Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

2021 – 2022
ATL

ATL – Ateliers Logiciels

lambdas

Les expressions lambda en Java

Dans ce document vous trouverez un introduction à la notion d'*expressions lambda* en Java.

Le code est disponible via git à l'adresse :
`https://git.esi-bru.be/ATL/lambda.git`

Table des matières

Version du document
29 novembre 2021

1 Interface fonctionnelle

Une **expression lambda** en Java — ou une *fonction anonyme* ou encore une ***closure*** — est un raccourci visant à alléger l'écriture des instances de classes anonymes lorsque ces classes implémentent une interface n'ayant qu'une seule méthode.

Ces interfaces sont appelées **interface fonctionnelle** (*functional interface*)

Les expressions lambda ont été introduites avec Java 8. Elles apparaissent le plus souvent dans des codes caractéristiques tels que la gestion des évènements dans une interface graphique ou le traitement des collections via les *stream*.

Elles ont cette allure :

```
(Type param1, Type param2) -> statement

() -> System.out.println("Hi");
(Integer i1, Integer i2) -> return i1 + i2;
(String s) -> {
   // do something with string s
   // and consume it
};
```

Rappel

Cette notion a été introduite en DEV2 - Développement I, ne pas hésiter à se rafraichir la mémoire.

Prenons deux exemples,

- ▷ la gestion des évènements liés à un bouton et ;
- ▷ le tri d'une collection.

1.1 Gestion des évènements liés à un bouton

Prenons la classe `EventHandlerTest` , une application JavaFX présentant un bouton qui permet d'afficher le message `"Hello World!"` dans un `Label` . La gestion de l'action sur le bouton est ici déléguée à une classe anonyme. Cette classe contient une seule méthode : `handle(ActionEvent event)` . Cette méthode définit ce qui se passe lorsque le bouton est actionné.

code/Lambda/src/esi/atl/lambda/EventHandlerExample1823.java

```
1
2 btn.setOnAction(new EventHandler<ActionEvent>() {
3    @Override
4    public void handle(ActionEvent event) {
5       textf.setText("Hello world!");
6    }
7 });
```

Le code peut être simplifié à l'aide d'une expression lambda :xpression lambda :

```
btn.setOnAction((ActionEvent event) -> {textf.setText("Hello world!");});
```

L'interface `EventHandler` est une interface fonctionnelle.

1.2 Tri d'une collection

Considérons d'extrait de code suivant :

code/Lambda/src/esi/atl/lambda/ComparatorExample0722.java

```
1
2  public static void main(String[] args) {
3      var list = new ArrayList<String>();
4      list.add("Welcome");
5      list.add("to");
6      list.add("hello world");
7      Collections.sort(list, new Comparator<String>() {
8          @Override
9          public int compare(String word1, String word2) {
10             return word1.length() - word2.length();
11         }
12     });
13     System.out.println(list);
14 }
```

Ce programme crée une liste de `String` et la trie en fonction de la longueur de ses éléments. Le tri s'effectue en utilisant la fonction utilitaire `sort` de la classe `Collections` qui a pour en-tête :

```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

> Besoin d'un rafraichissement pour comprendre `<? super T>` ? Il y a un TD dédié aux génériques (*generics*).

Dans notre cas, le paramètre générique `T` est `String`. L'interface `Comparator<String>` est une interface fonctionnelle, elle possède une seule méthode :

```
public int compare(String first, String second)
```

Dans notre exemple, le comparateur est instancié aux lignes 7 à 12 à l'aide d'une classe anonyme. Vous pouvez simplifier le code grâce à l'utilisation d'une expression lambda :

```
Collections.sort(list, (String w1, String w2) -> {
    return w1.length() - w2.length();
});
```

En résumé

- ▷ Une interface fonctionnelle est une interface ne définissant qu'une seule méthode abstraite.
- ▷ Le compilateur détermine les interfaces qui sont fonctionnelles.
- ▷ Il est encouragé de signaler les interfaces fonctionnelles avec l'annotation `@FunctionalInterface`
- ▷ Une interface fonctionnelle peut avoir plusieurs méthodes à condition qu'une seule de celles-ci soit abstraite. Les autres méthodes sont des méthodes statiques ou des default methods [a] .
- ▷ Une expression lambda peut-être utilisée pour définir une classe anonyme d'un type fonctionnel et instancier un objet de cette classe.

a. https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

2 Syntaxe des expressions lambda

La syntaxe de base d'une expression lambda a la forme suivante :

```
(Type1 arg1, Type2 arg2...) -> statement
```

Nous avons une liste de paramètres typés entre parenthèses, la flèche `->` , et, *statement* qui peut être une instruction ou un bloc d'instructions.

Question 1

Reprenons les classes `ComparatorExample` et `EventHandlerExample` . Parmi les expressions lambda suivantes, lesquelles sont acceptées par le compilateur ?

1. Suppression du type d'un des paramètres :

```
(w1, String w2) -> {
    return w1.length() - w2.length();
}
```

2. Suppression du type d'un des paramètres :

```
(String w1, w2) -> {
   return w1.length() - w2.length();
}
```

3. Suppression du type des 2 paramètres :

```
(w1,  w2) -> {
   return w1.length() - w2.length();
}
```

4. Suppression du type du paramètre :

```
(event) -> {textf.setText("Hi !");}
```

5. Suppression du type et des parenthèses des paramètres :

```
w1,  w2 -> {
   return w1.length() - w2.length();
}
```

6. Suppression des parenthèses du paramètre :

```
ActionEvent event -> {
   textf.setText("Hi!");
}
```

7. Suppression du type et des parenthèses du paramètre :

```
event -> {textf.setText("Hi!");}
```

8. Suppression du `return` :

```
(w1,  w2) -> {w1.length() -
    ↪ w2.length();}
```

9. Suppression des accolades :

```
(event) -> textf.setText("Hi!");
```

10. Suppression des accolades et du point virgule :

```
(event) -> textf.setText("Hi!")
```

11. Suppression du `return` et des accolades :

```
(w1,  w2) ->  w1.length() -
    ↪ w2.length();
```

12. Suppression du `return` , des accolades et du point virgule :

```
(w1,  w2) ->  w1.length() -
    ↪ w2.length()
```

Théorie

Les règles de simplification des expressions lambda sont couramment utilisées. Elles permettent d'obtenir un code plus concis et plus lisible.

▷ Le type des paramètres est facultatif lorsqu'il peut être inféré par le compilateur, ce qui est généralement le cas. On peut écrire :

```
(w1, w2) -> { return w1.length() - w2.length();}
```

Autre exemple :

```
(event) -> {textf.setText("Hello world!");}
```

- S'il y a un seul paramètre et que son type est inféré alors il n'est pas obligatoire de mettre ce paramètre entre parenthèses.
  Exemple :

  ```
  event -> {textf.setText("Hello world!");}
  ```

- Il est autorisé que le corps d'une expression lambda soit composé d'une expression (p.ex. : $b*b-4*a*c$ ou `word.length() < 17`). Dans ce cas, il n'y a pas d'accolades, et pas de point virgule puisqu'il s'agit d'une expression. La valeur retournée est la valeur de l'expression.
  Exemple :

  ```
  (w1, w2) -> w1.length() - w2.length()
  ```

  Notez l'absence de `return` , de point virgule, et d'accolades.
  ou encore :

  ```
  event -> textf.setText("Hello world!")
  ```

  Dans ce dernier cas, l'expression est un appel à la méthode `setText` . Notez l'absence d'accolades et l'absence de point virgule. L'expression lambda n'a pas de valeur de retour car la méthode `setText` n'en possède pas (`void`), cela correspond bien à la méthode `handle` de l'interface fonctionnelle :

  ```
  public void handle(ActionEvent event)
  ```

Reprenons nos 2 exemples de départ. Après simplification, nous obtenons pour le tri :

```
Collections.sort(list, (w1, w2) -> w1.length() - w2.length() );
```

et, pour la gestion du bouton :

```
btn.setOnAction(  event -> textf.setText("Hello world!") );
```

Remarque `this`

Une différence fondamentale entre une classe anonyme et une expression lambda est le sens du mot-clef `this` .

1. Dans une classe anonyme, `this` fait référence à l'objet de la classe anonyme.
2. Dans une lambda, `this` fait référence à l'objet courant de la classe dans laquelle est définie la lambda.

3 Références de méthodes et de constructeurs

Il est courant que le corps d'une lambda soit un simple appel à une autre méthode. Lorsque c'est le cas, les références de méthodes représentent une alternative aux expressions lambda avec une syntaxe concise et facile à lire. Il y a plusieurs cas à considérer :

1. appel à une méthode statique ;
2. appel à une méthode d'un objet spécifique ;

3. appel à une méthode d'un objet non spécifié mais dont la classe est spécifiée. Une méthode d'instance ;

Pour une référence à un constructeur, le principe est semblable aux références de méthodes à la différence que ce n'est pas une méthode qui sera exécutée mais un constructeur qui sera invoqué [1] .

3.1 Méthode statique

Supposons que nous ayons une classe `Person` et que la classe `MyUtil` possède une méthode statique `prettyPrint(Person p)` . On peut écrire :

```
var listPerson = new ArrayList<Person>();
listPerson.add(...); //adding some persons
listPerson.forEach(MyUtil::prettyPrint);
```

La référence de méthode de la dernière ligne utilise une référence à la méthode statique `prettyPrint` .

C'est équivalent à :

```
listPerson.forEach(p -> MyUtil.prettyPrint(p));
```

3.2 Méthode d'un objet spécifique

```
var list = new ArrayList<String>();
list.add("Welcome");
list.add("to");
list.add("hello world");
list.forEach(System.out::println);
```

La référence de méthode de la dernière ligne utilise une référence à la méthode `println` de l'objet `System.out` (l'attribut publique statique `out` de la classe `System`). Le compilateur infère les paramètres de l'expression lambda à partir du contexte (l'interface fonctionnelle).

Cette instruction est équivalente à :

```
list.forEach(s -> System.out.println(s));
```

3.3 Méthode d'instance

```
String[] stringArray = { "Barbara", "James", "Mary", "John",
        "Patricia", "Robert", "Michael", "Linda" };
Arrays.sort(stringArray, String::compareToIgnoreCase);
```

Ici, la méthode `compareToIngoreCase` est une méthode d'instance (pas statique). Notez que la syntaxe est exactement la même que pour une référence à une méthode statique décrite au point 3.1.

La méthode `compareToIngoreCase` de la classe `String` prend en paramètre une autre `String` .

Cette instruction est équivalente à :

```
Arrays.sort(stringArray, (first ,second) -> first.compareToIgnoreCase(second));
```

1. https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html

Le compilateur utilise le premier paramètre `first` comme objet courant, et le second objet, `second`, comme paramètre. Contrairement au cas précédent, l'objet `first` n'est pas spécifique, ce sera successivement différents éléments du tableau à trier.

3.4 Référence de constructeur

Une référence de constructeur permet d'instancier une interface fonctionnelle en invoquant un constructeur.

La syntaxe est

```
ClassName::new
```

Par exemple,

```
String::new
```

est équivalent à l'expression lambda

```
() -> new String()
```

4 Interfaces fonctionnelles standards

Parmi les interfaces de la librairie standard, certaines sont fonctionnelles. Par exemple, `Iterable`, `Runnable` et `Comparable`.

Java introduit des interfaces fonctionnelles génériques afin de simplifier l'utilisation des lambdas. Nous résumons quelques unes de ces interfaces dans le tableau ci-dessous.

Interface fonctionnelle	Paramètres	Retour	Méthode
`Predicate<T>`	`T`	`boolean`	`test`
`BiPredicate<T, U>`	`T, U`	`boolean`	`test`
`Consumer<T>`	`T`	`void`	`accept`
`Supplier<T>`	aucun	`T`	`get`
`Function<T,R>`	`T`	`R`	`apply`
`BiFunction<T,U,R>`	`T,U`	`R`	`apply`
`UnaryOperator<T>`	`T`	`T`	`apply`
`BinaryOperator<T>`	`T, T`	`T`	`apply`

Les prédicats sont essentiellement des expressions booléennes. Un `Consumer<T>` consomme un élément de type `T`, un `Supplier<T>` fournit une valeur de type `T`. Une `Function<T, R>` transforme un élément de type `T` en un élément de type `R`. Si les types `T` et `R` sont les mêmes on utilisera un `UnaryOperator`.

Illustration :

```
Predicate<String> p1 = (word) -> word.charAt(0) == 'A';
Predicate<Person> p2 = (person) -> person.getAge() <18;
Consumer<String> c1 = (word) -> System.out.println(word + " " + word);
Consumer<Person> c2 = (person) -> person.setName("");
Function<String, Person> f1 = (name) -> new Person(name);

// use of explicit methods from functional interface (test, apply...)
System.out.println(p1.test("Kim"));
Person john = new Person("Tony", 32);
System.out.println(p2.test(john));
c1.accept("lambda");
c2.accept(john);
```

```
// use in collections
var c = new ArrayList<Person>();
// ... fill c
var personsA = c.stream().filter(p1).collect(Collectors.toList());
c.stream().filter(p2).forEach(c1);
```

En plus de ces interfaces génériques, le package `java.util.function` [2] fournit des interfaces équivalentes pour chacun des types primitifs. Quelques unes de ces interfaces vous sont présentées dans le tableau ci-dessous.

Interface fonctionnelle	Paramètres	Retour	Méthode
`IntPredicate`	`int`	`boolean`	`test`
`DoublePredicate`	`double`	`boolean`	`test`
`IntConsumer`	`int`	`void`	`accept`
`DoubleConsumer`	`int`	`void`	`accept`
`IntFunction<T>`	`int`	`T`	`apply`
`DoubleFunction<T>`	`double`	`T`	`apply`
`ToIntFunction<T>`	`T`	`int`	`apply`
`ToDoubleFunction<T>`	`T`	`double`	`apply`
`IntToDoubleFunction`	`int`	`double`	`apply`
`DoubleToIntFunction`	`double`	`int`	`apply`

Ces interfaces génériques se rencontrent fréquemment dans les différentes librairies telles les *Collections* ou les *Streams*.

Question 2

En utilisant des classes `Shape, Circle, Rectangle, Point` , ou `Person` de votre choix, donnez un exemple pour chacune des interfaces suivantes :

1. `IntPredicate`
2. `BiPredicate<T,U>`
3. `DoubleConsumer`
4. `ToIntFunction<T>`
5. `DoubleFunction<T>`
6. `IntToDoubleFunction`

5 Accès aux variables locales

En plus des variables statiques, des paramètres et des variables déclarées dans le corps de l'expression lambda, celle-ci a accès aux variables locales, externes au corps de l'expression, qui sont effectivement finales, c'est à dire qui possèdent le modificateur `final` ou qui ne sont pas modifiées après leur première affectation.

2. https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Théorie

- ▷ Une variable locale effectivement finale est une variable :
  - ▷ qui a le modificateur `final` , ou bien
  - ▷ qui n'est pas modifiée après sa première affectation
- ▷ Une expression lambda a accès aux variables locales effectivement finales.
- ▷ Le terme *capture* désigne l'utilisation d'une variable locale externe
- ▷ Une *capturing lambda* est une expression lambda qui utilise des variables locales externes.
- ▷ Le terme *closure* désigne le code de l'expression lambda avec l'ensemble des variables locales capturées.

Dans l'exemple suivant, les paramètres `str` et `nb` de la méthode `manyPrint` , qui englobe l'expression lambda, sont des variables locales effectivement finales capturées et donc accessibles.

code/Lambda/src/esi/atl/lambda/Capture.java

```
1  public class Capture {
2     public static void manyPrint(String str, int nb) {
3        Runnable r = () -> {
4           for (int i = 0; i < nb; i++) {
5              System.out.println(str);
6              Thread.yield();
7           }
8        };
9        new Thread(r).start();
10    }
11    public static void main(String[] args) {
12       manyPrint("Expression", 10);
13       manyPrint("Lambda", 20);
14    }
15 }
```

6 Stream

Les flux (*stream*) ont été introduits avec Java 8 pour gérer les collections. Un *stream* est un flux de données, éventuellement infini en provenance d'une source (par exemple, une collection). Le *stream* ne contient pas les données, il les fait transiter afin que les méthodes qui suivent expriment un traitement sur les données. Le *stream* ne modifie pas la source des données, il exprime les traitements.

Les opérations sur les *streams* sont de deux types :

- ▷ les opérations intermédiaires qui ne font pas explicitement le traitement, mais le définissent (faire une correspondance (*map*), filtrer (*filter*)) et ;
- ▷ les opérations terminales qui déclenchent le traitement (prendre le premier élément, le plus grand, les 10 premiers, en faire une collection...)

Faites le tutorial `Streams by Examples` [3] du site `How to do in Java` pour manipuler les *streams*.

3. http://howtodoinjava.com/java-8/java-8-tutorial-streams-by-examples/