

TD et TP de Compléments en Programmation Orientée

Objet 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>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.

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.

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

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) 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 $T \rightarrow T$ 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. Nous vous rappelons que la méthode fonctionnelle de `UnaryOperator<T>` est de prototype `T apply(T)`.

Correction :

```
1 public class EntierTransformable implements Transformable<Integer> {  
2     private int val; //On peut aussi mettre Integer  
3  
4     public EntierTransformable(int val) {  
5         this.val = val;  
6     }  
7  
8     @Override  
9     public Integer getElement() {  
10        return val;  
11    }  
12  
13    @Override  
14    public void transform(UnaryOperator<Integer> trans) {  
15        val = trans.apply(val);  
16    }  
17 }
```

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...

Correction :

```
1 public static void main(String[] args) {  
2     EntierTransformable x = new EntierTransformable(2);  
3     x.transform(v -> v * 2);  
4     x.transform(v -> v + 15);  
5     x.transform(v -> 0);  
6 }
```

3. Écrivez une classe `Additionneur` dont les objets peuvent être utilisés comme fonction $x \rightarrow x + n$ pour la classe `EntierTransformable`.

En particulier, le programme ci-dessous doit afficher 15 :

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

Correction :

```
1 public class Additionneur implements UnaryOperator<Integer> {
2     private final int toAdd;
3
4     public Additionneur(int toAdd) {
5         this.toAdd = toAdd;
6     }
7
8     @Override
9     public Integer apply(Integer integer) {
10        return integer + toAdd;
11    }
12 }
```

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 ?

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

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

Correction : Il suffit de changer dans `Transformable`

```
1 Transformable<T> transform(UnaryOperator<T> trans);
```

et dans `EntierTransformable`

```
1 public EntierTransformable transform(UnaryOperator<Integer> trans){
2     val = trans.apply(val);
3     return this;
4 }
```

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 :

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

- `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 `IllegalArgumentException`) et `Optionnel<T> vide()` qui retourne un objet « vide », puis rendez le(s) constructeur(s) privé(s).

Correction :

```

1  public class Optionnel<T>{
2      private final T val;
3
4      private Optionnel(T val){
5          this.val = val;
6      }
7
8      public static<T> Optionnel<T> de(T elt){
9          if (elt == null)
10             throw new IllegalArgumentException();
11          return new Optionnel<T>(elt);
12      }
13
14      public static<T> Optionnel<T> vide(){
15          return new Optionnel<T>(null);
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 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 ») :

- 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 !)

```

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

```

- `MyArrayList<E> filtre(??? cond)` : retourne une nouvelle liste consistant en les éléments de `this` qui satisfont le prédicat `cond`.
- `void filtreInterne(??? cond)` : supprime de `this` les éléments de `this` qui ne satisfont pas le prédicat `cond`. Utilisez un itérateur !
- `<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)`)
- `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.
- `<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.

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

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`.

Correction :

```
1 //Les méthodes xxxxBis sont celle de la question 7
2
3 @SuppressWarnings("serial")
4 public class MyArrayList<E> extends ArrayList<E>{
5     public MyArrayList(){
6         super();
7     }
8
9     //sert à faire les tests plus facilement
10    public MyArrayList(Collection<? extends E> c){
11        super(c);
12    }
13
14    public MyArrayList<E> filtre(Predicate<E> cond){
15        MyArrayList<E> res = new MyArrayList<E>();
16        for(E elt: this){
17            if(cond.test(elt))
18                res.add(elt);
19        }
20        return res;
21    }
22
23    public void filtreInterne(Predicate<E> cond){
24        Iterator<E> it = iterator();
25        while(it.hasNext()){
26            E elt = it.next();
27            if(!cond.test(elt))
28                it.remove();
29        }
30    }
31
32    public <U> MyArrayList<U> map(Function<E,U> f){
33        MyArrayList<U> res = new MyArrayList<U>();
34        for(E elt: this){
35            res.add(f.apply(elt));
36        }
37        return res;
38    }
39
40    public Optionnel<E> trouve (Predicate<E> cond){
41        for(E elt: this){
42            if (cond.test(elt))
43                return Optionnel.<E>de(elt);
44        }
45        return Optionnel.<E>vide();
46    }
47
48    public <U> U reduit(U z, BiFunction<U, E, U> f){
49        U accu = z;
50        for(E elt: this){
51            accu =f.apply(accu,elt);
52        }
53        return accu;
54    }
55
56
57    //Question 7
58    //en utilisant méthode reduit
```

```

59 public <U> MyArrayList<U> mapBis(Function<E,U> f){
60     return
61         reduit(new MyArrayList<U>(),
62             (liste,elt) -> {liste.add(f.apply(elt));
63                 return liste;});
64 }
65
66 public MyArrayList<E> filtreBis(Predicate<E> cond){
67     return
68         reduit(new MyArrayList<E>(),
69             (liste,elt) -> {if(cond.test(elt)) liste.add(elt);
70                 return liste;});
71 }
72
73 //moins efficace car parcourt la liste en entier, même si trouvé en première position
74 public Optionnel<E> trouveBis (Predicate<E> cond){
75     return reduit(Optionnel.<E>vide(),
76         (opt, elt)-> (opt.estVide() && cond.test(elt)) ? Optionnel.<E>de(elt) : opt);
77 }
78 }

```

```

1 //partie du main correspondant à l'utilisation de reduit
2 List<Integer> l1 =
3     Arrays.asList(new Integer[]{20, 5, 8, 35, 45, 96, 12, 5, 36, 4});
4 MyArrayList<Integer> listInt = new MyArrayList<Integer>(l1);
5 System.out.println(listInt.<Integer>reduit(1,(n,m) -> n*m));
6 System.out.println(listInt.<Integer>reduit(listInt.get(0),(n,m) -> Math.max(n, m)));

```

Remarque : Dans l'idéal, on créerait l'interface `MyList<E>` étendant `list<E>` (en y ajoutant les méthodes demandées dans cet exercice), en l'implémentant par `MyArrayList<E>` et on changerait le type de retour de `map` et `filtre` en `MyList<E>`, afin d'autoriser de retourner un objet implémenté autrement.

Explication de `trouveBis` : la fonction appelée à chaque itération dit que si on a déjà un optionnel non-vidé, on le garde sans rien faire d'autre, sinon on regarde si l'élément de la collection en cours d'examen satisfait le prédicat ; si oui on retourne un optionnel qui le contient, si non on garde le dernier optionnel calculé (qui ne peut être que vidé). On remarque tout de même que la méthode est moins efficace, elle parcourt toute la liste quoi qu'il advienne.

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 vidé) 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 vidé, 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 vidé.

Correction : Ajouter les méthodes suivantes à la classe `Optionnel<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 }
```

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'arguments⁴.

4. Si ce constructeur provoque des *warnings* à la compilation, regardez si les annotations `@SafeVarargs` et/ou

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.

Correction : Voir fichier `ListeTableau.java` (en annexe).

`@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.