Lambda

Rémi Forax

Lambda

Lambda, ajouté en 2014 à Java 8

Façon succinte d’écrire une fonction qui implante une interface

Permet d’écrire un code générique qui prend en paramètre une lambda pour le spécialiser

Pourquoi ?

Comment remplacer les valeurs de chaines de caractères dans une liste ?

- Remplacer les null par "unknown"

```
public static void replaceAllNullByUnknown(List<String> strings) {
  for(var i = 0; i < strings.size(); i++) {
    String s = strings.get(i);
    strings.set(i, s == null? "unknown": s);
  }
}
```

Comment généraliser ce code ?

- Remplacer par la chaîne en majuscule

```
public static void replaceAllToUpperCase(List<String> strings) {
  for(var i = 0; i < strings.size(); i++) {
    String s = strings.get(i);
    strings.set(i, s.toUpperCase(Locale.ROOT));
  }
}
```

Et algorithmiquement, ce code est faux, mais on verra plus tard

Ecrire l’algorithme une seule fois

On isole le code commun

Le code commun

On écrit quoi, là ?

```
public static void replaceAll(List<String> strings, ...) {
  for(var i = 0; i < strings.size(); i++) {
    String s = strings.get(i);
    strings.set(i, ...);
  }
}
```

Et on l’appelle une fois avec

- replaceAll(strings, << s == **null**? "unknown": s >>);
- replaceAll(strings, << s.toUpperCase(Locale.ROOT) >>);

C’est quoi la syntaxe exacte, là ?

On utilise une interface !

On a deux codes de traitement différents

On créé une interface commune entre les deux

```
public interface Operation {
  String apply(String s);
}
```

Avec cela, on peut écrire l'algo générique

```
public static void replaceAll(List<String> strings, Operation op) {
  for(var i = 0; i < strings.size(); i++) {
    String s = strings.get(i);
    strings.set(i, op.apply(s));
  }
}
```

Avant Java 8

Comment on appelle replaceAll ?

On crée une classe qui implante l'interface

```
public class NullToUnknown implements Operation {
  @Override
  public String apply(String s) {
     return s == null? "unknown": s;
  }
}
```

On appelle replaceAll() avec un objet de cette classe

```
replaceAll(strings, new NullToUnknown())
```

Avant Java 8

C’est super verbeux de créer une classe quand ce que l’on veut passer c’est juste une fonction

- au point que avant Java 8, on préfèrera avoir 2 fois le même code

Java 8 ajoute les *lambda expressions*,

- une fonction anonyme qui implante une interface

Lambda expression

Algo générique

```
public static void replaceAll(List<String> strings, Operation op) {
  for(var i = 0; i < strings.size(); i++) {
    String s = strings.get(i);
    strings.set(i, op.apply(s));
  }
}
```

Interface (doit avoir une seule méthode abstraite)

```
public interface Operation {
  String apply(String s);
}
```

Lambda expressions

- replaceAll(strings, s -> s == null? "unknown": null);
- replaceAll(strings, s -> s.toUpperCase(Locale.ROOT));

Interface fonctionnelle

Une interface fonctionnelle est une interface avec une seule méthode d'instance abstraite

```
@FunctionalInterface
public interface Operation {
  String apply(String s);
}
```

L'annotation vérifie que l'interface a bien une seule méthode abstraite

Il y a sous-typage entre une lambda et une interface fonctionnelle

```
Operation op = s -> "*" + s + "*";
```

Comment marche le typage ?

```
public interface Operation {
  String apply(String s);
}
```

```
Operation op =  s - > "*" + s + "*";
```

Pour une lambda, le compilateur regarde le type dans lequel la lambda est mise

- Cela doit être une interface fonctionnelle
  - Extrait le type des paramètres de la méthode abstraite
    (String s) -> String
  - Type les paramètres de lambda et vérifie que le type de retour est le bon type

Syntaxe des lambdas

Il y a 3 syntaxes

La syntaxe diffère en fonction du nombre de paramètres

- zéro paramètre

  () - > "hello lambda"

- Un paramètre (les parenthèses sont optionnelles)

  x - > x * 2

- deux paramètres ou plus

  (x, y) - > x + y

C’est la même syntaxe que Scala ou JavaScript mais avec -> à la place de =>

Types des paramètres explicites

On peut indiquer le type des paramètres

- (String s) - > s == null? "unknown": s

Le type des paramètres n’est pas nécessaire car il est déduit à partir de la signature de la méthode abstraite de l’interface fonctionnelle

- Mais ça aide a avoir des messages d’erreur plus simples lorsqu’il y a un problème de typage
  - Pratique pour débugger

Lambda block

On peut ouvrir un bloc avec ‘{‘ et ‘}’ si on a plusieurs instructions

Comme dans un case de switch

```
Operation op = s - > {
  // il y a plusieurs instructions
  // donc il faut un return
  return ...
};
```

Capture des variables locales

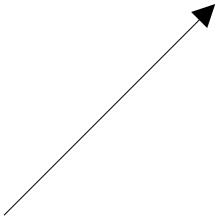
Capture de la valeur de la variable

Une lambda peut utiliser les variables de la méthode dans laquelle elle est déclarée

si la variable est assignée une seule fois
(si la variable est “effectivement final”)

Exemple

```
public static void appendAll(List<String> strings, String suffix) {
  replaceAll(strings, s -> s + suffix);
}
```



Ici, “suffix” est un paramètre, c’est bien une variable assignée une seule fois

Variable de boucle

La capture ne marche pas avec les variables de boucle

car la variable change de valeur plusieurs fois

```
for(var i = 0; i < 10; i++) {
  doSomething(x - > x + i);  // ne compile pas
}
```

mais on peut utiliser une variable intermédiaire

```
for(var i = 0; i < 10; i++) {
  var tmp = i;  // astuce !
  doSomething(x -> x + tmp);  // ok
}
```

java.util.function
(interfaces prédéfinies)

Interfaces prédéfinies

Pour replaceAll(), on a défini une interface, mais c'est verbeux

- Il existe déjà dans le package java.util.function, plein d'interfaces fonctionnelless ce qui évite d'en écrire de nouvelles

Interface de java.util.function déjà existante

```
public interface UnaryOperator<T> {
  T apply(T t);
}
```

mais cela demande d'apprendre les interfaces de java.util.function par coeur (oui !)

Algo générique

```
public static void replaceAll(List<String> strings, UnaryOperator<String> op) {
  for(var i = 0; i < strings.size(); i++) {
    String s = strings.get(i);
    strings.set(i, op.apply(s));
  }
}
```

Lambda expressions

- replaceAll(strings, s -> s == null? "unknown": null);
- replaceAll(strings, s -> s.toUpperCase(Locale.ROOT));

Package java.util.function

Interface paramétrée pour les types de fonctions usuels

- de 0 à 2 paramètres
  - Runnable, Supplier, Consumer, Function, UnaryOperator, etc
- version speciale pour les types primitifs

  IntSupplier () → int, LongSupplier () → long,
  DoubleSupplier () → double

  Predicate<T> (T) → boolean

  IntFunction<T> (int) → T

  ToIntFunction<T> (T) → int
- version spéciale si même type en paramètre et type de retour

  UnaryOperator (T) → T ou BinaryOperator (T, T) → T

  DoubleBinaryOperator (double, double) → double, …

Runnable et Supplier

java.lang.**Runnable** est équivalent à **() → void**

```
Runnable code = () -> { System.out.println("hello"); }
code.run();
```

Supplier<T> est équivalent à **() → T**

```
Supplier<String> factory = () -> "hello";
System.out.println(factory.get());
```

[Int|Long|Double]Supplier

```
IntSupplier factory = () -> 42;
System.out.println(factory.getAsInt());
```

Consumer

Consumer<T> est équivalent à **(T) → void**

```
Consumer<String> printer =
    s -> System.out.println(s);
printer.accept("hello");
```

[Int|Long|Double]Consumer

```
DoubleConsumer printer =
    d -> System.out.println(d);
printer.accept(42.0);
```

Predicate

Predicate<T> représente **(T) → boolean**

```
Predicate<String> isSmall = s -> s.length() < 5;
System.out.println(isSmall.test("hello"));
```

[Int|Long|Double]Predicate

```
LongPredicate isPositive = v -> v >= 0;
System.out.println(isPositive.test(42L));
```

Function

Function<T,U> représente **(T) → U**

```
Function<String, String> fun = s -> "hello " + s;
System.out.println(fun.apply("function"));
```

[Int|Long|Double]Function<T>

```
IntFunction<String[]> arrayCreator =
    size -> new String[size];
System.out.println(arrayCreator.apply(5).length);
```

To[Int|Long|Double]Function<T>

```
ToIntFunction<String> stringLength = s -> s.length();
System.out.println(stringLength.applyAsInt("hello"));
```

UnaryOperator

UnaryOperator<T> représente **(T) → T**

```
UnaryOperator<String> op = s -> "hello " + s;
System.out.println(op.apply("unary operator"));
```

[Int|Long|Double]UnaryOperator

```
IntUnaryOperator negate = x -> - x;
System.out.println(negate.applyAsInt(7));
```

BiPredicate et BiFunction

BiPredicate<T, U> représente **(T, U) → boolean**

```
BiPredicate<String, String> isPrefix =
    (s, prefix) -> s.startsWith(prefix);
System.out.println(isPrefix.test("hello", "hell"));
```

BiFunction<T, U, V> représente (T, U) → V

```
BiFunction<String, String, String> concat =
    (s1, s2) -> s1 + " " + s2;
System.out.println(concat.apply("hello", "Bob"));
```

BinaryOperator

BinaryOperator<T> représente **(T, T) → T**

```
BinaryOperator<String> concat =
    (s1, s2) -> s1 + " " + s2;
System.out.println(concat.apply("hello", "binop"));
```

[Int|Long|Double]BinaryOperator

```
IntBinaryOperator add = (a, b) -> a + b;
System.out.println(add.applyAsInt(40, 2));
```

Référence de méthode

Opérateur :: (*colon colon*)

Ecritures simplifiées des lambdas dans le cas où la lambda est

- une référence sur une méthode d'instance

  ToIntFunction<String> ref = String**::**length;

- une référence sur une méthode statique

  ToIntFunction<String> ref = Integer**::**parseInt;

- une référence sur une méthode d'instance avec toujours la même valeur en tant que this

  String s = ...
  IntSupplier ref = s**::**length;

- une référence sur un constructeur

  Supplier<String> ref = String**::new**;

- une référence sur la création d'un tableau

  IntSupplier<String[]> ref = String[]**::new**;

La convention de code dit que l'on doit préférer les *method références* aux lambdas

Methode d’instance vs static

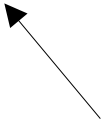
La syntaxe dans le cas d’une référence sur une méthode d’instance ou sur une méthode static est la même

Type::method

Ce qui fait la différence c’est comment “method” est déclarée dans la classe/interface Type

- Si elle est déclarée comme une méthode d’instance, c’est une réference sur une méthode d’instance
- Si elle est déclarée comme une méthode static, c’est une référénce sur une méthode static

Ne confondez pas avec Type.method qui appelle une méthode static



Il y a un “.” ici, pas un ::

Classe vs Record vs Lambda

3 façons d’écrire la même chose

Accès à x mais encapsulé, equals/hashCode/toString à la main

```
public class AddTwo implements IntUnaryOperator {
  private final int x;

  public int apply(int v) {
    return v + x;
  }
}
```

Accès à x() public, equals/hashCode/toString fourni

```
public record AddTwo(int x) implements IntUnaryOperator {
  public int apply(int v) {
    return v + x;
  }
}
```

Pas de nom de classe, pas d’accès à x, equals/hashCode/toString de java.lang.Object

```
public static IntUnaryOperator addTwo(int x) {
  return v -> v + x;
}
```

Classe vs Lambda

Vue fonctionnelle

- Une classe est une façon verbeuse d’écrire des lambdas

Vue objet

- Les lambdas sont les classes du pauvre
  (l’accès aux valeurs capturées n’est pas possible)

Les lambdas dans l’API des collections

Méthodes sur List

list.**forEach**(consumer) appelle le consumer avec chaque élément

```
List.of("hello", "list").forEach(System.out::println);
```

list.**replaceAll**(unaryOperator) remplace chaque élement par le résultat de l'appel de l'opérateur sur l'élément

```
Arrays.asList("hello").replaceAll(s - > s.toUpperCase(Locale.ROOT));
  // [Hello]
```

list.**toArray**(intFunction) renvoie un tableau contenant tous les éléments

```
List.of("hello", "list").toArray(String[]::new)
```

Méthodes sur List (2)

list.**removeIf**(predicate) supprime les élements pour lesquels le predicate est vrai

```
var list = new ArrayList<>(List.of("a", "b", "c"));
list.removeIf(s -> s.startsWith("b"));
  // [a, c]
```

list.**sort**(comparator) trie les éléments en les comparant 2 à 2

```
var list = Array.asList("foo", "bar", "baz");
list.sort((s1, s2) -> s2.compareTo(s1));
  // [foo, baz, bar]
```

Méthodes sur Map

map.**forEach**(biConsumer) appelle le biConsumer avec chaque clé/valeur

```
Map.of("a", 3, "b", 7)
   .forEach((k, v) -> System.out.println(k + ": " + v))
  // a: 3
  // b: 7
```

map.**replaceAll**(biFunction) replace les valeurs en appelant la biFonction avec chaque clé/valeur

```
var map = new HashMap<>(Map.of("a", 3, "b", 7));
map.replaceAll((k, v) -> v + 1);
  // {a=4, b=8}
```

Calcul sur les valeurs

map.**compute**(key, biFunction) appelle la biFunction pour calculer la nouvelle valeur (appelle avec la valeur à **null** si c'est un nouveau couple)

```
var map = new HashMap<>(Map.of("a", 3));
map.compute("a", (k, v) -> v + 1);
map.compute("b", (k, v) -> v == null? 0: v + 1);
 // {a=4, b=0}
```

map.**computeIfPresent**(key, biFunction) appelle la biFunction pour calculer la nouvelle valeur, ne fait rien si il n'y a pas de couple correspondant à la clé

```
var map = new HashMap<>(Map.of("a", 3));
map.computeIfPresent("a", (k, v) -> v + 1);
map.computeIfPresent("b", (k, v) -> v + 1);
 // {a=4}
```

Agréger des valeurs

map.**merge**(key, newValue, biFunction) ajoute le couple key/newValue si un couple existe pas et sinon appelle la biFunction avec les deux valeurs (celle du couple et newValue)

```
var map = new HashMap<>(Map.of("a", 13));
map.merge("a", 17, Math::max);
map.merge("b", 7, Math::max);
 // {a=17, b=7}
```

map.**computeIfAbsent**(key, function) appelle la fonction pour ajouter une valeur si il n'y a pas un couple avec la clé, et renvoie la valeur

```
var map = new HashMap<String, List<Integer>>();
map.computeIfAbsent("a", k -> new ArrayList<>()).add(1);
map.computeIfAbsent("a", k -> new ArrayList<>()).add(2);
 // {a=[1, 2]}
```

En résumé ...

Lambda

Une lambda est une façon simplifiée d’écrire une fonction qui implante une interface

- Une lambda peut capturer une valeur si la variable est effectivement finale

Permet d’écrire une méthode générique et de prendre en paramètre une lambda qui spécialise le code

```
list.replaceAll(s -> s == null? "unknown": s);
```