

# **Formation Java 8**

---

## **TP : Les Expressions Lambda**

---

# SOMMAIRE

I-	Prérequis : .....	3
II-	Objectifs .....	3
III-	Les Expressions Lambda par la pratique .....	3
a.	La valeur ajoutée des Expressions Lambda .....	3
b.	La syntaxe des Expressions Lambda .....	4
c.	Le corps d'une expression lambda .....	7
d.	Des exemples d'expressions lambda .....	8
e.	La portée des variables .....	9
f.	L'utilisation d'une expression lambda .....	12

## I- Prérequis :

- JDK 8.
- Eclipse.

## II- Objectifs

- ✓ Comprendre la valeur ajoutée des Expressions Lambda.
- ✓ Comprendre la syntaxe des Expressions Lambda.
- ✓ Maîtriser comment implémenter le corps d'une expression lambda.
- ✓ Quelques exemples d'expressions lambda.
- ✓ Comprendre la portée des variables.
- ✓ L'utilisation d'une expression lambda.

## III- Les Expressions Lambda par la pratique

### a. La valeur ajoutée des Expressions Lambda

- ❖ Avant Java 8, pour définir un thread, il était possible de créer une classe anonyme de type Runnable et de passer son instance en paramètre du constructeur d'un thread :

```
public static void test0() {  
    Thread monThread = new Thread(new Runnable() {  
        @Override  
        public void run() {  
            System.out.println("Mon traitement ");  
        }  
    });  
    monThread.start();  
}
```

- ❖ A partir de Java 8, il est possible d'utiliser une expression lambda en remplacement de la classe anonyme pour obtenir le même résultat avec une syntaxe plus concise.

L'utilisation d'une expression lambda évite d'avoir à écrire le code nécessaire à la déclaration de la classe anonyme et de la méthode.

Associées à d'autres fonctionnalités du langage (méthode par défaut, ...) et de l'API (Stream), les lambdas modifient profondément la façon dont certaines fonctionnalités sont codées en Java. Cet impact est plus important que certaines fonctionnalités des versions précédentes de Java comme les generics ou les annotations.

```
public static void test0_bis() {  
    Thread monThread = new Thread(() -> System.out.println("Mon traitement "));  
    monThread.start();  
}
```

## b. La syntaxe des Expressions Lambda

Dans la définition d'une expression lambda, l'opérateur **->** permet de séparer les paramètres des traitements qui les utiliseront. Les paramètres de l'expression doivent donc être déclarés à gauche de l'opérateur **->**.

Les paramètres d'une expression lambda doivent correspondre à ceux définis dans l'interface fonctionnelle.

Les paramètres de l'expression doivent respecter certaines règles :

- Une expression peut n'avoir aucun, un seul ou plusieurs paramètres.
  - Le type des paramètres peuvent être explicitement déclaré ou être inféré par le compilateur selon le contexte dans lequel l'expression est utilisée.
  - Les paramètres sont entourés de parenthèses, chacun étant séparé par une virgule.
  - Des parenthèses vides indiquent qu'il n'y a pas de paramètre.
  - S'il n'y a qu'un seul paramètre dont le type n'est pas explicitement précisé, alors l'utilisation des parenthèses n'est pas obligatoire.
- ❖ Une expression lambda peut ne pas avoir de paramètre.

```
public static void test1() {  
    Runnable monTraitement = () -> System.out.println("traitement");  
    monTraitement.run();  
}
```

- ❖ Pour préciser qu'il n'y a aucun paramètre, il faut utiliser une paire de parenthèses vide. Dans ce cas, la méthode de l'interface fonctionnelle ne doit pas avoir de paramètre. Si l'expression lambda ne possède qu'un seul paramètre alors il y a deux syntaxes possibles. La plus standard est d'indiquer le paramètre entre deux parenthèses.

```
public static void test2() {  
    Consumer<String> afficher = (param) -> System.out.println(param);  
    afficher.accept("tttt");  
}
```

- ❖ Il est aussi possible d'omettre les parenthèses uniquement si le type peut être inféré.

```
public static void test3() {  
    Consumer<String> afficher = param -> System.out.println(param);  
    afficher.accept("tttt");  
}
```

- ❖ Il n'est pas possible d'omettre les parenthèses si le type est précisé explicitement.

```
public static void test4() {  
    Consumer<String> afficher = String param -> System.out.println(param);  
}
```

- ❖ Si l'expression lambda possède plusieurs paramètres alors ils doivent être entourés obligatoirement par des parenthèses et séparés les uns des autres par une virgule.

```
public static void test5() {  
    BiFunction<Integer, Integer, Long> additionner = (val1, val2) -> (long) val1 + val2;  
}
```

- ❖ Il est possible de déclarer explicitement le type du ou des paramètres de l'expression lambda.

```
public static void test6() {  
    Consumer<String> afficher = (String param) -> System.out.println(param);  
}
```

- ❖ Généralement, le type d'un paramètre n'est pas obligatoire si le compilateur est capable de l'inférer : dans la plupart des cas, le compilateur est capable de déterminer le type du paramètre à partir de celui correspondant à l'interface fonctionnelle. Si ce n'est pas le cas, le compilateur génère une erreur et il est alors nécessaire de préciser ce type explicitement. Dans l'exemple ci-dessous, le fait que l'interface fonctionnelle Consumer soit typée avec un generic String permet au compilateur de savoir que le type du paramètre est String. L'exemple ci-dessus peut alors être écrit sans préciser le type du paramètre.

```
public static void test7() {  
    Consumer<String> afficher = (param) -> System.out.println(param);  
}
```

- ❖ Le type précisé pour un paramètre doit correspondre à celui défini dans le type l'interface fonctionnelle.

L'exemple ci-dessous génère une erreur à la compilation.

```
public static void test8() {  
    Consumer afficher = (String param) -> System.out.println(param);  
}
```

- ❖ Il n'est cependant pas possible de mixer dans la même expression des paramètres déclarés explicitement et inférés.

```
public static void test9() {  
    Comparator<String> comparator = (chaine1, String chaine2) ->  
    Integer.compare(chaine1.length(), chaine2.length());  
}
```

- ❖ Il est possible d'utiliser le modificateur final sur les paramètres si leur type est précisé explicitement.

```
public static void test10() {  
    Comparator<String> comparator = (final String chaine1, final String chaine2) ->  
    Integer.compare(chaine1.length(), chaine2.length());  
}
```

- ❖ Ce n'est pas possible si le type est inféré par le compilateur sinon il génère une erreur.

```
public static void test11() {  
    Comparator<String> comparator = (final chaine1, final chaine2) ->  
    Integer.compare(chaine1.length(), chaine2.length());  
}
```

- ❖ Il est aussi possible d'annoter les paramètres d'une expression lambda uniquement si leur type est déclaré explicitement.

```
public static void test12() {  
    Comparator<String> comparator = (@NotNull String chaine1, @NotNull String  
    chaine2) -> Integer.compare(chaine1.length(), chaine2.length());  
}
```

### c. Le corps d'une expression lambda

❖ Le corps d'une expression lambda est défini à droite de l'opérateur `->`. Il peut être :

- ✓ Une expression unique
- ✓ Un bloc de code composé d'une ou plusieurs instructions entourées par des accolades

Le corps de l'expression doit respecter certaines règles :

- Il peut n'avoir aucune, une seule ou plusieurs instructions
- Lorsqu'il ne contient qu'une seule instruction, les accolades ne sont pas obligatoires et la valeur de retour est celle de l'instruction si elle en possède une
- Lorsqu'il y a plusieurs instructions, elles doivent obligatoirement être entourées d'accolades
- La valeur de retour est celle de la dernière expression ou `void` si rien n'est retourné

Si le corps est simplement une expression, celle-ci est évaluée et le résultat de cette évaluation est renvoyé s'il y en a un.

```
public static void test13() {  
    BiFunction<Integer, Integer, Long> additionner = (val1, val2) -> (long) val1 + val2;  
}
```

❖ Il n'est jamais nécessaire de préciser explicitement le type de retour : le compilateur doit être en mesure de le déterminer selon le contexte. Si ce n'est pas le cas, le compilateur émet une erreur.

Si l'expression ne produit pas de résultat, celle-ci est simplement exécutée. Il n'est pas nécessaire d'entourer d'accolades si le corps de l'expression invoque une méthode dont la valeur de retour est `void`.

```
public static void test14() {  
    <String> afficher = (String param) -> System.out.println(param);  
}
```

❖ Les traitements d'une expression lambda peut contenir plusieurs opérations qui doivent être regroupées dans un bloc de code entouré d'accolades comme pour le corps d'une méthode.

```
public static void test15() {  
    Runnable monTraitement = () -> {  
        System.out.println("traitement 1");  
        System.out.println("traitement 2");  
    };  
}
```

```
}
```

- ❖ Le bloc de code est évalué comme si c'était celui d'une méthode : il est possible de terminer son exécution en utilisant l'instruction `return` ou en levant une exception. Si le bloc de code doit retourner une valeur, il faut utiliser le mot clé `return`.

```
public static void test16() {  
    Function<Integer, Boolean> isPositif = valeur -> {  
        return valeur >= 0;  
    };  
}
```

- ❖ Important : toutes les branches du code du corps de l'expression doivent obligatoirement retourner une valeur ou lever une exception si au moins une branche retourne une valeur. Si ce n'est pas le cas, le compilateur émet une erreur. L'exemple ci-dessous se compile pas car lorsque la valeur est négative, il n'y a pas de valeur de retour définie.

```
public static void test17() {  
    Function<Integer, Boolean> isPositif = valeur -> {  
        if (valeur >= 0) {  
            return true;  
        }  
    };  
}
```

- ❖ Les mots clés `break` et `continue` ne peuvent pas être utilisés comme instructions de premier niveau dans le bloc de code mais ils peuvent être utilisés dans des boucles.

#### d. Des exemples d'expressions lambda

Ci-après quelques exemples d'expressions lambda :

Exemple	Description
<code>() -&gt; 123</code> <code>() -&gt; { return 123 };</code>	N'accepter aucun paramètre et renvoyer la valeur 123
<code>x -&gt; x * 2</code>	Accepter un nombre et renvoyer son double
<code>(x, y) -&gt; x + y</code>	Accepter deux nombres et renvoyer leur somme
<code>(int x, int y) -&gt; x + y</code>	Accepter deux entiers et renvoyer leur somme
<code>(String s) -&gt; System.out.print(s)</code>	Accepter une chaîne de caractères et l'afficher sur la sortie standard sans rien renvoyer



<code>c -&gt; { int s = c.size(); c.clear(); return s; }</code>	Renvoyer la taille d'une collection après avoir effacé tous ses éléments. Cela fonctionne aussi avec un objet qui possède une méthode <code>size()</code> renvoyant un entier et une méthode <code>clear()</code>
<code>n -&gt; n % 2 != 0;</code>	Renvoyer un booléen qui précise si la valeur numérique est impaire
<code>(char c) -&gt; c == 'z';</code>	Renvoyer un booléen qui précise si le caractère est 'z'
<code>() -&gt; { System.out.println("Hello World"); };</code>	Afficher "Hello World" sur la sortie standard
<code>(val1, val2) -&gt; { return val1 &gt;= val2; } (val1, val2) -&gt; val1 &gt;= val2;</code>	Renvoyer un booléen qui précise si la première valeur est supérieure ou égale à la seconde
<code>() -&gt; { for (int i = 0; i &lt; 10; i++) traiter(); }</code>	Exécuter la méthode <code>traiter()</code> dix fois
<code>p -&gt; p.getSexe() == Sexe.HOMME &amp;&amp; p.getAge() &gt;= 7 &amp;&amp; p.getAge() &lt;= 77</code>	Renvoyer un booléen pour un objet possédant une méthode <code>getSexe()</code> et <code>getAge()</code> qui vaut true si le sexe est masculin et l'âge compris entre 7 et 77 ans

## e. La portée des variables

- ❖ Vis à vis de la portée et de la visibilité des variables, une expression lambda se comporte syntaxiquement comme un bloc de code imbriqué.  
Comme pour les classes anonymes internes, une expression lambda peut avoir accès à certaines variables définies dans le contexte englobant.  
Dans le corps d'une expression lambda, il est donc possible d'utiliser :
  - Les variables passées en paramètre de l'expression
  - Les variables définies dans le corps de l'expression
  - Les variables final définies dans le contexte englobant
  - Les variables effectivement final définies dans le contexte englobant : ces variables ne sont pas déclarées final mais une valeur leur est assignée et celle-ci n'est jamais modifiée. Il serait donc possible de les déclarer final sans que cela n'engendre de problème de compilation. Le concept de variables effectivement final a été introduit dans Java 8
- ❖ Une expression lambda peut avoir accès aux variables définies dans son contexte englobant.  
Dans l'exemple ci-dessous, les variables *message* et *repetition* ne sont pas définies dans l'expression lambda mais sont des paramètres de la méthode englobante qui sont accédées dans l'expression lambda. Ces variables ne sont pas déclarées final mais elles sont effectivement final donc elles sont utilisables dans l'expression lambda.

```

public static void test18(String message, int repetition) {
    Runnable r = () -> {
        for (int i = 0; i < repetition; i++) {
            System.out.println(message);
        }
    };
    new Thread(r).start();
}

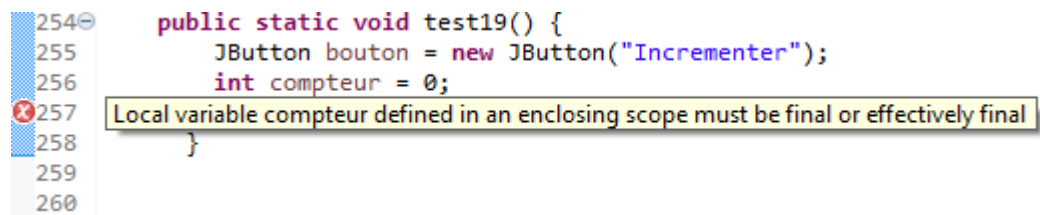
```

- ❖ L'accès aux variables du contexte englobant est limité par une contrainte forte : seules les variables dont la valeur ne change pas peuvent être accédées. La méthode ci-dessous ne se compile pas :

```

public static void test19() {
    JButton bouton = new JButton("Incrementer");
    int compteur = 0;
    bouton.addActionListener(event -> compteur++);
}

```



```

254 public static void test19() {
255     JButton bouton = new JButton("Incrementer");
256     int compteur = 0;
257     bouton.addActionListener(event -> compteur++);
258 }
259
260

```

Local variable compteur defined in an enclosing scope must be final or effectively final

- ❖ La même erreur de compilation est obtenue si la modification est faite dans le contexte englobant de l'expression lambda.

```

public static void test20() {
    JButton bouton = new JButton("Incrementer");
    int compteur = 0;
    compteur++;
    bouton.addActionListener(event -> System.out.println(compteur));
}

```

- ❖ Il est possible de passer en paramètre un objet mutable et de modifier l'état de cet objet. Le compilateur vérifie simplement que la référence que contient la variable ne change pas. Comme tout en Java, il est de la responsabilité du développeur de gérer les éventuels accès concurrents lors de ces modifications.

Comme avec les classes anonymes internes, il est aussi possible de définir un tableau d'un seul élément de type `int` et d'incrémenter la valeur de cet élément.

```
public static void test21() {  
    JButton bouton = new JButton("Incrementer");  
    int[] compteur = new int[1];  
    bouton.addActionListener(event -> compteur[0]++);  
}
```

- ❖ L'utilisation de cette solution de contournement n'est pas recommandée d'autant qu'elle n'est pas thread-safe. Il est préférable d'utiliser une variable de type `AtomicXXX` pour le compteur.

```
public static void test22() {  
    JButton bouton = new JButton("Incrementer");  
    AtomicInteger compteur = new AtomicInteger(0);  
    bouton.addActionListener(event -> compteur.incrementAndGet());  
}
```

- ❖ Le corps de l'expression lambda est soumis aux mêmes règles de gestion de la portée des variables qu'un bloc de code ordinaire.

Les variables locales et les paramètres déclarés dans une expression lambda doivent l'être comme s'ils l'étaient dans le contexte englobant : une expression lambda ne définit pas de nouvelle portée. Ainsi, il n'est pas possible de définir deux variables avec le même nom dans un même bloc de code, donc il n'est pas possible de définir une variable dans l'expression lambda si celle-ci est déjà définie dans le contexte englobant.

```
Personne p1 = new Personne("nom3", "prenom3");  
Personne p2 = new Personne("nom1", "prenom1");  
Personne p3 = new Personne("nom2", "prenom2");  
List<Personne> personnes = new ArrayList(3);  
personnes.add(p1);  
personnes.add(p2);  
personnes.add(p3);  
  
Comparator<Personne> triParNom = (Personne p1, Personne p2) ->  
{  
    return p2.getNom().compareTo(p1.getNom());  
};
```

- ❖ Le mot clé `this` fait référence à l'instance courante dans le bloc de code englobant et dans l'expression lambda.

```
public void test24() {  
    Runnable runnable = () -> { System.out.println(this.toString());};  
    Thread thread = new Thread(runnable);  
    thread.start();  
}
```

#### f.L'utilisation d'une expression lambda

- ❖ Une expression lambda ne peut être utilisée que dans un contexte où le compilateur peut identifier l'utilisation de son type cible (target type) qui doit être une interface fonctionnelle :
  - déclaration d'une variable
  - assignation d'une variable
  - valeur de retour avec l'instruction `return`
  - initialisation d'un tableau
  - paramètre d'une méthode ou d'un constructeur
  - corps d'une expression lambda
  - opérateur ternaire `?:`
  - cast
- ❖ Par exemple, la déclaration d'une expression lambda peut être utilisée directement en paramètre d'une méthode.

```
public static void test25() {  
    JButton bouton = new JButton("test");  
    bouton.addActionListener(event -> System.out.println("clik"));  
}
```

- ❖ Il est aussi possible de définir une variable ayant pour type une interface fonctionnelle qui sera initialisée avec l'expression lambda.

```
public static void test26() {  
    JButton bouton = new JButton("test");  
    ActionListener monAction = event -> System.out.println("clik");  
    bouton.addActionListener(monAction);  
}
```

- ❖ Une expression lambda est définie grâce à une interface fonctionnelle : une instance d'une expression lambda qui implémente cette interface est un objet. Cela permet à une expression lambda :
  - De s'intégrer naturellement dans le système de type de Java
  - D'hériter des méthodes de la classe `Object`

```
public static void test27() {
    Runnable monTraitement = () -> {
        System.out.println("mon traitement");
    };
    Object obj = monTraitement;
}
```

- ❖ Attention cependant, une expression lambda ne possède pas forcément d'identité unique : la sémantique de la méthode equals() n'est donc pas garantie.

Le type dans la déclaration, désigné par target type, est le type de l'interface fonctionnelle dans le contexte auquel l'expression lambda sera utilisée.

Une expression lambda sera transformée par le compilateur en une instance du type de son interface fonctionnelle selon le contexte dans lequel elle est définie :

- soit celle du type à laquelle l'expression est assignée
- soit celle du type du paramètre à laquelle l'expression est passée

L'expression lambda ne contient pas elle-même d'information sur l'interface fonctionnelle qu'elle implémente. Cette interface sera déduite par le compilateur en fonction de son contexte d'utilisation.

Le type de l'interface fonctionnelle est déterminé par le compilateur en fonction du contexte de son utilisation. Deux expressions lambda syntaxiquement identiques peuvent donc être compatibles avec plusieurs interfaces fonctionnelles et peuvent donc être compilées en deux objets de type différents. Une même expression lambda peut donc être assignée à plusieurs interfaces fonctionnelles tant qu'elle respecte leur contrat.

```
public static void test28() {
    LongFunction longFunction = x -> x * 2;
    IntFunction intFunction = x -> x * 2;
    Callable<String> monCallable = () -> "Mon traitement";
    Supplier<String> monSupplier = () -> "Mon traitement";
}
```

- ❖ Le compilateur associe une expression lambda à une interface fonctionnelle et comme une interface fonctionnelle ne peut avoir qu'une seule méthode abstraite :
  - Les types des paramètres doivent correspondre à ceux des paramètres de la méthode
  - Le type de retour du corps de l'expression doit correspondre à celui de la méthode
  - Toutes les exceptions levées dans le corps de l'expression doivent être compatibles avec les exceptions déclarées dans la clause throws de la méthode

```

public class Calculatrice {
    @FunctionalInterface
    interface OperationEntiere {
        long effectuer(int a, int b);
    }

    public long calculer(int a, int b, OperationEntiere operation) {
        return operation.effectuer(a, b);
    }

    public static void main(String[] args) {
        Calculatrice calc = new Calculatrice();
        OperationEntiere addition = (a, b) -> a + b;
        OperationEntiere soustraction = (a, b) -> a - b;

        System.out.println(calc.calculer(10, 5, addition));
        System.out.println(calc.calculer(10, 5, soustraction));
    }
}

```

Il est possible d'imbriquer des expressions lambda mais dans ce cas le code devient moins lisible.

**Fin du TP 1.**