Les lambdas en Java 8

Marianne Simonot

13 octobre 2015

.

Introduction : langages objet, langages fonctionnels

Pour avoir des SortedSet, il faut transmettre à Java une fonction de comparaison permettant d'ordonner les éléments. Paramétrer le SortedSet avec une fonction de comparaison permet de choisir la façon d'ordonner les éléments en fonction de ce que nous désirons.

Notre méthode `selectionCours` fonctionne de façon analogue. Nous voulions une méthode qui soit capable de séléctionner les cours vérifiant une propriété donnée mais quelconque. L'idée est alors de faire de cette propriété un paramètre de la méthode. On peut alors faire varier la propriété selon le genre de sélection visée.

Ce que nous avons rencontré avec les `SortedSet` et avec `selectionCours` est assez fréquent en programmation. Cela se produit dès que nous avons besoin de paramétrer une méthode par un comportement (une seule action).

Mais dans un langage objet, les paramètres des méthodes et les valeurs des variables ne peuvent être que des objets (ou des valeurs de types primitifs). Dans le jargon de la sémantique des langages, on dit que seuls les objets sont des citoyens de première classe.

Il existe une autre famille de langages, les langages fonctionnels dans lesquels les citoyens de première classe sont les fonctions. Dans ces langages, les fonctions peuvent être données comme valeur aux variables, et peuvent être des paramètres des méthodes ainsi que la valeur de retour des méthodes. Dans ces langages, programmer des `SortedSet` ou notre méthode de sélection est très aisé. Nous avons vu que cela est beaucoup plus laborieux dans un pur langage objet.

Depuis sa version 8 et conformément à la tendance actuelle des langages de programmation, Java s'ouvre au paradigme fonctionnel en introduisant les lambdas et les Streams.

Les lambdas en Java8

Avec les définitions suivantes :

```
public class Departement {
  private ArrayList<Enseignant> enseignants;
  private Set<Cours> lesCours;
...
   public Set<Cours> selectionnerCours(
                              ConditionCours condition) {
        Set<Cours> res = new HashSet<>();
        for (Cours c : lesCours) {
            if (condition.estVraieDe(c)) {
                res.add(c);
            }
        }
        return res;
    }

 }}
```

```
public interface ConditionCours {

  public boolean estVraiede(Cours c);
}
```

On peut utiliser `selectionnerCours` comme suit :

```
public class Lancement {
    public static void main(String[] args) throws Exception {
        Departement info = new Departement();
         ....
        System.out.println("les cours de L2 "+ info.selectionnerCours( c ->c.getNiveau().equals(Niveau.L2) ));
  // ou encore :
        System.out.println("les cours de L2 "+ info.selectionnerCours( (Cours c) ->c.getNiveau().equals(Niveau.L2) ));
```

`(Cours c) ->c.getNiveau().equals(Niveau.L2)` est une **expression lambda**. Elle désigne une fonction qui prend un paramètre c de type `Cours` et retourne la valeur booléenne équivalente à `c.getNiveau().equals(Niveau.L2)` Java reconnaît que cette fonction est du même type que la méthode `estVraie`. Comme dans l'interface `ProprieteCours` il n'y a qu'une seule méthode, Java se débrouille pour construire la classe correspondante implémentant l'interface.

`c ->c.getNiveau().equals(Niveau.L2)` est une expression lambda équivalente mais dans laquelle le type du paramètre a été omis. Pour la première fois en Java, nous pouvons laisser au langage le soin de deviner le type des paramètres. L'inférence de type ne réussit cependant pas toujours.

lambda expression : définition

une lambda expression est une fonction qui n'a pas de nom, mais qui a une liste de paramètres et un corps (le code qui la définit) et qui peut être passée en argument d'une méthode ou stockée dans une variable.

```
(Gens g) -> g.getAge() > 10;  fonction prenant un Gens et testant s'il a plus de 10 ans.

(Integer x) -> x + x;  fonction prenant un entier et retournant son double.

(Gens g1, Gens g2) -> g1.getAge() > g2.getAge();  fonction prenant 2 Gens et testant s'il le premier est plus agé.

x -> {   fonction prenant un entier et retournant 0 s'il et pair et 1 s'il est impair.
            if (x % 2 == 0) {
                return 0;
            } else {
```

```
            return 1;
        }
    };
```

Où utiliser une lambda expression ?

Les lambdas ne sont pas utilisables lorsqu'on définit une méthode. Les lambdas sont utilisables lors de l'appel de méthode, pour transmettre une valeur à un paramètre du type d'une **interface fonctionnelle**. Une interface fonctionnelle est une interface qui ne contient qu'une seule méthode. (Ceci n'est pas tout à fait vrai, nous y reviendrons).

ProprieteCours une interface fonctionnelle car elle ne contient qu'une seule méthode.

```
public interface ProprieteCours {

  public boolean estVraiede(Cours c);
}
```

Les lambdas ne servent à rien pour définir la méthode `selectionnerCours`. Il faut utiliser l'interface `ProprieteCours` pour typer le paramètre.

```
public class Departement {
public Set<Cours> selectionnerCours(ProprieteCours prop) {
  Set<Cours> res = new HashSet<>();
    for (Cours c : lesCours) {
            if (prop.estVraiede(c)) {
                res.add(c);
            }
    }
    return res;
  }}
```

Ici, on veut utiliser `selectionnerCours` avec la propriété "être de niveau L2". Il faut donc donner une valeur du type du paramètre c'est à dire de type `ProprieteCours`. Au lieu de créer une classe qui implémente l'interface puis de créer un objet instance de cette classe, on peut directement donner une expression lambda qui représente le code de l'unique méthode de l'interface :

```
public class Lancement {
  public static void main(String[] args) {
   Departement info = new Departement();
   info.selectionnerCours( c ->c.getNiveau().equals(Niveau.L2)
);
```

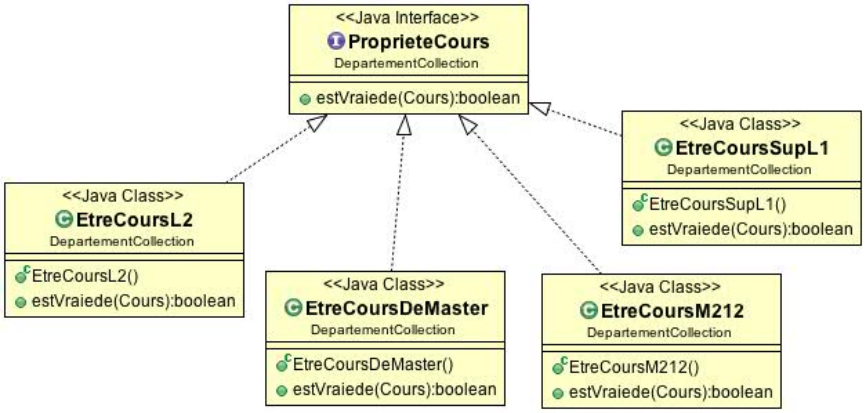
Passer des comportements avant Java 8 et après : comparaison

L'introduction des lambdas permet de réduire considérablement le code permettant de passer un comportement à une méthode.

exemple de `selectionnerCours`

avant Java8

A chaque fois qu'on veut utiliser la méthode avec une nouvelle propriété, il faut créer une classe qui implémente l'interface `ProprieteCours`, créer un objet avec le constructeur de cette classe et enfin donner cet objet comme valeur du paramètre de `selectionnerCours`. Ceci conduit à définir de très nombreuses implémentations de l'interface, chacune ne contenant que du code pour la méthode `estVraieDe`.

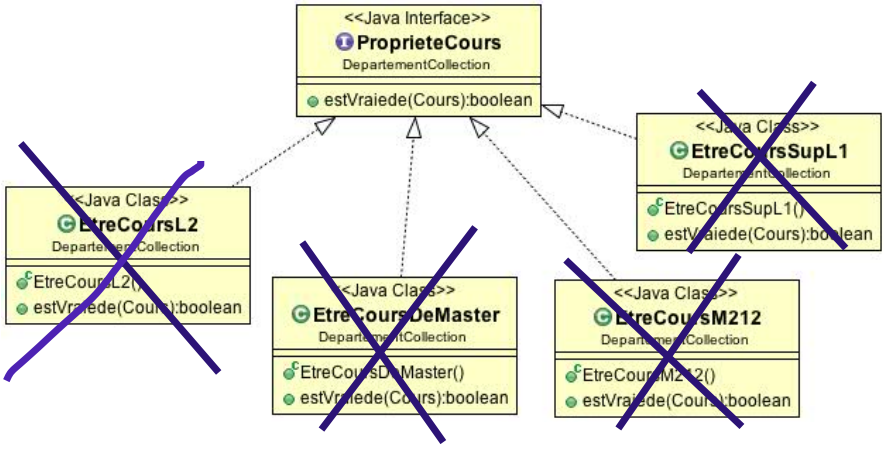


puis :

```
System.out.println("les cours de L2 "+ info.selectionnerCours(new EtreCoursL2()));
```

après Java8

Les nombreuses implémentations de l'interface deviennent inutiles. On utilise les lambdas pour créer à la volée les différentes propriétés dont nous avons besoin.




```
System.out.println("les cours de L2 "+ info.selectionnerCours((Cours c) -> c.getNiveau().equals(Niveau.L2)));
```