POLYTECH PARIS-SUD

Année 2018-2019

Polytech'Paris-Sud – 4ème année

Département Informatique

Interfaces « fonctionnelles » et « lambda expressions » en Java

Frédéric Voisin

Département Informatique

Les Interfaces « fonctionnelles »

- Les interfaces fonctionnelles : le support en Java pour l'utilisation des fonctions comme « valeur » (passage en paramètre, valeur d'une affectation, etc).
- Servent à typer des variables ou des paramètres pour des **références de méthodes** ou des définitions « au vol » de fonctions (« **lambda expressions** »)

```
Predicate<String> estVide = String::isEmpty;

Predicate<String> estGrande = (s-> s.size() > 10);
```

- Pour éviter des classes internes qui ne servent qu'à contenir une unique méthode (exemple : les « comparateurs » Java 1.4)
- Concept semblable dans d'autres langages (« `delegate` » C#)
- Mécanisme plus limité que dans les vrais langages fonctionnels

Les « lambda expressions »

- Elles correspondent à des fonctions anonymes, définies « au vol », avec une syntaxe allégée.
- Forme générale :

```
p -> p.age                                  version avec un seul paramètre
(x, y) - > x + y                            version avec plusieurs paramètres
(Figure f, Cercle c) -> { f::dessine(); c.dessine(); }
```

  le type peut être inféré par le compilateur

  Le corps de la lambda est soit une unique expression, soit un bloc d'instructions. Il existe des restrictions sur le corps de la lambda par rapport à son contexte (accès uniquement à des variables `final` ou « effectively `final` » de son contexte, pas de `break` non local, ...)
- La lambda ne définit pas une « portée » par rapport à l'environnement dans lequel elle apparaît.

Les « Interfaces fonctionnelles »

- Généralement introduites par l'annotation `@FunctionalInterface`
- Déclarent une **unique** méthode abstraite et, éventuellement, définissent des méthodes par défaut.
- Voir par exemple dans l'API Java `java.util.function` :

  `Function`, `Predicate`, `Consumer`, `Supplier`

  et des versions plus spécialisées : `IntPredicate`, `BiFunction`
- Se composent facilement
- Comme le nom l'indique, ce sont des interfaces, pas des classes !

L'interface `Predicate<T>`

- Méthode abstraite : `boolean test(T t)`
- Extensions (avec définitions par défaut) :

```
Predicate<T> negate()
Predicate<T> and(Predicate< ? super T> other)
Predicate<T> or (Predicate< ? super T> other)
```

Exemple :

```
Collection<Integer> filtre(List<Integer> col,
                           Predicate<Integer> p) {
   Collection<Integer> res = new ArrayList<>();
   for(Integer i : col) { if (p.test(i)) res.add(i); }
   return res;
}

ArrayList<Integer> c = Arrays.asList(new Integer[]{1, 26, 4});P
Predicate<Integer> p = e -> e % 2 == 0 && e > 10;
Collection<Integer> r = filtre(c, p.negate().and(e-> e > 5));
```

L'interface `Function<T, R>`

- Méthode abstraite : `R apply(T t)`
- Extensions (méthodes génériques ! Pourquoi ?) :

```
<V> Function<T, V> andThen(Function< ? super R,
                                     ? extends V> after)
<V> Function<V, R> compose(Function< ? super V,
                                     ? extends T> before)
```

`(f.compose(g))(x)` calcule `f(g(x))`

- Exemple :

```
<R, T> Collection<T> map(Function<? super R, ? extends T> f,
                         Collection <R> col) {
   Collection<T> res = new ArrayList<T>();
   for(R e : col) { res.add(f.apply(e)); }
   return res;
}
ArrayList<Integer> c = Arrays.asList(new Integer[]{1, 27, 3});
ArrayList<Integer> res = map(x → x * 2, c);
Collection<Object> m2 =  map(x -> x.toString(), c);
```

Les interfaces `Consumer<T>` et `Supplier<T>`

- La première modélise les « consommateurs de données », l'autre les « producteurs de données ».

  Pour `Consumer` : `void accept(T t)`

  Pour `Supplier` : `T get()`

- Autres exemples (voir l'interface `Stream<T>`) :

```
forEach(Consumer< ? super T> action)
```

```
<T> reduce(T neutral, BinaryOperator<T> accumulator)
```

Exemple avec cette dernière méthode :

```
int sigma(Stream<Integer> s) {
    return s.reduce(0, (a,b)-> a + b);
}
// return s.reduce(0, Integer::sum);
// return s.filter(x->x<25).reduce(0, Integer::sum);
```

Un dernier exemple

```
 Function<Integer, String> lettres = v ->  {
  String [] tab = new String[]{ "zero", "un", "deux", "trois",
                                "quatre", "cinq", "six",
                                "sept", "huit", "neuf" };
   return tab[v];
 };

List<String> a =
   Arrays.asList(new String[]{
      "Adieu", "Veaux" "Vaches", "Cochons"});

Collection<String> xx =
     map(lettres.compose(String::length), a);
```