.out.println(string); // affiche bonjour
```

### Ou encore

```
String string = "bonjour";
System.out.println(string);  // affiche bonjour
```

Une chaîne de caractère doit être située entre deux `"contenu"`

# Java

**L'opérateur + pour concaténer deux chaînes de caractère**

```
String string = "bon";
String string2 = "jour";
System.out.println(string + string2);
// affiche bonjour
```

# Java

## Quelques méthodes de la classe `String`

- `length()` : retourne le nombre de caractère de la chaîne.
- `indexOf(x)` : retourne l'indice de la première occurrence de la valeur de `x` dans la chaîne, -1 sinon.
- `contains()` : retourne `true` si la chaîne contient `x`, `false` sinon.
- `charAt(i)` : retourne le caractère d'indice `i` dans la chaîne.
- `substring(i, j)` : permet d'extraire une sous-chaîne de la chaîne à partir de l'indice `i` jusqu'au l'indice `j - 1`
- `equals(str)` : permet de comparer la chaîne à `str` et retourne `true` en cas d'égalité, `false` sinon.
- `replace(old, new)` : permet de remplacer toute occurrence de la chaîne `old` dans la chaîne courante par `new` et retourne la nouvelle chaîne
- ...

# Java

**Pour convertir une chaîne de caractère en majuscule**

```
System.out.println("bonjour".toUpperCase());
// affiche BONJOUR
```

# Java

**Pour convertir une chaîne de caractère en majuscule**

```
System.out.println("bonjour".toUpperCase());
// affiche BONJOUR
```

**Pour convertir un caractère en majuscule**

```
System.out.println(Character.toUpperCase('a'));
// affiche A
```

# Java

**Exemple :** `replace(old, new)`

```
String string = "bonjour les bons jours";
String string2 = string.replace("jour", "soir");
System.out.println(string2);
// bonsoir les bons soirs
```

# Java

**Exemple :** `replace(old, new)`

```
String string = "bonjour les bons jours";
String string2 = string.replace("jour", "soir");
System.out.println(string2);
// bonsoir les bons soirs
```

**Exemple :** `indexOf(str, fromIndex)`

```
String string = "bonjour les bons jours";
int pos = string.indexOf("bon", 5);
System.out.println(pos);
// affiche 12
```

# Java

**Exemple :** `replace(old, new)`

```
String string = "bonjour les bons jours";
String string2 = string.replace("jour", "soir");
System.out.println(string2);
// bonsoir les bons soirs
```

**Exemple :** `indexOf(str, fromIndex)`

```
String string = "bonjour les bons jours";
int pos = string.indexOf("bon", 5);
System.out.println(pos);
// affiche 12
```

**Exemple :** `split(regex)`

```
String string = "bonjour les bons jours";
System.out.println(Arrays.toString(string.split(" ")));
// affiche [bonjour, les, bons, jours]
```

# Java

## Ne jamais comparer les String avec ==

```
String s1 = "Wick";
String s2 = new String("Wick");
String s3 = "Wick";
String s4 = new String("Wick");

System.out.println(s1 == s2);
// affiche false

System.out.println(s1 == s3);
// affiche true

System.out.println(s2 == s4);
// affiche false
```

# Java

## Explication

- Les chaînes littérales ()créées sans l'opérateur `new`) sont optimisées dans le pool de chaînes, ce qui signifie que si une chaîne littérale avec la même valeur existe déjà, **Java** réutilisera la référence à cette chaîne.
- Les chaînes créées avec l'opérateur `new` sont créées explicitement même si une chaîne avec le même contenu existe déjà dans le pool de chaînes.

# Java

## Exemple de conversion d'une chaîne de caractère

```
String string = "2";
byte z = Byte.parseByte(string);
System.out.println(z); // affiche 2
```

# Java

### Exemple de conversion d'une chaîne de caractère

```
String string = "2";
byte z = Byte.parseByte(string);
System.out.println(z); // affiche 2
```

### Ceci déclenche une exception (arrêt d'exécution)

```
String string = "a";
byte z = Byte.parseByte(string);
System.out.println(z);
```

### Le résultat

```
Exception in thread "main" java.lang.NumberFormatException: For
    input string: "a"
```

# Java

**Exemple de conversion de `Integer` en `String`**

```
Integer x = 2;
String string = x.toString();
System.out.println(string); // affiche 2
```

# Java

## Quatre méthodes pour convertir `int` en `String`

```
int y = 3;
String chaine =((Integer) y).toString();
System.out.println(chaine); // affiche 3

int z = 4;
String str = Integer.toString(z);
System.out.println(str); // affiche 4

int t = 5;
String s = String.valueOf(t);
System.out.println(s); // affiche 5

int u = 6;
String s = "" + u;
System.out.println(s); // affiche 6
```

# Java

**Attention, le `cast` entre deux types incompatibles n'est pas autorisé**

```
int z = 4;
String str =(String) z;
System.out.println(str);
```

# Java

## Autres types textuels

- `CharSequence` : interface [en lecture seule] utilisée souvent pour typer les paramètres d'une méthode
- `StringBuilder` : classe permettant de construire des chaînes de caractères muables (mutables) [implémente `CharSequence`]
- `String` : classe permettant de construire des chapines de caractères immuables (mutables) [implémente `CharSequence`]
- `StringTokenizer` : classe permettant de décomposer une chaîne de caractère en tokens selon le·s séparateur·s choisi·s (plus rapide que `String.split()`) [n'implémente pas `CharSequence`]
- ...

# Java

## StringBuilder vs String

```
StringBuilder chars = new StringBuilder("Wick");
System.out.println(System.identityHashCode(chars));
// affiche 1510467688

chars.append("y");
System.out.println(System.identityHashCode(chars));
// affiche 1510467688

String string = "Wick";
System.out.println(System.identityHashCode(string));
// affiche 1995265320

string += "y";
System.out.println(System.identityHashCode(string));
// affiche 746292446

System.out.println(string.equals(chars));
// affiche false
```

# Java

## On peut aussi utiliser les méthodes statiques de la classe `Math`

- `Math.abs(x)` : retourne la valeur absolue de `x`
- `Math.pow(x, y)` : retourne la `x` puissance `y`
- `Math.max(x, y)` : retourne le max de `x` et `y`
- `Math.min(x, y)` : retourne le min de `x` et `y`
- `Math.sqrt(x)` : retourne la racine carré de `x`
- `Math.floor(x)`, `Math.ceil(x)` et `Math.round(x)` : retournent l'arrondi de `x`
- `Math.random()` : retourne une valeur aléatoire entre 0 et 1
- ...

# Java

**Exemple avec** `Math.floor(x)`, `Math.ceil(x)` **et** `Math.round(x)`

```
System.out.println(Math.round(1.9)); // affiche 2
System.out.println(Math.round(1.5)); // affiche 2
System.out.println(Math.round(1.4)); // affiche 1
System.out.println(Math.ceil(1.9)); // affiche 2
System.out.println(Math.ceil(1.5)); // affiche 2
System.out.println(Math.ceil(1.4)); // affiche 2
System.out.println(Math.floor(1.9)); // affiche 1
System.out.println(Math.floor(1.5)); // affiche 1
System.out.println(Math.floor(1.4)); // affiche 1
```

# Java

**Depuis Java 1.5, on peut importer statiquement la classe**

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

# Java

**Depuis Java 1.5, on peut importer statiquement la classe**

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

**Et ensuite**

```
System.out.println(round(1.9)); // affiche 2
System.out.println(round(1.5)); // affiche 2
System.out.println(round(1.4)); // affiche 1
System.out.println(ceil(1.9)); // affiche 2
System.out.println(ceil(1.5)); // affiche 2
System.out.println(ceil(1.4)); // affiche 2
System.out.println(floor(1.9)); // affiche 1
System.out.println(floor(1.5)); // affiche 1
System.out.println(floor(1.4)); // affiche 1
```

# Java

## Remarques

- En **Java**, certaines opérations arithmétiques peuvent lever des exceptions.
- D'autres retournent des valeurs spéciales comme `NaN` ou `Infinity`.

# Java

## Exceptions arithmétiques en **Java**

Une exception est levée uniquement pour la division entière par zéro

- `int a = 5 / 0;` ArithmeticException
- `long b = 5L / 0L;` ArithmeticException

Les flottants ne lèvent pas d'exception

- `double c = 5.0 / 0.0;` ⇒ `Infinity`

# Java

## NaN en **Java**

NaN (Not-a-Number) est retourné dans les cas suivants

- `0.0 / 0.0` ⇒ `NaN`
- `Math.sqrt(-1)` ⇒ `NaN`
- `Math.log(-1)` ⇒ `NaN`

Propriétés de `NaN`

- `NaN == NaN` est **toujours false**
- `Double.isNaN(x)` permet de tester si `x` est `NaN`

# Java

## Infinity en **Java**

Les opérations qui dépassent les limites des flottants retournent `Infinity`

- `5.0 / 0.0` ⇒ `Infinity`
- `-5.0 / 0.0` ⇒ `-Infinity`
- `Double.MAX_VALUE * 2` ⇒ `Infinity`

# Java

## Récapitulatif

| Opération | Résultat |
|---|---|
| `5 / 0` (int) | ArithmeticException |
| `5.0 / 0.0` (double) | `Infinity` |
| `0.0 / 0.0` | `NaN` |
| `Math.sqrt(-1)` | `NaN` |

# Java

## Pour lire une valeur saisie par l'utilisateur dans la console

- il faut utiliser la classe `Scanner`
- il faut préciser le type de la valeur à récupérer

# Java

## Instancier la classe `Scanner`

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

# Java

## Instancier la classe `Scanner`

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

## La classe `Scanner` est signalée en rouge, il faut l'importer

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

# Java

## Instancier la classe `Scanner`

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

## La classe `Scanner` est signalée en rouge, il faut l'importer

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

## Exemple de lecture d'un entier

```
int i = scanner.nextInt();
```

# Java

## Instancier la classe `Scanner`

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

## La classe `Scanner` est signalée en rouge, il faut l'importer

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

## Exemple de lecture d'un entier

```
int i = scanner.nextInt();
```

## Fermer le `scanner`

```
scanner.close();
```

# Java

## Remarques

- Pour lire une chaîne de caractère, il faut utiliser la méthode `next()` ou `nextLine()`
- Pour lire un nombre réel, il faut utiliser la méthode `nextFloat()`
- ...

# Java

## Exemple avec `next()`

```
Scanner scanner = new Scanner(System.in);
// si on saisit "bonjour tout le monde"
String string = scanner.next();
System.out.println(string);
// affiche bonjour
```

# Java

## Exemple avec `next()`

```
Scanner scanner = new Scanner(System.in);
// si on saisit "bonjour tout le monde"
String string = scanner.next();
System.out.println(string);
// affiche bonjour
```

## Exemple avec `nextLine()`

```
Scanner scanner = new Scanner(System.in);
// si on saisit "bonjour tout le monde"
String string = scanner.nextLine();
System.out.println(string);
// affiche bonjour tout le monde
```

# Java

**À chaque appel à `next()`, on récupère le token suivant**

```
Scanner scanner = new Scanner(System.in);
// si on saisit "bonjour tout le monde"
String string = scanner.next();
String s = scanner.next();
System.out.println(s);
// affiche tout
```

# Java

**À chaque appel à `next()`, on récupère le token suivant**

```
Scanner scanner = new Scanner(System.in);
// si on saisit "bonjour tout le monde"
String string = scanner.next();
String s = scanner.next();
System.out.println(s);
// affiche tout
```

**Le code suivant déclenche une exception car on ne peut affecter la chaîne `"tout"` à un entier**

```
Scanner scanner = new Scanner(System.in);
// si on saisit "bonjour tout le monde"
String string = scanner.next();
int v = scanner.nextInt();
System.out.println(v);
```

# Java

**Il n'existe pas une méthode spécifiques aux caractères, mais on peut faire**

```
Scanner scanner = new Scanner(System.in);
String string = scanner.next();
char c = scanner.next().charAt(0);
System.out.println(c);
```

# Java

## Exécuter si une condition est vraie

```
if (condition) {
    ...
}
```

# Java

## Exécuter si une condition est vraie

```
if (condition) {
    ...
}
```

## Exemple

```
var x = 3;
if (x >= 0) {
   System.out.println(x + " est positif");
}
```

# Java

## Exécuter si une condition est vraie

```
if (condition) {
    ...
}
```

## Exemple

```
var x = 3;
if (x >= 0) {
   System.out.println(x + " est positif");
}
```

**Pour les conditions, on utilise les opérateurs de comparaison**

# Java

## Opérateurs de comparaison

- == : pour tester l'égalité
- != : pour tester l'inégalité
- > : supérieur à
- < : inférieur à
- >= : supérieur ou égal à
- <= : inférieur ou égal à

# Java

## Opérateurs de comparaison

- == : pour tester l'égalité
- != : pour tester l'inégalité
- > : supérieur à
- < : inférieur à
- >= : supérieur ou égal à
- <= : inférieur ou égal à

En **Java**, on ne peut comparer deux valeurs de type incompatible.

# Java

## Exercice

Écrire un code **Java** qui demande à l'utilisateur de saisir un entier positif et qui affiche ensuite sa parité (sans `else`).

# Java

**Exécuter un premier bloc si une condition est vraie, un deuxième sinon (le bloc `else`)**

```
if (condition1) {
   ...
}
else {
   ...
}
```

# Java

**Exécuter un premier bloc si une condition est vraie, un deuxième sinon (le bloc `else`)**

```
if (condition1) {
   ...
}
else {
   ...
}
```

**Exemple**

```
var x = 3;
if (x > 0)
{
  System.out.println(x + " est strictement positif");
}
else
{
  System.out.println(x + " est négatif ou nul");
}
```

# Java

## Exercice

Écrire un programme **Java** qui permet de déterminer si une chaîne de caractères saisie par l'utilisateur contient un nombre pair ou impair de caractères.

# Java

**On peut enchaîner les conditions avec `else if` (et avoir un troisième bloc voire ... un nième)**

```
if (condition1)
{
    ...
}
else if (condition2)
{
    ...
}
...
else
{
    ...
}
```

# Java

## Exemple

```
var x = -3;
if (x > 0)
{
  System.out.println(x + " est strictement positif");
}
else if (x < 0)
{
  System.out.println(x + " est strictement négatif");
}
else
{
  System.out.println(x + " est nul");
}
```

# Java

## Exercice 1

Écrire un code **Java** qui

- demande à l'utilisateur de saisir deux entiers `a` et `b` et un caractère `op` (pouvant avoir comme valeurs `+`, `-`, `*` ou `/`)
- affiche le résultat de l'application de `op` sur `a` et `b`. Par exemple, si `a = 5`, `b = 2` et `op = *` alors le résultat est `10`.

# Java

## Exercice 2

Écrire un code **Java** qui

- demande à l'utilisateur de saisir trois entiers `a`, `b` et `c`
- affiche le résultat de l'équation $\texttt{a}x^2 \texttt{ + b}x \texttt{ + c = 0}$.

# Java

## Opérateurs logiques

- `&&` : et
- `||` : ou
- `!` : non
- `^` : ou exclusif

# Java

## Opérateurs logiques

- `&&` : et
- `||` : ou
- `!` : non
- `^` : ou exclusif



**Tester plusieurs conditions (en utilisant des opérateurs logiques)**

```
if (condition1 && !condition2 || condition3){
    ...
}
[else ...]
```

# Java

## Exercice 1

Écrire un code **Java** qui

- demande à l'utilisateur de saisir une année (un entier),
- affiche si l'année saisie est bissextile (voir `https://fr.wikipedia.org/wiki/Ann%C3%A9e_bissextile`).

# Java

## Exercice 2

Écrire un code **Java** qui

- demande à l'utilisateur de saisir deux entiers `a` et `b` différents de zéro,
- affiche le signe du résultat de la multiplication sans calculer le produit.

# Java

## Exercice 3

Écrire un code **Java** qui

- demande à l'utilisateur de saisir deux entiers `a` et `b`,
- détermine et affiche si le résultat de l'addition (sans calculer la somme) est pair ou impair.

# Java

## Structure conditionnelle `switch` : syntaxe

```
switch (nomVariable) {
  case constante-1:
    groupe-instructions-1;
    break;
  case constante-2:
    groupe-instructions-2;
    break;
  ...
  case constante-N:
    groupe-instructions-N;
    break;
  default:
    groupe-instructions-par-défaut;
}
```

# Java

## Remarques

- Le `switch` permet **seulement** de tester l'égalité
- Le `break` permet de quitter le `switch` une fois un bloc case exécuté
- Il est possible de regrouper plusieurs `case`
- Le bloc `default` est facultatif, il sera exécuté si la valeur de la variable ne correspond à aucune constante de `case`

# Java

## Structure conditionnelle avec `switch`

```
int x = 5;
switch (x) {
    case 1:
      System.out.println("un");
      break;
    case 2:
      System.out.println("deux");
      break;
    case 3:
      System.out.println("trois");
      break;
    default:
      System.out.println("autre");
}
```

# Java

## La variable dans `switch` peut être

- de types simples : `int`, `short`, `byte` et `char`
- de type wrappers correspondants : `Integer`, `Short`, `Byte` et `Character`
- de type énumération depuis **Java 6**
- de type chaîne de caractère depuis **Java 7**

# Java

## Et si on supprime un `break`

```
String x = "2";
switch (x) {
    case "1":
      System.out.println("un");
      break;
    case "2":
      System.out.println("deux");
    case "3":
      System.out.println("trois");
      break;
    default:
      System.out.println("autre");
}
// affiche deux trois
```

# Java

## Un multi-case pour un seul traitement [Java 7]

```
var x = 5;
switch (x) {
  case 1:
  case 2:
    System.out.println("un ou deux");
    break;
  case 3:
    System.out.println("trois");
    break;
  case 4:
  case 5:
    System.out.println("quatre ou cinq");
    break;
  default:
    System.out.println("autre");
}
```

# Java

## Un multi-case simplifié [depuis Java 14]

```
var x = 5;
switch (x) {
  case 1, 2:
    System.out.println("un ou deux");
    break;
  case 3:
    System.out.println("trois");
    break;
  case 4, 5:
    System.out.println("quatre ou cinq");
    break;
  default:
    System.out.println("autre");
}
```

# Java

**On peut simplifier l'écriture encore plus avec une expression lambda (->) [depuis Java 14]**

```
var x = 5;
switch (x) {
    case 1, 2 -> System.out.println("un ou deux");
    case 3 -> System.out.println("trois");
    case 4, 5 -> System.out.println("quatre ou cinq");
    default -> System.out.println("autre");
}
```

# Java

**Si on a plusieurs instructions, on ajoute { } [Java 14]**

```
var x = 5;
switch (x) {
    case 1, 2 -> {
        System.out.println("un ou");
        System.out.println("deux");
    }
    case 3 -> System.out.println("trois");
    case 4, 5 -> {
        System.out.println("quatre ou");
        System.out.println("cinq");
    }
    default -> System.out.println("autre");
}
```

# Java

## `switch` peut retourner une valeur [depuis Java 14]

```
var x = 5;
String result = switch (x) {
    case 1, 2 -> {
        var tmp = "un ou deux";
        yield tmp;
    }
    case 3 -> "trois";
    case 4, 5 -> {
        var tmp = "quatre ou cinq";
        yield tmp;
    }
    default -> "autre";
};
System.out.println(result);
```

# Java

## Remarques

Les `case` d'un même `switch` peuvent retourner des valeurs ayant un type différent.

# Java

## Exercice

Écrire un programme qui demande à l'utilisateur de saisir l'indice d'un mois (entier compris entre 1 et 12) et qui retourne le nombre de jours de ce mois

- si l'entier est égal à 1, 3, 5, 7, 8, 10 ou 12 le programme affiche 31
- sinon si l'entier est égal à 4, 6, 9 ou 11 le programme affiche 30
- sinon si l'entier est égal à 2, le programme demande à l'utilisateur de saisir l'année et lui retourne 29 si l'année est bissextile, 28 sinon (voir `https://fr.wikipedia.org/wiki/Ann%C3%A9e_bissextile`)
- pour toute autre valeur, le programme affiche une erreur

# Java

## Simplifions l'écriture avec l'expression ternaire

```
int x = 2;
String type = (x % 2 == 0) ? "pair" : "impair";
System.out.println(type); // affiche pair
```

# Java

**Simplifions l'écriture avec l'expression ternaire**

```
int x = 2;
String type = (x % 2 == 0) ? "pair" : "impair";
System.out.println(type); // affiche pair
```

**En Java, les deux dernières parties d'une expression ternaire peuvent retourner deux types différents**

```
int x = 2;
var type = (x % 2 == 0) ? true : "impair";
System.out.println(type); // affiche true
```

# Java

**Boucle `while` : à chaque itération on teste si la condition est vraie avant d'accéder aux traitements**

```
while (condition[s]) {
   ...
}
```

# Java

**Boucle `while` : à chaque itération on teste si la condition est vraie avant d'accéder aux traitements**

```
while (condition[s]) {
   ...
}
```



Attention aux boucles infinies, vérifier que la condition d'arrêt sera bien atteinte après un certain nombre d'itérations.

# Java

## Exemple

```
var i = 0;
while (i < 5) {
   System.out.println(i);
   i++;
}
```

# Java

## Exemple

```
var i = 0;
while (i < 5) {
   System.out.println(i);
   i++;
}
```

## Le résultat est

```
0
1
2
3
4
```

# Java

## Exercice 1

Écrire un programme qui permet d'afficher les nombres pairs inférieurs à 10.

# Java

## Exercice 2

Écrire un code programme qui

- demande à l'utilisateur de saisir deux entiers `a` et `b` (avec `a < b`),
- afficher les nombres pairs compris entre `a` et `b`.

# Java

## Exercice 3

Écrire un programme qui demande à l'utilisateur de saisir un entier positif `n` et qui calcule puis affiche la somme de tous les entiers positifs inférieurs à `n`.

# Java

**La Boucle `do` ... `while` exécute le bloc au moins une fois ensuite elle vérifie la condition**

```
do {
   ...
}
while (condition[s]);
```

# Java

**La Boucle `do` ... `while` exécute le bloc au moins une fois ensuite elle vérifie la condition**

```
do {
   ...
}
while (condition[s]);
```



Attention aux boucles infinies, vérifier que la condition d'arrêt sera bien atteinte après un certain nombre d'itérations.

# Java

## Exemple

```
var i = 0;
do {
   System.out.println(i);
   i++;
} while (i < 5);
```

# Java

## Exemple

```
var i = 0;
do {
   System.out.println(i);
   i++;
} while (i < 5);
```

## Le résultat est

```
0
1
2
3
4
```

# Java

## Exercice

Écrire un programme qui demande en boucle à l'utilisateur de saisir un entier jusqu'à l'obtention de 0 puis affiche la somme de tous ces entiers saisis.

# Java

## Boucle `for`

```
for (initialisation; condition[s]; incrémentation) {
   ...
}
```

# Java

## Boucle `for`

```
for (initialisation; condition[s]; incrémentation) {
   ...
}
```



Attention aux boucles infinies si vous modifiez la valeur du compteur à l'intérieur de la boucle.

# Java

## Exemple

```
for (var i = 0; i < 5; i++) {
  System.out.println(i);
}
```

# Java

## Exemple

```
for (var i = 0; i < 5; i++) {
  System.out.println(i);
}
```

## Le résultat est

```
0
1
2
3
4
```

# Java

## Exercice 1

Écrire un programme qui permet d'afficher les nombres pairs compris entre 0 et 10.

# Java

## Première solution

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

# Java

## Première solution

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



## Deuxième solution

```
for (var i = 0; i < 10; i += 2) {
  System.out.println(i);
}
```

# Java

**Étant donnée la chaîne de caractères suivante**

```
String maChaine = "Une première phrase. Et voici une
    deuxième.";
```

# Java

**Étant donnée la chaîne de caractères suivante**

```
String maChaine = "Une première phrase. Et voici une
    deuxième.";
```

## Exercice 2

Écrire un programme qui permet de compter le nombre de mots et de phrases dans `maChaine`.

# Java

## Exercice 3

Écrire un programme qui demande à l'utilisateur de saisir deux entiers positifs et qui calcule puis affiche leur plus grand diviseur commun.

# Java

## Exemple avec `break`

```
int j = 5;
do {
    System.out.println(j);
    if (j == 3) {
        break;
    }
    j--;
} while (j > 0);
```

# Java

## Exemple avec `break`

```
int j = 5;
do {
    System.out.println(j);
    if (j == 3) {
        break;
    }
    j--;
} while (j > 0);
```



## Résultat

```
5
4
3
```

# Java

**Le `break` avec un label permet de sortir de plusieurs niveaux de structure de contrôle imbriqués, pas seulement du `switch`**

```
outer:
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (j == 1) break outer; // sort des deux boucles
        System.out.println(i + "," + j);
    }
}
```

# Java

**Le `break` avec un label permet de sortir de plusieurs niveaux de structure de contrôle imbriqués, pas seulement du `switch`**

```
outer:
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (j == 1) break outer; // sort des deux boucles
        System.out.println(i + "," + j);
    }
}
```



## Explication

- Utilisation de `break outer;` pour quitter les deux boucles.
- Nécessite de nommer le bloc à quitter.

# Java

**Exemple avec** `continue`

```
int j = 5;
while (j > 0) {
    if (j == 3) {
        j--;
        continue;
    }
    j--;
    System.out.println(j);
}
```

# Java

**Exemple avec** `continue`

```
int j = 5;
while (j > 0) {
    if (j == 3) {
        j--;
        continue;
    }
    j--;
    System.out.println(j);
}
```



**Résultat**

```
4
3
1
0
```

# Java

## En **Java**, les tableaux sont statiques

- Tous les éléments doivent avoir le même type
- Il faut préciser une taille qu'on ne peut dépasser

# Java

## En **Java**, les tableaux sont statiques

- Tous les éléments doivent avoir le même type
- Il faut préciser une taille qu'on ne peut dépasser

### Déclaration d'un tableau : syntaxe

```
type [] nomTableau = new type[taille];
```

# Java

## En **Java**, les tableaux sont statiques

- Tous les éléments doivent avoir le même type
- Il faut préciser une taille qu'on ne peut dépasser

### Déclaration d'un tableau : syntaxe

```
type [] nomTableau = new type[taille];
```

### Déclaration d'un tableau d'entier de taille 2 : exemple

```
int [] tab = new int[2];
```

# Java

## Affectation de valeurs aux cases de tableau

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

# Java

## Affectation de valeurs aux cases de tableau

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

## Ceci déclenche une exception

```
tab[2] = 10;
```



## L'exception

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
        at org.eclipse.classes.FirstClass.main(FirstClass.java:11)
```

## On peut faire une déclaration + une initialisation

```
int[] tab = new int[] { 5, 3 };
```

**On peut faire une déclaration + une initialisation**

```
int[] tab = new int[] { 5, 3 };
```

**On peut faire une déclaration + une initialisation**

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

Comme si la taille du tableau a été fixée à deux

**On peut faire une déclaration + une initialisation**

```
int[] tab = new int[] { 5, 3 };
```

**On peut faire une déclaration + une initialisation**

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

Comme si la taille du tableau a été fixée à deux

**Donc, ceci déclenche aussi une exception**

```
tab[2] = 10;
```

**L'exception**

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
        at org.eclipse.classes.FirstClass.main(FirstClass.java:11)
```

# Java

**En Java, on ne peut afficher le contenu d'un tableau de la manière suivante**

```
System.out.println(tab);
// affiche [I@1c4af82c
```

# Java

**En Java, on ne peut afficher le contenu d'un tableau de la manière suivante**

```
System.out.println(tab);
// affiche [I@1c4af82c
```

**Mais on peut le convertir en `String` et ensuite l'afficher**

```
System.out.println(Arrays.toString(tab));
// affiche [5, 3]
```

# Java

## Pour parcourir le tableau

```
for (int i = 0; i < tab.length; i++){
    System.out.println(tab[i]);
}
```

# Java

## Pour parcourir le tableau

```
for (int i = 0; i < tab.length; i++){
    System.out.println(tab[i]);
}
```

## Où encore la version simplifiée (`foreach`)

```
for (int elt : tab){
    System.out.println(elt);
}
```

# Java

**Étant donné le tableau suivant**

```
String [] voitures = { "peugeot", "ford", "fiat", "mercedes", "seat" };
```

# Java

**Étant donné le tableau suivant**

```
String [] voitures = { "peugeot", "ford", "fiat", "mercedes", "seat" };
```

## Exercice 1

Écrire un programme qui permet de calculer le nombre total de caractères de toutes les chaînes présentes dans le tableau `voitures`.

# Java

**Étant données les deux variables suivantes**

```
String voiture = "ford";
String [] voitures = { "peugeot", "ford", "fiat", "mercedes", "seat", "
   ford", "fiat", "ford" };
```

# Java

**Étant données les deux variables suivantes**

```
String voiture = "ford";
String [] voitures = { "peugeot", "ford", "fiat", "mercedes", "seat", "
   ford", "fiat", "ford" };
```

## Exercice 2

Écrire un programme qui permet de calculer et afficher le nombre d'occurrence de `voiture` dans `voitures`.

# Java

## Déclaration d'un tableau à deux dimensions

```
type[][] nomTableau = new type[nbLignes][nbColonnes];
```

# Java

## Déclaration d'un tableau à deux dimensions

```
type[][] nomTableau = new type[nbLignes][nbColonnes];
```

## Exemple

```
int[][] tab2dim = new int[2][2];
```

# Java

## Déclaration + initialisation

```
int[][] tab2dim = new int[][]
{
     {1, 2},
     {3, 4}
};
```

# Java

## Déclaration + initialisation

```
int[][] tab2dim = new int[][]
{
     {1, 2},
     {3, 4}
};
```

## Ou

```
int[][] tab2dim =
{
     {1, 2},
     {3, 4}
};
```

# Java

## Parcourir un tableau à deux dimensions

```
for (int[] ligne : tab2dim) {
        for (int elt : ligne) {
                System.out.println(elt);
        }
}
```

# Java

## Parcourir un tableau à deux dimensions

```
for (int[] ligne : tab2dim) {
        for (int elt : ligne) {
                System.out.println(elt);
        }
}
```



## Ou

```
for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
                System.out.println(tab2dim[i][j]);
        }
}
```

**Étant données les variables suivantes**

```
int valeur = 5;
int[][] matrice =
{
    { 2, 3, 5, 7 },
    { 5, 1, 5, 5 },
    { 4, 2, 9, 5 },
    { 8, 5, 6, 5 }
};
```

**Étant données les variables suivantes**

```
int valeur = 5;
int[][] matrice =
{
    { 2, 3, 5, 7 },
    { 5, 1, 5, 5 },
    { 4, 2, 9, 5 },
    { 8, 5, 6, 5 }
};
```

## Exercice

Écrire un programme qui

- parcourt la matrice précédente,
- calcule le nombre d'occurrence de `valeur` dans chaque ligne de `matrice`,
- stocke le résultat dans un tableau `occurrences`.

# Java

**Étant données les variables suivantes**

```
int valeur = 5;
int[][] matrice =
{
    { 2, 3, 5, 7 },
    { 5, 1, 5, 5 },
    { 4, 2, 9, 5 },
    { 8, 5, 6, 5 }
};
```

# Java

**Étant données les variables suivantes**

```
int valeur = 5;
int[][] matrice =
{
    { 2, 3, 5, 7 },
    { 5, 1, 5, 5 },
    { 4, 2, 9, 5 },
    { 8, 5, 6, 5 }
};
```



## Exercice

Écrire un programme qui permet de déterminer si `valeur` est présente dans chaque ligne de `matrice`.

# Java

## Exercice

Écrire un programme qui calcule

1. la somme de deux matrices carrées,
2. le produit de deux matrices carrées.

# Java

## Une constante

un élément qui ne peut changer de valeur

# Java

## Une constante

un élément qui ne peut changer de valeur

## Nom de constantes en **Java**

- Utiliser que des lettres en majuscules.
- Pour les noms composés, utiliser underscore comme séparateur.

# Java

## Pour déclarer une constante

il faut utiliser le mot-clé `final`

# Java

## Pour déclarer une constante

il faut utiliser le mot-clé `final`

### Déclaration d'une constante

```
final double PI = 3.1415;
```

# Java

## Pour déclarer une constante

il faut utiliser le mot-clé `final`

### Déclaration d'une constante

```
final double PI = 3.1415;
```

### L'instruction suivante ne peut être acceptée

```
PI = 5;
```

# Java

## Le mot-clé `const` en **Java**

- `const` est un mot réservé en **Java** : cela signifie qu'on ne peut pas l'utiliser comme identifiant (nom de variable, de classe...).
- Cependant, contrairement à certains autres langages (comme **C++**), `const` n'est pas utilisé activement dans le langage **Java**. Il est réservé pour une éventuelle utilisation future, mais n'a actuellement aucune fonctionnalité.
- **final** est utilisé pour imposer de l'invariance.

# Java

## Une méthode

visibilité [+ `static`] + type de retour + nomMéthode([les paramètres]) + son implémentation

# Java

## Une méthode

visibilité [+ `static`] + type de retour + nomMéthode([les paramètres]) + son implémentation

**Exemple de déclaration d'une méthode**

```
public static int somme (int a, int b) {
  return a + b;
}
```

# Java

## Une méthode

visibilité [+ `static`] + type de retour + nomMéthode([les paramètres]) + son implémentation

### Exemple de déclaration d'une méthode

```
public static int somme (int a, int b) {
  return a + b;
}
```

### Exemple d'appel d'une méthode

```
System.out.println(somme(2, 3));
// affiche 5
```

# Java

## Exercice

Écrire une méthode `maximum2(a, b)` qui retourne le maximum entre `a` et `b`.

# Java

## Exercice

Écrire une méthode `maximum2(a, b)` qui retourne le maximum entre `a` et `b`.

## Exercice

Écrire une méthode `maximum3(a, b, c)` qui retourne le maximum entre `a`, `b` et `c` (vous pouvez utiliser la méthode `maximum2`) .

# Java

## Exercice

Écrire une méthode qui permet de déterminer si un nombre passé en paramètre est premier. La méthode retourne un booléen.

# Java

## Déclaration de méthode avec un nombre indéterminé de paramètres (`varargs`)

```
public static int somme(int... tab) {
  int res = 0;
  for (int elt : tab)
    res += elt;
  return res;
}
```

# Java

## Déclaration de méthode avec un nombre indéterminé de paramètres (`varargs`)

```
public static int somme(int... tab) {
  int res = 0;
  for (int elt : tab)
    res += elt;
  return res;
}
```

## Exemple d'appel

```
System.out.println(somme(2));
// affiche 2
System.out.println(somme(2, 3));
// affiche 5
System.out.println(somme(2, 3, 5));
// affiche 10
System.out.println(somme(2, 3, 5, 9));
// affiche 19
```

# Java

## Remarques

- **varargs** est disponible depuis **Java 5**
- On ne peut avoir qu'un seul `varargs` par méthode
- Le paramètre `varargs` doit être le dernier dans la liste de paramètres d'une méthode

# Java

## Exercice 1

Écrire une méthode qui accepte un nombre variable de paramètre de type `String` et qui retourne la chaîne la plus longue (ayant le plus grand nombre de caractères).

# Java

## Exercice 2

Écrire une méthode qui détermine si le premier paramètre est divisible par tous les autres paramètres. Le nombre de paramètres est variable et la méthode retourne un booléen.



### Exemple

```
estDivisiblePar(10, 2, 3, 5);
// false

estDivisiblePar(10, 2, 5);
// true
```

# Java

## Pour appeler une méthode avec **varargs**, on peut passer

- Aucun argument.
- Un nombre quelconque d'arguments.
- Un tableau explicite.

# Java

## Remarques

Comme **Java** supporte la surcharge de méthodes

- Pas de paramètres optionnels
- Pas de valeur par défaut pour les paramètres