Collections : Tables associatives

Michel Schinz

2020-03-16

1 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.

2 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 = new HashMap<>();
morse.put('a', ".-");
morse.put('j', ".---");
morse.put('v', "...-");
// … idem pour les autres lettres et la ponctuation.

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

2.1 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.

2.1.1 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`.

2.1.2 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,
- `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.

Notez que les méthodes `computeIfAbsent` et `merge` prennent des arguments de type `Function` et `BiFunction`. Ces types sont ceux d'interfaces représentant des fonctions — au sens mathématique — à un ou deux arguments. De loin la manière la plus simple de spécifier ces fonctions consiste à utiliser des *lambdas*, concept qui sera introduit dans une leçon ultérieure.

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

2.1.3 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.

2.1.4 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.

2.1.5 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.

2.2 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.

```
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 :

- `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.

2.3 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.

2.4 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`.

2.5 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());
```

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

2.6 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.

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`.

3 Références

- *Java Generics and Collections* de Maurice Naftalin et Philip Wadler, en particulier :
  - le chapitre 16, *Maps* sur les tables associatives.
- la documentation de l'API Java, en particulier les classes et interfaces suivantes :
  - l'interface `java.util.Map`,
  - la classe `java.util.HashMap`,
  - la classe `java.util.TreeMap`.