Université Paris Diderot **Programmation Orientée Objet**
L2 **Année 2023-2024**

TP n° 11

Itérateurs et Itérables, Collections, Ensembles

L'interface prédéfinie `Set` modélise la notion d'*ensemble*, une collection d'éléments sans duplication. Elle est une extension de l'interface `Collection`, modélisant la notion plus générale de collection d'éléments quelconque (*i.e.* avec des duplications éventuelles).

L'interface `Collection` est elle-même une extension de l'interface `Iterable` implémentée par tous les objets à même de construire un *itérateur*, un objet délivrant une certaine suite de valeurs à la demande – ou, de manière équivalente, à même d'être soumis à une boucle *for-each*, dont le corps sera exécuté pour chacune des valeurs de cette suite.

La librairie de Java fournit déjà des implémentations efficaces de l'interface `Set` (`HashSet`, `TreeSet` ...), mais à titre d'exercice, nous allons implémenter cette interface avec des moyens élémentaires. On rappelle qu'implémenter une interface consiste à :

— donner une implémentation effective de toutes ses méthodes ;

— remplir le *contrat* de cette interface : chaque méthode doit respecter la spécification donnée dans sa documentation.

Le deuxième point ne peut évidemment être vérifié par le compilateur : c'est à vous assurer que le contrat est respecté.

Exercice 1 Itérateurs.

Rappelons (*c.f.* TD 7) qu'un *itérateur* en Java est un objet capable de délivrer, à la demande, une certaine suite de valeurs. Un itérateur doit disposer de deux méthodes : une méthode permettant de déterminer s'il reste des valeurs à délivrer ; une méthode délivrant la valeur suivante.

L'interface générique `Iterator` modélise cette capacité avec les deux noms de méthodes suivants (plus deux autres méthodes avec une implémentation par défaut, pour le moment sans importance) :

```
// returns true if the iteration has more elements :
boolean hasNext();
// returns the next element in the iteration :
E next();
```

Écrire une classe `ArrayIterator<E>` implémentant cette interface, sur le modèle suivant :

```
public class ArrayIterator<E> implements Iterator<E> {
    private E[] elements;
    ...
}
```

La suite des éléments délivrés par une instance de cette classe sera celle des valeurs du tableau `elements` différentes de `null`. La méthode `next()` lèvera une exception `IllegalStateException` s'il n'y a pas d'élément suivant.

Ajouter à cette classe un constructeur prenant en argument un tableau, et tester le fonctionnement des méthodes `next()` et `hasNext()` dans un `main`, par exemple sur des instances de `ArrayIterator<String>` ou de `ArrayIterator<Integer>`.

Exercice 2 Itérables.

Un objet encapsulant un certain ensemble de valeurs est *itérable* s'il est capable de construire à la demande un itérateur délivrant ces valeurs [1]. L'interface prédéfinie `Iterable` modélise cette capacité à l'aide du nom de méthode suivant.

```
// returns an iterator over elements of type T.
Iterator<T> iterator()
```

Écrire une classe `ArraySet<E> implements Iterable<E>`, contenant les éléments suivants :

1. Un champ `E[] elements`.
2. Un constructeur `public ArraySet(E[] elements)` permettant d'initialiser ce champ.
3. Une classe interne `ArrayIterator implements Iterator<E>` ;
4. Une méthode `public Iterator<E> iterator()` renvoyant une nouvelle instance de `ArrayIterator<E>`.

   Le comportement de cet itérateur sera similaire à celui de l'exercice précédent, mais la suite des valeurs délivrées par cet itérateur sera cette fois celle des valeurs différentes de `null` du tableau `elements` de son objet englobant.

Testez à nouveau votre code. Rappelons qu'un objet implémentant `Iterable` peut être soumis à une boucle *for-each*. Les deux fragments de code ci-dessous sont opératoirement équivalents :

```
// premiere forme
Iterator<E> it = arrSet.iterator();
while (it.hasNext()) {
  E e = it.next();
  // ...
}

// seconde forme
for (E e : arrSet) {
  // ...
}
```

Exercice 3 Méthodes de base.

La classe `ArraySet<E>` implémentant l'interface `Iterable<E>`, elle peut être vue comme une première approximation d'une implémentation de `Set<E>` (qui est une extension de `Iterable<E>`) basée sur les tableaux. Dans cet exercice et les suivants, nous allons compléter cette classe et sa classe interne de manière à implémenter toutes les méthodes de `Set<E>`.

En anticipant un peu, nous appellerons *ensembles* les instances de `ArraySet`, et *élément* d'un ensemble tout objet référencé par une valeur (non nulle) de son tableau `element`.

Ajouter à la classe `ArraySet` les méthodes suivantes (n'hésitez pas à utiliser des boucles *for-each* lorsque cela permet d'alléger l'écriture) :

1. `public boolean contains(Object o)` vérifiant si `o` est élément d'un ensemble.
2. `int size()`, renvoyant le nombre d'éléments d'un ensemble.
3. `boolean isEmpty()` renvoyant `true` si un ensemble est vide.

1. Un itérateur ne peut délivrer une suite de valeurs qu'une seule fois, jusqu'à ce que sa méthode `hasNext()` réponde `false`. Un itérable peut construire un itérateur un nombre arbitraire de fois.

Exercice 4 Ajouts d'éléments.

Ajouter à la classe `ArraySet` une méthode `public boolean add(E e)`. La méthode doit ajouter `e` aux éléments d'un ensemble puis renvoyer `true`, à condition que :

— `e` ne soit pas égal à `null` ;

— `e` ne soit pas déjà une élément de l'ensemble ;

— la taille maximale de l'ensemble ne soit pas déjà atteinte.

Si l'une de ces conditions n'est pas satisfaite, l'ensemble ne sera pas modifié, et la méthode renverra `false`.

Exercice 5 Suppressions d'éléments

1. Ajouter à la classe interne `ArrayIterator` une méthode `public void remove()`. Cette méthode devra retirer d'un ensemble le dernier élément renvoyé par la toute dernière invocation de `next()`. Si cette méthode est invoquée avant au moins une invocation de `next()`, elle renverra une `IllegalStateException`.
2. Ajouter à la classe `ArraySet` une méthode `public boolean remove(Object o)`. Si `o` élément d'un ensemble, cette méthode doit le retirer de cet ensemble et renvoyer `true`. Sinon, elle doit renvoyer `false` (servez-vous de la méthode précédente sur un itérateur).
3. Ajouter à la classe `ArraySet` une méthode `void clear()` supprimant d'un ensemble tous ses éléments (servez-vous à nouveau de la méthode `remove()` d'un itérateur).

Exercice 6 Inclusion, Union, Complémentaire, Intersection

Ajouter à la classe `ArraySet` les méthodes suivantes – noter que l'interface `Collection` est une extension de `Iterable`, on peut donc soumettre à un *for-each* tout objet implémentant cette interface :

1. `boolean containsAll(Collection<?> c)` renvoyant `true` si tous les éléments de c appartiennent à un ensemble donné.
2. `boolean addAll(Collection<? extends E> c)` ajoutant à un ensemble (par `add`) tous les éléments de `c`, puis renvoyant `true` si l'ensemble a été effectivement modifié.
3. `boolean removeAll(Collection<?> c)` retirant d'un ensemble (par `remove`) pour tous les éléments de `c`, puis renvoyant `true` si l'ensemble a été effectivement modifié ;
4. `boolean retainAll(Collection<?> c)` retirant d'un ensemble tous les éléments qui n'appartiennent pas à `c`, puis renvoyant `true` si l'ensemble a été effectivement modifié.

Testez votre code. L'interface `Collection` est par exemple implémentée par la classe `LinkedList`, ou encore `Vector`.

Exercice 7 Extraction des éléments

Implémentez les méthodes suivantes :

— `Object[] toArray()` renvoyant un (nouveau) tableau d'objets contenant tous les éléments d'un ensemble (donc aucune reférence nulle).

— `<T> T[] toArray(T[] a)`.

Si le tableau `a` est suffisamment grand, cette méthode doit stocker en début du tableau tous les éléments de l'ensemble en le complétant par des `null`, puis renvoyer `a`.

Sinon, elle doit créer un nouveau tableau de même type que `a` (*c.f.* les remarques ci-dessous) de taille exactement égale au nombre d'éléments de l'ensemble, stocker dans ce tableau ses éléments, puis renvoyer le tableau.

Remarques. Pour l'implémentation de `<T> T[] toArray(T[] a)`, il est demandé dans la spécification de `Set` que dans le second cas, le type à l'exécution (*runtime type*) du tableau créé soit le même que celui de `a`[2]. L'écriture suivante est interdite (et ne garantirait pas même que `t` et `a` soient de même type) :

```
T[] t = new T[this.size()]; // ERREUR
```

On ne peut pas non plus se contenter du code suivant, qui crée un tableau de type `Object[]` :

```
T[] t = (T[]) new Object[this.size()]; // PROBLEME
```

Il est en revanche possible de demander la construction d'un objet qui, à l'exécution, *représentera* le type des éléments de `a` (via ce qu'on appelle la *réflexion*, l'usage d'objets exprimant les propriétés d'un objet telles que sa classe, les champs et méthodes de celle-ci, pour un tableau le type de ses éléments, etc) :

```
Class<?> c = a.getClass().getComponentType();
```

La méthode statique `Object Array.newInstance(Class type, int length)` (*c.f.* `Array`) permet d'autre part de créer un tableau de taille `length`, le type des éléments du tableau étant celui décrit par l'objet `type` :

```
T[] t = (T[])  Array.newInstance(c, this.size());
```

La conversion explicite d'une référence de type `Object` vers une reférence de type `T[]` entraînera cependant un message d'alerte du compilateur, que l'on peut supprimer en faisant précéder la méthode de l'annotation `@SuppressWarnings("unchecked")`.

Exercice 8 Modifiez la classe `ArraySet` de manière à adapter la taille du tableau au fur et à mesure des ajouts d'éléments à un ensemble. Le constructeur de `ArraySet` ne prendra plus de tableau en argument : le tableau `elements` sera créé dans ce constructeur avec une certaine taille initiale (mettons, 10), et la taille sera doublée avant un ajout lorsque celui-ci est plein. Il vous faudra utiliser le même procédé que celui décrit dans la remarque de l'exercice précédent (réflexion).

2. Il s'agira d'un type de la forme `B[]`, où `B` sera le type spécifié pour les éléments du tableau au moment de sa création (*e.g.* par `new B[10]`). Ce type ne sera pas nécessairement la valeur donnée à `T`.

Une déclaration de classe crée implicitement un nouveau type de même nom que celle-ci : dans une écriture telle que `A r = new B()`, où `A` est ancêtre de `B`, on peut sans ambiguïté parler de la *classe de* `r` (la classe de nom `B`, dont l'objet référencé par `r` est une instance) et du *type de* `r` (le nom de classe ou d'interface `A`, qui est le type de la variable `r`).

Dans `A[] a = new B[10]` par contre, `A[]` et `B[]` ne sont pas des noms de classes mais seulement des *types*. Le choix des concepteurs du langage est de convenir du fait que ce qu'on appelle "type de `a`" est dans ce cas `B[]`, et non le type `A[]` de la variable `a`.

Avec cette convention, le type d'un tableau est spécifié dès sa création, et invariable au cours du temps : un tableau créé avec `new B[10]` est de type `B[]`, et ses cases ne pourront recevoir que des valeurs de type `B`. Si `A` est ancêtre strict de `B`, les règles de typage autorisent l'écriture `A[] a = new B[10]`, mais ceci ne change pas le type du tableau : il sera impossible d'ajouter à `a` une instance de `A` sans provoquer une `ArrayStoreException`.