Université Paris Diderot **Programmation Orientée Objet**
L2, EIDD et M1 linguistiques **Année 2015-2016**

TP n° 10

Collections, itérateurs

Dans ce TP, nous allons nous familiariser avec les collections et la notion d'itérateur. Les deux premières parties consistent en l'implémentation des interfaces Set (collection qui contient des éléments différents (sans duplication)) et SortedSet (même chose, mais avec un ordre total sur les éléments). L'API Java définit déjà des implémentations efficaces de Set (HashSet, TreeSet...), mais c'est un bon exercice d'implémenter cette interface avec des approches élémentaires. La troisième partie est une courte application des collections aux cartes à jouer.

Notez que comme d'habitude, il n'est pas interdit de consulter la documentation de l'API correspondante (java.util qui définit les interfaces Set, Iterator...)

— `https://docs.oracle.com/javase/7/docs/api/`

Enfin, on rappelle qu'implémenter une interface consiste à :

— définir toutes les méthodes abstraites spécifiées dans l'interface ;

— remplir le *contrat* : c'est-à-dire que ce que fait la méthode concrète doit correspondre à la spécification indiquée dans la documentation de l'interface. Ce deuxième point ne peut pas être vérifié par le compilateur : à vous de réfléchir !

1 Retour au TD

Exercice 1 Implémentez les classes conçues ou esquissées en TD.

2 Implémentation de Set avec des tableaux

Exercice 2 On va d'abord définir un itérateur sur les tableaux. On profite de la généricité :

```
1 public class TabIter<E> implements Iterator<E> {
2     private E[] tableau;
3     ...
4 }
```

On rappelle qu'un itérateur fournit des méthodes pour parcourir des éléments :

— **boolean** hasNext() pour savoir s'il reste des éléments à parcourir,

— E next() pour obtenir le prochain élément à parcourir.

Implémentez ces méthodes dans TabIter. On utilisera un champ index qui stockera la position courante dans le tableau. On prendra comme convention que dès qu'un pointeur **null** est rencontré, il ne reste plus d'éléments à parcourir.

Observez (dans la documentation d'Iterator) que l'interface Iterator spécifie aussi une méthode remove() optionnelle. Comme toutes les méthodes abstraites d'Iterator doivent être définies dans TabIter, on doit aussi définir **void** remove(). Mais elle se contentera pour l'instant de lever l'exception UnsupportedOperationException (comme la méthode est optionnelle, cela suffit à remplir le contrat).

Le constructeur de TabIter ne prendra pas d'argument. N'hésitez pas à tester le bon fonctionnement des méthodes next() et hasNext().

Exercice 3 Dans un premier temps, notre implémentation de Set, TabSet, utilisera des tableaux de taille fixe (taille qui sera en paramètre du constructeur de TabSet). De même que les itérateurs, les collections sont *génériques*, on ne va pas s'en priver !

```
1 public class TabSet<E> implements Set<E> {
2     private E[] tableau;
3     ...
4 }
```

On adapte le code de l'exercice précédent pour que l'itérateur soit défini comme classe interne de TabSet. Les champs tableau et index sont définis dans TabSet.

Si on essaye de rajouter (méthode add) un élément à un ensemble "plein", l'exception IllegalArgumentException sera levée (question subtile : respecte-t-on le contrat, i.e. la spécification indiquée dans la documentation de Set ?). De plus, on n'autorise pas l'ajout de **null** au TabSet (ce pointeur est réservé aux cases vides du tableau sous-jacent) : on utilisera alors l'exception NullPointerException.

Voici les méthodes à implémenter :

- Iterator <E> iterator () renvoie un itérateur pour parcourir l'ensemble (on peut alors utiliser les boucles **for**(E e : tabset) {...} équivalentes à **while**(tabset . iterator () .hasNext ()) {E e = tabset . iterator () .next () ;...}}) ; on créera une instance de l'itérateur défini en classe interne,
- **int** size () renvoie le nombre d'éléments (ce n'est a priori pas la taille du tableau),
- **boolean** isEmpty() pour savoir si l'ensemble est vide,
- **boolean** contains (Object o) pour savoir si *o* appartient à l'ensemble,
- **boolean** add(E e) renvoie **true** et rajoute *e* à l'ensemble si *e* n'est pas déjà présent, renvoie **false** sinon,
- **boolean** addAll(Collection <? **extends** E> c) est similaire à add mais ajoute tous les éléments de *c* (*c* étant une collection, elle vient avec un itérateur qu'on pourra utiliser),
- **boolean** containsAll (Collection <?> c) est similaire à contains pour tous les éléments de *c*,
- Object [] toArray () convertit l'ensemble en tableau (attention ! c'est un nouveau tableau qui ne doit pas contenir de **null**),
- <T> T[] toArray(T[] a) remplit le tableau *a* par les éléments de l'ensemble (et **null** ensuite) si *a* est suffisamment grand, sinon, un nouveau tableau de type T[] est créé. Attention, pour créer ce tableau de même type que *a*, vous ne pouvez pas vous contenter du code suivant :

  ```
  1 T[] tab2 = (T[]) new Object[...]
  ```

  En effet, Object n'est a priori pas castable en T, et vous risquez d'obtenir des erreurs à l'exécution. Vous pouvez en revanche, *sans faire référence à* T, récupérer (dans un objet de classe Class) le type des éléments de *a*, de la façon suivante :

  ```
  1 Class c = a.getClass().getComponentType();
  ```

  Observez que vous manipulez ainsi une classe sans la connaître à l'avance, c'est ce qu'on appelle la *réflexion*. Pour instancier un tableau dont les éléments ont le type correspondant à un objet de classe Class, on a la méthode Array.newInstance(Class type , **int** length) définie dans par le paquet java.lang. reflect .Array.
- Pour l'instant, **boolean** remove(Object o), **boolean** removeAll(Collection <?> c), **boolean** retainAll (Collection <?> c) et **void** clear () lèvent l'exception UnsupportedOperationException.

En cas de doute, consultez la documentation. Faites des petits tests pour vous convaincre que chaque méthode fonctionne.

Exercice 4 On veut maintenant pouvoir enlever des éléments de l'ensemble. Pour cela, on va d'abord modifier la classe TabIter, l'itérateur sur les tableaux :

- On définit la méthode **void** remove() de façon à mettre à **null** la case courante du tableau (le dernier élément qui a été retourné par l'itérateur).
- Désormais, si on rencontre un pointeur **null** lorsqu'on parcourt le tableau, le prochain élément parcouru est le premier élément non-**null** après cette case, s'il existe. Modifiez en conséquence les méthodes next () et hasNext().

On implémente ensuite les méthodes suivantes dans la classe TabSet :

— **boolean** remove(Object o) renvoie **true** et enlève *o* de l'ensemble s'il est présent, renvoie **false** sinon,
— **boolean** removeAll(Collection<?> c) est similaire à remove pour tous les éléments de *c*,
— **boolean** retainAll (Collection<?> c) enlève les éléments de l'ensemble qui n'appartiennent pas à *c* (intersection), et renvoie **true** ssi l'ensemble a été effectivement modifié,
— **void** clear () enlève tous les éléments de l'ensemble.

Faites quelques tests.

Exercice 5 Modifiez la classe TabSet de façon à adapter la taille du tableau au fur et à mesure des ajouts :
— le constructeur de TabSet ne prend plus de taille en argument,
— lors de la création d'un TabSet, on crée un tableau de taille 10 (par exemple),
— lorsqu'on ajoute un élément, s'il ne reste plus de place dans le tableau, on crée un tableau de taille deux fois plus grande, on y copie tous les éléments de l'ancien tableau et on y ajoute le nouvel élément.

Respecte-t-on maintenant le contrat ?

3 Utiliser les collections : opérations sur les cartes à jouer

Exercice 6 Une carte est un couple de deux chaînes de caractères :
— la *valeur* : 7, 8, 9, 10, valet, dame, roi, as ;
— la *couleur* : pique, cœur, trèfle, carreau.

On définit la classe Carte avec deux champs non-mutables (finaux) correspondant à ces deux chaînes de caractères, les accesseurs correspondants, et un constructeur à deux arguments (un pour chaque chaîne).

Exercice 7 On veut interdire la présence de deux cartes de même valeur et couleur dans un ensemble de cartes : dans Carte, redéfinissez la méthode **boolean** equals(Object o) de façon à ce qu'elle renvoie **true** ssi *o* est une carte dont les champs valeur et couleur sont égaux à ceux de this.

Exercice 8 Normalement, les implémentations de Set que vous avez écrites dans ce TP ne testent que equals pour savoir si un objet est déjà présent. Mais des implémentations plus complexes (comme HashSet) testent aussi le hashCode() des objets, qui doit respecter le contrat suivant :

> Si deux objets sont égaux pour la méthode equals, alors un appel à hashCode() sur chacun de ces deux objets doit retourner le même **int**.

Vous devez donc redéfinir la méthode **int** hashCode() à chaque fois que vous redéfinissez la méthode equals. Ici, vous pouvez prendre par exemple :

```
1 @Override public int hashCode() {
2     return (2 * getValeur().hashCode() + 3 * getCouleur().hashCode());
3 }
```

Remarquez qu'une fonction aussi simple que **public int** hashCode() { **return** 0; } remplit aussi le contrat : pourquoi la solution proposée est-elle préférable ?

Exercice 9 On crée un HashMap<Carte,Integer> appelé scores pour accorder un certain nombre de points à chaque carte. Utilisez la méthode put(carte , score) pour le remplir.

Exercice 10 Définissez une classe EnsembleCartes qui étend TabSet<Carte> et qui comporte une méthode **int** score() pour obtenir le score total d'un ensemble de cartes (somme des scores de chaque carte). Vous utiliserez pour cela la méthode Integer get(Carte carte) du HashMap scores, ainsi que l'itération sur l'ensemble de cartes.