TP7 : Java 8 et les lambdas

Marianne Simonot

20 octobre 2015

.

exercice 1 :recoder les comparateurs avec des lambdas

- Créer un projet TP7 et y recopier votre TP6.
- Ouvrez votre classe qui implémente `Comparator<Cours>`. On veut faire disparaître cette classe tout en conservant la méthode qui la contient. Comme cette méthode compare des cours, il semble judicieux de la mettre dans `Cours`. Faites cela en lui donnant le nom que vous voulez puis détruisez la classe qui implémente `Comparator<Cours>`.

  Comme nous aurons peut être plusieurs méthodes de comparaison de cours, nous renommons la méthode `compare` de l'implémentation en `compareNiveauNom`

```
public class Cours{
 ....
    public static int compareNiveauNom(Cours o1, Cours o2) {
        if (o1.getNiveau().compareTo(o2.getNiveau()) == 0) {
            return o1. getIntitule (). compareTo(o2.getIntitule());
        } else
            return o1.getNiveau().compareTo(o2.getNiveau());
    }
}
```

- Corrigez les erreurs apparues dans `Departement` en utilisant une lambda expression à la création du `TreeSet`.

  La méthode `lesCoursParNiveauPuisNom` ne compile plus car la classe `lesCoursParNiveauPuisNom` n'existe plus. Remplacons l'appel au constructeur par une expression lambda :

```
    // les cours triés par niveau puis par nom.
    public SortedSet<Cours> lesCoursParNiveauPuisNom() {
        SortedSet<Cours> s = new TreeSet<Cours>(
                (t1, t2) -> Cours.compareNiveauNom(t1, t2));
        for (Cours c : lesCours) {
            s.add(c);
        }
        return s;
```

```
    }
```

– Faites la même chose avec vos autres classes de comparateurs.

Ajoutons dans enseignant les comparateurs d'enseignants :

```
    public static int compareHeureNom(Enseignant o1, Enseignant o2) {

        int n = o1.getHeuresEffectuees() - o2.getHeuresEffectuees();
        if (n == 0) {
            if (o1.getNom().compareTo(o2.getNom()) == 0) {
                return o1. getIdentifiant () - o2. getIdentifiant ();
            } else {
                return o1.getNom().compareTo(o2.getNom());
            }
        } else {
            return n;
        }
    }

    public static int compareNomIdentifiant(Enseignant o1, Enseignant o2) {
        if (o1.getNom().compareTo(o2.getNom()) == 0) {
            return o1. getIdentifiant () - o2. getIdentifiant ();
        } else
            return o1.getNom().compareTo(o2.getNom());
    }

}
```

On corrige ensuite les SortedSet dans Departement :

```
    // les intitulés des cours sans répétition et triés alphabétiquement.
    public SortedSet<String> lesIntitulesDeCoursTrie() {
        SortedSet<String> s = new TreeSet<String>((x, y) -> x.compareTo(y));
        for (Cours c : lesCours) {
            s.add(c. getIntitule ());
        }
        return s;
    }
    // Les noms des enseignants triés par noms
    public SortedSet<String> NomsEnseignants() {
        // SortedSet<String> lesNoms = new TreeSet<String>(new ComparateurAlphabetique());
        SortedSet<String> lesNoms = new TreeSet<String>((t1, t2) -> t1.compareTo(t2));
        for (Enseignant e : enseignants) {
            lesNoms.add(e.getNom());
        }
        return lesNoms;
    }
   public SortedSet<Enseignant> lesResponsablesL2NomPuisIdentifiant() {
        // SortedSet<Enseignant> s = new TreeSet<Enseignant>(new ComparateurEnseignantNomPuisidentifiant());
        SortedSet<Enseignant> s = new TreeSet<Enseignant>(
                (t1, t2) -> Enseignant.compareNomIdentifiant(t1, t2));
        for (Cours c : lesCours) {
            if (c.getNiveau().equals(Niveau.L2)) {
                s.add(c.getResponsable());
            }
        }
        return s;
    }
  public SortedSet<Enseignant> enseignantsSansHeuresCompTrieParNom() {
        // SortedSet<Enseignant> lesEns = new TreeSet<Enseignant>( new ComparateurEnseignantNomPuisidentifiant());
        SortedSet<Enseignant> lesEns = new TreeSet<Enseignant>(
                (t1, t2) -> Enseignant.compareNomIdentifiant(t1, t2));
```

```
        for (Enseignant e : enseignants) {
            if (e.heureComp() == 0) lesEns.add(e);
        }
        return lesEns;
    }

    // les enseignants sans heure comp triés par nbre d'heure faite puis par
    // nom.
    // Tests == vérifier que 2 enseignants de même nom , de meme heure faite
    // mais différents sont dans le set.

    public SortedSet<Enseignant> enseignantsSansHeuresCompTrieHeuresFaitesPuisNom() {
        // SortedSet<Enseignant> lesEns = new TreeSet<Enseignant>( new ComparateurEnseignantHeureFaitesPuisNom());
        SortedSet<Enseignant> lesEns = new TreeSet<Enseignant>(
                (x, y) -> Enseignant.compareHeureNom(x, y));
        for (Enseignant e : enseignants) {
            if (e.heureComp() == 0) lesEns.add(e);
        }
        return lesEns;
    }
```

exercice 2 : `Predicate<E>` , `Function<E,T>` et expressions lambdas associées

Les expressions lambdas peuvent s'utiliser partout ou l'on attend une instance d'une interface fonctionnelle. L'API Java fournit la majorité des interfaces fonctionnelles dont nous avons besoin. Nous allons manipuler 2 d'entre elles dans cet exercice.

1. Consulter la javadoc Java 8 de `Predicate` et de `Function`. Ces 2 interfaces ont des traits propres à Java 8. Prenez en connaissance en lisant l'aide mémoire qui suit.

Aide mémoire :

Les interfaces en Java 8 Jusqu'en Java 7, une interface ne peut contenir que des signatures de méthodes. En java 8, une interface peut contenir en plus des signatures qui représentent des méthodes abstraites, des méthodes statiques et des méthodes par défaut.

Nous connaissons déjà la notion de méthode statique.

La notion de méthode par défaut (et le mot clé `default` associé) est nouvelle. C'est une méthode concrète (signature + code) écrite dans une interface. Les classes qui implémentent l'interface héritent de ces méthodes et peuvent les redéfinir (comme dans le mécanisme d'héritage entre classes).

La grande nouveauté est qu'il est maintenant possible d'avoir de l'héritage multiple d'interface en Java :

```
public interface A{
 default void methode1(){
```

```
    sysout (“ AAAAAA”);
 }
}

public interface B{
 default void methode1(){
    sysout (“ BBBBBB”);
 }
}

public class C implements A,B{
// erreur à la compilation !!!!

}
```

La compilation détecte l'ambiguité suivante : dans C, quel code de méthode1 doit on prendre ?

Nous avons alors 2 solutions :

(a) Créer une implémentation de `methode1` dans C. Celle ci peut d'ailleurs faire appel à l'une des méthodes des interfaces au moyen de la syntaxe suivante : `A.super.methode1();`

(b) Décider qu'une des 2 interfaces étends l'autre. C'est alors le code de l'interface la plus spécifique qui est pris.

Interface fonctionnelle Une interface fonctionnelle est une interface **qui ne contient qu'une seule méthode abstraite**. Elle peut contenir autant de méthodes statiques et par défault qu'on le désire.

Predicate $< E >$ est une interface fonctionnelle qui représente n'importe quelle fonction booléenne. On aurait pu (du) utiliser cette interface pour `selectionnerCours`. L'unique méthode abstraite est `test` (qui joue le même rôle que notre `estVraieDe`).

Function $< E, T >$ est une interface fonctionnelle qui représente n'importe quelle fonction prenant un argument de type E et renvoyant un résultat de type T. (E et T désigne des types quelconques). Son unique méthode abstraite est `apply`

2. Téléchargez la classe `LancementLambdas` ainsi que les classes `Produit` et `Gens`. Enlevez les commentaires des instructions entre `// 1 .` et `// 2` et remplacer les `????` en suivant les consignes. Excécutez le `main` pour vérifier vos réponses.

   Faites de même pour les questions 2 à 11 du `main`.

```
public class ExerciceLambdas {

    public static void main(String[] args) {
```

```
Gens a, b, c, d;
a = new Gens("lulu", 18);
b = new Gens("toto", 17);
c = new Gens("lulu", 20);
d = new Gens("bibi", 20);

Produit p112et30 = new Produit(112, 30);
Produit p150et15 = new Produit(150, 15);
Produit p120et30 = new Produit(120, 30);
Produit p112et40 = new Produit(112, 40);
Produit p120et30Bis = new Produit(120, 30);

// completez avec des lambdas expressions et eventuellement le type de
// la variable .:
// p est une fonction qui prend un gens et teste s'il a plus de 18 ans.
Predicate<Gens> p = (Gens g) -> g.getAge() > 18;
System.out.println(p.test(a));
System.out.println(p.test(b));
System.out.println(p.test(c));

// p2 est une fonction qui prend un gens et teste s'il s'appelle "lulu".
Predicate<Gens> p2 = g -> g.getNom().equals("lulu");
System.out.println(p2.test(a));
System.out.println(p2.test(b));
System.out.println(p2.test(c));

// p3 est une fonction qui prend un gens et teste si son nom commence
// par l..
Predicate<Gens> p3 = g -> g.getNom().charAt(0) == 'l';
System.out.println(p3.test(a));
System.out.println(p3.test(b));
System.out.println(p3.test(c));

// p4 est une fonction prenant un produit et testant si son prix est sup
// a 35
Predicate<Produit> p4 = x -> x.getPrix() > 35;
System.out.println(p4.test(p112et40));
System.out.println(p4.test(p112et30));

// utiliser and or negate cf javadoc Predicate pour que p5 teste si un
// Gens s'appelle "lulu" et a plus de 18 ans.
Predicate<Gens> p5 = p.and(p2);
System.out.println(p5.test(a));
System.out.println(p5.test(b));
System.out.println(p5.test(c));
// sur qui p6 repondra t il true ?(b,c,d)
Predicate<Gens> p6 = p5.or(p3.negate());

// expliquer
// f1 est une fonction prenant une String et retournant sa longueur.
Function<String, Integer> f1 = (x -> x.length());
// L'unique methode abstraite de l'interface fonctionnelle Function<S,T>
// T apply(S x);
// appliquée à "bonjour" f1 retourne donc 6
System.out.println(f1.apply("bonjour"));

// fonction qui a x associe x+2
Function<Integer, Integer> f11 = (x -> x + 2);
System.out.println(f11.apply(2));
System.out.println(f11.apply(4));

// fonction qui prend un gens et retourne son age
Function<Gens, Integer> f2 = (x -> x.getAge());
System.out.println(f2.apply(a));
System.out.println(f2.apply(b));
```

```
        // fonction qui prend un gens et retourne la premierre lettre de son
        // nom. (enlever le type)
        Function<Gens, Character> f3 = (x -> x.getNom().charAt(0));
        System.out.println(f3.apply(a));
        System.out.println(f3.apply(b));

        // utiliser andThen pour que f4 prenne un gens et retourne le double de
        // son age.
        Function<Gens, Integer> f4 = f2.andThen(x -> x * x);

        // faire typer :
        Function<Gens, Character> f5 = ((Function<Gens, String>) x -> x
                .getNom()).andThen(y -> y.charAt(0));

    }

}
```

Exercice 3

1. Recoder `selectionnerCours` en utilisant `Predicate` à la place de `ProprieteCours`

```
public Set<Cours> selectionnerCours(Predicate<Cours> condition) {
    Set<Cours> res = new HashSet<>();
    for (Cours c : lesCours) {
        if (condition.test(c)) {
            res.add(c);
        }
    }
    return res;
}
```

2. Définir une méthode `public Set<Cours> lesCoursDe(Enseignant e)` qui retourne l'ensemble des cours dont e est responsable. Cette méthode doit impérativement appeler `selectionnerCours`.

```
public Set<Cours> lescoursDe(Enseignant e) {
    return selectionnerCours(x -> x.getResponsable().equals(e));
}
```