Claude Delannoy

Programmer en Java

9e édition

© Groupe Eyrolles, 2000-2014, ISBN : 978-2-212-14007-1

22

Les collections et les algorithmes

Java dispose d'une bibliothèque de classes utilitaires (*java.util*) permettant de manipuler les principales structures de données que sont les vecteurs dynamiques, les ensembles, les listes chaînées, les queues et les tables associatives. Ces classes ont beaucoup évolué au fil des différentes versions de Java, mais leurs concepteurs ont cherché à privilégier la simplicité, la concision, l'homogénéité, l'universalité (notamment par la généricité) et la flexibilité. C'est ainsi que les classes relatives aux vecteurs, aux listes, aux ensembles et aux queues implémentent une même interface (*Collection*) qu'elles complètent de fonctionnalités propres. Nous commencerons par examiner les concepts communs qu'elles exploitent ainsi : généricité, itérateur, ordonnancement et relation d'ordre. Nous verrons également quelles sont les opérations qui leur sont communes : ajout ou suppression d'éléments, construction à partir des éléments d'une autre collection...

Nous étudierons ensuite en détail chacune de ces structures, à savoir :

- les listes, implémentées par la classe *LinkedList*,
- les vecteurs dynamiques, implémentés par les classes *ArrayList* et *Vector*,
- les ensembles, implémentés par les classes *HashSet* et *TreeSet*,
- les queues avec priorité, implémentées par la classe *PriorityQueue* (introduite par le JDK 5.0) ;
- les queues à double entrée, implémentées par la classe *ArrayDeque* (introduite par Java 6).

Puis nous vous présenterons des algorithmes à caractère relativement général permettant d'effectuer sur toutes ou certaines de ces collections des opérations telles que la recherche de maximum ou de minimum, le tri, la recherche binaire...

Nous terminerons enfin par les tables associatives qu'il est préférable d'étudier séparément des autres collections car elles sont de nature différente (notamment, elles n'implémentent plus l'interface *Collection* mais l'interface *Map*).

Notons que Java 8 a introduit de nouvelles fonctionnalités permettant notamment d'associer un strream à une collection, introduisant ainsi des facilités réservées traditionnellemnt aux langages dits fonctionnels et facilitant également le calcul parallèle.

1 Concepts généraux utilisés dans les collections

1.1 La généricité suivant la version de Java

Depuis le JDK 5.0, les collections sont manipulées par le biais de classes génériques implémentant l'interface *Collection<E>*, *E* représentant le type des éléments de la collection. Tous les éléments d'une même collection sont donc de même type *E* (ou, à la rigueur, d'un type dérivé de *E*). Ainsi, à une liste chaînée *LinkedList<String>*, on ne pourra pas ajouter des éléments de type *Integer* ou *Point*.

Avant le JDK 5.0, les collections (qui implémentaient alors l'interface *Collection*) pouvaient contenir des éléments d'un type objet quelconque. Par exemple, on pouvait théoriquement créer une liste chaînée (*LinkedList*) contenant à la fois des éléments de type *Integer*, *String*, *Point*... En pratique, ce genre de collection hétérogène était peu employé, de sorte que le JDK 5.0 n'apporte pas de véritable limitation sur ce plan.

En revanche, avant le JDK 5.0, comme nous le verrons par la suite, l'accès à un élément d'une collection nécessitait systématiquement le recours à l'opérateur de cast. Par exemple, avec une liste chaînée d'éléments supposés être de type *String*, il fallait employer systématiquement la conversion *(String)* à chaque consultation. En outre, rien n'interdisait d'introduire dans la liste des éléments d'un type autre que *String* avec, à la clé, des risques d'erreur d'exécution dues à des tentatives de conversions illégales lors d'une utilisation ultérieure de l'élément en question. Depuis le JDK 5.0, la collection étant simplement paramétrée par le type, le recours au cast n'est plus nécessaire. En outre, il n'est plus possible d'introduire par mégarde des éléments d'un type différent de celui prévu car ces tentatives sont détectées dès la compilation.

D'une manière générale, les modifications apportées par le JDK 5.0 aux collections sont suffisamment simples pour que nous traitions simultanément l'utilisation des collections avant et depuis le JDK 5.0. Ce parallèle, complété par les connaissances exposées dans le chapitre

relatif à la programmation générique, vous permettra, le cas échéant, de combiner des codes génériques et des codes non génériques.

Par souci de simplification, lorsqu'elle sera triviale, la différence entre générique et non générique pourra ne pas être explicitée. Par exemple, au lieu de dire "l'interface *Collection<E>* (*Collection* avant le JDK 5.0)", nous dirons simplement "l'interface *Collection<E>*" ou "l'interface *Collection*>" suivant que *E* aura ou non un intérêt dans la suite du texte. En revanche, dans les exemples de code, nous ferons toujours la distinction ; plus précisément, le code est écrit de façon générique et des commentaires expriment les modifications à apporter pour les versions antérieures au JDK 5.0.

Dans tous les cas, on ne perdra pas de vue que lorsqu'on introduit un nouvel élément dans une collection Java, on n'effectue pas de copie de l'objet correspondant. On se contente d'introduire dans la collection la référence à l'objet. Il est même permis d'introduire la référence *null* dans une collection (cela n'aurait pas de sens si l'on recopiait effectivement les objets). Cette possibilité devra toutefois être évitée, dans la mesure où elle pourra créer des ambiguïtés lorsque l'on aura affaire à des méthodes susceptibles de renvoyer la valeur *null* pour indiquer un déroulement anormal.

1.2 Ordre des éléments d'une collection

Par nature, certaines collections, comme les ensembles, sont dépourvues d'un quelconque ordonnancement de leurs éléments. D'autres, en revanche, comme les vecteurs dynamiques ou les listes chaînées voient leurs éléments naturellement ordonnés suivant l'ordre dans lequel ils ont été disposés. Dans de telles collections, on pourra toujours parler, à un instant donné, du premier élément, du deuxième, ... du nième, du dernier.

Indépendamment de cet ordre naturel, on pourra, dans certains cas, avoir besoin de classer les éléments à partir de leur valeur. Ce sera par exemple le cas d'un algorithme de recherche de maximum ou de minimum ou encore de tri. Lorsqu'il est nécessaire de disposer d'un tel ordre sur une collection, les méthodes concernées considèrent par défaut que ses éléments implémentent l'interface *Comparable* (*Comparable<E>* depuis le JDK 5.0) et recourent à sa méthode *compareTo*[1]. Mais il est également possible de fournir à la construction de la collection ou à l'algorithme concerné une méthode de comparaison appropriée par le biais de ce qu'on nomme un objet comparateur.

Bien que les ensembles soient des collections théoriquement non ordonnées, nous verrons que, pour des questions d'efficacité, leur implémentation les agencera de façon à optimiser les tests d'appartenance d'un élément. Mais seul le type *TreeSet* exploitera les deux possibilités décrites ici ; le type *HashSet* utilisera, quant à lui, une technique de hachage basée sur une autre méthode *hashCode*.

1. L'interface *Comparable* ne prévoit que cette méthode.

1.2.1 Utilisation de la méthode compareTo

Certaines classes comme *String*, *File* ou les classes enveloppes (*Integer*, *Float*...) implémentent l'interface *Comparable* et disposent donc d'une méthode *compareTo*. Dans ce cas, cette dernière fournit un résultat qui conduit à un ordre qu'on peut qualifier de naturel :

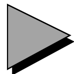
- ordre lexicographique pour les chaînes, les noms de fichier ou la classe *Character*,
- ordre numérique pour les classes enveloppes numériques.

Bien entendu, si vos éléments sont des objets d'une classe *E* que vous êtes amené à définir, vous pouvez toujours lui faire implémenter l'interface *Comparable* et définir la méthode :

```
public int compareTo (E o)       // public int compareTo (Object o)   <-- avant JDK 5.0
```

Celle-ci doit comparer l'objet courant (*this*) à l'objet *o* reçu en argument et renvoyer un entier (dont la valeur exacte est sans importance) :

- négatif si l'on considère que l'objet courant est "inférieur" à l'objet *o* (au sens de l'ordre qu'on veut définir),
- nul si l'on considère que l'objet courant est égal à l'objet *o* (il n'est ni inférieur, ni supérieur),
- positif si l'on considère que l'objet courant est "supérieur" à l'objet *o*.



Remarques

1 Notez bien que, avant le JDK 5.0, l'argument de *compareTo* était de type *Object*. Dans le corps de la méthode, on était souvent amené à le convertir dans un type objet précis. Il pouvait s'avérer difficile d'ordonner correctement des collections hétérogènes car la méthode *compareTo* utilisée n'était plus unique : son choix découlait de l'application des règles de surdéfinition et de polymorphisme. Elle pouvait même alors différer selon que l'on comparait *o1* à *o2* ou *o2* à *o1*.

2 Faites bien attention à ce que la méthode *compareTo* définisse convenablement une relation d'ordre. En particulier si *o1* < *o2* et si *o2* < *o3*, il faut que *o1* < *o3*.

3 Si vous oubliez d'indiquer que la classe de vos éléments implémente l'interface *Comparable<E>* (*Comparable* avant le JKD5.0), leur méthode *compareTo* ne sera pas appelée car les méthodes comparent des objets de "type *Comparable<E>*".

1.2.2 Utilisation d'un objet comparateur

Il se peut que la démarche précédente (utilisation de *compareTo*) ne convienne pas. Ce sera notamment le cas lorsque :

- les éléments sont des objets d'une classe existante qui n'implémente pas l'interface *Comparable*,
- on a besoin de définir plusieurs ordres différents sur une même collection.

Il est alors possible de définir l'ordre souhaité, non plus dans la classe des éléments mais :

- soit lors de la construction de la collection,
- soit lors de l'appel d'un algorithme.

Pour ce faire, on fournit en argument (du constructeur ou de l'algorithme) un objet qu'on nomme un comparateur. Il s'agit en fait d'un objet d'un type implémentant l'interface *Comparator<E>*[1] (ou *Comparator* avant le JDK 5.0) qui comporte une seule méthode :

```
public int compare (E o1, E o2)                 // depuis le JDK 5.0
public int compare (Object o1, Object o2)       // avant  le JDK 5.0
```

Celle-ci doit donc cette fois comparer les objets *o1* et *o2* reçus en argument et renvoyer un entier (dont la valeur exacte est sans importance) :

- négatif si l'on considère que *o1* est inférieur à *o2*,
- nul si l'on considère que *o1* est égal à *o2*,
- positif si l'on considère que *o1* est supérieur à *o2*.

Notez qu'un tel objet qui ne comporte aucune donnée et une seule méthode est souvent nommé *objet fonction*. On pourra le créer par *new* et le transmettre à la méthode concernée. On pourra aussi utiliser directement une classe anonyme. Nous en rencontrerons des exemples par la suite.

1.3 Égalité d'éléments d'une collection

Toutes les collections nécessitent de définir l'égalité de deux éléments. Ce besoin est évident dans le cas des ensembles (*HashSet* et *TreeSet)* dans lesquels un même élément ne peut apparaître qu'une seule fois. Mais il existe aussi pour les autres collections ; par exemple, même si elle a parfois peu d'intérêt, nous verrons que toute collection dispose d'une méthode *remove* de suppression d'un élément de valeur donnée.

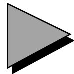
Cette égalité est, à une exception près, définie en recourant à la méthode *equals* de l'objet. Ainsi, là encore, pour des éléments de type *String*, *File* ou d'une classe enveloppe, les choses seront naturelles puisque leur méthode *equals* se base réellement sur la valeur des objets. En revanche, pour les autres, il faut se souvenir que, par défaut, leur méthode *equals* est celle héritée de la classe *Object*. Elle se base simplement sur les références : deux objets différents apparaîtront toujours comme non égaux (même s'ils contiennent exactement les mêmes valeurs). Pour obtenir un comportement plus satisfaisant, il faudra alors redéfinir la méthode *equals* de façon appropriée, ce qui ne sera possible que dans des classes qu'on définit soi-même.

---

1. Ne confondez pas *Comparable* et *Comparator*. D'autre part, depuis Java 8, on trouve en outre dans l'interface *Comparator*, une méthode statique *comparing* facilitant la création d'un comparateur à partir d'expressions lambda. Ces possibilités sont étudiées dans le chapitre relatif aux expressions lambda et aux stream.

Par ailleurs, comme nous l'avons déjà dit, il est tout à fait possible d'introduire la référence *null* dans une collection. Celle-ci est alors traitée différemment des autres références, afin d'éviter tout problème avec la méthode *equals*. Notamment, la référence *null* n'apparaît égale qu'à elle-même. En conséquence, un ensemble ne pourra la contenir qu'une seule fois, les autres types de collections pouvant la contenir plusieurs fois.

Enfin, et fort malheureusement, nous verrons qu'il existe une classe (*TreeSet*) où l'égalité est définie, non plus en recourant à *equals*, mais à *compareTo* (ou à un comparateur fourni à la construction de la collection). Néanmoins, là encore, les choses resteront naturelles pour les éléments de type *String*, *File* ou enveloppe.



Remarque

En pratique, on peut être amené à définir dans une même classe les méthodes *compareTo* et *equals*. Il faut alors tout naturellement prendre garde à ce qu'elles soient compatibles entre elles. Notamment, il est nécessaire que *compareTo* fournisse *0* si et seulement si *equals* fournit *true*. Cette remarque devient primordiale pour les objets que l'on risque d'introduire dans différentes collections (la plupart emploient *compareTo* par défaut, mais *TreeSet* emploie *equals*).

1.4 Les itérateurs et leurs méthodes

Ces itérateurs sont des objets qui permettent de "parcourir" un par un les différents éléments d'une collection. Ils ressemblent à des pointeurs (tels que ceux de C ou C++) sans en avoir exactement les mêmes propriétés.

Il existe deux sortes d'itérateurs :

- *monodirectionnels* : le parcours de la collection se fait d'un début vers une fin ; on ne passe qu'une seule fois sur chacun des éléments ;
- *bidirectionnels* : le parcours peut se faire dans les deux sens ; on peut avancer et reculer à sa guise dans la collection.

1.4.1 Les itérateurs monodirectionnels : l'interface Iterator

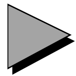
Propriétés d'un itérateur monodirectionnel

Chaque classe collection dispose d'une méthode nommée *iterator* fournissant un itérateur monodirectionnel, c'est-à-dire un objet d'une classe implémentant l'interface *Iterator<E>* (*Iterator* avant le JDK 5.0). Associé à une collection donnée, il possède les propriétés suivantes :

- À un instant donné, un itérateur indique ce que nous nommerons une *position courante* désignant soit un élément donné de la collection, soit la fin de la collection (la position courante se trouvant alors en quelque sorte après le dernier élément). Comme on peut s'y

attendre, le premier appel de la méthode *iterator* sur une collection donnée fournit comme position courante, le début de la collection.

- On peut obtenir l'objet désigné par un itérateur en appelant la méthode *next* de l'itérateur, ce qui, en outre, avance l'itérateur d'une position. Ainsi deux appels successifs de *next* fournissent deux objets différents (consécutifs).
- La méthode *hasNext* de l'itérateur permet de savoir si l'itérateur est ou non en fin de collection, c'est-à-dire si la position courante dispose ou non d'une position suivante, autrement dit si la position courante désigne ou non un élément.



Remarques

1 Depuis le JDK 5.0, la méthode *next* fournit un résultat de type *E*. Avant le JDK 5.0, elle fournissait un résultat de type général *Object*. La plupart du temps, pour pouvoir exploiter l'objet correspondant, il fallait effectivement en connaître le type exact et effectuer une conversion appropriée de la référence en question.

2 De toute évidence, pour pouvoir ne passer qu'une seule fois sur chaque élément, l'itérateur d'une collection doit se fonder sur un certain ordre. Dans les cas des vecteurs ou des listes, il s'agit bien sûr de l'ordre naturel de la collection. Pour les collections apparemment non ordonnées comme les ensembles, il existera quand même un ordre d'implémentation[1] (pas nécessairement prévisible pour l'utilisateur) qui sera exploité par l'itérateur. En définitive, toute collection, ordonnée ou non, pourra toujours être parcourue par un itérateur.

Canevas de parcours d'une collection

On pourra parcourir tous les éléments d'une collection *c<E>*, en appliquant ce canevas :

```
    // depuis JDK 5.0[2]                       // avant JDK 5.0
Iterator<E> iter = c.iterator () ;         Iterator iter = c.iterator() ;
while ( iter.hasNext() )                    while ( iter.hasNext() )
{ E o = iter.next () ;                      { Object o = iter.next() ;
  // utilisation de o                          // utilisation de o
}                                           }
```

Canevas de parcours d'une collection

1. Nous verrons qu'il s'agit précisément de l'ordre induit par *compareTo* (ou un comparateur) pour *TreeSet* et de l'ordre induit par la méthode *hashCode* pour *HashSet.*

2. Nous verrons un peu plus loin qu'il est possible dans certains cas de simplifier ce canevas en utilisant la boucle *for... each*.

La méthode *iterator* renvoie un objet désignant le premier élément de la collection s'il existe. La méthode *next* fournit l'élément désigné par *iter* et avance l'itérateur à la position suivante.

Si l'on souhaite parcourir plusieurs fois une même collection, il suffit de réinitialiser l'itérateur en appelant à nouveau la méthode *iterator*.

La méthode remove de l'interface Iterator

L'interface *Iterator* prévoit la méthode *remove* qui supprime de la collection le dernier objet renvoyé par *next*.

Voici par exemple comment supprimer d'une collection *c* tous les éléments vérifiant une *condition* :

```
     // depuis JDK 5.0                         // avant JDK 5.0
Iterator<E> iter = c.iterator() ;        Iterator iter = c.iterator() ;
while (c.iter.hasNext())                  while (c.iter.hasNext())
{ E = iter.next() ;                       { Object o = iter.next() ;
  if (condition) iter.remove() ;            if (condition) iter.remove() ;
}                                         }
```

Suppression d'une collection c *des éléments vérifiant une condition*

Notez bien que *remove* ne travaille pas directement avec la position courante de l'itérateur, mais avec la dernière référence renvoyée par *next* que nous nommerons *objet courant*. Alors que la position courante possède toujours une valeur, l'objet courant peut ne pas exister. Ainsi, cette construction serait incorrecte (elle conduirait à une exception *IllegalStateException*) :

```
Iterator<E> iter ;                          // Iterator iter ;   <-- avant JDK 5.0
iter = c.iterator() ;
iter.remove() ;           // incorrect
```

En effet, bien que l'itérateur soit placé en début de collection, il n'existe encore aucun élément courant car aucun objet n'a encore été renvoyé par *next*. Pour supprimer le premier objet de la collection, il faudra d'abord l'avoir "lu"[1], comme dans cet exemple où l'on suppose qu'il existe au moins un objet dans la collection *c* :

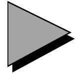
```
Iterator <E> iter ;                         // Iterator iter ;   <-- avant JDK 5.0
iter = c.iterator() ;
iter.next () ;       // se place après le premier objet
iter.remove() ;      // supprime le premier objet
```

On notera bien que l'interface *Iterator* ne comporte pas de méthode d'ajout d'un élément à une position donnée (c'est-à-dire en général entre deux éléments). En effet, un tel ajout n'est réalisable que sur des collections disposant d'informations permettant de localiser, non seule-

1. Ici encore, cette association entre la notion d'itérateur et d'élément renvoyé par *next* montre bien qu'un itérateur se comporte différemment d'un pointeur usuel.

ment l'élément suivant, mais aussi l'élément précédent d'un élément donné. Ce ne sera le cas que de certaines collections seulement, disposant précisément d'itérateurs bidirectionnels.

En revanche, nous verrons que toute collection disposera d'une méthode d'ajout d'un élément à un emplacement (souvent sa fin) indépendant de la valeur d'un quelconque itérateur. C'est d'ailleurs cette démarche qui sera le plus souvent utilisée pour créer effectivement une collection.



Remarques

1 Nous avons vu que la méthode *iterator* peut être appelée pour réinitialiser un itérateur. Il faut alors savoir qu'après un tel appel, il n'existe plus d'élément courant.

2 La classe *Iterator* dispose d'un constructeur recevant un argument entier représentant une position dans la collection. Par exemple, si *c* est une collection, ces instructions :

```
ListIterator <E> it ;                   // ListIterator it ;   <-- avant JDK 5.0
it = c.listIterator (5) ;
```

créent l'itérateur *it* et l'initialisent de manière à ce qu'il désigne le sixième élément de la collection (le premier élément portant le numéro 0). Là encore, on notera bien qu'après un tel appel, il n'existe aucun élément courant. Si l'on cherche par exemple à appeler immédiatement *remove*, on obtiendra une exception.

Par ailleurs, cette opération nécessitera souvent de parcourir la collection jusqu'à l'élément voulu (la seule exception concernera les vecteurs dynamiques qui permettront l'accès direct à un élément de rang donné).

Parcours unidirectionnel d'une collection avec for... each (JDK 5.0)

La boucle dite *for... each* permet de simplifier le parcours d'une collection *c<E>*, en procédant ainsi :

```
for (E o : c)
{ utilisation de o
}
```

Ici, la variable *o* prend successivement la valeur de chacune des références des éléments de la collection. Toutefois, ce schéma n'est pas exploitable si l'on doit modifier la collection, en utilisant des méthodes telles que *remove* ou *add* qui se fondent sur la position courante d'un itérateur.

1.4.2 Les itérateurs bidirectionnels : l'interface ListIterator

Certaines collections (listes chaînées, vecteurs dynamiques) peuvent, par nature, être parcourues dans les deux sens. Elles disposent d'une méthode nommée *listIterator* qui fournit un itérateur bidirectionnel. Il s'agit, cette fois, d'objet d'un type implémentant l'interface *ListIterator<E>* (dérivée de *Iterator<E>*). Il dispose bien sûr des méthodes *next*, *hasNext* et *remove* héritées de *Iterator*. Mais il dispose aussi d'autres méthodes permettant d'exploiter son caractère bidirectionnel, à savoir :

- comme on peut s'y attendre, des méthodes *previous* et *hasPrevious*, complémentaires de *next* et *hasNext*,
- mais aussi, des méthodes d'addition[1] d'un élément à la position courante (*add*) ou de modification de l'élément courant (*set*).

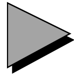
Méthodes previous et hasPrevious

On peut obtenir l'élément précédant la position courante à l'aide de la méthode *previous* de l'itérateur, laquelle, en outre, recule l'itérateur sur la position précédente. Ainsi deux appels successifs de *previous* fournissent deux objets différents.

La méthode *hasPrevious* de l'itérateur permet de savoir si l'on est ou non en début de collection, c'est-à-dire si la position courante dispose ou non d'une position précédente.

Par exemple, si *l* est une liste chaînée (nous verrons qu'elle est du type *LinkedList*), voici comment nous pourrons la parcourir à l'envers :

```
ListIterator <E> iter ;                        // ListIterator iter ;   <-- avant JDK 5.0
iter = l.listIterator (l.size()) ;  /* position courante : fin de liste*/
while (iter.hasPrevious())
{ E o = iter.previous () ;             // Object o=iter.previous() ;  <-- avant JDK 5.0
    // utilisation de l'objet courant o
}
```



Remarque

Un appel à *previous* annule, en quelque sorte, l'action réalisée sur le pointeur par un précédent appel à *next*. Ainsi, cette construction réaliserait une boucle infinie :

```
iter = l.listIterator () ;
E elem ;                                        // Object elem ;     <-- avant JDK 5.0
while (iter.hasNext())
{ elem = iter.next() ;
  elem = iter.previous() ;
}
```

Méthode add

L'interface *ListIterator* prévoit une méthode *add* qui ajoute un élément à la position courante de l'itérateur. Si ce dernier est en fin de collection, l'ajout se fait tout naturellement en fin de collection (y compris si la collection est vide). Si l'itérateur désigne le premier élément, l'ajout se fera avant ce premier élément.

Par exemple, si *c* est une collection disposant d'un itérateur bidirectionnel, les instructions suivantes ajouteront l'élément *elem* avant le deuxième élément (en supposant qu'il existe) :

```
ListIterator<E> it ;                           // ListIterator it ;   <-- avant JDK 5.0
it = c.listIterator() ;
it.next() ;       /* premier élément = élément courant */
it.next() ;       /* deuxième élément = élément courant */
it.add (elem) ;   /* ajoute elem à la position courante, c'est-à-dire */
                  /* entre le premier et le deuxième élément          */
```

On notera bien qu'ici, quelle que soit la position courante, l'ajout par *add* est toujours possible. De plus, contrairement à *remove*, cet ajout ne nécessite pas que l'élément courant soit

1. En général, une telle opération est plutôt nommée *insertion*. Mais, ici, la méthode se nomme *add* et non *insert* !

défini (il n'est pas nécessaire qu'un quelconque élément ait déjà été renvoyé par *next* ou *previous*).

Par ailleurs, *add* déplace la position courante après l'élément qu'on vient d'ajouter. Plusieurs appels consécutifs de *add* sans intervention explicite sur l'itérateur introduisent donc des éléments consécutifs. Par exemple, si *l* est une liste chaînée (de type *LinkedList*) et si l'on déclare :

```
ListIterator<E> it ;                              // ListIterator it ;   <-- avant JDK 5.0
it = l.ListIterator () ;
```

ces deux séquences sont équivalentes :

```
iter.add (el1) ;                                   l.add (el1) ;
iter.add (el2) ;                                   l.add (el2) ;
iter.add (el3) ;                                   l.add (el3) ;
```

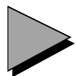
Méthode set

L'appel *set (elem)* remplace par *elem* l'élément courant, c'est-à-dire le dernier renvoyé par *next* ou *previous*, à condition que la collection n'ait pas été modifiée entre temps (par exemple par *add* ou *remove*). N'oubliez pas que les éléments ne sont que de simples références ; la modification opérée par *set* n'est donc qu'une simple modification de référence (les objets concernés n'étant pas modifiés).

La position courante de l'itérateur n'est pas modifiée (plusieurs appels successifs de *set*, sans action sur l'itérateur, reviennent à ne retenir que la dernière modification).

Voici par exemple comment l'on pourrait remplacer par *null* tous les éléments d'une collection *c* vérifiant une *condition* :

```
ListIterator<E> it ;                               // ListIterator it ;  <-- avant JDK 5.0
it = c.listIterator() ;
while (it.hasNext())
{ E o = it.next() ;                           // Object o = it.next() ;  <-- avant JDK 5.0
  if (condition) it.set(null) ;
}
```



Remarque

N'oubliez pas que *set*, comme *remove*, s'applique à un élément courant (et non comme *add* à une position courante). Par exemple, si *it* est un itérateur bidirectionnel, la séquence suivante est incorrecte et provoquera une exception *IllegalStateException* :

```
it.next() ;
it.remove() ;
it.set (el) ;
```

1.4.3 Les limitations des itérateurs

Comme on a déjà pu le constater, la classe *Iterator* ne dispose pas de méthode d'ajout d'un élément à un emplacement donné. Cela signifie que la seule façon d'ajouter un élément à une

collection ne disposant pas d'itérateurs bidirectionnels, consiste à recourir à la méthode *add* (définie par l'interface *Collection*) travaillant indépendamment de tout itérateur.

D'une manière générale, on peut dire que les méthodes de modification d'une collection (ajout, suppression, remplacement) se classent en deux catégories :

- celles qui utilisent la valeur d'un itérateur à un moment donné,
- celles qui sont indépendantes de la notion d'itérateur.

Cette dualité imposera quelques précautions ; en particulier, **il ne faudra jamais modifier le contenu d'une collection (par des méthodes de la seconde catégorie) pendant qu'on utilise un itérateur**. Dans le cas contraire, en effet, on a aucune garantie sur le comportement du programme.

1.5 Efficacité des opérations sur des collections

Pour juger de l'efficacité d'une méthode d'une collection ou d'un algorithme appliqué à une collection, on choisit généralement la notation dite "de Landau" (*O(...)*) qui se définit ainsi :

> *Le temps t d'une opération est dit en* O(x) *s'il existe une constante* k *telle que, dans tous les cas, on ait :* t <= kx.

Comme on peut s'y attendre, le nombre *N* d'éléments d'une collection pourra intervenir. C'est ainsi qu'on rencontrera typiquement :

- des opérations en *O(1)*, c'est-à-dire pour lesquelles le temps est constant (plutôt borné par une constante, indépendante du nombre d'éléments de la collection) ; on verra que ce sera le cas des additions dans une liste ou des ajouts en fin de vecteur ;
- des opérations en *O(N)*, c'est-à-dire pour lesquelles le temps est proportionnel au nombre d'éléments de la collection ; on verra que ce sera le cas des additions en un emplacement quelconque d'un vecteur.
- des opérations en *O(LogN)...*

D'une manière générale, on ne perdra pas de vue qu'une telle information n'a qu'un caractère relativement indicatif ; pour être précis, il faudrait indiquer s'il s'agit d'un maximum ou d'une moyenne et mentionner la nature des opérations concernées. Par exemple, accéder au *nième* élément d'une collection en possédant *N*, en parcourant ses éléments un à un depuis le premier, est une opération en *O(i)* pour un élément donné. En revanche, en moyenne (*i* étant supposé aléatoirement réparti entre 1 et *N*), ce sera une opération en *O(N/2)*, ce qui par définition de *O* est la même chose que *O(N).*

1.6 Opérations communes à toutes les collections

Les collections étudiées ici implémentent toutes au minimum l'interface *Collection*, de sorte qu'elles disposent de fonctionnalités communes. Nous en avons déjà entrevu quelques unes liées à l'existence d'un itérateur monodirectionnel (l'itérateur bidirectionnel n'existant pas

dans toutes les collections). Ici, nous nous contenterons de vous donner un aperçu des autres possibilités, sachant que ce n'est que dans l'étude détaillée de chacun des types de collection que vous en percevrez pleinement l'intérêt. D'autre part, les méthodes liées aux collections sont récapitulées en annexe G.

Par ailleurs, on notera bien que cette apparente homogénéité de manipulation par le biais de méthodes de même en-tête ne préjuge nullement de l'efficacité d'une opération donnée pour un type de collection donné. De même, certaines fonctionnalités pourront s'avérer peu intéressantes pour certaines collections : par exemple, les itérateurs s'avéreront peu usités avec les vecteurs dynamiques.

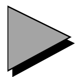
1.6.1 Construction

Comme on peut s'y attendre, toute classe collection *C<E>* dispose d'un constructeur sans argument[1] créant une collection vide :

```
C<E> c = new C<E>() ;                        // C c = new C () ;  <-- avant JDK 5.0
```

Elle dispose également d'un constructeur recevant en argument une autre collection (n'importe quelle classe implémentant l'interface *Collection* : liste, vecteur dynamique, ensemble) :

```
  /* création d'une collection c2 comportant tous les éléments de c */
C<E> c2 = new C<E> (c) ;                    // C c2 = new C (c) ;   <-- avant JDK 5.0
```



Remarque

Avant le JDK 5.0, les collections pouvaient être hétérogènes. Aucun contrôle n'était réalisé à la compilation concernant les types respectifs des éléments de *c2* et de *c*. Depuis le JDK 5.0, le type des éléments de *c* doit être compatible (identique ou dérivé) avec celui des éléments de *c2*. On peut s'en apercevoir en examinant (en annexe G) l'en-tête du constructeur correspondant. Par exemple pour *C* = *LinkedList*, on trouvera :

```
LinkedList (Collection <? extends E> c)
```

1.6.2 Opérations liées à un itérateur

Comme mentionné précédemment, toutes les collections disposent d'une méthode *iterator* fournissant un itérateur monodirectionnel. On pourra lui appliquer la méthode *remove* pour supprimer le dernier élément renvoyé par *next*. En revanche, on notera bien qu'il n'existe pas de méthode générale permettant d'ajouter un élément à une position donnée de l'itérateur. Une telle opération ne sera réalisable qu'avec certaines collections disposant d'itérateurs bidirectionnels (*ListIterator*) qui, eux, comportent une méthode *add*.

1. Bien qu'il s'agisse d'une propriété commune à toute collection, elle ne peut pas être prévue dans l'interface *Collection*, puisque le nom d'un constructeur est obligatoirement différent d'une classe à une autre.

1.6.3 Modifications indépendantes d'un itérateur

Toute collection dispose d'une méthode *add (element)*, indépendante d'un quelconque itérateur, qui ajoute un élément à la collection. Son emplacement exact dépendra de la nature de la collection : en fin de collection pour une liste ou un vecteur ; à un emplacement sans importance pour un ensemble.

Par exemple, quelle que soit la collection *c<String>* (ou *c*, avant le JDK 5.0), les instructions suivantes y introduiront les objets de type *String* du tableau *t* :

```
String t[] = { "Java", "C++", "Basic", "JavaScript"} ;
   .....
for (String s : t) c.add(s) ;
                 // for (int i=0 ; i<t.length ; i++) c.add (t[i]) ;   <-- avant JDK 5.0
```

De même, les instructions suivantes introduiront dans une collection *c<Integer>* (ou *c* avant le JDK 5.0) les objets de type *Integer* obtenus à partir du tableau d'entiers *t* :

```
int t[] = {2, 5, -6, 2, -8, 9, 5} ;
   .....
for (int v : t) c.add (v) ;
     // for (int i=0 ; i<t.length ; i++) c.add (new Integer (t[i])) ;  <-- avant JDK 5.0
```

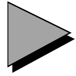
La méthode *add* fournit la valeur *true* lorsque l'ajout a pu être réalisé, ce qui sera le cas avec la plupart des collections, exception faite des ensembles ; dans ce cas, on obtient la valeur *false* si l'élément qu'on cherche à ajouter est déjà "présent" dans l'ensemble (c'est-à-dire s'il existe un élément qui lui soit égal au sens défini au paragraphe 1.3).

De la même façon, toute collection dispose d'une méthode *remove (element)* qui recherche un élément de valeur donnée (paragraphe 1.3) et le supprime s'il existe en fournissant alors la valeur *true*. Cette opération aura surtout un intérêt dans les cas des ensembles où elle possède une efficacité en *O(1)* ou en *O(Log N)*. Dans les autres cas, elle devra parcourir tout ou partie de la collection avec, donc, une efficacité moyenne en *O(N)*.

1.6.4 Opérations collectives

Toute collection *c* dispose des méthodes suivantes recevant en argument une autre collection *ca* :

- *addAll (ca)* : ajoute à la collection *c* tous les éléments de la collection *ca*,
- *removeAll (ca)* : supprime de la collection *c* tout élément apparaissant égal (paragraphe 1.3) à un des éléments de la collection *ca*,
- *retainAll (ca)* : supprime de la collection *c* tout élément qui n'apparaît pas égal (paragraphe 1.3) à un des éléments de la collection *ca* (on ne conserve donc dans *c* que les éléments présents dans *ca*).



Remarque

Là encore, comme pour la construction d'une collection à partir d'une autre, aucune restriction ne pesait sur les types des éléments de *c* et de *ca* avant le JDK 5.0. Depuis le JDK 5.0, il est nécessaire que le type des éléments de *ca* soit compatible (identique ou dérivé) avec celui de *c*. Ainsi, comme on pourra le voir en Annexe G, l'en-tête de *addAll* pour une collection donnée sera de la forme :

```
addAll (Collection <? extends E> c)
```

1.6.5 Autres méthodes

La méthode *size* fournit la taille d'une collection, c'est-à-dire son nombre d'éléments tandis que la méthode *isEmpty* teste si elle est vide ou non. La méthode *clear* supprime tous les éléments d'une collection.

La méthode *contains (elem)* permet de savoir si la collection contient un élément de valeur égale (paragraphe 1.3) à *elem*. Là encore, cette méthode sera surtout intéressante pour les ensembles où elle possède une efficacité en *O(1)* (pour *HashSet*) ou en *O(Log N)* (pour *TreeSet*).

La méthode *toString* est redéfinie dans les collections de manière à fournir une chaîne représentant au mieux le contenu de la collection. Plus précisément, cette méthode *toString* fait appel à la méthode *toString* de chacun des éléments de la collection. Dans ces conditions, lorsque l'on a affaire à des éléments d'un type *String* ou enveloppe, le résultat reflète bien la valeur effective des éléments. Ainsi, dans les exemples du paragraphe 1.6.3, la méthode *toString* fournirait les chaînes :

```
Java C++ Basic JavaScript
2 5 -6 2 -8 9 5
```

Dans les autres cas, si *toString* n'a pas été redéfinie, on n'obtiendra que des informations liées à l'adresse des objets, ce qui généralement présentera moins d'intérêt.

N'oubliez pas que *toString* se trouve automatiquement appelée dans une instruction *print* ou *println*. Vous pourrez exploiter cette possibilité pour faciliter la mise au point de vos programmes en affichant très simplement le contenu de toute une collection. Ainsi, toujours avec nos exemples du paragraphe 1.6.3, l'instruction

```
println ("Collection = " + c) ;
```

affichera :

```
Collection = Java C++ Basic JavaScript
Collection = 2 5 -6 2 -8 9 5
```

Enfin, deux méthodes nommées *toArray* permettent de créer un tableau (usuel) d'objets à partir d'une collection.

1.7 Structure générale des collections

Nous venons de voir que toutes les collections implémentent l'interface *Collection* et nous en avons étudié les principales fonctionnalités. Vous pourrez généralement vous contenter de connaître les fonctionnalités supplémentaires qu'offrent chacune des classes *LinkedList*, *ArrayList*, *Vector*, *HashSet, TreeSet, PriorityQueue* et *ArrayDeque*. Mais, dans certains cas, vous devrez avoir quelques notions sur l'architecture d'interfaces employée par les concepteurs de la bibliothèque. Elle se présente comme suit :

Collection

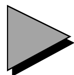
List	implémentée par *LinkedList*, *ArrayList* et *Vector*
Set	implémentée par *HashSet*
SortedSet	implémentée par *TreeSet*
NavigableSet	implémentée par *TreeSet (Java 6)*
Queue (*JDK 5.0)*	implémentée par *LinkedList, PriorityQueue*
Deque (Java 6)	implémentée par *ArrayDeque*, *LinkedList*

Vous verrez que le polymorphisme d'interfaces sera utilisé dans certains algorithmes. Par exemple, à un argument de type *List* pourra correspondre n'importe quelle classe implémentant l'interface *List*, donc *LinkedList*, *ArrayList* ou *Vector* mais aussi une classe que vous aurez créée.

Dans certains cas, vous pourrez utiliser ou rencontrer des instructions exploitant ce polymorphisme d'interfaces :

```
List<E> l ;                                          //  List l ;   <-- avant JDK 5.0
    /* l pourra être n'importe quelle collection */
    /* implémentant l'interface List */
Collection c1<E> ;                            // Collection c1 ;    <-- avant JDK 5.0
c1 = new LinkedList<E> () ;              // c1 = new LinkedList () ; <-- avant JDK 5.0
   /* OK mais on ne pourra appliquer à c1 */
   /* que les méthodes prévues dans l'interface Collection */
```

Une telle démarche présente l'avantage de spécifier le comportement général d'une collection, sans avoir besoin de choisir immédiatement son implémentation effective. Mais elle a quand même des limites, dans la mesure où certaines implémentations d'une interface donnée disposent de méthodes qui leur sont spécifiques et qui ne sont donc pas prévues dans l'interface qu'elles implémentent. Il n'est alors pas possible d'y faire appel.



Remarque

Les méthodes de la plupart des collections sont "non synchronisées", ce qui leur confère une certaine efficacité. En contrepartie, il n'est pas possible de les utiliser telles quelles dans des threads qui effectueraient des accès "concurrents" à une même collection. Mais, il est possible de définir ce que l'on nomme des "enveloppes synchronisées" que nous étudierons à la fin de ce chapitre. D'autre part, il existe en fait d'autres collections synchronisées figurant dans le paquetage *java.util.concurrent* que nous n'étudierons pas ici.

2 Les listes chaînées - classe LinkedList

2.1 Généralités

La classe *LinkedList* permet de manipuler des listes dites "doublement chaînées". À chaque élément de la collection, on associe (de façon totalement transparente pour le programmeur) deux informations supplémentaires qui ne sont autres que les références à l'élément précédent et au suivant[1]. Une telle collection peut ainsi être parcourue à l'aide d'un itérateur bidirectionnel de type *ListIterator* (présenté au paragraphe 1.4).

Le grand avantage d'une telle structure est de permettre des ajouts ou des suppressions à une position donnée avec une efficacité en *O(1)* (ceci grâce à un simple jeu de modification de références).

En revanche, l'accès à un élément en fonction de sa valeur ou de sa position dans la liste sera peu efficace puisqu'il nécessitera obligatoirement de parcourir une partie de la liste. L'efficacité sera donc en moyenne en *O(N)*.

2.2 Opérations usuelles

Construction et parcours

Comme toute collection, une liste peut être construite vide ou à partir d'une autre collection *c* :

```
 /* création d'une liste vide */
LinkedList<E> l1 = new LinkedList<E> () ;
                              // LinkedList l1=new LinkedList() ;   <-- avant JDK 5.0
 /* création d'une liste formée de tous les éléments de la collection c */
LinkedList<E> l2 = new LinkedList<E> (c) ;
                          // LinkedList l2 = new LinkedList (c) ;   <-- avant JDK 5.0
```

D'autre part, comme nous l'avons déjà vu, la méthode *listIterator* fournit un itérateur bidirectionnel doté des méthodes *next*, *previous*, *hasNext*, *hasPrevious*, *remove*, *add* et *set* décrites au paragraphe 1.4.2. En outre, la classe *LinkedList* dispose des méthodes spécifiques *getFirst* et *getLast* fournissant respectivement le premier ou le dernier élément de la liste.

Ajout d'un élément

La méthode *add* de *ListIterator* (ne la confondez pas avec celle de *Collection*) permet d'ajouter un élément à la position courante, avec une efficacité en *O(1)* :

1. En toute rigueur, les éléments d'une telle liste sont, non plus simplement les références aux objets correspondants, mais des objets appelés souvent nœuds formés de trois références : la référence à l'objet, la référence au nœud précédent et la référence au nœud suivant. En pratique, nous n'aurons pas à nous préoccuper de cela.

```
LinkedList <E> l ;                              // LinkedList l ;  <-- avant JDK 5.0
   .....
ListIterator <E> iter ;                      // ListIterator iter ;  <-- avant JDK 5.0
iter = l.listIterator () ;  /* iter désigne initialement le début de la liste */
  /* actions éventuelles sur l'itérateur (next et/ou previous) */
iter.add (elem) ;   /* ajoute l'élément elem à la position courante */
```

Rappelons que la "position courante" utilisée par *add* est toujours définie :

- si la liste est vide ou si l'on n'a pas agit sur l'itérateur, l'ajout se fera en début de liste,
- si *hasNext* vaut *false*, l'ajout se fera en fin de liste.

On notera bien que l'efficacité en *O(1)* n'est effective que si l'itérateur est convenablement positionné. Si l'on cherche par exemple à ajouter un élément en *nième* position, alors que l'itérateur est "ailleurs", il faudra probablement parcourir une partie de la liste (*k* éléments) pour que l'opération devienne possible. Dans ce cas, l'efficacité ne sera plus qu'en O(*k*).

La classe *LinkedList* dispose de méthodes spécifiques aux listes *addFirst* et *addLast* qui ajoutent un élément en début ou en fin de liste avec une efficacité en *O(1)*.

Bien entendu, la méthode *add* prévue dans l'interface *Collection* reste utilisable. Elle se contente d'ajouter l'élément en fin de liste, indépendamment d'un quelconque itérateur, avec une efficacité en *O(1)*.

N'oubliez pas qu'il ne faut pas modifier la collection pendant qu'on utilise l'itérateur (voir paragraphe 1.4.3). Ainsi le code suivant est à éviter :

```
while (iter.hasNex())
{ ...
  if (...) l.add (elem) ;      // déconseillé : itérateur en cours d'utilisation
      else l.addFirst (elem) ; // idem
  iter.next() ;
}
```

Suppression d'un élément

Nous avons déjà vu que la méthode *remove* de *ListIterator* supprime le dernier élément renvoyé soit par *next*, soit par *previous*. Son efficacité est en *O(1)*.

La classe *LinkedList* dispose en outre de méthodes spécifiques *removeFirst* et *removeLast* qui suppriment le premier ou le dernier élément de la liste avec une efficacité en *O(1)*.

On peut aussi, comme pour toute collection, supprimer d'une liste un élément de valeur donnée (au sens du paragraphe 1.3) avec *remove (element)*. Cette fois l'efficacité sera en *O(i), i* étant le rang de l'élément correspondant dans la liste (donc en moyenne en *O(N)*). En effet, étant donné qu'il n'existe aucun ordre lié aux valeurs, il est nécessaire d'explorer la liste depuis son début jusqu'à la rencontre éventuelle de l'élément. La méthode *remove* fournit *false* si la valeur cherchée n'a pas été trouvée.

2.3 Exemples

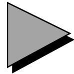
Exemple 1

Voici un exemple de programme manipulant une liste de chaînes (*String*) qui illustre les principales fonctionnalités de la classe *ListIterator*. Elle contient une méthode *affiche*, statique, affichant le contenu d'une liste reçue en argument.

```
import java.util.* ;
public class Liste1
{ public static void main (String args[])
  { LinkedList<String> l = new LinkedList<String>() ;
                              //  LinkedList l = new LinkedList() ;  <-- avant JDK 5.0
    System.out.print ("Liste en A : ") ; affiche (l) ;
    l.add ("a") ; l.add ("b") ;  // ajouts en fin de liste
    System.out.print ("Liste en B : ") ; affiche (l) ;

    ListIterator<String> it = l.listIterator() ;
                              //  LinkedList l = new LinkedList() ;  <-- avant JDK 5.0
    it.next() ; // on se place sur le premier element
    it.add ("c") ; it.add ("b") ;  // et on ajoute deux elements
    System.out.print ("Liste en C : ") ; affiche (l) ;

    it = l.listIterator() ;
    it.next() ; // on progresse d'un element
    it.add ("b") ; it.add ("d") ;     // et on ajoute deux elements
    System.out.print ("Liste en D : ") ; affiche (l) ;

    it = l.listIterator (l.size()) ;   // on se place en fin de liste
    while (it.hasPrevious())     // on recherche le dernier b
    { String ch = it.previous() ;
                          //  String ch = (String) it.previous() ;   <-- avant JDK 5.0
      if (ch.equals ("b"))
      { it.remove() ;  // et on le supprime
        break ;
      }
    }
    System.out.print ("Liste en E : ") ; affiche (l) ;

    it = l.listIterator() ;
    it.next() ; it.next() ;   // on se place sur le deuxieme element
    it.set ("x") ;            // qu'on remplace par "x"
    System.out.print ("Liste en F : ") ; affiche (l) ;
  }
  public static void affiche (LinkedList<String> l)
                      // public static void affiche (LinkedList l)  <-- avant JDK 5.0
  { ListIterator<String> iter = l.listIterator ();
                          // ListIterator iter = l.listIterator ();  <-- avant JDK 5.0
    while (iter.hasNext()) System.out.print (iter.next() + " ") ;
    System.out.println () ;
  }
}
```

```
Liste en A :
Liste en B : a b
Liste en C : a c b b
Liste en D : a b d c b b
Liste en E : a b d c b
Liste en F : a x d c b
```

*Utilisation d'une liste de chaînes (*String*)*



Remarques

1 Ici, nous aurions pu nous passer de la méthode *affiche* en utilisant implicitement la méthode *toString* de la classe *LinkedList* dans un appel de *print*, comme par exemple :

```
System.out.println ("Liste en E : " + l) ;
```

au lieu de :

```
System.out.print ("Liste en E : ") ; affiche (l) ;
```

L'affichage aurait été presque le même :

```
Liste en E : [a b d c b]
```

2 On pourrait penser à remplacer :

```
it = l.listIterator() ;
it.next() ; it.next() ;   // on se place sur le deuxieme element
it.set ("x") ;            // qu'on remplace par "x"
```

par :

```
it = l.listIterator(2) ;
it.set ("x") ;            // erreur, il n'y a plus d'element courant
```

Cela n'est pas possible car après l'initialisation de *it*, l'élément courant n'est pas défini. En revanche (bien que cela ait peu d'intérêt ici), on pourrait procéder ainsi :

```
it = l.listIterator(1) ;
next () ;
it.set ("x") ;
```

Exemple 2

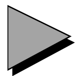
Voici un exemple montrant comment utiliser une liste chaînée pour afficher à l'envers une suite de chaînes lues au clavier.

```
import java.util.* ;
public class Liste2
{ public static void main (String args[])
  { LinkedList<String> l = new LinkedList<String>() ;
                              // LinkedList l = new LinkedList() ;   <-- avant JDK 5.0
```

```
    /* on ajoute a la liste tous les mots lus au clavier */
    System.out.println ("Donnez une suite de mots (vide pour finir)") ;
    while (true)
    { String ch = Clavier.lireString() ;
      if (ch.length() == 0) break ;
      l.add (ch) ;
    }
    System.out.println ("Liste des mots a l'endroit :") ;
    ListIterator<String> iter = l.listIterator() ;
                         // ListIterator iter = l.listIterator() ;   <-- avant JDK 5.0
    while (iter.hasNext()) System.out.print (iter.next() + " ") ;
    System.out.println () ;
    System.out.println ("Liste des mots a l'envers :") ;
    iter = l.listIterator(l.size()) ;  // iterateur en fin de liste
    while (iter.hasPrevious()) System.out.print (iter.previous() + " ") ;
    System.out.println () ;
  }
}
```

```
Donnez une suite de mots (vide pour finir)
Java
C++
Basic
JavaScript
Pascal

Liste des mots a l'endroit :
Java C++ Basic JavaScript Pascal
Liste des mots a l'envers :
Pascal JavaScript Basic C++ Java
```

Inversion de mots



Remarque

Ici, il n'est pas possible d'utiliser la méthode *toString* de la liste pour l'afficher à l'envers.

2.4 Autres possibilités peu courantes

L'interface *List* (implémentée par *LinkedList*, mais aussi par *ArrayList*) dispose encore d'autres méthodes permettant de manipuler les éléments d'une liste à la manière de ceux d'un vecteur, c'est-à-dire à partir de leur rang *i* dans la liste. Mais alors que l'efficacité de ces méthodes est en *O(1)* pour les vecteurs dynamiques, elle n'est qu'en *O(N)* en moyenne pour les listes ; elles sont donc généralement peu utilisées. Vous en trouverez la description à l'annexe G. À simple titre indicatif, mentionnons que vous pourrez :

- supprimer le *nième* élément par *remove (i)*,
- obtenir la valeur du *nième* élément par *get(i)*,

- modifier la valeur du *nième* élément par *set (i, elem)*.

De même, il existe d'autres méthodes basées sur une position, toujours en *O(N)* (mais, cette fois, elles ne font pas mieux avec les vecteurs !) :

- ajouter un élément en position *i* par *add(i, elem)*,
- obtenir le rang du premier ou du dernier élément de valeur (*equals*) donnée par *indexOf (elem)* ou *lastIndexOf (elem)*,
- ajouter tous les éléments d'une autre collection *c* à un emplacement donné par *addAll (i, c).*

2.5 Méthodes introduites par Java 5 et Java 6

Depuis le JDK 5.0, la classe *LinkedList* implémente également l'interface *Queue*. On y trouve alors une méthode d'ajout (*offer*) qui, contrairement à *add,* ne déclenche pas d'exception en cas de pile pleine. On y trouve également des méthodes de consultation destructive (*poll*) ou non destructive (*peek*), qui font double emploi avec *getFirst* et *removeFirst* (qui appartiennent à la classe *LinkedList*, mais pas à l'interface *List*).

Depuis Java 6, la classe *LinkedList* implémente en outre l'interface *Deque* (queue à double entrée) présentée plus loin. Elle se trouve alors dotée d'un jeu complet de méthodes d'action soit en début, soit en fin de liste (ajout, consultation destructive ou non), avec, en plus, la possibilité de choisir le comportement en cas d'anomalie (soit exception, soit valeur de retour particulière). Là encore, ces méthodes font double emploi avec *getFirst*, *getLast*, *removeFirst* et *removeLast* (qui appartiennent à la classe *LinkedList*, mais pas à l'interface *List*).

Les interfaces *Queue* et *Deque* sont présentées aux paragraphes 5 et 6.

3 Les vecteurs dynamiques - classe ArrayList

3.1 Généralités

La classe *ArrayList* offre des fonctionnalités d'accès rapide comparables à celles d'un tableau d'objets. Bien qu'elle implémente, comme *LinkedList*, l'interface *List*, sa mise en œuvre est différente et prévue pour permettre des accès efficaces à un élément de rang donné, c'est-à-dire en *O(1)* (on parle parfois d'accès direct à un élément comme dans le cas d'un tableau).

En outre, cette classe offre plus de souplesse que les tableaux d'objets dans la mesure où sa taille (son nombre d'éléments) peut varier au fil de l'exécution (comme celle de n'importe quelle collection).

Mais pour que l'accès direct à un élément de rang donné soit possible, il est nécessaire que les emplacements des objets (plutôt de leurs références) soient contigus en mémoire (à la manière de ceux d'un tableau). Aussi cette classe souffrira d'une lacune inhérente à sa nature : l'addition ou la suppression d'un objet à une position donnée ne pourra plus se faire

en *O(1)* comme dans le cas d'une liste[1], mais seulement en moyenne en *O(N)*. En définitive, les vecteurs seront bien adaptés à l'accès direct à condition que les additions et les suppressions restent limitées.

Par ailleurs, cette classe dispose de méthodes permettant de contrôler l'emplacement alloué à un vecteur à un instant donné.

3.2 Opérations usuelles

Construction

Comme toute collection, un vecteur dynamique peut être construit vide ou à partir d'une autre collection *c* :

```
  /* vecteur dynamique vide */
ArrayList <E> v1 = new ArrayList <E> () ;
                               // ArrayList v1 = new ArrayList () ;  <-- avant JDK 5.0
  /* vecteur dynamique contenant tous les éléments de la collection c */
ArrayList <E> v2 = new ArrayList <E>(c) ;
                             // ArrayList v2 = new ArrayList (c) ;   <-- avant JDK 5.0
```

Ajout d'un élément

Comme toute collection, les vecteurs disposent de la méthode *add (elem)* qui se contente d'ajouter l'élément *elem* en fin de vecteur avec une efficacité en *O(1)*.

On peut aussi ajouter un élément *elem* en un rang *i* donné à l'aide de la méthode *add (i,elem)*

Dans ce cas, le nouvel élément prend la place du *nième*, ce dernier et tous ses suivants étant décalés d'une position. L'efficacité de la méthode est donc en O(*N-i*), soit en moyenne en *O(N)*.

Suppression d'un élément

La classe *ArrayList* dispose d'une méthode spécifique *remove* permettant de supprimer un élément de rang donné (le premier élément est de rang 0, comme dans un tableau usuel). Elle fournit en retour l'élément supprimé. Là encore, les éléments suivants doivent être décalés d'une position. L'efficacité de la méthode est donc en O(*N-i*) ou *O(N)* en moyenne.

```
ArrayList <E> v ;                                // ArrayList v ;   <-- avant JDK 5.0
   .....
  /* suppression du troisième élément de v  qu'on obtient dans o */
E o = v.remove (3) ;                   // Object o = v.remove (3) ;  <-- avant JDK 5.0
```

On ne confondra pas cette méthode *remove* de *ArrayList* avec la méthode *remove* d'un itérateur (rarement utilisé avec les vecteurs).

1. Dans le cas de l'insertion à la position courante car l'insertion en un rang donné aurait elle aussi une efficacité moyenne en *O(N)*.

On peut aussi (comme pour toute collection) supprimer d'un vecteur un élément de valeur donnée *elem* (comme défini au paragraphe 1.3) avec *remove (elem)*. L'efficacité est en *O(i)* (*i* étant le rang de l'élément correspondant dans le vecteur), donc là encore en moyenne en *O(N)*. En effet, étant donné qu'il n'existe aucun ordre lié aux valeurs, il est nécessaire d'explorer le vecteur depuis son début jusqu'à l'éventuelle rencontre de l'élément. La méthode *remove* fournit *false* si la valeur cherchée n'a pas été trouvée.

La classe *ArrayList* possède une méthode spécifique *removeRange* permettant de supprimer plusieurs éléments consécutifs (de *n* à *p*) :

```
ArrayList <E> v ;                                 // ArrayList v ;   <-- avant JDK 5.0
   .....
v.removeRange (3, 8) ;   // supprime les éléments de rang 3 à 8 de v
```

Accès aux éléments

Ce sont précisément les méthodes d'accès ou de modification d'un élément en fonction de sa position qui font tout l'intérêt des vecteurs dynamiques puisque leur efficacité est en *O(1)*.

On peut connaître la valeur d'un élément de rang *i* par *get(i)*. Généralement, pour parcourir tous les éléments de type *E* d'un vecteur *v*, on procédera ainsi :

```
  // depuis le JDK 5.0                 // avant JDK 5.0
for (E e : v)                      for (int i=0 ; i<v.size() ; i++)
{  // utilisation de e             { // utilisation de v.get(i)
}                                  }
```

Voici par exemple une méthode statique recevant un argument de type *ArrayList* et en affichant tous les éléments, d'abord dans une version générique (depuis le JDK 5.0) :

```
public <E> static void affiche (ArrayList <E> v)
{ for (E e : v)
     System.out.print (e + " ") ;
  System.out.println () ;
}
```

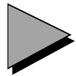
ou dans une version antérieure au JDK 5.0 :

```
public static void affiche (ArrayList v)
{ for (int i = 0 ; i<v.size() ; i++)
     System.out.print (v.get(i) + " ") ;
  System.out.println () ;
}
```

On peut remplacer par *elem* la valeur de l'élément de rang *i* par *set (i, elem)*. Voici par exemple comment remplacer par la référence *null* tout élément d'un vecteur dynamique *v* vérifiant une *condition* :

```
for (int i = 0 ; i<v.size() ; i++)    // for... each pas utilisable ici
  if (condition) set (i, null) ;
```

Comme d'habitude, la méthode *set* fournit la valeur de l'élément avant modification.



Remarque

La classe *ArrayList* implémente elle aussi l'interface *List* et, à ce titre, dispose d'un itérateur bidirectionnel qu'on peut théoriquement utiliser pour parcourir les éléments d'un vecteur. Toutefois, cette possibilité fait double emploi avec l'accès direct par *get* ou *set* auquel on recourra le plus souvent. En revanche, elle sera implicitement utilisée lors d'un parcours par la boucle *for... each*.

3.3 Exemple

Voici un programme créant un vecteur contenant dix objets de type *Integer*, illustrant les principales fonctionnalités de la classe *ArrayList* :

```
import java.util.* ;
public class Array1
{ public static void main (String args[])
  { ArrayList <Integer> v = new ArrayList <Integer> () ;
                                // ArrayList v = new ArrayList () ;  <-- avant JDK 5.0
    System.out.println ("En A : taille de v = " + v.size() ) ;

      /* on ajoute 10 objets de type Integer */
    for (int i=0 ; i<10 ; i++) v.add (new Integer(i)) ;
    System.out.println ("En B : taille de v = " + v.size() ) ;

      /* affichage du contenu, par acces direct (get) a chaque element */
    System.out.println ("En B : contenu de v = ") ;
    for (Integer e : v)        // for (int i = 0 ; i<v.size() ; i++)  <-- avant JDK 5.0
       System.out.print (e + " ") ;           // System.out.print (v.get(i)+" ") ; <--
    System.out.println () ;

      /* suppression des elements de position donnee */
    v.remove (3) ;
    v.remove (5) ;
    v.remove (5) ;
    System.out.println ("En C : contenu de v = " + v) ;

      /* ajout d'elements a une position donnee */
    v.add (2, new Integer (100)) ;
    v.add (2, new Integer (200)) ;
    System.out.println ("En D : contenu de v = " + v) ;

      /* modification d'elements de position donnee */
    v.set (2, new Integer (1000)) ;  // modification element de rang 2
    v.set (5, new Integer (2000)) ;  // modification element de rang 5
    System.out.println ("En D : contenu de v = " + v) ;
  }
}
```

```
En A : taille de v = 0
En B : taille de v = 10
En B : contenu de v =
0 1 2 3 4 5 6 7 8 9
En C : contenu de v = [0, 1, 2, 4, 5, 8, 9]
En D : contenu de v = [0, 1, 200, 100, 2, 4, 5, 8, 9]
En D : contenu de v = [0, 1, 1000, 100, 2, 2000, 5, 8, 9]
```

Utilisation d'un vecteur dynamique d'éléments de type Integer

3.4 Gestion de l'emplacement d'un vecteur

Lors de sa construction, un objet de type *ArrayList* dispose d'une "capacité" *c*, c'est-à-dire d'un nombre d'emplacements mémoire contigus permettant d'y stocker *c* éléments. Cette capacité peut être définie lors de l'appel d'un constructeur ou fixée par défaut. Elle ne doit pas être confondue avec le nombre d'éléments du vecteur, lequel est initialement nul.

Au fil de l'exécution, il peut s'avérer que la capacité devienne insuffisante. Dans ce cas, une nouvelle allocation mémoire est faite avec une capacité incrémentée d'une quantité fixée à la construction ou doublée si rien d'autre n'est spécifié. Lors de l'accroissement de la capacité, il se peut que les nouveaux emplacements nécessaires ne puissent pas être alloués de façon contiguë aux emplacements existants (il en ira souvent ainsi). Dans ce cas, tous les éléments du tableau devront être recopiés dans un nouvel emplacement, ce qui prendra un certain temps.

Par ailleurs, on disposera de méthodes permettant d'ajuster la capacité au fil de l'exécution :

- *ensureCapacity (capaciteMini)* : demande d'allouer au vecteur une capacité au moins égale à *capaciteMini* ; si la capacité est déjà supérieure, l'appel n'aura aucun effet ;
- *trimToSize()* : demande de ramener la capacité du vecteur à sa taille actuelle en libérant les emplacements mémoire non utilisés ; ceci peut s'avérer intéressant lorsqu'on sait que la taille de la collection ne changera plus par la suite.

3.5 Autres possibilités peu usuelles

La classe *ArrayList* dispose encore de quelques méthodes peu usitées. Vous en trouverez la liste en annexe G.

Par ailleurs, comme *ArrayList* implémente l'interface *List*, elle dispose encore de quelques méthodes permettant :

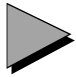
- d'obtenir le rang du premier ou du dernier élément de valeur donnée (voir paragraphe 1.3) par *indexOf (elem)* ou *lastIndexOf (elem)*,
- d'ajouter tous les éléments d'une autre collection *c* à un emplacement donné *i* par *addAll (i, c)*.

3.6 L'ancienne classe Vector

Dans les versions antérieures à la version 2.0, Java ne disposait pas des collections que nous venons de décrire ici. En revanche, on y trouvait une classe nommée *Vector* permettant, comme *ArrayList* de manipuler des vecteurs dynamiques. Elle a été remaniée dans la version 2 de Java, de façon à implémenter l'interface *List* ; elle peut donc être utilisée comme une collection (vous trouverez la liste des méthodes de *Vector* en annexe G).

Comme nous l'avons dit, la plupart des collections sont "non synchronisées", autrement dit leurs méthodes n'ont jamais le qualificatif *synchronized*. Deux threads différents ne peuvent donc pas accéder sans risque à une même collection. En revanche, la classe *Vector* est "synchronisée". Cela signifie que deux threads différents peuvent accéder au même vecteur, mais au prix de temps d'exécution plus longs qu'avec la classe *ArrayList*.

Mais nous verrons qu'il est possible de définir des "enveloppes synchronisées" des collections, donc en particulier d'un vecteur *ArrayList* qui jouera alors le même rôle que *Vector*. De plus, le paquetage *java.util.concurent* propose des collections synchronisées (que nous n'étudierons pas ici). En définitive, il est utile de connaître cette classe *Vector* car elle a été fort utilisée dans d'anciens codes...



Remarques

1 La classe *Vector* dispose d'une méthode *capacity* qui fournit la capacité courante d'une collection ; curieusement, *ArrayList* ne possède pas de méthode équivalente.

2 Les versions antérieures de Java disposaient encore d'autres classes qui restent toujours utilisables mais dont nous ne parlons pas dans cet ouvrage. Citons : *Enumeration* (jouant le même rôle que les itérateurs), *Stack* (pile), *HashTable* (jouant le même rôle que *HashMap* étudié plus loin).

4 Les ensembles

4.1 Généralités

Deux classes implémentent la notion d'ensemble : *HashSet* et *TreeSet*. Rappelons que, théoriquement, un ensemble est une collection non ordonnée d'éléments, aucun élément ne pouvant apparaître plusieurs fois dans un même ensemble. Chaque fois qu'on introduit un nouvel élément dans une collection de type *HashSet* ou *TreeSet*, il est donc nécessaire de s'assurer qu'il n'y figure pas déjà, autrement dit que l'ensemble ne contient pas un autre élément qui lui soit égal (au sens défini au paragraphe 1.3). Nous avons vu que dès que l'on s'écarte d'éléments de type *String*, *File* ou enveloppe, il est généralement nécessaire de se préoccuper des méthodes *equals* ou *compareTo* (ou d'un comparateur) ; si on ne le fait pas, il faut accepter que deux objets de références différentes ne soient jamais identiques, quelles que soient leurs valeurs !

Par ailleurs, bien qu'en théorie un ensemble ne soit pas ordonné, des raisons évidentes d'efficacité des méthodes de test d'appartenance nécessitent une certaine organisation de l'information. Dans le cas contraire, un tel test d'appartenance ne pourrait se faire qu'en examinant un à un les éléments de l'ensemble (ce qui conduirait à une efficacité moyenne en *O(N)*). Deux démarches différentes ont été employées par les concepteurs des collections, d'où l'existence de deux classes différentes :

- *HashSet* qui recourt à une technique dite de *hachage*, ce qui conduit à une efficacité du test d'appartenance en *O(1)*,
- *TreeSet* qui utilise un arbre binaire pour ordonner complètement les éléments, ce qui conduit à une efficacité du test d'appartenance en *O(Log N)*.

En définitive, dans les deux cas, les éléments seront ordonnés même si cet ordre est moins facile à appréhender dans le premier cas que dans le second (avec *TreeSet*, il s'agit de l'ordre induit par *compareTo* ou un éventuel comparateur).

Dans un premier temps, nous présenterons les fonctionnalités des ensembles dans des situations où leurs éléments sont d'un type qui ne nécessite pas de se préoccuper de ces détails d'implémentation (*String* ou enveloppes).

Dans un deuxième temps, nous serons amenés à vous présenter succinctement les structures réellement utilisées afin de vous montrer les contraintes que vous devrez respecter pour induire un ordre convenable, à savoir :

- définir les méthodes *hashCode* et *equals* des éléments avec la classe *HashSet* (*equals* étant aussi utilisée pour le test d'appartenance),
- définir la méthode *compareTo* des éléments (ou un comparateur) avec la classe *TreeSet* (cette fois, c'est cet ordre qui sera utilisé pour le test d'appartenance).

4.2 Opérations usuelles

Construction et parcours

Comme toute collection, un ensemble peut être construit vide ou à partir d'une autre collection :

```
  // ensemble vide
HashSet<E> e1 = new HashSet<E> () ;  // HashSet e1 = new HashSet() ;  <-- avant JDK 5.0
  // ensemble contenant tous les éléments de la collection c
HashSet<E> e2 = new HashSet<E>(c) ;  // HashSet e2 = new HashSet(c) ; <-- avant JDK 5.0
  // ensemble vide
TreeSet<E> e3 = new TreeSet<E>() ;   // TreeSet e3 = new TreeSet() ;  <-- avant JDK 5.0
   // ensemble contenant tous les éléments de la collection c
TreeSet<E> e4 = new TreeSet<E>(c) ;  // TreeSet e4 = new TreeSet(c) ; <-- avant JDK 5.0
```

Les deux classes *HashSet* et *TreeSet* disposent de la méthode *iterator* prévue dans l'interface *Collection*. Elle fournit un itérateur monodirectionnel (*Iterator*) permettant de parcourir les différents éléments de la collection :

```
HashSet<E> e ; // ou TreeSet<E> e      // HashSet e ; ou TreeSet e ;   <-- avant JDK 5.0
   .....
Iterator<E> it = e.iterator() ;         // Iterator it = e.iterator() ; <-- avant JDK
5.0
while (it.hasNext())
{ E o = it.next() ;                    // Object o = it.next() ;       <-- avant JDK 5.0
  // utilisation de o
}
```

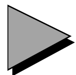
Ajout d'un élément

Rappelons qu'il est impossible d'ajouter un élément à une position donnée puisque les ensembles ne disposent pas d'un itérateur bidirectionnel (d'ailleurs, comme au niveau de leur implémentation, les ensembles sont organisés en fonction des valeurs de leurs éléments, l'opération ne serait pas réalisable).

La seule façon d'ajouter un élément à un ensemble est d'utiliser la méthode *add* prévue dans l'interface *Collection*. Elle s'assure en effet que l'élément en question n'existe pas déjà :

```
HashSet<E> e ; E elem ;            // Hashset e ; Object elem ;      <-- avant JDK 5.0
   .....
boolean existe = e.add (elem) ;
if (existe) System.out.println (elem + " existe deja") ;
      else System.out.println (elem + " a ete ajoute") ;
```

Rappelons que grâce aux techniques utilisées pour implémenter l'ensemble, l'efficacité du test d'appartenance est en *O(1)* pour le type *HashSet* et en *O(Log N)* pour le type *TreeSet*.



Remarque

On ne peut pas dire que *add* ajoute l'élément en "fin d'ensemble" (comme dans le cas des collections étudiées précédemment). En effet, comme nous l'avons déjà évoqué, les ensembles sont organisés au niveau de leur implémentation, de sorte que l'élément devra quand même être ajouté à un endroit bien précis de l'ensemble (endroit qui se concrétisera lorsqu'on utilisera un itérateur). Pour l'instant, on devine déjà que cet endroit sera imposé par l'ordre induit par *compareTo* (ou un comparateur) dans le cas de *TreeSet* ; en ce qui concerne *HashSet*, nous le préciserons plus tard.

Suppression d'un élément

Nous avons vu que pour les autres collections, la méthode *remove* de suppression d'une valeur donnée possède une efficacité en *O(N)*. Un des grands avantages des ensembles est d'effectuer cette opération avec une efficacité en *O(1)* (pour *HashSet*) ou en *O(Log N)* (pour *TreeSet*). Ici, la méthode *remove* renvoie *true* si l'élément a été trouvé (et donc supprimé) et *false* dans le cas contraire :

```
TreeSet<E> e ; E o ;                    // TreeSet e ; Object o ;  <-- avant JDK 5.0
   .....
boolean trouve = e.remove (o) ;
if (trouve) System.out.println (o + " a ete supprime") ;
      else System.out.println (o + " n'existe pas ") ;
```

Par ailleurs, la méthode *remove* de l'itérateur monodirectionnel permet de supprimer l'élément courant (le dernier renvoyé par *next*) le cas échéant :

```
TreeSet<E> e ;                                        // TreeSet e ;   <-- avant JDK 5.0
   .....
Iterator<E> it = e.iterator () ;        // Iterator it = e.iterator() ; <-- avant JDK 
5.0
it.next () ; it.next () ;  /* deuxième élément = élément courant */
it.remove () ;             /* supprime le deuxième élément */
```

Enfin, la méthode *contains* permet de tester l'existence d'un élément, avec toujours une efficacité en *O(1)* ou en *O(Log N)*.

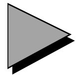
4.3 Exemple

Voici un exemple dans lequel on crée un ensemble d'éléments de type *Integer* qui utilise la plupart des possibilités évoquées précédemment. La méthode statique *affiche* est surtout destinée ici à illustrer l'emploi d'un itérateur car on aurait pu s'en passer en affichant directement le contenu d'un ensemble par *print* (avec toutefois, une présentation légèrement différente).

```
import java.util.* ;
public class Ens1
{ public static void main (String args[])
  { int t[] = {2, 5, -6, 2, -8, 9, 5} ;
    HashSet<Integer>ens = new HashSet<Integer>() ;
                                    // HashSet ens = new HashSet() ; <-- avant JDK 5.0
       /* on ajoute des objets de type Integer */
    for (int v : t)              // for (int i=0 ; i< t.length ; i++) <-- avant JDK 5.0
    { boolean ajoute = ens.add(v) ;
                  //  boolean ajoute = ens.add (new Integer (t[i])) ; <-- avant JDK 5.0
      if (ajoute) System.out.println ("  On ajoute " + v) ;
              // if(ajoute) System.out.println("  On ajoute "+t[i]) ; <-- avant JDK 5.0
        else System.out.println ("  " + v + " est deja present") ;
                  // else System.out.println ("  " + t[i] + " est deja present") ; <--
    }
    System.out.print ("Ensemble en A = ") ; affiche (ens) ;
       /* on supprime un eventuel objet de valeur Integer(5) */
    Integer cinq = 5 ;            // Integer cinq = new Integer (5) ; <-- avant JDK 5.0
    boolean enleve = ens.remove (cinq) ;
    if (enleve) System.out.println ("  On a supprime 5") ;
    System.out.print ("Ensemble en B = ") ; affiche (ens) ;
       /* on teste la presence de Integer(5) */
    boolean existe = ens.contains (cinq) ;
    if (!existe) System.out.println ("  On ne trouve pas 5") ;
  }
  public static <E> void affiche (HashSet<E> ens)
                        // public static void affiche (HashSet ens)  <-- avant JDK 5.0
  { Iterator<E> iter = ens.iterator () ;       // Iterator iter = ens.iterator () ; <--
    while (iter.hasNext())
    { System.out.print (iter.next() + " ") ;
    }
    System.out.println () ;
  }
}
```

```
  On ajoute 2
  On ajoute 5
  On ajoute -6
  2 est deja present
  On ajoute -8
  On ajoute 9
  5 est deja present
Ensemble en A = 2 9 -6 -8 5
  On a supprime 5
Ensemble en B = 2 9 -6 -8
  On ne trouve pas 5
```

Utilisation d'un ensemble d'éléments de type Integer



Remarque

Nous aurions pu employer le type *TreeSet* à la place du type *HashSet*. Le programme modifié dans ce sens figure sur le site Web d'accompagnement sous le nom *Ens1a.java*. Vous pourrez constater que le classement des éléments au sein de l'ensemble est différent (ils sont rangés par valeur croissante).

4.4 Opérations ensemblistes

Les méthodes *removeAll*, *addAll* et *retainAll*, applicables à toutes les collections, vont prendre un intérêt tout particulier avec les ensembles où elles vont bénéficier de l'efficacité de l'accès à une valeur donnée. Ainsi, si *e1* et *e2* sont deux ensembles :

- *e1.addAll(e2)* place dans *e1* tous les éléments présents dans *e2*. Après exécution, la réunion de *e1* et de *e2* se trouve dans *e1* (dont le contenu a généralement été modifié).
- *e1.retainAll (e2)* garde dans *e1* ce qui appartient à *e2*. Après exécution, on obtient l'intersection de *e1* et de *e2* dans *e1* (dont le contenu a généralement été modifié).
- *e1.removeAll (e2)* supprime de *e1* tout ce qui appartient à *e2*. Après exécution, on obtient le "complémentaire de *e2* par rapport à *e1*" dans *e1* (dont le contenu a généralement été modifié).

Exemple 1

Voici un exemple appliquant ces opérations ensemblistes à des ensembles d'éléments de type *Integer*. Notez que nous avons dû prévoir une méthode utilitaire (*copie*) de recopie d'un ensemble dans un autre (il existe un algorithme *copy* mais il ne s'applique qu'à des collections implémentant l'interface *List*).

```
import java.util.* ;
public class EnsOp
{ public static void main (String args[])
  { int t1[] = {2,    5, 6,    8, 9} ;
    int t2[] = {   3,    6, 7,    9} ;
    HashSet <Integer> e1 = new HashSet <Integer>(), e2 = new HashSet<Integer> () ;
            // HashSet e1 = new HashSet(), e2 = new HashSet() ;   <-- avant JDK 5.0
    for (int v : t1)  e1.add (v) ;
          //  for (int i=0 ; i< t1.length ; i++) e1.add (new Integer (t1[i])) ;
    for (int v : t2) e2.add (v) ;
          //  for (int i=0 ; i< t2.length ; i++) e2.add (new Integer (t2[i])) ;
    System.out.println ("e1 = " + e1) ; System.out.println ("e1 = " + e2) ;

    // reunion de e1 et e2 dans u1
    HashSet <Integer>  u1 = new HashSet <Integer> () ;
                                   // HashSet u1 = new HashSet () ; <-- avant JDK 5.0
    copie (u1, e1) ;  // copie e1 dans u1
    u1.addAll (e2) ;
    System.out.println ("u1 = " + u1) ;

    // intersection de e1 et e2 dans i1
    HashSet <Integer> i1 = new HashSet <Integer> () ;
                                   // HashSet i1 = new HashSet () ; <-- avant JDK 5.0
    copie (i1, e1) ;
    i1.retainAll (e2) ;
    System.out.println ("i1 = " + i1) ;
  }

  public static <E> void copie (HashSet<E> but, HashSet<E> source)
         // public static void copie (HashSet but, HashSet source) <-- avant JDK 5.0
  { Iterator<E> iter = source.iterator() ;
                           // Iterator iter = source.iterator () ;  <-- avant JDK 5.0
    while (iter.hasNext())
    { but.add (iter.next()) ;
    }
  }
}
```

```
e1 = [9, 8, 6, 5, 2]
e1 = [9, 7, 6, 3]
u1 = [9, 8, 7, 6, 5, 3, 2]
i1 = [9, 6]
```

Opérations ensemblistes

Exemple 2

Voici un second exemple montrant l'intérêt des opérations ensemblistes pour déterminer les lettres et les voyelles présentes dans un texte. Notez qu'ici nous avons considéré les lettres

comme des chaînes (*String*) de longueur 1 ; nous aurions pu également utiliser la classe enveloppe *Character*). On notera la présence de la lettre "espace".

```
import java.util.* ;
public class Ens2
{ public static void main (String args[])
  { String phrase = "je me figure ce zouave qui joue" ;
    String voy = "aeiouy" ;
    HashSet <String> lettres = new HashSet <String>() ;
                              // HashSet lettres = new HashSet() ;  <-- avant JDK 5.0
    for (int i=0 ; i<phrase.length() ; i++)
      lettres.add (phrase.substring(i, i+1)) ;
    System.out.println ("lettres presentes : " + lettres) ;

    HashSet <String> voyelles = new HashSet<String>() ;
                             // HashSet voyelles = new HashSet() ;  <-- avant JDK 5.0
    for (int i=0 ; i<voy.length() ; i++)
      voyelles.add (voy.substring (i, i+1)) ;
    lettres.removeAll (voyelles) ;
    System.out.println ("lettres sans les voyelles : " + lettres) ;
  }
}
```

```
lettres presentes : [c, a,  , z, v, u, r, q, o, m, j, i, g, f, e]
lettres sans les voyelles : [c,  , z, v, r, q, m, j, g, f]
```

Détermination des lettres présentes dans un texte

4.5 Les ensembles HashSet

Jusqu'ici, nous n'avons considéré que des ensembles dont les éléments étaient d'un type *String* ou enveloppe pour lesquels, comme nous l'avons dit en introduction, nous n'avions pas à nous préoccuper des détails d'implémentation.

Dès que l'on cherche à utiliser des éléments d'un autre type objet, il est nécessaire de connaître quelques conséquences de la manière dont les ensembles sont effectivement implémentés. Plus précisément, dans le cas des *HashSet*, vous devrez définir convenablement :

- la méthode *equals* : c'est toujours elle qui sert à définir l'appartenance d'un élément à l'ensemble,
- la méthode *hashCode* dont nous allons voir comment elle est exploitée pour ordonnancer les éléments d'un ensemble, ce qui va nous amener à parler de "table de hachage".

4.5.1 Notion de table de hachage

Une table de hachage est une organisation des éléments d'une collection qui permet de retrouver facilement un élément de valeur donnée[1]. Pour cela, on utilise une méthode (*hashCode*) dite "fonction de hachage" qui, à la valeur d'un élément (existant ou recherché), asso-

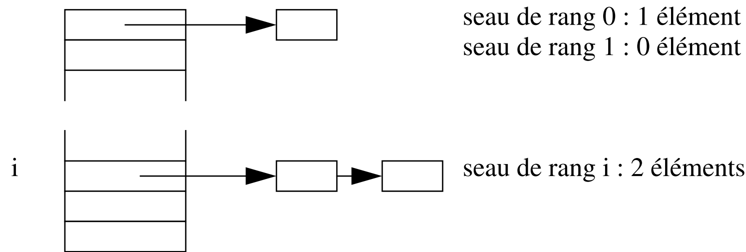
cie un entier. Un même entier peut correspondre à plusieurs valeurs différentes. En revanche, deux éléments de même valeur doivent toujours fournir le même code de hachage.

Pour organiser les éléments de la collection, on va constituer un tableau de *N* listes chaînées (nommées souvent seaux). Initialement, les seaux sont vides. À chaque ajout d'un élément à la collection, on lui attribuera un emplacement dans un des seaux dont le rang *i* (dans le tableau de seaux) est défini en fonction de son code de hachage *code* de la manière suivante :

i = code % N

S'il existe déjà des éléments dans le seau, le nouvel élément est ajouté à la fin de la liste chaînée correspondante.

On peut récapituler la situation par ce schéma :



Comme on peut s'y attendre, le choix de la valeur (initiale) de *N* sera fait en fonction du nombre d'éléments prévus pour la collection. On nomme "facteur de charge" le rapport entre le nombre d'éléments de la collection et le nombre de seaux *N*. Plus ce facteur est grand, moins (statistiquement) on obtient de seaux contenant plusieurs éléments ; plus il est grand, plus le tableau de références des seaux occupe de l'espace. Généralement, on choisit un facteur de l'ordre de 0.75. Bien entendu, la fonction de hachage joue également un rôle important dans la bonne répartition des codes des éléments dans les différents seaux.

Pour retrouver un élément de la collection (ou pour savoir s'il est présent), on détermine son code de hachage *code*. La formule *i = code % N* fournit un numéro *i* de seau dans lequel l'élément est susceptible de se trouver. Il ne reste plus qu'à parcourir les différents éléments du seau pour vérifier si la valeur donnée s'y trouve (*equals*). Notez qu'on ne recourt à la méthode *equals* que pour les seuls éléments du seau de rang *i* (nous verrons plus loin en quoi cette remarque est importante).

Avec Java, les tables de hachage sont automatiquement agrandies dès que leur facteur de charge devient trop grand (supérieur à 0.75). On retrouve là un mécanisme similaire à celui de la gestion de la capacité d'un vecteur. Certains constructeurs d'ensembles permettent de choisir la capacité et/ou le facteur de charge (voyez l'annexe G).

1. N'oubliez pas que la valeur d'un élément est formée de la valeur de ses différents champs (on pourrait aussi parler d'état).

4.5.2 La méthode hashCode

Elle est donc utilisée pour calculer le code de hachage d'un objet. Les classes *String*, *File* et les classes enveloppes définissent une méthode *hashCode* utilisant la valeur effective des objets (c'est pourquoi nous avons pu constituer sans problème des *HashSet* d'éléments de ce type). En revanche, les autres classes ne (re)définisssent pas *hashCode* et l'on recourt à la méthode *hashCode* héritée de la classe *Object*, laquelle se contente d'utiliser comme "valeur" la simple adresse des objets. Dans ces conditions, deux objets différents de même valeur auront toujours des codes de hachage différents.

Si l'on souhaite pouvoir définir une égalité des éléments basée sur leur valeur effective, il va donc falloir définir dans la classe correspondante une méthode *hashCode* :

```
int hashCode ()
```

Elle doit fournir le code de hachage correspondant à la valeur de l'objet.

Dans la définition de cette fonction, il ne faudra pas oublier que le code de hachage doit être compatible avec *equals*. Deux objets égaux pour *equals* doivent absolument fournir le même code, sinon ils risquent d'aller dans deux seaux différents ; dans ce cas, ils n'apparaîtront plus comme égaux (puisque l'on ne recourt à *equals* qu'à l'intérieur d'un même seau). De même, on ne peut pas se permettre de définir seulement *equals* sans (re)définir *hashCode*.

4.5.3 Exemple

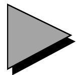
Voici un exemple utilisant un ensemble d'objets de type *Point*. La classe *Point* redéfinit convenablement les méthodes *equals* et *hashCode*. Nous avons choisi ici une détermination simple du code de hachage (somme des deux coordonnées).

```
import java.util.* ;
public class EnsPt1
{ public static void main (String args[])
  { Point p1 = new Point (1, 3), p2 = new Point (2, 2) ;
    Point p3 = new Point (4, 5), p4 = new Point (1, 8) ;
    Point p[] = {p1, p2, p1, p3, p4, p3} ;
    HashSet<Point> ens = new HashSet<Point> () ;
                                   // HashSet ens=new HashSet() ;  <-- avant JDK 5.0
     for (Point px : p)          // for (int i=0 ; i<p.length ; i++)  <-- avant JDK 5.0
    { System.out.print ("le point ") ;
      px.affiche() ;                            // p[i].affiche() ; <-- avant JDK 5.0
      boolean ajoute = ens.add (px) ;
                                // boolean ajoute = ens.add (p[i]) ; <-- avant JDK 5.0
      if (ajoute) System.out.println (" a ete ajoute") ;
             else System.out.println ("est deja present") ;
      System.out.print ("ensemble = " ) ; affiche(ens) ;
    }
  }
  public static void affiche (HashSet<Point> ens)
                          //  public static void affiche (HashSet ens) <-- avant JDK 5.0
  { Iterator<Point> iter = ens.iterator() ;
                                // Iterator iter = ens.iterator() ;  <-- avant JDK 5.0
```

```
    while (iter.hasNext())
    { Point p = iter.next() ;       // Point p = (Point)iter.next() ;  <-- avant JDK 5.0
      p.affiche() ;
    }
    System.out.println () ;
  }
}
class Point
{ Point (int x, int y) { this.x = x ; this.y = y ; }
  public int hashCode ()
  { return x+y ; }
  public boolean equals (Object pp)
  { Point p = (Point) pp ;
    return ((this.x == p.x) & (this.y == p.y)) ;
  }
  public void affiche ()
  { System.out.print ("[" + x + " " + y + "] ") ;
  }
  private int x, y ;
}
```

```
le point [1 3]  a ete ajoute
ensemble = [1 3]
le point [2 2]  a ete ajoute
ensemble = [2 2] [1 3]
le point [1 3] est deja present
ensemble = [2 2] [1 3]
le point [4 5]  a ete ajoute
ensemble = [4 5] [2 2] [1 3]
le point [1 8]  a ete ajoute
ensemble = [1 8] [4 5] [2 2] [1 3]
le point [4 5] est deja present
ensemble = [1 8] [4 5] [2 2] [1 3]
```

Exemple de redéfinition de hashCode



Remarque

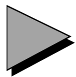
Le choix de la fonction de hachage a été dicté ici par la simplicité. En pratique, on voit que plusieurs points peuvent avoir le même code de hachage (il suffit que la somme de leurs coordonnées soit la même). Dans la pratique, il faudrait choisir une formule qui éparpille bien les codes. Mais cela n'est possible que si l'on dispose d'informations statistiques sur les valeurs des coordonnées des points.

4.6 Les ensembles TreeSet

4.6.1 Généralités

Nous venons de voir comment les ensembles *HashSet* organisaient leurs éléments en table de hachage, en vue de les retrouver rapidement (efficacité en *O(1)*). La classe *TreeSet* propose une autre organisation utilisant un "arbre binaire", lequel permet d'ordonner totalement les éléments. On y utilise, cette fois, la relation d'ordre usuelle induite par la méthode *compareTo* des objets ou par un comparateur (qu'on peut fournir à la construction de l'ensemble).

Dans ces conditions, la recherche dans cet arbre d'un élément de valeur donnée est généralement moins rapide que dans une table de hachage mais plus rapide qu'une recherche séquentielle. On peut montrer que son efficacité est en *O(Log N)*. Par ailleurs, l'utilisation d'un arbre binaire permet de disposer en permanence d'un ensemble totalement ordonné (trié). On notera d'ailleurs que la classe *TreeSet* dispose de deux méthodes spécifiques *first* et *last* fournissant respectivement le premier et le dernier élément de l'ensemble.



Remarque

On notera bien que, dans un ensemble *TreeSet*, la méthode *equals* n'intervient ni dans l'organisation de l'ensemble, ni dans le test d'appartenance d'un élément. L'égalité est définie uniquement à l'aide de la méthode *compareTo* (ou d'un comparateur). Dans un ensemble *HashSet*, la méthode *equals* intervenait (mais uniquement pour des éléments de même numéro de seau).

4.6.2 Exemple

Nous pouvons essayer d'adapter l'exemple du paragraphe 4.5.3, de manière à utiliser la classe *TreeSet* au lieu de la classe *HashSet*. La méthode *main* reste la même, à ceci près qu'on y utilise le type *TreeSet* en lieu et place du type *HashSet*.

Nous modifions la classe *Point* en supprimant les méthodes *hashCode* et *equals* et en lui faisant implémenter l'interface *Comparable* en redéfinissant *compareTo*. Ici, nous avons choisi d'ordonner les points d'une manière qu'on qualifie souvent de "lexicographique" : on compare d'abord les abscisses ; ce n'est qu'en cas d'égalité des abscisses qu'on compare les ordonnées. Notez bien que l'égalité n'a lieu que pour des points de mêmes coordonnées.

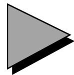
Voici notre nouvelle classe *Point* :

```
class Point implements Comparable   // ne pas oublier implements ....
{ Point (int x, int y) { this.x = x ; this.y = y ; }
  public int compareTo (Object pp)
  { Point p = (Point) pp ;    // egalite si coordonnees egales
    if (this.x < p.x) return -1 ;
     else if (this.x > p.x) return 1 ;
      else if (this.y < p.y) return -1 ;
       else if (this.y > p.y) return 1 ;
        else return 0 ;
  }
```

```
    public void affiche ()
    { System.out.print ("[" + x + " " + y + "] ") ; }
    private int x, y ;
  }
```

Le programme complet ainsi modifié figure sur le site Web d'accompagnement sous le nom *EnsPt2.java*. Voici les résultats obtenus :

```
le point [1 3]  a ete ajoute
ensemble = [1 3]
le point [2 2]  a ete ajoute
ensemble = [1 3] [2 2]
le point [1 3] est deja present
ensemble = [1 3] [2 2]
le point [4 5]  a ete ajoute
ensemble = [1 3] [2 2] [4 5]
le point [1 8]  a ete ajoute
ensemble = [1 3] [1 8] [2 2] [4 5]
le point [4 5] est deja present
ensemble = [1 3] [1 8] [2 2] [4 5]
```



Remarque

Depuis Java 6, les ensembles *TreeSet* implémentent en outre l'interface *NavigableSet* qui prévoit des méthodes exploitant l'ordre total induit par l'organisation de l'ensemble (en un arbre binaire). Ces méthodes permettent de retrouver et, éventuellement, de supprimer l'élément le plus petit ou le plus grand au sens de cet ordre, ou encore de trouver l'élément le plus proche (avant ou après) d'une "valeur" donnée. Il est possible de parcourir les éléments dans l'ordre inverse de l'ordre "naturel". Enfin, certaines méthodes permettent d'obtenir une "vue" (cette notion sera présentée ultérieurement) d'une partie de l'ensemble, formée des éléments de valeur supérieure ou inférieure à une valeur donnée. Ces différentes méthodes sont récapitulées en annexe G.

Notez que certaines d'entre elles peuvent fournir en résultat la valeur *null* qui risque de se confondre avec la référence à un élément si l'on a accepté cette possibilité. Si tel est le cas, il reste cependant possible de lever l'ambiguïté en testant simplement la valeur de *contains(null)*.

5 Les queues (JDK 5.0)

5.1 L'interface *Queue*

Le JDK 5.0 a introduit une nouvelle interface *Queue* (dérivée elle aussi de *Collection*), destinée à la gestion des files d'attente (ou queues). Il s'agit de structures dans lesquelles on peut :

- introduire un nouvel élément, si la queue n'est pas pleine,
- prélever le premier élément de la queue,

L'introduction d'un nouvel élément dans la queue se fait à l'aide d'une nouvelle méthode *offer* qui présente sur la méthode *add* (de l'interface *Collection*) l'avantage de ne pas déclencher d'exception quand la queue est pleine ; dans ce cas, *offer* renvoie simplement la valeur *false*.

Le prélèvement du premier élément de la queue peut se faire :

- de façon destructive, à l'aide de la méthode *poll* : l'élément ainsi prélevé est supprimé de la queue ; la méthode renvoie *null* si la queue est vide,
- de façon non destructive à l'aide de la méthode *peek*.

5.2 Les classes implémentant l'interface Queue

Deux classes implémentent l'interface *Queue* :

- La classe *LinkedList*, modifiée par le Java 5, pour y intégrer les nouvelles méthodes. On notera que, depuis Java 6, *LinkedList* implémente également l'interface *Deque* (présentée ci-après) disposant de méthodes d'action à la fois sur le début et sur la fin de la liste.
- La classe *PriorityQueue*, introduite par Java 5, permet de choisir une relation d'ordre ; dans ce cas, le type des éléments doit implémenter l'interface *Comparable* ou être doté d'un comparateur approprié. Les éléments de la queue sont alors ordonnés par cette relation d'ordre et le prélèvement d'un élément porte alors sur le "premier" au sens de cette relation (on parle du "plus prioritaire", d'où le nom de *PriorityQueue*).

Toutes les méthodes concernées sont décrites en annexe G.

6 Les queues à double entrée Deque (Java 6)

6.1 L'interface Deque

Java 6 a introduit une nouvelle interface *Deque*, dérivée de *Queue*, destinée à gérer des files d'attente à double entrée, c'est-à-dire dans lesquelles on peut réaliser l'une des opérations suivantes à l'une des extrémités de la queue :

- ajouter un élément,
- examiner un élément,
- supprimer un élément.

Pour chacune des ces 6 possibilités (3 actions, 2 extrémités), il existe deux méthodes :

- l'une déclenchant une exception quand l'opération échoue (pile pleine ou vide, selon le cas),
- l'autre renvoyant une valeur particulière (*null* pour une méthode de prélèvement ou d'examen, *false* pour une méthode d'ajout).

Voici la liste de ces méthodes (*First* correspondant aux actions sur la tête, *Last* aux actions sur la queue et *e* désignant un élément) :

	Exception	Valeur spéciale
Ajout	addFirst (e) addLast (e)	offerFirst () offerLast ()
Examen	getFirst () getLast()	peekFirst () peekLast ()
Suppression	removeFirst () removeLast ()	pollFirst () pollLast ()

A noter que les méthodes de l'interface *Queue* restent utilisables, sachant que celles d'ajout agissent sur la queue, tandis que celles d'examen ou de suppression agissent sur la tête.

Deux classes implémentent l'interface *Deque* :

- la classe *LinkedList*, modifiée à nouveau par Java 6, de façon appropriée ;
- la nouvelle classe *ArrayDeque* présentée ci-après.

6.2 La classe *ArrrayDeque*

Il s'agit d'une implémentation d'une queue à double entrée sous forme d'un tableau (éléments contigus en mémoire) redimensionnable à volonté (comme l'est *ArrayList*). On notera bien que, malgré son nom, cette classe n'est pas destinée à concurrencer *ArrayList*, car elle ne dispose pas d'opérateur d'accès direct à un élément. Il s'agit simplement d'une implémentation plus efficace que *LinkedList* pour une queue à double entrée.

Hormis les constructeurs, les méthodes spécifiques à cette implémentation sont *descendingIterator*, *removeFirstOccurrence* et *removeLastOccurrence*.

7 Les algorithmes

La classe *Collections* fournit, sous forme de méthodes statiques, des méthodes utilitaires générales applicables aux collections, notamment :

- recherche de maximum ou de minimum,
- tri et mélange aléatoire,
- recherche binaire,
- copie...

Ces méthodes disposent d'arguments d'un type interface *Collection* ou *List*. Dans le premier cas, l'argument effectif pourra être une collection quelconque. Dans le second cas, il devra s'agir obligatoirement d'une liste chaînée (*LinkedList*) ou d'un vecteur dynamique (*ArrayList* ou *Vector*). Nous allons examiner ici les principales méthodes de la classe *Collections*, sachant qu'elles sont toutes récapitulées en annexe G.

7.1 Recherche de maximum ou de minimum

Ces algorithmes s'appliquent à des collections quelconques (implémentant l'interface *Collection*). Ils utilisent une relation d'ordre définie classiquement :

- soit à partir de la méthode *compareTo* des éléments (il faut qu'ils implémentent l'interface *Comparable*),
- soit en fournissant un comparateur en argument de l'algorithme.

Voici un exemple de recherche du maximum des objets de type *Point* d'une liste *l*. La classe *Point* implémente ici l'interface *Comparable* et définit *compareTo* en se basant uniquement sur les abscisses des points. L'appel :

```
Collections.max (l)
```

recherche le "plus grand élément" de *l*, suivant cet ordre.

Par ailleurs, on effectue un second appel de la forme :

```
Collections.max (l, new Comparator() { ..... }
```

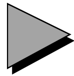
On y fournit en second argument un comparateur anonyme, c'est-à-dire un objet implémentant l'interface *Comparator* et définissant une méthode *compare* (revoyez le paragraphe 1.2.2). Cette fois, nous ordonnons les points en fonction de leur ordonnée.

```
import java.util.* ;
public class MaxMin
{ public static void main (String args[])
  { Point p1 = new Point (1, 3) ; Point p2 = new Point (2, 1) ;
    Point p3 = new Point (5, 2) ; Point p4 = new Point (3, 2) ;
    LinkedList <Point> l = new LinkedList <Point> () ;
                           // LinkedList l = new LinkedList() ;  <-- avant JDK 5.0
    l.add (p1) ; l.add (p2) ; l.add (p3) ; l.add (p4) ;

       /* max de l, suivant l'ordre defini par compareTo de Point */
    Point pMax1 = Collections.max(l) ;
                      //Point pMax1 = (Point)Collections.max (l) ; <-- avant JDK 5.0
    System.out.print ("Max suivant compareTo =  ") ; pMax1.affiche() ;
    System.out.println () ;

       /* max de l, suivant l'ordre defini par un comparateur anonyme */
    Point pMax2 = (Point)Collections.max (l, new Comparator()
                      {  public int compare (Object o1, Object o2)
                         { Point p1 = (Point) o1 ; Point p2 = (Point) o2 ;
                           if (p1.y < p2.y) return -1 ;
                             else if (p1.y == p2.y) return 0 ;
                               else return 1 ;
                         }
                      }) ;
    System.out.print ("Max suivant comparator = " ) ; pMax2.affiche() ;
  }
}
```

```
class Point implements Comparable
{ Point (int x, int y) { this.x = x ; this.y = y ; }
  public void affiche ()
  { System.out.print ("[" + x + " " + y + "] ") ;
  }
  public int compareTo (Object pp)
  { Point p = (Point) pp ;
    if (this.x < p.x) return -1 ;
      else if (this.x == p.x) return 0 ;
        else return 1 ;
  }
  public int x, y ;    // public ici, pour simplifier les choses
}
```

```
Max suivant compareTo =  [5 2]
Max suivant comparator = [1 3]
```

Recherche de maximum d'une liste de points



Remarques

1 Ici, notre collection, comme toute liste, dispose d'un ordre naturel. Cela n'empêche nullement d'y définir un ou plusieurs autres ordres, basés sur la valeur des éléments.

2 La méthode *compareTo* ou le comparateur utilisés ici sont très simplistes. Notre but était essentiellement de montrer que plusieurs ordres différents peuvent être appliqués (à des instants différents) à une même collection.

7.2 Tris et mélanges

La classe *Collections* dispose de méthodes *sort* qui réalisent des algorithmes de "tri" des éléments d'une collection qui doit, cette fois, implémenter l'interface *List*. Ses éléments sont réorganisés de façon à respecter l'ordre induit soit par *compareTo*, soit par un comparateur. L'efficacité de l'opération est en *O(N Log N)*. Le tri est "stable", ce qui signifie que deux éléments de même valeur (au sens de l'ordre induit) conservent après le tri leur ordre initial.

La classe *Collections* dispose également de méthodes *shuffle* effectuant un mélange aléatoire des éléments d'une collection (implémentant, là encore, l'interface *List*). Cette fois, leur efficacité dépend du type de la collection ; il est :

- en *O(N)* pour un vecteur dynamique,
- en *O(N * N)* pour une liste chaînée,
- en *O(N * a(N))* pour une collection quelconque (que vous aurez pu définir), *a(N)* désignant l'efficacité de l'accès à un élément quelconque de la collection.

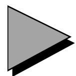
Voici un exemple dans lequel nous trions un vecteur d'éléments de type *Integer*. Rappelons que la méthode *compareTo* de ce type induit un ordre naturel. Nous effectuons ensuite un

mélange aléatoire puis nous trions à nouveau le tableau en fournissant à l'algorithme *sort* un comparateur prédéfini nommé *reverseOrder* ; ce dernier inverse simplement l'ordre induit par *compareTo*.

```
import java.util.* ;
public class Tri1
{ public static void main (String args[])
  { int nb[] = {4, 9, 2, 3, 8, 1, 3, 5} ;
    ArrayList<Integer> t = new ArrayList <Integer>() ;
                  // ArrayList t = new ArrayList() ;      <-- avant JDK 5.0
    for (Integer v : nb) t.add (v) ;
                  // for (int i=0 ; i<nb.length ; i++) t.add (new Integer(nb[i])) ; <-
    System.out.println ("t initial      = " + t) ;
    Collections.sort (t) ;
    System.out.println ("t trie         = " + t) ;
    Collections.shuffle (t) ;
    System.out.println ("t melange      = " + t) ;
    Collections.sort (t, Collections.reverseOrder()) ;
    System.out.println ("t trie inverse = " + t) ;
  }
}
```

```
t initial      = [4, 9, 2, 3, 8, 1, 3, 5]
t trie         = [1, 2, 3, 3, 4, 5, 8, 9]
t melange      = [2, 9, 8, 5, 1, 4, 3, 3]
t trie inverse = [9, 8, 5, 4, 3, 3, 2, 1]
```

Tri et mélange aléatoire d'une liste d'éléments de type Integer



Remarque

Depuis Java 8, l'interface *List* dispose d'une méthode *sort* permettant à chaque type de liste de disposer d'une méthode spécialisée, éventuellement plus efficace que celle de la classe *Collections*. Nous l'étudierons dans le chapitre consacré aux expressions lambda et aux streams.

7.3 Autres algorithmes

La plupart des algorithmes de la classe *Collections* sont récapitulés en annexe G. Vous y noterez la présence d'un algorithme de recherche binaire *binarySearch*. Comme les algorithmes précédents, il se base sur l'ordre induit par *compareTo* ou un comparateur. Il suppose, en revanche, que la collection est déjà convenablement ordonnée suivant cet ordre. Il possède une efficacité en *O(Log N)* pour les vecteurs dynamiques, en *O(N Log N)* pour les listes chaînées et en *O(a(N) Log N)* d'une manière générale.

En outre, il possède la particularité de définir l'endroit où viendrait s'insérer (toujours suivant l'ordre en question) un élément de valeur donnée (non présent dans la collection). Plus précisément :

```
binarySearch (collection, valeur)
```

fournit un entier *i* représentant :

- la position de *valeur* dans la collection, si elle y figure,
- une valeur négative telle que *valeur* puisse s'insérer dans la collection à la position de rang *-i-1*, si elle n'y figure pas.

8 Les tables associatives

8.1 Généralités

Une table associative permet de conserver une information associant deux parties nommées *clé* et *valeur*. Elle est principalement destinée à retrouver la valeur associée à une clé donnée. Les exemples les plus caractéristiques de telles tables sont :

- le dictionnaire : à un mot (clé), on associe une valeur qui est sa définition,
- l'annuaire usuel : à un nom (clé), on associe une valeur comportant le numéro de téléphone et, éventuellement, une adresse,
- l'annuaire inversé : à un numéro de téléphone (qui devient la clé), on associe une valeur comportant le nom et, éventuellement, une adresse.

On notera que les ensembles déjà étudiés sont des cas particuliers de telles tables, dans lesquelles la valeur serait vide.

Depuis le JDK 5.0, les tables associatives sont génériques, au même titre que les collections, mais elles sont définies par deux paramètes de type (celui des clés, noté généralement *K*, celui des valeurs, noté généralement *V*) au lieu d'un.

8.2 Implémentation

Comme pour les ensembles, l'intérêt des tables associatives est de pouvoir y retrouver rapidement une clé donnée pour en obtenir l'information associée. On va donc tout naturellement retrouver les deux types d'organisation rencontrés pour les ensembles :

- table de hachage : classe *HashMap*,
- arbre binaire : classe *TreeMap*.

Dans les deux cas, seule la clé sera utilisée pour ordonnancer les informations. Dans le premier cas, on se servira du code de hachage des objets formant les clés ; dans le second cas, on se servira de la relation d'ordre induite par *compareTo* ou par un comparateur fixé à la construction.

L'accès à un élément d'un *HashMap* sera en *O(1)* tandis que celle à un élément d'un *TreeMap* sera en *O(Log N)*). En contrepartie de leur accès moins rapide, les *TreeMap* seront (comme les *TreeSet*) ordonnés en permanence suivant leurs clés.

8.3 Présentation générale des classes *HashMap* et *TreeMap*

Comme nous l'avons signalé, les classes *HashMap* et *TreeMap* n'implémentent plus l'interface *Collection* mais une autre interface nommée *Map*. Ceci provient essentiellement du fait que leurs éléments ne sont plus à proprement parler des objets mais des "paires" d'objets c'est-à-dire une association entre deux objets.

Ajout d'information

La plupart des constructeurs créent une table vide. Pour ajouter une clé à une table, on utilise la méthode *put* à laquelle on fournit la clé et la valeur associée ; par exemple, si *K* désigne le type des clés et *V* celui des valeurs :

```
 /* création d'une table vide */
HashMap <K, V> m = new HashMap <K, V> () ;
                                // HashMap m = new HashMap () ;       <-- avant JDK 5.0
  .....
/* ajoute à m, un élément associant la clé "m" (String) à la valeur 3 (Integer) */
m.put ("m", 3) ;               //  m.put ("m", new Integer (3)) ;     <-- avant JDK 5.0
```

Si la clé fournie à *put* existe déjà, la valeur associée remplacera l'ancienne (une clé donnée ne pouvant figurer qu'une seule fois dans une table). D'ailleurs, *put* fournit en retour soit l'ancienne valeur si la clé existait déjà, soit *null*.

Notez que, comme pour les autres collections, les clés et les valeurs doivent être des objets. Il n'est théoriquement pas nécessaire que toutes les clés soient de même type, pas plus que les éléments. En pratique, ce sera presque toujours le cas pour des questions évidentes de facilité d'exploitation de la table.

Recherche d'information

On obtient la valeur associée à une clé donnée à l'aide de la méthode *get*, laquelle fournit *null* si la clé cherchée n'est pas présente (*K* représente le type de la clé) :

```
K o = get ("x") ;  // fournit la valeur associée à la clé "x" // K = Object avant JDK
5.0
if (o == null) System.out.println ("Aucune valeur associée à la clé x") ;
```

L'efficacité de cette recherche est en *O(1)* pour *HashMap* et en *O(Log N)* pour *TreeMap*.

La méthode *containsKey* permet de savoir si une clé donnée est présente (au sens défini au paragraphe 1.3), avec la même efficacité.

Suppression d'information

On peut supprimer un élément d'une table en utilisant la méthode *remove*, laquelle fournit en retour l'ancienne valeur associée si la clé existe ou la valeur *null* dans le cas contraire :

```
K cle = "x" ;                                        // faire K = Object avant JDK 5.0
K val = remove (cle) ;   // supprime l'élément (clé + valeur) de clé "x"
if (val != null)
   System.out.println ("On a supprimé l'élément de clé " + cle
                      + " et de valeur" + val) ;
else System.out.println ("la clé " + clé + " n'existe pas") ;
```

8.4 Parcours d'une table ; notion de vue

En théorie, les types *HashMap* et *TreeMap* ne disposent pas d'itérateurs. Mais on peut facilement, à l'aide d'une méthode nommée *entrySet*, "voir" une table comme un ensemble de "paires", une paire n'étant rien d'autre qu'un élément de type *Map.Entry* réunissant deux objets (de types *a priori* quelconques). Les méthodes *getKey* et *getValue* du type *Map.Entry* permettent d'extraire respectivement la clé et la valeur d'une paire. Nous vous proposons un canevas de parcours d'une table utilisant ces possibilités ; ici, nous avons préféré séparer la version générique (depuis JDK 5.0) de la version non générique (avant JDK 5.0)

```
HashMap <K, V> m ;
   .....
Set <Map.entry<K, V> > entrees = m.entrySet () ; // entrees est un ensemble de "paires"
Iterator <Map.entry<K, V> > iter = entrees.iterator() ;    // itérateur sur les paires
while (iter.hasNext())                                     // boucle sur les paires
{ Map.Entry <K, V> entree = (Map.Entry)iter.next() ;  // paire courante
  K cle   = entree.getKey () ;      // clé de la paire courante
  V valeur = entree.getValue() ;    // valeur de la paire courante
    .....
}
```

Canevas de parcours d'une table (depuis JDK 5.0)

```
HashMap m ;
   .....
Set entrees = m.entrySet () ;           // entrees est un ensemble de "paires"
Iterator iter = entrees.iterator() ;    // itérateur sur les paires
while (iter.hasNext())                             // boucle sur les paires
{ Map.Entry entree = (Map.Entry)iter.next() ;   // paire courante
  Object cle   = entree.getKey () ;             // clé de la paire courante
  Object valeur = entree.getValue() ;           // valeur de la paire courante
    .....
}
```

Canevas de parcours d'une table (avant JDK 5.0)

Notez que l'ensemble fourni par *entrySet* n'est pas une copie des informations figurant dans la table *m*. Il s'agit de ce que l'on nomme une "vue". Si l'on opère des modifications dans *m*, elles seront directement perceptibles sur la vue associée. De plus, si l'on applique la méthode

remove à un élément courant (paire) de la vue, on supprime du même coup l'élément de la table. En revanche, il n'est pas permis d'ajouter directement des éléments dans la vue elle-même.

8.5 Autres vues associées à une table

En dehors de la vue précédente, on peut également obtenir :

- l'ensemble des clés à l'aide de la méthode *keySet* :

```
HashMap m ;
   .....
Set cles = m.keySet () ;
```

  On peut parcourir classiquement cet ensemble à l'aide d'un itérateur. La suppression d'une clé (clé courante ou clé de valeur donnée) entraîne la suppression de l'élément correspondant de la table *m*.

- la "collection" des valeurs à l'aide de la méthode *values* :

```
Collection valeurs = m.values () ;
```

  Là encore, on pourra parcourir cette collection à l'aide d'un itérateur ; la suppression d'un élément de cette collection (élément courant ou élément de valeur donnée) entraîne la suppression de l'élément correspondant de la table *m*.

  On notera que l'on obtient une collection et non un ensemble car il est tout à fait possible que certaines valeurs apparaissent plusieurs fois.

Là encore, il ne sera pas permis d'ajouter directement des éléments dans la vue des clés ou dans la vue des valeurs (de toute façon, cela n'aurait guère de sens puisque l'information serait incomplète).

8.6 Exemple

Voici un programme qui constitue une table (*HashMap*) associant des clés de type *String* et des valeurs, elles aussi de type *String*. Il illustre la plupart des fonctionnalités décrites précédemment et, en particulier, les trois vues qu'on est susceptible d'associer à une table. On notera qu'ici il n'est pas utile de se préoccuper des méthodes *hashCode* et *equals*, ce qui serait nécessaire si l'on travaillait avec des clés ou des valeurs d'un type quelconque.

```
import java.util.* ;
public class Map1
{ public static void main (String args[])
  { HashMap <String, String> m = new HashMap <String, String> () ;
                                     // HashMap m = new HashMap() ;  <-- avant JDK 5.0
    m.put ("c", "10") ; m.put ("f", "20") ; m.put ("k", "30") ;
    m.put ("x", "40") ; m.put ("p", "50") ; m.put ("g", "60") ;
    System.out.println ("map initial :                        " + m) ;
```

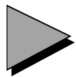
```
    // retrouver la valeur associee a la cle "f"
    String ch = m.get("f") ;      // String ch = (String)m.get("f") ; <-- avant JDK 5.0
    System.out.println ("valeur associee a f :                 " + ch) ;

    // ensemble des valeurs (attention, ici Collection, pas Set)
    Collection<String> valeurs = m.values () ;
                               // Collection valeurs = m.values() ;  <- avant JDK 5.0
    System.out.println ("liste des valeurs initiales :         " + valeurs) ;
    valeurs.remove ("30") ; // on supprime la valeur "30" par la vue associee
    System.out.println ("liste des valeurs apres sup :         " + valeurs) ;

    // ensemble des cles
    Set<String> cles = m.keySet () ;        // Set cles = m.keySet() ; <-- avant JDK 5.0
    System.out.println ("liste des cles initiales :            " + cles) ;
    cles.remove ("p") ;   // on supprime la cle "p" par la vue associee
    System.out.println ("liste des cles apres sup :            " + cles) ;

    // modification de la valeur associee a la clé x
    String old = m.put("x", "25") ;
                         // String old = (String)m.put("x", "25") ; <- avant JDK 5.0
    if (old != null)
    System.out.println ("valeur associee a x avant modif :     " + old) ;
    System.out.println ("map apres modif de x :                " + m) ;
    System.out.println ("liste des valeurs apres modif de x : " + valeurs) ;

    // On parcourt les entrees (Map.Entry) du map jusqu'a trouver la valeur 20
    // et on supprime l'element correspondant (suppose exister)
    Set<Map.Entry<String, String> > entrees = m.entrySet () ;
    Iterator<Map.Entry<String, String> > iter = entrees.iterator() ;
                                 // Set entrees = m.entrySet () ;  <-- avant JDK 5.0
                                  // Iterator iter = entrees.iterator() ;         <-

    while (iter.hasNext())
    { Map.Entry<String, String> entree = iter.next() ;
      String valeur = entree.getValue() ;
                    // Map.Entry entree = (Map.Entry)iter.next() ; <-- avant JDK 5.0
                     // String valeur = (String)entree.getValue() ;              <-
      if (valeur.equals ("20"))
        { System.out.println ("valeur 20 " + "trouvee en cle "
                             + entree.getKey()) ;
          iter.remove() ; // suppression sur la vue associee
          break ;
        }
    }
    System.out.println ("map apres sup element suivant 20 :   " + m) ;

    // on supprime l'element de cle "f"
    m.remove ("f") ;
    System.out.println ("map apres suppression f :             " + m) ;
    System.out.println ("liste des cles apres suppression f : " + cles) ;
    System.out.println ("liste des valeurs apres supp de f  : " + valeurs) ;
  }
}
```

```
map initial :                       {c=10, x=40, p=50, k=30, g=60, f=20}
valeur associee a f :               20
liste des valeurs initiales :       [10, 40, 50, 30, 60, 20]
liste des valeurs apres sup :       [10, 40, 50, 60, 20]
liste des cles initiales :          [c, x, p, g, f]
liste des cles apres sup :          [c, x, g, f]
valeur associee a x avant modif :   40
map apres modif de x :              {c=10, x=25, g=60, f=20}
liste des valeurs apres modif de x : [10, 25, 60, 20]
valeur 20 trouvee en cle f
map apres sup element suivant 20 :  {c=10, x=25, g=60}
map apres suppression f :           {c=10, x=25, g=60}
liste des cles apres suppression f : [c, x, g]
liste des valeurs apres supp de f  : [10, 25, 60]
```

Utilisation d'une table de type HashMap

Nous aurions pu utiliser sans problèmes une table de type *TreeMap*. Le programme modifié dans ce sens figure sur le site Web d'accompagnement sous le nom *Map2.java*. À titre indicatif, voici les résultats qu'il fournit (seul l'ordre des clés est modifié) :

```
map initial :                       {c=10, f=20, g=60, k=30, p=50, x=40}
valeur associee a f :               20
liste des valeurs initiales :       [10, 20, 60, 30, 50, 40]
liste des valeurs apres sup :       [10, 20, 60, 50, 40]
liste des cles initiales :          [c, f, g, p, x]
liste des cles apres sup :          [c, f, g, x]
valeur associee a x avant modif :   40
map apres modif de x :              {c=10, f=20, g=60, x=25}
liste des valeurs apres modif de x : [10, 20, 60, 25]
valeur 20 trouvee en cle f
map apres sup element suivant 20 :  {c=10, g=60, x=25}
map apres suppression f :           {c=10, g=60, x=25}
liste des cles apres suppression f : [c, g, x]
liste des valeurs apres supp de f  : [10, 60, 25]
```



Remarque

Depuis Java 6, les tables de type *TreeMap* implémentent également l'interface *NavigableMap* qui prévoit des méthodes exploitant l'ordre total induit sur les clés, par l'organisation en un arbre binaire. Ces méthodes permettent de retrouver et, éventuellement, de supprimer les éléments correspondant à la plus petite ou à la plus grande clé, ou encore de trouver l'élément ayant la clé la plus proche (avant ou après) d'une "valeur" donnée. Il est possible de parcourir les éléments dans l'ordre inverse de l'ordre naturel des clés. Enfin, on peut obtenir une vue d'une partie du map, en se fondant sur la valeur d'une clé qui sert en quelque sorte de "délimiteur". Toutes ces méthodes sont récapitulées en annexe G.

9 Vues synchronisées ou non modifiables

Nous venons de voir comment on pouvait obtenir une vue associée à une table. La vue correspondante est alors une "autre façon de voir" la collection. Elle peut interdire certaines opérations : par exemple, dans la vue des clés, on peut supprimer une clé, mais on ne peut pas en ajouter.

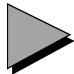
Cette notion de vue se généralise quelque peu puisque Java dispose en fait de méthodes (dans la classe *Collections*) permettant d'associer à n'importe quelle collection :

- soit une vue dite synchronisée,
- soit une vue non modifiable.

Dans une vue synchronisée, les méthodes modifiant la collection ont le qualificatif *synchronized* de sorte qu'elles ne peuvent être appelées simultanément par deux threads différents. Rappelons que les collections de Java 2 sont "non synchronisées" (exception faite de la classe *Vector*).

Dans une vue non modifiable, les méthodes modifiant la collection déclenchent une exception.

Vous trouverez la liste de ces méthodes en annexe G.



Remarque

Les méthodes *synchronizedCollection* et *unmodifiableCollection* (et uniquement celles-là) fournissent une vue dans laquelle la méthode *equals* n'utilise pas la méthode *equals* des éléments de la collection d'origine. Il en va de même pour une éventuelle méthode *hashCode*.