Les expressions Lambda Introduction

AMMOR Fatimazahra

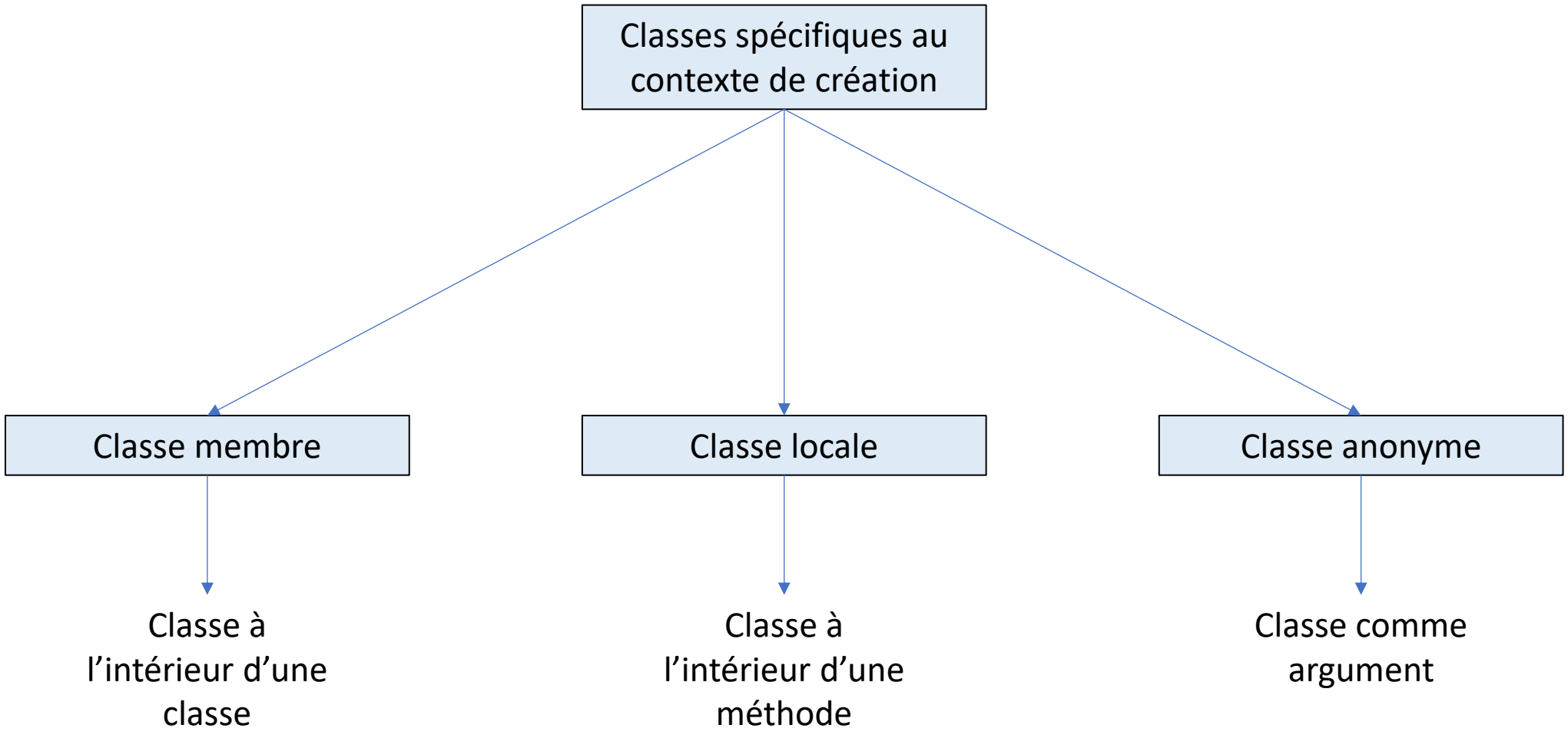



Les types de classe en java

- En Java, bien qu'il est possible d'écrire plusieurs classes dans un même fichier, la règle communément utilisée est de n'en écrire qu'une seule. Il existe cinq types de classe en Java :

1. ***Les classes publiques :*** sont de loin les plus utilisées, on peut même concevoir des applications entières en n'utilisant que celles-là. Il existe deux types de règles à suivre lorsque l'on écrit du code Java. Les premières sont des obligations, le plus souvent imposées par la JLS (Java Language Specification), et les autres sont des bonnes habitudes. La première bonne habitude à prendre, consiste à écrire correctement ses identificateurs : noms de classes, de méthodes ou de champs. En Java, un identificateur de champ ou de méthode commence systématiquement par une minuscule, et ne comporte pas de caractère "_", bien qu'il soit possible d'utiliser ce caractère. Lorsqu'un identificateur est composé de plusieurs mots, alors on met en majuscule la première lettre des mots qui le composent. Par exemple, age_marin n'est pas un identificateur conventionnel en Java, on écrira ageMarin. On appelle cette façon d'écrire les variables le *camel case* , du fait de l'aspect bosselé qu'elles prennent.
2. ***Les classes internes :*** On peut pas avoir plus d'une classe publique dans un fichier donné. En revanche, on peut ajouter d'autres classes, non publiques. Cependant, cette façon de faire n'est pas conseillée.

Les types de classe en java



Les types de classe en java

3. ***Les classes membres :*** Une classe membre est une classe déclarée à l'intérieur d'une autre classe. Ce qui permet de regrouper des classes qui sont fortement liées, et de créer des structures de données plus complexes. Elle peut être statique (liée à la classe et non pas à l'instance) ou non (liée à une instance de la classe englobante et peut accéder aux membres non statiques de cette instance), final ou non, public, private ou protected.

Pourquoi utiliser des classes membres?

- ***Encapsulation***: Les classes membres permettent d'encapsuler des données et des méthodes qui sont étroitement liées à une classe particulière, améliorant ainsi la lisibilité et la maintenabilité du code.
- ***Organisation***: Elles aident à organiser un code complexe en regroupant des classes qui ont une relation étroite.
- ***Réutilisabilité***: Une classe membre peut être réutilisée dans différentes instances de la classe englobante.

Les types de classe en java

```
public class Vehicule {
    private String marque;
    private Moteur moteur; // Moteur est une
                           classe membre non-statique

    // ... constructeurs, getters, setters

    public class Moteur {
        private int puissance;
        private int cylindree;

        // ... constructeurs, getters, setters

        public void demarrer() {
            System.out.println("Le moteur démarre.");
        }
    }
```

```
    public static void main(String[] args) {
  Vehicule maVoiture = new Vehicule(); // Instanciation de
                                       la classe Vehicule
  Vehicule.Moteur moteurDeMaVoiture = maVoiture.new
                Moteur(); // Instanciation du moteur
        moteurDeMaVoiture.demarrer();
    }
}
```

- **Encapsulation:** La classe Moteur est étroitement liée à la classe Vehicule. En la définissant comme une classe membre, nous encapsulons la notion de « Moteur" à l'intérieur de la classe Vehicule, ce qui améliore la lisibilité et la maintenabilité du code.
- **Organisation:** Cela permet de modéliser de manière plus précise la relation entre un vehicule et son moteur.
- **Réutilisabilité:** Si nous voulons créer d'autres types de véhicules avec des caractéristiques de moteurs différentes, nous pouvons facilement étendre la classe Véhicule et personnaliser la classe membre Moteur.

Les types de classe en java

4. ***Les classes locales :*** Une classe locale est déclarée dans une méthode et donc ne fait pas partie des membres de la classe qui contient cette méthode. Elle a une portée limitée à ce bloc de code et ne peut pas être accédée depuis l'extérieur. Pourquoi utiliser une classe locale ?
   - ***Encapsulation***: Tout comme les classes membres, les classes locales permettent d'encapsuler du code spécifique à une méthode. Cela rend le code plus lisible et plus facile à maintenir.
   - ***Réutilisabilité***: Bien qu'elles soient limitées à un bloc, les classes locales peuvent être réutilisées plusieurs fois à l'intérieur de ce bloc, ce qui peut être utile pour factoriser du code commun.
   - ***Flexibilité***: Les classes locales peuvent accéder aux variables locales de la méthode où elles sont définies (à condition qu'elles soient déclarées final), ce qui offre une certaine flexibilité dans leur implémentation.

```
public class ExempleClasseLocale {
    public void trierTableau(Integer[] tableau) {
        class Comparateur implements
                                Comparator<Integer> {
            @Override
            public int compare(Integer o1, Integer o2) {
                return o2 - o1; // Tri décroissant
            }
        }

        Arrays.sort(tableau, new Comparateur());
    }
}
```

Une classe anonyme

Les types de classe en java

5. ***Les classes anonymes :*** Une classe anonyme est déclarée et instanciée en une seule ligne de code, sans lui donner de nom explicite. Elle est souvent utilisée pour fournir une implémentation rapide et concise d'une interface ou d'une classe abstraite. Le code de la classe est écrit entre accolades, directement après l'appel au constructeur.

```
new Type() {
   // Corps de la classe anonyme
};
```

Type = une classe (abstraite ou non) ou une interface.
Corps de la classe = les méthodes à implémenter ou à redéfinir.
La création de l'objet se fait en même temps que la définition de la classe.

```
Type t = new Type() {
     @Override
      public int maMethode(int a, int b) {
               return a + b;
      }
};
```

Les types de classe en java

Pourquoi utiliser une classe anonyme ?

- ***Concision***: Évite la création d'une classe séparée pour une implémentation simple.
- ***Flexibilité***: Permet de créer des objets à la volée, souvent utilisés comme arguments de méthodes.
- ***Implémentations simples***: Parfaite pour des implémentations d'interfaces où seules quelques méthodes sont surchargées.

Cependant, à utiliser avec précautions : il est très facile de rendre un code complètement illisible par une utilisation déraisonnée des classes anonymes. On peut citer comme inconvénients:

- ***Moins de réutilisabilité***: Une classe anonyme est spécifique à l'endroit où elle est créée.
- ***Complexité***: Pour des implémentations plus complexes, il peut être plus clair de créer une classe nommée.
- ***Limitations*** : comme l'impossibilité de déclarer des constructeurs publics ou des membres statiques.

Les types de classe en java

```
interface MonInterface {
   void maMethode();
}

// Utilisation d'une classe anonyme
MonInterface monObjet =
            new MonInterface() {
                @Override
                public void maMethode() {
                        System.out.println("Hello!");
                 }
            };
MethodeUtiliseClasseAnonyme(monObjet);
```

Utilisation avec
variable intermédiaire

```
List<String> maListe = Arrays.asList("banana", "apple",
"orange");
Collections.sort(maListe,
                new Comparator<String>() {
                      @Override
                       public int compare(String s1, String s2) {
                              return s1.compareTo(s2);
                                      // Tri par ordre alphabétique
                              }
                }
);
```

Utilisation directe
comme argument

```
public class Presentation {
        public void sePresenter(){
        System.out.println(« Hello world »);
        }
}
```

```
public class Etudiant {
        public static void main(String [] args) {
                Presentation p = new Presentation();
                p.sePresenter();
        }
}
```

Hello world

```
public class Presentation {
        public void sePresenter(){
        System.out.println(« Hello world »);
        }
}
```

```
public class Etudiant {
        public static void main(String [] args) {
                Presentation p = new Presentation() {
                        @Override
                        public void sePresenter(){
                                System.out.println(« Bonjour !!»);
                        }
                };
        }
        p.sePresenter();    }
```

Bonjour

```
public class Presentation {
        public void sePresenter(){
        System.out.println(« Hello world »);
        }
}
```

```
public class Etudiant {
        public static void main(String [] args) {
                Presentation p = new Presentation(){
                        @Override
                        public void sePresenter()
                        {  System.out.println(« Bonjour !!»);   }
                };
                Presentation p2 = new Presentation();
                p.sePresenter();
                p2.sePresenter();
        }}
```

Bonjour
Hello world

```
public class Personne {
        public void sePresenter(){
            System.out.println(« Salut, je suis une personne »);
        }
        …
}
```

```
public class Etudiant extends Personne {
        @Override
        public void sePresenter(){
            System.out.println(« Salut, je suis un etudiant »);
        }
}
```

```
public class Professeur extends Personne {
         @Override
        public void sePresenter(){
            System.out.println(« Salut, je suis un professeur »);
        }
}
```

```
public class Administrateur extends Personne {
         @Override
        public void sePresenter(){
         System.out.println(« Salut, je suis un administrateur »);
        }
}
```

```
public class Personne {
        public void sePresenter(){
        System.out.println(« Salut, je suis une personne »);
        }
}
```

```
public class Main {
    public static void main(String [] args) {
        Personne etudiant = new Personne(){
            @Override
            public void sePresenter(){
            System.out.println(« Salut, je suis un etudiant »);
            }
            };
        Personne professeur = new Personne(){
            @Override
            public void sePresenter(){
            System.out.println(« Salut, je suis un professeur »);
            }
            };
         Personne administrateur = new Personne(){
            @Override
            public void sePresenter(){
            System.out.println(« Salut, je suis un administrateur »);
            }
            };
            etudiant.sePresenter();
     }}
```

```
abstract class Forme {
    protected double dimension1;
    protected double dimension2;
    public Forme(double dimension1, double dimension2) {
        this.dimension1 = dimension1;
        this.dimension2 = dimension2;
    }
    public abstract double calculerAire();
}

class Rectangle extends Forme {
    public Rectangle(double largeur, double hauteur) {
                super(largeur, hauteur);
                }
    @Override
    public double calculerAire() {
                return dimension1 * dimension2;
                }
}
```

```
public class ClasseAnonymeExemple {
    public static void main(String[] args) {
        List<Forme> formes = Arrays.asList(
                new Rectancgle(5,5),
                new Forme(4, 3) {
                        @Override
                        public double calculerAire() {
                            return dimension1 * dimension2 * 0.5;
                                    // Triangle rectangle
                            }
                }
        );
        for (Forme forme : formes) {
            System.out.println("Aire : " + forme.calculerAire());
        }
    }
}
```

Privilégier la redéfinition dans une classe anonyme / la redéfinition de la méthode au sein de la classe

- **Implémentations simples et ponctuelles:** Lorsque vous avez besoin d'une implémentation rapide d'une interface et que cette implémentation n'est utilisée qu'à un seul endroit dans votre code.
- **Objets temporaires:** Pour créer des objets qui ne vivent que le temps d'une opération et qui ne nécessitent pas d'être réutilisés.
- **Lambdas et expressions fonctionnelles:** Les classes anonymes sont souvent utilisées en lien avec les lambdas pour des traitements concis et fonctionnels.

Privilégier la redéfinition de la méthode au sein de la classe / la redéfinition dans une classe anonyme

- **Réutilisabilité:** Lorsque vous avez besoin de réutiliser l'implémentation à plusieurs endroits dans votre code.
- **Polymorphisme:** Si vous souhaitez créer plusieurs classes qui implémentent la même interface pour un comportement variable.
- **Tests unitaires:** Il est plus facile de tester une classe séparée qu'une classe anonyme.
- **Héritage:** Si vous souhaitez créer une hiérarchie de classes qui implémentent l'interface.
- **Lisibilité:** Pour des implémentations plus complexes, une classe séparée peut améliorer la lisibilité du code.

```
Set<String> bibliotheque = new TreeSet<>();
```

On fait rien ➔ l'ordre naturel

Classe de comparaison

```
Comparator<Livre> comparateurLivre = new
ComparaisonLivre();
Set<Livre> bibliotheque = new TreeSet<>(comparateurLivre);
```

Redéfinition au sein de la classe Livre

```
Set<Livre> bibliotheque = new TreeSet<>();   // il utilise la
méthode compare() redéfinit
```

Utilisation d'une classe de comparaison : ComparaisonLivre

```
import java.util.Comparator;

class ComparaisonLivre implements Comparator<Livre> {
    @Override
    public int compare(Livre moi, Livre autre) {
        return moi.getNom().compareTo(autre.getNom());
    }
}
```

```
Set<Livre> bibliotheque = new TreeSet<>(
    new Comparator<Livre>() {
        @Override
        public int compare(Livre livre1, Livre livre2) {
            return livre1.getNom().compareTo(livre2.getNom());
        }
    }
);
```

Utilisation d'une classe anonyme

➔Pas besoin d'utiliser toute une classe

➔Une classe que nous l'utiliserons qu'une seul fois

➔Éviter d'encombrer notre projet

Exemple de classe anonyme

Exercice 1:

1- Créer une classe Produit : ID, nom, description, prix.

2- Créer la classe GestionProduit. Cette classe devra trier et afficher les produits selon leurs prix.
NB: Utiliser un LinkedList pour stocker les produits et une classe anonyme pour trier avec Comparator.

3- Supprimer les produits ayant un prix supérieur à 300dh en utilisant la méthode removeIf qui implémente l'interface fonctionnelle Predicate

Exercice 2:

1- Créer une classe qui me permet de trier une liste de String (ArrayList) par ordre alphabétique, toujours en utilisant la classe anonyme qui implémente l'interface fonctionnelle Comparator.

2- Afficher votre liste en utilisant forEach et qui utilise l'interface fonctionnelle Consumer.

Corrections!

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

class Produit {
    String nom;
    double prix;

    // Constructeur, getters et setters
}
public class Example {
    public static void main(String[] args) {
        List<Produit> produits = new LinkedList<>();
        produits.add(new Produit("Téléphone", 500));
        produits.add(new Produit("Ordinateur", 1000));
        produits.add(new Produit("Tablette", 300));
```

```
        // Tri par prix croissant en utilisant une classe anonyme
        Collections.sort(produits, new Comparator<Produit>() {
            @Override
            public int compare(Produit p1, Produit p2) {
                return Double.compare(p1.getPrix(), p2.getPrix());
            }
        });

        // Afficher les produits triés
        for (Produit produit : produits) {
            System.out.println(produit.nom + " : " + produit.prix + " dhs");
        }
        produits.removeIf(new Predicate<Produit>() {
                                    @Override
                                    public boolean test(Produit p) {
                                            return p.getPrix()>300;
                                    }
                            });
}}
```

```
import java.util.*;

public class TrierMots {
    public static void main(String[] args) {
        List<String> mots = new ArrayList<>(List.of("PAPA", "MAMAN", "MOI", "TOI", "NOUS", "MERCI"));
        Collections.sort(mots, new Comparator<String>(){
                                                    @Override
                                                    public int compare(String s1, String s2) {
                                                                                    return s1.compareTo(s2);
                                                     }
                        });
        mots.forEach(new Consumer<String>() {
                                                    @Override
                                                    public void accept(String s) {
                                                                                    System.out.println(s);
                                                    }
                        });
    }
}
```

Les interfaces fonctionnelles

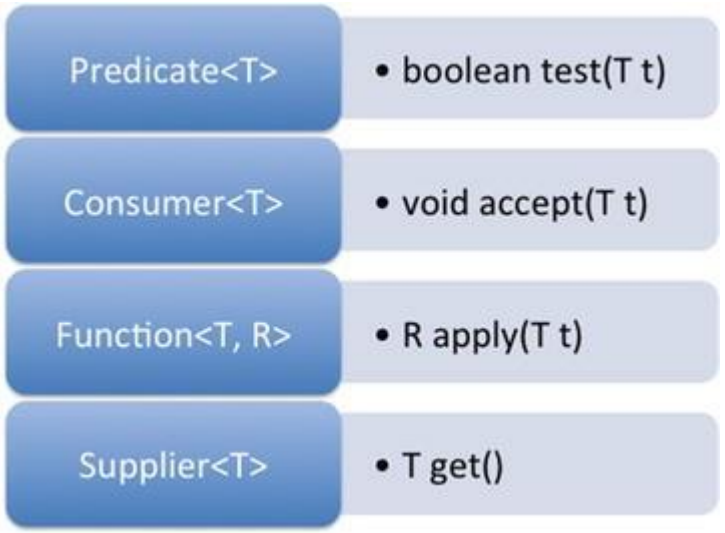
Les interfaces fonctionnelles

- Comparator est une interface fonctionnelle
- Interface = contrat
- Interface fonctionnelle = contient une seule méthode abstraite. Cependant, elle peut contenir des méthodes par défaut et statiques qui ont une implémentation, en plus de la méthode unique non implémentée.

```
public interface InterfaceFonctionnel {
        public void show();                                  // Méthode abstraite
        public default int addition(int a, int b) {          // Méthode par défaut
                return a + b;
        }
        public static int soustraction(int a, int b) {       // Méthode static
                return a - b;
        }
}
```

- Une interface fonctionnelle peut être implémentée par une expression Lambda

Les interfaces fonctionnelles prédéfinies



Prend un T en entrée et retourne un booleen

Prend un T en entrée et ne retourne rien

Prend un T en entrée et retourne un R

Ne prend rien en entrée et retourne un T

Ajouter l'annotation @FunctionalInterface
Le compilateur marque une erreur si on spécifie plusieurs méthodes abstraites

On peut créer nos propres interfaces fonctionnelles

```
@FunctionalInterface
interface MaFonction {
   String appliquer(int x);
}
```

Les expressions Lambda

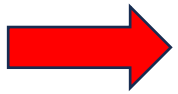
Expressions Lambda

- Une interface fonctionnelle peut-être instanciée en utilisant lambda expression.
- Lambda expression permet d'instancier et fournir une implémentation d'une interface fonctionnelle.
- La déclaration du lambda expression nécessite la précision de l'interface fonctionnelle associée.
- Les arguments d'entrée, la valeur de retour de l'expression lambda doivent respecter la signature de la méthode de l'interface fonctionnelle associée.
- Le symbole « -> » permet de séparer entre les arguments d'entrée et le corps du traitement proposé par lambda expression.

```
NomInterface nomLambdaExpression = ([ArgumentsEntree]) -> {// traitement}
```

Interface fonctionnelle → expression lambda

```
new InterfaceFonctionnelle() {
        @Override
        public TypeRetour maMethode(arguments)
                                {corps}
}
```



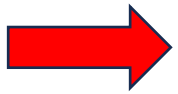
```
(T arg1, T arg2) -> {corps}
```

```
Set<Livre> bibliotheque = new TreeSet<>(
    new Comparator<Livre>() {
        @Override
        public int compare(Livre livre1, Livre livre2) {
            return (livre1. getGenre().compareTo(livre2.getGenre()));
        }
    }
);
```

```
Set<Livre> bibliotheque = new TreeSet<>(
    (Livre livre1, Livre livre2) -> {
            return (livre1. getGenre().compareTo(livre2. getGenre()));
        }
);
```

Interface fonctionnelle → expression lambda

(T arg1, T arg2) -> {corps}



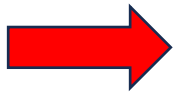
(arg1, arg2) -> {corps}

```
Set<Livre> bibliotheque = new TreeSet<>(
    (Livre livre1, Livre livre2) -> {
            return (livre1. getGenre().compareTo(livre2. getGenre()));
        }
);
```

```
Set<Livre> bibliotheque = new TreeSet<>(
    (livre1, livre2) -> {
            return (livre1. getGenre().compareTo(livre2. getGenre()));
        }
);
```

Interface fonctionnelle → expression lambda

(arg1, arg2) -> {corps}



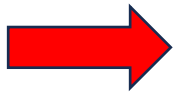
(arg1, arg2) -> return (qqch);

```
Set<Livre> bibliotheque = new TreeSet<>(
    (livre1, livre2) -> {
            return (livre1. getGenre().compareTo(livre2. getGenre()));
        }
);
```

```
Set<Livre> bibliotheque = new TreeSet<>(
    (livre1, livre2) -> return (livre1. getGenre().compareTo(livre2. getGenre()));
);
```

Interface fonctionnelle → expression lambda

(arg1, arg2) -> return (qqch);



(arg1, arg2) -> qqch;

```
Set<Livre> bibliotheque = new TreeSet<>(
    (livre1, livre2) -> {
            return (livre1. getGenre().compareTo(livre2. getGenre()));
        }
);
```

```
Set<Livre> bibliotheque = new TreeSet<>(
    (livre1, livre2) -> livre1. getGenre().compareTo(livre2. getGenre());
);
```

Expressions Lambda : exemples

```
Boutton1.addActionListener(new ActionListener () {
        @Override
        public void actionPerformed(ActionEvent event) {
                setBackground(Color.BLUE);
        }
});
```

boutton1.addActionListener(event -> setBackground(Color.BLUE));

```
taskList.execute(new Runnable() {
        @Override
        public void run() {
                traitementTacheDeFond(imageName);
        }
});
```

taskList.execute(() -> traitementTacheDeFond(imageName));

Les interfaces fonctionnelles prédéfinies

`Supplier<T>`	`() -> "This is a message"`
`Consumer<T>` `BiConsumer<T, U>`	`s -> System.out.println(s)` `(element, sink) -> sink.accept(element)`
`Predicate<T>` `BiPredicate<T, U>`	`s -> s.length() > 0` `(i1, i2) -> i1 > i2`
`Function<T, R>` `UnaryOperator<T>`	`s -> s.length()` `s -> s.toUpperCase()`
`BiFunction<T, U, R>` `BinaryOperator<T>`	`(word, sentence) -> sentence.indexOf(word)` `(i1, i2) -> i1 + i2`

Interface fonctionnelle → expression lambda

Exercice 1:

1- Créer une classe Produit : ID, nom, description, prix.

2- Créer la classe GestionProduit. Cette classe devra trier et afficher les produits selon leurs prix.
NB: Utiliser un LinkedList pour stocker les produits et une classe anonyme pour trier avec Comparator.

3- Supprimer les produits ayant un prix supérieur à 300dh en utilisant la méthode removeIf qui implémente l'interface fonctionnelle Predicate

Exercice 2:

1- Créer une classe qui me permet de trier une liste de String (ArrayList) par ordre alphabétique, toujours en utilisant la classe anonyme qui implémente l'interface fonctionnelle Comparator.

2- Afficher votre liste en utilisant forEach et qui utilise l'interface fonctionnelle Consumer.

Corrections!

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

class Produit {
    String nom;
    double prix;

    // Constructeur, getters et setters
}
```

```
public class Example {
    public static void main(String[] args) {
        List<Produit> produits = new LinkedList<>();
        produits.add(new Produit("Téléphone", 500));
        produits.add(new Produit("Ordinateur", 1000));
        produits.add(new Produit("Tablette", 300));

        // Tri par prix croissant en utilisant une classe anonyme
        Collections.sort(produits, (p1, p2) -> Double.compare(p1.getPrix(), p2.getPrix()));
        produits.removeIf(p -> p.getPrix()>300);

        // Afficher les produits triés
        for (Produit produit : produits) {
            System.out.println(produit.nom + " : " + produit.prix + " dhs");
        }   }}
```

```
import java.util.*;

public class TrierMots {
    public static void main(String[] args) {
        List<String> mots = new ArrayList<>(List.of("PAPA", "MAMAN", "MOI", "TOI", "NOUS", "MERCI"));
        Collections.sort(mots, (s1, s2) -> s1.compareTo(s2));
        mots.forEach(s -> System.out.println(s));
    }
}
```