Collections : listes

Pratique de la programmation orientée-objet
Michel Schinz – 2015-02-23

Collections

Collection

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

- 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.

Collections étudiées

Nous étudierons les trois types de collection suivants, les plus souvent rencontrés en pratique :

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.

Mises en œuvre

Pour chaque type de collection (liste, ensemble, …) il existe un très grand nombre de **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.

Choix d'une mise en œuvre

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.

Collections dans l'API Java

Collections dans l'API 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 (très mal nommé) paquetage `java.util`.

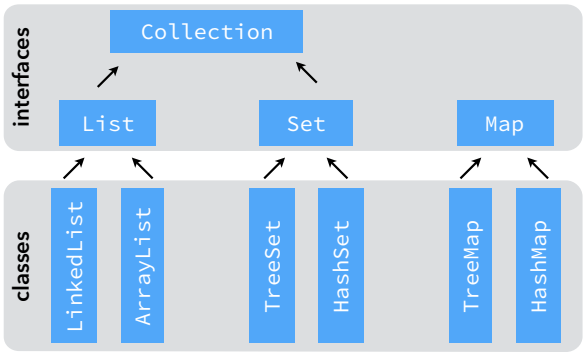
Organisation

Pour chaque type de collection (liste, ensemble, etc.), l'API Java contient généralement :

- une interface, qui décrit les opérations offertes par la collection en question,
- une classe abstraite, qui implémente l'interface et contient le code commun à toutes les mises en œuvre,
- plusieurs sous-classes concrètes de la classe abstraite, qui sont les mises en œuvre de la collection, ayant chacune leurs caractéristiques propre.

Par exemple, pour les listes, l'interface est `List`, la classe abstraite `AbstractList` et parmi les mises en œuvre concrètes figurent `ArrayList` et `LinkedList`.

Collections de l'API Java



(vue très partielle et simplifiée)

Règle des collections

En dehors des énoncés new, utilisez toujours les interfaces (`List`, `Set`, `Map`, etc.) plutôt que les mises en œuvre (`ArrayList`, `LinkedList`, etc.).

Par exemple, écrivez

```
List<String> l = new ArrayList<>();
```

plutôt que

```
ArrayList<String> l = new ArrayList<>();
```

Le code est ainsi plus général, et changer la mise en œuvre utilisée se fait très facilement.

Règle des interfaces

La règle des collections est en fait un cas particulier de la règle plus générale suivante :

En dehors des énoncés new, utilisez toujours les interfaces – si elles existent – plutôt que les classes concrètes.

Cette règle est connue en anglais sous la forme suivante :

Program to an interface, not an implementation.

Collections / Arrays

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`.
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 à ne pas confondre l'interface `Collection` et la classe `Collections` !).

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 !

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 liste.
On pourrait en conclure que les collections sont toujours modifiable, 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`.
(Ce concept de méthode optionnelle n'est pas un concept du langage Java, seulement une convention [au demeurant discutable] utilisée par les concepteurs de l'API).

Collections (non)modifiables

Par convention, une collection donnée est toujours soit :

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

Collections (non)modifiables

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

- utiliser une méthode retournant une collection immuable, et donc non modifiable (`emptyList` de `Collections`, `asList` de `Arrays`, etc.), ou
- obtenir une vue non modifiable sur une collection via les méthodes `unmodifiable…` de `Collections` (`unmodifiableList`, etc.).

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

L'interface `Collection`

L'interface `Collection`

L'interface `Collection` sert de super-interface commune aux listes (interface `List`) et à leurs variantes, ainsi qu'aux ensembles (interface `Set`).
Comme nous l'avons vu, les listes sont ordonnées mais les ensembles ne le sont pas. Dès lors, seules les méthodes qui ne dépendent 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`

L'interface `Collection` représente 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
}
```

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

Méthodes de consultation

Les méthodes ci-dessous, comme toutes celles qui ne modifient pas la collection, sont obligatoires :

- `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.

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 `UnsupportedOperation…`).

- `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.

Méthodes de suppression

Les méthodes de suppression ci-dessous sont optionnelles :

- `void clear()` : supprime tous les éléments de la collection,
- `boolean remove(E 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 retainAll(Collection<E> c)` : supprime tous les éléments qui ne se trouvent pas dans la collection donnée.

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

Collection n° 1 : la liste

(et ses variantes)

Listes

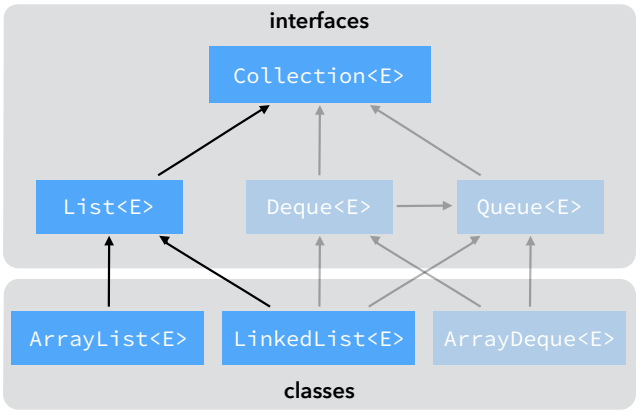
Une **liste** (*list*) est une séquence ordonnée et dynamique (donc à taille variable) d'objets.
Les listes sont très similaires aux tableaux, au point que la différence entre les deux est souvent floue. Toutefois, les tableaux sont généralement de taille fixe et à accès aléatoire tandis que les listes sont de taille variable et à accès séquentiel.
(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*)).

Cas particuliers

Quelques cas particuliers des listes se rencontrent assez fréquemment pour avoir un nom propre, p.ex. :
- les piles,
- les queues,
- 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.

Listes de l'API Java



L'interface `List`

Le concept de liste est représenté dans l'API 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.

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 la position donnée ou lève une exception si celle-ci est invalide,
- `int indexOf(E e)` : retourne la position de la première occurrence de l'élément donné, ou -1 s'il ne se trouve pas dans la liste,
- `int lastIndexOf(E e)` : retourne la position 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 est à la position 0.

Méthodes d'ajout

L'interface `List` ajoute deux variantes des méthodes d'ajout qui permettent d'ajouter un élément à une position donnée :

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

Ces méthodes lèvent une exception si la position donnée est invalide.

Méthodes de modification

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

- `E remove(int i)` : supprime et retourne l'élément à la position donnée,
- `E set(int i, E e)` : remplace l'élément à la position donnée par celui donné, et retourne l'ancien élément.

Ces méthodes lèvent une exception si la position donnée est invalide.

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 positions données, la première étant inclusive, la seconde exclusive.

Les modifications apportées à une telle vue sont répercutées sur la liste sous-jacente. Cela permet, par exemple, de supprimer tous les éléments dans un intervalle donné :

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

Mélange et tri

La classe `Collections` 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 shuffle(List<T> l)` : mélange aléatoirement les éléments de la liste donnée.

Listes immuables (1)

La classe `Arrays` offre une méthode de construction de liste immuable :

- `<T> List<T> asList(T... a)` : retourne une liste immuable contenant les éléments donnés.

Cette méthode offre un moyen simple de créer une liste constante, p.ex. :

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

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

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

Listes immuables (2)

La classe `Collections` offre également des méthodes permettant de créer des listes immuables contenant un même élément une ou plusieurs 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.

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 : il s'agit d'une vue, donc les éventuelles modifications ultérieures de la liste seront répercutées dans 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]!
```

Listes immuables

Etant donné que `unmodifiableList` retourne une vue sur une liste, elle 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 en question,
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.

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 :

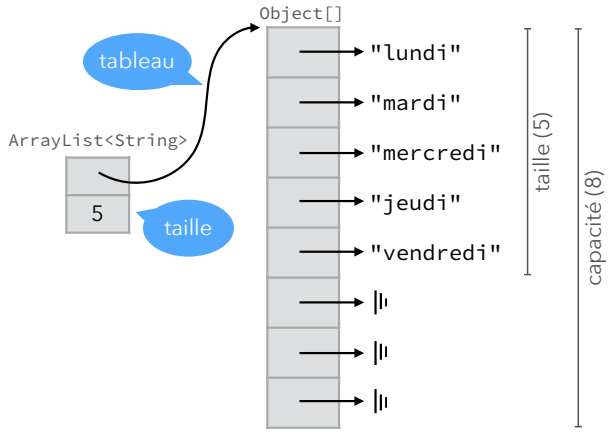
```
List<…> immutableList =
  Collections.unmodifiableList(
    new ArrayList<>(list));
```


Mise en œuvre n°1 : tableaux-listes

Les tableaux-listes

Les tableaux-listes (classe `ArrayList`) – que vous connaissez déjà sous le nom de tableaux dynamiques – sont à mi-chemin entre les tableaux et les listes, d'où leur nom.
Les éléments d'un tableau-liste sont stockés dans un tableau normal qui est, au besoin, « agrandi » par copie dans un nouveau tableau plus grand.

Tableau-liste



Tableaux-listes

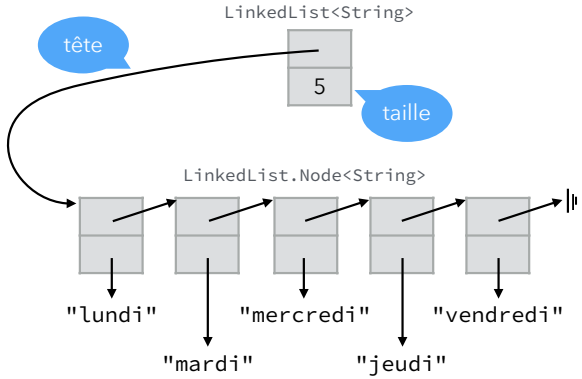
La grande force des tableaux-listes est qu'ils permettent d'accéder à un élément dont on connaît l'index en temps constant, c-à-d en O(1).
Leur faiblesse est que l'insertion d'un élément à une position quelconque implique de déplacer tous les éléments se trouvant au-dessus, et la complexité de cette opération est donc O(*n*).
Cela dit, si l'insertion se fait uniquement à la fin de la liste, elle a alors une complexité de O(1) *amortie*.

Mise en œuvre n°2 : listes chaînées

Les listes chaînées

Les éléments d'une liste chaînée sont référencés par des **nœuds** (*node*) qui sont chaînés entre-eux. C'est-à-dire que chaque nœud possède une référence vers au moins un de ses voisins.
Lorsque chaque nœud possède une référence vers un seul de ses voisins, on parle de **liste simplement chaînée** (*singly linked list*) ; lorsque chaque nœud possède une référence vers ses deux voisins, on parle de **liste doublement chaînée** (*doubly linked list*)**.**
Finalement, lorsque le dernier nœud et le premier nœud sont voisins – et donc liés par une ou deux références – on parle de **liste circulaire** (*circular list*).

Liste (simplement) chaînée



Liste chaînée

Contrairement aux tableaux-listes, les listes chaînées ne permettent pas d'accéder à un élément dont on connaît l'index en O(1). Avec une liste chaînée, cette même opération a une complexité de O(*n*).
En contrepartie, l'insertion d'un élément à une position quelconque n'implique pas de déplacer des éléments et peut donc se faire en O(1). Mais attention : cela n'est vrai qu'en faisant l'hypothèse qu'on possède déjà une référence sur un nœud voisin de celui que l'on désire insérer, faute de quoi la simple recherche de ce nœud a une complexité de O(*n*)…

Piles, queues et deques

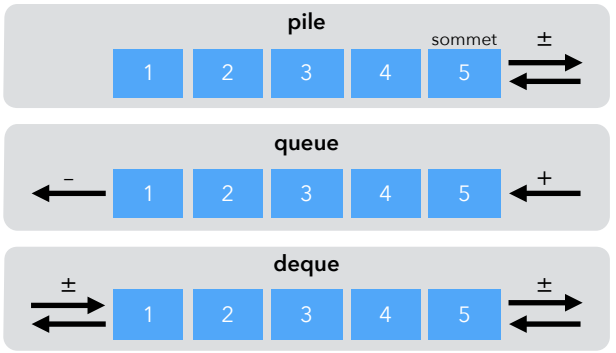
Piles

Une **pile** (*stack*) est une liste dans laquelle les éléments sont toujours insérés ou supprimés de la même extrémité, appelée le **sommet** (*top*) de la pile.
Leur nom vient du fait qu'elles sont similaires aux piles d'objets physiques, p.ex. une pile d'assiettes.
En anglais, une pile est parfois appelée **LIFO**, pour *last-in, first-out*, car le dernier élément que l'on place sur une pile est le premier à en sortir.

Queues et « deques »

Une **queue** (*queue*) est une liste dans laquelle les éléments sont toujours ajoutés à une extrémité et retirés de l'autre extrémité.
En anglais, une queue est parfois appelée **FIFO**, pour *first-in, first-out*, car le premier élément que l'on place dans une queue est le premier à en sortir.
Un(e?) « **deque** » (néologisme anglais tiré de *double-ended queue*) est une généralisation d'une queue dans laquelle les éléments peuvent être ajoutés ou supprimés à n'importe laquelle des deux extrémités – mais nulle part ailleurs.

Piles, queues, deques



+ : ajout, – : suppression, ± : ajout/suppression

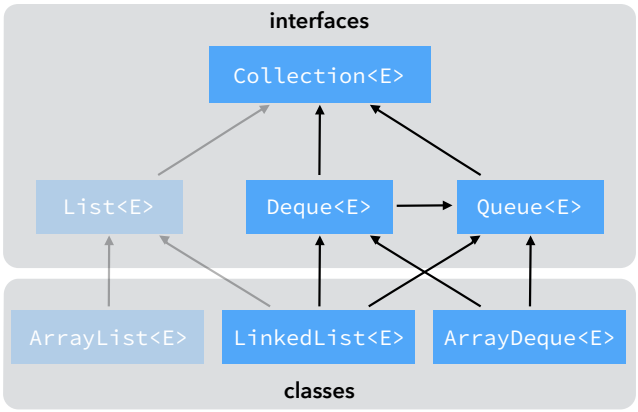
Producteur / consommateur

En informatique, 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.

Queues et deques bornés

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.

Queues de l'API Java



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 – 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.

Méthodes de 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.)

Méthodes d'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.

Méthodes de 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.

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 de la page suivante.

L'interface Deque

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

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'un 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.

Parcours des collections

Parcours d'une collection

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));
```

Pourquoi s'agit-il d'une mauvaise idée ?

Parcours via `get`

Utiliser la méthode `get` est une mauvaise idée car, dans le cas des listes chaînées, cette méthode 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 pour une boucle qui n'examine chaque élément qu'une seule fois…
Comment faire mieux ?

Parcours par boucle *for-each*

Les listes – et d'autres collections – peuvent heureusement être parcourues au moyen de la boucle *for-each*. La boucle d'impression peut donc se récrire ainsi :

```
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.

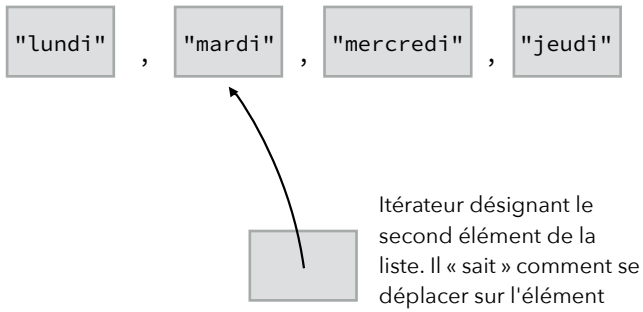
Comment est-ce possible ? Grâce à la notion d'itérateur !

Itérateur

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.

Itérateur



Itérateur désignant le second élément de la liste. Il « sait » comment se déplacer sur l'élément suivant.

Interface `Iterator`

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
}
```

Interface `Iterator`

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`.

Parcours par itérateur

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, notre boucle d'impression peut s'écrire également ainsi :

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


Itérateur / boucle *for-each*

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 100% 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 néanmoins 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 collection, 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.

Boucle *for-each*, `Iterable`

La boucle *for-each* peut en fait être utilisée sur n'importe quel objet qui implémente l'interface `Iterable`.
Cette interface, très simple, ne possède qu'une seule méthode, la méthode `iterator` vue précédemment :

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

Son paramètre de type représente le type des éléments parcourus par l'itérateur.

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.