Function et Lambda

Rémi Forax
forax@univ-mlv.fr

Calcul du minimum & maximum

```
public static int min(int[] values) {
  if (values.length == 0) {
    throw new IllegalArgumentException();
  }
  int min = values[0];
  for(int i = 1; i < values.length; i++) {
    min = Math.min(min, values[i);
  }
  return min;
}

public static int max(int[] values) {
  if (values.length == 0) {
    throw new IllegalArgumentException();
  }
  int max = values[0];
  for(int i = 1; i < values.length; i++) {
    max = Math.max(max, values[i]);
  }
  return max;
}
```

Comment partager le code commun ?

Il suffit d'utiliser un paramètre !

```
private static final int OP_MIN = 1;
private static final int OP_MAX = 2;

public static int minOrMax(int[] values, int typeOfOp) {
  if (values.length == 0) {
    throw new IllegalArgumentException();
  }
  int value = values[0];
  for(int i = 1; i < values.length; i++) {
   if (typeOfOp == OP_MIN)
    value = Math.min(value, values[i]);
  } else {
    value = Math.max(value, values[i]);
  }
  return min;
}
```

Ahhhh, test sur un type !!!!

Rappel approche objet

Chaque objet sait quelle est son opération

- 1 objet pour Math.min et
- 1 objet pour Math.max

et on passe l'objet en paramètre de minOrMax

Il faut donc 2 classes, et une interface qui permet de manipuler de façon commune des objets des deux classes

Solution objet

```
interface IntBinaryOperator {
  public abstract int applyAsInt(int left, int right);
}

public static int minOrMax(int[] values,
                                          IntBinaryOperator op) {
  if (values.length == 0) {
    throw new IllegalArgumentException();
  }
  int value = values[0];
  for(int i = 1; i < values.length; i++) {
   value = op.applyAsInt(value, values[i]);
  }
  return min;
}
```

Solution objet (suite)

```
private static final IntBinaryOperator OP_MIN =
  new MinBinaryOperator();
private static final IntBinaryOperator OP_MAX =
  new MaxBinaryOperator();

class MinBinaryOperator implements IntBinaryOperator {
   public int applyAsInt(int left, int right) {
      return Math.min(left, right);
   }
}

class MaxBinaryOperator implements IntBinaryOperator {
   public int applyAsInt(int left, int right) {
      return Math.min(left, right);
   }
}
```

Solution cool mais verbeuse !

Functional interface

Conceptuellement, une interface avec une seule méthode est équivalent à un pointeur de fonction en C

- On ajoute juste des noms

Une interface fonctionnelle (functional interface) est une interface qui possède une seul méthode abstraite

- Dans l'exemple, IntBinaryOperator est une fonctional interface

Method reference

En Java, une interface fonctionnelle permet de faire une conversion automatique d'une référence à une méthode vers l'interface fonctionnelle

```
IntBinaryOperator op = Math::min;
```

La syntaxe :: permet de référencé une méthode en indiquant le nom de la classe puis les nom de la méthode

- Le type des paramètres est calculée à partir de l'interface fonctionnelle

@FunctionalInterface

Annotation qui permet de demander au compilateur de vérifier que l'interface possède bien une seule méthode abstraite

- marche comme @Override
- pratique pour documenter

```
@FunctionalInterface
interface IntBinaryOperator {
  public abstract int applyAsInt(int left, int right);
}
```

Comment le compilateur trouve la méthode référencée

```
IntBinaryOperator op = Math::min;
```

À partir de la fonctional interface, le compilateur trouve la méthode abstraite

- int applyAsInt(int left, int right)

Il est déduit une signature fonctionnelle

```
int(int,int)
```

Le compilateur utilise la classe et le nom de la method reference

```
Math.sin + int(int,int) = int Math.sin(int,int)
```

Et à l'exécution

Le compilateur génére un code qui demande de créer un objet d'une classe qui implante l'interface fonctionelle et dont la méthode abstraite appel la méthode référencée

La machine virtuelle crée à l'exécution la classe correspondante et une instance de celle-ci

- pour une même conversion, la classe utilisée est toujours la même

Exemple

```
private static IntBinaryOperator getMinOp() {
  return Math::min;
}

public static void main(String[] args) {
  IntBinaryOperator min1 = getMinOp();
  System.out.println(min1.getClass());  // Lambda$$1
  IntBinaryOperator min2 = getMinOp();
  System.out.println(min2.getClass());  // Lambda$$1

  System.out.println(
          min1.getClass() == min2.getClass()); // true
}
```

Solution Objet (en moins de lignes)

```
@FunctionalInterface
interface IntBinaryOperator {
  public abstract int applyAsInt(int left, int right);
}

private static int minOrMax(int[] values, IntBinaryOperator op) {
  if (values.length == 0) {
    throw new IllegalArgumentException();
  }
  int value = values[0];
  for(int i = 1; i < values.length; i++) {
   value = op.applyAsInt(value, values[i]);
  }
  return min;
}

public static int min(int[] values) {
  return minOrMax(values, Math::min);
}

public static int min(int[] values) {
  return minOrMax(values, Math::max);
}
```

Et si on veut faire la somme ?

```
@FunctionalInterface
interface IntBinaryOperator {
  public abstract int applyAsInt(int left, int right);
}

private static int reduce(int[] values, IntBinaryOperator op) {
  if (values.length == 0) {
    throw new IllegalArgumentException();
  }
  int value = values[0];
  for(int i = 1; i < values.length; i++) {
   value = op.applyAsInt(value, values[i]);
  }
  return min;
}
private static int add(int left, int right) {
  return left + right;
}

public static int sum(int[] values) {
  return reduce(values, EnclosingClass::add);
}
```

On doit écrire une méthode statique pour pouvoir faire la conversion

Lambda

La syntaxe des lambdas est une syntaxe raccourcie qui permet d'écrire une fonction anonyme qui va être convertit en objet dont la classe implante une interface fonctionnelle

```
public static int sum(int[] values) {
  return reduce(values,
                (int left, int right) -> left + right);
}
```

Une lambda est une fonction anonymes

Inférence du type des paramètres

Un peu comme pour les méthodes références,
il est possible d'éviter de spécifier les types des paramètres

```
public static int sum(int[] values) {
  return reduce(values,
                        (left, right) -> left + right));
}
```

De la même façon, le compilateur trouve la signature fonctionnelle à partir de l'interface fonctionnelle

Syntaxe des lambdas

Ils existents deux types de lambda

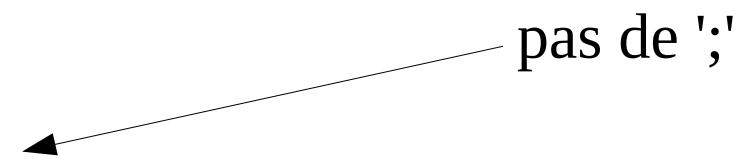
- Les lambdas expressions
  - x -> x + 1
  - list.forEach(e -> System.out.println(e));
- Les lambdas blocks
  - x -> {
    System.out.println("hello lambda");
    }
  - list.forEach(e -> {
    map.put(e.getName(), e);
    });

Syntaxe des lambdas

Et la syntaxe varie suivant le nombre de paramètre

Zéro paramètre

- () -> System;out.println("hello")
- () -> { System;out.println("hello"); }



Un paramètre

- x -> x + 1 ou (x) -> x + 1

  les parenthèses ne sont pas obligatoire

Deux paramètres et plus

- (x, y) -> x + y

Et compter le nombre de 3 ?

```
private static int reduce(int[] values, IntBinaryOperator op) {
  if (values.length == 0) {
    throw new IllegalArgumentException();
  }
  int value = values[0];
  for(int i = 1; i < values.length; i++) {
    value = op.applyAsInt(value, values[i]);
  }
  return value;
}

public static int countNumberOf(int[] values, int value) {
  return reduce(values, (v, sum) - > sum + (v == value)? 1: 0);
}

public static void main(String[] args) {
  int[] values = ...
  System.out.println(countNumberOf(values, 3));
}
```

Lambda et variable locale

Une lambda peut utiliser la valeur d'une variable locale

```
int countNumberOf(int[] values, int value) {
  return reduce(values,
        (v, sum) - > sum + (v == value)? 1: 0);
}
```

Comme les variables locales peuvent mourrir avant que le code de la lambda soit exécutée, le compilateur copie la valeur à la création de la lambda

Durée de vie des variables locales

Une variable locale peut être morte au moment où où le code de la lambda est exécutée

```
IntBinaryOperation countByMultiple(int multiple) {
  return (v, sum) - > sum + multiple;
}

public static void main(String[] args) {
  IntBinaryOperation binOp = countByMultiple(2);
  binOp.applyAsInt(2, 3);
}
```

Fin du scope

Utilisation du code la lambda

Capture de valeur de variable locale

Lors de la conversion vers l'interface fonctionnelle, si une lambda utilise des variables locales, les valeurs de ces variables sont recopiées et envoyées en paramètre à la lambda

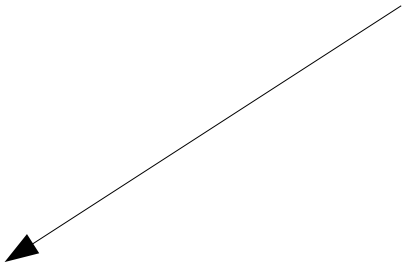
- Une lambda qui capture des valeurs de variable est différentes à chaque appel car la variable peut avoir une valeur différentes
  - La lambda ne paut pas être constante
- Le compilateur ne permet que la capture de variable assigné une seule fois

Effectivement final

Il n'est pas permis de capturer la valeur d'une variable dont la valeur change

Le compilateur vérifie que la variable est déclarée final ou effectivement final

```
IntBinaryOperation countByMultiple(int multiple) {
  multiple = 7;
  return (v, sum) - > sum + multiple;
}
```

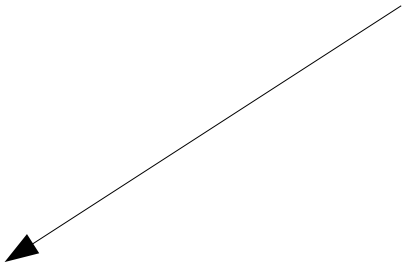


Capture impossible, la variable est pas effectivement final

Effectivement final

Pour la même raison, il est impossible d'incrémenter ou de modifier une capture d'une variable à l'intérieur de la lambda

```
IntBinaryOperation countByMultiple(int multiple) {
  multiple = 7;
  return (v, sum) - > sum + multiple++;
}
```



Capture impossible, la variable est pas effectivement final

Capture de champs

Il est aussi possible d'accéder au valeur des champs, dans ce cas c'est la valeur de this qui est capturé

donc il est possible de modifier la valeur de champs dans une lambda

```
public class Foo {
   private int multiple = 1;

   IntBinaryOperation cumulateBy() {
     return (v, sum) - > sum + multiple++;
     // equivalent à (v, sum) - > sum + this.multiple++;
   }
}
```

Method reference et capture

Les méthodes références possèdent aussi un mécanisme de capture qui permet de capturer la valeur d'une réfrence sur laquelle sera cherchée la méthode

```
ArrayList<String> list = ...
PrintStream out = System.out;
list.forEach(out::println);
```

Functional interface prédéfinie

Le package java.util.function contient des functional interfaces prédéfinies

- functionnal interfaces génériques
  void -> void j.l.Runnable.run
  void -> T j.u.function.Supplier<T>.get
  T -> void j.u.function.Consumer<T>.accept
  T -> boolean j.u.function.Predicate<T>.test
  T -> R j.u.function.Function<T, R>.apply
  T -> T j.u.function.UnaryOperator<T>.apply
  T,U -> R j.u.function.BIFunction<T,U,R>.apply
  T,T -> T j.u.function.BinaryOperator<T>.apply
- Spécialisées pour les types primitives
  int -> R j.u.function.IntFunction<R>.apply
  T -> int j.u.function.ToIntFunction<T>.applyAsInt