TD et TP de Compléments en Programmation Orientée Objet n° 8 : Itérateurs, streams
(Correction)

I) Streams

Astuce : pour ces exercices, il est parfois possible d'utiliser les conversions automatiques de votre IDE. Dans IntellIJ IDEA, placez votre curseur sur le mot-clé `for` *ou sur une des opérations d'un stream, faites "alt + entrée", puis regardez les choix proposés.*

Ceci permet de gagner du temps et garantit des transformations correctes, mais ne permet pas toujours de voir toutes les transformations possibles. Il reste plus que conseillé de comprendre ce que fait le code à tranformer avant de le manipuler.

Exercice 1 : Conversions de *pipelines* de *streams* en boucles `for`.

Expliquez ce que calculent les instructions suivantes puis traduisez-les en boucles `for` qui calculent la même valeur dans une variable.

1.
```
int x = appartements.stream()
    .filter(a -> a.nbPieces >= 3)
    .map(a -> a.prix)
    .reduce(Integer.MAX_VALUE, (a, b) -> (a<b)?a:b);
```

Ici, `appartements` est une liste d'`Appartement` où `Appartement` est la classe suivante :

```
public class Appartement {
   public int nbPieces;
   public int prix;
   public String lieu;
   ...
}
```

Correction :

```
int x = Integer.MAX_VALUE;
for (Appartement a : appartements)
    if (a.nbPieces >= 3 && a.prix < x) x = a.prix;
```

2.
```
List<PetitObjet> inventaire = maison.meubles.stream()
    .flatMap(m -> m.contenu.stream())
    .collect(Collectors.toList())
```

Où `maison` est de type `Maison` et l'on suppose définies les classes suivantes :

```
public class PetitObjet { }
public class Meuble { List<PetitObjet> contenu; }
public class Maison { List<Meuble> meubles; }
```

Correction :

```
List<PetitObjet> inventaire = new LinkedList<>();
for (var meuble : maison.meubles)
    for (var petitObjet : meuble.contenu)
        inventaire.add(petitObjet);
```

Exercice 2 : Conversions de boucles `for` en *pipelines* de *streams*.

Écrivez les blocs `for` suivants comme *pipelines* de *streams* (en utilisant des lambda-expressions).

1. (inspiré du tutoriel d'Oracle)

```
for (Person p : roster) {
    if (p.getGender() == Person.Gender.FEMALE) {
        System.out.println(p.getName());
    }
}
```

On suppose ici les types `Person` et `Gender` déjà définis, ainsi que leurs membres utilisés dans l'exemple.
. Astuce : utilisez l'opération intermédiaire `filter` et l'opération terminale `forEach`.

Correction :

```
roster.stream()
      .filter(p -> p.getGender() == Person.Gender.FEMALE)
      .map(Person::getName)
      .forEach(System.out::println);
```

2.

```
int effectif = 0;
for (var c : universite.composantes)
    for (var f : c.filieres)
        for (var e : f.etudiants)
            effectif++;
```

Où `universite` est de type `Universite` et sont définies les classes suivantes :

```
public class Universite { List<Composante> composantes; }
public class Composante { List<Filiere> filieres; }
public class Filiere { List<Etudiant> etudiants; }
public class Etudiant { }
```

Correction :

```
universite.composantes.stream()
    .flatMap(c -> c.filieres.stream())
    .flatMap(f -> f.etudiants.stream())
    .count();
```

Exercice 3 : Requête avec des streams (partiellement inspiré du tutoriel d'Oracle)

On vous donne sur Moodle les deux classes `Album` et `tracks` ainsi qu'un début de main où est initialisée une liste d'albums.

Essayez de répondre à ces questions en utilisant les streams.

1. Retournez une chaîne de caractères contenant la liste des titres des albums séparés par un point virgule. Utilisez entre autre `reduce`. (S'il y a un point virgule au début, ce n'est pas grave, on verra ça plus tard).

Correction :

```
1   System.out.println(albums.stream().map(a -> a.name).reduce("",(s,t)-> s+ "; "+t));
```

Comment supprimer ce point-virgule en trop ?

Correction :

```
System.out.println(albums.stream()
            .map(a -> a.name)
            .reduce("",(s,t)-> (s.equals("")?t:s+ "; "+t)));
```

2. Retournez une liste des albums ayant 4 tracks. (Utilisez `collect` à la fin).

Correction :

```
System.out.println(albums.stream()
            .filter(a -> a.tracks.size() == 4)
            .map(a -> a.name).collect(Collectors.toList()));
```

3. On veut maintenant sous forme de chaîne de caractères, la liste des albums avec pour chaque la liste de leur tracks. (Il faut utiliser deux niveaux de streams).
   L'affichage de la chaîne devra donner ceci :

```
Latin Music 1:_lorem_ipsum_sit
Latin Music 2:_dolor_amet_adispising
More Latin Music:_elit_lorem_tempor_aliqua
Latin Music For Ever:_elit_enim_amet_aliqua
Common Latin Music:_minim_veniam_laboris
```

Correction :

```
   System.out.println(albums.stream()
            .map(a -> a.name
                 + a.tracks.stream().map(t -> t.name).reduce(":",(s,t)-> s+ "_"+t)
            )
            .reduce("",(s,t)-> s+ "\n"+t));
```

4. Convertissez le code suivant en une instruction qui utilise les lambda-expressions et les opérations d'agrégation au lieu de boucles `for` imbriquées. Astuce : construisez un *pipeline* invoquant les opérations `filter`, `anyMatch`, `sorted` et `collect` dans cet ordre.

```
    List<Album> favs = new ArrayList<>();
    for (Album a : albums) {
        boolean hasFavorite = false;
        for (Track t : a.tracks) {
            if (t.rating >= 4) {
                hasFavorite = true;
                break;
            }
        }
        if (hasFavorite)
            favs.add(a);
    }
    Collections.sort(favs, Comparator.comparing(a -> a.name));
```

Correction :

```
    List<Album> favs = albums.stream()
            .filter(a -> a.tracks.stream()
            .anyMatch(t -> t.rating >= 4))
            .sorted(Comparator.comparing(a -> a.name))
            .collect(Collectors.toList());
```

II) Itérateurs et itérables

Exercice 4 : Fibonacci

Un itérable ou une collection ne contient pas forcément une représentation explicite de ses éléments. Il est aussi possible d'en implémenter une où les éléments sont calculés par une formule.

Par exemple, on peut définir une suite qui contient tous les termes de la suite de Fibonacci, ou bien la collection des n premiers termes de la suite de Fibonacci (n passé au constructeur).

Rappels :

- Suite de fibonacci :
  - $u_0 = 1$;
  - $u_1 = 1$;
  - pour $i > 1$, $u_i = u_{i-1} + u_{i-2}$.
- Les interfaces de l'API : (certaines méthodes ayant un comportement par défaut ne sont pas indiquées)

```java
public interface Iterator<E> {
  boolean hasNext();
  E next() throws noSuchElementException;
  void remove(); //methode default dans l'interface
}

public interface Iterable<E> {
   Iterator<E> iterator();
}
```

À faire :

1. Progammez `class IterateurFibo implements Iterator<Integer>` dont la méthode `hasNext()` retourne `true` les n premières fois et `false` après et la méthode `next()` qui retourne tour à tour les termes de la suite de fibonacci.

   La méthode `remove()` par défaut se résume à la seule instruction `throw new UnsupportedOperationException();` ce qui nous convient, on n'y touchera donc pas.

Correction :

```java
// Remarque: il manque le throw NosuchElementException dans next()!
public class IterateurFibo implements Iterator<Integer> {
    private int courant = 0, suivant = 1;

    public IterateurFibo(int decompte) {
        this.decompte = decompte;
    }

    private int decompte;

    @Override
    public boolean hasNext() {
        return decompte > 0;
    }

    @Override
    public Integer next() {
        decompte--;
        int nouveau = courant + suivant;
        courant = suivant;
        suivant = nouveau;
        return courant;
    }
}
```

2. Programmez `class Fibo implements Iterable<Integer>`, paramétrée par n, telle que la méthode `iterator()` retourne une instance d'`IterateurFibo` paramétrée pour le même n.
3. Testez en vérifiant que

```
1 Fibo fib = new Fibo(20);
2 for (int n: fib)
3    System.out.println(n);
4 for (int n: fib)
5    System.out.println(n);
```

affiche bien les 20 premiers termes de la suite de Fibonnacci deux fois de suite.

Correction :

```
1  // Remarque: il manque le throw NosuchElementException dans next()!
2
3  class Fibo implements Iterable<Integer> {
4     private final int taille;
5
6     public Fibo(int taille) {
7        this.taille = taille;
8     }
9
10    @Override
11    public Iterator<Integer> iterator() {
12       return new IterateurFibo(taille);
13    }
14
15    public static void main(String[] args) {
16           Fibo fib = new Fibo(20);
17               for (int n: fib)
18                  System.out.println(n);
19               for (int n: fib)
20                  System.out.println(n);
21    }
22 }
```

4. En fait, rendre visible la classe `IterateurFibo` n'a pas d'intérêt, déclarez-la en classe privée non statique interne à la classe lstinlineFibo. Utilisez le fait qu'une classe interne non statique a accès aux attributs de la classe englobante. Il devrait donc y avoir des petites modifications à faire.

   Testez avec le programme précédent.

   Correction : Modifs : le constructeur d'`IterateurFibo` n'a pas besoin d'argument puisqu'il récupère la valeur de l'attribut `taille` de la classe englobante pour initialiser `decompte`.

5. On veut maintenant faire la même chose avec une classe anonyme au lieu de la classe interne. Pour varier un peu, on va l'utiliser pour une suite géométrique, c'est-à-dire la suite $1, r, r^2, r^3, \ldots$ (où r est un paramètre appelé la "raison").

   Pour économiser du temps de calcul, on calculera un élément en multipliant l'élément précédent par la raison.

   Pour faire la classe anonyme, rappelez-vous qu'elle peut avoir des attributs (initialisés en même temps que leur déclarations puisqu'elle n'a pas de constructeur).

Correction :

```java
// Remarque: Là  aussi il manque le throw NosuchElementException dans next()!

public class Geom implements Iterable<Integer>{

    private int limite;
    private int raison;

    public Geom(int raison,int limite){
        this.limite = limite;
        this.raison = raison;
    }

    @Override
    public Iterator<Integer> iterator(){
        return new Iterator<Integer>(){
            int courant = 1;
            int decompte = limite;

            @Override
            public boolean hasNext() {
                return decompte > 0;
            }

            @Override
            public Integer next() {
                decompte--;
                courant = courant * raison;
                return courant;
            }
        };
    }
}
```

6. Notez que toute instance de `Fibo` représente un itérable infini. Est-ce que le calcul suivant va se terminer (assez vite) ou bien continuer indéfiniment (ou bien jni eusqu'à dépasser la mémoire allouée par la JVM) :

```java
new Fibo().stream().filter(x -> x > 100).findFirst().ifPresent(System.out::println);
```

En cas de terminaison, que fait cette instruction ?

Correction : `filter` calcule le stream consistant en les éléments supérieurs à 100 du stream d'origin. Ce stream est infini mais n'est pas calculé explicitement, `filter` retourne donc immédiatement. Puis, `findFirst` retourne le premier élément trouvé dans ce stream infini. Cette recherche termine toujours (soit le stream est vide et l'optionnel vide est retourné, soit ce n'est pas le cas, et seul le premier élément est calculé, puis mis dans un optionnel qui sera retourné).
L'instruction retourne donc le premier terme de la suite de Fibonacci strictement supérieur à 100.

7. Même question pour :

```java
var listFibo = new Fibo().stream().collect(Collectors.toList());
```

Correction : Il s'agit ici de mettre tous les éléments du stream (infini) dans une liste, qui est une collection contenant explicitement tous ses éléments, qui doivent tous être calculés pour fabriquer la liste. Comme la suite de Fibonacci est infinie,

ce calcul nécessite un temps et une mémoire infinis. Donc cette liste ne sera jamais retournée.

8. Écrivez un *pipeline* de *stream* prenant comme source une nouvelle instance de `Fibo` et calculant la somme des 10 premiers termes de la suite de `Fibonacci`.

Correction : En convertissant vers `IntStream` via `mapToInt` pour pouvoir utiliser `sum` :

```
1       int somme = new Fibo().stream().limit(10).mapToInt(x -> x).sum();
```

Sinon, avec `reduce` :

```
1       int somme = new Fibo().stream().limit(10).reduce(0, (s,x) -> s + x);
```

III) S'il vous reste du temps

Terminez en priorité le TP7 jusque l'exercice 5.

Exercice 5 : 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. Mêmes 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, par exemple :

```
1  @FunctionalInterface
2  interface TriFunction<T, U, V, R> {
3     R apply(T,U,V);
4  }
```

Correction :

```
1     <T, U, R> Function<T, Function<U, R>> curryfy2(BiFunction<T, U, R> bf) {
2        return t -> u -> bf.apply(t, u);
3     }
4
5     <T, U, R> BiFunction<T, U, R> decurryfy2(Function<T, Function<U, R>> bf) {
6        return (t, u) -> bf.apply(t).apply(u);
7     }
8
9     <T, U, V, R> Function<T, Function<U, Function<V, R>>> curryfy3(TriFunction<T, U, V, R> tf) {
10       return t -> u -> v -> tf.apply(t, u, v);
11    }
12
```

```
13      <T, U, V, R> TriFunction<T, U, V, R> decurryfy3(Function<T, Function<U, Function<V, R>>> tf) {
14          return (t, u, v) -> tf.apply(t).apply(u).apply(v);
15      }
```