TD et TP de Compléments en Programmation Orientée Objet n° 7 : Collections, Lambdas, génériques

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` [1].

Attention ! Les deux exercices sur les Optionnels sont à rendre (Exos 3 et 5)

I) Collections et Génériques

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.
2. Idem, avec en plus leurs rôles respectifs dans l'équipe.
3. Marqueurs de buts lors du dernier match (en se rappelant la séquence).
4. Affectation des étudiants à un groupe de TD.
5. Pour chaque groupe de TD, la « liste » des enseignants.
6. Étudiants présents lors de chaque TP de Java ce semestre.

II) Génériques et Lambdas expressions

Exercice 2 : Objets transformables

On donne l'interface suivante :

```
1  interface Transformable<T> {
2     T getElement();
3     void transform(UnaryOperator<T> trans);
4  }
```

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. Vous pouvez remplacer le « 11 » par « 8 » ou « 13 », par exemple, pour d'autres versions de Java.

1. Écrivez la classe `EntierTransformable` qui implémente cette interface pour des entiers. Nous vous rappelons que la méthode fonctionnelle de `UnaryOperator<T>` est de prototype `T apply(T)`.
2. Écrivez un `main()` qui instancie un tel objet (en initialisant l'entier à 2, par exemple), 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));
x.transform(new Additionneur(5));
System.out.println(x.getElement());
```

4. Comment transformer votre interface et votre classe `EntierTransformable` pour qu'au lieu d'écrire le code donné dans l'exercice ci-dessus, on puisse écrire le code ci-dessous ?

```
   EntierTransformable x = new EntierTransformable(12);
   System.out.println(x.transform(new Additionneur(3))
             .transform(new Additionneur(5))
             .getElement());
```

Indication : Rappelez-vous de ce que vous avez fait dans le mini-projet Builder.

Exercice 3 : 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).
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`.
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.

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

Exercice 4 : Opérations d'agrégation sur une liste

Écrivez une classe `MyArrayList<E>` qui étend la classe `ArrayList<E>` en y ajoutant les méthodes d'instance suivantes : « `???` » = trouvez la bonne interface fonctionnelle (cf. cours, pages intitulées « Catalogue des interfaces fonctionnelles de Java 8 ») :

1. Constructeurs et en-tête de la classe : Pour éviter un warning à propos de la sérialisation, on mettra avant `public class...` `@SuppressWarnings("serial")`.

   Par ailleurs, il sera pratique d »implémenter un constructeur vide et le constructeur `public MyArrayList(Collection<? extends E> c)`.

   Celui-ci permettra d'initialiser facilement (mais pas forcément efficacement !) un tableau de la façon suivante : (à n'utiliser que pour des tests rapides !)

```
List<String> l =
   Arrays.asList(new String[]{"Lorem", "ipsum", "dolor","sit", "amet",
                              "consectetur", "adipiscing", "elit",
                              "Proin", "et", "suscipit", "elit"});
MyArrayList<String> liste = new MyArrayList<String>(l);
```

2. `MyArrayList<E> filtre(??? cond)` : retourne une nouvelle liste consistant en les éléments de `this` qui satisfont le prédicat `cond`.
3. `void filtreInterne(??? cond)` : supprime de `this` les éléments de `this` qui **ne satisfont pas** le prédicat `cond`. Utilisez un itérateur !
4. `<U> MyArrayList<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)`)
5. `Optional<E> trouve(??? cond)` [3] : retourne un optionnel contenant un élément de la liste satisfaisant le prédicat `cond`, s'il en existe, sinon l'optionnel vide.
6. `<U> U reduit(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.reduit(0, (a,x)-> a + x)`.

   Écrivez et testez les appels permettant d'utiliser `reduit` pour calculer le produit, puis le maximum d'une liste d'entiers.
7. Parmi les méthodes ci-dessus, lesquelles peuvent être définies à l'aide des autres et comment ?

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`.

Exercice 5 : Optionnels fonctionnels

On souhaite maintenant compléter l'API de `Optionnel` de l'exercice 3 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.) :

3. ou `Optionnel<E> trouve(??? cond)`, en utilisant la classe de l'exercice 3

— `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.

S'il vous reste du temps...

Exercice 6 : Listes abstraites

Construisez votre propre collection (générique) basée sur les tableaux Java. L'idée générale est d'implémenter `List<E>`, en mettant les données dans les n premières cases du tableau s'il y a n données. Cette propriété devra être maintenue même lorsqu'on ajoute ou enlève un élément du tableau :

— Vous implémenterez l'interface `List<E>`.
— Pour cela, il est conseillé d'étendre la classe abstraite `AbstractList<E>` (pour simplifier le travail : il ne reste que `get(int)`, `size()`, `set(int, E)`, `add(int, E)` et `remove(int)` à implémenter).
— Vous programmerez d'abord deux constructeurs : un sans argument qui créera un tableau d'une taille par défaut.
Comme l'attribut tableau ne peut pas être instancié avec `new E[n]`, vous devrez écrire quelque chose du genre `(E[])new Object[n]`.
Si vous utilisez un IDE, remarquez qu'il signale un problème à cet endroit là (regardez ce qu'il dit), sinon compilez avec `javac` pour regarder l'avertissement.
Quels soucis peut théoriquement provoquer cette solution ? En quoi ceux-ci ne pourront pas en réalité se produire ici ?
Si vous pouvez prouver qu'il n'y a aucun risque, ajoutez un commentaire expliquant cela et insérez `@SuppressWarnings("unchecked")` sur la ligne précédant le constructeur pour que l'IDE ou le compilateur ne signale plus de problème ici à l'avenir.
— Prévoyez le redimensionnement du tableau quand on dépasse sa capacité actuelle lors de l'ajout d'un élément. Quelques méthodes intermédiaires privées vous éviteront de la recopie de code.

Testez votre implémentation, notamment :

1. Ajoutez un constructeur prenant un nombre variable d'arguments [4].
Syntaxe : `public MaClasse(E... elts)` dans le corps du constructeur `elts` est un tableau de E.
Pour créer un tableau de même type qu'un autre, vous pouvez utiliser `Arrays.copyOf(T[] original` int newLength), cf javadoc.
2. Testez votre classe dans un `main()` comportant une boucle ajoutant et enlevant des éléments au hasard en fin de liste (en vérifiant, le cas échéant, que la liste n'est pas vide).
3. Testez aussi les méthodes des interfaces `List<E>` et `ListIterator<E>` dont vous n'avez pas eu besoin à la question précédente.

4. Si ce constructeur provoque des *warnings* à la compilation, regardez si les annotations `@SafeVarargs` et/ou `@SuppressWarnings("varargs")` peuvent vous aider à vous en débarrasser. Encore une fois, si vous ajoutez ces annotations, vous **devez** aussi ajouter un commentaire justifiant que votre programme est malgré tout correct, c.-à-d. que les *warnings* supprimés étaient sans objet.