Chapitre 10

Les Conteneurs, ou Structures Collectives, en Java

On a souvent besoin de stocker ensemble (c'est à dire dans une même structure) un certain nombre d'objets similaires. Par exemple, les utilisateurs d'une bibliothèque, ou l'ensemble des pièces d'un jeu ...

On parle d'une *collection* d'objets, et Java définit toute une panoplie de *structures collectives*, c'est à dire de classes permettant de réaliser facilement des manipulations courantes de ces collections : ajout, suppression, tri, etc.

1 Les tableaux : simples, mais limités

Le tableau constitue la plus simple des structures collectives. Les éléments y sont rangés en séquence et on accède très rapidement à n'importe quel élément à partir de son index, ce qui est facilité par la syntaxe utilisant les doubles crochets [ ]. Il peut suffire dans bien des applications. Nous commencerons par étudier la classe java Arrays, qui propose un ensemble de méthodes utiles pour manipuler des tableaux.

Cependant, on peut parfois se retrouver limité par le manque de souplesse des tableaux à l'utilisation :

- leur taille est déterminée une seule fois (au moment de l'instanciation), et on ne peut plus la modifier après coup ;
- on les parcoure avec un indice, et on peut facilement se tromper quand on écrit du code d'utilisation de ces indices ;
- ...

Dans ce cas, on pourra recourir à des structures de données plus évoluées pour ranger des données.

N.B. Les classes que nous étudierons dans ce chapitre ne sont pas chargées automatiquement par java. Elles font toutes partie d'un package nommé java.util qu'il faut charger

explicitement. Pour cela, il faut ajouter

```
import java.util.* ;
```

en tête des fichiers Java qui utilisent ce package.

2 La classe `Arrays` : Opérations de base sur les tableaux

La classe `Arrays` définit, à travers un certain nombre de méthodes statiques, des opérations utilitaires sur les tableaux communiqués en paramètre à ces méthodes. Ces opérations sont par exemple les suivantes :

- tri d'un tableau, par la méthode `sort(...)`,
- comparaison du *contenu* de 2 tableaux, par la méthode `equals(...)`,
- copie du tableau avec la méthode copyOf(...),
- remplissage d'un tableau avec un seule valeur, par la méthode `fill(...)`,
- recherche d'un élément dans un tableau *trié*, par la méthode `binarySearch(...)`,
- représentation en chaîne de caractères du contenu par la méthode (statique) toString(...),
- etc.

Ainsi, si `tab` est un tableau d'entiers, on peut le trier le plus simplement du monde par l'instruction

```
Arrays.sort(tab) ;
```

Les méthodes de Arrays sont surchargées de manière à accepter en paramètres des tableaux de n'importe quel type primitif, plus des tableaux de type Object.

N.B. Pour découvrir toutes les méthodes disponibles, il suffit de consulter la classe Arrays dans l'API Java.

3 Les comparateurs

Le tri d'un tableau suppose que les éléments du tableau soient comparables à l'aide d'une relation d'ordre. Les types primitifs sont automatiquement dotés d'une relation d'ordre naturelle. Par contre pour les types objet, il faut spécifier explicitement cette relation d'ordre en implantant une interface dédiée.

Un programmeur dispose de deux manières pour munir les objets qu'il définit d'une relation d'ordre.

3.1 L'objet implante l'interface `Comparable`

On a évoqué dans la partie 2 du chapitre 9 qu'en implantant l'interface Comparable, on s'engageait à implanter une méthode nommée compareTo(). Cette méthode accepte un

paramètre de n'importe quel type dérivé de Object, et retourne un entier :

int compareTo(Object o)

Elle permet le tri des objets que l'on définit, en indiquant comment on compare l'objet désigné par this (moi en PDL++) avec l'objet *o* reçu en paramètre.

L'entier retourné par `compareTo()` est
- négatif si moi est plus petit que *o* ;
- nul si moi et *o* sont égaux ;
- positif si si moi est plus grand que *o*.

N.B. La syntaxe à base de transtypage dans l'exemple ci-dessous est correcte, bien que datée. Les versions actuelles de Java (depuis Java 1.5) permettent une écriture plus souple grâce aux *types génériques*. Nous les présenterons au chapitre 13 de ce cours, mais nous nous en tenons pour l'instant à cette syntaxe utilisant le transtypage. Ceci afin de ne pas faire abstraction de ce concept que vous pourriez rencontrer dans vos projets de développement Java.

Exemple. On considère une classe Paire avec deux attributs entiers, et on fait le choix que deux instances de Paire sont comparées par rapport à la somme de leurs attributs.

```
public class Paire implements Comparable {
    public int x, y ;

    public int compareTo(Object o) {
      return (this.x + this.y) - (((Paire) o).x + ((Paire) o).y) ;
    }
}
```

Un tableau d'objets qui implantent l'interface `Comparable` peut être trié au moyen de la méthode statique `Arrays.sort(. . .)`.

Par exemple, soit un tableau tabPaires d'instances de Paire. On peut trier ce tableau par Arrays.sort(tabPaires). La méthode Arrays.sort() se sert de la méthode compareTo() pour ranger les instances les unes par rapport aux autres.

3.2 Une classe séparée implante l'interface `Comparator`

On peut aussi décrire la relation d'ordre non pas dans la classe, mais dans une classe extérieure, séparée de l'objet lui-même. On fournira, en plus du tableau à trier, une instance de cette relation d'ordre en paramètre à la méthode `Arrays.sort(. . .)`.

L'avantage est que l'on peut définir plusieurs manières de trier des objets (par ordre croissant, décroissant, etc.) Chacune est décrite dans sa propre classe. Au moment de réaliser le tri, on choisit la relation d'ordre à appliquer en la fournissant en paramètre.

Pour implanter l'interface `Comparator`, on doit définir deux méthodes :

– une méthode `boolean equals(Object o)`,
– une méthode `int compare(Object o1, Object o2)`.

N.B. La classe `Object` possède déjà une méthode `equals(...)` et donc celle-ci est héritée, mais il est vivement conseillé de la redéfinir quand même car equals() dans Object ne compare pas les contenus mais les références.

La méthode `compare(Object o1, Object o2)` accepte deux paramètres $o1$ et $o2$ qui sont les deux objets à comparer. Selon que $o1$ est plus petit, égal ou plus grand que $o2$, la méthode doit renvoyer respectivement un entier négatif, nul ou positif.

N.B. La remarque, dans la partie 3.1, sur la syntaxe datée s'applique aussi à l'exemple suivant.

Exemple.

```
import java.util.*;

public class DecroissantPaire implements Comparator {

    public int compare(Object o1, Object o2) {
      Paire p1 = (Paire) o1, p2 = (Paire) o2;
      return (p2.x + p2.y) - (p1.x + p1.y);
    }
}
```

Ainsi, on peut trier `tabPaires`, qui est un tableau de `Paire`, en ordre décroissant, au moyen de

```
Arrays.sort(tabPaires, new DecroissantPaire());
```

4 Conteneurs en Java

En plus des tableaux, Java fournit un certain nombre de conteneurs qui peuvent s'utiliser directement pour y ranger des objets.

On distingue deux types de conteneurs :
– les conteneurs de type `Collection` : ils permettent des groupements d'objets individuels,
– les conteneurs de type `Map` : ils permettent des groupements d'association de type clé/valeur.

On ne s'intéresse dans ce cours qu'aux conteneurs de type `Collection`. D'autres conteneurs seront étudiés en Licence 3.

Le type de conteneur `Collection` est lui même décomposé en plusieurs sous-types de conteneurs, parmi lesquels : les `List`, où l'ordre des éléments est primordial, et les `Sets`, qui ne peuvent pas contenir de doublons.

4.1 Méthodes communes à tous les objets `Collection`

Le type Collection est une *interface* (cf. chapitre 9), définie dans l'API Java. Elle définit un ensemble de méthodes qu'on peut appliquer à chaque objet de type Collection. Parmi ces méthodes :

- `add(Object o)`, pour ajouter un élément à la collection,
- `remove(Object o)`, pour retirer l'objet o de la collection, s'il y est présent,
- `contains(Object o)`, qui indique si l'objet o est présent ou pas dans la collection,
- `clear()`, qui permet de retirer tous les éléments de la collection,
- `size()`, qui indique combien d'élément sont contenus dans la collection,
- etc.

4.2 `ArrayList` : le conteneur à tout faire

En première approche, une `ArrayList` peut être vue comme un tableau dont la taille peut changer automatiquement.

Ce conteneur implante l'interface `Collection`, et il dispose donc de toutes les méthodes communes aux collections. De plus, il peut être manipulé comme un tableau par l'utilisation d'indices. La syntaxe avec les doubles crochets n'est cependant pas valable, et il faut passer par des méthodes dédiées. Voici un aperçu de ces méthodes.

En plus de la méthode `add(Object o)` dont dispose toute collection, `ArrayList` possède une méthode

`add(int i, Object o)`

pour insérer un objet o à un indice i donné (les éléments aux indices supérieurs sont décalés en conséquence). Si l'indice i n'est pas précisé en paramètre, l'objet est ajouté à la fin.

Pour retirer un élément d'indice i sans « laisser de trou », c'est à dire en redécalant vers la gauche tous les éléments qui suivent : remove(i)

Pour remplacer l'élément d'indice i par un autre élément (sans décaler le reste du tableau), on dispose de la méthode

`set(int i, Object o)`.

On accède à un élément d'indice i au moyen de la méthode

`get(int i)`.

L'indice d'un objet o (s'il est présent dans le tableau) peut être obtenu au moyen de la méthode

`indexOf(Object o)`.

Pour connaître les détails d'utilisation de ces méthodes, ainsi que les autres méthodes disponibles, il faut consulter l'API de la classe `ArrayList`.

Dans beaucoup de situations courantes où un conteneur est nécessaire, le conteneur `ArrayList` est suffisant.

Exemple.

```
ArrayList al = new ArrayList();
al.add(new Rectangle(2, 3));
al.add(new Rectangle(5, 5));
al.add(new Cercle(8));
al.add(new Rectangle(10, 10));
al.remove(2);
```

Ce code range deux rectangles (2×3 et 5×5) dans une ArrayList nommée al, puis un cercle de rayon 8, puis à nouveau un rectangle 10×10. Ensuite, l'instruction al.remove(2) retire le cercle et la collection ne contient plus que les 3 rectangles rangés consécutivement.

Exercice. Implantez en Java une structure de type FIFO (file d'attente : *First In, First Out*) au moyen d'une ArrayList.

4.3 Les autres conteneurs de type `Collection`

Parmi les conteneurs de type `Collection`, il existe deux autres types de `List`.

- La classe `Vector`. Elle présente des fonctionnalités similaires à `ArrayList`, mais elle est devenue obsolète.
- La classe `LinkedList`. La structure sous-jacente est celle de liste chaînée.

Il existe également, parmi les conteneurs de type `Collection`, des conteneurs qui ne sont pas des listes mais des ensembles (`Set`). Un `Set` se distingue d'une liste par le fait qu'on ne peut pas y trouver plusieurs occurrences du même objet.

- La classe `HashSet` : elle repose sur la structure de table de hachage.
- La classe `TreeSet` : elle repose sur une structure d'arbre.

5 Les itérateurs

Si l'on ne veut pas avoir à se poser la question du type d'un conteneur pour savoir comment le parcourir (par les méthodes propres à ce conteneur), on peut recourir au concept d'itérateur.

Un itérateur est un objet permettant de parcourir les éléments d'un conteneur sans avoir à se préoccuper du *type* du conteneur. L'itérateur désigne un élément en particulier du conteneur, et on fait varier cet élément désigné afin de passer en revue tous les éléments du conteneur.

Tout conteneur en Java possède une méthode intitulée `iterator()`, qui renvoie un objet de type Iterator, c'est à dire un itérateur sur ce conteneur. A l'origine, l'itérateur est positionné sur le premier élément du conteneur.

Remarque. Iterator est une interface de l'API Java.

On parcoure alors le conteneur au moyen d'une séquence d'appels à la méthode `next()` de l'itérateur : chaque invocation de `next()` renvoie l'objet actuellement désigné dans le conteneur, puis se positionne sur l'élément suivant.

La méthode `hasNext()` de l'itérateur permet de vérifier s'il existe *encore* d'autres objets non encore parcourus dans le conteneur.

La méthode `remove()` de l'itérateur permet de retirer du conteneur parcouru par l'itérateur le dernier élément renvoyé par la méthode `next()`.

Dans l'exemple suivant, on considère une ArrayList nommée tab, que l'on remplit avec 10 instances d'une classe A quelconque. Ensuite on parcoure tab au moyen d'un itérateur en affichant à chaque fois l'élément courant.

Exemple.

```
import java.util.*;

public class Iter {
  public static void main(String[] args) {
    ArrayList tab = new ArrayList();
    for (int i=0; i < 10; i++)
      tab.add(new A());
    Iterator it = tab.iterator();
    while (it.hasNext())
      System.out.println(it.next());
  }
}
```

ou encore, pour afficher les périmètres des 3 rectangles contenus dans la liste al de tout à l'heure :

```
Iterator it = al.iterator();
while (it.hasNext())
      System.out.println( ((Forme) it.next()).perimetre() );
```

N.B. Dans l'exemple précédent, il est nécessaire de transtyper le résultat de it.next(), car it.next() renvoie une référence à un Object. Or la méthode perimetre() n'est pas définie dans la classe Object. Par contre elle est définie de manière abstraite dans la classe Forme. L'utilisation de types génériques (voir le chapitre 13) permettra d'éviter ce transtypage.