TD et TP n° 7 : Collections, Lambdas, génériques
(Correction)

Rappel : Si vous avez besoin de savoir précisément quel est le package d'une classe ou interface, quelles méthodes y sont définies, ce qu'une méthode renvoie dans des cas spéciaux ou quelles exceptions elle lance. Vous le trouverez dans la page `https://docs.oracle.com/en/java/javase/11/docs/api/index.html`.

Attention : certains exercices ou questions, marqués d'une astérisque (*) nécessitent d'avoir lu auparavant le chapitre du cours sur les lambda-expressions. Par ailleurs, ils sont un peu plus difficiles que les autres exercices de ce TP, et peuvent donc être traités en dernier.
Remarque : le chapitre sur les lambda-expressions ne sera pas présenté en cours magistral et sera supposé connu pour la suite des TP et du CM.

I) Comprendre les collections

Exercice 1 : Modéliser

On rappelle quelques interfaces de collections importantes :
- `List<E>` : liste d'éléments avec un ordre donné, accessibles par leur indice.
- `Set<E>` : ensemble d'éléments sans doublon.
- `Map<K,E>` : ensemble d'associations (clé dans `K`, valeur dans `E`), tel qu'il n'existe qu'une seule association faisant intervenir une même clé.

Ces interfaces peuvent être composées les unes avec les autres pour définir des types plus complexes. Exemple : un ensemble de séquences d'entiers se note `Set<List<Integer>>`.
Dans chacune des situations suivantes, donnez un type qui convient pour la modéliser :

1. Donnée des membres de l'équipe de France de Football.

   Correction : `Set<Joueur>` (pour des raisons d'efficacité algorithmique, on peut aussi choisir `List<Joueur>` : on fait l'économie du test de doublon)

2. Idem, avec en plus leurs rôles respectifs dans l'équipe.

   Correction : `Map<Joueur, Role>` (On peut inverser si chaque rôle n'existe qu'en un seul exemplaire dans l'équipe. Dans ce cas, on a une bijection entre les joueurs et leurs rôles.)

3. Marqueurs de buts lors du dernier match (en se rappelant la séquence).

   Correction : `List<Joueur>`, voire, si on veut aussi l'information sur le temps où le but a été marqué : `SortedMap<Temps, Joueur>`.

4. Affectation des étudiants à un groupe de TD.

   Correction : `Map<Etudiant, GroupeTD>`

5. Pour chaque groupe de TD, la « liste » des enseignants.

Correction : `Map<GroupeTD, Set<Enseignant>>`

6. Étudiants présents lors de chaque TP de Java ce semestre.

   Correction : `Map<Seance, Set<Etudiant>>` ou `List<Map<GroupeTD, Set<Etudiant>>>` ou `Map<Semaine, Map<GroupeTD, Set<Etudiant>>>` (seulement si un même groupe n'a qu'un TP par semaine).

II) Collections maison

Correction : Pour la correction des exercices de cette partie, voir les fichiers java joints au corrigé.

Les collections de Java sont bien pratiques mais ont quelques défauts, notamment :

1. Pas de distinction entre collections mutables et immuables : toutes les types sont munis d'opérations censés modifier le contenu (notamment ajouter ou supprimer des éléments). La spécification dit que ces opérations peuvent ne pas être implémentées et quitter sur l'exception `NotImplementedException`, mais ce serait mieux d'avoir une certitude dès la compilation (l'absence de telles méthodes dans l'interface fournirait une telle certitude)
2. Pas de traitements par lot (comme `map`, `fold`, `filter` ...) en dehors de `forEach` (implémentée par défaut grâce aux itérateurs).

Pour résoudre le problème 1, notre *framework* « UP Collections » définit toute une série d'interfaces, réparties dans des paquetages différents en fonction de la mutabilité :

```java
package up.collections;

public interface Collection<E> {
   int size();
   boolean contains(E elem);
   Iterator<E> iterator();
}
```

```java
package up.collections;

public interface List<E> extends Collection<E> {
   E get(int index);
}
```

```java
package up.collections;

public interface Iterator<E> {
   E next();
   boolean hasNext();
}
```

```java
package up.collections.immutable;

/**
 * Toute classe implémentant cette interface s'engage à respecter
 * le fait que le contenu de la collection soit non modifiable.
 */
public interface Collection<E> extends up.collections.Collection<E> {
}
```

```java
package up.collections.immutable;

public interface List<E> extends Collection<E>, up.collections.List<E> {

}
```

On suppose qu'il y a aussi des interfaces `Set`, `Queue` et ainsi de suite ; et que toutes ces interfaces ont aussi leur analogue dans le *package* `up.collections.mutable`. Dans ce dernier *package*, les interfaces ont, évidemment, quelques méthodes en plus, comme `clear`, `set`, `add`, `remove`, `pull` ou `push` pour permettre la mutation.
Cela dit, ce TP se concentre sur les implémentations immuables.
Un dernier avertissement avant de commencer cette série d'exercices : les classes et interfaces de *UP Collections* portent souvent les mêmes noms que celles du JDK (`java.util`). N'hésitez pas à donner leurs noms complets pour lever les ambiguïtés (par exemple `java.util.List` ou `up.collections.immutable.List` au lieu de juste `List`). Méfiez vous aussi des imports automatiques.

Exercice 2 : Diagramme de classes

Pour sûr d'y voir clair avant de commencer à programmer, dessinez un diagramme de classes pour les interfaces décrites ci-dessus.

Exercice 3 : Implémentation basique et fabrique statique

1. Implémentez `up.collections.immutable.List` par une classe `up.collections.immutable.ArrayList` contenant un tableau d'éléments.
   Son constructeur a pour signature `(E... elements)` *(Se documenter au préalable sur les méthodes et constructeurs à nombre variable d'arguments.)*
   Remarque : il faudra, à un moment, implémenter `up.collections.Iterator`.
2. Ajoutez à l'interface `up.collections.immutable.List` une fabrique statique `static List<E> of(E... );` de telle sorte que l'appel `List.of(1, 2, 3)` retourne la liste $[1, 2, 3]$.

Exercice 4 : Implémentations gratuites !

Un inconvénient de *UP collections* est que ce *framework* est tout neuf et ne contient qu'une seule implémentation. Une façon d'en obtenir facilement d'autres est d'*adapter* des implémentations existantes de collections, notamment celles du JDK (*package* `java.util`).

1. Écrivez un adaptateur de `java.util.List` vers `up.collections.immutable.List`.
   Attention, cela nécessite aussi d'écrire une classe d'adaptation de `java.util.Iterator` vers `up.collections.Iterator`.
   Notez qu'il sera toujours possible de modifier (de façon détournée) votre liste dite « immuable » si jamais un alias externe vers la liste adaptée a été conservé.
   Notez aussi qu'on ne peut pas faire de copie défensive ici, la classe de la liste à adapter n'étant pas connue. Tout au plus, on peut recopier cette liste dans une autre liste dont l'implementation aura été choisie avant, mais ce ne serait plus le patron adaptateur, dans ce cas !
2. Ajoutez une méthode `static List<E> fromJDKList(java.util.List<E> list)`, dans l'interface `up.collections.immutable.List`, qui instancie l'adaptateur que vous venez d'écrire à partir de la liste en argument et retourne cette nouvelle instance.

Exercice 5 : Itérer facilement

1. Un autre inconvénient de *UP collections* est que le langage Java n'a pas de support dédié (boucles *for-each* : `for (var E: collection){ ... }` pour itérer nos collections.
   Que cela ne tienne, il suffirait pour cela de les « adapter » vers `java.util.Iterable` de telle sorte à pouvoir écrire :
   `for (var E: collection.asJavaCollection()){ ... }`.
   Écrivez une telle méthode `asJDKIterable` dans l'interface `up.collections.Collection`.
   (Se poser d'abord les bonnes questions : Quel est le type de retour de `asJDKIterable` ? Une instance de quelle classe retourne-t-elle ? Où, comment, et combien de fois applique-t-on le patron adaptateur ?)
2. (*) Ajoutez à `up.collections.Collection` la méthode `void forEach(Consumer<E> action)` de telle sorte que l'instruction

```
1  uneCollection.forEach(x -> { System.out.println(x); });
```

   ait le même effet que l'instruction

```
1  for(var x: maCollection) { System.out.println(x); }
```

   (De façon générale, `forEach` appelle `action.consume(x)` pour tout élément `x` de la collection `this`.)

Exercice 6 : Opérations d'agrégation (*)

Ci-dessous, « `???` » est à remplacer à chaque fois par la bonne interface fonctionnelle (cf. cours, pages intitulées « Catalogue des interfaces fonctionnelles de Java »).
Ajoutez à `up.collections.immutable.List` les méthodes par défaut suivantes :

1. `List<E> filter(??? predicate)` : retourne une nouvelle liste consistant en les éléments de `this` qui satisfont le prédicat `predicate`.
2. `<U> List<U> map(??? f)` : retourne une liste dont les éléments sont tous les éléments de `this` auxquels on a appliqué la fonction `f` (Par exemple, pour obtenir depuis une liste de chaînes de caractères, la liste des longueurs de ses éléments : `maListeDeString.map(s -> s.length())`) ou de manière équivalente `maListeDeString.map(String::length)`)
3. `Optional<E> find(??? predicate)` [1] : retourne un optionnel contenant un élément de la liste satisfaisant le prédicat `predicate`, s'il en existe, sinon l'optionnel vide.
4. `<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)`.
   Écrivez et testez les appels permettant d'utiliser `fold` pour calculer le produit, puis le maximum d'une liste d'entiers.
5. Parmi les méthodes ci-dessus, lesquelles peuvent être définies à l'aide des autres et comment ?

Exemples d'utilisation :

1. ou `Optionnel<E> trouve(??? cond)`, en utilisant la classe de l'exercice 7

```java
public static void main(String[] args) {
   var l1 = List.of(1, 2, 3, 4);
   System.out.println(l1); // [1, 2, 3, 4]
   var l2 = l1.map(x -> x * 2);
   System.out.println(l2); // [2, 4, 6, 8]
   Function<Integer,List<Integer>> f = x -> List.of(x,x);
   var l3 = l2.filter(x -> x != 4);
   System.out.println(l4); // [2, 6, 8]
   var sum = l3.fold(0, (x, y) -> x + y);
   System.out.println(sum); // 16
   System.out.println(l3.find(x -> x == 4)); // Optional.empty
   System.out.println(l3.find(x -> x == 6)); // Optional[6]
}
```

Syntaxe : Si vous voulez pouvoir mettre plusieurs instructions dans une lambda expression, mettez-les entre accolades. Par exemple, `point-> {point.x = 2; return point;}` pour une fonction de `Point2D` vers `Point2D`.

III) Optionnels

Exercice 7 : Une classe générique simple, les « optionnels »

Quand une fonction peut renvoyer soit quelque chose de type `T` soit rien, permettre de retourner **`null`** pour « rien » peut provoquer des erreurs. On voudrait plutôt retourner un type ayant une instance réservée pour la valeur « rien » (les autres instances encapsulant une « vraie » valeur). C'est ce qu'on propose avec la classe générique `Optionnel<T>`[2], à programmer dans l'exercice.

1. Programmez une telle classe. Cette classe aura un unique attribut de type `T`, sa nullité sera considéré comme une valeur « vide ». Mettez-y un constructeur et les méthodes suivantes :
   - **`boolean`** `estVide()` : retourne **`true`** si l'objet ne contient pas d'élément, **`false`** sinon.
   - `T get()` : retourne l'élément, lance `NoSuchElementException` (package `java.util`) si l'optionnel est vide.
   - `T ouSinon(T sinon)` : retourne l'élément s'il existe, `sinon` sinon.
2. Toilettage : Ajoutez des fabriques statiques `Optionnel<T> de(T elt)` (pour `elt` non **`null`**, sinon on lance `IllegalArgumenException`) et `Optionnel<T> vide()` qui retourne un objet « vide », puis rendez le(s) constructeur(s) privé(s).

Correction :

```java
public class Optionnel<T>{
   private final T val;

   private Optionnel(T val){
      this.val = val;
   }

   public static<T> Optionnel<T> de(T elt){
      if (elt == null)
         throw new IllegalArgumentException();
      return new Optionnel<T>(elt);
   }

   public static<T> Optionnel<T> vide(){
      return new Optionnel<T>(null);
```

2. L'API de java propose justement `Optional<T>` à cet effet.

```
16      }
17
18      public boolean estVide(){
19          return val == null;
20      }
21
22      public T get(){
23          if (val == null)
24              throw new NoSuchElementException();
25          return val;
26      }
27
28      T ouSinon (T sinon){
29          return val != null ? val : sinon;
30      }
31  }
```

3. Amélioration plus difficile : Afin d'optimiser, faites en sorte que `Optionnel<T> vide()` retourne toujours la même instance `VIDE` : Il faudra créer `VIDE` sans paramètre générique : `private static Optionnel VIDE = new Optionnel<>(null);` et faire un cast approprié dans le code de `vide()`. Vous pourrez ensuite supprimer les warnings de `javac` en mettant `@SuppressWarnings("unchecked")` avant la méthode et `@SuppressWarnings("rawtypes")` devant la déclaration de `VIDE`.

Correction :

```
1  //Ce qui change uniquement
2     @SuppressWarnings("rawtypes")
3     private static final Optionnel VIDE = new Optionnel<>(null);
4
5    @SuppressWarnings("unchecked")
6     public static<T> Optionnel<T> vide(){
7        return (Optionnel<T>)VIDE;
8     }
```

4. Application : écrivez et testez une méthode qui cherche le premier entier pair d'une liste d'entiers et retourne un optionnel le contenant, si elle le trouve, ou l'optionnel vide sinon.

Correction :

```
1  public static Optionnel<Integer> cherchePremierPair(List<Integer> liste){
2     for(Integer i : liste){
3        if(i != null && i %2 == 0)
4              return Optionnel.de(i);
5     }
6     return Optionnel.vide();
7  }
```

Exercice 8 : Optionnels fonctionnels*

On souhaite maintenant compléter l'API de `Optionnel` de l'exercice 7 en fournissant des méthodes permettant d'exécuter du code conditionnellement en fonction de l'état (vide ou non vide) de l'optionnel.
Concrètement, on demande les méthodes suivantes (« `???` » = trouvez la bonne interface fonctionnelle.) :

- `Optionnel<T> filtre(Predicate<T> cond)` : retourne l'optionnel lui-même s'il contient une valeur et qu'elle satisfait le prédicat `cond` ; retourne `Optionnel.vide()` sinon.
- `void siPresent(??? f)` : si l'optionnel contient une valeur `v`, alors exécute `f` avec le paramètre `v`. Si l'optionnel est vide, ne fait rien.
- `<U> Optionnel<U> map(??? f)` : si l'optionnel contient une valeur `v`, alors retourne un optionnel contenant le résultat de `f` appliqué à `v`. Sinon retourne l'optionnel vide.

Correction : Ajouter les méthodes suivantes à la classe `Optionnnel<T>` :

```
1   public Optionnel<T> filtre(Predicate<T> cond){
2      if(val != null && cond.test(val))
3         return this;
4      else
5         return Optionnel.<T>vide();
6   }
7
8   public void siPresent(Consumer<T> f){
9      if(val != null)
10        f.accept(val);
11  }
12
13  public <U> Optionnel<U> map (Function<T,U> f){
14     if(val != null){
15        U u=f.apply(val);
16        if(u != null)
17           return Optionnel.<U>de(u);
18     }
19     return Optionnel.<U>vide();
20  }
```