# ***The Java Object-Oriented Approach***

(Java SE 11 Developer Certification 1Z0-819)

## Table des matières

# I. Introduction to OOP: Classes and Objects :

## • *Objects :*

**Définition d'un objet** :

| |
|---|
| C'est une construction logicielle qui modélise des concepts du monde réel. Ce sont des instances de classe. |

## • *Classes :*

**Définition** :

| |
|---|
| Ce sont des prototypes ou des « **plans** » à partir desquels ont peu instancier des objets. Ces plans contiennent toutes les propriété et méthodes qui sont communes à tous les objets qui en découleront. |

Pour définir une classe en Java, ont utilise le mot clef **CLASS**. Dans l'exemple, on définit une classe avec ses **attributs** (les propriétés de l'objet) et des **méthodes** (qui sont ce que l'objet peut faire) qui sont visibles car elles sont suivies de parenthèses et d'accolades.

```
public class FlightPlan {
        String id;
        String departure;
        String destination; //ici trois attributs

        Void start(){   //ici une méthode
        }
}
```

## • *Understanding Constructors :*

Le constructeur : C'est une méthode particulière qui permet d'instancier un objet à partir d'une classe.

- Ces méthodes ont obligatoirement le même nom que leur classe et n'ont pas de type de retour.
- Ils existent même s'ils ne sont pas expressément déclarés, car la création d'une classe induit la création d'un constructeur invisible par défaut.
- Ont peut en créer plusieurs différents par classes.

- On peut les ajouter avec des modificateurs d'accès. (Grace à l'utilisation des mot-clefs PUBLIC, PRIVATE, etc…).

Ici le constructeur à un corps VIDE mais on voit qu'il a le même nom que sa classe :

```
public class avion {
    int vitesse;
    int altidudeMax;

    public avion(){        //Constructeur
    }
}
```

Dans l'exemple suivant on peut voir que j'ajoute au constructeur un paramètre (ce qui se trouve entre les parenthèses) qui va me permettre de définir la vitesse de l'avion dès que je vais instancier l'objet. Le mot clef **this** permet de faire référence à l'attribut de classe. Cela permet de ne pas le confondre avec le paramètre car les deux ont le même nom.

```
public class avion {
    int vitesse;
    int altidudeMax;

   public avion(int vitesste ){
   this.vitesse=vitesste;
   }
}
```

Il faut être attentif à une chose dans le cas précédent, je ne peux plus faire appel au constructeur par défaut si j'en défini un moi-même dans la classe. On peut voir du coup que la première ligne de l'exemple suivant devient invalide car dans l'exemple précédent j'ai créé un constructeur qui nécessite qu'on lui indique la vitesse en paramètre :

```
Avion airbus = new Avion(); //PAS VALIDE

Avion airbus = new Avion(800); //VALIDE
```

**La surcharge** (*overload*) :

Cela signifie que l'on peut écrire plusieurs constructeurs différents au sein d'une même classe. L'intérêt est de prévoir des cas divers de création d'objet.

Dans l'exemple suivant j'ai deux constructeurs. L'un nécessite que l'on donne la vitesse et le nombre de passagers et l'autre juste la vitesse. Cela me permet de créer l'objet, même si je n'ai pas encore toutes les informations disponibles :

```
public class Avion {
    int vitesse;
    int passagers;

    public Avion(int vitesse) {  //constructeur 1
        this.vitesse = vitesse;
    }
    public Avion(int vitesse, int passagers) { //constructeur 2
        this.vitesse = vitesse;
        this.passagers = passagers;
    }
}
```

**Les blocs d'initialisations** :

- **Non-static (instance initializer) :**

Ce sont des blocs qui sont délimités par des accolades {} à l'intérieur d'une classe et qui permettent d'initialiser des attributs en leur donnant une valeur.

```
public class Avion {
    int vitesse;
    String nom;
    int passagers;

    {
        passagers=150;
        System.out.println("le nombre de passager est de "+passagers);
    }
}
```

S'il est vrai qu'on peut donner une valeur à un attribut simplement dans sa définition de départ, le bloc d'initialisation a le petit avantage de pouvoir implémenter des méthodes complexes comme des boucles ou des *if* dès le **départ**. Petite précision quand même, le compiler JAVA invoque d'abord le constructeur **PUIS** le bloc d'initialisation même si cela n'apparaît pas évident.

- **Static (class initializer)** :

Idem sauf que le bloc comprend le mot clef static et qu'il est appelé **AVANT** la méthode MAIN. Un des intérêts est de pouvoir procéder à des opérations avant le chargement d'une classe.

```
public class Foo {

    //instance variable initializer
    String s = "abc";

    //constructor
    public Foo() {
        System.out.println("constructor called");
    }

    //static initializer
    static {
        System.out.println("static initializer
called");
    }

    //instance initializer
    {
        System.out.println("instance initializer
called");
    }

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

Output:

```
static initializer called
instance initializer called
constructor called
instance initializer called
constructor called
```

Ici par exemple on voit deux choses. J'ai instancié deux objets de la classe **FOO**. On voit qu'en sortie (*output*), le *static initializer* ne sera appelé qu'une seule fois, tandis que le constructeur et l'*instance initializer* le seront deux fois, car une pour chaque objet créé.

Pour plus de précisions : https://javagoal.com/static-block-in-java/

- ***Instantiating Objects***

Il y'a trois étapes/parties à l'instanciation d'un objet :

```
Avion airbus = new Avion();
```

- **La déclaration** : On donne le nom et le type de la variable. Ici : Avion (le type) airbus (le nom). Cela créera une variable qui contiendra une référence au type Avion.
- **L'instanciation** : Elle se fait grâce au mot clef **NEW**. Cet opérateur alloue de la mémoire pour l'objet Avion et retourne une référence pour cette adresse mémoire.
- **L'initialisation :** C'est ce qui suit le new, ici ce sera Avion dont le constructeur sera appelé. Il attribuera des valeurs aux différents attributs.

On peut donc tout à fait créer plusieurs variables qui pointent vers le même objet :

```
Avion airbus = new Avion();
Avion autreAirbus= airbus;
```

La variable autreAirbus pointera alors vers la même référence mémoire.

- ***Destroying Objects :***

Le nettoyage ou la « destruction » d'objet en JAVA se fait grâce ***au Garbage collector*** qui est automatique mais dont l'exécution peut être configurée.

Il contient usuellement trois phases :

1. **Marquage** : identifie les objets qui sont utilisés ou non.
2. **Suppression** : retire les objets qui sont non-référencés, ce, pour libérer de la mémoire.
3. **Compactage** : (étape optionnelle) regroupe les objets restants.

Illustration :


On a ici des objets qui occupent un espace mémoire (en orange). Il reste de l'espace libre en gris.


Ici le *garbage collector* trouve des objets qui sont non-référencés. (Etape 1).

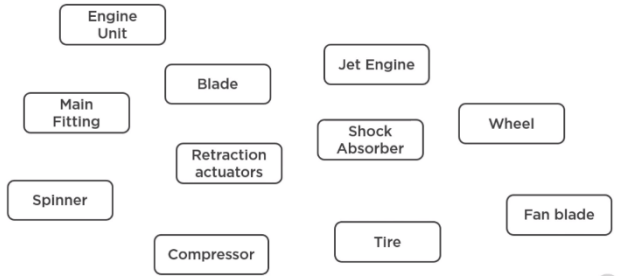
Le *garbage collector* les supprime. (Etape 2).

Le *garbage collector* regroupe (compact) les objets restants. (Etape 3).

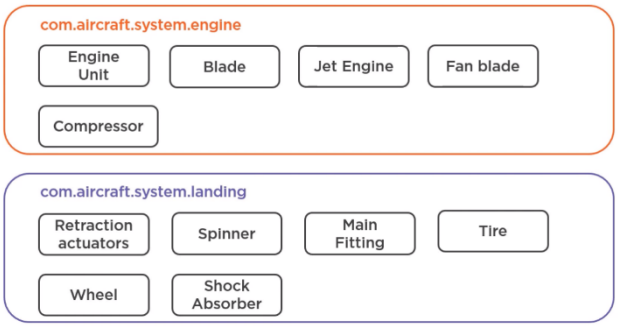
- ***Organizing Classes with Packages :***

**Définition du package** :

| C'est une « *namespace* » qui permet de regrouper des classes ou des interfaces qui ont des liens entre-elles. |
|---|




Imaginons que nous écrivions un code mais qu'on ne regroupe pas les classes dans différents package, ont obtiendrait ceci en vue générale.




A l'inverse ici on regroupe les classes dans deux différents packages : engine et landing, ce qui améliore la lisibilité et la compréhension générale des interactions entre les classes.

# II. Modeling State and Behavior :

- ***Variables :***

Selon leur emplacement dans le code, les variables peuvent être classées en 4 catégories :

- **Variable d'instance** : ce sont là où les objets stockent leur état. Elles sont appelées variable d'instance car elles sont uniques à chaque instance d'une classe. Ce sont elles qui sont accessibles et modifiables par les méthodes get/set. (ndr, Attribut, champ et variable selon certains documents sont des synonymes, mais la dénomination la plus rencontrée est celle de variable).

Ici un exemple de classe *Avion* avec trois variable d'instance : *vitesse*, *nom* et *passager* :

```
public class Avion {
    int vitesse;
    String nom;
    private int passagers;
}
```

- **Variable de classe :** Ce sont les champs (variable) définis avec un modificateur **STATIC** et optionnellement **FINAL**. Ils sont **uniques** à la classe plutôt qu'à l'instance. Dans le cas suivant c'est la vitesse maximale théorique qui appartient à cette catégorie :

```
public class Avion {
    int vitesse;
    String nom;
    int passagers;
    final static int vitesseMaxTheorique = 980;
}
```

- **Variable locale** : Ce sont celles qui sont déclarées à l'intérieur de méthode et ne sont visible **qu'à l'intérieur** de celle-ci.

- **Paramètre** : Ce sont les variables déclarées dans la signature des méthodes. Ici la méthode décollage prend un paramètre de vitesse.

```
public void decollage (int vitesseDecollage){
    System.out.println("L'avion décolle et sa vitesse est de "+
vitesseDecollage+" km/h");
}
```

- ***Methods :***

Il y' a 6 parties qui peuvent constituer une déclaration de méthode :

1. Le modificateur d'accès : Ici public.
2. Le type de retour : Ici void (mais peut-être une valeur ou un objet).
3. Le nom de la méthode : Ici décollage.
4. Les paramètres : Ici int vitesseDecollage
5. Les exceptions : Ici throws IOException
6. Le corps : Ce qui se trouve entre les accolades.

Le retour, le type et le nom sont les 3 parties **OBLIGATOIRES**.

```
public void decollage (int vitesseDecollage) throws IOException {
    System.out.println("L'avion décolle et sa vitesse est de "+
vitesseDecollage+" km/h");
}
```

La combinaison du nom de la méthode et de ses paramètres constitue la **SIGNATURE** de méthode : *decollage (int vitesseDecollage).*

**La surcharge de méthode** (*method overloading*) permet d'avoir plusieurs méthodes avec le **MÊME** nom **SI** les paramètres diffèrent.

```java
public double calculateRange() {
    // Assume current aircraft speed and fuel consumption
}

public double calculateRange(double speed) {
    // Provide speed, calculate new fuel consumption and return result
}

public double calculateRange(int speed) {
    // Provide speed as integer
}
```

Ici la première méthode n'a pas de paramètre. Elle est surchargée deux fois car on change ses paramètres. Il suffit d'ailleurs de changer le type du paramètre pour que cela soit valable.

Si on n'avait changé que le nom du paramètre de la deuxième méthode sans changer le type, cela n'aurait pas fonctionné car ils seraient considérés comme ayant la même signature.

- ***Passing information to a Method :***

Principe général :

| On doit passer par les paramètres pour transmettre des informations à une méthode. |
|---|

Problème : les noms utilisés doivent être uniques, donc :

- Deux paramètres ne peuvent pas avoir le même nom.

```java
public void logFuelCapacity (String file, File file)  {

}
On ne peut pas déclarer une variable locale qui a le même nom qu'un
paramètre.
public void logFuelCapacity (String file, int remainingFuel)  {
    int remainingFuel = 400;
}
```

**MAIS** : un paramètre, ou une variable locale, peut avoir le même nom qu'une variable de classe et du coup changer sa valeur, c'est ce qu'on appelle **l'ombrage de champ (FIELD SHADOWING).** Dans l'exemple suivant on peut dire que le paramètre vitesse « fait de l'ombre » (*is shadowing*) à la variable de classe vitesse mais le mot clef **THIS** nous permet de lever toute ambiguïté pour la distinction des deux (this empêche le *field shadowing*). Ce cas est couramment utilisé dans les constructeurs.

```java
public class Avion {
    int vitesse;

    public Avion(int vitesse) {
        this.vitesse = vitesse;
    }
```

Différence entre argument et paramètre :

- Les paramètres sont des variables définies dans une déclaration de méthode (Ici int a, int b).

- Les arguments sont les valeurs réelles transmises à la méthode (Ici 1,2).

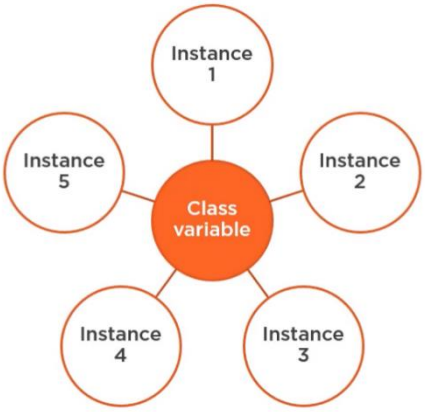
```
// a and b are parameters, a variable defined in
// the method declaration
private void swap(int a, int b) {
    int aux = a; a = b; b = aux;
}

// 1 and 2 are arguments, they are the value passed
// into the method
swap(1, 2);
```

# III. Understanding Static Fields, Methods ans classes :

Le modificateur **STATIC** signifie que quelque chose est directement lié à une classe et non à une instance de celle-ci. Il peut être utilisé pour modifier :

- Des variables.
- Des méthodes.
- Des blocks d'initialisation.
- Des classes imbriquées.

## • *Static Fields :*




Ce sont les variables de classes. Chaque instance d'une classe partagera cette variable si elle existe.

Si elle est **PUBLIC**, on peut y accéder simplement en utilisant le nom de classe suivi du nom de la variable :

```
public class Avion {
    public static int vitesse=400;

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

    System.out.println(Avion.vitesse);
}
```

Il est tout à fait possible de n'utiliser que le mot vitesse donc sans le nom de la classe en faisant un import de la variable. Donc la différence ici est que je n'ai pas besoin du préfixe de classe « *Avion* » mais juste « *vitesse* ».

```
import static object.Avion.vitesse;

public static void main(String[] args) {

    System.out.println(vitesse);
}
```

Les variables de classes peuvent aussi être constantes grâce au mot clef **FINAL**. Dans ce cas la convention veut que l'on nomme la variable en majuscule séparée par des *underscore* si nécessaire.

Exemple : static final int **SPEED_OF_SOUND** =343 ;

- ***Static Methods :***

Le principe est similaire à celui de la variable de classe, on peut faire appel à une méthode *static* sans devoir créer une instance de classe.

Mis à part le mot clef **STATIC**, elles ne diffèrent pas tellement des méthodes usuelles. Elles sont souvent créées pour être des méthodes d'aide ou utilitaires.

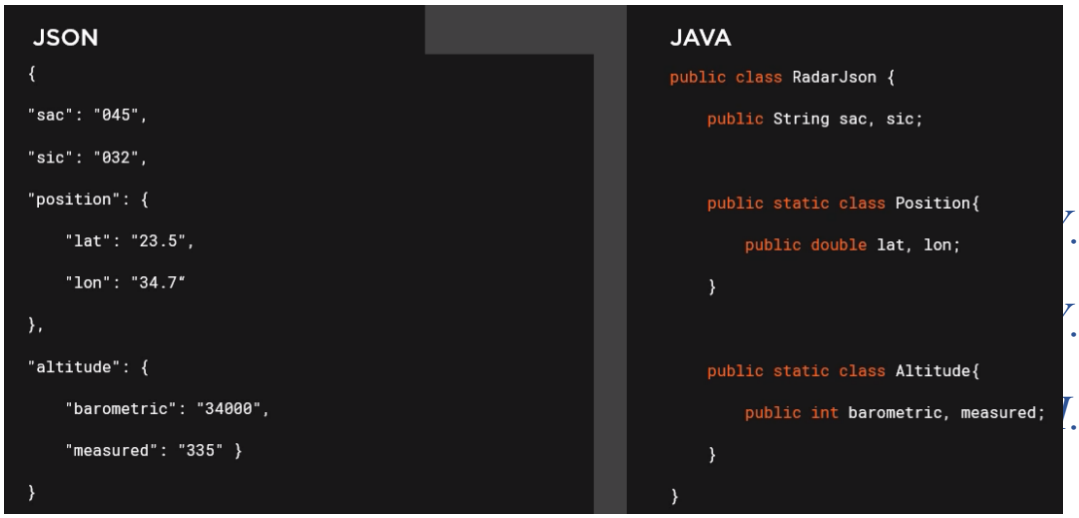
On les invoque de la même manière que les variables de classes, c'est-à-dire avec le nom de la classe suivie du nom de la méthode.

- ***Static Nested Classes :***

Il s'agit ici d'une classe déclarée à l'intérieur d'une autre et auquel on ajoute le modificateur **STATIC**.

Cela peut être utile dans la mesure où cette sous-classe ne doit être utilisée que par la classe dans laquelle elle est définie. Il semblerait que cela soit souvent utilisée dans la sérialisation d'objet (dans le cas d'utilisation de Json par exemple).

Dans l'exemple qui suit on peut voir que les *nested* classes servent à regrouper différentes catégories d'information qui appartiennent au fichier Json et qu'on va regrouper dans deux *nested classes* : position et altitude.

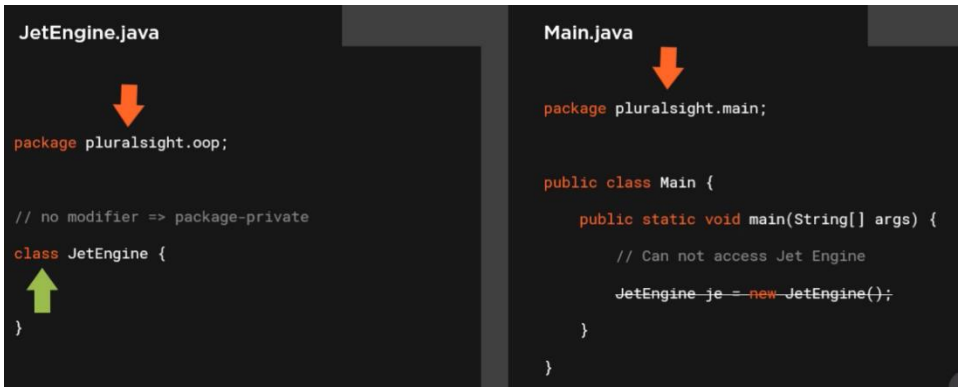



# *VII. Using Encapsulation and Inheritance :*

- ***Access Modifiers :***

Ils spécifient si une classe ou ses membres peuvent être accessible par une autre classe. Il y'a deux grands niveaux d'accès :

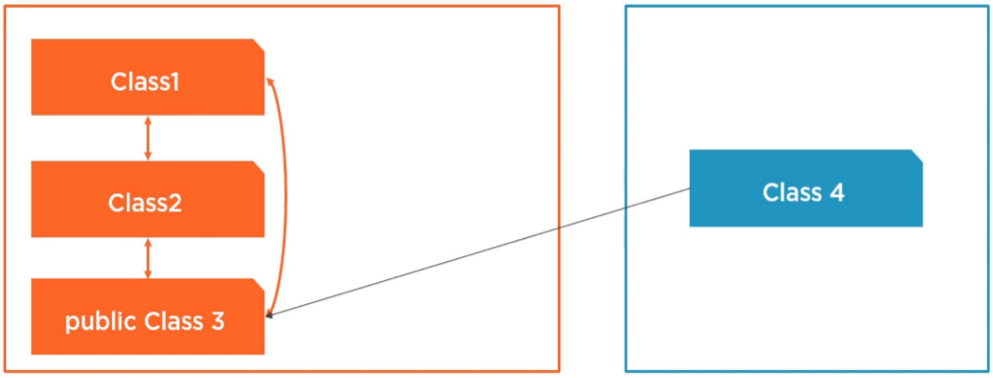
A. *Top level* : au niveau de la classe. Il y'a deux modificateur d'accès disponibles :
   1. **Package Private** : la classe n'est visible que par les classes du même package. C'est le modificateur d'accès **PAR DEFAUT** lorsqu'on en défini aucun.
   2. **Public** : la classe est visible partout.




Dans cet exemple la classe *JetEngine* ne sera pas accessible car elle ne fait pas partie du même pack que la classe *main* et elle est par défaut en privé.

Ici on voit que les classes en orange peuvent accéder les unes aux autres car elles sont dans le même package, quel que soit leur modificateur d'accès.

Par contre dans un autre package, la classe 4 aura à accès **uniquement** à la classe 3 car son modificateur d'accès est **PUBLIC**.




B. *Membre level* : au niveau de la variable ou de la méthode. Il en existe 4 :

1. **Public** : idem que pour la classe (accessible partout).
2. **Protected** : visible que par les classes de son package mais elle peut être héritée par une sous-classe (classe qui hérite) de sa classe qui est définie dans un package différent.
3. **Package Private** (celle par défaut, donc si rien de spécifié) : pareil que pour les classes (même package).
4. **Private** : visible uniquement à l'intérieur de sa propre classe.

**Récapitulatif** :

| | Class | Package | Subclass | World |
|---|---|---|---|---|
| public | Y | Y | Y | Y |
| protected | Y | Y | Y | N |
| no modifier | Y | Y | N | N |
| private | Y | N | N | N |

- ***Encapsulation :***

**Définition** :

L'encapsulation se crée sur la base des modificateurs d'accès. C'est le principe de regroupement des données avec les méthodes qui opèrent sur ces données dans une classe.

Cela permet deux choses :

- Cacher l'information ou les données du monde extérieur.
- Cacher les modes de fonctionnement interne et les détails d'implémentation.

```
public class Aircraft {

    public int altitude;

    public int speed;

}


a.altitude = 0;

a.speed = 0;
```

Ici si je décide de créer une classe *Avion* avec deux variables d'instance : *l'altitude* et la *vitesse*. Ces deux variables sont accessibles à tout le monde, et donc peuvent être changées n'importe où et par n'importe qui, ce qui n'est pas une pratique sécurisée.

Donc pour une pratique plus sécurisée je modifie cette classe en mettant les variables en *private* ce qui permettra de ne pas pouvoir modifier *l'altitude* et la *vitesse* n'importe où. Le principe général veut qu'il ne faille pas exposer plus d'information que nécessaire.

```
public class Aircraft {

    private int altitude, speed;

    public void land() {

        // Gradually decrease speed and altitude safely

    }

}

a.land(); // internal details are hidden, you only have access to public API
```

Afin de permettre un certain contrôle sur l'accès aux variables d'instances on va alors utiliser des *getters* et des *setters* :

```
public class Avion {
    private int vitesse;

    public int getVitesse() {
        return vitesse;
    }

    public void setVitesse(int vitesse) {
        this.vitesse = vitesse;
    }
}
```

- ***Inheritance (l'héritage):***

Définition :

| C'est une technique qui permet de définir une nouvelle classe en la basant sur une classe déjà existante. Elle en acquiert du coup les caractéristiques et les comportements. |
|---|

Par défaut et manière non visible toutes les classes héritent de la classe de base OBJECT de Java.

La classe Object possède quelques méthodes qui seront du coup, universelles à toutes les classes :

- toString()
- getClasse()
- hashCode()
- equals()
- clone()

Les classes qui héritent d'une autre, sont appelées **SUBCLASSE** en anglais. Elles ont le mot clef **EXTENDS** suivi du nom de la classe dont elles héritent. Ici on fait une classe principale *Animal* avec une variable d'instance *name* :

```
public class Animal {
    String name;

}
```

Ici on fait hériter la classe *dog* de la classe *Animal*, comme ça on récupère les caractéristiques et les comportements :

```
public class Dog extends Animal{
}
```

On peut aussi bien redéfinir (*override*), ou définir de nouvelles méthodes et caractéristiques.

**L'OVERRIDING** donne la possibilité de modifier une méthode existante de sa classe supérieure à condition qu'elle contienne :

- Le même nom.
- Le même nombre et type de paramètre.
- Le même type de retour.
- Que la méthode à remplacer ne soit pas **FINAL**.

| Attention à ne pas confondre *l'overriding* (la redéfinition) et *l'overloading* (la surcharge) !! |
|---|

Ici une classe hélicoptère hérite de la classe Aircraft et redéfini (*override*) la méthode *land()* :

```java
public class Aircraft {

    public void land() {

        System.out.println("Aircraft landing");

    }

}
```

```java
public class Helicopter extends Aircraft {

    @Override

    public void land() {

        System.out.println("Helicopter
landing");

    }

}
```

Le mot clé **SUPER** : il permet d'accéder aux méthodes ou au constructeur de la classe supérieure et d'en récupérer le résultat. Une des utilités est qu'en cas *d'overriding* de cumuler le résultat de la classe supérieur à celui de la classe qui hérite. Donc ici je créé une méthode chez l'animal qui affichera « l'animal fait un bruit : » :

```java
public class Animal {
     String name;
    public void faitBruit(){
        System.out.println("L'animal fait un bruit :");
    }
}
```

Ensuite je fais hériter la classe *Animal* par la classe *chien* mais je récupère le résultat « l'animal fait un bruit » grâce à *super.faitBruit()* ;

```java
public class Dog extends Animal{
    @Override
    public void faitBruit(){
        super.faitBruit();
        System.out.println("Wouf");
    }
}
```

J'aurais comme résultat la combinaison des deux : « *L'animal fait un bruit :* », « *Wouf* ».

**SUPER** fonctionne comme le **THIS** mais pour la classe supérieure.

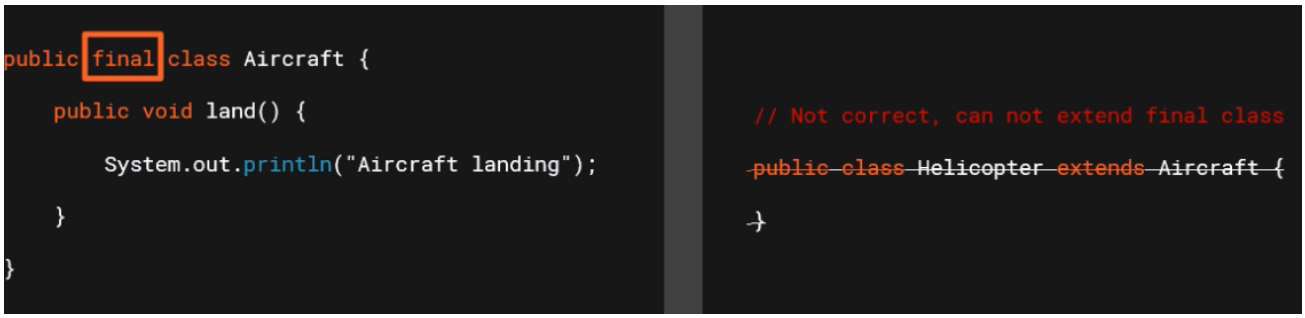
On peut utiliser cette fonction avec un constructeur. Par exemple je défini une classe *animal* uniquement avec une variable *nom* :

```java
public class Animal {
    private String name;

    public Animal(String name) {
        this.name = name;
    }
}
```

Ensuite, je crée une classe *Dog* qui hérite de la classe *Animal* mais comme dans la classe d'origine, je veux garder un constructeur avec le nom et en plus je veux ajouter un nouveau paramètre. Du coup j'utilise *super* pour réutiliser le constructeur d'origine dans le constructeur de la classe :

```
public class Dog extends Animal{
    String race;

    public Dog(String name, String race) {
        super(name);
        this.race = race;
    }
}
```

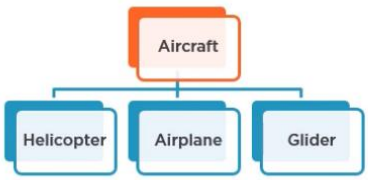
A noter qu'une classe avec le mot clef **FINAL** ne peut pas être héritée !!




- ***Abstract Classes and Methods :***

La classe abstraite (*abstract class*) est une classe qui ne **PEUT PAS** être instanciée. Elle est conçue pour être **HERITEE** et utilisée comme base pour une classe plus concrète. On la définit en utilisant mot clef **ABSTRACT** devant le mot clef CLASS.

Elle est déclarée sans **AUCUNE** méthode implémentée (les méthodes n'ont donc pas de corps). Les sous-classes doivent les remplacer et les compléter si nécessaire. Leur utilité est de fournir des sorte de *plans* (*blueprint*) pour les sous classes qui en hériteront.




It would be very unrealistic to expect the same landing procedure for all subclasses

However, they need to implement this action, but in their own way

An abstract class can be used to define a skeleton, or baseline. We abstract, but we leave freedom to subclasses to come with their own behavior in a consistent way.

Par exemple on sait que ces trois appareils doivent tous être capable d'atterrir. Comme l'atterrissage ne se fait pas de la même manière pour les trois ont leur fourni une méthode qui est **COMMUNE** mais suffisamment abstraite pour qu'ils puissent la faire correspondre à leurs besoins propres.

Dès lors, lorsqu'on définit une classe abstraite Aircraft, on peut le faire comme dans cet exemple :

On voit que la méthode *land()* n'a aucun corps (ce qui est contenu entre les accolades{}). Les sous classes qui héritent seront alors **OBLIGEES** de définir le corps de ces méthodes.

```
public abstract class Aircraft {
    private int altitude;

    // Too complex to be abstracted
    public abstract void land();

    public int getAltitude() {
        return this.altitude;
    }
}
```

# VIII. Understanding Intefaces and Polymorphisme :

- ***Interfaces :***

**Définition** :

| Une interface est un TYPE de REFERENCE (donc ne pas dire que c'est une classe !!) |
|---|

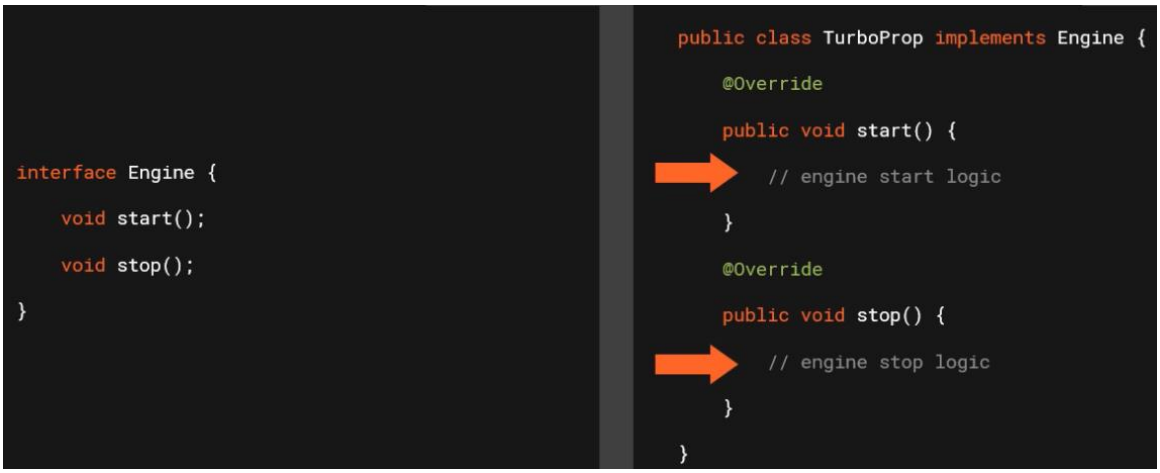
Elle peut contenir :

- Des signatures de méthode.
- Des méthodes par défaut.
- Des méthodes statiques.
- Des types imbriqués (*nested types*).
- Des constantes.

Elles ne peuvent pas :

- Être instanciée.
- Contenir un constructeur.
- D'habitude les méthodes n'ont pas de corps (à l'exception des méthodes statique et *default*).

Un des grands avantages est qu'une classe peut implémenter une ou **PLUSIEURS** interfaces.



```java
interface Engine {
    void start();
    void stop();
}
```

```java
public class TurboProp implements Engine {
    @Override
    public void start() {
        // engine start logic
    }
    @Override
    public void stop() {
        // engine stop logic
    }
}
```

Ici on a implémenté l'interface *Engine* (qui contient deux signatures de méthode), à la classe *TuboProp* qui du coup est obligé de définir le corps de ces méthodes.

Par défaut, les méthodes d'une interface sont **PUBLICS** et **ABSTRAITES**. A savoir que la classe qui implémente ces méthodes devra les redéfinir avec un niveau d'accès public. (Bien que cela ne soit pas nécessaire, on peut toujours ajouter le mot clef public devant les méthodes de l'interface *Engine*).

```java
interface Engine {
    // public, static, final by default
    int MIN_OPERATING_TEMPERATURE = -50;
    public void start();
    public void stop();
}
```

Les interfaces peuvent aussi contenir des constantes qui seront **PUBLIC**, **STATIC** et **FINAL** par défaut (int MIN…. dans l'exemple de gauche).

- ***Défault and Static methods :***

A. **Default** :

Si on ajoute une nouvelle méthode (*healthCheck*()) à notre interface *engine* on peut avoir un problème car on **DOIT** redéfinir (*override*) **TOUTES** les méthodes dans les classes qui héritent de cet interface, ce qui peut être long et fastidieux s'il y'en a beaucoup.

Changer des choses dans l'interface peut donc « casser » les classes qui les implémentent déjà si on ne fait pas attention. Dans ce cas on peut utiliser le mot clef DEFAULT sur une méthode pour contrecarrer ce problème.

Il s'agit d'une méthode définie dans une interface, et qui a une implémentation réelle. Elles permettent aux interfaces d'évoluer ou d'être modifiée sans forcer les changements dans les classes qui implémentent cette interface.

```java
interface Engine {
    void start();
    void stop();
    default String healthCheck() {
        // logic goes here
         return "OK";
    }
}
```

```java
public class TurboProp implements Engine {
    @Override
    public void start() {}

    @Override
    public void stop() {}
}

Engine e = new TurboProp();
e.healthCheck();
```

Comme on peut le constater :

- On a créé une nouvelle méthode *healthCheck*() avec le mot clef *default*.
- Cette méthode doit avoir un corps (*body*), ici elle retourne simplement un String « ok ».
- Elle n'apparait pas dans la classe qui l'implémente (*TurboProp*), donc on ne doit pas la redéfinir (*override)*.
- Quand je créé une instance de *TurboProp*, je peux utiliser la méthode *healthCheck()*.

Ses propriétés :

1. Elle est implicitement publique.
2. Elle doit avoir une implémentation.
3. Elle peut être *override* mais ce n'est pas obligatoire.

B. **Static** :

Une méthode *static* peut être implémentée dans une interface. Tout comme dans une classe normale, il faudra l'implémenter (définir le corps) et elle pourra être utilisée de la même manière.

```java
interface Engine {
    int MIN_TEMP = -50;
    void start();
    void stop();

    static boolean canStart(int outsideTemp) {
        return outsideTemp > MIN_TEMP;
    }
}
```

```java
// A static interface method is linked to the
// interface, not to classes that implement it
boolean canStart = TurboProp.canStart(-30);
```

Par contre une classe qui implémentera cette interface ne pourra pas utiliser cette méthode car elle est liée à l'interface.

L'utilité est juste pratique en vérité. Cela nous évite de devoir créer une classe complète alors que l'on veut simplement créer une méthode accessible partout.

- ***Functional Interface :***

Ce sont des interfaces qui contiennent une seule méthode **ABSTRAITE** (mais elles peuvent avoir de multiples *default* ou *static methods*).

Pour les créer il faut :

- Ecrire une seule méthode.
- Mettre l'annotation @FunctionalInterface au-dessus du nom de la classe.

Depuis Java 8 on peut utiliser les expressions lambdas pour implémenter une interface fonctionnelle.

Ici je défini une interface fonctionnelle. On voit qu'il n'y a rien dans la méthode *depart*() :

```
@FunctionalInterface
public interface DebutEvenement {
    void depart();
}
```

Du coup je peux récupérer cette méthode que j'implémente dans une variable grâce à une expression lambda. Dans la deuxième ligne, j'utilise la variable suivie du nom de méthode pour l'exécuter :

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

    DebutEvenement demarrage = () -> System.out.println("debut du
programme");
    demarrage.depart();
}
```

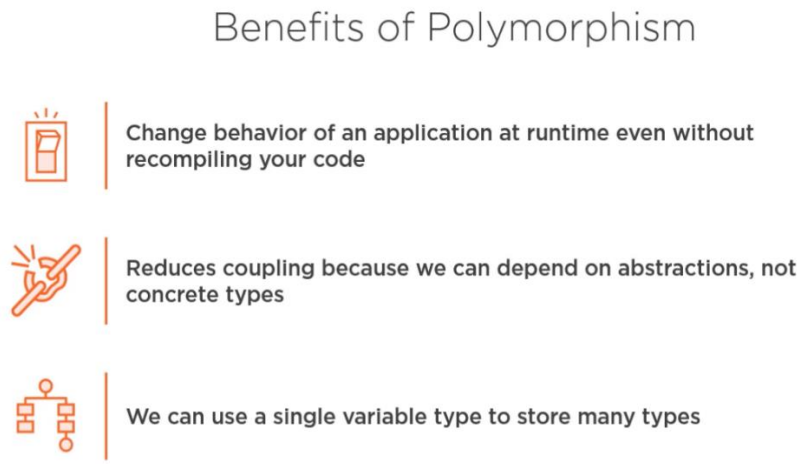
L'utilité générale est de nous permettre de nous passer de créer des classes inutiles ou écrire beaucoup de code standard et ce en implémentant ces méthodes à l'aide de lambda.

- ***Polymorphism :***

**Définition** :

| |
|---|
| C'est la capacité d'une abstraction (une interface ou une classe de base) à prendre de nombreuses formes au **moment de l'exécution**. |

Le moyen le plus courant d'obtenir le polymorphisme est avec un *override* de méthode et cela que ça soit par l'héritage ou l'implémentation d'interface.




Comment choisir si on veut utiliser une classe abstraire ou une Interface :

**Abstract Class** :

| |
|---|
| - Quand on veut fournir des fonctionnalités de base aux sous-classes.<br>- Quand on veut fournir un modèle pour les futures sous-classes.<br>- Quand on a besoin de créer des méthodes abstraites qui ne sont pas publiques. |

**Interface** :

| |
|---|
| - Quand on a besoin d'un plus haut niveau d'abstraction.<br>- Quand on veut un lien (un « couplage ») moins important avec les autres classes.<br>- Quand on a besoin d'implémenter plusieurs interfaces. |

# IX. Defining Enumerations and Nested Classes :

- ***Enum types :***

**Définition** :

| Type de donnée qui permet à une variable de contenir un certain nombre de valeurs prédéfinies. On doit les utiliser dès qu'on a besoin de constituer un ensemble fixe de constantes. |
|---|

Déclaration : on utilise le mot clef **ENUM,** ensuite des constantes séparées par des virgules :

```
public enum CouleurAnimal {
    BLEU,
    BLANC,
    NOIR,
    ORANGE
}
```

Pour l'utiliser, on peut facilement le faire en combinant l'enum avec un switch. Ici je lui demande de me retourner un string du nom de la couleur correspondante :

```
static public String ChoixCouleur(CouleurAnimal cl){
    switch (cl) {
        case BLEU:
            return "bleu";
        case BLANC:
            return "blanc";
        case NOIR:
            return "noir";
        case ORANGE:
            return "orange";
        default: return null;
    }
}
```

Caractéristiques du type ENUM :

- Il peut avoir un corps (body) qui peut inclure des champs et des méthodes.
- Des méthodes statiques sont d'office implémentée par le compilateur. (ex : *values()* et *valuesOf()*).
- Il peut avoir des propriétés assignées à chaque valeur constante.
- On peut jouter un constructeur.

Une des particularités de la classe ENUM est que l'on peut attribuer des valeurs à chaque constante. Il faut dans ce cas définir :

- Une variable finale qui contiendra la valeur.
- Un constructeur.
- Un geteur.

Dans l'exemple suivant j'associe des valeurs de types String à chaque énumération :

```
public enum CouleurAnimal {
    BLEU("bleu"),
    BLANC("blanc");

    private final String couleur;

    CouleurAnimal(String couleur) {
        this.couleur = couleur;
    }

    public String getCouleur() {
        return couleur;
    }
}
```

Je peux alors récupérer ces valeurs de cette manière si je passe dans le main :

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

String couleur = CouleurAnimal.BLANC.getCouleur();
    System.out.println(couleur);

}
```

- ***Nested class :***

Ce sont de classes définies à l'intérieur d'une autre classe.

Il y'en a deux types :

- Les statiques (*Static nested classes*)[1] .
- Les non-statiques (*Inner classes*).

Caractéristiques des *Inner classes* (les non statiques) :

- Elles ont un accès direct aux champs et méthodes de la classe externe.
- Comme elles sont associée à une instance, elles ne peuvent pas contenir de membre statique.
- Pour les instancier, il faut d'abord instancier un objet de la classe externe.

[1] Ndr : elles seront vues plus loin dans la matière de la certification.

```java
public class Aircraft {

    private final int altitudeFl;

    private final boolean isRvsmCapable;


    public class VerticalSeparation {

        private int separationInFeet;

        VerticalSeparation() {

            if (altitudeFl >= 290 && altitudeFl <= 410 && isRvsmCapable) { separationInFeet = 1000;}

            else {  separationInFeet = 2000; }

        }

        public int getSeparationInFeet() { return separationInFeet; }

    }


    public int getSeparationFeet() { VerticalSeparation vsep = new VerticalSeparation(); return vsep.getSeparationInFeet(); }

}
```

Dans ce cas-ci l'auteur a créé une classe Aircraft dans laquelle il créé une classe interne qui lui permettra de faire des calculs concernant l'obligation pour des aéronefs d'avoir une certaine distance de séparation[2]. L'instanciation se fera alors de cette manière :

```java
Aircraft a = new Aircraft(300,true);
Aircraft.VerticalSeparation vsep = a.new VerticalSeparation();
```

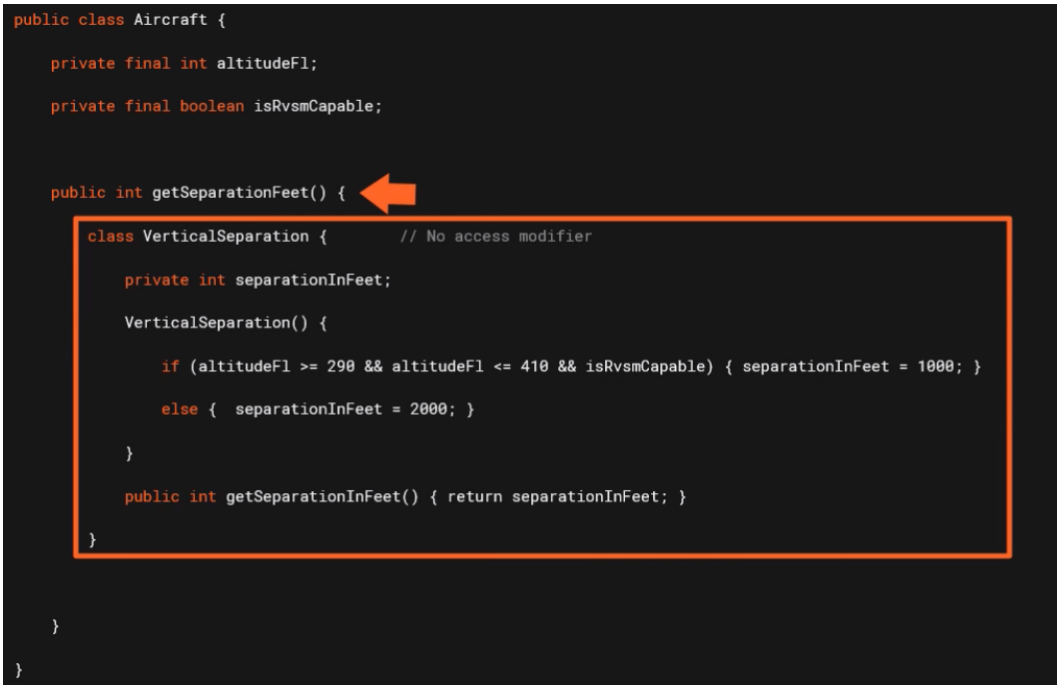
Les classes internes peuvent être subdivisée en deux catégories : **LOCAL Class** et **LAMBDA**.

[2] Ndr : malheureusement c'est l'exemple fournir par l'auteur de la vidéo. Un exemple plus simple aurait été plus pertinent.

- ***Local class :***

C'est une classe définie dans un **BLOC** de code. Elles peuvent l'être dans une méthode (ce qui se fait le plus), dans une boucle ou encore dans un If.

Ici ce qu'il fait c'est qu'il déclare la classe locale VerticalSeparation() dans une méthode car il ne prévoit pas de l'utiliser ailleurs (l'implémentation n'a pas changée) :



```
public class Aircraft {

    private final int altitudeFl;

    private final boolean isRvsmCapable;


    public int getSeparationFeet() {
        class VerticalSeparation {        // No access modifier

            private int separationInFeet;

            VerticalSeparation() {

                if (altitudeFl >= 290 && altitudeFl <= 410 && isRvsmCapable) { separationInFeet = 1000; }

                else {  separationInFeet = 2000; }

            }

            public int getSeparationInFeet() { return separationInFeet; }

        }


    }
}
```

L'instanciation se fera alors via la méthode :

```
VerticalSeparation vsep = new VerticalSeparation(); return vsep.getSeparationInFeet();
```

Tout comme les classes internes les classes locales peuvent accéder à tous les membres d'instance de sa classe englobante.

Elle peut en plus accéder aux variables locales définies dans la même portée (*scope*) ou bloc de code à condition qu'elle soit finale ou effectivement finale (c'est-à-dire qu'on lui attribue une valeur dans la classe et que cette valeur ne doit pas changer sans pour autant lui assigner le mot clef FINAL).

Enfin, elle peut aussi accéder aux paramètres de la méthode dans laquelle elle est définie.

**Restrictions** de la classe locale :

- Elle ne peut pas contenir de de membres statiques à l'exception des constantes (champ *final static* de primitif ou String).
- Cela ne peut pas être une interface, que des classes.
- Elle ne peut pas être instancier à partir de l'extérieur du bloc dans lequel elle a été définie.
- Elle n'a pas de modificateur d'accès supérieur (pas de contrôleur d'accès).

- ***Anonymous Classes :***

Elle peut être assimilée à une classe locale simplifiée. C'est un excellent moyen de déclarer et d'instancier une classe en même temps. C'est possible car les classes anonymes sont traitées comme des expressions et peuvent être utilisé comme telles.

Les accès possibles de la classe anonyme sont :

- Tous les membres d'instance de sa classe englobante.
- Les variables locales définies dans la même portée (*scope*) si elles sont finales ou effectivement finales.
- Aux paramètres de la méthode si elle est définie dans celle-ci.

Les restrictions sont :

- Elle ne peut pas contenir de membre statique à l'exception des constantes.
- Elle ne peut pas avoir de constructeur.

Hormis cela, on peut tout à fait y ajouter des champs, des méthodes, des classes locales ou des *instance initializer* qui n'existent pas dans la classe ou l'interface d'origine.

```
// The interface can have as many members as possible

public interface UnitConvertor {

    int convert();

}
```

Pour créer une classe anonyme, on déclare simplement ici une interface *UnitConvertor* qui a une méthode *convert()*.

Ensuite, dans une méthode quelconque (*someMethod()*), on va instancier un objet de l'interface qu'on appelle **feetToF1**.

La seule différence par rapport aux instanciations usuelles est qu'on va en plus ajouter des **ACCOLADES** après ce qui nous oblige à définir un corps (*body*).

On va alors s'en servir pour surcharger (*override*) la méthode *convert()* en ajoutant des données.

On voit qu'à **l'INTERIEUR** de la méthode il peut manipuler l'objet *feetToF1* en le soumettant à un print.

```
public void someMethod() {

    int feet = 2000;   // Final or effectively final

    UnitConvertor feetToF1 = new UnitConvertor() {
        @Override
        public int convert() {
            return feet / 100;
        }
    };

    System.out.println(feetToF1.convert());
}
```

Il est d'ailleurs tout à fait possible de faire de multiples implémentations :

On voit ici qu'il instancie plusieurs objets *UnitConvertor* qui surcharge la méthode convert() de manière différente sans que cela pose problème.


- ***Lambda Expression :***

**Définition** :

| C'est un moyen de représenter une interface fonctionnelle en utilisant une expression. |
|---|

Elle est composée de trois parties :

1. Les paramètres (de 0 à plusieurs) : séparé par des virgules et entre parenthèses s'il y'a plus d'un seul paramètre.
2. L'opérateur : ->
3. Le corps de la fonction. S'il y'a plusieurs instructions dans le corps, il faut qu'elles soient dans des **accolades**.

Globalement une expression lambda sert à écrire du code moins chargé et est utile pour la programmation fonctionnelle. Elle permet notamment de définir une fonction à l'endroit ou elle est utilisée.

Quelques exemples :

a) **() -> 123** : N'accepte aucun paramètre et renvoie la valeur 123.
b) **x -> x * 2** : Accepte un paramètre et renvoie son double.
c) **(x, y) -> x + y** : Accepte deux paramètres et renvoie leur somme.
d) **(val1, val2) -> val1 >= val2** : Renvoie un booléen qui préciser si la première valeur est supérieur à la deuxième.