Collections

CS-108

Michel Schinz

2025-03-18

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 est intéressant d'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.

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, files, « 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 clefs.

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** (*Java standard library*) — 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.

Le fait que les techniques de mise en œuvre des ensembles et des tables associatives soient similaires peut sembler surprenant au premier abord. Toutefois, à la réflexion, on se rend compte qu'il y a une très grande similarité entre ces deux types de collections :

- un ensemble peut être vu comme une table associative dont seules les clefs importent, les valeurs étant ignorées,
- une table associative peut être vue comme un ensemble de paires clef/valeur — pour peu que la valeur soit ignorée dans les tests d'égalité, les comparaisons et le hachage.

En pratique, la bibliothèque Java tire parti de ces similarités, puisque `HashSet` est mis en œuvre au moyen de `HashMap`, tandis que `TreeSet` est mise en œuvre au moyen de `TreeMap`.

13 Références

- Java Generics and Collections de Maurice Naftalin et Philip Wadler, en particulier :
  - le chapitre 15, *Lists* sur les listes,
  - le chapitre 13, *Sets* sur les ensembles,
  - le chapitre 16, *Maps* sur les tables associatives.
- 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 :
  - l'interface `java.util.Collection`,
  - les classes `java.util.Collections` et `java.util.Arrays`,
  - l'interface `java.util.Iterator`.
  - pour les listes :
    - l'interface `java.util.List`,
    - les classes `java.util.ArrayList` et `java.util.LinkedList`,
  - pour les ensembles :
    - l'interface `java.util.Set`,
    - les classes `java.util.HashSet` et `java.util.TreeSet`,
  - pour les tables associatives :
    - l'interface `java.util.Map`,
    - les classes `java.util.HashMap` et `java.util.TreeMap`.

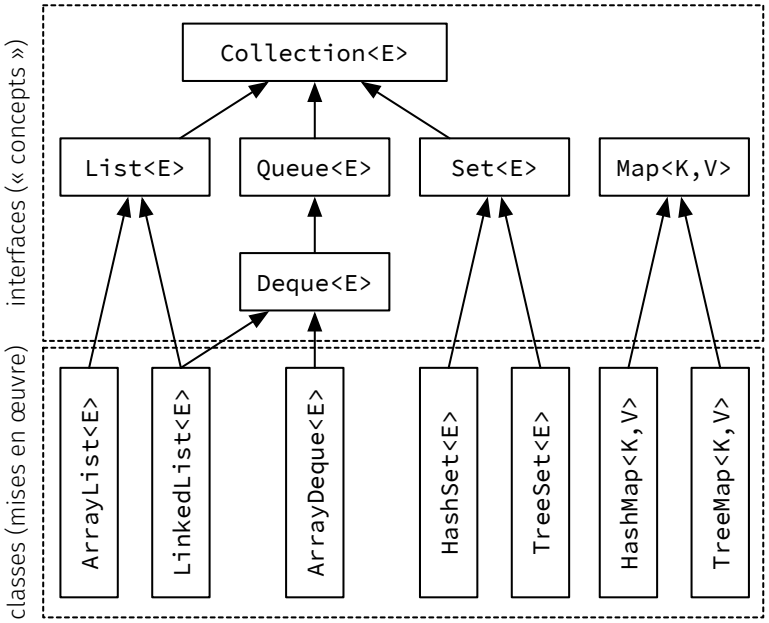

Fig. 1 : Hiérarchie partielle et simplifiée des collections Java

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`. 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 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, et son paramètre de type `E` (pour *element*) représente 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.

3.2.1 Méthodes de consultation

Les méthodes de consultation ci-dessous permettent d'obtenir différentes informations au sujet de la collection :

- `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,

12.5 Parcours

L'interface `Map` n'étend malheureusement pas l'interface `Iterable`, et il n'est donc pas possible de parcourir directement les paires clef/valeur d'une table associative au moyen d'un itérateur.

Il est par contre possible de parcourir ces paires clef/valeur par l'intermédiaire de la vue fournie par `entrySet` :

```
Map<String, Integer> m = /* ... */;
for (Map.Entry<String, Integer> e: m.entrySet())
  System.out.println(e.getKey() + "->" + e.getValue());
```

D'autre part, l'interface `Map` offre la méthode `forEach`, qui n'est pas celle de `Iterable` mais qui a le même but. Elle prend en argument un consommateur de type `BiConsumer<K,V>`, c-à-d une fonction à deux arguments — ici, la clef et la valeur — ne retournant rien.

La méthode `forEach` est destinée à être utilisée avec une lambda, et offre une manière particulièrement agréable de parcourir les paires clef/valeur d'une table associative. Par exemple, la boucle ci-dessus peut s'écrire plus simplement ainsi :

```
Map<String, Integer> m = /* ... */;
m.forEach((k, v) -> System.out.println(k + "->" + v));
```

Attention : comme pour les ensembles, l'ordre de parcours dépend de la mise en œuvre utilisée. Avec `TreeMap`, le parcours se fait par ordre croissant des clefs, tandis qu'avec `HashMap`, le parcours se fait dans un ordre quelconque.

12.6 Mises en œuvre des tables associatives

La bibliothèque Java offre deux mises en œuvre principales des tables associatives, `TreeMap` et `HashMap`. Comme la similarité de leurs noms l'indique, ces mises en œuvres sont basées sur les mêmes techniques que les classes `TreeSet` et `HashSet`. Dès lors, elles ont les mêmes caractéristiques principales, à savoir :

- `TreeMap` exige que ses *clefs* soient comparables, et utilise cette caractéristique pour offrir les opérations principales en O(log n), tout en garantissant que les paires clef/valeur sont parcourues par ordre croissant des clefs,
- `HashMap` exige que ses *clefs* soient hachables, et utilise cette caractéristique pour offrir les opérations principales en O(1), mais ne donne aucune garantie quant à l'ordre de parcours des paires clef/valeur.

Notez que seules les clefs doivent être comparables (pour `TreeMap`) ou hachables (pour `HashMap`), aucune exigence n'est placée sur les valeurs d'une table associative.

- `K getKey()`, retourne la clef de la paire,
- `V getValue()`, retourne la valeur de la paire.

De plus, elle offre une méthode optionnelle permettant de modifier la valeur associée à la clef :

- `void setValue(V v)`, remplace la valeur de la paire par celle donnée.

12.3 Tables immuables

Comme les interfaces `List` et `Set`, l'interface `Map` possède une famille de méthodes statiques nommées `of` et permettant de créer des tables associatives immuables. Les versions à 0, 1 et 2 arguments sont :

- `<K, V> Map<K, V> of()`, qui retourne une table associative immuable vide,
- `<K, V> Map<K, V> of(K k, V v)`, qui retourne une table associative immuable contenant uniquement la paire clef/valeur donnée,
- `<K, V> Map<K, V> of(K k1, V v1, K k2, V v2)`, qui retourne une table associative immuable contenant les deux paires clef/valeur données.

Des méthodes similaires prenant jusqu'à 10 paires clef/valeur existent.
De plus, l'interface `Map` offre une méthode statique permettant de construire une table associative immuable contenant les mêmes éléments qu'une autre table donnée :

- `<K,V> Map<K,V> copyOf(Map<K,V> m)`, retourne une table associative immuable contenant les mêmes éléments que la table donnée.

12.4 Tables non modifiables

Tout comme pour les listes, la classe `Collections` offre une méthode permettant d'obtenir une vue non modifiable sur une table associative :

- `<K,V> Map<K, V> unmodifiableMap(Map<K, V> m)`

Comme d'habitude, il faut prendre garde au fait qu'il s'agit d'une vue et que toute éventuelle modification à la table sous-jacente sera répercutée sur la vue ! Celle-ci est donc non modifiable mais pas forcément immuable.

Règle des tables associatives immuables : Pour obtenir une table associative immuable à partir d'une table associative quelconque, utilisez la méthode `copyOf` de l'interface `Map`.

- `boolean containsAll(Collection<E> c)`, retourne vrai ssi la collection contient tous les éléments de la collection donnée.

3.2.2 Méthodes d'ajout

Les méthodes d'ajout ci-dessous permettent d'ajouter un ou plusieurs éléments à une collection :

- `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.

3.2.3 Méthodes de suppression

Les méthodes de suppression permettent de supprimer un ou plusieurs éléments de la collection :

- `void clear()`, supprime tous les éléments de la collection,
- `boolean remove(Object e)`, supprime l'élément donné, s'il se trouve dans la collection ; le type de l'argument devrait être `E` mais est `Object` pour des raisons historiques,
- `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 donné,
- `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.

3.2.4 Lambdas

La méthode `removeIf` utilise l'interface `Predicate` qui représente un prédicat logique, c-à-d une fonction retournant une valeur booléenne. La définition simplifiée de cette interface est :

```java
public interface Predicate<T> {
  public abstract boolean test(T t);
}
```

La méthode `test` doit retourner vrai si et seulement si l'argument qu'on lui donne satisfait le prédicat. Par exemple, un prédicat déterminant si un entier est nul pourrait être défini ainsi :

```java
public class IsZero implements Predicate<Integer> {
  @Override
  public boolean test(Integer i) { return i == 0; }
}
```

Une fois cette classe définie, on pourrait l'utiliser pour supprimer d'une collection d'entiers `c` tous les entiers nuls au moyen de la méthode `removeIf` :

```java
Collection<Integer> c = /* ... */;
c.removeIf(new IsZero());
```

En pratique, devoir définir une nouvelle classe uniquement pour pouvoir supprimer tous les éléments nuls d'une collection est très lourd. Java offre toutefois la notion de **lambda**, qui sera le sujet d'une leçon ultérieure, qui permet de considérablement simplifier le code ci-dessus. Ainsi, au moyen d'une lambda, il est possible de supprimer tous les éléments nuls d'une liste d'entiers en écrivant simplement :

```java
Collection<Integer> c = /* ... */;
c.removeIf(i -> i == 0);
```

sans devoir définir une classe comme `IsZero`.

De nombreuses méthodes des collections sont destinées à être utilisées avec des lambdas, et dans le reste de cette leçon elles seront signalées au moyen de la lettre grecque lambda entre crochets : [λ].

3.2.5 Méthodes optionnelles

Le fait que l'interface `Collection` possède des méthodes permettant de modifier le contenu de la collection, comme `add` ou `remove`, pourrait laisser penser qu'une collection peut toujours être modifiée.

En réalité, 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 :

12.1.4 Suppression

Les méthodes de suppression ci-dessous sont optionnelles :

- `void clear()`, vide la table en supprimant toutes les associations clef/valeur,
- `V remove(Object k)`, supprime la clef donnée de la table ainsi que la valeur qui lui était associée ; retourne cette dernière ou `null` si la clef n'était pas présente,
- `boolean remove(Object k, Object v)`, supprime la clef donnée de la table ssi elle est associée à la valeur donnée ; retourne vrai ssi la table a été modifiée en conséquence.

Comme précédemment, l'utilisation de `Object` en lieu et place de `K` ou `V` est une erreur historique.

12.1.5 Vues sur les clefs et valeurs

Les méthodes ci-dessous permettent d'obtenir des vues sur les clefs, les valeurs ou les paires clefs/valeurs :

- `Set<K> keySet()`, retourne une vue sur l'ensemble des clefs de la table,
- `Collection<V> values()`, retourne une vue sur les valeurs de la table,
- `Set<Map.Entry<K, V» entrySet()`, retourne une vue sur l'ensemble des associations clef/valeur de la table.

Si la table est modifiable, ces vues le sont également et les modifications qu'on y apporte sont reportées sur la table — et inversement.

12.2 L'interface `Map.Entry`

L'interface `Map.Entry` — imbriquée statiquement dans l'interface `Map` — représente une association entre une clef et une valeur, aussi appelée paire clef/valeur.

Tout comme l'interface `Map`, l'interface `Entry` est générique et prend deux paramètres de type : le type `K` de la clef et le type `V` de la valeur qui lui est associée.

```java
public interface Map<K, V> {
  public static interface Entry<K, V> {
    // ... méthodes
  }
}
```

L'interface `Map.Entry` offre deux méthodes permettant respectivement d'accéder à la clef et à la valeur de la paire clef/valeur qu'elle représente :

12.1.2 Ajout et modification

Les méthodes d'ajout/modification ci-dessous sont comme d'habitude optionnelles :

- `V put(K k, V v)`, associe la valeur donnée avec la clef donnée et retourne la valeur qui lui était associée ou `null` s'il n'y en avait aucune,
- `V putIfAbsent(K k, V v)`, si la clef donnée n'est pas encore associée à une valeur, l'associe à la valeur donnée et retourne `null` ; sinon, retourne la valeur associée à la clef,
- `void putAll(Map<K, V> m)`, copie toutes les associations clef/valeur de la table donnée dans la table à laquelle la méthode est appliquée.
- `V computeIfAbsent(K k, Function<K,V> f)`, si la clef donnée n'est pas encore associée à une valeur, l'associe au résultat de la fonction appliquée à la clef, et retourne cette valeur ; sinon, retourne la valeur déjà associée à la clef [λ],
- `V merge(K k, V v, BiFunction<V,V,V> f)`, si la clef donnée n'est pas encore associée à une valeur, lui associe la valeur donnée ; sinon, remplace la valeur qui lui est actuellement associée par la fonction appliquée à la valeur actuelle et la valeur donnée ; retourne la valeur associée à la clef [λ].

D'autres méthodes d'ajout/modification existent, parmi lesquelles `compute` et `computeIfPresent` mais sont plus rarement utiles et donc pas décrites ici.

12.1.3 Remplacement

Les méthodes de remplacement ci-dessous sont optionnelles :

- `V replace(K k, V v)`, si la table contient une valeur associée à la clef donnée, la remplace par la valeur donnée et retourne l'ancienne valeur ; sinon, ne modifie pas la table et retourne `null`,
- `boolean replace(K k, V v1, V v2)`, si la table associe actuellement la clef donnée à la première valeur donnée, lui associe la seconde valeur donnée et retourne vrai ; sinon, ne modifie pas la table et retourne faux,
- `void replaceAll(BiFunction<K,V,V> f)`, remplace chaque *valeur* de la table par le résultat de l'application de la fonction passée à la paire clef/valeur à laquelle elle appartient [λ].

- **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`, ou encore `List.of`), ou
- obtenir ce qu'on appelle 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.

4 Listes

Une **liste** (*list*) est une collection ordonnée 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].

Quelques cas particuliers des listes se rencontrent assez fréquemment pour avoir un nom propre : les piles, les files 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 :

[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).

```
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 d'une 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 l'exception `IndexOutOfBoundsException` si l'index est invalide. Comme toutes les méthodes modifiant une collection, elles sont optionnelles.

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,


```
String java = "java";
for (int i = 0; i < java.length(); ++i)
  System.out.print(morse.get(java.charAt(i)) + "␣");
```

12.1 L'interface `Map`

Le concept de table associative est représenté dans l'API Java par l'interface `Map` du paquetage `java.util`. Cette interface est générique et prend deux paramètres de type nommés `K` et `V` qui représentent respectivement le type des clefs (*keys*) et celui des valeurs (*values*) :

```
public interface Map<K, V> {
  // ... méthodes
}
```

A noter que contrairement aux interfaces `List` et `Set`, l'interface `Map` n'hérite pas de `Collection`, ni de `Iterable`.

Les principales méthodes de cette interface sont présentées ci-après, parfois avec un type simplifié pour des raisons pédagogiques.

12.1.1 Consultation

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

- `boolean isEmpty()`, retourne vrai ssi la table associative est vide.
- `int size()`, retourne le nombre d'associations clef/valeur contenues dans la table associative.
- `boolean containsKey(Object k)`, retourne vrai ssi la table contient la clef donnée.

Le type de l'argument de la dernière méthode devrait être `K`, mais est `Object` pour de malheureuses raisons historiques.

Les méthodes ci-dessous permettent d'obtenir la valeur associée à une clef et sont probablement les méthodes les plus utilisées des tables associatives :

- `V get(Object k)`, retourne la valeur associée à la clef donnée, ou `null` si cette clef n'est pas présente dans la table.
- `V getOrDefault(Object k, V d)`, retourne la valeur associée à la clef donnée, ou la valeur par défaut donnée si la clef n'est pas présente dans la table.

Là aussi, le type de la clef devrait être `K` et pas `Object`.

`TreeSet` doit son nom au fait qu'elle stocke les éléments de l'ensemble dans un **arbre binaire de recherche** (*binary search tree*), tandis que `HashSet` doit le sien au fait qu'elle les stocke dans une **table de hachage** (*hash table*). Ces concepts seront examinés ultérieurement, mais il importe déjà de savoir que grâce à eux, `TreeSet` peut mettre en œuvre les opérations principales sur les ensembles (ajout, test d'appartenance, etc.) en $O(\log n)$, tandis que `HashSet` fait encore mieux et les met en œuvre en $O(1)$, ce qui est remarquable !

Malgré ses performances moindres, `TreeSet` offre néanmoins un avantage par rapport à `HashSet`, déjà mentionné plus haut : étant donné qu'elle stocke les éléments de l'ensemble sous forme triée, ceux-ci sont parcourus dans cet ordre. `HashSet`, quant à elle, ne donne aucune garantie concernant l'ordre de parcours. Dès lors, malgré ses meilleures performances, `HashSet` n'est pas forcément toujours préférable à `TreeSet`.

> **Règle `HashSet` / `TreeSet` :** Utilisez `HashSet` comme mise en œuvre des ensembles en Java, sauf lorsqu'il est utile de parcourir les éléments en ordre croissant, auquel cas vous pourrez lui préférer `TreeSet`.

11 Tables associatives

Une **table associative** (*map*) ou **dictionnaire** (*dictionary*) est une collection qui associe des **valeurs** (*values*) à des **clefs** (*keys*).

Par exemple, l'index d'un livre est une table associative qui associe à différents mots (les clefs) la liste des numéros de pages sur lesquelles ce mot apparaît (les valeurs).

En informatique, un tableau — ou une liste — peut être vu comme un cas particulier d'une table associative dont les clefs sont les entiers compris entre 0 et la taille du tableau, et les valeurs sont les éléments du tableau.

12 Tables associatives en Java

Dans la bibliothèque Java, le concept de table associative est représenté par l'interface `Map` et parmi les mises en œuvres figurent les classes `HashMap` et `TreeMap`.

L'extrait de programme ci-dessous illustre leur utilisation en traduisant en code Morse le mot *java*. Pour ce faire, une table associant leur encodage en Morse (les valeurs) aux caractères de l'alphabet (les clefs) est tout d'abord construite. Cela fait, la chaîne *java* est parcourue, caractère par caractère, et la traduction en Morse de chacun d'entre eux est obtenue de la table et affichée à l'écran :

```
Map<Character, String> morse =
  Map.of('a', ".-",
         'j', ".---",
         'v', "...-");
// ... à compléter avec le reste de l'alphabet morse
```


- `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 [λ].

5.1.4 Sous-liste

Finalement, l'interface `List` offre une méthode pour obtenir ce que l'on nomme une vue sur une portion d'une liste comprise entre deux index :

- `List<E> subList(int b, int e)`, retourne une vue sur la sous-liste composée des éléments dont les index sont compris entre `b` (inclusif) et `e` (exclusif).

Il faut faire très attention au fait que cette méthode ne retourne *pas* une copie de la portion de la liste originale comprise entre les index `b` et `e`, mais bien une **vue** (*view*) sur cette portion.

Cela implique que toutes les modifications effectuées sur la liste originale sont répercutées sur la vue, et inversement. Ainsi, on peut par exemple supprimer tous les éléments compris entre les index 4 (inclusif) et 6 (exclusif) d'une liste `l` en effaçant le contenu d'une vue sur ces éléments :

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

5.1.5 Tri et mélange

L'interface `List` offre une méthode permettant de trier les éléments d'une liste :

- `void sort(Comparator<E> c)`, trie les éléments de la liste au moyen du comparateur donné [λ] ; si celui-ci est nul, l'ordre naturel des éléments est utilisé.

Les notions de comparateur et d'ordre naturel seront examinées dans une leçon ultérieure.

L'interface `List` n'offre aucune méthode permettant de mélanger les éléments de la liste, mais la classe `Collections` (avec un *s* !) en offre une :

- `<T> void shuffle(List<T> l, Random r)`, mélange aléatoirement les éléments de la liste donnée, en utilisant le générateur de valeurs aléatoires donné.

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`, ce qui est logique étant donné que les tableaux ne peuvent être redimensionnés.

5.1.7 Listes immuables

L'interface `List` offre une méthode statique — en réalité un ensemble de méthodes surchargées — permettant de construire une liste immuable constituée d'éléments quelconques :

- `<E> List<E> of(E... es)`, retourne une liste immuable contenant les éléments donnés.

De plus, elle offre une méthode statique permettant de construire une liste immuable contenant les mêmes éléments qu'une collection donnée :

- `<E> List<E> copyOf(Collection<E> c)`, retourne une liste immuable contenant les éléments de la collection donnée.

Finalement, la classe `Collections` offre une méthode de construction de liste immuable contenant un élément unique apparaissant plusieurs fois :

- `<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 :

Règle des ensembles immuables : Pour obtenir un ensemble immuable à partir d'un ensemble quelconque, utilisez la méthode `copyOf` de l'interface `Set`.

10.4 Parcours

Etant donné que l'interface `Set` implémente (indirectement) l'interface `Iterable`, les éléments d'un ensemble peuvent être parcourus au moyen d'un itérateur ou de la boucle *for-each*, comme ceux d'une liste.

Attention : contrairement aux éléments d'une liste, les éléments d'un ensemble ne sont pas ordonnés. Dès lors, l'ordre de parcours des éléments d'un ensemble dépend de la mise en œuvre utilisée :

- `TreeSet` les parcourt dans l'ordre croissant,
- `HashSet` les parcourt dans un ordre arbitraire, qui peut changer d'une exécution à l'autre d'un programme, et même être différent entre deux instances contenant les *mêmes* éléments !

10.5 Mises en œuvres des ensembles

On l'a dit, la bibliothèque Java offre deux mises en œuvre principales des ensembles. Avant de les examiner et de les comparer, posons-nous toutefois la question de leur utilité. Ne serait-il pas possible d'utiliser simplement des listes (sans doublons) afin de représenter les ensembles ?

En théorie, oui, et cela est même relativement simple sachant que la seule différence entre une liste « normale » et une liste représentant un ensemble est que cette dernière ne contient pas de doublons. Pour garantir cette propriété, il suffit d'utiliser la méthode `contains` fournie par l'interface `Collection` afin de s'assurer de l'absence d'un élément avant de l'ajouter à la liste.

Malheureusement, représenter un ensemble au moyen d'une liste est coûteux, car les principales opérations (ajout, test d'appartenance, etc.) ont une complexité de $O(n)$. Intuitivement, cela est dû au fait que chacune de ces opérations nécessite un parcours de la totalité des éléments de la liste.

D'autres mises en œuvre, plus efficaces, sont donc fournies dans la bibliothèque Java. Néanmoins, chacune d'entre elles exige que certaines opérations puissent être effectuées sur les éléments de l'ensemble, afin de pouvoir les organiser en mémoire. Les deux mises en œuvre que nous examinerons, et leurs exigences concernant les éléments, sont :

- `TreeSet`, qui exige que les éléments de l'ensemble puissent être triés,
- `HashSet`, qui exige que les éléments de l'ensemble puissent être « hachés », notion examinée dans une leçon ultérieure.

10.1 L'interface `Set`

Le concept d'ensemble est représenté dans l'API Java par l'interface `Set` du paquetage `java.util`. Tout comme `Collection` — dont elle hérite — cette interface est générique et son paramètre de type représente le type des éléments de l'ensemble :

```
public interface Set<E> extends Collection<E> { }
```

Cette interface n'ajoute aucune méthode à celles héritées de `Collection`, son seul but étant d'offrir un type distinct pour les ensembles.

Aux principales opérations sur les ensembles mathématiques correspond une méthode de l'interface `Set`, avec une différence importante : contrairement aux opérations mathématiques, les méthodes modifient l'ensemble auquel on les applique. La table ci-dessous donne les méthodes correspondant aux opérations mathématiques.

Opération	Méthode
union ($\cup$)	`addAll`
test d'appartenance ($\in$?)	`contains`
test d'inclusion ($\subseteq$?)	`containsAll`
différence ($\setminus$)	`removeAll`
intersection ($\cap$)	`retainAll`

10.2 Ensembles immuables

Tout comme pour les listes, l'interface `Set` offre une méthode statique permettant de construire un ensemble immuable :

- `<E> Set<E> of(E... es)`, retourne un ensemble immuable contenant les éléments donnés.

De plus, elle offre une méthode statique permettant de construire un ensemble immuable contenant les mêmes éléments qu'une collection donnée :

- `<E> Set<E> copyOf(Collection<E> c)`, retourne un ensemble immuable contenant les éléments de la collection donnée.

10.3 Ensembles non modifiables

Tout comme pour les listes, la classe `Collections` offre une méthode permettant d'obtenir une vue non modifiable sur un ensemble :

- `<E> Set<E> unmodifiableSet(Set<E> s)`

Comme d'habitude, il faut prendre garde au fait qu'il s'agit d'une vue et que toute éventuelle modification à l'ensemble sous-jacent sera répercutée sur la vue ! Celle-ci est donc non modifiable mais pas forcément immuable.

Dès lors, comme avec les listes, pour obtenir une version immuable d'un ensemble, il faut copier celui-ci (aussi profondément que nécessaire).


```
List<Integer> list = new ArrayList<>();
list.add(1);
List<Integer> view = Collections.unmodifiableList(list);
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 utiliser la méthode statique `copyOf` de l'interface `List`, qui retourne une copie immuable de la liste qu'on lui donne en argument.

> **Règle des listes immuables** : Pour obtenir une liste immuable à partir d'une liste quelconque, utilisez la méthode `copyOf` de l'interface `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. Dans cette table, n est le nombre d'éléments de la liste.

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, files 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 files 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 **file** (*queue*) est une liste dont les éléments sont toujours insérés à une extrémité et retirés de l'autre,
- une « **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.

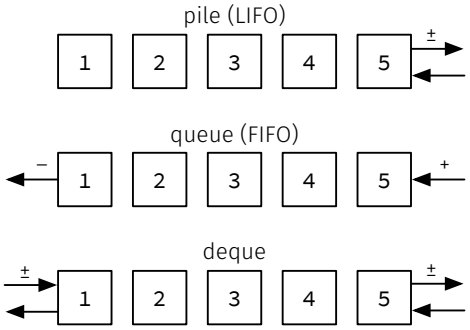

Fig. 2 : Piles, files et deques

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

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

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

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

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 Ensembles

Un **ensemble** (*set*) est une collection non ordonnée d'objets dans laquelle un objet peut apparaître au plus une fois. Cette notion d'ensemble correspond à la notion mathématique.

10 Ensembles en Java

Dans la bibliothèque Java, le concept d'ensemble est représenté par l'interface `Set` et parmi les mises en œuvres figurent les classes `HashSet` et `TreeSet`.

L'extrait de programme ci-dessous illustre leur utilisation en créant tout d'abord l'ensemble des voyelles non accentuées de l'alphabet latin puis en l'utilisant pour déterminer le nombre de voyelles que comporte le mot anglais *deinstitutionalization* :

```
Set<Character> vowels = Set.of('a','e','i','o','u','y');

String word = "deinstitutionalization";
int vowelCount = 0;
for (int i = 0; i < word.length(); ++i) {
  if (vowels.contains(word.charAt(i)))
    vowelCount += 1;
}
System.out.println("Le␣mot␣" + word + "␣contient␣"
                   + vowelCount + "␣voyelles.");
```

- `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 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. Cette méthode offre un troisième moyen de parcourir les éléments d'une collection, que nous examinerons dans la leçon consacrée aux lambdas.

Avec une telle file, le producteur ne place une valeur dans la file 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 file à son rythme.

7 Piles, files et deques en Java

Dans la bibliothèque Java, les files 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 file 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 file 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 file :

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

(Par tête de file on entend l'extrémité de la file 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 file :

- `boolean add(E e)`, ajoute l'élément donné à la file 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 file et retourne vrai si cela a été possible — c-à-d si la file 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 file :

- `E remove()` : supprime et retourne l'élément en tête de file, ou lève une exception si celle-ci est vide,
- `E poll()` : supprime et retourne l'élément en tête de file s'il existe, ou ne fait rien et retourne `null` si la file 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 file 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 file 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 file ou une « deque », utilisez `ArrayDeque`. Pour représenter une liste dans toute sa généralité, utilisez `ArrayList` si les opérations d'indexation (`get`, `set`) dominent ou si les ajouts/suppressions sont toujours faits à la fin de la liste, sinon `LinkedList`.

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,