TP7 : Java 8 et les lambdas

Marianne Simonot

Légende des exercices
⊛ : exercice obligatoire.
⊕ : exercice permettant de répéter les notions des exercices obligatoires. Peut être sauté si l'exercice obligatoire qui précède vous semble trivial, ou peut être fait à la maison si vous avez du retard...
⊙ : exercice à faire lorsque vous avez fini le tp en avance.
.

Exercice 1 ⊛

On a dans le corrigé de Departement la méthode :

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

1. Dans un main, utilisez cette méthode et une expression lambda pour obtenir les cours de L1. Affichez le résultat.
2. Dans un main, utilisez cette méthode et une expression lambda pour obtenir les cours de "Simonot". Affichez le résultat.

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`.

Exercice 3 ⊛

Recoder `selectionnerCours` en utilisant l'interface fonctionnelle prédéfinie `Predicate` à la place de `ProprieteCours`

exercice 4 ⊛

1. Définir dans `LancementLambdas` une méthode :

```
public static ArrayList<Gens> appliquerAChacun(ArrayList<Gens> l, Function<Gens, Gens> f)
```

   la liste de gens retournée par cette méthode doit être constituée des éléments de l auxquels on a appliqué la fonction f.

2. Dans le main, ajoutez la liste suivante :

```
ArrayList<Gens> lesGens = new ArrayList<>();
lesGens.add(a);
lesGens.add(b);
lesGens.add(c);
lesGens.add(d);
```

3. Dans le main, utilisez cette méthode et une expression lambda pour obtenir une liste constituée des éléments de lesGens avec leur age augmenté de 2 ans. Affichez le résultat.
4. Dans le main, utilisez cette méthode et une expression lambda pour obtenir une liste constituée des éléments de lesGens avec leur nom passé en Majuscule. Affichez le résultat.

exercice 5 :recoder les comparateurs avec des lambdas

Pour créer un TreeSet, on donne au constructeur une instance d'une implémentation de l'interface `Comparator<E>`. Comme c'est une interface fonctionnelle, on peut remplacer cette instance par une expression lambda. C'est ce que vous allez faire maintenant.

— ⊛Supprimez la classe `ComparateurCoursNiveauNom` (mais gardez la quelques part car vous allez réutiliser le code qu'elles contient).
— ⊛Corrigez les erreurs apparues dans `Departement` en utilisant une lambda expression à la création du `TreeSet<Cours>`.
— ⊕Faites la même chose avec vos autres classes de comparateurs.

◎Pour aller plus loin : un peu de généricité

Ce paragraphe va permettre de découvrir une notion qui est en dehors du programme. VOUS NE SEREZ PAS EVALUES LA DESSUS. Ceci s'adresse donc à ceux qui ont du temps, aux curieux etc...

Nous allons découvrir comment définir des méthodes encore plus puissantes grâce à la généricité. Reprenons notre méthode de l'exercice 4 :

```
public static ArrayList<Gens> appliquerAChacun(ArrayList<Gens> l, Function<Gens, Gens> f) {
    ArrayList<Gens> res = new ArrayList<>();
    for (Gens g : l) {
        res.add(f.apply(g));
    }
    return res;
}

Puis dans le main :
    System.out.println("les gens" + lesGens);
    System.out.println("augmenter l'age des gens de 2"
            + appliquerAChacun(lesGens, gg -> new Gens(gg.getNom(), gg.getAge() + 2)));
    System.out.println("passer les noms en majuscule"
            + appliquerAChacun(lesGens, gg -> new Gens(gg.getNom().toUpperCase(), gg.getAge())));
```

Si nous voulions maintenant avoir la même méthode mais non pas sur une liste de gens mais sur une liste de produit, nous écririons a peu près le même code. La seule chose qui changerait serait le type des paramètres et de la valeur de retour de la méthode. Plus généralement encore , peu importe les types concrets que nous utilisons : si nous avons une fonction $f : A- > B$ et une liste $l1 = \{x1, x2..., xn\}$ d'éléments de type A alors le code que nous avons écrit permet d'obtenir la liste $\{f(x1), f(x2), ..., f(xn)\}$. Les éléments de la nouvelle liste sont tous de type B.

Il est possible en Java d'écrire des méthodes qui marchent pour n'importe quel type. On appelle cela des méthodes génériques. Pour le faire il faut indiquer à Java dans la signature de la méthode que l'on va utiliser des variables de type qui désignent non pas des types concrets existants mais n'importe quel type :

```
public static <E, F> ArrayList<F> appliquer(ArrayList<E> l, Function<E, F> f) {
    ArrayList<F> res = new ArrayList<>();
    for (E g : l) {
        res.add(f.apply(g));
    }
    return res;
}
```

C'est le `<E,F>` avant le type de retour de la méthode qui indique que la méthode est générique et que E et F dans le reste de la méthode désignent n'importe quels types.

1. Recopiez cette méthode dans la classe `LancementLambdas`
2. Dans le main, utilisez cette méthode et une expression lambda pour obtenir une liste constituée des éléments de lesGens avec leur age augmenté de 2 ans. Affichez le résultat.
3. Dans le main, utilisez cette méthode et une expression lambda pour obtenir une liste constituée des éléments de lesGens avec leur nom passé en Majuscule. Affichez le résultat.
4. Dans le main, utilisez cette méthode et une expression lambda pour obtenir la liste des noms des gens de lesGens. Affichez le résultat.
5. Dans le main, créez une liste de Produit `l` et utilisez cette méthode et une expression lambda pour obtenir la liste de leur prix. Affichez le résultat.
6. Dans le main, créez une liste de mots (String) et utilisez cette méthode et une expression lambda pour obtenir la liste des premiers caractères des mots. Affichez le résultat.
7. définir dans `LancementLambdas` une méthode générique qui prend une liste `l` d'éléments de type `E` quelconque et un `Predicate<E> p` et qui retourne la liste des éléments de `l` qui vérifient `p` (C'est la version génériique de notre méthode `selectionCours`).
8. utilisez cette méthode et une expression lambda pour obtenir la liste des produits de `l` plus chers que 20 euros. Affichez le résultat.