Collections : listes

Michel Schinz

2019-03-04

1 Introduction

En programmation, il est souvent nécessaire de manipuler non seulement des valeurs individuelles, mais aussi des groupes de valeurs. Par exemple, un programme de gestion de carnet d'adresses doit pouvoir gérer un nombre quelconque, et a priori inconnu, d'adresses.

En Java, les tableaux offrent un moyen de stocker un nombre arbitraire d'objets. Mais les tableaux ne sont pas idéaux dans toutes les situations, pour différentes raisons. Par exemple, le fait que leur taille soit fixée au moment de la création les rend difficiles à utiliser lorsque le nombre d'éléments à stocker varie.

Dès lors, il faut avoir d'autres moyens que les tableaux pour stocker et organiser des groupes d'objets. En Java, la bibliothèque standard fournit un ensemble de classes dans ce but, sujet de cette leçon et des suivantes.

2 Collections

On appelle **collection**, ou **structure de données (abstraite)** (*[abstract] data structure*), un objet servant de conteneur à d'autres objets. Par exemple :

- les tableaux,
- les listes et leurs variantes : piles, queues, « deques »,
- les ensembles,
- les tables associatives.

Chaque type de collection a ses caractéristiques propres, ses forces et ses faiblesses. Le choix de la collection à utiliser dans un cas particulier dépend donc de ce qu'on veut en faire.

Nous étudierons les trois types de collection suivants, les plus souvent rencontrés en pratique, non seulement en Java mais dans la plupart des langages de programmation actuels :

1. les **listes** (*lists*), collection ordonnée dans laquelle un élément donné peut apparaître plusieurs fois,

2. les **ensembles** (*sets*), collection non ordonnée dans laquelle un élément donné peut apparaître au plus une fois,

3. les **tables associatives** (*maps*) ou **dictionnaires** (*dictionaries*), collection associant des valeurs à des clef.

Pour chacun de ces trois types de collection il existe plusieurs **mises en œuvre** ou **implémentations** (*implementations*) différentes. Par exemple, une liste peut être mise en œuvre au moyen d'un tableau dans lequel les éléments sont stockés côte à côte, ou au moyen de nœuds chaînés entre eux via des références.

Les diverses mises en œuvre d'une collection font généralement des compromis différents, qui impliquent qu'une mise en œuvre donnée sera la meilleure dans certaines situations, mais pas dans toutes. Le choix de la mise en œuvre est donc déterminé par l'utilisation qui est faite de la collection. Il est dès lors important de bien connaître les caractéristiques des mises en œuvres à disposition.

3 Collections en Java

La bibliothèque standard Java — appelée aussi API Java, pour *application programming interface* — fournit un certain nombre de collections dans ce qui s'appelle le *Java Collections Framework (JCF)*. Tout son contenu se trouve dans le paquetage `java.util`, au demeurant très mal nommé.

3.1 Organisation

Pour chaque type de collection (liste, ensemble, etc.), la bibliothèque Java contient généralement :

- une interface, qui décrit les opérations offertes par la collection en question,

- plusieurs classes implémentant l'interface et qui sont les mises en œuvre de la collection, ayant chacune leurs caractéristiques propre.

De plus, il arrive parfois que les classes de mise en œuvre, ou en tout cas certaines d'entre elles, héritent d'une classe abstraite fournissant le code commun.

La figure 1 présente une vision simplifiée de la hiérarchie de collections que nous étudierons. Les classes abstraites fournissant le code commun ont été omises de cette hiérarchie, car elles ne constituent qu'un détail de mise en œuvre.

En plus des méthodes définies dans les interfaces des différentes collections, on trouve des méthodes statiques relatives aux collections dans les classes `Collections` et `Arrays`.

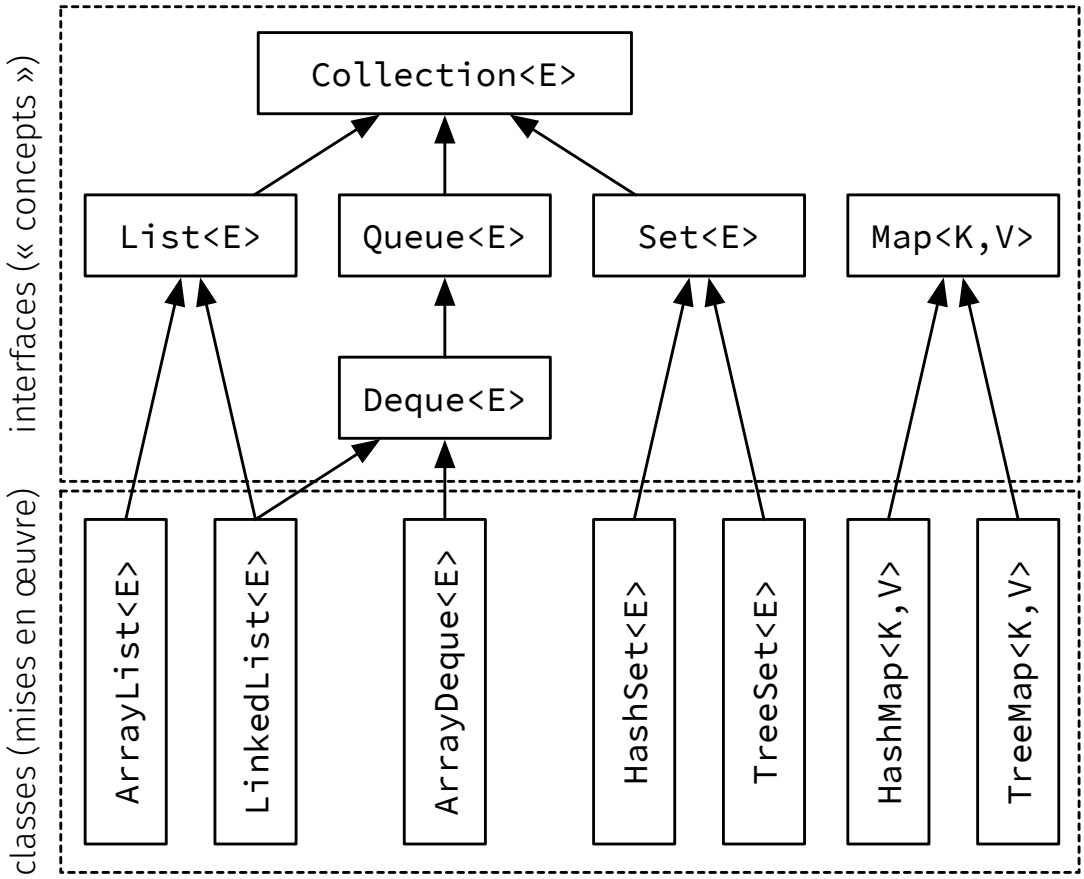

Fig. 1: Hiérarchie partielle et simplifiée des collections Java

Ces deux classes ont pour seul but de regrouper des méthodes statiques, et ne sont donc pas instanciables — leur constructeur est privé. Leurs principales méthodes seront présentées en même temps que les collections concernées.

Attention : ne confondez pas l'interface `Collection` (sans *s*) et la classe `Collections` (avec un *s*) !.

3.2 Vues

Une **vue** (*view*) est un type particulier de collection qui expose une partie d'une autre collection.

Par exemple, il est possible d'obtenir une vue sur une sous-liste d'une liste au moyen de la méthode `subList`.

Attention : une vue ne copie pas la collection à laquelle elle donne accès. Donc toute modification à la collection originale est reflétée dans la vue, et inversement !

3.3 Collections (non) modifiables

Les interfaces des différents types de collection contiennent des méthodes permettant de modifier la collection. Par exemple, l'interface `List` contient une méthode `add` permettant l'ajout d'un élément à la fin d'une liste.

On pourrait en conclure que les collections sont toujours modifiables, mais ce n'est pas le cas : toutes les méthodes de modification sont désignées comme **optionnelles**, ce qui signifie qu'elles ont le droit de simplement lever l'exception `UnsupportedOperationException` pour signaler que l'opération en question n'est pas offerte.

Ce concept de méthode optionnelle n'est pas un concept du langage Java, seulement une convention — au demeurant discutable — utilisée par les concepteurs de la bibliothèque.

Par convention, et à quelques exceptions près, une collection donnée est toujours soit :

- **modifiable**, auquel cas *aucune* de ses méthodes optionnelles ne lève l'exception `UnsupportedOperationException`, soit
- **non modifiable**, auquel cas *toutes* ses méthodes optionnelles lèvent l'exception `UnsupportedOperationException`.

Toutes les classes de mise en œuvre des collections de la bibliothèque Java (`ArrayList`, `LinkedList`, `HashSet`, etc.) sont modifiables. Pour obtenir une collection non modifiable, on peut soit :

- utiliser une méthode retournant une collection immuable, et donc non modifiable (p.ex. `emptyList` de `Collections`, qui retourne une liste immuable vide), ou
- obtenir une vue non modifiable sur une collection via les méthodes de `Collections` dont le nom commence par `unmodifiable` (`unmodifiableList`, etc.).

Attention toutefois : les vues non modifiables ne sont pas forcément immuables ! Nous y reviendrons.

3.4 L'interface `Collection`

L'interface `Collection` sert d'interface mère aux interfaces `List`, représentant les listes, et `Set`, représentant les ensembles.

Comme nous l'avons vu, les listes sont ordonnées mais les ensembles ne le sont pas. Dès lors, seules les méthodes ne dépendant pas d'une notion d'ordre sont définies dans l'interface `Collection`. Par exemple, `Collection` ne contient pas de méthode pour insérer un élément à une position donnée, puisque cette opération n'a un sens que si ceux-ci sont ordonnés. Une telle méthode existe par contre dans l'interface `List`, comme nous le verrons.

L'interface `Collection` représente donc une collection dont on ne sait pas si elle est ordonnée ou non. Elle est bien entendu générique, son paramètre de type représentant le type des éléments de la collection :

```
public interface Collection<E> {
  // … méthodes
}
```

Ses méthodes les plus importantes sont présentées ci-après, parfois avec un type simplifié pour des raisons pédagogiques.

3.4.1 Méthodes de consultation

Les méthodes ci-dessous, comme toutes celles qui ne modifient pas la collection, sont obligatoires, c-à-d qu'elles ne lèvent jamais `UnsupportedOperationException` :

- `boolean isEmpty()`, retourne vrai ssi la collection est vide,
- `int size()`, retourne le nombre d'éléments contenus dans la collection,
- `boolean contains(Object e)`, retourne vrai ssi la collection contient l'élément donné ; le type de l'argument est malheureusement `Object` et non pas `E` pour des raisons historiques,
- `boolean containsAll(Collection<E> c)`, retourne vrai ssi la collection contient tous les éléments de la collection donnée.

3.4.2 Méthodes d'ajout

Les méthodes d'ajout ci-dessous, comme toutes les méthodes de modification, sont optionnelles, c-à-d qu'elles peuvent lever l'exception `UnsupportedOperationException` :

- `boolean add(E e)`, ajoute l'élément donné à la collection,
- `boolean addAll(Collection<E> c)`, ajoute à la collection tous les éléments de la collection donnée.

La valeur de retour de ces méthodes indique si le contenu de la collection a changé suite à l'ajout. Si la collection est une liste, cela est toujours le cas, mais si la collection est un ensemble — qui n'admet pas de doublons — ce n'est pas forcément le cas.

3.4.3 Méthodes de suppression

Les méthodes de suppression ci-dessous sont optionnelles :

- `void clear()`, supprime tous les éléments de la collection,
- `boolean remove(Object e)`, supprime l'élément donné, s'il se trouve dans la collection,
- `boolean removeAll(Collection<E> c)`, supprime tous les éléments de la collection donnée,
- `boolean removeIf(Predicate<E> p)`, supprime tous les éléments qui satisfont le prédicat,
- `boolean retainAll(Collection<E> c)`, supprime tous les éléments qui ne se trouvent pas dans la collection donnée.

Les quatre dernières méthodes retournent vrai ssi le contenu de la collection a changé.

La méthode `removeIf` a pour but d'être utilisée avec une lambda (concept qui sera étudié dans une leçon ultérieure), et permet l'écriture de code concis et clair. Par exemple, l'extrait de programme ci-dessous supprime tous les éléments pairs d'une collection d'entiers :

```
Collection<Integer> collection = …;
collection.removeIf(x -> x % 2 == 0);
```

4 Listes

Une **liste** (*list*) est une séquence ordonnée et dynamique (donc à taille variable) d'objets.

Les listes sont très similaires aux tableaux, au point que la différence entre les deux est souvent floue. Toutefois, les tableaux sont généralement de taille fixe et à accès aléatoire, tandis que les listes sont généralement de taille variable et à accès séquentiel[1].

1. L'accès aléatoire signifie que l'accès à un élément dont on connaît l'index se fait en O(1), alors que l'accès séquentiel signifie que la même opération se fait en O(n).

Quelques cas particuliers des listes se rencontrent assez fréquemment pour avoir un nom propre : les piles, les queues et les « deques ». Souvent, ces cas particuliers peuvent être mis en œuvre de manière plus efficace que les listes dans toute leur généralité. Il n'est donc pas rare de trouver des classes modélisant ces cas particuliers des listes dans les bibliothèques.

5 Listes en Java

5.1 L'interface `List`

Le concept de liste est représenté dans la bibliothèque Java par l'interface `List` du paquetage `java.util`. Tout comme `Collection` — dont elle hérite — cette interface est générique et son paramètre de type représentant le type des éléments de la liste :

```
public interface List<E> extends Collection<E> {
  // … méthodes
}
```

Les principales méthodes que cette interface ajoute à celles de `Collection` sont présentées ci-après, parfois avec un type simplifié pour des raisons pédagogiques.

5.1.1 Méthodes de consultation

L'interface `List` ajoute les trois méthodes suivantes aux méthodes de consultation de `Collection` :

- `E get(int i)`, retourne l'élément qui se trouve à l'index donné ou lève une exception s'il est invalide,
- `int indexOf(Object e)`, retourne l'index de la première occurrence de l'élément donné, ou –1 s'il ne se trouve pas dans la liste,
- `int lastIndexOf(Object e)`, retourne l'index de la dernière occurrence de l'élément donné, ou –1 s'il ne se trouve pas dans la liste.

Notez que, comme dans les tableaux, le premier élément de la liste a l'index 0.

5.1.2 Méthodes d'ajout

L'interface `List` ajoute deux variantes des méthodes d'ajout qui permettent d'ajouter un élément à un index donné :

- `void add(int i, E e)`, insère l'élément donné à l'index donné de la liste,

- `boolean addAll(int i, Collection<E> c)`, insère tous les éléments de la collection donnée à l'index donné de la liste.

Ces méthodes lèvent une exception si l'index est invalide.

5.1.3 Méthodes de modification

L'interface `List` ajoute les deux méthodes suivantes aux méthodes de modification et suppression de `Collection` :

- `E remove(int i)`, supprime et retourne l'élément à l'index donné, ou lève une exception si l'index est invalide,
- `E set(int i, E e)`, remplace l'élément à l'index donné par celui donné et retourne l'ancien élément, ou lève une exception si l'index est invalide,
- `void replaceAll(UnaryOperator<E> op)`, remplace chaque élément par le résultat de l'application de l'opérateur à cet élément.

La méthode `replaceAll` est prévue pour être utilisée avec une lambda. Par exemple, on peut remplacer tous les nombres d'une liste d'entiers par leur carré ainsi :

```
List<Integer> l = …;
l.replaceAll(x -> x * x);
```

5.1.4 Vue sur une sous-liste

Finalement, l'interface `List` offre une méthode pour obtenir une vue sur une sous-liste :

- `List<E> subList(int b, int e)`, retourne une vue sur la sous-liste entre les index donnés, le premier étant inclusif, le second exclusif.

Comme il s'agit d'une vue, et non d'une copie, les modifications qu'on lui apporte sont répercutées sur la liste sous-jacente. Cela permet, par exemple, de supprimer tous les éléments dans un intervalle donné :

```
// Supprime les éléments d'index 4 et 5 :
l.subList(4, 6).clear();
```

5.1.5 Mélange et tri

La classe `Collections` (avec un *s* !) possède des méthodes — statiques, bien entendu — permettant de trier et de mélanger les éléments d'une liste :

- `<T> void sort(List<T> l)`, trie la liste donnée par ordre croissant ; la classe des éléments doit implémenter l'interface `Comparable`, que nous examinerons ultérieurement,
- `<T> void sort(List<T> l, Comparator<T> c)`, trie la liste donnée par ordre croissant, en utilisant le comparateur donné pour comparer les éléments,
- `<T> void shuffle(List<T> l)`, mélange aléatoirement les éléments de la liste donnée.

5.1.6 Vues sur un tableau

La classe `Arrays` offre une méthode permettant d'obtenir une vue sur un tableau sous la forme d'une liste :

- `<T> List<T> asList(T... a)`, retourne une vue sur un tableau contenant les éléments donnés.

La vue retournée est partiellement modifiable : il est possible d'utiliser la méthode `set` pour modifier ses éléments, mais toute utilisation d'une méthode changeant la taille de la liste (p.ex. `add` ou `remove`) provoque la levée de l'exception `UnsupportedOperationException`.

Cette méthode est souvent utilisée pour créer une liste de taille fixe composée d'éléments connus, p.ex. :

```
List<String> seasons =
  Arrays.asList("printemps", "été", "automne", "hiver");
```

5.1.7 Listes immuables

La classe `Collections` offre une méthode de construction de liste vide immuable :

- `<T> List<T> emptyList()`, retourne une liste vide immuable,

ainsi que des méthodes créant des listes immuables composées d'un élément unique répété un certain nombre de fois :

- `<T> List<T> singletonList(T e)`, retourne une liste immuable de longueur 1 contenant uniquement l'élément donné,
- `<T> List<T> nCopies(int n, T e)`, retourne une liste immuable de longueur donnée contenant uniquement l'élément donné, répété autant de fois que nécessaire.

5.1.8 Vues non modifiables

La méthode `unmodifiableList` de `Collections` retourne une vue non modifiable d'une liste :

- `<T> List<T> unmodifiableList(List<T> l)`.

Mais attention : comme il s'agit d'une vue, les éventuelles modifications ultérieures de la liste sont visibles à travers la vue ! Exemple :

```
List<Integer> list = new ArrayList<>();
list.add(1);
List<Integer> view = Collections.unmodifiableList(l);
list.add(2);
System.out.println(view); // imprime [1,2]!
```

En d'autres termes, `unmodifiableList` ne permet pas à elle seule d'obtenir une liste immuable à partir d'une liste quelconque. Pour faire cela, il faut procéder en deux temps :

1. copier la liste,
2. obtenir une vue non modifiable sur cette copie.

Pour peu que personne n'ait accès à la copie — ce qui permettrait de la modifier — la vue est alors effectivement une liste immuable. Attention toutefois, si les éléments de cette liste ne sont pas immuables, ils doivent aussi être copiés, et ainsi de suite !

Règle des listes immuables : Pour obtenir une liste immuable à partir d'une liste quelconque, obtenez une vue non modifiable d'une copie de cette liste.

De plus, faites la copie au moyen du constructeur de copie de `ArrayList`, cette mise en œuvre étant la plus efficace pour les listes immuables :

```
List…<> immutableList =
  Collections.unmodifiableList(new ArrayList<>(list));
```

5.2 Mises en œuvre des listes

La bibliothèque Java offre deux mises en œuvres principales de l'interface `List`, à savoir les **tableaux-listes**, ou tableaux dynamiques, (classe `ArrayList`) et les **listes chaînées** (classe `LinkedList`).

Le choix de l'une ou l'autre de ces mises en œuvre dans une situation donnée dépend de l'utilisation qui est faite de la liste. La table ci-dessous, qui compare les complexités des opérations les plus fréquentes pour les deux mises en œuvre, peut servir de guide.

Opération	`ArrayList`	`LinkedList`
ajout (`add`), suppression (`remove`)	O(n)	O(1)
accès (`get`), modification (`set`)	O(1)	O(n)

Mais attention : les complexités données ci-dessus ne sont valables que dans des cas bien précis !

En particulier, l'ajout et la suppression d'un élément dans une liste chaînée est en O(1) uniquement s'il n'est pas nécessaire de parcourir la liste en premier lieu pour accéder au point d'ajout ou de suppression. En pratique, cela n'est donc vrai que si on utilise les méthodes `add` et `remove` d'un itérateur de liste (voir §8.3), ou si l'ajout ou la suppression se font en début ou en fin de liste (premier ou dernier élément).

D'autre part, l'ajout et la suppression d'un élément dans un tableau-liste sont en O(1) lorsqu'elles se font à la fin de la liste.

6 Piles, queues et deques

Certains cas particuliers des listes sont assez fréquents pour avoir leur nom propre, et souvent une mise en œuvre spécifique plus efficace que celle des listes générales. Les plus importants d'entre eux sont les piles, les queues et les deques :

- une **pile** (*stack*) est une liste dont les éléments sont toujours insérés ou supprimés à la même extrémité, appelée le **sommet** (*top*),
- une **queue** (*queue*) est une liste dont les éléments sont toujours insérés à une extrémité et retirés de l'autre,
- un(e ?) « **deque** » (néologisme anglais signifiant *double-ended queue*) est une liste dont les éléments sont toujours insérés et supprimés à l'une des deux extrémités.

Ces trois cas particuliers des listes sont présentés graphiquement dans la figure 2 ci-dessous.

En anglais, une pile est parfois appelée **LIFO** (*last in, first out*) car le dernier élément qu'on y place est le premier à en sortir. Une queue, quant à elle, est parfois appelée **FIFO** (*first in, first out*) car le premier élément qu'on y place est le premier à en sortir.

6.1 Producteur/consommateur

En programmation, les queues et les deques sont souvent utilisés pour échanger des données entre un producteur — qui produit des valeurs et les place dans une queue — et un consommateur — qui utilise les valeurs produites en les obtenant de la queue.

Cette organisation permet au producteur et au consommateur de travailler indépendamment l'un de l'autre, chacun à son rythme.

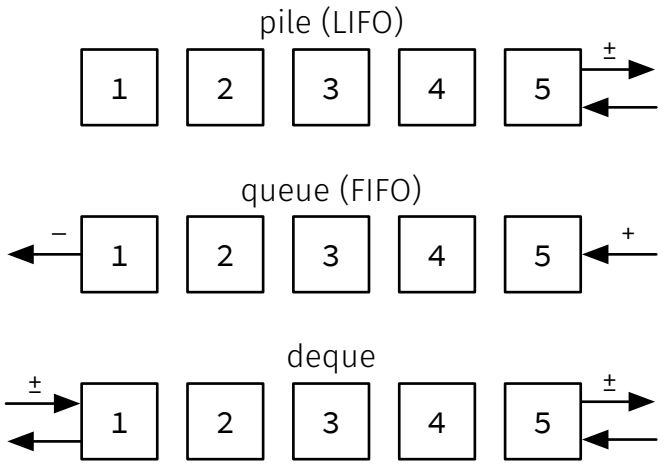

Fig. 2: Piles, queues et deques

Lorsque producteur et consommateur travaillent à leur rythme et ne communiquent que via une queue, le risque existe que le producteur travaille beaucoup plus vite que le consommateur, faisant grossir la queue de communication jusqu'à utiliser toute la mémoire à disposition.

Pour éviter ce problème, les queues et les deques peuvent être **bornées** (*bounded*), c-à-d que leur capacité peut être limitée.

Avec une telle queue, le producteur ne place une valeur dans la queue que si celle-ci n'est pas pleine, et attend que ce soit le cas sinon. Le consommateur, quant à lui, continue à vider la queue à son rythme.

7 Piles, queues et deques en Java

Dans la bibliothèque Java, les queues sont représentées par l'interface `Queue`, les deques par l'interface `Deque`. Comme toujours, plusieurs classes de mises en œuvre existent, comme illustré sur la figure 1.

7.1 L'interface `Queue`

L'interface `Queue`, qui hérite de l'interface `Collection`, n'ajoute (ou ne redéfinit) que trois paires de méthodes.

Les deux méthodes formant une paire se distinguent par la manière dont elles signalent une erreur, c-à-d le fait que la queue soit pleine ou vide : la première méthode utilise une exception, la seconde une valeur de retour spéciale.

La version utilisant une valeur de retour spéciale est généralement plus facile à utiliser en présence d'une queue bornée, car il est alors relativement normal que celle-ci soit pleine ou vide.

7.1.1 Consultation

L'interface `Queue` offre la paire de méthodes suivantes pour consulter — sans le supprimer — l'élément en tête de queue :

- `E element()`, retourne l'élément en tête de queue, ou lève une exception si la queue est vide,
- `E peek()`, retourne l'élément en tête de queue, ou `null` si elle est vide.

(Par tête de queue on entend l'extrémité de la queue de laquelle les éléments sont retirés.)

7.1.2 Ajout

L'interface `Queue` redéfinit ou ajoute la paire de méthodes suivante pour ajouter un élément à la queue :

- `boolean add(E e)`, ajoute l'élément donné à la queue et retourne vrai, ou lève une exception si celle-ci est bornée et pleine,
- `boolean offer(E e)`, essaie d'ajouter l'élément donné à la queue et retourne vrai si cela a été possible — c-à-d si la queue n'est pas bornée ou pas pleine — et faux sinon.

7.1.3 Suppression

L'interface `Queue` offre la paire de méthodes suivantes pour supprimer l'élément en tête de queue :

- `E remove()` : supprime et retourne l'élément en tête de queue, ou lève une exception si celle-ci est vide,
- `E poll()` : supprime et retourne l'élément en tête de queue s'il existe, ou ne fait rien et retourne `null` si la queue est vide.

7.2 L'interface `Deque`

L'interface `Deque` ne fait (presque) que généraliser l'interface `Queue` en offrant deux variantes de chacune des méthodes de `Queue`, une par extrémité de la deque.

Ces méthodes ne sont donc pas présentées en détail mais résumées dans la table ci-dessous :

Equivalent `Queue`	au début	à la fin
`element`	`getFirst`	`getLast`
`peek`	`peekFirst`	`peekLast`
`add`	`addFirst`	`addLast`
`offer`	`offerFirst`	`offerLast`
`remove`	`removeFirst`	`removeLast`
`poll`	`pollFirst`	`pollLast`

7.3 Mises en œuvre

Une liste chaînée peut servir de relativement bonne mise en œuvre d'une queue ou d'une deque, raison pour laquelle `LinkedList` implémente l'interface `Deque` — et donc `Queue`.

`ArrayList`, par contre, serait une très mauvaise mise en œuvre d'une queue ou d'une deque, car l'ajout ou la suppression d'élément au début de la liste est en $O(n)$. Pour cette raison, une mise en œuvre légèrement différente mais aussi basée sur un tableau redimensionné au besoin est fournie dans la classe `ArrayDeque`.

Règle des listes : Pour représenter une pile, une queue ou un « deque », utilisez `ArrayDeque`. Pour représenter une liste dans toute sa généralité, utilisez `ArrayList` si les opérations d'indexation (`get`, `set`) dominent, sinon `LinkedList`.

Note : `ArrayList` peut également s'utiliser comme une pile, pour peu que les ajouts/suppressions se fassent toujours à la fin de la liste et pas au début.

8 Parcours des collections

Il est très fréquent de devoir parcourir les éléments d'une collection. Comment faire ?

Par exemple, admettons que l'on désire parcourir une liste de chaînes de caractères pour afficher ses éléments à l'écran. Une première — mais très mauvaise — idée consiste à utiliser une boucle `for` et la méthode `get`, comme pour un tableau :

```
List<String> l = …;
for (int i = 0; i < l.size(); ++i)
  System.out.println(l.get(i));
```

Cette solution est mauvaise car, dans le cas des listes chaînées, la méthode `get` a une complexité de $O(n)$, où n est la taille de la liste. La boucle d'impression a alors une complexité de $O(n^2)$, ce qui est clairement insatisfaisant sachant qu'elle n'examine les éléments qu'une seule fois…

Pour faire mieux, on peut utiliser la boucle *for-each*, car les listes — et d'autres collections — peuvent être parcourues ainsi. La boucle d'impression peut donc se récrire comme suit :

```
List<String> l = …;
for (String s: l)
  System.out.println(s);
```

Dans le cas des listes en tout cas, cette boucle a une complexité de O(n), même avec les listes chaînées. Comment est-ce possible ? Grâce à la notion d'itérateur !

8.1 Itérateurs

Un **itérateur** (*iterator*) ou **curseur** (*cursor*) est un objet qui désigne un élément d'une collection.

Un itérateur permet d'une part d'obtenir l'élément qu'il désigne, et sait d'autre part se déplacer efficacement sur l'élément suivant — et parfois précédent — de la collection.

Dans la bibliothèque Java, le concept d'itérateur est décrit par l'interface générique `Iterator`. Son paramètre de type représente le type des éléments de la collection parcourue par l'itérateur :

```
public interface Iterator<E> {
  // … méthodes
}
```

L'interface `Iterator` est très simple et ne contient que trois méthodes, dont une (`remove`) est optionnelle :

- `boolean hasNext()`, retourne vrai ssi il reste au moins un élément à parcourir,
- `E next()`, retourne l'élément suivant et avance l'itérateur sur son successeur, ou lève une exception s'il ne reste plus d'éléments,
- `void remove()`, supprime le dernier élément retourné par `next`, ou lève une exception si `next` n'a pas encore été appelée, ou si `remove` a déjà été appelée une fois depuis le dernier appel à `next`.

Les collections dont on peut parcourir les éléments offrent toutes une méthode `iterator` permettant d'obtenir un nouvel itérateur désignant le premier élément. Au moyen de cette méthode, la boucle d'impression ci-dessus peut s'écrire également ainsi :

```
List<String> l = …;
Iterator<String> i = l.iterator();
while (i.hasNext()) {
  String s = i.next();
  System.out.println(s);
}
```

Nous connaissons maintenant deux techniques (efficaces) pour parcourir une liste : la boucle *for-each* et les itérateurs. Laquelle préférer ?

En termes d'efficacité, ces deux techniques sont rigoureusement équivalentes, la boucle *for-each* étant récrite par le compilateur Java en une boucle basée sur un itérateur.

Par contre, la boucle *for-each* est plus concise et facile à comprendre. Elle est toutefois moins générale, puisqu'il n'est p.ex. pas possible de supprimer un élément de la collection lors du parcours, comme le permet la méthode `remove` de l'itérateur.

> **Règle des itérateurs :** Pour parcourir une liste, utilisez la boucle *for-each* sauf dans le cas où vous avez besoin d'accéder directement à l'itérateur.
>
> En particulier, évitez de parcourir une liste au moyen de la méthode `get`, sauf si vous avez la certitude qu'il s'agit d'un tableau-liste.

8.2 L'interface `Iterable`

La boucle *for-each* peut en fait être utilisée sur n'importe quel objet qui implémente l'interface `Iterable`, ce qui est entre autre le cas de `Collection`.

L'interface `Iterable`, très simple, ne possède qu'une seule méthode abstraite, la méthode `iterator` vue précédemment :

```
public interface Iterable<E> {
  public Iterator<E> iterator();
}
```

Son paramètre de type (`E`) représente le type des éléments parcourus par l'itérateur.

En plus de cette méthode abstraite, l'interface `Iterable` offre une méthode par défaut, `forEach(Consumer<E> c)`, destinée à être utilisée avec une lambda. Elle applique le consommateur à chaque élément fourni par l'itérateur retourné par `iterator`, dans l'ordre. On peut p.ex. l'utiliser pour afficher tous les éléments d'une liste, chacun sur une ligne séparée :

```
List<Integer> l = Arrays.asList(1, 2, 3, 4, 5);
l.forEach(System.out::println); // affiche 1, puis 2, etc.
```

Cette méthode offre une troisième technique de parcours des collections en $O(n)$! Son utilisation est toutefois à réserver aux cas où le corps de la boucle est très petit, p.ex. un simple appel de méthode comme ci-dessus. Dans les cas plus complexes, une boucle *for-each* est généralement plus claire.

8.3 Itérateurs de listes

En plus de la méthode `iterator` qui fournit un itérateur de type `Iterator`, l'interface `List` définit une méthode nommée `listIterator` fournissant un itérateur de type `ListIterator` offrant des méthodes additionnelles pour :

- se déplacer en arrière (`hasPrevious` et `previous`),
- connaître l'index des éléments voisins de l'itérateur (`nextIndex`, `previousIndex`),
- insérer un élément dans la liste, à l'endroit désigné par l'itérateur (`add`),
- changer l'élément désigné par l'itérateur (`set`).

Logiquement, `add` et `set` sont optionnelles.

9 Références

- *Java Generics and Collections* de Maurice Naftalin et Philip Wadler,
- différents documents fournis par Oracle au sujet des collections, en particulier :
  - le tutoriel *Java Collections Framework*,
  - *Collections Framework Overview*,
  - *Outline of the Collections Framework*,
- la documentation de l'API Java, en particulier les classes et interfaces suivantes :
  - les interface `java.util.Collection` et `java.util.List`,
  - les classes `java.util.Collections` et `java.util.Arrays`,
  - les classes `java.util.ArrayList` et `java.util.LinkedList`,
  - l'interface `java.util.Iterator`.