Les lambdas

Une lambda est une fonction anonyme (c'est-à-dire une fonction qui est déclarée sans être associée à un nom). Le terme *lambda* est emprunté à la méthode formelle du lambda-calcul. Les fonctions lambda (ou plus simplement les lambdas) sont utilisées dans la programmation fonctionnelle. Elles permettent d'écrire des programmes plus conscis et elles permettent de créer des closures (*fermetures*).

Java est uniquement un langage orienté objet. Cela signifie que les fonctions n'existent pas en Java. Cependant, depuis la version 8, il est possible d'écrire des lambdas. Nous verrons qu'il s'agit en fait d'un sucre syntaxique.

Syntaxe des lambdas

En Java, les lambdas s'écrivent sous la forme :

```
(paramètres) -> { corps }
```

Les parenthèses ne sont pas obligatoires si la lambda n'a qu'un seul paramètre. Si le compilateur peut inférer le type des paramètres alors il n'est pas obligatoire de déclarer le type des paramètres. Les accolades peuvent être omises si la lambda n'a qu'une instruction et si le contexte le permet. Si le corps de la lambda ne contient qu'une instruction, on peut omettre le point-virgule à la fin de l'instruction.

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

(String s) -> { System.out.println(s); }

() -> 42
```

Beaucoup de méthodes acceptent en paramètre une lambda. C'est notamment le cas de la méthode forEach déclarée par l'interface Iterable. On peut donc effectuer un traitement sur chaque élément d'une collection avec une lambda.

```
Collection<String> collection = new ArrayList<>();
collection.add("un");
collection.add("deux");
collection.add("trois");
```

```
collection.forEach(e -> System.out.println(e));
```

Si une lambda doit retourner une valeur et qu'elle ne comporte qu'une instruction, alors le mot-clé **return** peut être omis. Dans ce cas, c'est le resultat de l'évaluation de l'expression qui sera implicitement retourné.

Par exemple, la méthode de tri sort déclarée par l'interface List peut recevoir en paramètre une lambda pour comparer les éléments de la liste deux à deux. Cette lambda prend deux paramètres correspondant à deux éléments de la liste et retourne un nombre négatif si le premier est plus petit que le second, zéro si les deux éléments sont identiques et un nombre positif si le premier est plus grand que le second.

```
List<Integer> liste = new ArrayList<>();
liste.add(1);
liste.add(2);
liste.add(3);
liste.add(4);

// tri la liste en plaçant en premier les nombres pairs
liste.sort((e1, e2) -> (e1 % 2) - (e2 % 2));

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

Lambda et closure

Une lambda définit une *closure* (fermeture), c'est-à-dire qu'elle définit un environnement lexical constitué de toutes les variables et de tous les attributs qu'elle capture dans son environnement d'exécution. Le corps d'une lambda peut donc accéder au contenu d'une variable déclarée dans la méthode englobante.

```
List<String> prenoms = new ArrayList<>();
prenoms.add("Murielle");
prenoms.add("Jean");
prenoms.add("Michelle");

List<String> helloList = new ArrayList<>();

prenoms.forEach(e -> helloList.add("Hello " + e));
```

```
// [Hello Murielle, Hello Jean, Hello Michelle]
System.out.println(helloList);
```

Comme pour la déclaration de classe anonyme, une lambda ne peut pas modifier le contenu d'une variable ou d'un paramètre. Par contre, il n'est pas nécessaire de déclarer comme **final** une variable ou un paramètre pour pouvoir y accéder dans une lambda. Le compilateur émettra une erreur si on tente de modifier une variable ou un paramètre capturé par la closure.

```
List<Integer> liste = new ArrayList<>();
liste.add(1);
liste.add(2);
liste.add(3);
liste.add(4);

int i = 0;
liste.forEach(e -> i += e); // ERREUR DE COMPILATION : la variable i ne peut pas être modifiée
```

Les interfaces fonctionnelles

Comme Java ne supporte pas la notion de fonction, les lambdas correspondent à des implémentations d'interface. Une interface qui ne déclare qu'une seule méthode abstraite peut être implémentée par une lambda.

Si nous déclarons l'interface ci-dessous :

```
package com.cgi.udev;

public interface OperationSimple {

  int calculer(int i);

}
```

Alors partout où le programme attend une implémentation de cette interface, il est possible de fournir une lambda :

```
OperationSimple os = i -> 2 * i;

System.out.println(os.calculer(10)); // 20
```

Une interface qui ne déclare qu'une seule méthode abstraite est appelée *interface fonctionnelle*.

L'annotation FunctionalInterface peut être utilisée lors de la déclaration de l'interface. Elle permet d'identifier pour le compilateur que cette interface peut être implémentée par des lambdas. Le compilateur peut ainsi contrôler que l'interface ne comporte qu'une seule méthode abstraite et signaler une erreur dans le cas contraire.

```
package com.cgi.udev;

@FunctionalInterface
public interface OperationSimple {

  int calculer(int i);

}
```

Il est donc très simple d'introduire des lambdas même avec des bibliothèques et des applications qui ont été développées avant puis portées vers Java 8.

Afin d'éviter aux développeurs de créer systématiquement leurs interfaces, le package java.util.function déclare les interfaces fonctionnelles les plus utiles. Par exemple, l'interface java.util.function.IntUnaryOperator permet d'utiliser une interface fonctionnelle qui accepte un entier en paramètre et qui retourne un autre entier. Nous pouvons nous en servir pour définir un régulateur de vitesse dans une classe *Voiture*.

```
package com.cgi.udev;

import java.util.function.IntUnaryOperator;

public class Voiture {

  private int vitesse;
  private IntUnaryOperator regulateurDeVitesse = v -> v;

  public void accelerer(int deltaVitesse) {
    this.vitesse = regulateurDeVitesse.applyAsInt(this.vitesse + deltaVitesse);
  }

  public void setRegulateurDeVitesse(IntUnaryOperator regulateur) {
    this.regulateurDeVitesse = regulateur;
  }
```

```
  public int getVitesse() {
    return vitesse;
  }

}
```

```
Voiture v = new Voiture();
v.setRegulateurDeVitesse(vitesse -> vitesse > 110 ? 110 : vitesse);

v.accelerer(90);
System.out.println(v.getVitesse()); // 90

v.accelerer(90);
System.out.println(v.getVitesse()); // 110
```

L'opérateur :: de référence de méthode

Plutôt que de déclarer une lambda pour implémenter une interface fonctionnelle, il est possible d'indiquer directement une référence de méthode si la signature est compatible avec la méthode de l'interface fonctionnelle.

Si nous reprenons un exemple vu précédemment :

```
Collection<String> collection = new ArrayList<>();
collection.add("un");
collection.add("deux");
collection.add("trois");

collection.forEach(e -> System.out.println(e));
```

La méthode forEach attend en paramètre une instance qui implémente l'interface fonctionnelle Consumer. L'interface Consumer déclare la méthode *accept* qui prend un type **T** en paramètre et ne retourne rien. Si maintenant nous comparons cette signature avec celle la méthode println, cette dernière attend un objet en paramètre et ne retourne rien. La signature de println est compatible avec celle de l'interface fonctionnelle Consumer. Donc, plutôt que de déclarer une lambda, il est possible d'utiliser l'opérateur **::** pour passer la référence de la méthode println :

```
Collection<String> collection = new ArrayList<>();
collection.add("un");
collection.add("deux");
```

```
collection.add("trois");

collection.forEach(System.out::println); // passage de la référence de la méthode
```

> Notez que dans l'exemple ci-dessus, la référence de la méthode println est celle de l'instance de l'objet contenu dans l'attribut *out*.

Il est également possible de référencer les constucteurs d'une classe. Cela aboutira à la création d'un nouvel objet à chaque appel. Par exemple, nous pouvons utiliser l'interface fonctionnelle Supplier. Cette interface fonctionnelle peut être implémentée en utilisant un constructeur sans paramètre. Ainsi, si nous définissons une classe *Voiture* avec un constructeur sans paramètre :

```
package com.cgi.udev;

public class Voiture {

    public Voiture() {
      // ...
    }

}
```

Nous pouvons utiliser la référence de ce constructeur pour créer une implémentation de l'interface fonctionnelle Supplier :

```
Supplier<Voiture> garage = Voiture::new;

Voiture v1 = garage.get(); // crée une nouvelle instance
Voiture v2 = garage.get(); // crée une nouvelle instance
```

Les constructeurs peuvent être référencés grâce à la syntaxe

```
NomDeLaClasse::new
```

Revision #1
Created 2025-02-10 23:24:56 UTC by Nicolas
Updated 2025-02-10 23:25:57 UTC by Nicolas