TD et TP de Compléments en Programmation Orientée Objet n° 6 : Collections, Lambdas, génériques

Une lambda qui calcule le déterminant : produit, produit, puis différence : 3 bifunctions ? idem pour le produit scalaire, det, vect

Exercice 1 : Un peu de Consumer

1. Ecrivez une classe `MaListe<E>` qui implémente `List<E>` par simple héritage de `LinkedList`
2. Définissez y la méthode `void pourChacun(Consumer<E> action)` ; son travail consistera à consommer successivement chaque élément de la liste.
3. Dans une classe `Test` construisez une liste contenant quelques entiers, et testez votre méthode `pourChacun` en lui fournissant en argument une lambda expression de sorte que le résultat affiche les éléments de la liste en retournant à la ligne à chaque fois ;
4. Vous pouvez obtenir le même résultat en transmettant la référence de la fonction `println` au lieu d'une lambda expression. Retrouvez la notation correspondante.
5. Montrez qu'en transmettant en argument à `pourChaque` une classe anonyme on peut faire en sorte que l'affichage numérote aussi les lignes.
6. Reformatez votre solution en déclarant une variable de type `Consumer` et transmettez cette variable à `pourChaque`
7. Définissez une classe `Personne` sa méthode de présentation `toString` et construisez une `MaListe` de quelques personnes.
8. On souhaite afficher cette liste de personne, avec la numérotation de ligne, en utilisant tel quel le `Consumer` de la question 5. Quelles modifications faut-il faire pour que son utilisation soit compatible.
9. Dans cette question, on va tourner un peu autour des types. On souhaite avoir un `Consumer` dont l'acceptation concerne une liste complète. C'est à dire qu'on pourra l'utiliser sous cette forme : `my_consumer.accept(maListe)`
   Ecrivez un tel `Consumer` dont le travail consiste à afficher les éléments de la liste ligne par ligne en les numérotant. Testez le. Vous devriez pouvoir l'écrire avec une lambda expression.

Exercice 2 : Autres interfaces

(cf. cours et `java.util.function`)
Ajoutez à `MaListe` les méthodes suivantes :

1. `List<E> filter(Predicate<E> pred)` : retourne une nouvelle liste consistant en les éléments de `this` qui satisfont le prédicat.
2. `<U> List<U> map(Function<E,U> f)` : retourne une liste dont les éléments sont tous les éléments de `this` auxquels on a appliqué la fonction `f`
3. `<U> U fold(U z, BiFunction<U, E, U> f)` : initialise un accumulateur a avec `z`, puis, pour chaque élément x de `this`, calcule $a =$ `f` (a, x) et finalement retourne a.
   Exemple : pour demander la somme d'une liste d'entiers : `l.fold(0, (a,x)-> a + x)`.
4. Écrivez et testez les appels permettant d'utiliser `fold` pour calculer le produit, puis le maximum d'une liste d'entiers.

Exercice 3 : Objets transformables

On donne l'interface suivante :

```
interface Transformable<T> {
    T getElement();
    void transform(UnaryOperator<T> trans);
}
```

Les instances de cette interface seront typiquement des objets avec un état (attribut) de type `T`, modifiable en passant des fonctions $T \to T$ à la méthode `transform`. Par exemple, avec un attribut de type `String`, pour lui concaténer la chaîne `"toto"` : `obj.transform(s -> s + "toto")` ; ou bien pour la passer en minuscules : `obj.transform(String::toLowerCase)`.

1. Écrivez la classe `EntierTransformable` qui implémente cette interface pour des entiers.
2. Écrivez un `main()` qui instancie un tel objet (en initialisant l'entier à 0), puis lui applique les opérations suivantes : multiplication par 2, ajout de 15, réinitialisation à 0...
3. Écrivez une classe `Additionneur` dont les objets peuvent être utilisés comme fonction $x \to x + n$ pour la classe `EntierTransformable`.

   En particulier le programme ci-dessous doit afficher 15 :

```
EntierTransformable x = new EntierTransformable(12);
x.transform(new Additionneur(3));
System.out.println(x.getElement());
```

Exercice 4 : Curryfication

La *curryfication* est l'opération consistant à transformer une fonction de type $(T_1 \times T_2 \times T_3 \cdots \times T_n) \to R$ en $T_1 \to (T_2 \to (T_3 \to (\cdots \to R) \ldots))$.

L'intérêt est de permettre une application partielle en ne donnant que le(s) premier(s) argument(s), ce qui retournera une nouvelle fonction. Par exemple si l'addition curryfiée s'écrit $add = x \to (y \to (x + y))$, alors la valeur de $add(3)$ est la fonction $y \to (3 + y)$.

1. Écrivez une méthode qui prend une fonction binaire de type $(T \times U) \to R$ et retourne sa version curryfiée, de type $T \to (U \to R))$ (comment ces types se traduisent-ils à l'aide des interfaces de `java.util.function` ?).
2. Écrivez la méthode inverse.
3. Même questions pour les fonctions ternaires. Avant de vous lancer dans cette question, remarquez qu'il n'existe pas d'interface java modélisant les fonctions ternaires et qu'il faudra donc définir une interface adaptée (exemple : `interface TriFunction<T, U, V, R> { R apply(T,U,V);}`).

Exercice 5 :

(Suite de l'exercice 3)

1. Écrivez la classe `OperateurUnaireTransformable<T> implements Transformable<UnaryOperator<T>>`, dont les objets représenteront des opérateurs de $T \to T$, modifiables par d'autres opérateurs (de $(T \to T) \to (T \to T)$)
2. Écrivez par exemple un opérateur qui prend un opérateur f de $T \to T$ et retourne l'opérateur f^{10} (f 10 fois composé avec lui-même).

3. Écrivez un `main()` qui instancie l'opérateur "multiplication par 2" en tant qu'objet de la classe `OperateurUnaireTransformable<Integer>`, puis fabrique l'opérateur "multiplication par 2^{10}" en utilisant la méthode `transform()` avec l'opérateur défini juste précédemment.