



# Lambda et Stream

Pr. L.KARBIL

## 1. Fonction Lambda

### Définition

Une fonction lambda est une méthode anonyme (sans nom) qui peut être utilisée pour écrire du code concis. Elle simplifie l'écriture de classes anonymes ou de fonctions qui n'ont qu'une seule méthode à implémenter.

### Syntaxe

La syntaxe d'une lambda est :

(paramètres) -> { corps de la fonction }

- **Paramètres** : Les paramètres nécessaires pour la fonction. S'il y a un seul paramètre, les parenthèses sont optionnelles.
- **Flèche (->)** : Sépare les paramètres du corps de la fonction.
- **Corps** : Le code à exécuter. Il peut être une seule ligne ou un bloc {}.

### Exemples

#### 1. Lambda avec un seul paramètre :

x -> x \* x

#### 2. Lambda avec plusieurs paramètres :

(a, b) -> a + b

#### 3. Lambda avec un bloc :

```
(a, b) -> {
    int sum = a + b;
    return sum;
}
```

### Utilisation courante

Les lambdas sont souvent utilisées avec des interfaces fonctionnelles, qui sont des interfaces ayant une seule méthode abstraite, comme celles du package java.util.function :

- Consumer<T> : Accepte une entrée et ne renvoie rien.
- Function<T, R> : Accepte une entrée et renvoie un résultat.
- Predicate<T> : Accepte une entrée et renvoie un boolean.



- Supplier<T> : Ne prend aucune entrée et renvoie une sortie.

**Exemple :**

```
import java.util.function.Consumer;

public class Main {

    public static void main(String[] args) {
        // Consumer qui imprime un message
        Consumer<String> printer = message -> System.out.println(message);
        printer.accept("Bonjour, les fonctions lambda !");
    }
}
```

## Rappel des interfaces fonctionnelles

Les lambdas sont étroitement liées aux interfaces fonctionnelles, définies comme des interfaces ayant une seule méthode abstraite (annotées avec @FunctionalInterface). Ces interfaces permettent de définir un comportement avec les lambdas.

Exemples d'interfaces fonctionnelles :

1. Runnable (dans java.lang) :

```
@FunctionalInterface
public interface Runnable {
    void run();
}
```

## Création d'une Lambda

**Exemple sans lambda (avant Java 8) :**

```
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        System.out.println("Exécution d'un thread");
    }
};
new Thread(runnable).start();
```

**Exemple avec lambda (Java 8 et après) :**



```
Runnable runnable = () -> System.out.println("Exécution d'un thread");
new Thread(runnable).start();
```

## Cas Pratiques

1. Utilisation de **Predicate** pour filtrer une liste :

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class LambdaExample {
    public static void main(String[] args) {

        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

        // Définir une condition avec un Predicate
        Predicate<String> longerThan3 = name -> name.length() > 3;

        // Filtrer les noms avec une boucle
        for (String name : names) {

            if (longerThan3.test(name)) {

                System.out.println(name); // Affiche "Alice", "Charlie", "David"
            }
        }
    }
}
```

2. Utilisation de **Function** pour transformer une valeur :

```
import java.util.function.Function;

public class LambdaExample {
    public static void main(String[] args) {
        Function<Integer, String> intToString = num -> "Nombre : " + num;

        // Appliquer la transformation
        System.out.println(intToString.apply(10)); // Affiche "Nombre : 10"
    }
}
```

## 2. Streams en Java

Qu'est-ce qu'un Stream ?



Un Stream est une séquence d'éléments qu'on peut traiter de manière fonctionnelle. Cela signifie que tu peux **manipuler, filtrer, trier et transformer** ces éléments **sans écrire de boucles ou de code complexe**.

#### Analogie :

Pense à un Stream comme à une chaîne de montage dans une usine :

1. **Les objets** (par exemple des voitures) arrivent sur la chaîne.
2. À chaque étape, on peut effectuer une action (peinture, assemblage, vérification, etc.).
3. Une fois toutes les actions terminées, on obtient le résultat final (des voitures finies).

#### Pourquoi utiliser les Streams ?

1. **Code concis et lisible** : Plus besoin d'écrire des boucles complexes.
2. **Traitement en pipeline** : Permet de chaîner plusieurs opérations en une seule.
3. **Traitement paresseux** : Les Streams traitent les données uniquement quand c'est nécessaire, ce qui peut améliorer les performances.
4. **Parallélisme** : Facile à exécuter en mode parallèle.

#### Comment utiliser un Stream ?

Un Stream passe généralement par trois étapes principales :

1. **Source** : D'où viennent les données (par exemple, une liste ou un tableau).
2. **Opérations intermédiaires** : Ce qu'on fait avec les données (filtrage, transformation, tri).
3. **Opération terminale** : Ce qu'on obtient à la fin (une liste, un résultat unique, etc.).

#### Syntaxe générale :

```
collection.stream()  
.operationIntermediaire1()  
.operationIntermediaire2()  
.operationTerminale();
```

#### Exemples détaillés

##### Exemple 1 : Filtrer une liste

##### Problème :



Tu as une liste de nombres et tu veux récupérer uniquement les nombres pairs.

**Sans Stream :**

```
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> nombres = List.of(1, 2, 3, 4, 5, 6);
        List<Integer> nombresPairs = new ArrayList<>();

        for (int nombre : nombres) {
            if (nombre % 2 == 0) {
                nombresPairs.add(nombre);
            }
        }

        System.out.println(nombresPairs); // [2, 4, 6]
    }
}
```

**Avec Stream :**

```
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> nombres = List.of(1, 2, 3, 4, 5, 6);
        List<Integer> nombresPairs = nombres.stream()
            .filter(nombre -> nombre % 2 == 0) // Garder uniquement les nombres pairs
            .toList(); // Collecter les résultats dans une liste
    }
}
```



```
System.out.println(nombresPairs); // [2, 4, 6]
}
}
```

- **Explication des étapes :**

1. **stream()** : Crée un Stream à partir de la liste.
2. **filter()** : Garde uniquement les éléments qui respectent une condition (ici  $\text{nombre \% 2 == 0}$ ).
3. **toList()** : Convertit le résultat final en une liste.

### Exemple 2 : Transformer une liste

#### Problème :

Tu veux convertir une liste de noms en majuscules.

#### Sans Stream :

```
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> noms = List.of("Ali", "Fatou", "Demba");
        List<String> nomsMajuscules = new ArrayList<>();

        for (String nom : noms) {
            nomsMajuscules.add(nom.toUpperCase());
        }

        System.out.println(nomsMajuscules); // [ALI, FATOU, DEMBA]
    }
}
```

#### Avec Stream :

```
import java.util.List;
```



```
public class Main {
    public static void main(String[] args) {
        List<String> noms = List.of("Ali", "Fatou", "Demba");

        List<String> nomsMajuscules = noms.stream()
            .map(String::toUpperCase) // Convertir chaque élément en majuscules
            .toList();

        System.out.println(nomsMajuscules); // [ALI, FATOU, DEMBA]
    }
}
```

- **Explication :**

- **map()** : Transforme chaque élément du Stream (ici, en appelant toUpperCase sur chaque chaîne).

### **Exemple 3 : Trouver des éléments spécifiques**

#### **Problème :**

Tu veux savoir si une liste contient un mot qui commence par "F".

#### **Avec Stream :**

```
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> noms = List.of("Ali", "Fatou", "Demba");

        boolean contientF = noms.stream()
            .anyMatch(nom -> nom.startsWith("F")); // Vérifie si un élément commence par "F"

        System.out.println(contientF); // true
    }
}
```



```
}
```

```
}
```

- **Explication :**

- **anyMatch()** : Vérifie si **au moins un élément** satisfait la condition.

#### **Exemple 4 : Trier une liste**

##### **Problème :**

Tu veux trier une liste de nombres dans l'ordre décroissant.

##### **Avec Stream :**

java

Copier le code

```
import java.util.List;
```

```
public class Main {
```

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

```
        List<Integer> nombres = List.of(5, 3, 1, 4, 2);
```

```
        List<Integer> triDecroissant = nombres.stream()
```

```
            .sorted((a, b) -> b - a) // Tri dans l'ordre décroissant
```

```
            .toList();
```

```
        System.out.println(triDecroissant); // [5, 4, 3, 2, 1]
```

```
}
```

```
}
```

#### **Exemple 5 : Réduire les données**

##### **Problème :**

Tu veux calculer la somme d'une liste de nombres.

##### **Avec Stream :**

```
import java.util.List;
```



```
public class Main {  
  
    public static void main(String[] args) {  
  
        List<Integer> nombres = List.of(1, 2, 3, 4, 5);  
  
        int somme = nombres.stream()  
            .reduce(0, (a, b) -> a + b); // Additionner tous les éléments  
  
        System.out.println(somme); // 15  
    }  
}
```

- **Explication :**

- **reduce()** : Combine tous les éléments en un seul résultat (ici, la somme).

### Principales opérations sur les Streams

#### 1. Opérations intermédiaires (produisent un nouveau Stream) :

- **filter(Predicate)** : Filtre les éléments.
- **map(Function)** : Transforme les éléments.
- **sorted(Comparator)** : Trie les éléments.
- **distinct()** : Élimine les doublons.
- **limit(n)** : Garde les n premiers éléments.

#### 2. Opérations terminales (produisent un résultat) :

- **collect()** : Convertit les éléments en une collection (liste, ensemble, etc.).
- **reduce()** : Combine tous les éléments.
- **forEach()** : Applique une action sur chaque élément.
- **count()** : Compte les éléments.
- **anyMatch()** : Vérifie si un élément respecte une condition.

### Exemple détaillé

Supposons que nous avons une liste de nombres et que nous voulons :

1. Filtrer les nombres pairs.



2. Multiplier les nombres restants par 2.
3. Les trier.
4. Les imprimer.

```
import java.util.Arrays;  
  
import java.util.List;  
  
import java.util.stream.Collectors;  
  
  
public class StreamExample {  
    public static void main(String[] args) {  
  
        List<Integer> numbers = Arrays.asList(3, 8, 12, 7, 5, 10);  
  
  
        // Pipeline de traitement  
  
        List<Integer> result = numbers.stream()  
  
            .filter(n -> n % 2 == 0)      // Étape 1 : Filtrer les nombres pairs  
  
            .map(n -> n * 2)          // Étape 2 : Multiplier par 2  
  
            .sorted()                  // Étape 3 : Trier  
  
            .collect(Collectors.toList()); // Étape 4 : Collecter dans une liste  
  
  
        // Imprimer les résultats  
  
        System.out.println(result); // [16, 20, 24]  
    }  
}
```

### 3 .Combinaison Lambda et Streams

Les lambdas sont couramment utilisées avec les streams pour définir des comportements personnalisés. Voici un exemple qui combine les deux pour travailler avec une liste de noms.

**Exemple :**

```
import java.util.Arrays;  
  
import java.util.List;
```



```
public class LambdaAndStreams {  
    public static void main(String[] args) {  
  
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");  
  
        // Filtrer les noms avec plus de 3 lettres, les convertir en majuscules, et les trier  
        names.stream()  
            .filter(name -> name.length() > 3) // Lambda pour filtrer  
            .map(String::toUpperCase)      // Lambda pour transformer en majuscules  
            .sorted()                   // Trier  
            .forEach(System.out::println); // Lambda pour imprimer  
    }  
}
```

**Résultat :**

```
ALICE  
CHARLIE  
DAVID
```