



# Programmation Orientée Objet Java



Les bases de Java



v1.0 - 01.02.2024

# Plan

---

- Nommage
- Structure d'un programme
- Mots réservés de Java
- Package
- Classe
  - Constructeur
- Les attributs
- Les méthodes
- Les blocs de code
- Les variables
- Commentaires
- Le mot clé static
- Le mot clé final
- Les opérateurs
- Types
  - primitifs,
  - les tableaux,
  - les classes Standards
- Instructions
  - conditionnelles
  - d'itération

# Pour commencer

---

# A savoir

---

Utilisation de **Java 8+** pour ce cours d'initiation à Java

Mais seules les API (fonctionnalités) principales de Java seront étudiées

Utiliser la documentation des API Java : <https://docs.oracle.com/javase/8/docs/api/>



Apprendre à utiliser l'IDE pour coder "efficacement" en Java

# Les différents types de nommage

Il existe différents types de nommage et chaque type porte un nom :

- **PascalCase** : écriture en minuscule sauf la première lettre de chaque mot en majuscule  
Exemple : JeSuisEnPascalCase
- **camelCase** : écriture en PascalCase mais qui commence par une minuscule  
Exemple : jeSuisEnCamelCase
- **snake\_case** : écriture en minuscule avec les mots séparés par un souligné (underscore)  
Exemple : je\_suis\_en\_snake\_case
- **SCREAMING\_SNAKE\_CASE** : écriture en majuscule avec les mots séparés par un souligné (underscore)  
Exemple : JE\_SUIS\_EN\_SCREAMING\_SNAKE\_CASE
- **kebab-case** : écriture en minuscule avec les mots séparés par un tiret  
Exemple : je-suis-en-kebab-case

## En Java

- } Le nom des **classes (constructeurs), interfaces, enums et annotations**
- } Le nom des **attributs, variables, paramètres et méthodes**
- } 
- } Les **constantes**
- } 

# Règles de nommage

---

Le code en langage Java est sensible à la casse (= distinction minuscule/majuscule).

Les déclarations (classes, attributs, méthodes, variables, paramètres, ...) au sein de votre code doivent être en alphanumérique (pas d'accent, ni de caractères spéciaux) avec l'underscore (souligné) autorisé.

Référence : <https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html>

# Structure d'un programme

---

# Java : Structure d'un code Java (1/9)

---

Un programme Java est constitué de code Java qui doit être défini au sein d'une classe Java contenue dans un fichier.

## Convention de nommage :

- Un fichier java doit avoir l'extension **.java**
- Un fichier ne peut comporter qu'une **seule classe Java principale**
- Un fichier Java doit avoir le **même nom que la classe principale** qu'il contient  
Une **classe Java** doit commencer par une **majuscule** (convention)

Exemple avec le fichier : **HelloWorld.java** dont la classe principale se nomme **HelloWorld**



# Java : Structure d'un code Java (2/9)

```
1 package fr.esgi.poo.java;
2
3 import java.lang.Integer;
4
5 public class MySampleClass {
6     public static final double PI = 3.1415926;
7     public Integer myNumber;
8     private boolean isOK;
9
10    public MySampleClass(int anIntNumber) {
11        this.myNumber = new Integer(anIntNumber);
12    }
13
14    public boolean isOK() {
15        return isOK;
16    }
17
18    public void setOK(boolean newOK) {
19        this.isOK = newOK;
20    }
21 }
```

## Explications générales

Les instructions en Java doivent se terminer par un point-virgule ;

Mais une instruction peut tenir sur plusieurs lignes

Java est **sensible à la casse**

Un **bloc de code** doit commencer par une accolade ouvrante { et se terminer par une accolade fermante }

L'indentation du code est libre mais doit permettre sa bonne lisibilité

# Java : Structure d'un code Java (3/9)

```
1 package fr.esgi.poo.java;
2
3 import java.lang.Integer;
4
5 public class MySampleClass {
6     public static final double PI = 3.1415926;
7     public Integer myNumber;
8     private boolean isOK;
9
10    public MySampleClass(int anIntNumber) {
11        this.myNumber = new Integer(anIntNumber);
12    }
13
14    public boolean isOK() {
15        return isOK;
16    }
17
18    public void setOK(boolean newOK) {
19        this.isOK = newOK;
20    }
21 }
```

## Explication de la ligne 1

Le mot réservé **package** permet de définir le nom du package de la classe

Le nom complet de la classe est donc **fr.esgi.poo.java.MySampleClass**

Le nom complet de la classe doit être **UNIQUE** dans mon projet Java

# Java : Structure d'un code Java (4/9)

```
1  package fr.esgi.poo.java;
2
3  import java.lang.Integer;
4
5  public class MySampleClass {
6      public static final double PI = 3.1415926;
7      public Integer myNumber;
8      private boolean isOK;
9
10     public MySampleClass(int anIntNumber) {
11         this.myNumber = new Integer(anIntNumber);
12     }
13
14     public boolean isOK() {
15         return isOK;
16     }
17
18     public void setOK(boolean newOK) {
19         this.isOK = newOK;
20     }
21 }
```

## Explication de la ligne 3

Le mot réservé **import** permet de définir le nom complet (avec son package) d'une classe utilisée au sein de notre classe (dépendance)

## Explication de la ligne 5

Le mot réservé **public** permet de définir la portée de la classe

Le mot réservé **class** permet de définir la classe Java avec son nom **MySampleClass**

# Java : Structure d'un code Java (5/9)

```
1 package fr.esgi.poo.java;
2
3 import java.lang.Integer;
4
5 public class MySampleClass {
6     public static final double PI = 3.1415926;
7     public Integer myNumber;
8     private boolean isOK;
9
10    public MySampleClass(int anIntNumber) {
11        this.myNumber = new Integer(anIntNumber);
12    }
13
14    public boolean isOK() {
15        return isOK;
16    }
17
18    public void setOK(boolean newOK) {
19        this.isOK = newOK;
20    }
21 }
```

## Explication de la ligne 6

Le mot réservé **public** permet de définir la portée de la variable

Les mots réservés **static final** permet de définir que **PI** est une constante

Le mot réservé **double** permet de définir que la constante **PI** définie est de type nombre décimal

La constante définie a la valeur 3.1415926

# Java : Structure d'un code Java (6/9)

```
1  package fr.esgi.poo.java;
2
3  import java.lang.Integer;
4
5  public class MySampleClass {
6      public static final double PI = 3.1415926;
7      public Integer myNumber;
8      private boolean isOK;
9
10     public MySampleClass(int anIntNumber) {
11         this.myNumber = new Integer(anIntNumber);
12     }
13
14     public boolean isOK() {
15         return isOK;
16     }
17
18     public void setOK(boolean newOK) {
19         this.isOK = newOK;
20     }
21 }
```

## Explication de la ligne 7

Le mot réservé **public** permet de définir la portée de l'attribut (variable) **myNumber** qui est déclaré de type **Integer** (nombre entier)

## Explication de la ligne 8

Le mot réservé **private** permet de définir la portée de l'attribut (variable) **isOK** qui est déclaré de type primitif **boolean**

# Java : Structure d'un code Java (7/9)

```
1  package fr.esgi.poo.java;
2
3  import java.lang.Integer;
4
5  public class MySampleClass {
6      public static final double PI = 3.1415926;
7      public Integer myNumber;
8      private boolean isOK;
9
10     public MySampleClass(int anIntNumber) {
11         this.myNumber = new Integer(anIntNumber);
12     }
13
14     public boolean isOK() {
15         return isOK;
16     }
17
18     public void setOK(boolean newOK) {
19         this.isOK = newOK;
20     }
21 }
```

## Explication des lignes 10-12

Ce bloc de code définit un **constructeur** de la classe

Une classe Java peut avoir plusieurs constructeurs

Un constructeur est appelé à l'instanciation de la classe et **renvoie donc automatiquement une instance** de la classe

Elle permet aussi d'affecter la valeur de **valeur** à l'attribut/variable d'instance (préfixée de **this**) **myNumber**

# Java : Structure d'un code Java (8/9)

```
1 package fr.esgi.poo.java;
2
3 import java.lang.Integer;
4
5 public class MySampleClass {
6     public static final double PI = 3.1415926;
7     public Integer myNumber;
8     private boolean isOK;
9
10    public MySampleClass(int anIntNumber) {
11        this.myNumber = new Integer(anIntNumber);
12    }
13
14    public boolean isOK() {
15        return isOK;
16    }
17
18    public void setOK(boolean newOK) {
19        this.isOK = newOK;
20    }
21 }
```

## Explication des lignes 14-16

Ce bloc de code définit une **méthode** de la classe

Cette **méthode** n'a pas de paramètre mais renvoie un booléen comme l'indique le type de sortie et le mot réservé **return**

Le booléen retourné correspond à l'attribut/variable d'instance **isOK**

Cette méthode est appelée un 'getter'

# Java : Structure d'un code Java (9/9)

```
1  package fr.esgi.poo.java;
2
3  import java.lang.Integer;
4
5  public class MySampleClass {
6      public static final double PI = 3.1415926;
7      public Integer myNumber;
8      private boolean isOK;
9
10     public MySampleClass(int anIntNumber) {
11         this.myNumber = new Integer(anIntNumber);
12     }
13
14     public boolean isOK() {
15         return isOK;
16     }
17
18     public void setOK(boolean newOK) {
19         this.isOK = newOK;
20     }
21 }
```

## Explication des lignes 18-20

Ce bloc de code définit une **méthode** de la classe

Cette **méthode** a un paramètre de type booléen nommé **newOK**, mais ne renvoie aucune valeur comme l'indique le mot réservé **void**

Cette méthode a pour fonction d'affecter la valeur de **newOK** à l'attribut/variable d'instance **isOK**

Cette méthode est appelée un 'setter'



## TD03.01-HelloWorld

### Exercice :

1. Dans votre IDE, créez un nouveau projet Java
2. Définissez un nom de package
3. Créez une classe **HelloWorld**
4. Ajoutez la méthode **public static void main(String[ ] args)**
5. Au sein de cette méthode, ajoutez
  - a. Une variable de type **String** ayant pour valeur **"Hello World!"**
  - b. Le code **System.out.println();** alimenté de votre variable
6. Exécutez ce code et regardez le résultat affiché dans la console  
Que voyez-vous ?
7. Mettez un point d'arrêt dans votre code et lancez l'exécution en mode debug  
Observez la valeur de votre variable ?



Un peu de ligne de commande pour s'amuser :

- Compiler votre programme
- Exécuter votre programme

# Mots réservés

---

# Mots réservés : C'est quoi ?

---

Les mots réservés (keywords) sont des mots du langage qui ont une particularité/fonction spécifique :

**abstract**, assert<sup>java 1.4</sup>, **boolean**, **break**, **byte**, **case**, **catch**, **char**, **class**, const<sup>non utilisé</sup>, **continue**, **default**, **do**, **double**, **else**, enum<sup>java 5</sup>, **extends**, **final**, **finally**, **float**, **for**, goto<sup>non utilisé</sup>, **if**, **implements**, **import**, **instanceof**, **int**, **interface**, **long**, native, **new**, non-sealed<sup>java 15</sup>, **package**, **private**, **protected**, **public**, **return**, **short**, **static**, strictfp<sup>java 1.2</sup>, **super**, **switch**, **synchronized**, **this**, **throw**, **throws**, **transient**, **try**, **var**, **void**, **volatile**, **while**

Les valeurs réservés :

**true**, **false**, **null**

Identifiants réservés :

permits<sup>java 15</sup>, record<sup>java 14</sup>, sealed<sup>java 15</sup>, var<sup>java 10</sup>, yield<sup>java 13</sup>



Remarque : **NE PAS** utiliser un mot/valeur/identifiant réservé pour nommer vos variables !

# Package

---

# Package : Qu'est-ce ?

---

Un package Java permet de regrouper des classes dans un ensemble (paquet) appelé **package**. Les packages permettent donc de hiérarchiser, d'organiser les classes.

Un programme Java doit comprendre au moins un **package** dans lequel les classes Java seront intégrées.

De plus, chaque classe doit appartenir à un package c'est pourquoi, il faut **en tout début de chaque classe** indiquer avec le mot clé **package** le nom du package de la classe.

Exemple : `package fr.esgi.poo.java.monpackage;`

Remarques :

- Un package correspond en fait à un dossier du système de fichiers.
- Souvent le nom de package est défini comme un nom de domaine Web mais inversé.

# Package : Nommage

---

Le nom d'un **package** s'écrit en **lettres minuscules** avec les mots séparés par un point.

Les chiffres ou le caractère souligné (underscore) sont autorisés au sein des mots.

Les mots ne doivent pas commencer par un chiffre. Par exemple ~~fr.esgi.2i~~ ❌

Certains packages sont réservés. Par exemple ~~java.esgi.classe2~~ ❌

## Exemples de notation de package :

- fr.esgi.poo.java ✓
- android.net.http ✓
- java.lang ✓
- org.apache.http ✓
- org.w3c.dom ✓

Le nom complet d'une **classe** est défini par **<son package>.<nom de la classe>**.

Exemple : Le nom complet de la classe "Truc" dans le package "fr.esgi.poo" sera donc "fr.esgi.poo.Truc"

# Package : Utilisation

---

Si une classe Java souhaite utiliser une autre classe Java, il faudra que cela soit précisé dans la classe. Il faut pour cela utiliser le mot clé **import**.

La déclaration des **import** doit être faite en début de classe.

Il est possible d'utiliser l'astérisque **\*** pour éviter de définir les classes d'un même package.

## Exemples :

- `import fr.esgi.poo.Truc;`
- `import fr.esgi.poo.deuxieme.annee.Machin;`
- `import java.lang.*;`

<https://docs.oracle.com/javase/tutorial/java/package/index.html>

# Packages du langage Java

---

Voici quelques packages standards Java :

- `java.io` : Accès aux flux entrants et sortants
- `java.lang` : Classes et interfaces fondamentales
- `java.math` : Opérations mathématiques
- `java.net` : Accès aux réseaux
- `java.util` : Utilitaires (collections, internationalisation, logging, expressions régulières, ...)
- ...
- `javax.*` : Packages d'extension



# Modules Java

Java 9+

Depuis **Java 9** (Jigsaw), un “super packaging” est apparu sous le nom de **Module**.

Pourquoi ?

L'objectif est de favoriser la programmation modulaire c'est-à-dire avec un découpage en modules réutilisables.

Comment ?

En regroupant les packages dans un même module.

Par exemple, le module **java.base** contient les packages :

- java.lang
- java.io
- java.net
- java.util
- ...

# Exemples de projets Java

---

# Projets Java populaires

---

Quelques projets Java populaires disponibles sur Github : <https://github.com/topics/java>

- Dagger
- Apache Commons : <https://commons.apache.org/> et <https://commons.apache.org/proper/commons-lang/apidocs/index.html>
- Apache Dubbo : <https://github.com/apache/dubbo/tree/3.2/dubbo-common/src/main/java/org/apache/dubbo>
- ElasticSearch
- Fastjson
- Guava : <https://github.com/google/guava>
- Jenkins
- Junit
- Mockito
- Okhttp
- Retrofit

# Bonnes pratiques du bon développeur

---

# Bonnes pratiques du développeur

---

## Bonnes pratiques du développeur :

- NE PAS écrire du code QUE pour vous ! Il doit être facilement compréhensible
- Coder avec une présentation soignée : code aéré, correctement indenté, lisible, ...  
=> agréable à lire => plus facile à comprendre
- Commenter votre code = expliquer les parties les moins évidentes 👍
- Découper votre code et ne pas avoir de gros blocs de code 👍
  - Classe = 500 lignes max
  - Constructeur/Méthode = 50 lignes max
- Avoir un code facilement compréhensible (principe **KISS**: **K**eeP **I**t **S**imple **S**tupid)
- Coder avec une indentation qui dépasse les 5 niveaux 😡
- Factoriser votre code = NE PAS avoir 2 fois le même bloc de code dans votre projet (principe **DRY**: **D**on't **R**epeat **Y**ourself)
- Utiliser des noms explicites pour les classes, constantes, variables, méthodes, ... ❤️
- Le code doit être testé, et retesté.... Et retesté ! 🎆🎆

# Bonnes pratiques du développeur Java

---

## Bonnes pratiques du développeur Java :

- Les fichiers de code Java dans des fichiers .java
- Toujours un package racine “unique”
- Répartition des classes dans des packages (par exemple des classes pour écrire/afficher des logs dans un package “logger”, ...)
- Respect du nommage des packages
- Respect du nommage des classes (constructeurs), interfaces, enums et annotations  
=> leur nom commence par une **MAJUSCULE**
- Respect du nommage des variables et méthodes  
=> leur nom commence par une **minuscule**
- Utiliser des commentaires Javadoc

# Classe

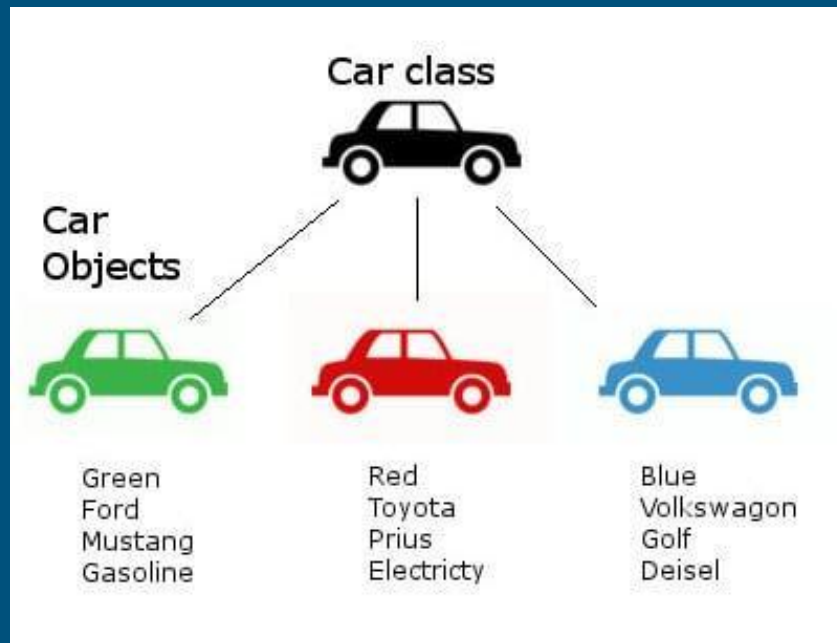
---

# Classe : Qu'est ce ?

Une **classe** Java permet de définir un **type d'objets**. C'est un "moule" à objets.

Elle sert donc à décrire les caractéristiques d'un type d'objets et ses actions.

Exemple : La classe "Car" pourrait définir par exemple, la longueur, la largeur, le type d'essence, ... des voitures. Elle pourrait définir aussi ce que peut faire la voiture, tel qu'avancer, reculer, tourner, ...





# Classe : Ses composantes

---

Une classe est une nouvelle structure de données, un type de donnée

Une **classe** est constituée de 2 composantes :

- Les **attributs** (fields) de la classe  
Ils définissent les propriétés/caractéristiques de l'objet  
Ce sont les données membres de l'objet
- Les **méthodes** (methods) de la classe  
Elles définissent les opérations/traitements/actions/comportements de l'objet  
C'est le code de l'objet

# Classe : Déclaration (1/2)

---

Une **classe** est définie au sein d'un fichier qui a le même nom que celui de la classe principale qu'il contient.

Exemple : la classe "Car" sera définie au sein du fichier "Car.java"

Pour définir une classe, elle doit être préfixée par le mot réservé **class**

Le nom d'une classe doit s'écrire en **PascalCase** (chaque mot qui la compose commence par une Majuscule).

Exemple : **class** MaClasseQuiSaitToutFaire { ... }

A la compilation, une classe se compile en un **fichier .class** comportant son nom.

Exemple : MaClasseQuiSaitToutFaire.java -> *compilation* -> MaClasseQuiSaitToutFaire.class

# Classe : Déclaration (2/2)

---

La syntaxe de déclaration d'une classe est :

```
<portée> <static> class NomDeLaClasse (extends <parentClass>, implements <interface>, ...) {  
    // Le corps de la classe  
}
```

La <portée> peut être soit **public**, **protected**, "package" ou **private**

# Classe : Constructeur (1/3)

Un constructeur d'une classe permet de définir la manière d'initialiser un objet de cette classe.

Une classe peut avoir de **0 à N constructeurs** qui doivent avoir un nombre de paramètres différents et de types différents

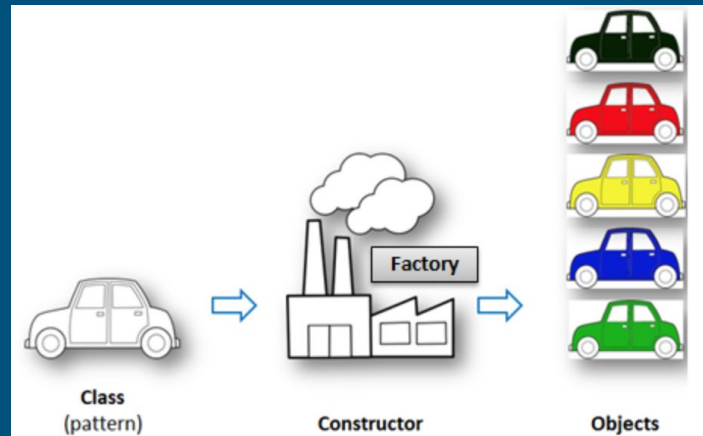
Les constructeurs d'une même classe ont **tous le même nom** : le nom de la classe 

Si la classe ne déclare pas de constructeur, un constructeur sans paramètre (issu du type Object) sera son constructeur par défaut.



Les constructeurs peuvent avoir plusieurs spécificités :

- Avoir une **portée/visibilité** spécifique : **public**, **protected**, "package" ou **private**
- Avoir de 0 à n paramètres



# Classe : Constructeur (2/3)

---

La syntaxe de déclaration d'un constructeur est :

```
<portée> NomDuConstructeur(<type param1> param1, ..., <type paramN> paramN) {  
    // Du code  
}
```

La **<portée>** peut être soit **public**, **protected**, "package" ou **private**

Il n'est pas obligatoire de spécifier des paramètres au constructeur. Un constructeur sans paramètre est possible et courant.

Remarques :

- Pas de type de retour du fait que c'est forcément une instance de la classe qui est renvoyée
- Un constructeur ne peut pas être déclaré **static**

# Classe : Constructeur (3/3)

La convention de nommage Java définit que le **nom des constructeurs** est égal au nom de sa classe. Il doit être écrit en **PascalCase**.



Exemple pour la classe :

```
public class MachinChose {  
    public MachinChose() { ... }  
    public MachinChose(int i) { ... }  
    public MachinChose(String chaine) { ... }  
    public MachinChose(int i, String chaine) { ... }  
    ....  
}
```

Notez que ces constructeurs n'ont pas le même prototype (signature)



Les constructeurs sont généralement déclarés au début de la classe.

# Classe : Instanciation

---

Pour utiliser une **classe** Java (au sein d'une autre **classe** Java), il va falloir créer une **instance** de cette classe (= instancier cette classe, c'est-à-dire la charger en mémoire et l'initialiser).

Pour réaliser cela, il faut utiliser le mot clé **new** suivi d'un des constructeurs de la classe à instancier. Par conséquent, l'instanciation d'une classe consiste à appeler un de ses constructeurs avec l'instruction **new**

Exemple :

```
Truc premierTruc = new Truc();  
Truc deuxiemeTruc = new Truc();
```

Remarque : L'instanciation d'une classe va demander à la JVM d'allouer un espace mémoire pour stocker la structure et les données de la classe.

## TD03.02-Multi Constructeur

### Exercice :

1. Dans votre IDE, créez un nouveau projet Java nommé par exemple “PointInSpace”
2. Définissez un nom de package
3. Créez une classe nommée **Point** qui va permettre de modéliser un point dans l'espace :
  - a. Définissez les attributs adéquates de type entier
  - b. Définissez plusieurs constructeurs (4 par exemple) pour initialiser les attributs
4. Créez une nouvelle classe (nom à votre convenance) et y ajouter la méthode **public static void main(String[] args)** qui va permettre de créer des points en utilisant tous leurs constructeurs  
Ne pouvez-vous pas simplifier votre code ?
5. Modifiez votre classe afin que les attributs puissent avoir une valeur comprise entre 0 et 5 (0 étant la valeur par défaut).  
Voyez-vous l'intérêt d'avoir simplifié votre code ?
6. Codez la méthode **toString()** afin d'afficher dans la console les coordonnées des points dans l'espace que vous allez créer à l'aide de chaque constructeur



# Classe : Destruction

---

En Java, il existe une méthode prédéfinie nommée `finalize()` qui sert à libérer la mémoire occupée (des données chargées et référencées par la classe) mais dont son exécution n'est pas garantie.

Mais alors ?

Un mécanisme appelé **Garbage Collector** (Ramasse Miettes en FR) se charge de nettoyer la mémoire, c'est à dire, qu'il vérifie que des objets ne sont plus utilisés (c'est à dire qui ne sont plus référencés) et si c'est le cas, libère la mémoire qu'ils occupent.

# Classe : D'autres types de déclaration

---

Contrairement à ce qu'on a vu précédemment, il est aussi possible de déclarer une classe au sein d'une autre classe. On parle alors de **Nested Class**.

- **Inner Class** : C'est une classe non **static** définie au sein d'une classe
- **Static Nested Class** : C'est une classe **static** définie au sein d'une classe
- **Local Class** : C'est une classe définie au sein d'un bloc de code (généralement une méthode)
- **Anonymous Class** : C'est une classe définie et instanciée en même temps

<https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html>

## TD03.03a-Classe et Constructeur

### Objectif du projet "Éclairage" :

Piloter à l'aide d'un interrupteur, l'éclairage (variable) de 2 lampes

### Exercice "Éclairage" (1/4) :

1. Dans votre IDE, créez un nouveau projet Java nommé par exemple Lighting
2. Définissez un nom de package
3. Créez une classe **Main**
4. Ajoutez la méthode **public static void main(String[] args)** mais laissez la vide
5. Créez une autre classe nommée **Lamp**
  - a. Définissez un constructeur avec un paramètre (sert à définir si la lampe est allumée ou éteinte) de type **boolean**
  - b. Définissez un constructeur avec un paramètre (sert à définir le niveau d'intensité de la lampe) de type **int**
6. Créez une autre classe nommée **Switch**
  - a. Définissez un constructeur avec 2 paramètres (sert à définir que l'interrupteur contrôle 2 lampes) de type **Lamp**
7. Dans la méthode **main** de la classe **Main**, instanciez 2 classes **Lamp** et une classe **Switch**

Remarque : Les attributs des classes seront définis au prochain TD

# Les attributs

---

# Attributs (1/3)

---

Un attribut (field) définit une caractéristique/propriété de classe.



Les **attributs** doivent être déclarés (bonne pratique) au début de la classe avant les constructeurs.

Les **attributs** peuvent avoir plusieurs spécificités :

- Avoir une **portée** spécifique : **public**, **protected**, "package" ou **private**
- Être **static** ou non



Au sein d'une même classe, deux attributs (variables d'instance) ne peuvent avoir le **même nom** (avec la même casse)

# Attributs (2/3)

---

La syntaxe de déclaration d'un attribut est :

`<portée> <static> <type> nomDeLaVariable = <valeur ou instantiation>;`

La `<portée>` peut être soit `public`, `protected`, "package" ou `private`

Utiliser `static` si l'on veut que l'attribut soit un attribut de classe et rien sinon

Le `<type>` peut être un type primitif, tableau ou objet

La déclaration d'une valeur ou d'une instantiation est optionnelle

Exemples :

```
public float maMoyenne;
```

```
private boolean isOpen = true;
```

```
String mot1 = "hello", mot2 = "world";
```

# Attributs (3/3)

---

La convention de nommage Java définit que le **nom des attributs (variables)** :

- Doit être écrit en **camelCase**

Syntaxe d'appel :

<instance de la classe>.<attribut de la classe> = <valeur>;

<type> <variable> = <instance de la classe>.<attribut de la classe>;

Exemples d'appel :

myCar.countWheels = 4;

int carWheels = myCar.countWheels;

# Attributs et leur portée

---

Les **attributs** peuvent avoir 4 portées différentes :

- **public** : la variable est visible par sa classe et toutes les autres classes.
- **protected** : la variable est visible par sa classe, par toutes ses sous-classes et les classes du même package
- "package" : la variable est visible par sa classe et les classes du même package
- **private** : la variable est visible QUE par sa classe



Il est souvent conseillé de définir les attributs en **private** (Cf. Encapsulation)

Les **attributs** NON **static** sont appelés "variables membres/d'instance".

Les **attributs** peuvent être aussi **static**.

Dans ce cas, ils sont appelés "variables de classe".



## TD03.03b-Attributs

### Exercice “Eclairage” (2/4) :

1. Dans votre classe **Lamp**
  - a. Définissez un attribut booléen **isOn**  
Il sert à définir si la lampe est allumée ou éteinte
  - b. Définissez un attribut entier **level**  
Il sert à définir le niveau d'intensité de la lampe
  - c. Définissez une constante entière **LEVEL\_MIN** avec la valeur 0
  - d. Définissez une constante entière **LEVEL\_MAX** avec la valeur 9
  - e. Modifiez les 2 constructeurs pour utiliser les attributs
2. Dans votre classe **Switch**
  - a. Définissez un attribut de type **Lamp** nommé **lamp1**  
Il sert à définir la 1ère lampe à contrôler
  - b. Définissez un attribut de type **Lamp** nommé **lamp2**  
Il sert à définir la 2ème lampe à contrôler
  - c. Modifiez le constructeur pour utiliser ces attributs

# Les méthodes

---

# Méthodes (1/4)

---

Une **méthode** définit une action que peut réaliser une classe.

Une **méthode** est toujours définie au sein d'une classe et doit avoir une **empreinte unique\*/prototype** dans la classe.

Une méthode **ne peut être définie** au sein d'un constructeur ou d'une méthode !

Les méthodes peuvent avoir plusieurs spécificités :

- Avoir une **portée** spécifique : **public**, **protected**, "package" ou **private**
- Être **static** ou non
- Avoir de 0 à n paramètres
- Retourner 0 ou 1 valeur/objet



**\*Empreinte unique** : 2 méthodes ne peuvent pas dans une même classe (ou par héritage) avoir à la fois : le même nom, le même nombre de paramètres et les mêmes types de paramètres et dans le même ordre

# Méthodes (2/4)

---

La syntaxe de déclaration d'une méthode est :

```
<portée> <static> <type retour | void> nomDeLaMéthode(<type param1> param1, ..., <type paramN> paramN) {  
    // Du code  
}
```

La <portée> peut être soit **public**, **protected**, "package" ou **private**

Utiliser **static** si l'on veut que la méthode soit une méthode de classe et rien sinon

Le <type retour> peut être un type primitif, tableau ou objet ou rien en utilisant **void**

# Méthodes (3/4)

---

La convention de nommage Java définit que le **nom des méthodes** :

- Doit être écrit en **camelCase**



Il est conseillé qu'un nom de méthode contienne un verbe (pour traduire la notion d'action)

Exemples :

```
public float sum() { ... }
```

```
private void display(String text) { ... }
```

# Méthodes (4/4)

---

Pour **exécuter la méthode** d'une classe (non statique), il suffit qu'une instance de cette classe appelle cette méthode avec les paramètres tels qu'ils ont été définis.

## Syntaxe d'appel :

<instance de la classe>.<nom de la méthode>;

<type retour de la méthode> <variable> = <instance de la classe>.<nom de la méthode>;

## Exemples :

```
int wordLength = aWord.length();
```

```
total = calculator.sum(1, 2);
```

```
sentence = mySentence.doConcatenation("il était ", 1, " fois");
```

# Méthodes et leurs paramètres

---

Les méthodes peuvent avoir de **0 à n paramètres**.



Pour une meilleure lisibilité, il est conseillé que le nombre de paramètres soit faible (de l'ordre de 10 max).

Chacun des paramètres doit avoir un **type** (primitif, tableau, objet ou une fonction) et le **nom d'une variable**

Les paramètres d'une méthode doivent tous avoir des noms différents

La définition des paramètres est séparée par une virgule

Exemples :

```
public void myMethod(boolean b, int i, Thing aThing) { ... }
```

```
protected float calcAmountTTC(float amountHT) { ... }
```

# Méthodes et leurs paramètres

---

Le mécanisme de passage des paramètres est différent pour :

- Un type primitif

Dans ce cas, le passage du paramètre est fait par valeur



**La modification de la variable de type primitif passée en paramètre ne sera effective qu'au sein de la méthode et non en dehors de la méthode !!**

- Un objet ou un tableau

Dans ce cas, le passage du paramètre est fait par référence



**C'est une référence de l'objet qui est passée, le contenu de l'objet modifié au sein de la méthode sera conservé modifié hors de la méthode !!**



# Méthodes et leur valeur retour

---

Les méthodes peuvent avoir une valeur (primitif, tableau ou objet) de retour ou pas.

Si la méthode renvoie une valeur (son résultat) alors :

- Le type du résultat est défini devant le nom de la méthode
- Une instruction `return` doit obligatoirement être utilisée pour renvoyer le résultat

Si la méthode ne renvoie pas de valeur alors :

- Le mot clé `void` est défini devant le nom de la méthode
- La méthode s'exécute jusqu'à sa dernière ligne à moins d'avoir une instruction `return` suivie de rien

# Méthodes et leur portée

---

Les méthodes peuvent avoir 4 portées différentes :

- **public** : la méthode est visible par sa classe et toutes les autres classes.
- **protected** : la méthode est visible par sa classe, par toutes ses sous-classes et les classes du même package
- "package" : la méthode est visible par sa classe et les classes du même package
- **private** : la méthode est visible QUE par sa classe

Les méthodes (non **static**) sont appelées des **méthodes d'instance**.

# Méthode statique

---

Les méthodes peuvent être aussi **static**.

Dans ce cas, les méthodes sont appelées des **méthodes de classe** (et non d'instance).



Au sein des méthodes **static**, il n'est pas possible d'utiliser les variables membres/d'instances et les méthodes d'instance, mais seulement les variables et méthodes de classe (**static** également) et les paramètres qui lui sont passés.

# Méthode statique particulière

---

Une méthode statique particulière permet d'être le point d'entrée d'exécution d'une classe qui la définit.

Il s'agit de la méthode :

```
public static void main(String[ ] args)
```

Attention à respecter scrupuleusement le prototype de cette méthode !

# Méthodes pratiques

---

## Affichage dans la console :

Pour afficher un message dans la console, il faut utiliser les méthodes

`System.out.print()` et `System.out.println()`

## Saisie de caractères :

Il est possible de saisir des valeurs dans la console pour les lire par notre programme. Pour cela, il faut utiliser le code suivant :

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

```
String str = sc.nextLine();
```

## TD03.03c-Méthodes

### Exercice "Eclairage" (3/4) :

#### 1. Dans votre classe **Lamp**

- a. Définissez une méthode **switchOn** sans paramètre dont le but est de modifier la valeur des variables **isOn** et **level**
- b. Définissez une méthode **switchOn** avec un paramètre entier dont le but est de modifier la valeur des variables **isOn** et **level**
- c. Définissez une méthode **switchOff** sans paramètre dont le but est de modifier la valeur des variables **isOn** et **level**



#### 2. Dans votre classe **Switch**

- a. Définissez une méthode **switchOnLamp1** sans paramètre qui va allumer notre **lamp1**
- b. Définissez une méthode **switchOnLamp2** avec paramètre entier qui va allumer notre **lamp2** avec un certain niveau d'intensité
- c. Définissez une méthode **switchOff** sans paramètre pour éteindre d'un seul coup nos deux lampes

# Les blocs de code

---

# Les blocs de code : C'est quoi ?

---

Un bloc de code correspond à un **ensemble d'instructions** qui vont être exécutées séquentiellement et définies au sein des accolades : `{` et `}`

En Java, les instructions doivent TOUJOURS être définies dans un bloc de code, c'est à dire :



- un constructeur,
- une méthode,
- ou un autre bloc de code.

**Seule la déclaration** des variables (de classe ou d'instance) ou des constantes peut être faite hors d'un constructeur ou d'une méthode.



# Les blocs de code : Déclaration

---



La **déclaration d'une méthode** en Java doit TOUJOURS être faite au sein d'une classe.

La **déclaration d'une variable** peut être faite :

- au coeur de la classe  
On parle de variable membre/d'instance si **non static** et de variable de classe si **static**  
La variable sera visible dans toute la classe (au sein des constructeurs ou des méthodes)
- dans un constructeur ou une méthode  
La variable NE sera visible QUE dans le bloc du constructeur ou de la méthode
- Dans un bloc de code  
La variable NE sera visible QUE dans le bloc de code

# Les variables

---

# Les variables

---

Une **variable** Java sert à contenir une valeur ou à référencer une instance de classe.

En Java, il existe 4 types de variables :

- Attribut (variable membre/d'instance)
- Variable de classe
- Variable locale
- Paramètre de méthode

Une **variable** est “forcément” **définie avec son type** lors de sa déclaration et ne pourra en changer !

La déclaration d'une variable est de la forme :

`<type> maVariable = <uneValeur | unObjet>;` ou `<type> maVariable;`



Depuis **Java 10**, il est possible de ne pas déclarer le type d'une **variable locale** en utilisant le mot réservé **var**.  
Cela s'appelle l'inférence de type.

# Les variables

---

La convention de nommage Java définit que le **nom des variables** :

- Doit être écrit en **camelCase**

Mais le compilateur ne “crie” pas si on ne respecte pas ce nommage.

```
// Noms de variables distinctes
```

```
int unevariable = 0;
```

```
int uneVariable;
```

```
int UneVariable;
```

```
int uNeVaRiAbLe = 123;
```



Rappel : Une variable définie dans un bloc ne sera “visible” que dans ce bloc.

# Variables d'instance : Attributs

---

Un **attribut** (variable d'instance/variable membre) est une variable qui définit une propriété qu'aura chaque instance de la classe.

Elle n'est pas **static**.

```
public class UnExempleDeClasse {  
    public int unAttribut;  
    private boolean estAttributPrive;  
}
```

Remarque : Leur nom est parfois préfixé par un “m” minuscule (pour “member”).

Elle a une portée qui sera définie par les mots réservés : **public**, **protected**, “package” ou **private**

# Variables de classe

---

Une **variable de classe** est une variable qui définit une propriété commune à toutes les instances de la classe.

En d'autres termes, une **variable de classe** aura une valeur unique pour toutes les instances de la classe.

Pour qu'une variable soit une variable de classe, elle doit être déclarée avec le mot réservé **static**.

```
public class UnExempleDeClasse {  
    public static int unAttributStatique;  
    private static boolean estAttributPriveStatique;  
}
```

Elle a une portée qui sera définie par les mots réservés : **public**, **protected**, "package" ou **private**

# Variables locales

Une **variable locale** est une variable définie au sein d'un constructeur, d'une méthode ou bien d'un bloc de code.

```
public class UnExempleDeClasse {  
  
    public void foo(boolean val1, short val2) {  
        int i = 0;  
        if (true) {  
            boolean b = true;  
        }  
    }  
}
```



Pas de notion de **portée** !

Ne peut être déclarée avec le mot réservé **static** !

# Variables : Paramètres de méthode/constructeur

Un **paramètre** est une variable intégrée dans la définition (prototype) des méthodes ou constructeurs.

Les méthodes/constructeurs peuvent avoir de **0 à N paramètres**.

Conseil : Ne pas excéder 10 paramètres si possible pour des raisons de lisibilité/compréhension.

```
public class UnExempleDeClasse {  
    public void foo(boolean val1, short val2) {  
        int i = 0;  
        if (true) {  
            boolean b = true;  
        }  
    }  
}
```



Pas de notion de **portée** !

Ne peut être déclarée avec le mot réservé **static** !



# Les constantes

---

Une **constante** (variable non modifiable) est définie en Java au sein de la classe avec la syntaxe :  
`<portée> static final <type> MA_CONSTANTE = <valeur>;`

On constate plusieurs choses :

- `<portée>` vaut généralement **public** ou **private**
- **static** pour indiquer qu'elle est globale à toutes les instances de la classe
- **final** pour indiquer qu'elle est non modifiable
- La **convention de nommage** des constantes est en SCREAMING\_SNAKE\_CASE (majuscule)

## TD03.03d-Variables

### Exercice “Eclairage” (4/4) :

1. Dans votre classe **Lamp**
  - a. Modifiez en privé les 2 attributs (variables membres) **isOn** et **level**
  - b. Définissez les méthodes “getter” pour récupérer leurs valeurs
2. Dans votre classe **Switch**
  - a. Définissez une méthode **switchOn** avec un paramètre de type **Lamp** et une valeur entière **level**. Écrivez du code pour allumer la lampe.
3. Exécutez votre application afin de vous assurez que votre programme fonctionne bien en affichant l'état des lampes dans la console après chaque manipulation avec l'interrupteur :
  - a. Allumer la lampe1, puis la lampe2
  - b. Eteindre la lampe1
  - c. Eteindre les 2 lampes
  - d. ...
4. Comment faire pour allumer à 50% la lampe 1 ?



- Simplifiez et optimisez votre code
- Implémentez la méthode **toString()** de la classe **Lamp**

# Commentaires

---

# Commentaires : C'est quoi ?

---

Les commentaires sont des blocs qui ne sont pas interprétés à l'exécution de la classe.

Les commentaires servent à :

- masquer du code au compilateur
- documenter le code pour en faciliter la compréhension des développeurs.

Il existe 3 types de commentaires en Java :

- Le commentaire ligne
- Le commentaire bloc
- Le commentaire **Javadoc**

# Commentaire ligne

---

Le commentaire ligne permet de commenter une seule ligne de code

Une ligne commentée doit donc commencer par //

```
// ceci est un commentaire ligne
```

# Commentaire bloc

---

Le commentaire bloc permet de commenter plusieurs lignes de code

Un commentaire bloc doit commencer par `/*` et se terminer par `*/`

```
/* ceci est un commentaire bloc  
Cette ligne là est aussi un commentaire  
Et celle-ci aussi  
*/
```

# Commentaire de documentation (Javadoc)

---

Le **commentaire Javadoc** est un commentaire bloc. Il permet donc de commenter plusieurs lignes de code.

Par contre, ce commentaire va pouvoir être interprété comme un “vrai” commentaire de documentation. En effet, la plateforme Java intègre un outil nommé **javadoc** qui permet de générer de la documentation au format HTML en interprétant justement les **commentaires Javadoc** intégrés dans le code.

Un commentaire bloc doit commencer par **/\*\*** et se terminer par **\*/**

```
/** ceci est un commentaire Javadoc  
    Cette ligne là est aussi un commentaire  
    Et celle-ci aussi  
    */
```

Un exemple de Javadoc : <https://docs.oracle.com/javase/8/docs/api/>

# Structure d'une classe Java

---



# Structure d'une classe Java : Convention

---

```
package aaa.bbb.ccc;

import a.b.T;
..
import c.d.TT;

class MyClass {

    // Définition des constantes static final

    // Définition des attributs static (=variables de classe)
    // Définition des attributs (=variables d'instance)

    // Définition des constructeurs

    // Définition des méthodes

}
```

Référence : <https://www.oracle.com/java/technologies/javase/codeconventions-fileorganization.html>

# Le mot clé **static**

---

# Le mot clé **static** : classe

---

Le mot clé **static** s'applique aux classes, attributs, méthodes et bloc de code.



On NE peut PAS le définir devant une variable locale ou un paramètre de méthode.

## Classe déclarée **static**

Déclaration principalement utilisée pour les **classes membres** (Inner Class).

```
public static class UnExempleDeClasseStatique {  
    // Code de la classe  
}
```

# Le mot clé **static** : attribut

## Attribut déclaré **static** = Variable de classe

Une variable définie comme un attribut (variable membre/d'instance) mais **static** devient une **variable de classe**.

Contrairement à un attribut (variable membre/d'instance), une variable de classe :



- est partagée entre toutes les instances d'une même classe
- n'existe qu'une seule fois en mémoire

### Un exemple de déclaration :

```
public class UnExempleDeClasse {  
    public static int unAttributStatique;  
}
```

### Un exemple d'utilisation :

```
// Pour l'utiliser, il faut l'appeler :  
UnExempleDeClasse.unAttributStatique;
```



C'est bien le **nom de la classe** et non  
d'une variable d'instance.  
Vous avez bien lu !



# Le mot clé **static** : méthode

## Méthode déclarée **static**

Une méthode définie **static** devient une méthode “globale” à la classe c’est à dire qu’elle :

- ne s’appliquera pas aux instances de la classe (objets)
- ne pourra donc pas utiliser les attributs de la classe



Une méthode **static** a la même syntaxe (portée, valeur retour, paramètres) qu’une méthode d’instance mais avec le mot clé **static** en plus.

NB : Une méthode **static** ne s’applique pas à une instance, mais à la classe !

### Un exemple de déclaration :

```
public class UnExempleDeClasse {  
    public static void maMethode() {  
    }  
}
```

### Un exemple d’utilisation :

```
// Pour l’utiliser, il faut l’appeler :  
UnExempleDeClasse.maMethode();
```



C’est bien le **nom de la classe** et non d’une variable d’instance.  
Vous avez bien lu !



# Le mot clé `static` : bloc

---

## Bloc déclaré `static`

Un bloc `static` est un bloc de code préfixé de `static`. Il doit être défini au sein de la classe, généralement entre la déclaration des attributs et des constructeurs.

Le rôle d'un bloc `static` est d'initialiser les propriétés de la classe.

Ce bloc est exécuté une seule fois à la première instanciation de la classe avant même l'appel au constructeur !!

NB : L'utilisation de bloc `static` est plutôt **déconseillée** !

```
public class UnExempleDeClasse {  
    public static int valeur = 0;  
  
    static {  
        valeur++;  
    }  
  
    public UnExempleDeClasse() {  
    }  
}
```

## TD03.04-Static

### Exercice :

1. Dans votre IDE, créez un nouveau projet Java nommé par exemple “TD Static”
2. Définissez un nom de package
3. Créez une classe **Launcher**
4. Ajoutez la méthode **public static void main(String[] args)** mais laissez-la vide
5. Créez une classe **Bike** contenant
  - a. une variable (de classe) **public static** defaultWheelCount de type **int** avec la valeur 2
  - b. un attribut **public** wheelCount de type **int**
  - c. un attribut **private** label de type **String**
  - d. un constructeur avec un paramètre de type **String** pour initialiser label
  - e. une méthode **display** qui va afficher : label, wheelCount et defaultWheelCount
6. Dans la classe **Launcher**, instanciez myBike1 de type **Bike** avec un label spécifique. Définissez wheelCount de myBike1.  
Faites de même pour myBike2 et myBike3.
7. Appelez la méthode **display** sur les 3 vélos.
8. Modifiez la valeur de l'attribut de classe defaultWheelCount
9. Appelez la méthode display sur les 3 vélos. Que constatez-vous ?

On pourra coder dans la classe **Bike** une méthode **static display()** similaire à la méthode existante non **static**



On pourra écrire des commentaires Javadoc et générer la documentation

Le mot clé **final**





# Le mot clé final

---

Le mot clé **final** sert à indiquer que l'élément qui le déclare est non modifiable.  
L'élément est donc à affectation unique.

Le mot clé **final** s'applique aux variables de classe, aux attributs (variables d'instance/membres) ou aux variables locales, aux méthodes, aux paramètres d'une méthode et aux classes.

Une **variable** de type primitif déclarée **final** ne pourra avoir sa valeur modifiée.

Une **variable d'Objet ou tableau** déclarée **final** pourra avoir ses attributs (variables membres) ou éléments modifiés, mais pas sa référence.

Une **méthode déclarée final** ne pourra être surchargée dans ses sous-classes (Cf. POO).

Une **classe déclarée final** ne pourra pas être héritée (Cf. POO).

### Exercice :

1. Dans votre IDE, créez un nouveau projet Java nommé par exemple “TD Final”
2. Définissez un nom de package
3. Créez une classe **Percent** qui aura un attribut de type **int** nommé “entier”. Il pourra être **private** (dans ce cas, ajoutez ses getter/setter **public**) ou bien **public**  
Cette classe va servir à représenter un % avec une valeur entière comprise entre 0 et 100.
4. Créez une classe **Launcher**
5. Ajouter les méthodes suivantes :
  - a. “updateInt” qui renvoie **void** et prend un paramètre de type **int**. Elle mettra à 2 la variable passée en paramètre.
  - b. Même chose avec la méthode “updateIntFinal” qui a son paramètre à **final**  
Que constatez-vous ?
  - c. “updateAndReturnInt”, idem que “updateInt”, mais renvoie la variable modifiée.
  - d. “updatePercent” qui renvoie **void** et prend un paramètre de type **Percent**. Elle mettra à 4 la valeur de l’attribut de l’objet passé en paramètre.
  - e. Même chose avec la méthode “updatePercentFinal” qui a son paramètre à **final**  
Que constatez-vous ?
6. Ajouter la méthode **public static void main(String[ ] args)**
7. Dans la méthode **main**, instancier la classe **Launcher**, déclarez une variable de type **int** à 1 et une variable de type **Percent**
8. Appelez les méthodes codées ci-dessus. **Qu’observez-vous ?**

# Opérateurs

---

# Opérateurs : C'est quoi ?

---

Les **opérateurs** sont des symboles spécifiques du langage permettant d'effectuer une opération.

Les opérateurs sont spécifiques suivant les types de données sur lesquelles effectuer l'opération.

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html>

# Opérateur d'affectation

---

L'opérateur d'affectation est le =

Cet opérateur sert à affecter/donner/définir une valeur à une variable

# Opérateurs arithmétiques

---

Les opérateurs arithmétiques sont :

- + addition pour les nombres  
Exemple :  $n = a + b$  ou  $n += a$  (équivalent à  $n = n + a$ )  
concaténation pour les chaînes de caractères (String)
- soustraction pour les nombres  
Exemple :  $n = a - b$  ou  $n -= a$  (équivalent à  $n = n - a$ )
- \* multiplication pour les nombres  
Exemple :  $n = a * b$  ou  $n *= a$  (équivalent à  $n = n * a$ )
- / division pour les nombres  
Exemple :  $n = a / b$  ou  $n /= a$  (équivalent à  $n = n / a$ )
- % le reste de la division entière pour les nombres  
Exemple :  $n = a \% b$  ou  $n \% = a$  (équivalent à  $n = n \% a$ )

# Opérateurs unaires

---

Les opérateurs unaires sont :

**+** pour définir un nombre positif (optionnel)

Exemple : `n = +3;`

**-** pour définir un nombre négatif

Exemple : `n = -2;`

**++** incrémente le nombre de 1

→ soit en pré-incrémentation (exemple `++i`) ou post-incrémentation (exemple `i++`)

**--** décrémente le nombre de 1

→ soit en pré-décrémentation (exemple `--i`) ou post-décrémentation (exemple `i--`)

**!** complément logique pour les nombres et inverse la valeur d'un booléen

Exemple : `trueBoolean = !falseBoolean`

# Opérateurs d'égalité/relation

Les opérateurs d'égalité/relation sont :

**==** pour vérifier l'égalité

Exemple : `if (a == b)`

**!=** pour vérifier l'inégalité

Exemple : `if (a != b)`

**>** pour vérifier le supérieur strict

Exemple : `if (a > b)`

**>=** pour vérifier le supérieur ou égal

Exemple : `if (a >= b)`

**<** pour vérifier l'inférieur strict

Exemple : `if (a < b)`

**<=** pour vérifier l'inférieur ou égal

Exemple : `if (a <= b)`



En Java, la comparaison d'égalité des objets ne se fait pas avec **==**

Le **==** entre 2 variables représentants des objets, teste simplement si les 2 variables référencent le même objet.

Avec certains objets, il est possible de les comparer (comparer leur contenu) avec la méthode **equals()** qui aura été implémentée.



# Opérateurs de condition

---

Les opérateurs de condition sont :

**&&** pour définir la condition ET

Exemple : `if (a == 1 && b == 2)`

**||** pour définir la condition OU

Exemple : `if (a == 1 || b == 2)`

**? ... :** pour définir la condition if-then-else (opérateur ternaire)

Exemple : `int n = (a > 0) ? a : b; // équivaut à if (a > 0) n = a; else n = b;`

# Opérateurs de comparaison de types d'objets

---

L'opérateur pour comparer des types d'objet est le mot-clé : `instanceof`

Exemple : `if (unMot instanceof String) { ... }`

Remarque : Une variable qui référence un objet `null` n'est donc pas une instance de cet objet. L'opérateur `instanceof` renvoie dans ce cas `false` !

# Opérateurs binaires

---

Les opérateurs binaires (agissant sur les bits) sont :

- ~ pour faire le complément à 2 (inversion des bits)
- << pour faire un décalage signé d'un bit vers la gauche
- >> pour faire un décalage signé d'un bit vers la droite
- >>> pour faire un décalage non signé d'un bit vers la droite
- & pour faire un ET binaire
- ^ pour faire un OU exclusif binaire
- | pour faire un OU inclusif binaire

# Opérateurs : parenthésage

---

Comme pour les expressions mathématiques, le **parenthésage** va servir à regrouper ensemble ou non des expressions logiques qui seront résolues à leur exécution.

## TD03.06-Opérateurs

### Exercice : Application Calculette

1. Dans votre IDE, créez un nouveau projet Java nommé par exemple “CalcAlphaNum”
2. Définissez un nom de package
3. Créez une classe nommée **CalcAlphaNum** dans laquelle vont être définies les opérations suivantes :
  - a. Addition entre 2 nombres entiers
  - b. Addition de 2 chaînes de caractères (concaténation)
  - c. Incrément (ou décrétement) d'un entier
  - d. Egalité de 2 nombres entiers
  - e. Egalité entre 2 chaînes de caractères
  - f. Max entre 3 nombres entiers
4. Créez une classe **Main**
5. Ajoutez la méthode **public static void main(String[] args)** qui servira à instancier ma calculette, à lancer les calculs et à afficher les résultats

# Types



---

# Types : C'est quoi ?

---

Les types définissent la nature d'une variable ou d'un objet.

Le langage Java comprend plusieurs catégories de types :

- Les types primitifs
- Les tableaux
  - de types primitifs
  - d'objets
- Les classes (définissent le type des objets)
  - standards Java 
  - issus de librairie
  - les nôtres 

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>

# Les types primitifs (1/2)

---

Les types primitifs permettent de définir la nature d'une variable.

Les types primitifs sur les nombres entiers sont :

- **byte** : entier dont la valeur est comprise entre -128 et 127. Valeur par défaut : **0**
- **short** : entier dont la valeur est comprise entre -32768 et 32767. Valeur par défaut : **0**
- **int** : entier dont la valeur est comprise entre  $-2^{31}$  et  $2^{31}-1$ . Valeur par défaut : **0**
- **long** : entier dont la valeur est comprise entre  $-2^{63}$  et  $2^{63}-1$ . Valeur par défaut : **0L**

Les types primitifs sur les nombres décimaux sont :

- **float** : décimal (encodage sur 32 bits). Valeur par défaut : **0.0f**
- **double** : décimal (encodage sur 64 bits). Valeur par défaut : **0.0d**



# Les types primitifs (2/2)

---

## Le type primitif booléen :

- **boolean** : deux valeurs possibles **true** et **false**. Valeur par défaut : **false**

## Le type primitif caractère :

- **char** : un caractère (encodage unicode sur 16 bits). Valeur par défaut : `'\u0000'`

# Les tableaux (1/4)

---

Les tableaux forment des structures de données de même type.

Un tableau permet donc de stocker **plusieurs éléments (valeurs) d'un même type**.

Un tableau est de **taille fixe** qui doit être définie à son initialisation.

Un tableau est noté avec les crochets ouvrant **[** et fermant **]**.

Les éléments d'un tableau de taille **T** ont une position (index) allant de **0** à **T-1**

La déclaration d'un tableau de **type** est de la forme : **type[ ] monTableau;** ou **type monTableau[ ];**

L'initialisation d'un tableau de **type** est de la forme : **monTableau = new type[10];**

La déclaration et l'initialisation peuvent être faites en même temps : **type[ ] monTableau = new type[10];**  
ou **type monTableau[ ] = new type[10];**

# Les tableaux (2/4)

---

Comment écrire un élément (valeur) dans un tableau ?

```
int[ ] monTableau = new int[10];
```

```
monTableau[0] = 123; ✓
```

```
monTableau[9] = 43; ✓
```

```
monTableau[-1] = 123; ✗ index incorrect
```

```
monTableau[11] = 123; ✗ index incorrect
```

```
monTableau[6] = 3.14; ✗ type incorrect
```

Comment lire un élément (valeur) dans un tableau ?

```
int[ ] monTableau = new int[10];
```

```
monTableau[4] = 123; ✓
```

```
int maValeur = monTableau[4]; ✓ => la variable maValeur vaut 123
```

# Les tableaux (3/4)

---

Déclaration simplifiée d'un tableau (initialisation explicite) :

```
int[ ] monTableau = {0, 1, 2, 3, 5, 8, 13, 21}; // Tableau de 8 éléments
```

Tableaux multidimensionnels :

Il est possible de déclarer des tableaux à plusieurs dimensions.

```
int[ ][ ] maGrille = new int[6][4]; // Tableau de 6x4=24 éléments
```

Pour définir une valeur, il faut préciser les positions (index) des 2 dimensions

```
maGrille[0][0] = 654; ✓
```

```
maGrille[5][3] = 98; ✓
```

```
maGrille[6][2] = 123; ✗ index de la dimension 1 incorrect
```

```
maGrille[0][5] = 123; ✗ index de la dimension 2 incorrect
```

# Les tableaux (4/4)

---

## Taille d'un tableau :

La taille d'un tableau peut être obtenu avec le mot clé **length**.

```
int taille = monTableau.length; // taille correspond à la taille du tableau telle que initialement définie
```

## Autres caractéristiques des tableaux :

- Les tableaux sont considérés en Java comme des objets
- Redimensionner un tableau n'est pas possible. Sa taille est définie à son initialisation
- Copier les éléments d'un tableau est possible avec la commande  
**System.arraycopy(Object src, int srcPos, Object dest, int destPos, int length)**
- D'autres opérations sont possibles sur les tableaux avec la classe **java.util.Arrays**
  - La recherche d'un élément pour en récupérer la position (index) dans le tableau
  - La comparaison de deux tableaux
  - Le remplissage "automatique" d'un tableau
  - Le tri des éléments d'un tableau

## TD03.07-Les tableaux

### Exercice : Application **Fibonacci**

1. Dans votre IDE, créez un nouveau projet Java nommé par exemple “Fibonacci”
2. Définissez un nom de package
3. Créez une classe **Start**
4. Ajouter la méthode **public static void main(String[] args)** qui servira à calculer la suite des 20 premiers nombres de Fibonacci.
5. Créez une classe **Fibonacci**, dans laquelle :
  - a. Déclarez un tableau de **int** nommé “fibonacci”
  - b. Créez une méthode **fill()** qui va alimenter le tableau
    - i. Définissez la valeur à l’index 0 du tableau à 0
    - ii. Définissez la valeur à l’index 1 du tableau à 1
    - iii. Calculez la valeur à l’index i comme la somme de l’index i-1 et de l’index i-2
    - iv. Codez une boucle pour remplir le tableau de manière automatique
  - c. Ajouter la méthode **display()** qui servira à afficher la suite de Fibonacci
6. Dans la méthode **main** de la classe **Start** :
  - a. Instanciez la classe **Fibonacci**
  - b. Appelez la méthode **fill()**
  - c. Appelez la méthode **display()**

# Les classes standards

**Java** contient plusieurs types d'objets "standards", très souvent utilisés et qu'il est important de connaître :

Type primitif	Classe Java correspondante	Usage
<b>byte</b> [-128..127]	<code>java.lang.Byte</code>	Entier signé très court (8 bits)
<b>short</b> [-32768..32767]	<code>java.lang.Short</code>	Entier signé court (16 bits)
<b>int</b> [ $-2^{31}$ .. $2^{31}-1$ ]	<code>java.lang.Integer</code>	Entier signé (32 bits)
<b>long</b> [ $-2^{63}$ .. $2^{63}-1$ ]	<code>java.lang.Long</code>	Entier signé long (64 bits)
<b>float</b>	<code>java.lang.Float</code>	Flottant (32 bits)
<b>double</b>	<code>java.lang.Double</code>	Flottant double précision (64 bits)
<b>boolean</b>	<code>java.lang.Boolean</code>	Booléen
<b>char</b>	<code>java.lang.Character</code>	Un caractère (UTF-16)

# La classe Character

---

La classe `java.lang.Character` permet de représenter un seul caractère.  
Elle est l'équivalente du type primitif `char`.

```
char c = 'Z';  
Character cc = new Character();
```

Les **apostrophes** sont utilisées pour définir le type caractère.



A ne pas confondre avec la classe `java.lang.String` qui permet de gérer les chaînes de caractères.



# La classe String (1/9)

---

La classe `java.lang.String` est la classe de base pour gérer les chaînes de caractères.

Quelques exemples :

```
String unPetitMot = new String("mais un grand soulagement");
```

```
String unLongMot = "anticonstitutionnellement";
```

```
String unePhaseDecoupee = "il " + "était " + "une " + "fois";
```

Les **guillemets** définissent automatiquement une chaîne de caractères.

NB : Les objets de type `String` ne sont pas modifiables (immuables). Modifier une `String` revient en fait à créer une nouvelle `String`.

# La classe String (2/9)

---

Remarques : D'autres classes permettent de gérer des chaînes de caractères :

- `java.lang.StringBuffer`
- `java.lang.StringBuilder`

# La classe String (3/9)

---

Il est possible d'insérer des caractères spéciaux dans les chaînes de caractères avec le caractère d'échappement `\` suivi d'une autre caractère.

Quelques caractères spéciaux	Affichage
<code>\'</code>	Single quote
<code>\"</code>	Double quote
<code>\\</code>	Backslash
<code>\n</code>	Newline
<code>\t</code>	Tab
<code>\r</code>	Carriage return
<code>\b</code>	Backspace

# La classe String (4/9)

---

La **concaténation** de chaînes de caractères se fait avec l'opérateur +

Exemple :

```
String s1 = "bon";  
String s2 = "jour";  
String s = s1 + s2; // s vaut "bonjour"
```

La **recherche** dans une chaînes de caractères se fait avec les méthodes `indexOf()` et `lastIndexOf()`

Exemple :

```
String s1 = "Une phrase avec une recherche";  
int idx1 = s1.indexOf("phra"); // idx1 vaut 4  
int idx2 = s1.lastIndexOf("che"); // idx2 vaut 26
```

# La classe String (5/9)

---

La **comparaison** de chaînes de caractères se fait avec la méthode `equals()`

Exemple :

```
String s1 = "bon";
```

```
String s2 = "BON";
```

```
boolean egal = s1.equals(s2); // egal vaut false
```



NB : Le comparateur '==' ou '!=' ne teste pas le contenu des objets pour les comparer mais leurs adresses respectives.

Comme les **String** sont des objets, la comparaison de leur contenu ne doit pas être faite avec '==' ou '!='

La **comparaison** de chaînes de caractères peut aussi se faire avec la méthode `compareTo()`

# La classe String (6/9)

---

La **détermination de la longueur** d'une chaîne de caractères est renvoyé par la méthode `length()`

Exemple :

```
String s1 = "une chaîne de caractères un peu longue";
```

```
int longueur = s1.length(); // longueur vaut 38
```

La **suppression des espaces en début et fin** de chaîne de caractères est possible grâce à la méthode `trim()`

Exemple :

```
String s1 = "   des espaces   ";
```

```
String s = s1.trim(); // s vaut "des espaces"
```

# La classe String (7/9)

---

La **modification de la casse** de chaîne de caractères est possible grâce aux méthodes `toLowerCase()` et `toUpperCase()`

Exemple :

```
String s1 = "Il était une FOIS";  
String s2 = s1.toLowerCase(); // s2 vaut "il était une fois"  
String s3 = s1.toUpperCase(); // s3 vaut "IL ETAIT UNE FOIS"
```

L'**accès aux caractères** d'une chaîne de caractères est possible grâce à la méthode `charAt()`

Exemple :

```
String s = "Une petite phrase";  
char c1 = s.charAt(0); // c1 vaut 'U'  
char c2 = s.charAt(7); // c2 vaut 'i'
```

# La classe String (8/9)

---

La **modification d'un caractère** de chaîne de caractères est possible grâce à la méthode `replace()`

Exemple :

```
String s1 = "Bonjour";
```

```
String s2 = s1.replace('o', 'a'); // s2 vaut "Banjaur"
```

L'**extraction de sous-chaînes de caractères** d'une chaîne de caractères est possible grâce à la méthode `substring()`

Exemple :

```
String s = "Une petite phrase";
```

```
String s1 = s.substring(4); // s1 vaut "petite phrase"
```

```
String s2 = s.substring(0, 3); // s2 vaut "Une"
```



# La classe String (9/9)

Depuis **Java 15**, il est possible de définir un bloc de texte dans une **String** à l'aide du triple "

Exemple :

## AVANT Java 15

```
String stringBeforeJava15 = "<html>\n" +  
    "  <body>\n" +  
    "    <p>Hello, world</p>\n" +  
    "  </body>\n" +  
    "</html>";
```

## DEPUIS Java 15

```
String stringSinceJava15 = ""  
<html>  
  <body>  
    <p>Hello, world</p>  
  </body>  
</html>  
""  
;
```

# La valeur null

---

Par défaut, à la déclaration d'un objet, s'il n'est instancié, il fera référence à la valeur **null**.

Si un objet qui n'est pas instancié essaie d'accéder à un de ses attributs ou méthodes, une erreur (Exception en Java) de type **NullPointerException** sera "levée".

```
Rectangle r; // équivalent à Rectangle r = null;  
  
r.longueur = 20; // lève un NullPointerException si r est non initialisé (null)  
  
int aire = r.calculerAire(); // lève un NullPointerException si r est non initialisé (null)
```

Depuis **Java 8**, la classe **Optional** (type paramétré) a été intégrée pour limiter ce "souci" de **NPE**. Elle permet d'encapsuler un objet afin qu'il ne soit jamais vu comme **null**, mais comme vide. Principalement à utiliser comme type de retour des méthodes (sauf tableau ou liste).

# Les conversions de Types (1/4)

---

Il est possible selon les types de **convertir un type en un autre**.

C'est ce qu'on appelle le cast ou le transtypage. Cela permet de “forcer” le type des variables.

Le cast est défini par un **type entre parenthèses** (type résultat) pour indiquer de convertir la variable d'un certain type dans ce type résultat.

Exemple :

```
int entier = 5;
```

```
float flottant = (float) entier;
```

Cette conversion n'est bien sûr pas toujours possible car incompatible.

Exemple : `int entier = (int) “UnDeuxTrois”;`

# Les conversions de Types (2/4)

---

La conversion des types primitifs en leurs classes équivalentes et vice-versa est automatique (depuis **Java 5**).

Cela s'appelle l'**autoboxing** (conversion de type primitif en classe) et l'**unboxing** (conversion de classe en type primitif).

Exemple :

```
int a = 1;
```

```
Integer integerA = a; // OK sans cast (autoboxing)
```

```
Integer integerB = new Integer(8);
```

```
int b = integerB; // OK sans cast (unboxing)
```

# Les conversions de Types (3/4)

---

Le **cast** est prioritaire sur les autres opérateurs.



La conversion peut entraîner une **perte d'informations**.

Petit test pour illustrer cela :

```
int a = 5;
```

```
int b = 4;
```

Que vaut :

- `(double)(a/b) ?`
- `(double)a/b ?`
- `(int)((double)a/b) ?`

# Les conversions de Types (4/4)

---

Des méthodes définies sur des objets permettent aussi de réaliser des conversions.

## Exemple 1 :

```
String monTexte = new String("10");  
Integer monNombre = new Integer(monTexte);  
int i = monNombre.intValue(); // méthode intValue de la classe Integer
```

## Exemple 2 :

```
int i = 10;  
Integer monEntier = new Integer(i);  
long j = monEntier.longValue(); // méthode longValue de la classe Integer
```

# Instructions

---

# Instructions : C'est quoi ?

---

Les **instructions** sont des commandes prédéfinies du langage permettant de **contrôler l'exécution du code** Java.

Deux types d'**instructions** existent en Java :

- Les instructions conditionnelles
- Les instructions d'itération

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html>



# Instructions conditionnelles

---

# Instructions conditionnelles

---

Le langage Java possède **2 types d'instructions conditionnelles** :

- if...else if...else
- switch...case

# Condition : if...else if...else

---

## if...else if...else

La syntaxe de cette condition est en Java :

```
if (condition1) {  
    // code exécuté si condition1 est vraie  
}  
else if (condition2) {  
    // code exécuté si condition1 est fausse mais condition2 est vraie  
}  
else {  
    // code exécuté si condition1 est fausse et condition2 est fausse  
}
```

### Remarques :

La *condition* peut être issue du résultat de plusieurs conditions

Le bloc **else** est optionnel mais est le dernier bloc de la condition

Des blocs **else if** peuvent être définis entre le bloc **if** et le bloc **else**

# Condition : switch...case

## switch...case

La syntaxe de cette condition est en Java :

```
switch (variable) {  
    case valeur1 :  
        // code exécuté si la variable vaut valeur1  
        break;  
    case valeur2 :  
        // code exécuté si la variable vaut valeur2  
        break;  
    default :  
        // code exécuté sinon  
        break;  
}
```

La *variable* ne peut être que de type :

- **byte**, **short**, **char** et **int**
- **Byte**, **Short**, **Character** et **Integer**

Depuis **Java 5**, d'autres types sont possibles :

- **String**
- **Enum** (énumération)



Mais les types primitifs **boolean**, **long**, **double** et **float** ne sont pas valides.

Le bloc **default** est optionnel

Chaque **case...break** définit un bloc de code (les accolades sont inutiles)

# Condition : switch...case

## AVANT Java 14

```
switch(j) {  
    case LUNDI: faireLesCours();  
    case MERCREDI: sortirLesPoubelles(); break;  
    case MARDI: entraînementFoot(); break;  
    case VENDREDI: peuPasJaiPiscine(); break;  
    case SAMEDI:  
        weekend();  
        break;  
    default:  
        System.out.println("Rien a faire le jeudi !");  
}
```

## DEPUIS Java 14

```
switch(j) {  
    case LUNDI, MERCREDI -> sortirLesPoubelles();  
    case LUNDI -> faireLesCours();  
    case MARDI -> entraînementFoot();  
    case VENDREDI -> peuPasJaiPiscine();  
    case SAMEDI, DIMANCHE -> weekend();  
    default  
        -> System.out.println("Rien a faire le jeudi !");  
}
```

### Les évolutions :

- Nouvelle syntaxe : -> remplace le : et disparition du **break**;
- Plusieurs constantes dans un **case**
- Le **switch** devient une expression qui peut être affectée à une variable
- Pour renvoyer un résultat d'un **case**, utiliser le mot réservé **yield**

Preview Java 12

Preview Java 13

## TD03.08-Instructions conditionnelles

### Exercice : Application **TirageAuSort**

1. Dans votre IDE, créez un nouveau projet Java nommé par exemple “TirageAuSort”
2. Définissez un nom de package
3. Créez une classe nommée **Dice** qui aura une méthode pour lancer le dé et renvoyer la face
  - a. Définissez 1 constante pour la valeur max du dé à 6 faces
  - b. Créez une méthode **roll()** qui va utiliser la méthode **java.util.Random.nextInt(int)** pour renvoyer un nombre aléatoire correspondant à la face du dé
4. Créez une autre classe nommée **Draws** dans laquelle vous allez coder 100 tirages au sort successifs d'un dé, mémoriser les résultats et les afficher.
  - a. Définissez 1 constante pour le nombre total de tirages
  - b. Créez 1 attribut par face du dé pour comptabiliser les tirages (privilégier un tableau plutôt que 6 attributs comme compteurFace1, ..., compteurFace6)
  - c. Créez une méthode **rollDraws()** pour exécuter 100 tirages successifs en instanciant un dé et appeler sa méthode **roll()**. Comptabilisez les résultats des tirages.
    - i. Implémentez la condition à l'aide d'un “if ... else if .... else ...”
    - ii. Implémentez la condition à l'aide d'un “switch .... case ...”
    - iii. Une autre solution encore plus simple ?
  - d. Créez une méthode **displayDraws()** pour afficher les résultats des compteurs
5. Créez une classe **Starter**
6. Ajouter la méthode **public static void main(String[] args)** qui servira à instancier notre tirage au sort, lancer les tirages et à afficher les résultats des tirages dans la console

# Instructions d'itération

---

# Instructions d'itération

---

Le langage Java possède **3 types d'instructions d'itération** (boucles) :

- while
- do...while
- for



# Itération : while

---

## while

```
while (condition) {  
    // code exécuté tant que la condition est vraie  
}
```

### Remarque :

La *condition* peut être issue du résultat de plusieurs conditions

# Itération : do...while

---

## do...while

```
do {  
    // code exécuté et réexécuté tant que la condition est vraie  
} while (condition);
```

### Remarques :

La *condition* peut être issue du résultat de plusieurs conditions

Contrairement à la boucle 'while', le bloc 'do...while' exécute au moins une fois le code qu'il contient

# Itération : for

## for

```
for (initialisation; terminaison; incrémentation) {  
    // code exécuté tant que la terminaison n'est pas atteinte  
}
```

Depuis **Java 5**

```
for (Type variable : Collection/Array) {  
    // code exécuté tant que tous les éléments de la collection n'ont  
    pas été parcourus  
}
```

### Remarques :

L'*initialisation* permet généralement de déclarer et d'initialiser une variable

La *terminaison* définit une condition de continuation de la boucle

L'*incrémentation* définit comment faire évoluer la variable

### Remarque :

Le *Type* est le type de chaque élément de la *Collection* ou de l'*Array*

# Instructions d'itération

---

Les instructions d'itération peuvent être stoppées avant la fin logique de leur exécution à l'aide du mot réservé **break**

Le mot clé **break** permet de quitter immédiatement une itération ou une condition

Quant au mot clé **continue**, il permet de sauter une itération pour passer à la suivante

## TD03.09-Instructions d'itération

### Exercice : Application **Suite de Padovan**

1. Dans votre IDE, créez un nouveau projet Java nommé par exemple “SuiteDePadovan”
2. Définissez un nom de package
3. Créez une classe **Launcher**
4. Ajoutez la méthode **public static void main(String[] args)** qui servira à lancer le calcul de la suite des 20 premiers nombres de la suite de Padovan de 3 manières différentes
5. Créez une classe **Padovan** avec un attribut un tableau de **int** nommé “nombres” qui va contenir les nombres de Padovan et une constante **MAX**
6. Créez une méthode **padovanWhile()** qui va calculer la suite à l'aide d'un boucle **while**
7. Créez une méthode **padovanDoWhile()** qui va calculer la suite à l'aide d'un boucle **do...while**
8. Créez une méthode **padovanFor()** qui va calculer la suite à l'aide d'un boucle **for**
9. Dans les 3 méthodes :
  - a. Définissez la valeur à l'index 0 du tableau à 1
  - b. Définissez la valeur à l'index 1 du tableau à 1
  - c. Définissez la valeur à l'index 2 du tableau à 1
  - d. Calculez la valeur à l'index  $i$  comme la somme de l'index  $i-2$  et de l'index  $i-3$
  - e. Remplissez le tableau
10. Ajoutez la méthode **display()** qui servira à afficher la suite de Padovan

# Questions/Réponses

---