Université Paris Diderot **Programmation Orientée Objet**
L2 **Année 2021-2022**

TP n° 11

Collections et Itérateurs

Dans ce TP, nous allons nous familiariser avec les *collections* et la notion d'*itérateur*. La première partie (obligatoire) consiste en l'implémentation de l'interfaces Set (collection qui contient des éléments différents, sans duplication). 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 deuxième partie (facultative) est une courte application des collections.

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 vous en assurer !

1 Implémentation de Set avec des tableaux

Exercice 1 Itérateur sur les tableaux. Dans un premier temps on va définir un itérateur sur les tableaux. On rappelle qu'un itérateur correspond à une tête de lecture qui permet de lire les éléments d'un objet les uns après les autres. L'interface générique Iterator<E> 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 pour les tableaux dans une classe générique TestIter<E> :

```
public class TestIter<E> implements Iterator<E> {
    private E[] tableau;
    ...
}
```

Pour cela on utilisera un champ index qui stockera la position courante dans le tableau. L'itérateur devra lire les éléments du tableau de gauche à droite, et on prendra comme convention que dès qu'une case vide (un pointeur `null`) est rencontrée il ne reste plus d'éléments à parcourir. Cet itérateur ne permettra donc pas forcément de lire toutes les cases d'un tableau si certaines d'entre elles sont vides.

La méthode next() lèvera une exception IllegalStateException s'il n'y a pas d'élément suivant.

Observez (dans la documentation d'Iterator) que l'interface Iterator spécifie aussi une méthode remove() optionnelle. Cette méthode a une implémentation par défaut qui se contente de lever l'exception UnsupportedOperationException (comme la méthode est optionnelle, cela suffit à remplir le contrat). On ne la redéfinira pas pour le moment, mais plus tard on voudra pouvoir l'utiliser pour supprimer un élément d'un ensemble.

Testez votre itérateur. Écrivez un constructeur pour TestIter qui prend en argument un tableau, et dans une classe Test créez un objet de type TestIter et testez le bon fonctionnement des méthodes next() et hasNext().

Exercice 2 Une implémentation partielle : tableaux itérables. L'interface Iterable<E> de Java contient une seule méthode abstraite (en plus de deux méthodes par défaut), **public** Iterator<E> iterator(), qui doit créer un nouvel itérateur. On veut représenter nos ensembles à l'aide de tableaux itérables :

```
TabSet<E> implements Iterable<E>
```

Définissez une classe TabSet<E> contenant :

— un tableau d'éléments de E ;

— une classe interne TabIter **implements** Iterator<E> ;

— une méthode **public** Iterator<E> iterator() qui renvoie un TabIter ;

— un constructeur qui prend en argument un entier n, et construit un ensemble vide dont le tableau sous-jacent est de taille n. Attention, il est impossible de créer directement un tableau dont les éléments sont d'un type paramétrique E. Il faut donc utiliser la syntaxe suivante :

```
tableau = (E[]) new Object[n];
```

On remarquera que cela crée un warning à la compilation (de manière logique, puisque la correction des types ne peut pas être vérifiée). On peut supprimer ce warning en utilisant :

```
@SuppressWarnings("unchecked")
```

Le fonctionnement de l'itérateur TabIter sera différent de celui de TestIter de l'exercice précédent :

— il ne possède pas son propre tableau d'éléments, mais utilise celui de TabSet à la place ;

— il ne s'arrête plus lorsqu'il tombe sur une case vide, mais l'ignore et essaie de lire la case suivante, jusqu'à trouver la prochaine case non vide.

— En revanche la méthode TabIter peut encore lever une exception de la même manière que pour la classe TabIter.

Dans les questions suivantes, toutes les opérations de lecture et de modification du tableau doivent être gérées au moyen de la classe interne TabIter.

Exercice 3 Méthodes de base. Puisque TabSet<E> implémente Iterable<E>, si set appartient à la classe TabSet<E> on peut utiliser les boucles *foreach* avec la syntaxe **for**(E e : set){...} équivalentes au code suivant :

```
Iterator<E> it = new set.iterator();
while(it.hasNext()) {E e = it.next();...}});
```

Dans la classe TabSet, écrivez les méthodes suivantes (n'hésitez pas à utiliser une boucle *foreach*) :

- **public boolean** contains (Object o), qui vérifie si un objet *o* est dans l'ensemble.
- **int** size(), qui renvoie le nombre d'éléments (ce n'est a priori pas la taille du tableau),
- **boolean** isEmpty() pour savoir si l'ensemble est vide,

Exercice 4 Ajouter des éléments. On veut pouvoir ajouter des objets dans un ensemble. La caractéristique d'un ensemble est qu'un même élément ne peut pas y appartenir plusieurs fois. Dans notre implémentation il n'est donc pas utile (et même gênant) de faire apparaître un même élément dans plusieurs cases d'un même tableau.

Dans la classe TabSet, écrivez une méthode **public boolean** add(E e). Elle doit renvoyer **false** si l'élément e est **null**, s'il est déjà dans l'ensemble ou s'il n'y a plus de place, en affichant le cas de figure à chaque fois. Sinon elle range e à la première place libre et renvoie **true**.

Exercice 5 Supprimer des éléments On veut maintenant implémenter des méthodes permettant de supprimer des éléments.

- Dans la classe TabIter, écrivez une méthode **public void** remove() qui met simplement à **null** la dernière case du tableau à avoir été lue (c'est-à-dire la dernière case du tableau dont la valeur a été renvoyée par la méthode next()). L'idée est que le code suivant :

  ```
  E e = it.next();
  it.remove();
  ```

  supprime l'élément e du tableau. Si cette méthode est appelée alors que l'itérateur n'a pas encore commencé son parcours de l'ensemble, elle levera une IllegalStateException.
- Utilisez la méthode précédente pour écrire dans la classe TabSet une méthode **public boolean** remove (Object o) qui renvoie **true** et enlève *o* de l'ensemble s'il est présent, renvoie **false** sinon,
- Dans la classe TabSet, écrivez une méthode **void** clear() qui enlève tous les éléments de l'ensemble. Cette méthode fera appel au remove() de la classe TabIter mais pas à celui de la classe TabSet afin de s'assurer de ne parcourir le tableau qu'une fois.

Exercice 6 Implémenter l'interface Set. On souhaite désormais que TabSet<E> implémente l'interface Set<E>. Écrivez des méthodes :

- **boolean** containsAll(Collection<?> c), qui renvoie **true** ssi tous les éléments de c appartiennent à l'ensemble. ;
- **boolean** addAll(Collection<? **extends** E> c), qui est similaire à add mais ajoute tous les éléments de c, et renvoie **true** ssi l'ensemble a été effectivement modifié ;
- **boolean** removeAll(Collection<?> c) qui est similaire à remove pour tous les éléments de c, et renvoie **true** ssi l'ensemble a été effectivement modifié ;
- **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 ;

Notez que la classe Collection implémente l'interface Iterable, il est donc possible d'itérer sur les éléments avec une boucle *foreach*. Fournissez également une implémentation par défaut des méthodes manquantes de l'interface Set, qui lance une exception UnsupportedOperationException.

Exercice 7 Test. Dans une classe Test, créez des instances de TabSet<Integer>, TabSet<Boolean>, ..., et testez dessus chacune des méthodes précédentes.

Exercice 8 Implémentez les méthodes suivantes :

- 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, il est demandé (dans la spécification de Set), que le type à l'exécution (*runtime type*) de l'objet retourné soit le même que celui de a. Cela veut dire que pour créer ce tableau de même type que a, vous ne pouvez pas vous contenter du code suivant :

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

  En effet, dans ce cas le tableau serait de type Object[] 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 :

```
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 dispose de la méthode Array.newInstance(Class type, **int** length) définie dans le paquet java.lang.reflect.Array. Notez que vous aurez toujours besoin du "cast" (T[]) et de @SuppressWarnings.

Exercice 9 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 fixe (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.