Chapitre 14

Les expressions lambda

1. Définition et principes
2. Syntaxe d’écriture et Exemple
3. Les interfaces fonctionnelles prédéfinies
4. Quelques dérivées des expressions lambda

1 Définition et principes

2 Syntaxe d’écriture et Exemple

3 Les interfaces fonctionnelles prédéfinies

4 Quelques dérivées des expressions lambda

Définition et principes (1/3)

- Supposons l'interface Java suivant :

```
public interface ManipMontant {

    double TAUX_TVA = 0.20d;

    double calculTTC(double montant) ;
    double calculHT(double montant) ;


}
```

- Classiquement en Java, pour exploiter une telle interface, il faut déclarer une classe qui l'implémente et qui donne un corps à sa/ses méthode.s déclarée.s :

```
public class OperationTVA implements ManipMontant {

    public double calculTTC(double montant){
        return montant*(1+TAUX_TVA) ;
    }

    public double calculHT(double montant){
        return montant/(1+TAUX_TVA) ;
    }

}
```

Ainsi, la classe OperationTVA peut donc être exploitée : OperationTVA obj = new OperationTVA(); => obj.calculTTC(15.7d) = 18.84 et obj.calculHT(9.8) = 8.16

Définition et principes (2/3)

- Les **expressions lambda** ont été introduites dans Java à partir de sa version 8 et s'inspire des concepts utilisés dans la programmation fonctionnelle, pour s'affranchir quelque peu de la logique expliquée dans la précédente slide

- Le terme **expression** dans « expression lambda » n'a rien à voir avec le formalisme des *expressions logiques* et *conditionnelles* étudiées dans les chapitres 6 et 7 sur les opérateurs et les structures conditionnelles.

~~(a > b) && (c == d)~~

~~!(a > b)~~

- **Une Expression Lambda** est un mécanisme rapide et efficace (un raccourci) permettant d'implémenter une interface sans s'encombrer de la définition d'une classe pour pouvoir l'exploiter (comme à la slide page 4 précédente)

  ★ La seule et unique condition est que cette interface doit être une **interface fonctionnelle**

Définition et principes (3/3)

- Rappel – **Interface fonctionnelle** :
  > Interface pouvant contenir des déclarations de constantes, de méthodes *static* et *default*, **mais disposant d'une et une seule unique méthode abstraite**
  > Java propose de déclarer une telle interface en y ajoutant l'annotation @FonctionalInterface pour bien la marquer comme telle :

```
@FunctionalInterface
public interface MonInterface {

    <type-retour> monUniqueMethodeAbstraite(<paramètres>) ;

    //éventuelles déclaration de constantes

    //éventuelles méthodes statiques ou default
}
```

- **Une Expression Lambda** se réduit à une fonction/méthode créée à la volée dans une classe, mais correspondant à l'implémentation de la seule et unique méthode abstraite déclarée dans une interface fonctionnelle.
  > Son avantage est que, elle permet de :
  - réduire le nombre de lignes de code dans un programme, car sa structuration est peu verbeuse
  - pouvoir utiliser une méthode comme un objet que l'on peut manipuler à sa guise et même la passer en paramètre à une autre méthode

1 Définition et principes

2 Syntaxe d’écriture et Exemple

3 Les interfaces fonctionnelles prédéfinies

4 Quelques dérivées des expressions lambda

Syntaxe d'écriture et exemple

- Une expression lambda s'écrit avec le format suivant :

  (paramètres d'entrée) -> { corps }

  > paramètres d'entrée : paramètres que prend la méthode abstraite déclarée dans l'interface fonctionnelle

  > corps : implémentation donnée à la méthode abstraite

- Exemples :

  A/ 0 paramètre d'entrée :

  `() -> System.out.println("Pas de paramètre");` (exp lambda qui ne prend aucun paramètre et imprime un texte en sortie sur la console)

  B/ 1 paramètre d'entrée :

  `x -> System.out.println("paramètre : " + x);` (exp lambda qui prend un paramètre x et l'imprime en sortie)

  `(String x) -> System.out.println("paramètre : " + x);` (exp lambda qui prend un paramètre x et l'imprime en sortie)

  C/ 2 paramètres d'entrée :

  `(int x, long y) -> System.out.println("paramètres : " + x + ", " + y);` (exp lambda qui prend deux paramètres x et y et les imprime en sortie)

  `(x, y) -> { System.out.println("paramètres : " + x + ", " + y); return x+y; }` (exp lambda à 2 paramètres x et y, les imprime et retourne leur somme)

  `(x, y) -> x+y;` (exp lambda qui prend 2 paramètres x et y, et retourne leur somme)

Syntaxe d'écriture et exemple

- Au regard des exemples précédents, une décortication des formes d'écriture des expressions lambda s'impose :

> Le type des paramètres d'entrée : la déclaration du type de données des paramètres d'entrée est facultative. Car, en fonction des valeurs effectives de ceux-ci, le compilateur déterminera leurs types. Le compilateur fait l'inférence de types.

> La parenthèse autour des paramètres d'entrée :

- est obligatoire si l'expression lambda n'a pas de paramètre d'entrée. *Cf. exemple A/*
- est obligatoire si l'expression lambda a deux paramètres d'entrée ou plus. *Cf. exemples C/*
- facultative si l'expression lambda a un unique paramètre d'entrée. *Cf. exemple 1 de B/*

=> Mais si on ajoute le type de cet unique paramètre, alors elle devient obligatoire. *Cf exemple 2 de B/*

> Les accolades du corps :

- sont obligatoires si le corps contient plus d'une instruction. *Cf. exemple 2 de C/*
- sont facultatifs si le corps contient une unique instruction. *Cf. exemples A/, B/ et exemples 1 et 3 de C/*

> Le mot clé return :

- est obligatoire et doit être la dernière instruction, lorsque le corps contient plusieures instructions et que l'expression lambda doit retourner un résultat. *Cf. exemple 2 de C/*
- est facultatif, mais sous-entendu, lorsque le corps contient une unique instruction d'évaluation. *Cf. exemple 3 de C/*

Exemple (1/2)

Considérons l'interface fonctionnelle suivante :

```java
@FunctionalInterface
public interface Operation {

    double calcul(double x, double y);

}
```

Ecrire un programme Java qui exploite cette interface fonctionnelle pour réaliser divers calculs (addition, soustraction, multiplication, division, puissance) avec ses deux paramètres.

```java
public class Main {

   private static double calculer(double x, double y, Operation op) {
       return op.calcul(x, y);
   }

   public static void main(String args[]) {

       //exp lambda avec la déclaration de type et sans accolades
       Operation addition = (double x, double y) -> x + y;

       //exp lambda sans déclaration de type et sans accolades
       Operation soustraction = (x, y) -> x - y;

       //exp lambda avec déclaration de type et 'return' entre accolades
       Operation multiplication = (double x, double y) -> {
          return x * y;
       };

       //exp lambda avec déclaration de type, sans 'return' et sans les accolades
       Operation division = (double x, double y) -> x / y;

       //exp lambda sans déclaration de type, sans 'return' et sans accolades
       Operation puissance = (x, y) -> Math.pow(x, y);

       System.out.println("9 + 3 = " + calculer(9, 3, addition));
       System.out.println("9 - 3 = " + calculer(9, 3, soustraction));
       System.out.println("9 x 3 = " + calculer(9, 3, multiplication));
       System.out.println("9 / 3 = " + calculer(9, 3, division));
       System.out.println("9 ^ 3 = " + calculer(9, 3, puissance));
   }

}
```

Résultat :

```
9 + 3 = 12.0
9 - 3 = 6.0
9 x 3 = 27.0
9 / 3 = 3.0
9 ^ 3 = 729.0
```

Exemple (2/2)

S'il fallait résoudre ce même problème en faisant du Java sans expression lambda, ça donnerait ceci…

```java
@FunctionalInterface
public interface Operation {

    double calcul(double x, double y);

}
```

```java
public class Multiplication implements Operation {

    double calcul(double x, double y){
       return x * y;
    }

}
```

```java
public class Addition implements Operation {

    double calcul(double x, double y){
       return x + y;
    }

}
```

```java
public class Soustraction implements Operation {

    double calcul(double x, double y){
       return x - y;
    }

}
```

```java
public class Division implements Operation {

    double calcul(double x, double y){
       return x / y;
    }

}
```

```java
public class Puissance implements Operation {

    double calcul(double x, double y){
       return Math.pow(x, y);
    }

}
```

```java
public class Main {

   private static double calculer(double x, double y, Operation op) {
       return op.calcul(x, y);
   }

   public static void main(String args[]) {

       //instanciation objet addition
       Addition addition = new Addition();

       //instanciation objet soustraction
       Soustraction soustraction = new Soustraction();

       //instanciation objet multiplication
       Multiplication multiplication = new Multiplication();

       //instanciation objet division
       Division division = new Division();

       //instanciation objet puissance
       Puissance puissance = new Puissance();

       System.out.println("9 + 3 = " + calculer(9, 3, addition));
       System.out.println("9 - 3 = " + calculer(9, 3, soustraction));
       System.out.println("9 x 3 = " + calculer(9, 3, multiplication));
       System.out.println("9 / 3 = " + calculer(9, 3, division));
       System.out.println("9 ^ 3 = " + calculer(9, 3, puissance));
   }

}
```

Résultat :

```
9 + 3 = 12.0
9 - 3 = 6.0
9 x 3 = 27.0
9 / 3 = 3.0
9 ^ 3 = 729.0
```

1 Définition et principes

2 Syntaxe d’écriture et Exemple

3 Les interfaces fonctionnelles prédéfinies

4 Quelques dérivées des expressions lambda

Les interfaces fonctionnelles prédéfinies (1/3)

- Java fournit plusieurs interfaces fonctionnelles par défaut dans le package *java.util* dont on peut immédiatement implémenter et utiliser à l'aide d'expression lambda dans toute classe

  > Leurs buts est de couvrir un ensemble de cas d'utilisation classiques que l'on peut rencontrer en programmation

  > Voici quelques unes les plus utilisées :

  - **Predicate :** déclare une méthode *test* qui prend un paramètre et retourne *true* ou *false* si une condition est vérifiée
  - **Consumer :** déclare une méthode *accept* qui prend un paramètre et ne retourne rien
  - **Supplier :** déclare une méthode *get* qui ne prend pas de paramètre, mais retourne un résultat
  - **Function :** déclare une méthode *apply* qui prend un paramètre et retourne un résultat
  - **BiFunction :** déclare une méthode *apply* qui prend deux paramètres de types éventuellement différents et retourne un résultat ayant un type donné
  - **BinaryOperator :** extension du BiFunction avec le même type pour les paramètres et la valeur de retour
  - **Comparable :** *cf*. chap 13, déclare la méthode *compareTo* pour le tri
  - **Comparator :** *cf*. chap 13, déclare la méthode *compare* pour le tri

  > A noter que ces interfaces sont toutes des interfaces génériques (cf. cours sur la généricité – chapitre 4)

Les interfaces fonctionnelles prédéfinies (2/3)

- Exemples d'utilisation des expressions lambda sur les interfaces fonctionnelles prédéfinies

Predicate :

```
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

}
```

Predicate<String> isEmpty = (str) -> str.isEmplty();

Consumer :

```
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);

}
```

Consumer<String> csr = (str) -> System.out.println(str);

Supplier :

```
@FunctionalInterface
public interface Supplier<T> {

    T get();

}
```

Supplier<Integer> a = () -> 12589;

Function :

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

}
```

Function<Integer, Double> aireDisque = (4) -> 3.14*Math.pow(4, 2);

Les interfaces fonctionnelles prédéfinies (3/3)

- Exemples d'utilisation des expressions lambda sur les interfaces fonctionnelles prédéfinies

BiFunction :

```
@FunctionalInterface
public interface BiFunction<T, U, R> {

    R apply(T t, U u);

}
```

```
String str = "Hello";
Integer i  = 2;
BiFunction<String, Integer, List> list = (str, i) -> {
    List<Object> a = new ArrayList<>();
    a.add(str);     a.add(i);
    return a;
};
```

System.out.println(list.get(0)) ⇔ Hello

System.out.println(list.get(1) ⇔ 2

BinaryOperator :

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, T, T> {

    void apply(T t);

}
```

```
String str1 = "Bonjour ";
String str2 = "Monsieur ";
String str3 = "Dupont";
BinaryOperator<String, String, String> concat = (str1, str2, str3) -> {
  return str1 + str2 + str3;
};
```

System.out.println(concat) ⇔ Bonjour Monsieur Dupont

1 Définition et principes

2 Syntaxe d’écriture et Exemple

3 Les interfaces fonctionnelles prédéfinies

4 Quelques dérivées des expressions lambda

Quelques dérivées des expressions lambda

- Les **expressions lambda** ont entrainé beaucoup de changements dans le langage Java sur la manipulation de certains objets, la syntaxe de certaines instructions et même la manière d'écrire des programmes de façon globale

- Nous présentons ici :
  - ★ La boucle ***foreach*** sur les collections de type List/ArrayList
  - ★ Une autre forme d'écriture de la structure conditionnelle ***switch/case*** avec une syntaxe apparenté au lambda

  - ★ Dans le chapitre suivant, nous présenterons les ***streams***, un autre apport majeur qui a modifié de façon importante le visage du langage

La boucle *foreach*

Considérons la collection ArrayList suivante, déclarée et initialisée :

```
ArrayList<String> myList = new ArrayList<String>() {
   {
     add("Alpha");
     add("Beta");
     add("Gamma");
   }
};
```

- Avant Java 8, pour parcourir une telle liste, on était obligé d'utiliser une boucle ***for*** beaucoup trop verbeuse :
  Exemple :

```
for (String str : myList) {
   System.out.println(str);
}
```

ou

```
for (int i = 0; i < myList.size(); i++) {
   System.out.println(myList.get(i));
}
```

- Aujourd'hui, c'est beaucoup plus simple avec la boucle *foreach :*

```
myList.forEach(str -> System.out.println(str));
```

Cette boucle *foreach* est une méthode public de la classe ArrayList qui prend en paramètre une expression lambda implémentant un **Consumer**. Son but est alors d'itérer tour à tour sur chaque élément (ici, ***str*** situé à gauche du symbole ->), et d'effectuer un traitement sur cet élément (ici, `System.out.println(str)` situé à droite du symbole ->)

Sa syntaxe de définition est :

```
public void forEach(Consumer<E> action)
```

Puisque *foreach* prend un objet de type **Consumer** en paramètre :

- L'objet passé en paramètre est une instance de classe ou une expression lambda qui implémente la méthode *void accept(E e)*

- L'instruction *foreach* ne peut pas retourner de résultat.
  > On ne peut pas faire par exemple :

```
String s = myList.forEach(str -> { return str; });
```

La structure conditionnelle *switch/case*

Considérons que nous voulons écrire une structure conditionnelle capable de tester si un cas de situation survient parmi plusieurs cas possible :

- Avant Java 14, voici un exemple de ce qu'on aurait écrit :

```
switch (jour) {

    case "Lundi" :
    System.out.println(1);
    break;

    case "Mardi" :
    System.out.println(2);
    break;

    case "Mercredi" :
    System.out.println(3);
    break;

    case "Jeudi" :
    System.out.println(4);
    break;

    case "Vendredi" :
    System.out.println(5);
    break;

    default:
    System.out.println(-1);
    break;
}
```

- Aujourd'hui, cela se traduit simplement en :

```
switch (jour) {

   case "Lundi" -> System.out.println(1);

   case "Mardi" -> System.out.println(2);

   case "Mercredi" -> System.out.println(3);

   case "Jeudi" -> System.out.println(4);

   case "Vendredi" -> System.out.println(5);

   default -> {
     System.out.println(-1);
   }

}
```

- Les instructions *break* ne sont plus obligatoires
- Les expressions ci-dessus, n'en sont en réalité pas des expressions lambdas, car elle ne sont pas adossées sur les interfaces fonctionnelles, mais il n'en demeure pas moins vrai que c'est cette notion qui a inspiré cette forme d'écriture du switch


SUCCESS