Chapitre 6

Expressions Lambdas

Introduction

- Les expressions Lambdas sont introduites en Java à partir de la version 8
- Ils permettent de réduire et de simplifier le code :
  - Meilleur lisibilité
  - Meilleur maintenabilité

Interfaces fonctionnelles

- Une interface fonctionnelle est une interface qui ne comporte qu'une seule méthode abstraite.
- Elle peut cependant contenir des méthodes par défaut ou statiques.
- Exemples :
  - L'interface Comparator (méthode compare)
  - L'interface ActionListener (méthode actionPerformed)
  - L'interface Runnable (méthode run)

Principe

- Remplacer le code d'implémentation de la méthode de l'interface :

```
new InterfaceFoncionelle() {
 @Override
 public TypeRetour méthode (liste des arguments) {corps}
```

- Par :

```
(liste des arguments) -> {corps}
```

Exemple 1

- L'exemple :

```
btnQuitter.addActionListener ( new ActionListener(){
          @Override
          public void actionPerformed(ActionEvent e) {
                 System.exit(0);
          }
        }
      );
```

- Devient :

```
btnQuitter.addActionListener (
  (ActionEvent e)-> {System.exit(0)} );
```

- Ou encore :

```
btnQuitter.addActionListener ( e-> System.exit(0));
```


Exemple 2

- sort() est une méthode statique de la classe Arrays qui permet de trier des tableaux d'objets.
  - Arrays.sort(tab, new CompNom()); //exp. avec comparator
- Nous devons implémenter l'interface Comparator et donc redéfinir la méthode « compare ».
- Supposons que nous voulons trier un tableau de chaines de caractères (tabChaines) selon la longueur des chaines et non pas par ordre alphabétique.

Exemple 2

- Le code classique serait :

```
Arrays.sort(tabChaines,new Comparator<String>(){
    @Override
    public int compare(String s1, String s2){
     return (s1.length()-s2.length());
    }
   }
);
```

- Avec l'expression Lambda on aura :

```
Arrays.sort(tabChaines,
 (String s1, String s2)-> {return (s1.length()-s2.length());}
);
```


Simplification

- Le compilateur peut faire une déduction des types des arguments et donc pas besoin de les préciser.
- Si le corps de la méthode contient une seule instruction on peut :
  - se débarrasser des accolades aussi.
  - Se débarrasser de l'instruction return
- L'exemple précédent devient :

```
Arrays.sort(tabChaines,
   (s1, s2)-> s1.length()-s2.length();
);
```

Simplification

- Dans le cas ou nous disposons d'un seul argument, les parenthèses des arguments peuvent être omises.
- Dans le premier exemple du ActionListener nous aurons :

```
btnQuitter.addActionListener (
    e-> System.exit(0);
);
```

- Remarque : Dans le cas ou il n y a pas d'arguments, on doit mettre des parenthèses vide comme suit :
  - () -> traitement;

Exemple

- Soit la classe Etudiant :

```
public class Etudiant {
   private String CNE;
   private String nom;
   private double moyenne;

   public Etudiant(String CNE, String nom, double moyenne) {
      this.CNE = CNE;
      this.nom = nom;
      this.moyenne = moyenne;
   }
// on suppose que getters & setters et toString() sont fournis
}
```

- Nous souhaitons remplir un TreeSet avec des objets Etudiant à trier par moyenne

Solution 1 avec classe explicite

```
public class CompMoyenne implements Comparator<Etudiant>{
  @Override
@Override
    public int compare(Etudiant e1, Etudiant e2) {
      double m1=e1.getMoyenne(), m2=e2.getMoyenne();
        if (m1==m2) return 0;
          else if  (m1<m2)        return -1;
              else return 1; } }
```

```
import java.util.*;
public class ComparableTest {
 public static void main(String[] args) {
Comparator cmp=new CompMoyenne();
TreeSet etudiants = new TreeSet(cmp);
  etudiants.add(new Etudiant("2014/354", "Ali",15.5));
  etudiants.add(new Etudiant("2014/358", "Omar",12.5 ));
  etudiants.add(new Etudiant("2014/398", "Taha",13.6));
  etudiants.add(new Etudiant("2014/253", "Anass",14.3));
 System.out.println(etudiants);
 } }
```


Solution 2 avec classe anonyme

```
import java.util.*;
public class ComparableTest {
 public static void main(String[] args) {
  TreeSet etudiants = new TreeSet(new Comparator<Etudiant>(){
    @Override
      public int compare(Etudiant e1, Etudiant e2) {
       double m1=e1.getMoyenne(), m2=e2.getMoyenne();
           if (m1==m2) return 0;
             else if  (m1<m2)          return -1;
                  else return 1;  }
});
  etudiants.add(new Etudiant("2014/354", "Ali",15.5));
  etudiants.add(new Etudiant("2014/358", "Omar",12.5 ));
  etudiants.add(new Etudiant("2014/398", "Taha",13.6));
  etudiants.add(new Etudiant("2014/253", "Anass",14.3));
 System.out.println(etudiants);
 }
}
```

Solution 3 avec expressions lambdas

```
import java.util.*;
public class ComparableTest {
 public static void main(String[] args) {
TreeSet etudiants = new TreeSet((e1,e2)->{
          double m1=e1.getMoyenne(), m2=e2.getMoyenne();
            if (m1==m2) return 0;
              else if  (m1<m2)         return -1;
                    else return 1;
});
  etudiants.add(new Etudiant("2014/354", "Ali",15.5));
  etudiants.add(new Etudiant("2014/358", "Omar",12.5 ));
  etudiants.add(new Etudiant("2014/398", "Taha",13.6));
  etudiants.add(new Etudiant("2014/253", "Anass",14.3));
  System.out.println(etudiants);
 }
}
```


Solution 4 avec expressions lambdas

```
import java.util.*;
public class ComparableTest {
 public static void main(String[] args) {
 Comparator<Etudiant>  cmp = (e1,e2)->{
      double m1=e1.getMoyenne(), m2=e2.getMoyenne();
            if (m1==m2) return 0;
              else if  (m1<m2)         return -1;
                    else return 1;
};
TreeSet etudiants = new TreeSet(cmp);
etudiants.add(new Etudiant("2014/354", "Ali",15.5));
etudiants.add(new Etudiant("2014/358", "Omar",12.5 ));
etudiants.add(new Etudiant("2014/398", "Taha",13.6));
etudiants.add(new Etudiant("2014/253", "Anass",14.3));
System.out.println(etudiants); }
}
```

Annotation @FunctionalInterface

- Si nous décidons de créer une interface avec une seule méthode abstraite, nous pouvons ajouter l'annotation @FunctionalInterface.
- Cette annotation est optionnelle mais utile car elle déclenche une erreur si l'interface n'est pas fonctionnelle

Annotation @FunctionalInterface

- Exercice d'application :
  - Écrivez une expression lambda qui prend deux entiers en paramètres et renvoie leur somme.
- Solution :

```
@FunctionalInterface
interface Addition {
   int add(int a, int b);
}
```

```
public class Main {
   public static void main(String[] args) {
      Addition addition = (a, b) -> a + b;
      System.out.println(addition.add(5, 3)); // 8
   }
}
```

Interfaces fonctionnelles de l'API Java

- Le package ***java.util.function*** définit un ensemble d'interfaces fonctionnelles.
- Ces interfaces fonctionnelles représentent :
  - Les fonctions unaires (à un argument) et binaires (à deux arguments),
  - Les opérateurs (fonctions dont le type de retour est identique au type de l'argument) unaires et binaires,
  - Les prédicats (fonctions dont le type de retour est booléen) unaires et binaires,
  - Les producteurs et consommateurs.

Fonctions unaires et binaires

- Interface Function :
  - Représente une fonction à un argument.
  - Le type de cet argument et le type de retour de la fonction sont les paramètres de type de cette interface, nommés respectivement T et R
  - Se compose des méthodes :
    - **R apply(T t)** : *méthode abstraite* qui prend un argument de type T et renvoie un résultat de type R.
    - **andThen(Function<? super R, ? extends V> after)** : *méthode par défaut* qui retourne une fonction composée qui applique d'abord la fonction actuelle à son entrée, puis applique la fonction fournie au résultat.

Fonctions unaires et binaires

- **compose(Function<? super V, ? extends T> before)** : *méthode par défaut* qui retourne une fonction composée qui applique d'abord la fonction fournie à son entrée, puis applique la fonction actuelle au résultat.
- **identity()** : *méthode statique* qui qui retourne une fonction Function qui renvoie toujours son argument d'entrée.

- Exemple avec l'interface Function

```
import java.util.function.Function;
public class FunctionExample {
   public static void main(String[] args) {
      // 1. Méthode apply (abstraite)
      Function<String, Integer> lengthFunction = s -> s.length();
      int length = lengthFunction.apply("Hello, World!");
      System.out.println("Longueur de la chaîne : " + length);
```


Fonctions unaires et binaires

```
// 2. Méthodes andThen et compose (par défaut)
      Function<Integer, Integer> multiplyBy2 = x -> x * 2;
      Function<Integer, String> intToString = Object::toString;
      // Composée : d'abord multiplier par 2, puis convertir en chaîne
      Function<Integer, String> composedFunction = multiplyBy2.andThen(intToString);
      // Appliquer la fonction composée
      String result = composedFunction.apply(5);
      System.out.println("Résultat : " + result);
      // 3. Méthode identity (statique)
      Function<Integer, Integer> identityFunction = Function.identity();
      int identityResult = identityFunction.apply(42);
      System.out.println("Résultat de la fonction identity : " + identityResult);
   }
}
```

```
Longueur de la chaine : 13
Résultat : 10
Résultat de la fonction identity : 42
```

Fonctions unaires et binaires

- Interface BiFunction :
  - Représente une fonction à deux arguments.
  - Les types sont donnés par les paramètres de type T et U, le type du résultat étant donné par R.
  - Elle se compose des méthodes :
    - **R apply(T t, U u)** : *méthode abstraite* qui prend deux arguments de type T et U et renvoie un résultat de type R.
    - **andThen(Function<? super R, ? extends V> after)** : *méthode par défaut* qui retourne une fonction composée qui applique d'abord la fonction actuelle à son entrée, puis applique la fonction fournie au résultat.

Fonctions unaires et binaires

- Exemple avec BiFunction pour la concaténation de deux chaînes de caractères et renvoi du résultat.

```
import java.util.function.BiFunction;

public class BiFunctionExample {
   public static void main(String[] args) {
      // Exemple d'utilisation de BiFunction pour concaténer deux chaînes
      BiFunction<String, String, String> concatenateStrings = (s1, s2) -> s1 + s2;

      // Appel de la fonction avec deux arguments
      String result = concatenateStrings.apply("Hello, ", "World!");

      // Affichage du résultat
      System.out.println(result);  // affiche : Hello, World!
   }
}
```

Opérateurs unaires et binaires

- Interface UnaryOperator :
  - Représente un opérateur unaire, c-à-d un cas particulier de fonction à un argument dont le type de l'argument et le type de retour sont identiques.
  - public interface UnaryOperator<T> extends Function<T, T>{}
  - Exemple : fonction de calcul de valeur absolue sur les réels
  - UnaryOperator<Double> abs = x -> Math.abs(x);
  - Utilisation :
    - abs.apply(-1.2); // => 1.2
    - abs.apply(Math.PI); // => 3.1415…

Opérateurs unaires et binaires

- Interface BinaryOperator :
  - Représente un opérateur binaire, c-à-d un cas particulier de fonction à deux arguments dont le type des arguments et le type de retour sont identiques :
  - public interface BinaryOperator<T> extends BiFunction<T, T, T>{}
  - Exemple : l'addition sur les entiers est un opérateur binaire qui pourrait se définir ainsi :
  - BinaryOperator<Integer> plus = (x, y) -> x + y;
  - Utilisation : plus.apply(8, 9); // => 17

Prédicats unaires et binaires

- Interface Predicate :
  - représente un prédicat à un argument, c-à-d une fonction à un argument qui retourne true ou false.
  - public interface Predicate<T> { public boolean test(T x); }
  - Exemple : prédicat déterminant si une chaîne de caractères est vide
  - Predicate<String> stringIsEmpty = x -> x.isEmpty();
  - Utilisation :
    - stringIsEmpty.test(""); // => true
    - stringIsEmpty.test("not empty!"); // => false

Prédicats unaires et binaires

- Interface BiPredicate
  - similaire à Predicate, représente un prédicat à deux arguments
  - public interface BiPredicate<T, U> { public boolean test(T x, U y); }
  - Exemple : prédicat testant si une chaîne est la re
  - présentation textuelle d'un objet pourrait se définir ainsi :
  - BiPredicate<String, Object> isTextReprOf = (s, o) -> s.equals(o.toString());
  - Utilisation :
    - isTextReprOf.test("1", 1); // => true
    - isTextReprOf.test("2", "a"); // => false

Prédicats unaires et binaires

- Remarque : Les interfaces Predicate et BiPredicate offrent des méthodes par défaut permettant d'obtenir de nouveaux prédicats à partir de prédicats existants.
- Ces méthodes sont :
  - **and**, qui calcule la conjonction de deux prédicats,
  - **or**, qui calcule la disjonction de deux prédicats, et
  - **negate**, qui calcule la négation d'un prédicat.
- Utilisation :

```
Predicate<Integer> p = x -> x <=0;
Predicate<Integer> q = x -> x <= 5;
Predicate<Integer> r = p.and(q);   // 0 <= x && x <= 5
Predicate<Integer> s = p.or(q);    // 0 <= x || x <= 5
Predicate<Integer> t = p.negate(); // 0 > x
```


Producteurs et consommateurs

- Interface Supplier :
  - représente un fournisseur de valeurs, c-à-d une fonction sans argument retournant une valeur d'un type donné
  - public interface Supplier<T> { public T get(); }
  - Exemple : un fournisseur constant ne produisant que l'entier 0
    - Supplier<Integer> zero = () -> 0;
    - Utilisation : zero.get(); // => 0

Producteurs et consommateurs

- Interface Consumer :
  - représente un consommateur de valeur, c-à-d une fonction à un argument ne retournant rien.
  - public interface Consumer<T> { public void accept(T x);}
  - Exemple : une fonction imprimant une chaîne à l'écran est un consommateur de chaîne
    - Consumer<String> printString = s -> { System.out.println(s); };
    - Utilisation : printString.accept("hello"); // affiche hello

Références de méthodes

- sont une fonctionnalité qui simplifie l'utilisation d'expressions lambda lorsqu'elles appellent des méthodes déjà existantes.
- Plutôt que de fournir une implémentation directe dans l'expression lambda, vous pouvez utiliser une référence de méthode pour référencer une méthode existante.
- Il existe plusieurs types de références de méthode en Java

Références de méthodes

- Références de méthode statique :
  - **Syntaxe** : NomDeLaClasse::nomDeLaMethodeStatique
  - Exemple :
    - Function<String, Integer> parseIntFunction = Integer::parseInt;
- Références de méthode d'instance :
  - **Syntaxe** : instance::nomDeLaMethode
  - Exemple :
    - List<String> names = Arrays.asList("Ali", "Omar", "Taha");
    - names.forEach(System.out::println);

Références de méthodes

- Références de méthode d'instance de classe arbitraire
  - **Syntaxe** : NomDuType::nomDeLaMethode
  - Exemple :
    - List<String> names = Arrays.asList("Ali", "Omar", "Taha");
    - names.sort(String::compareToIgnoreCase);
- Références de constructeur :
  - **Syntaxe** : NomDeLaClasse::new
  - Exemple :
    - Supplier<List<String>> listSupplier = ArrayList::new;