L3. Programmation orientée objet. Cours 7

Marie-Pierre Béal (Cours de Rémi Forax)

Lambda

Lambda

Lambda

Les lambda ont été ajoutées 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 ?

Réutiliser des patterns de code

Par exemple supprimer les éléments d'une liste si une fonction est vraie

```
public static void removeIf(List<String> list,
                            ??function?? predicate) {
  var iterator = list.iterator();
  while(iterator.hasNext()) {
    var element = iterator.next();
    if (... ??predicate(element)?? ...) {
      iterator.remove();
    }
  }
}
```

Typage == une interface

En Java, pour abstraire plusieurs comportements possibles, on utilise une interface

```
public interface Predicate<E> {
  boolean test(E element);
}
```

Comme cela on peut typer `removeIf()`

```
public static void removeIf(List<String> list,
                            Predicate<String> predicate) {
  var iterator = list.iterator();
  while(iterator.hasNext()) {
    var element = iterator.next();
    if (predicate.test(element)) {
      iterator.remove();
    }
  }
}
```

Utilisation == une classe

On envoie une instance d'une classe qui implante l'interface

```
public class StartWithFooPredicate implements Predicate<String> {
  public boolean test(String element) {
    return element.startsWith("foo");
  }
}
...
var list = ...
removeIf(list, new StartWithFooPredicate());
```

avec `removeIf()`

```
public static void removeIf(List<String> list,
                            Predicate<String> predicate) {
  ...
}
```

Écriture très lourde

Le compilateur pourrait trouver toutes les infos en bleu tout seul

```
public class StartWithFooPredicate
                                    implements Predicate<String>{
   public boolean test(String element) {
     return element.startsWith("foo");
   }
}
...
var list = ...
removeIf(list, new StartWithFooPredicate());
```

avec `removeIf()`

```
public static void removeIf(List<String> list,
                            Predicate<String> predicate) {
  ...
}
```

Simplification du code avec une lambda

Une lambda est une instance d'une classe qui implante l'interface

```
public class StartWithFooPredicate implements Predicate<String> {
  public boolean test(String element) {
    return element.startsWith("foo");
  }
}
```

```
...
var list = ...
// removeIf(list, new StartWithFooPredicate());
removeIf(list,(element) -> return element.startsWith("foo"););
```

avec `removeIf()`

```
public static void removeIf(List<String> list,
                            Predicate<String> predicate) {
  ...
}
```

Une lambda

Une lambda est une instance d'une classe qui implante l'interface

- La classe est générée à l'exécution
- Le typage vient de l'interface

```
var list = ...
removeIf(list, (element)->{return element.startsWith("foo");});
```

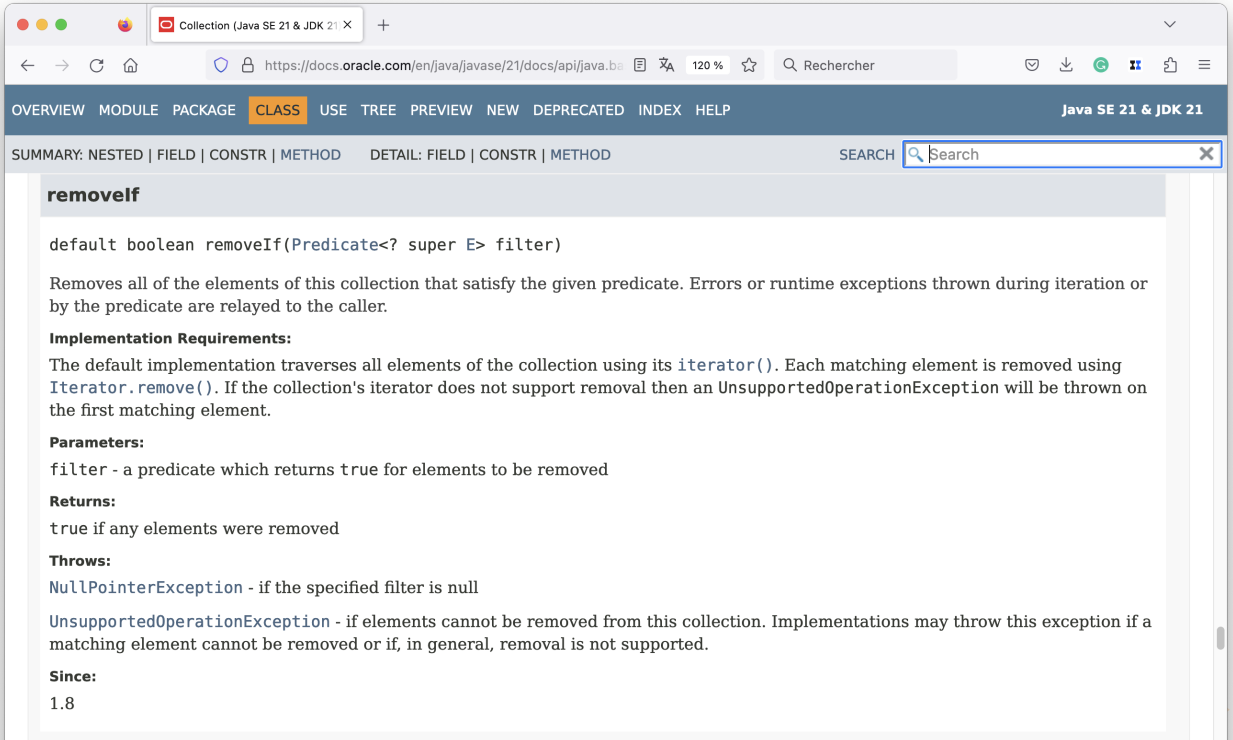
avec `removeIf()`

```
public static void removeIf(List<String> list,
                            Predicate<String> predicate) {
  ...
}
```

`list.removeIf(predicate)`

La méthode `removeIf()` existe déjà dans le JDK, c'est une méthode de `java.util.Collection`

```
var list = new ArrayList<>(List.of("foo", "bar", "baz"));
list.removeIf((s) -> s.startsWith("b"));
IO.println(list); // [foo]
```



Lambda instruction / expression

S'il n'y a qu'une expression, le bloc et le return sont optionnels

- Lambda instruction (entre accolade)

```
(element) -> { return true; }
(element) -> {
  IO.println("hello lambda");
  return true;
}
```

- Lambda expression (une expression, pas de return)

```
(element) -> true
(element) -> element.startsWith("foo")
```

Lambda : syntaxe des paramètres

Pour le cas le plus fréquent, un paramètre, les parenthèses ne sont pas obligatoires

- Zéro paramètre
  - `() -> IO.println("hello lambda")`
- Un paramètre
  - `e -> IO.println(e)`
- Deux paramètres ou plus
  - `(e1, e2) -> IO.println(e1 + " " + e2)`

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

Typage des paramètres

Les types des paramètres sont optionnels car le compilateur les déduit de la méthode abstraite de l'interface

```
Predicate<String> p = e -> e.startsWith("foo");
     // String remplace E
     // Le type inféré est (String) -> boolean
     // C'est la signature (type) de la lambda
```

avec

```
interface Predicate<E> {
  boolean test(E element);
}
```

Interface fonctionnelle

Comment cela marche si l'interface a plusieurs méthodes abstraites ?

- Cela ne marche pas, il faut qu'il y ait une seule méthode abstraite

Définition : Une interface avec une seule méthode abstraite est appelée interface fonctionnelle

- Le type d'une lambda est une interface fonctionnelle

Annotation @FunctionalInterface

Elle demande au compilateur de vérifier que l'interface a une seule méthode abstraite (donc qu'elle peut être le type d'une lambda)

```
@FunctionalInterface
public interface Predicate<E> {
  boolean test(E element);
}
```

Une interface fonctionnelle peut avoir des méthodes statiques, des méthodes par défaut, des méthodes privées mais doit avoir une seule méthode abstraite.

Typage d'une lambda

Une même expression lambda peut être typée par différentes interfaces

```
interface IntBinaryOperator {
  int apply(int x, int y);
}
interface DoubleBinaryOperator {
  double apply(double a, double b);
}
...
IntBinaryOperator op = (u, v) -> u + v;
                       // u et v sont des int
DoubleBinaryOperator op = (u, v) -> u + v;
                      // u et v sont des double
```

Typage d'une lambda (2)

On ne peut typer une lambda qu'avec une interface fonctionnelle

```
Object o = x -> x;  // compile pas, pas une interface
String s = x -> x;  // compile pas, pas une interface
var v = x -> x;     // compile pas, pas une interface
```

```
interface Foo {
  void f1();
  int f2(int i);
}
...
Foo foo = x -> x; // compile pas (2 méthodes abstraites !)
```

Type cible

Le compilateur regarde le type cible de la lambda

- Assignation

```
Predicate<String> p = e -> true;
```

- Appel de méthode (utilise le type des paramètres)

```
list.removeIf(e -> true);
```

- Type argument pour les collections

```
List.<Predicate<String>>of(e -> true, e -> false);
Map.<String, Predicate<String>>of("true", e -> true, "false", e -> false);
```

- Type de retour avec return

```
public static Predicate<String> foo() {
  return e -> true;
}
```

Fonction d'ordre supérieur

Fonction d'ordre supérieur

Définition : une fonction d'ordre supérieur est une fonction qui a au moins une des propriétés suivantes :

- elle prend en paramètre une fonction (lambda)
- elle renvoie en valeur de retour une fonction (lambda)

Dans notre exemple `removeIf()` est une fonction d'ordre supérieur

Problème

On souhaite écrire une méthode statique qui supprime tous les noms qui finissent par ".txt"

On peut écrire

- `list.removeIf(endsWithSuffix(".txt"));`

Questions :

- Quel est la signature de `endsWithSuffix()` ?
- Quel est le code de `endsWithSuffix()` ?

Fonction d'ordre supérieur

On écrit

```
list.removeIf(endsWithSuffix(".txt"));
...
static Predicate<String> endsWithSuffix(String suffix) {
  return s -> s.endsWith(suffix);
}
```

mais quel est la durée de vie de la variable "`suffix`" ?

Capture de paramètre

Une lambda est plus qu'une fonction anonyme car elle est capable de capturer la valeur des variables englobantes

```
static Predicate<String> endsWithSuffix(String suffix) {
  return s -> s.endsWith(suffix);
}
```

et qu'est ce qui se passe si on change la valeur ?

Capture de variable

Pour être capturable, une variable doit être effectivement final (le compilateur peut prouver que la variable ne change pas de valeur)

```
static Predicate<String> endsWithSuffix(String suffix) {
  var value = suffix;
  Predicate<String> predicate = e -> e.endsWith(value);
  value = value + "1";  // on change la valeur
                        // ne compile pas
  return predicate;
}
```

Si la valeur change, le compilateur refuse de compiler

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
}
```

Classe/Record vs Lambda

Deux façons d'écrire la même chose

- "suffix" est un champ accessible

```
public record EndsWithSuffix(String suffix) implements Predicate<String> {
  public boolean test(String s) {
    return s.endsWith(suffix);
  }
}
```

- pas d'accès à "suffix"

```
public static Predicate<String> endsWithSuffix(String suffix) {
  return s -> s.endsWith(suffix);
}
```

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)

Référence de méthode

Référence de méthode : simplifier encore l'écriture

Si on veut supprimer tous les élements null d'une liste, il existe

- Dans java.util.Collection :
  - `boolean removeIf(Predicate<? super E> filter)`
- `java.util.Objects.isNull(Object o)`

Donc on peut écrire

```
list.removeIf(e -> Objects.isNull(e));
```

La syntaxe : :

Permet de créer une lambda sur une méthode statique déjà existante

```
list.removeIf(Objects::isNull)
```

est équivalent à

```
list.removeIf((param) -> Objects.isNull(param))
```

avec param le paramètre

- ici : `list.removeIf(e -> Objects.isNull(e))`

: : sur les méthodes d'instance

La syntaxe marche aussi sur les méthodes d'instances

- Avec un "this" (receveur) fixé

```
String s = "foo";
list.removeIf(s::equals);
// <=> list.removeIf((param) -> s.equals(param))
```

- Avec un "this" (receveur) pris en paramètre

```
list.removeIf(String::isBlank)
// <=> list.removeIf(e -> e.isBlank())
```

: : et les constructeurs

: : marche aussi sur les constructeurs

- Sur un constructeur

```
BiFunction<Integer, Integer, Point> f = Point::new;
// <=> f = (param1, param2) -> new Point(param1, param2);
```

- Sur les créations de tableaux

```
Function<Integer, String[]> f = String[]::new;
// <=> f = (int length) -> new String[length];
```

java.util.function
(interfaces prédéfinies)

Interfaces prédéfinies

Lorsque l'on écrit une méthode générique, on va prendre en paramètre une interface fonctionnelle.

- On peut écrire sa propre interface
- Mieux, on peut utiliser celles déjà existantes
  - Elles sont définies dans le package `java.util.function`

Package `java.util.function`

Interface paramétrée pour les types de fonctions usuels

- de 0 à 2 paramètres
  - `Runnable`, `Supplier`, `Consumer`, `Function`, `UnaryOperator`, etc
- version spéciale 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 on a un même type comme 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 = () -> { IO.println("hello"); }
code.run();
```

`Supplier<T>` est équivalent à `() -> T`

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

`[Int|Long|Double]Supplier`

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

Consumer

`Consumer<T>` est équivalent à `(T) -> void`

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

`[Int|Long|Double]Consumer`

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

Predicate

`Predicate<T>` représente `(T) -> boolean`

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

`[Int|Long|Double]Predicate`

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

Function

`Function<T,U>` représente `(T) -> U`

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

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

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

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

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

UnaryOperator

`UnaryOperator<T>` représente `(T) -> T`

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

`[Int|Long|Double]UnaryOperator`

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

BiPredicate et BiFunction

`BiPredicate<T, U>` représente `(T, U) -> boolean`

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

`BiFunction<T, U, V>` représente `(T, U) -> V`

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

BinaryOperator

`BinaryOperator<T>` représente `(T, T) -> T`

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

`[Int|Long|Double]BinaryOperator`

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

Les lambdas dans l'API des collections

Méthodes sur les `List`

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

```
List.of("hello", "list").forEach(IO::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 les `List` (2)

`list.removeIf(predicate)` supprime les éléments 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 deux à deux

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

Méthodes sur les Map

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

```
Map.of("a", 3, "b", 7)
  .forEach((k, v) -> IO.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 s'il n'y a pas de couple avec la clé 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 s'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);
  // {a=[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.removeIf(s -> Objects.isNull(s));`

ou

- `list.removeIf(Objects::isNull);`