Les collections

Christina Boura, Stéphane Lopes

christina.boura@uvsq.fr,
stephane.lopes@uvsq.fr

4 avril 2016

UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN-EN-YVELINES

Agenda du jour

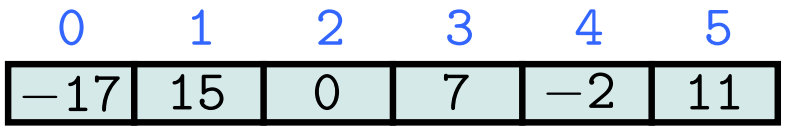
Définition

Une collection est un objet qui regroupe plusieurs éléments en une seule unité.

- Les collections peuvent être utilisées pour stocker et manipuler des données et pour transmettre des données d'une méthode à une autre.
- Une collection regroupe généralement des objets du même type.

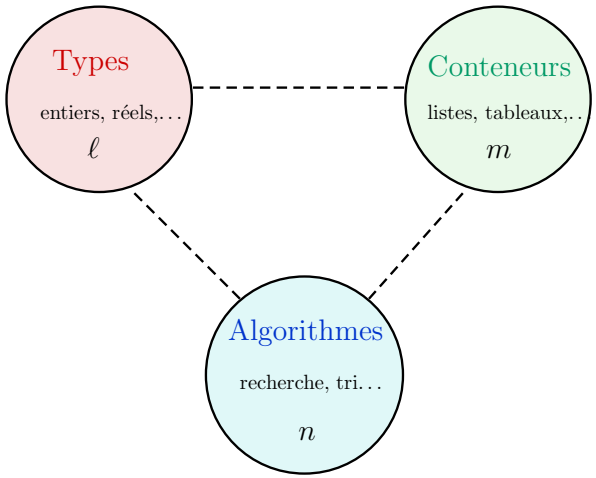
Les tableaux pour gérer des objets du même type

Les tableaux peuvent être utilisés pour stocker un groupe d'éléments du même type (type primitive ou objets). Mais :



- Les tableaux ne permettent pas l'allocation dynamique ; leur taille est fixe et ne peut pas changer.
- Leur structure est trop simple sans fonctionnalités avancées.

Motivation



- $\ell \times m$ versions différentes du même algorithme si l'on veut coder tous les cas.
- Pour n algorithmes, $\ell \times m \times n$ programmes.
- La généricité permet de paramétrer par rapport aux types : $m \times n$ programmes.
- Si on a une abstraction de la notion de collection (un algorithme fonctionne sur toute collection) : n programmes.

Avantages

Les collections permettent de :

- Réduire l'effort lors de l'implémentation.
- Améliorer les performances des programmes ainsi que leur qualité.
- Réduire le coût de conception de nouvelles API.
- Réduire l'effort d'apprentissage et d'utilisation des API.
- Réutiliser des logiciels.

Evolution historique de la librairie des Collections

Avant `JDK 1.2`

- Pas d'architecture complète.
- Le traitement des objets du même type se faisait à l'aide des `tableaux` et des classes `Hashtable` et `Vector`.

`JDK 1.2` : Introduction d'un cadre reposant sur trois composants :

- les interfaces
- les implémentations
- les algorithmes

À partir de la version `JDK 5.0` :

- Contrôle de type amélioré avec les génériques (Generics).
- Meilleure utilisation de la librairie de collections.

Les Interfaces

Une interface définit un comportement qui doit être implémenté par une classe, sans implémenter ce comportement.

C'est un ensemble de méthodes abstraites, et de constantes.

Différence entre interfaces et classes abstraites

- Une interface n'implémente aucune méthode, quant une classe abstraite le peut.
- Une interface peut dériver de plusieurs autres interfaces quant une classe abstraite peut implémenter plusieurs interfaces mais n'a qu'une seule super-classe.
- Des classes non liées hiérarchiquement peuvent implémenter la même interface.

Les interfaces pour gérer les collections

- Les interfaces sont utilisées pour manipuler des collections et les transmettre d'une méthode à une autre.
- Les interfaces permettent de manipuler les collections indépendamment des différentes implémentations.
- Une implémentation a la possibilité de ne pas supporter toutes les méthodes de modification de l'interface (lancement de l'exception `UnsupportedOperationException`)
- Les implémentations du JDK implémentent toutes les méthodes optionnelles.

Cadre pour les collections

Un cadre pour les collections (collection framework) est une architecture unifiée pour représenter et manipuler des collections.

Il comporte trois composants :

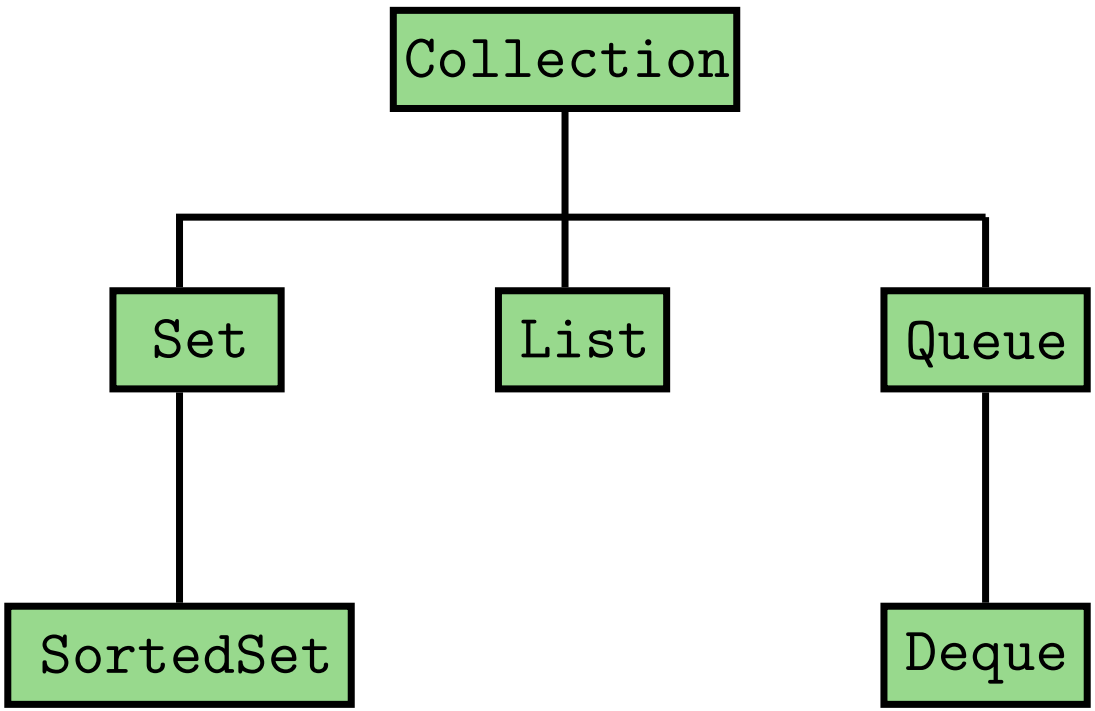
- des interfaces permettant de manipuler des collections indépendamment de leurs implémentations
- des implémentations représentant les structures de données proprement dites
- des algorithmes effectuant des traitements par l'intermédiaire des interfaces

Généralités

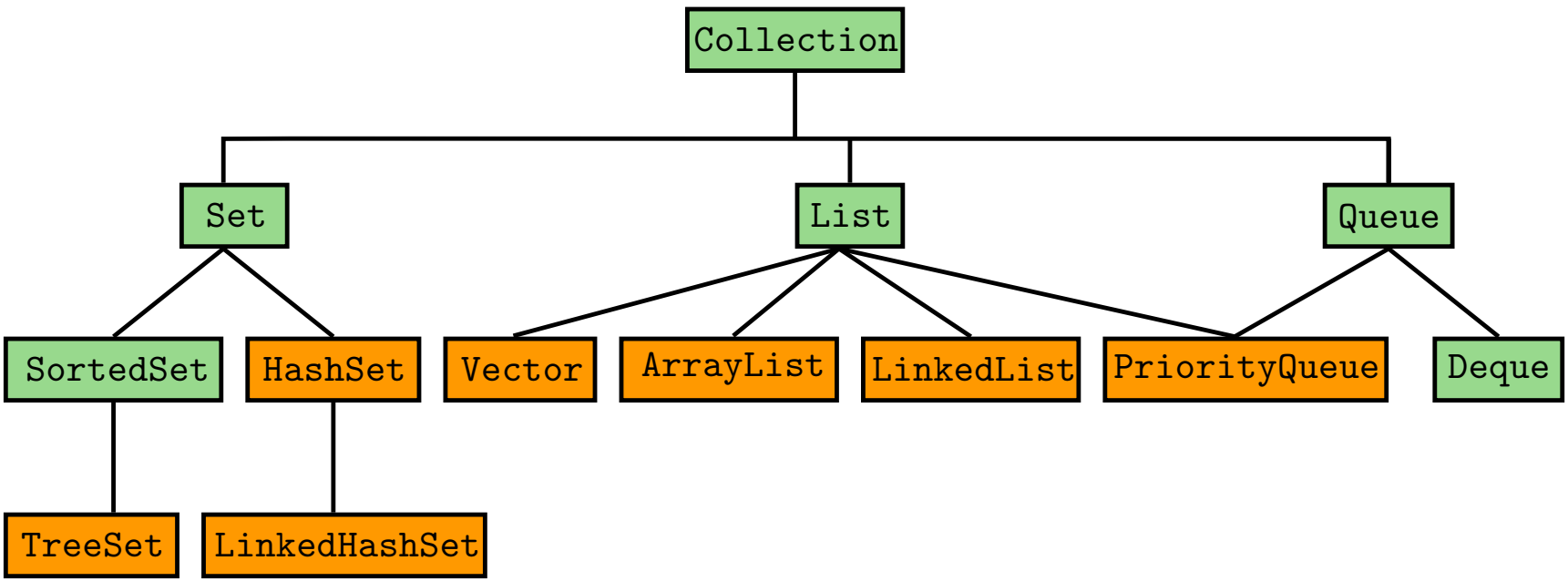
- Une collection Java ne peut pas contenir une donnée d'un type primitif, mais uniquement des objets.
- Les composants de la librairie de collections se trouvent dans le package `java.util`.
- Le découpage interface/implémentation repose sur le modèle de conception `Pont` (`Bridge`).

Agenda du jour

Hierarchie des Interfaces Collection



Hierarchie des Interfaces et des classes Collection



Les opérations basiques de l'interface Collection

- `int size()` : Renvoie le nombre d'éléments de la collection.
- `boolean contains(Object o)` : Retourne `true` si la collection contient l'élément en paramètre.
- `boolean add(E e)` : Ajoute l'élément en paramètre à la collection (optionnelle).
- `boolean remove(E e)` : Enlève l'élément en paramètre de la collection (optionnelle).
- `Iterator<E> iterator()` : Renvoie un iterator pour les éléments de cette collection.

Les opérations toArray de l'interface Collection

Les méthodes `toArray` font le lien entrer les collections et les anciens API qui travaillent sur les tableaux.

Elles permettent d'inserer les éléments d'une collection dans un tableau.

- `Object[] toArray()`
  - **Ex.** : `Object[] a = c.toArray();`
- `<T> T[] toArray(T[] a)`
  - **Ex.** : `String[] a = c.toArray(new String[0]);`

Les opérations de traitement par lot de l'interface Collection

Les opérations de traitement par lot (bulk operations) appliquent un traitement sur la collection entière.

- `boolean containsAll(Collection<?> c)` : Renvoie `true` si tous les éléments fournis en paramètre sont présents dans la collection.
- `boolean addAll(Collection<?> c)` : Ajouter tous les éléments de la collection fournie en paramètre dans la collection (optionnelle).
- `boolean removeAll(Collection<?> c)` : Supprimer tous les éléments fournis en paramètre de la collection (optionnelle).
- `boolean retainAll(Collection<?> c)` : Ne laisser dans la collection que les éléments fournis en paramètre ; les autres éléments sont supprimés (optionnelle).
- `void clear()` : Supprimer tous les éléments de la collection

Parcourir une collection : La boucle `for-each`

Le moyen le plus simple de parcourir une collection (ou un tableau) consiste à utiliser une boucle `for` spéciale

```
for(String element : uneCollectionDeChaines) {
}
```

- La boucle `for-each` ne permet pas de modifier la collection lors de l'itération.
- Utiliser cette boucle dans ces cas-ci.

La notion de l'itérateur

Un itérateur est un objet qui permet de parcourir tous les éléments d'une collection, sans avoir connaissance de son implémentation.

Exemple : Une collection de films `CollectionFilms`, qu'on souhaite rendre itérable.

```
public class CollectionFilms implements Iterable<Film> {

    private Vector<Film> films;

     public CollectionFilms() {
      film = new Vector<Film>();
     }

     public void add(Film film) {
      films.add(film);
     }

     public Iterator<Film> iterator() {
      return films.iterator();
     }
}
```

Suite de l'exemple

L'utilisateur peut utiliser la collection de la façon suivante :

```
CollectionFilms cfilms = new CollectionFilms();
Iterator<Film> filmIterator = cfilms.iterator();

while (filmIterator.hasNext()) {
 Film f = filmIterator.next();
 System.out.println(f.getName());
}
```

Ou encore, il peut utiliser la boucle `for-each` :

```
CollectionFilms cfilms = new CollectionFilms();
for (Film f : cfilms) {
 System.out.println(f.getName());
}
```

- L'utilisateur n'a pas besoin de savoir qu'on utilise une collection de type `Vector` pour implémenter notre collection.
- On peut changer l'implémentation pour utiliser des `LinkedList` ou des `ArrayList`, le code de l'utilisateur restera le même.

L'interface Iterator

```
public interface Iterator<E>
```

L'interface `Iterator` nécessite l'implémentation des méthodes suivantes :

- `boolean hasNext()`
- `Object next()`
- `void remove()`

L'opération `remove()` est optionnelle. Dans ce cas son implémentation pourrait être :

```
public void remove() {
throw new UnsupportedOperationException();
}
```

Parcourir une collection avec un itérateur

La méthode `next()` peut lever une exception de type `NoSuchElementException` si appelée alors que la fin du parcours des éléments est atteinte.

Pour éviter la levée de cette exception : appeler la méthode `hasNext()` :

```
Iterator iterateur = collection.iterator();
while (iterateur.hasNext()) {
    System.out.println("objet = "+iterator.next());
}
```

Supprimer un élément

La méthode `remove()` permet de supprimer l'élément renvoyé par le dernier appel à la méthode `next()`.

- Il est impossible d'appeler la méthode `remove()` sans un appel correspondant à `next()`.
- On ne peut donc pas appeler la méthode `remove()` deux fois de suite.

```
Iterator iterateur = collection.iterator();
if (iterateur.hasNext()) {
    iterateur.next();
    iterateur.remove();
}
```

L'interface Set

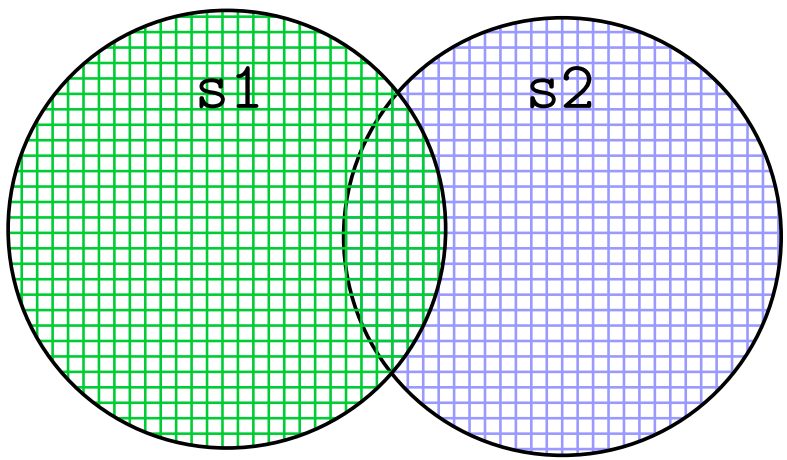
- l'interface `Set` représente une collection dont tous les éléments sont distincts.
- Ce type de collection permet de modéliser le concept mathématique d'ensemble.
- L'interface `Set` n'ajoute aucune méthode à l'interface `Collection`.
- Deux ensembles sont égaux s'ils contiennent les mêmes éléments.

Remarque : À la différence des interfaces `List` et `Collection`, l'ajout d'un élément dans un `Set` peut échouer, si cet élément s'y trouve déjà.

- Pour cette raison la méthode `add()` retourne un booléen.

Méthodes de traitement par lot de l'interface Set

- `s1.containsAll(s2)` renvoie `true` si $s2 \subseteq s1$
- `s1.addAll(s2)` transforme `s1` en $s1 \cup s2$
- `s1.retainAll(s2)` transforme `s1` en $s1 \cap s2$
- `s1.removeAll(s2)` transforme `s1` en $s1 \setminus s2$



L'interface SortedSet

```
public interface SortedSet<E> extends Set<E>
```

- L'interface `SortedSet` hérite de l'interface `Set`.
- Tous les objets de cet ensemble sont automatiquement triés dans un ordre que nous devons préciser.
- L'itération sur les éléments d'un `SortedSet` se fait dans l'ordre croissant.

Les opérations héritées de `Set` fonctionnent à l'identique sauf :

- l'itérateur respecte l'ordre
- `toArray` conserve l'ordre

Comment ordonner des objets ? (I)

Problème : Comment ordonner des objets selon leur ordre naturel :

- lexicographique pour les chaînes,
- chronologique pour les dates,
- ...

Solution en Java : Implémenter l'interface `Comparable`

```
public interface Comparable<T> {
      public int compareTo(<T> o);
}
```

La méthode compareTo

L'interface `Comparable` a une unique méthode

`compareTo(T o)`

qui renvoie un entier `n` :

- `n < 0` : L'objet o est plus petit que notre objet.
- `n = 0` : L'objet o est égal à notre objet.
- `n > 0` : L'objet o est plus grand que notre objet.

- Si l'argument n'est pas du bon type, `compareTo` doit lancer l'exception `ClassCastException`.

Comment ordonner des objets ? (II)

Problème : Comment ordonner des objets selon un ordre particulier (différent de l'ordre naturel) ?

Solution en Java : Fournir un comparateur, ç.-à-d. une instance d'une classe implémentant l'interface `Comparator`.

```
public interface Comparator<T> {
      public int compare(T o1, T o2);
}
```

- La méthode `compare` qui fonctionne de la même façon que la méthode `compareTo` de `Comparable`.
- Si l'argument n'est pas du bon type, `compare` doit lancer l'exception `ClassCastException`.

Exemple

Implémentation de la méthode compareTo()

Définir un ordre naturel un ensemble d'élèves :

- Ordre alphabétique Nom Prenom

```
public  class ComparableEleves  implements Comparable<Eleve> {

    // Déclaration des variables
    // Autres méthodes de la classe

    // méthode imposée par l'interface Comparable
    public  int compareTo(Eleve eleve) {

        if (getNom().equals(eleve.getNom())) {
            return getPrenom().compareTo(eleve.getPrenom()) ;
        }  else {
            return getNom().compareTo(eleve.getNom()) ;
        }
    }
}
```

Exemple

Implémentation de la méthode compare()

Définir un ordre quelconque pour un ensemble d'élèves :

- Comparer deux éléves selon leur âge.

```
class ComparatorEleves implements Comparator<Eleve> {

    public int compare(Eleve eleve1, Eleve eleve2) {
        return eleve1.getAge() - eleve2.getAge();
    }
}
```

Méthodes de SortedSet

- `E first()` : Renvoie le premier élément de l'ensemble.
- `E last()` : Renvoie le dernier élément de l'ensemble.
- `SortedSet headSet(E toElement)` : Renvoie un sous-ensemble contenant les premiers éléments de l'ensemble à partir de celui fourni en paramètre exclus.
- `SortedSet tailSet(E fromElement)` : Renvoie un sous-ensemble contenant les derniers éléments de l'ensemble à partir de celui fourni en paramètre inclus.
- `SortedSet subSet(E fromElement, E toElement)` : Renvoie un sous-ensemble dont les bornes sont ceux fournis en paramètre : `fromElement` est inclus et `toElement` est exclus.
- `Comparator<? super E> comparator()` : Renvoie l'instance de type `Comparator` associée à l'ensemble.

L'interface `List`

Une collection de type `List` est une collection ordonnée d'éléments.

- Les doublons sont autorisés.
- Ça peut être vu comme des "tableaux extensibles à volonté".
- L'interface `List` possède des opérations pour :
  - Accéder aux éléments d'une liste par leurs indices.
  - Renvoyer l'indice d'un objet que l'on recherche.
  - Étendre la sémantique des itérateurs.
  - Manipuler des sous-listes.
- Deux listes sont égales si elles possèdent les mêmes éléments dans le même ordre.

Quelques méthodes de l'interface `List` (I)

```
public interface List<E> extends Collection<E>
```

- `E get(int index)` : Renvoie l'élément à la position fournie en paramètre.
- `E set(int index, E e)` : Remplace l'élément à la position fournie en paramètre (optionnelle).
- `boolean add(E e)` : Ajoute l'élément fourni en paramètre à la fin de la liste (optionnelle).
- `void add(int index, E e)` : Ajoute l'élément fourni en paramètre à la position `index` (optionnelle).
- `boolean remove(Object o)` : Supprime l'élément à la position fournie en paramètre (optionnelle).
- `boolean addAll(Collection<? extends E> c)` : Ajoute les éléments fournis en paramètre à la fin de la liste (optionnelle).

Quelques méthodes de l'interface List (II)

Recherche :

- `int indexOf(Object o)` : Renvoie la première position dans la liste de l'élément fourni en paramètre.
- `int lastIndexOf(Object o)` : Renvoie la dernière position dans la liste de l'élément fourni en paramètre.

Itération :

- `ListIterator<T> listIterator()` : Renvoie un itérateur positionné sur le premier élément de la liste.
- `ListIterator<T> listIterator(int index)` : Renvoie un itérateur positionné sur l'élément dont l'index est fourni en paramètre.

Sous-liste :

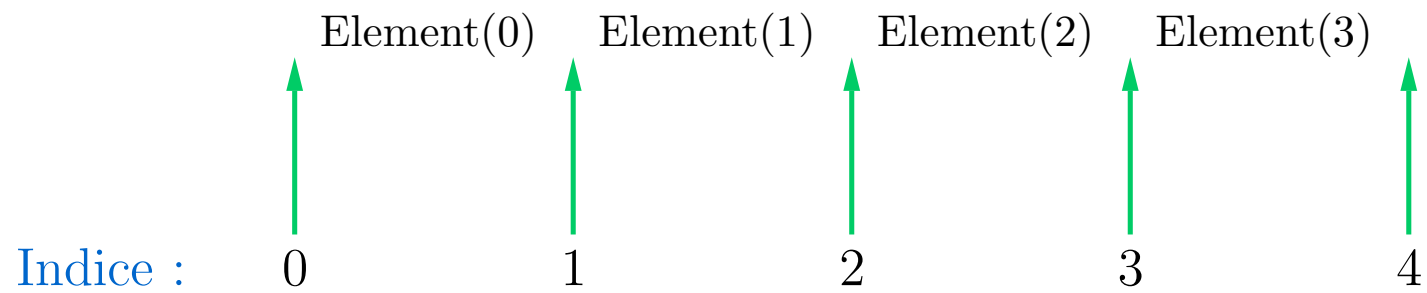
- `List<T> subList(int fromIndex, int toIndex)` : Renvoie une liste partielle de la collection contenant les éléments compris entre les index fournis en paramètre.

Itérateur de List

Une interface particulière permet de parcourir la liste dans les deux sens :

```
public interface ListIterator<E> extends Iterator
```

- Un itérateur de liste peut être vu comme un marqueur se trouvant entre deux éléments.



Quelques méthodes de ListIterator

Les trois méthodes héritées de Iterator (hasNext, next, et remove) ont la même fonctionnalité dans les deux interfaces.

Les méthodes

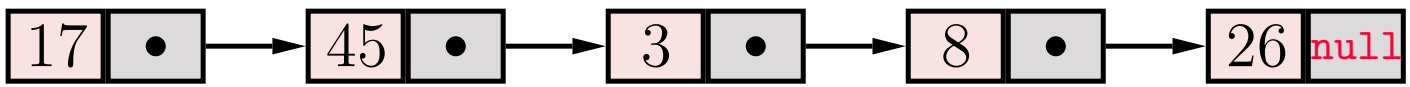
- boolean hasPrevious()
- E previous()

fonctionnent de façon analogue que les méthodes hasNext() et next().

- int nextIndex() : Renvoie l'indice de l'élément qui serait retourné par un appel à next().
- int previousIndex() : Renvoie l'indice de l'élément qui serait retourné par un appel à previous().

Les listes chaînées

Une liste chaînée est une liste dont chaque élément est relié au suivant par une référence à ce dernier.



- Opérations de base : L'insertion, la modification et la suppression d'un élément quelconque de la liste.
- La complexité des opérations effectuées sur une liste chaînée est proportionnelle à l'index sur lequel on effectue ces opérations.

Les listes doublement chaînées

Une liste doublement chaînée est une liste dont chaque élément possède deux références :

- Une référence sur l'élément qui le suit et
- une référence sur l'élément qui le précède.

La classe `LinkedList` est une implémentation de la liste doublement chaînée.

La classe ArrayList

Un objet de la classe `ArrayList` est un tableau dont la taille s'adapte automatiquement au nombre d'éléments de la collection.

- Inconvenient : Besoin d'instancier un nouveau tableau et copier les éléments dedans.
- La classe `ArrayList` est l'implémentation la plus simple de l'interface `List`.
- L'accès à un élément se fait grâce à son index.
- Elle implémente toutes les méthodes de l'interface `List`.

Élements d'une collection ArrayList

- Les ArrayList acceptent tout type de données, y compris null.

```
ArrayList a = new ArrayList();

a.add(13);
a.add("Bonjour !");
a.add(27.50f);
a.add('c');
```

Quelques méthodes particulières de `ArrayList`

À part les méthodes de `List` que la classe `ArrayList` implémente, elle possède les trois méthodes suivantes :

- `Object clone()` : Retourne une copie du tableau. Obsolète : Utiliser plutôt un constructeur en lui passant la liste en question en paramètre.
- `void ensureCapacity(int minCapacity)` : Augmente la capacité du tableau pour s'assurer qu'il puisse contenir le nombre d'éléments passé en paramètre.
- `void trimToSize()` : Ajuste la capacité du tableau sur sa taille actuelle.

Quelques remarques sur l'utilisation d'`ArrayList`

Ajout d'un nombre important d'éléments :

- Forcer l'agrandissement de cette capacité avec la méthode `ensureCapacity()`.
- Meilleure performance en changeant la taille une seule fois si de nombreux éléments doivent être ajoutés plutôt que de changer la taille plusieurs fois selon les besoins.
- Évite une perte de temps liée au recalcul de la taille de la collection.
- Un constructeur permet de préciser la capacité initiale.

Lors de l'ajout d'un élément dans la collection, si le tableau de stockage est trop petit alors un nouveau, plus grand est créé.

- Le temps d'ajout d'un élément n'est pas constant.
- Les temps d'exécution de l'insertion ou la suppression d'un élément à une position quelconque est variable puisque cela peut nécessiter l'adaptation de la position d'autres éléments.

LinkedList ou ArrayList ?

- `ArrayList` est plus rapide (permet un accès par position en temps constant, pas d'allocation à chaque ajout, . . .)
- On peut passer une capacité au constructeur de `ArrayList`.
- `ArrayList` possède les opérations `ensureCapacity` et `trimToSize` en plus de celle de l'interface `List`.
- `LinkedList` est linéaire pour l'accès par position.
- `LinkedList` est adaptée si on fait beaucoup d'insertions/suppressions en milieu de liste (opérations en temps constant mais le facteur constant est élevé).
- `LinkedList` possède les opérations `addFirst`, `getFirst`, `removeFirst`, `addLast`, `getLast`, et `removeLast`.

On choisit `ArrayList` sauf si on a beaucoup de modifications en milieu de liste.

Les files (queue)

Une file est une structure de données qui permet de réaliser une FIFO (First In First Out) :

- Les premiers éléments ajoutés à la file seront les premiers à être récupérés.

L'interface Queue

L'interface `Queue`, ($\geq$ JDK 5.0), définit les fonctionnalités pour une file d'objets.

- Il s'agit d'une collection qui permet de stocker des éléments avant leur traitement.

Trois opérations standards :

- Ajouter un élément.
- Obtenir un élément.
- Consulter le prochain élément disponible : cette opération ne le retire pas de la collection.

Méthodes de l'interface Queue

```
public interface Queue<E> extends Collection<E>
```

- `boolean add(E e)` : Ajouter un élément.
- `boolean offer(E e)` : Ajouter un élément.
- `E element()` : Obtenir l'élément en tête de la structure sans le retirer.
- `E peek()` : Obtenir l'élément en tête de la structure sans le retirer.
- `E poll()` : Obtenir l'élément en tête de la structure et le retirer.
- `E remove()` : Obtenir l'élément en tête de la structure et le retirer.

En cas d'échec

Deux comportements différents en cas d'échec :

- Lever une exception
- Renvoyer un booléen indiquant le succès de l'opération.

	Lever une exception	Retourner une valeur spéciale
Ajouter un élément à la fin	`add()`	`offer()`
Obtenir et retirer le 1^{er} élément	`remove()`	`poll()`
Obtenir sans retirer le 1^{er} élément	`element()`	`peek()`

Agenda du jour

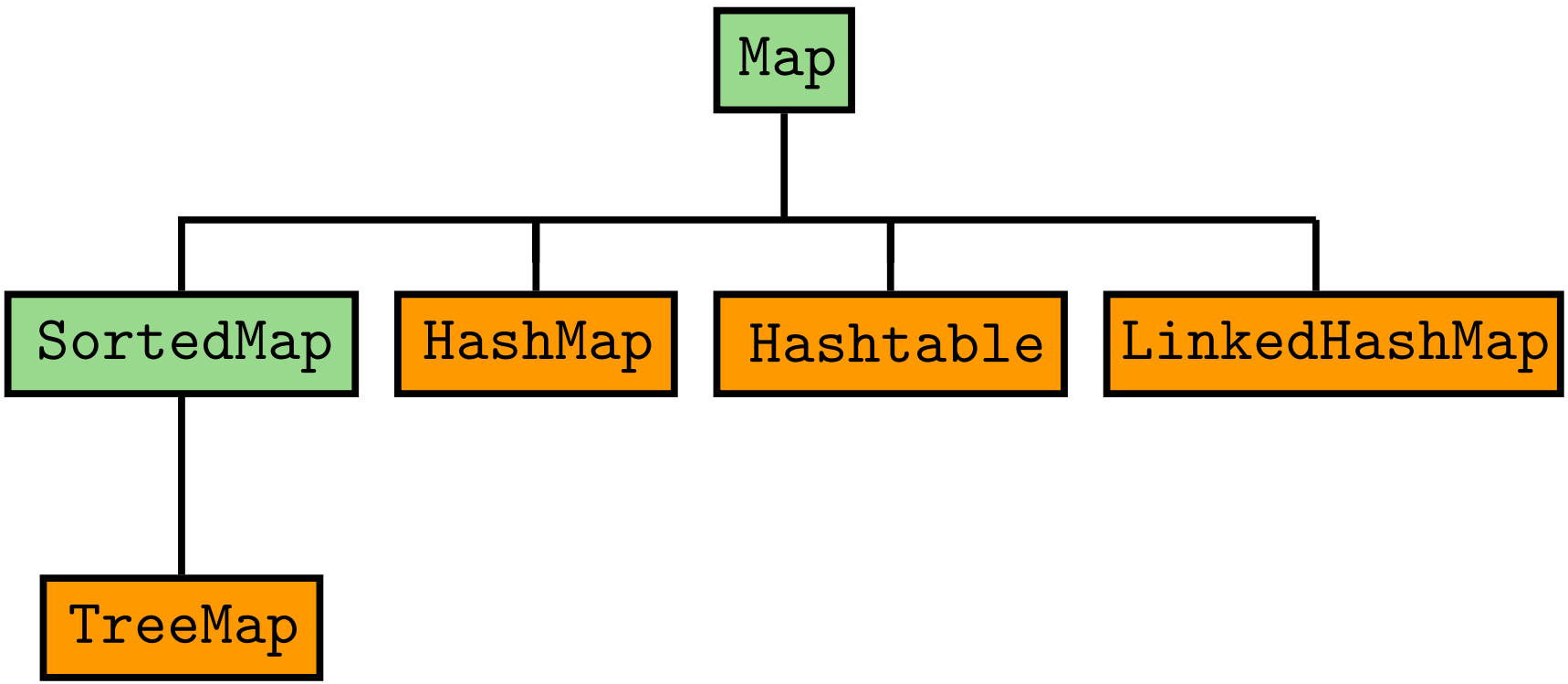
L'interface Map

Un objet de type Map associe une clé à une valeur.

- Peut être vu comme un tableau associatif.
- La clé doit être unique.
- La valeur peut être associée à plusieurs clés.

Deux objets Map sont égaux s'ils représentent les mêmes associations clé/valeur.

Hierarchie des Map



Les méthodes de base

```
public interface Map<K,V>
```

- `V put(K key, V value)` : Insère la clé et sa valeur associée fournies en paramètres.
- `V get(Object key)` : Renvoie la valeur associée à la clé fournie en paramètre.
- `V remove(Object key)` : Supprime l'élément dont la clé est fournie en paramètre.
- `boolean containsKey(Object key)` : Indique si la clé est contenue dans la collection.
- `boolean containsValue(Object value)` : Indique si la valeur est contenue dans la collection.
- `int size()` : Renvoie le nombre d'éléments de la collection.
- `boolean isEmpty()` : Indique si la collection est vide.

Autres méthodes

Traitement par lot

- `void putAll((Map<? extends K,? extends V> m))` : Insère toutes les clés/valeurs de l'objet fourni en paramètre
- `void clear()` : Supprimer tous les éléments de la collection

"Vue de collection"

- `Set<K> keySet()` : Renvoie un ensemble contenant les clés de la collection.
- `Collection<V> values()` : Renvoie une collection qui contient toutes les valeurs des éléments.
- `Set<Map.Entry<K,V>> entrySet()` : Renvoie un ensemble contenant les paires clé/valeur de la collection.

Parcourir une Map

Les méthodes de "vue de collection" permettent de voir une `Map` comme une collection de trois façons différentes :

- `keySet` représente l'ensemble des clés
- `values` représente la collection des valeurs
- `entrySet` représente l'ensemble des couples (clé, valeur)

Ces méthodes fournissent l'unique moyen pour parcourir une `Map`.

Interface Entry

```
public static interface Map.Entry<K,V>
```

- `K getKey()` : Renvoie la clé correspondante.
- `V getValue()` : Renvoie la valeur correspondante.
- `V setValue(V value)` : Remplace la valeur correspondante par la valeur fournie en paramètre.

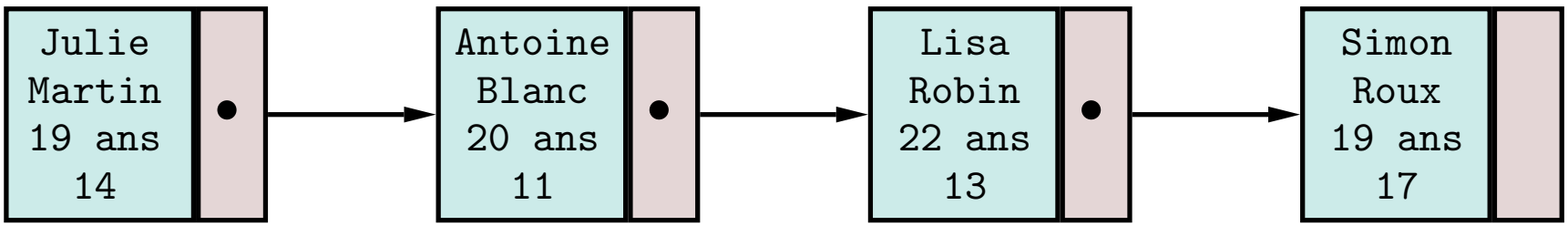
Un exemple

On a une collection d'objets où chaque objet représente un élève : (nom, prénom, âge, note de POO).

Exemple d'une instance de la classe `Eleve` :

Question : Quelle structure de données dois-je utiliser afin de récupérer facilement des informations sur un élève précis ?

Approche avec des listes chaînées



Défaut des listes chaînes :

- Il n'est pas possible d'accéder directement à un élément précis.
- Il faut parcourir la liste en avançant d'élément en élément jusqu'à trouver celui qu'on recherche.
- Performance pauvre lorsque on a à faire à des listes volumineuses.

Approche avec des tableaux

tab[0]	tab[1]	tab[2]	tab[3]
Julie Martin 19 ans 14	Antoine Blanc 20 ans 11	Lisa Robin 22 ans 13	Simon Roux 19 ans 17

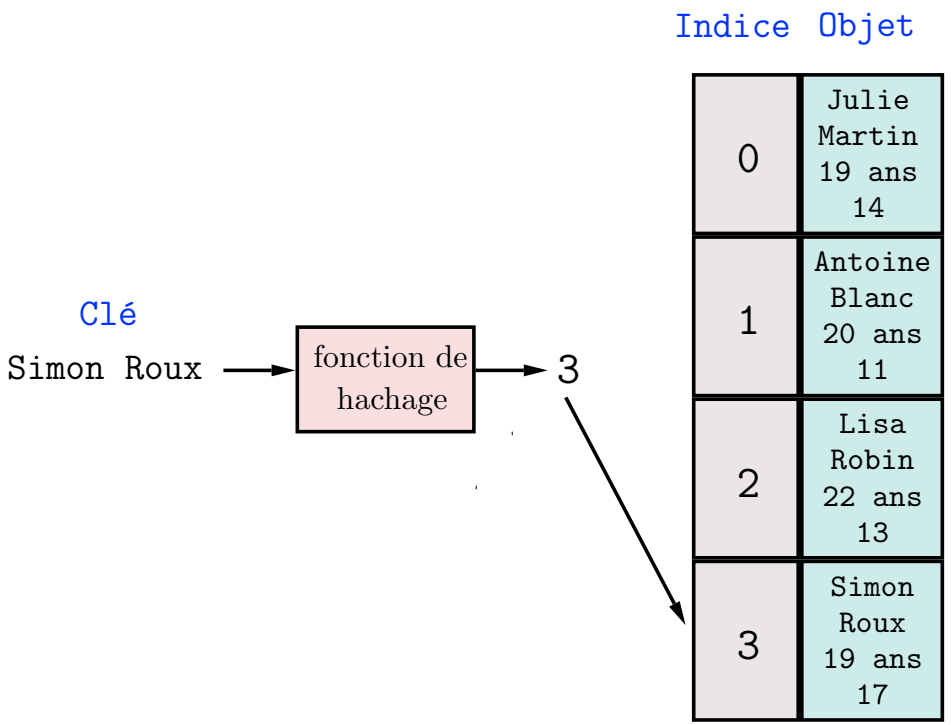
On peut accéder directement à une case du tableau en utilisant son indice.

Mais : Si on recherche par exemple une information sur l'élève Simon Roux on ne peut pas écrire

```
tab["Simon Roux"];
```

Les tables de hachage

Une table de hachage est une structure qui permet une association clé-élément.

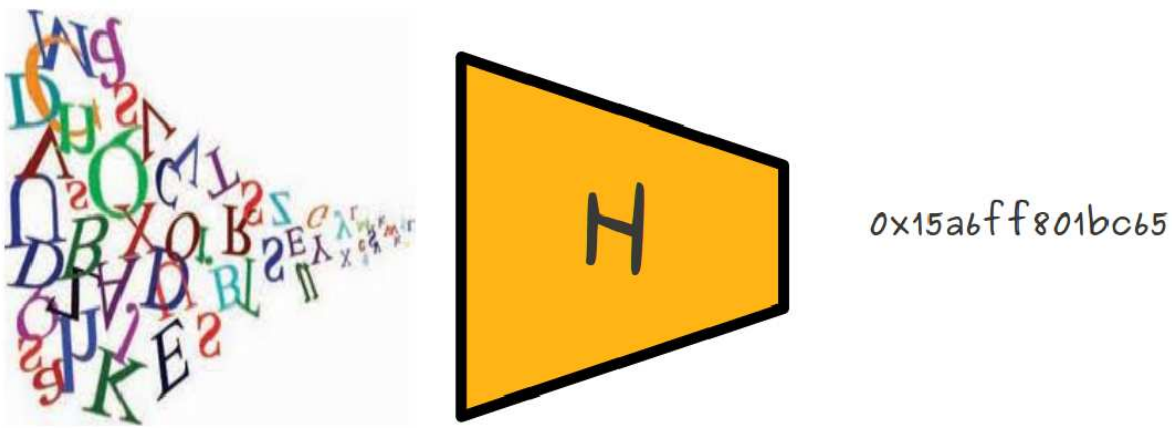


Les tables de hachage constituent un compromis entre des listes chaînées et des tableaux.

Les fonctions de hachage

Une fonction de hachage est une fonction qui calcule, à partir d'une donnée fournie en entrée, une empreinte (ou haché) servant à identifier rapidement la donnée.

$$H : \{0,1\}^* \to \{0,1\}^n$$



La méthode hashCode() de la classe String

```
public int hashCode()
```

Si s est une chaîne de caractères de longueur n alors l'appel s.hashCode() retourne l'entier :

$$s[0] \cdot 31^{n-1} + s[1] \cdot 31^{n-2} + \ldots + s[n-1]$$

Condition pour une fonction de hachage

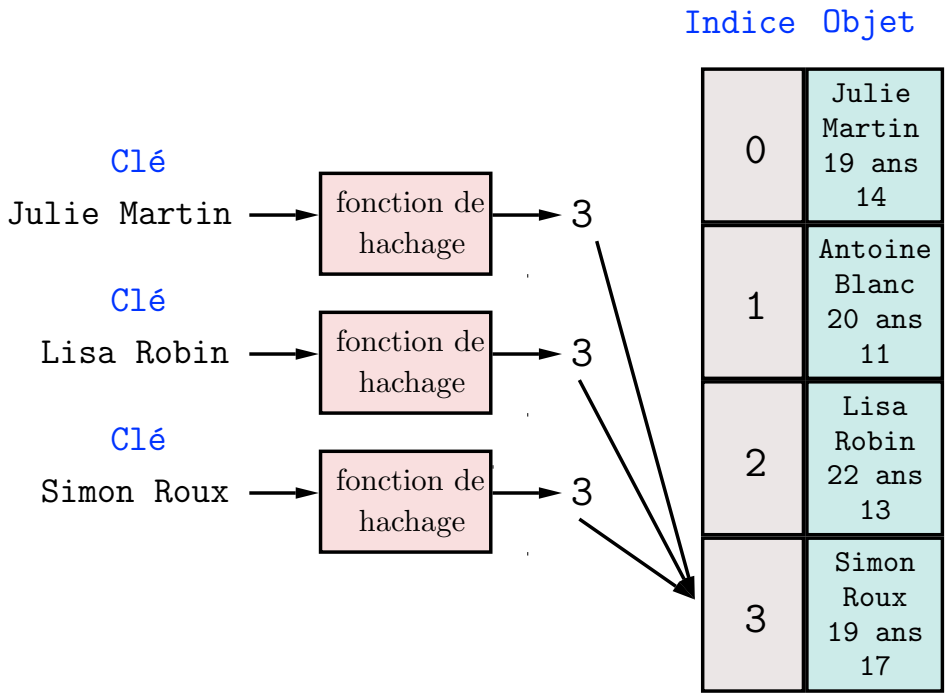
Une fonction de hachage H doit respecter la contrainte suivante :

- Si $k_1 = k_2$ alors $H(k_1) = H(k_2)$.

Cependant si $k_1 \neq k_2$, il est difficile (voir impossible) de garantir que $H(k_1) \neq H(k_2)$.

Les collisions

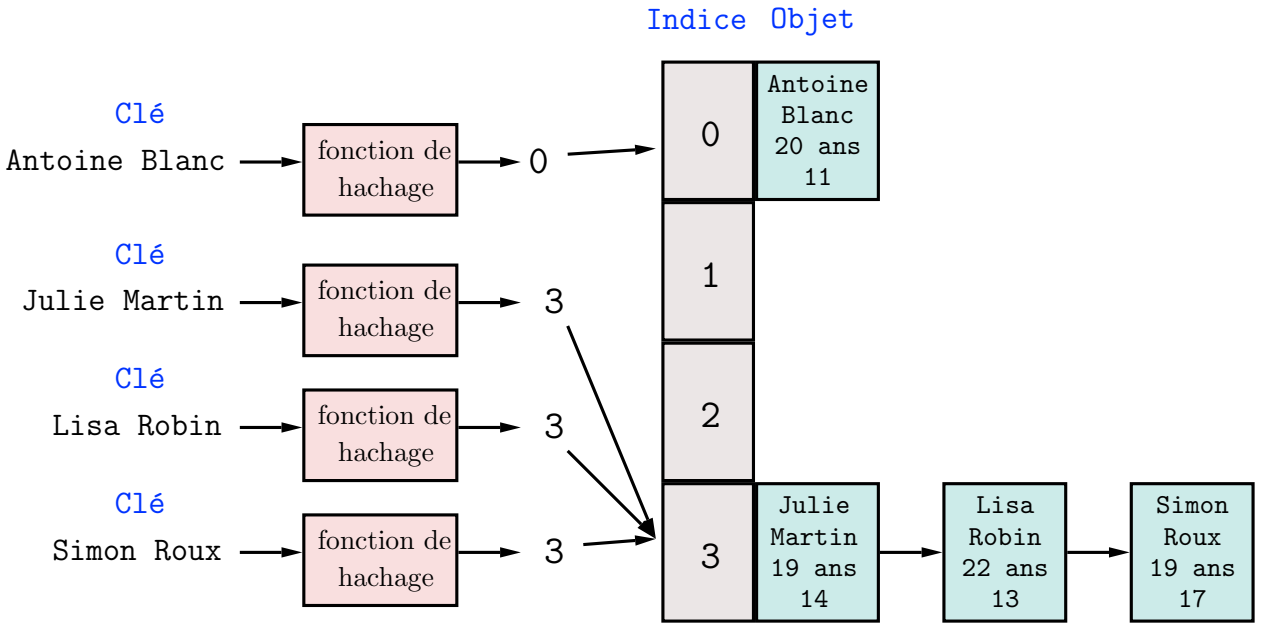
Quand la fonction de hachage renvoie le même nombre pour deux clés différentes, on dit qu'il y a collision.



Raisons pouvant expliquer une collision :

- La fonction de hachage est de mauvaise qualité.
- Le tableau est trop petit.

Une approche pour gérer les collisions



Chaque case contient une `liste chaînée` de tous les éléments dont le haché de leur clé correspond à l'indice de la case.

- Une mauvaise fonction de hachage peut aggraver considérablement les performances.
- Utiliser une fonction de hachage qui répartit les clés de façon équilibrée dans les différentes cases.

La classe Hashtable

```
Class Hashtable<K,V>
```

- Utilise la structure d'une table de hachage pour ranger les clés.
- Pour stocker et retirer des objets d'une `Hashtable`, les objets qui jouent le rôle de la clé doivent implementer la méthode `hashCode()` ainsi que la méthode `equals()`.

Un exemple

- Créer une table de hachage d'entiers.
- Les noms des entiers sont utilisés comme clé.

```
Hashtable<String, Integer> entiers = new Hashtable<String, Integer>();

entiers.put("un", 1);
entiers.put("deux", 2);
entiers.put("trois", 3);


Integer n = entiers.get("deux");
if (n != null) {
    System.out.println("deux = " + n);
}
```

Exercice

- Créer une table de hachage contenant les 10 premiers entiers associés à leur carré.

```
Hashtable<Integer, Integer> entiersCarre = new Hashtable<Integer, Integer>();

for(int i = 0; i< 10; i++) {
    entiersCarre.put(i, i*i);
}
```

Exercice (suite)

- Afficher les nombres pairs et leur carré.

```
Hashtable<Integer, Integer> entiersCarre = new Hashtable<Integer, Integer>();

for(int i = 0; i< 10; i++) {
    entiersCarre.put(i, i*i);
}

for(Map.Entry<Integer, Integer> element : entiersCarre.entrySet()) {
    if(element.getKey() % 2 == 0) {
        System.out.println(element.getKey() + " -> " + element.getValue());
    }
}
```