Programmation Orientée Objet

Collections

Bibliothèques

- Développement d'un projet
  - Structuration du programme en classes/paquetages
  - Réutilisation de code existant

- Bibliothèques (« librairies »)
  - Librairies graphiques, de conteneurs, d'accès à une BD...
  - Moins de code à développer
  - Code plus efficace, plus sûr, maintenu

Bibliothèque de conteneurs : Collections

- Collections
  - ensemble de classes
  - fournissent des structures de données
  - des algorithmes pour les manipuler
- Principales collections
  - listes List
  - ensembles Set
  - files et piles Queue, Deque
  - dictionnaires Map
- Définies dans le paquetage java.util

Interfaces

- interface Collection<E>
  - collection d'objets de type E
  - type générique paramétré par le type E des objets stockés
  - méthodes :
    - boolean contains(Object o)
    - boolean add(E e) // vrai s'il est ajouté
    - boolean remove(Object o) // vrai s'il est retiré
    - boolean isEmpty()
    - int size()
    - Iterator<E> iterator()

Code générique

- Exemple

```
class Couple<Truc> {
    private Truc truc1,truc2;
    public Couple(Truc a,Truc b){
        truc1=a;truc2=b ;
    }
    public String toString(){
        return truc1.toString()+truc2.toString() ;
    }
}
```

Itération sur une collection

```
Collection<E> coll = // ...

for (E e : coll) {  // Pour tout élément de la collection
    // traiter e
}
```

Interface iterator :
https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

Exercice 1 : Ecrire le code équivalent en utilisant les itérateurs

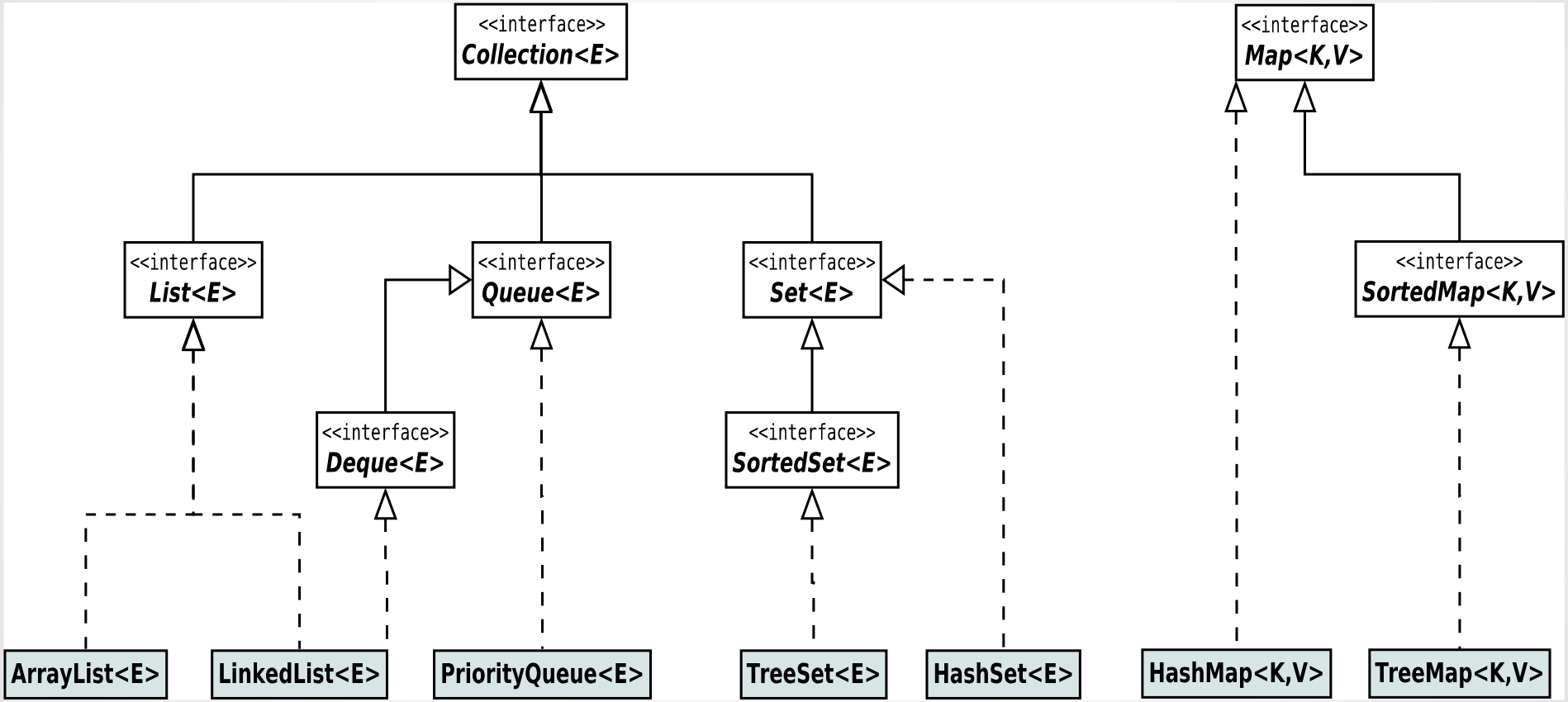
```
Collection<E> coll = // ...

for (E e : coll) {  // Pour tout élément de la collection
    // traiter e
}
```

→ avec les itérateurs.

Interfaces

- interface Collection<E>
- Sous-interfaces
  - interface List<E> : structure séquentielle (liste d'éléments dans un certain ordre contrôlé par l'utilisateur)
  - interface Queue<E> : piles, files
  - interface Set<E> : ensemble (tous les éléments sont différents)

Hiérarchie des collections



Quel sont les classes que vous avez utiliser pour le projet ?

- Voir sur socrative…

interface List<E>

- Méthodes
  - boolean add(int index, E e)
  - E get(int index)
  - E remove(int index)
- class LinkedList<E> : Liste doublement chaînée
  - Insertion et suppression d'un élément à une certaine position en temps constant
  - Accès au i-ème élement en *O(n)*
- class ArrayList<E> : Liste basée sur un tableau qui se redimensionne automatiquement si besoin
  - Accès au i-ème élément en temps constant
  - Insertion et suppression d'un élément en *O(n)*

interface Queue<E>

- Définit une file (FIFO : *first in first out*)
  - boolean add(E e) // ajout en fin et retourne true
  - E remove() // enlève et retourne le premier élément
  - boolean isEmpty() // vrai ssi la file est vide
  - E element() // retourne le premier élément
- Interface implémentée par LinkedList<E>
- Opérations en temps constant

interface Deque<E>

- *Double Ended Queue* : spécialise une Queue<E> avec des méthodes d'insertion et de retrait à la fois en tête et en queue de liste
  - void addFirst(E e) // ajout en tête
  - void addLast(E e) // ajout en fin
  - E removeFirst() // récupère et retire le premier élément
  - E removeLast() // récupère et retire le dernier élément
  - E getFirst() // premier élément
  - E getLast() // dernier élément
- Deque<E> peut être utilisé comme une pile (sommet : premier élément)
- Implémenté par LinkedList<E>
- Opérations en temps constant

class PriorityQueue<E>

- File d'attente avec priorité
- Les éléments de type E doivent être munis d'une relation d'ordre
- L'élément le plus prioritaire est le plus petit selon cette relation d'ordre
- Méthodes :
  - boolean add(E e) // ajout d'un élément, retourne true
  - E remove() // retrait de l'élément le plus prioritaire
  - E element() // retourne l'élément le plus prioritaire
- Opérations add et remove en $O(\log n)$, opération element en temps constant

Relation d'ordre total sur E

Deux façon de définir une relation d'ordre total sur E

- Relation d'ordre « naturelle » :

  E implémente l'interface Comparable<E>

  méthode int compareTo(E e)
  - renvoie *0* si this = e
  - renvoie *un entier strictement positif* si this > e
  - renvoie *un entier strictement négatif* si this < e

- Autre relation d'ordre sur E :

  classe C qui implémente l'interface Comparator<E>

  méthode int compare(E e1, E e2)

  Objet de type C passé en paramètre lors de la construction de la file.

File d'attente avec priorités

- Implémentation : à l'aide d'un tas

  (structure d'arbre telle que tout noeud a une valeur inférieure à celle de ses fils)

- Remarque :

  Le parcours de la file avec iterator() ne parcourt pas la file dans l'ordre de priorités des éléments.

Exercice 2’

Ecrire une classe comparateur générique qui compare 2 éléments **Bidule** en fonction de leur représentation via toString. Son constructeur prendra en argument un booléen indiquant si la relation d’ordre est dans l’ordre alphabétique(true) ou alphabétique inverse(false)

Exercice 2 : Ensemble ordonné

- Soit le programme suivant : Quelle sera la sortie? Si besoin proposer des modifications pour que les chaînes soient affichées dans l'ordre inverse alphabétique.

```
import java.util.*;

public class Exercice2 {

  public static void main(String[] args){

    Collection<StringBuffer> coll=new PriorityQueue<StringBuffer>();

    coll.add(new StringBuffer("Bonjour"));

    coll.add(new StringBuffer("Salut"));

    coll.add(new StringBuffer("Hello"));

    coll.add(new StringBuffer("Coucou"));

    for(StringBuffer sb:coll){

        System.out.println(sb.toString()) ;

    }

}
```

Exercice 3 :

- Proposer un programme gérant un ensemble d'étudiants (nom, prénom, notePOO), les élèves sont ajoutés dans leur ordre d'arrivé, ils sont ensuite énumérés :
  - Soit par ordre alphabétique
  - Soit par mérite (meilleure note en premier)
  - Soit en FIFO

  Ecrire le(s) codes correspondants.

interface Set<E>

- Set<E> : ensemble d'éléments de type E (donc un élément apparaît au plus une fois dans un Set)
- L'égalité entre deux éléments est testée par la méthode boolean equals(Object e), qui doit donc être redéfinie correctement dans E.
- Peut contenir l'élément null.
- Le comportement d'un ensemble est non spécifié si un élément de l'ensemble est modifié en modifiant la relation d'égalité sur les éléments.
- Méthodes
  - boolean add(E e) // retourne false si l'élément est déjà présent
  - boolean remove(E e) // retourne false si l'élément était absent

class HashSet<E>

- Implémentation de Set<E> avec une table à adressage dispersé
- Le parcours de l'ensemble avec iterator() est réalisé dans un ordre quelconque.
- La méthode int hashCode() doit éventuellement être redéfinie.
- Les méthodes equals et hashCode doivent être cohérentes :
  - Deux objets « equals » doivent avoir le même hashCode
  - Le calcul du hashCode doit donc dépendre uniquement des valeurs utilisées pour tester l'égalité.
- Opérations en temps quasi constant s'il y a peu de collisions dans la table à adressage dispersé.
- Peut contenir la valeur null.

Table à adressage dispersé

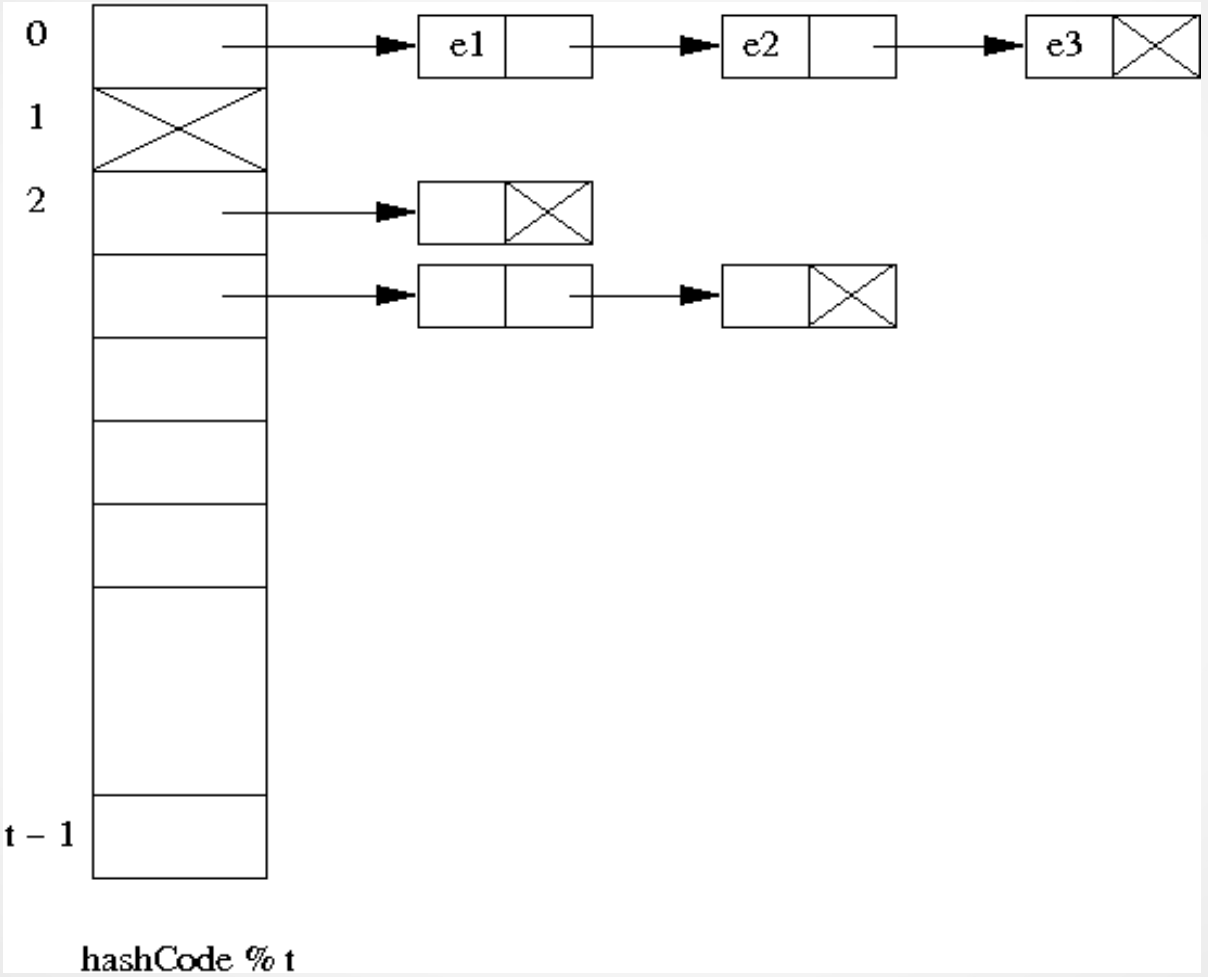


Table de « hachage »

interface SortedSet<E> extends Set<E>

- Ensemble ordonné
- Les éléments de E doivent être munis d'une relation d'ordre total :
  - soit implémentent l'interface Comparable<E> (ordre « naturel » ;
  - soit on définit une autre classe qui implémente l'interface Comparator<E>.
- La relation d'ordre doit être cohérente avec equals : la comparaison entre deux éléments vaut 0 ssi les éléments sont égaux.
- Comme avec pour les Set, hashCode doit être cohérent avec equals : deux éléments égaux doivent avoir le même hashCode.
- L'itérateur (iterator()) parcourt l'ensemble ordonné dans l'ordre croissant.

class TreeSet<E> implements SortedSet<E>

- Implémentation d'un ensemble ordonné
- Les opérations de base (add, remove, contains) sont en $O(\log n)$
- Implémenté à l'aide d'un arbre de recherche équilibré
- Méthodes
  - E first() // Elément le plus petit de l'ensemble
  - E last() // Elément le plus grand de l'ensemble
- Ne peut pas contenir la valeur null

Utilisation des collections

- On utilise *l'interface la plus générale possible* pour typer un objet collection
- On utilise la classe concrète uniquement lors de la construction de l'objet collection

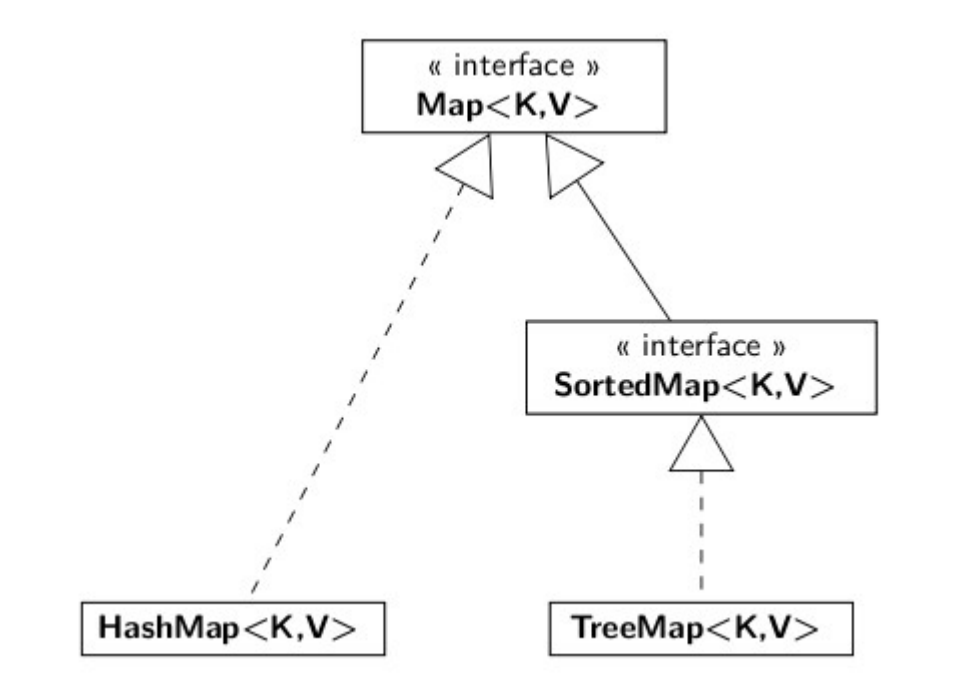
```
Set<Etudiant> s = new TreeSet<Etudiant>();
Etudiant e = new Etudiant("Martin", "Pierre", 12);
s.add(e);
for (Etudiant c : s) {
   System.out.println(c);
}
```

- Si par la suite on veut utiliser un HashSet à la place d'un TreeSet (plus efficace pour ce qu'on a à faire), il suffit de modifier le programme *à un seul endroit* :

```
Set<Etudiant> s = new HashSet<Etudiant>();
Etudiant e = new Etudiant("Martin", "Pierre", 12);
s.add(e);
for (Etudiant c : s) {
   System.out.println(c);
}
```

Dictionnaires (associations)

- Map<K,V> spécifie une association (fonction) entre une clé de type K et une valeur de type V



interface Map<K,V>

- Une map ne peut contenir plusieurs fois la même clé. A chaque clé est associée une unique valeur.
- La méthode equals doit être correctement définie sur les clés.

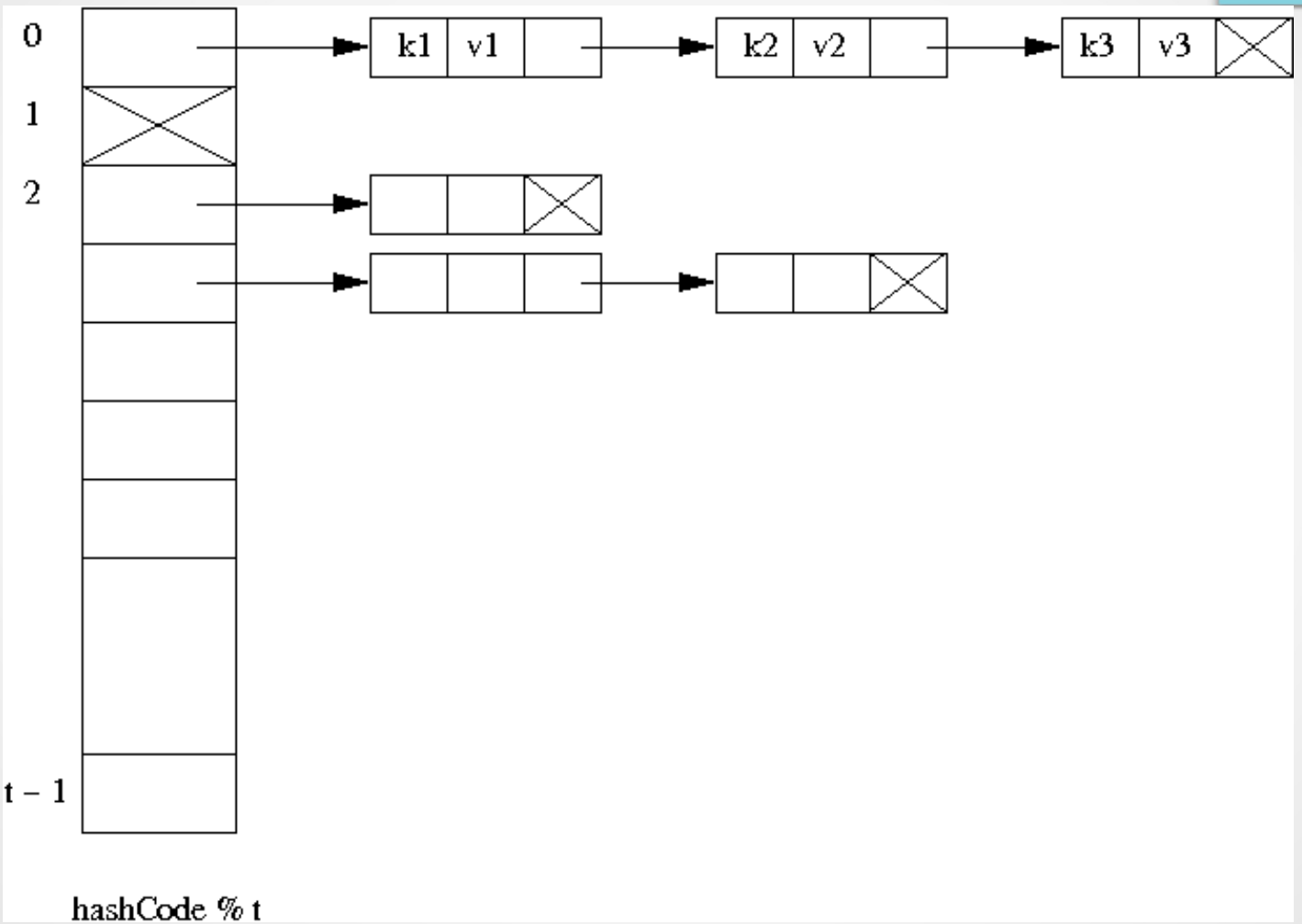
interface Map<K,V>

- Méthodes
  - V put(K key, V value) // associe la valeur à la clé
    // retourne la valeur précédente associée à la clé, et null si aucune valeur n'était associée à la clé
  - V get(Object key) // retourne la valeur associée à la clé, ou null si aucune valeur n'était associée à la clé
  - V remove(Object key) // supprime la valeur associée à la clé
    // retourne la valeur qui était associée à la clé, ou null si aucune valeur n'était associée à la clé
  - boolean containsKey(Object key)
  - boolean containsValue(Object value)
  - int size()
  - int isEmpty()
  - Set<K> keySet() // L'ensemble des clés de cette map
  - Collection<V> values() // La collection des valeurs de cette map

class HashMap<K,V>

- Une HashMap est une Map implémentée à l'aide d'une table à adressage dispersé.
- Autorise la valeur null pour une clé ou une valeur.
- Opérations de base (get, put) en temps quasi constant s'il y a peu de collisions dans la table à adressage dispersé.

HashMap<K,V>



Dictionnaire implémenté par une table à adressage dispersé

interface SortedMap<K,V> extends Map<K,V>

- Map avec une relation d'ordre total *sur les clés*

class TreeMap<K,V> implements SortedMap<K,V>

- Implémentation de SortedMap avec un arbre de recherche équilibré
- Opérations de base (containsKey, get, put, remove) en $O(\log n)$
- Ne peut contenir la clé null
- Peut contenir la valeur null

Classes enveloppes (« wrappers »)

- Les collections et les dictionnaires ne peuvent pas contenir « directement » des valeurs de type primitif
- Set<int> s ; // error : required reference, found int
- Pour contourner ce problème, on utilise des classes enveloppes (« wrapper ») :
  - Integer, Float, Long, Double, Boolean, Character...
  - Classes contenant un attribut du type primitif (non mutable)

```
Integer x = new Integer(1); // int -> Integer
int i = x.intValue();       // Integer -> int
Set<Integer> s = new HashSet<Integer>();
for (int j=0; j<10; j++)
      s.add(new Integer(j)) ;
for (Integer y : s)
      System.print(y.intValue());
```

Autoboxing, unboxing

- On a des conversions automatiques entre un type primitif et sa classe enveloppe

```
Integer x = 1; // Autoboxing
int i = x;          // Unboxing

Set<Integer> s = new HashSet<Integer>();

for (int j = 0; j < 10; j++)
      s.add(j); // Autoboxing

for (Integer y : s)
      System.out.print(y); // Unboxing
```

Exercice 4 :

Soit les opérations suivantes sur une Collection<E> c :

c.add(o1) ;

c.add(o2) ;

c.add(o3) ;

c.contains(o3) ;

Parmi : hashCode, equals, compareTo de la classe E seront appelées et combien de fois.

- En considérant : LinkedList, ArrayList, TreeSet, HashSet

Exercice 5 : Test de performance

- Justifier les résultats :

Structure	Insertion	Test d'exist.	Accès(i)	Ajout(i)	Suppression(i)	Suppression(0)
HashSet	59 ms	2 ms				
TreeSet	48 ms	4 ms				
LinkedList	21 ms	2304 ms	2292 ms	2452 ms	3209 ms	4 ms
ArrayList	14 ms	881 ms	1 ms	119 ms	111 ms	53 ms

Exercice 6 : Généricité

- Soit l'interface

```
public interface Reversible<E> {
    // return a new ensemble E with éléments reversed
    public E reverse() ;
}
```

- Ecrire le code d'une classe générique, ainsi qu'un test ArrayListReversible qui implémente cette méthode