Java : les collections

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en Programmation par contrainte (IA)
Ingénieur en Génie logiciel

elmouelhi.achref@gmail.com

Plan

Introduction

Les collections, c'est quoi ?

- sont des objets
- permettent de regrouper et gérer plusieurs objets

Introduction

Pourquoi ne pas utiliser les tableaux ?

Introduction

Pourquoi ne pas utiliser les tableaux ?

Pour plusieurs raisons

- Il faut connaitre à l'avance la taille du tableau
- Si on veut dépasser la taille déclarée, il faut créer un nouveau tableau puis copier l'ancien (certains langages ont proposés l'allocation dynamique mais ça reste difficile à manipuler)
- Il est difficile de supprimer ou d'ajouter un élément au milieu du tableau
- Il faut parcourir tout le tableau pour localiser un élément (problème d'indexation)
- Les tableaux ne peuvent contenir des éléments de type différent

Introduction

Plusieurs types de collections proposés

- `List` : tableaux extensibles à volonté, accessibles via leurs indice ou valeur
- `Set` : collection qui n'accepte pas de doublons
- `Map` : collection qui exige une clé unique pour chaque élément
- `Queue` : collection gérée comme une file d'attente (`FIFO` : First In First Out)

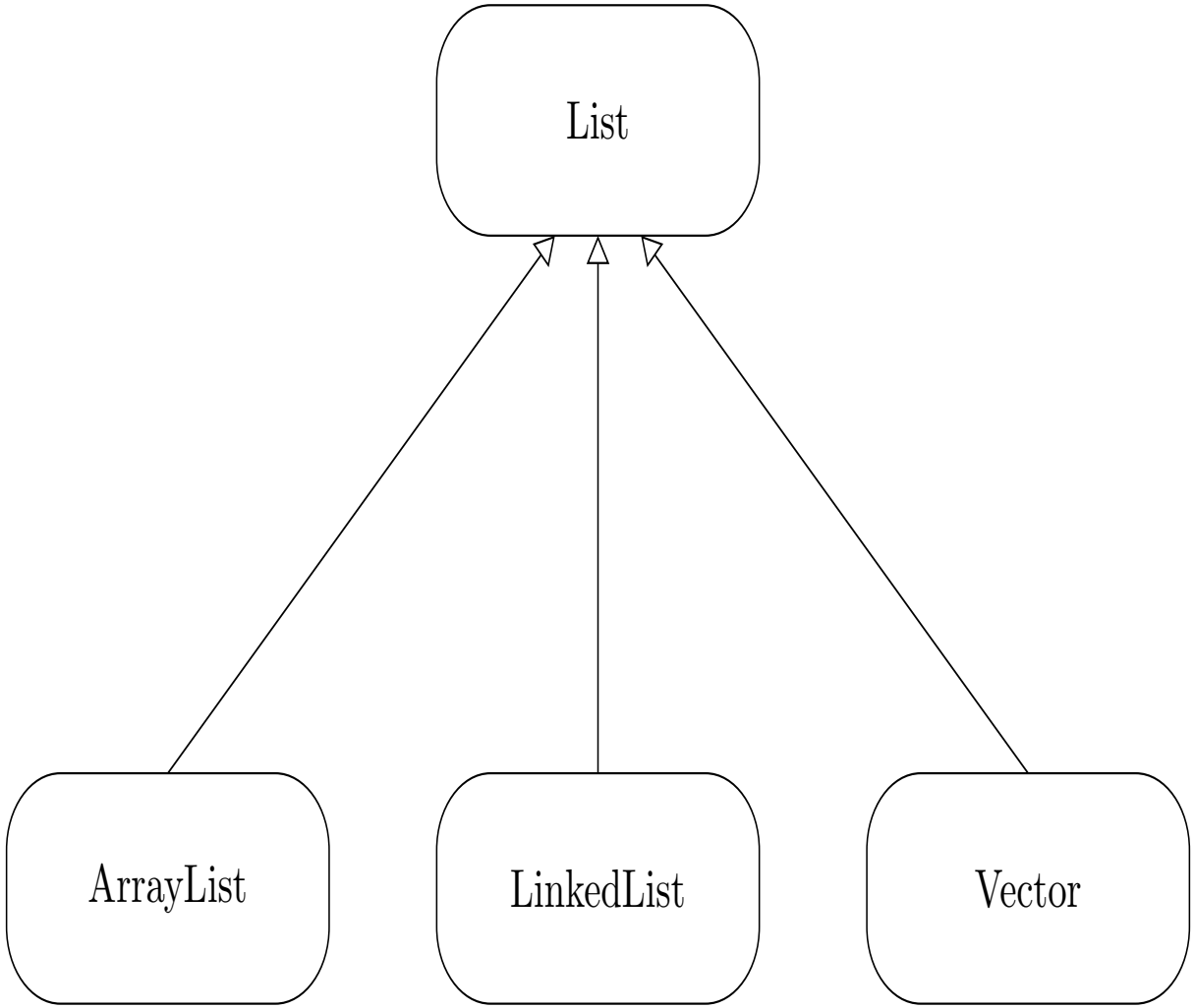
Introduction

Plusieurs types de collections proposés

- `List` : tableaux extensibles à volonté, accessibles via leurs indice ou valeur
- `Set` : collection qui n'accepte pas de doublons
- `Map` : collection qui exige une clé unique pour chaque élément
- `Queue` : collection gérée comme une file d'attente (`FIFO` : First In First Out)

Tous les imports de ce chapitre sont de `java.util.*;`

Les listes



Les listes

ArrayList

- Pas de limite de taille
- Possibilité de stocker tout type de données (y compris `null`)

Les listes

ArrayList : exemple

```
import java.util.ArrayList;
public class Test {
  public static void main(String[] args) {
    ArrayList liste = new ArrayList();
    liste.add(3);
    liste.add("Bonjour");
    liste.add(3.5);
    liste.add('c');
    for(int i = 0; i < liste.size(); i++){
      System.out.println("element d'indice " + i + "
          = " + liste.get(i));
    }
  }
}
```

Les listes

Autres méthodes de `ArrayList`

- `add(index,value)` : insère `value` à la position d'indice `index`
- `remove(index)` : supprime l'élément d'indice `index` de la liste
- `remove(object)` : supprime l'objet `object` de la liste
- `removeAll()` : supprime tous les éléments de la liste
- `set(index, object)` : remplace la valeur de l'élément d'indice `index` de la liste par `object`
- `isEmpty()` : retourne `true` si la liste est vide
- `contains(object)` : retourne `true` si object appartient à la liste
- ...

Les listes

Qu'affiche le programme suivant ?

```
import java.util.ArrayList;
public class Test {
  public static void main(String[] args) {
    ArrayList liste = new ArrayList();
    liste.add(0);
    liste.add("bonjour");
    liste.add(2);
    liste.remove(1);
    liste.set(1, "bonsoir");
    for(Object elt : liste) {
      System.out.print(elt + " ");
    }
  }
}
```

Les listes

Qu'affiche le programme suivant ?

```
import java.util.ArrayList;
public class Test {
  public static void main(String[] args) {
    ArrayList liste = new ArrayList();
    liste.add(0);
    liste.add("bonjour");
    liste.add(2);
    liste.remove(1);
    liste.set(1, "bonsoir");
    for(Object elt : liste) {
      System.out.print(elt + " ");
    }
  }
}
```

0 bonsoir

Les listes

`LinkedList` (ou bien liste chaînée en français)

- C'est une liste dont chaque élément a deux références : une vers l'élément précédent et la deuxième vers l'élément suivant.
- Pour le premier élément, l'élément précédent vaut `null`
- Pour le dernier élément, l'élément suivant vaut `null`

Les listes

LinkedList : exemple

```
import java.util.LinkedList;
public class Test {

  public static void main(String[] args) {
    LinkedList liste = new LinkedList();
    liste.add(5);
    liste.add("Bonjour ");
    liste.add(7.5f);
    for(int i = 0; i < liste.size(); i++)
      System.out.println("element d'indice " + i + "
          = " + liste.get(i));
  }
}
```

Les listes

Autres méthodes de `LinkedList`

- `addFirst(object)` : insère l'élément `object` au début de la liste
- `addLast(object)` : insère l'élément `object` comme dernier élément de la liste (exactement comme `add()`)
- ...

Les listes

Autres méthodes de `LinkedList`

- `addFirst(object)` : insère l'élément `object` au début de la liste
- `addLast(object)` : insère l'élément `object` comme dernier élément de la liste (exactement comme `add()`)
- ...

Remarque

- On peut parcourir une liste chaînée avec un `Iterator`
- Un itérateur est un objet qui a pour rôle de parcourir une collection

Les listes

`LinkedList` : parcours avec un itérateur

```
import java.util.LinkedList;
public class Test {

  public static void main(String[] args) {
    LinkedList liste = new LinkedList();
    liste.add(5);
    liste.add("Bonjour ");
    liste.add(7.5f);
    ListIterator li = liste.listIterator();
    while(li.hasNext())
      System.out.println(li.next());
  }
}
```

Les listes

Qu'affiche le programme suivant ?

```
import java.util.LinkedList;
public class Test {
  public static void main(String[] args) {
    LinkedList l = new LinkedList();
    l.add(0);
    l.add("bonjour");
    l.addFirst("premier");
    l.addLast("dernier");
    String s = "Salut";
    l.add(s);
    int  value = 2;
    l.add(value);
    l.remove("dernier");
    l.remove(s);
    l.remove((Object)value);
    ListIterator li = l.listIterator();
    while(li.hasNext())
      System.out.print(li.next() + " ");
  }
}
```

Les listes

Qu'affiche le programme suivant ?

```
import java.util.LinkedList;
public class Test {
  public static void main(String[] args) {
    LinkedList l = new LinkedList();
    l.add(0);
    l.add("bonjour");
    l.addFirst("premier");
    l.addLast("dernier");
    String s = "Salut";
    l.add(s);
    int  value = 2;
    l.add(value);
    l.remove("dernier");
    l.remove(s);
    l.remove((Object)value);
    ListIterator li = l.listIterator();
    while(li.hasNext())
      System.out.print(li.next() + " ");
  }
}
```

premier 0 bonjour

Les listes

On peut utiliser la généricité pour imposer un type à nos listes

```
LinkedList<Integer>liste = new LinkedList<Integer>()
   ;
```

Les listes

On peut utiliser la généricité pour imposer un type à nos listes

```
LinkedList<Integer>liste = new LinkedList<Integer>()
   ;
```

Ou

```
List<Integer>liste = new LinkedList<Integer>();
```

Les listes

On peut utiliser la généricité pour imposer un type à nos listes

```
LinkedList<Integer>liste = new LinkedList<Integer>()
   ;
```

Ou

```
List<Integer>liste = new LinkedList<Integer>();
```

La même chose pour `ArrayList`

```
List<Integer>liste = new ArrayList<Integer>();
```

Les listes

Considérons le tableau suivant

```
Integer [] tab = {2, 3, 5, 1, 9};
```

Pour convertir le tableau `tab` en liste

```
Integer [] tab = {2, 3, 5, 1, 9};
List<Integer>liste = new LinkedList(Arrays.asList(
   tab));
```

Les listes

Considérons le tableau suivant

```
Integer [] tab = {2, 3, 5, 1, 9};
```

Pour convertir le tableau `tab` en liste

```
Integer [] tab = {2, 3, 5, 1, 9};
List<Integer>liste = new LinkedList(Arrays.asList(
   tab));
```

Ou en plus simple

```
List<Integer> ent = Arrays.asList(tab);
```

Les listes

Considérons le tableau suivant

```
Integer [] tab = {2, 3, 5, 1, 9};
```

Pour convertir le tableau `tab` en liste

```
Integer [] tab = {2, 3, 5, 1, 9};
List<Integer>liste = new LinkedList(Arrays.asList(
   tab));
```

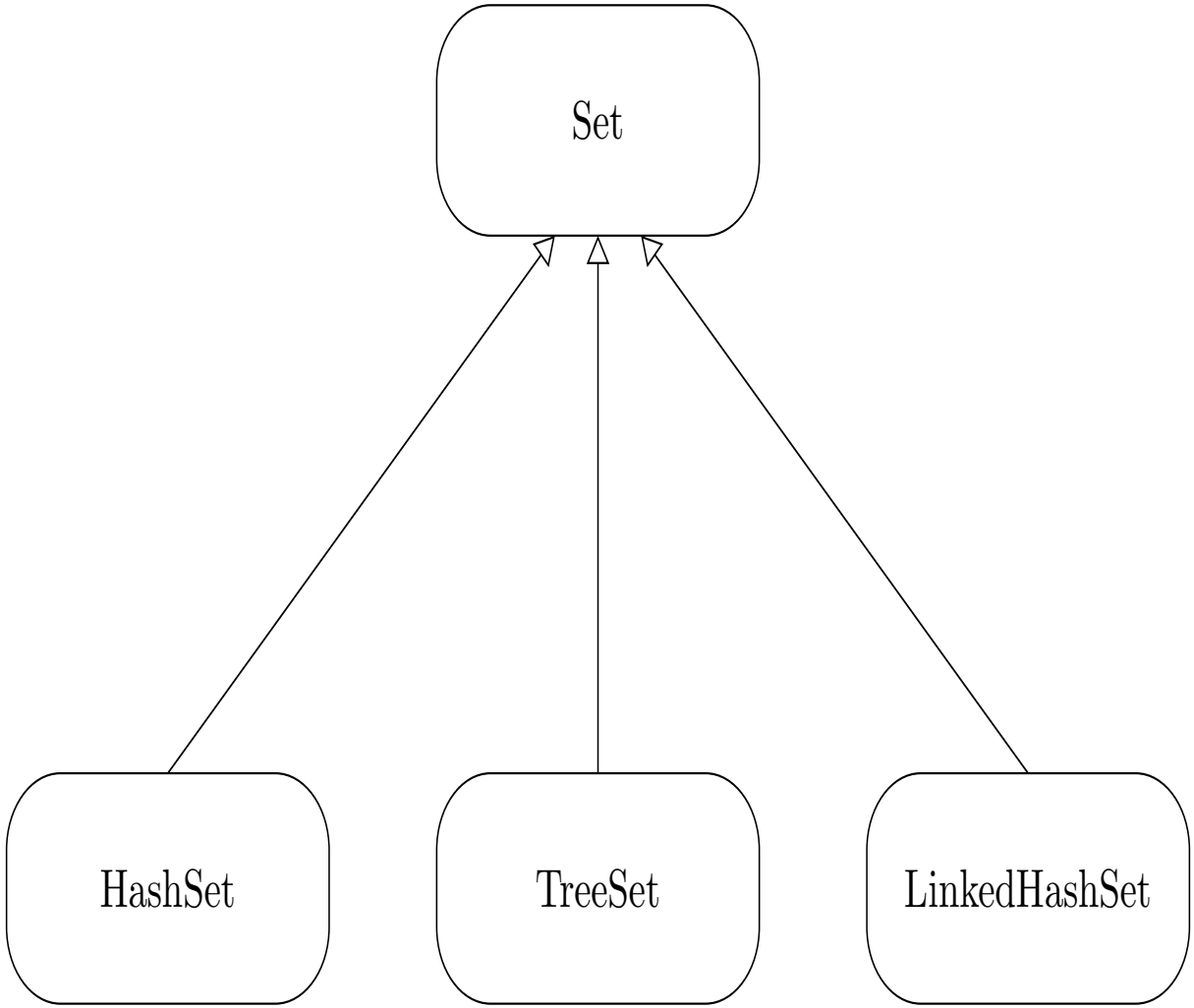
Ou en plus simple

```
List<Integer> ent = Arrays.asList(tab);
```

On peut le faire aussi sans créer le tableau

```
List<Integer>liste = Arrays.asList(2, 7, 1, 3);
```

Les Set



Les Set

`HashSet`

- Collection utilisant une table de hachage
- Possibilité de parcourir ce type de collection avec un objet `Iterator`
- Possibilité d'extraire de cet objet un tableau d'`Object`

Les Set

`HashSet` : exemple avec itérateur

```
import java.util.HashSet;
import java.util.Iterator;
public class Test {
  public static void main(String[] args) {
    HashSet hs = new HashSet();
    hs.add("bonjour");
    hs.add(69);
    hs.add('c');
    Iterator it = hs.iterator();
    while(it.hasNext())
      System.out.println(it.next());
  }
}
```

Les Set

`HashSet` : exemple avec conversion en tableau

```
import java.util.HashSet;
import java.util.Iterator;
public class Test {
  public static void main(String[] args) {
    HashSet hs = new HashSet();
    hs.add(7);
    hs.add(2);
    hs.add(5);
    Object[] obj = hs.toArray();
    for(Object o : obj)
      System.out.println(o);
  }
}
// l'affichage se fait dans l'ordre
```

Les Set

`TreeSet`

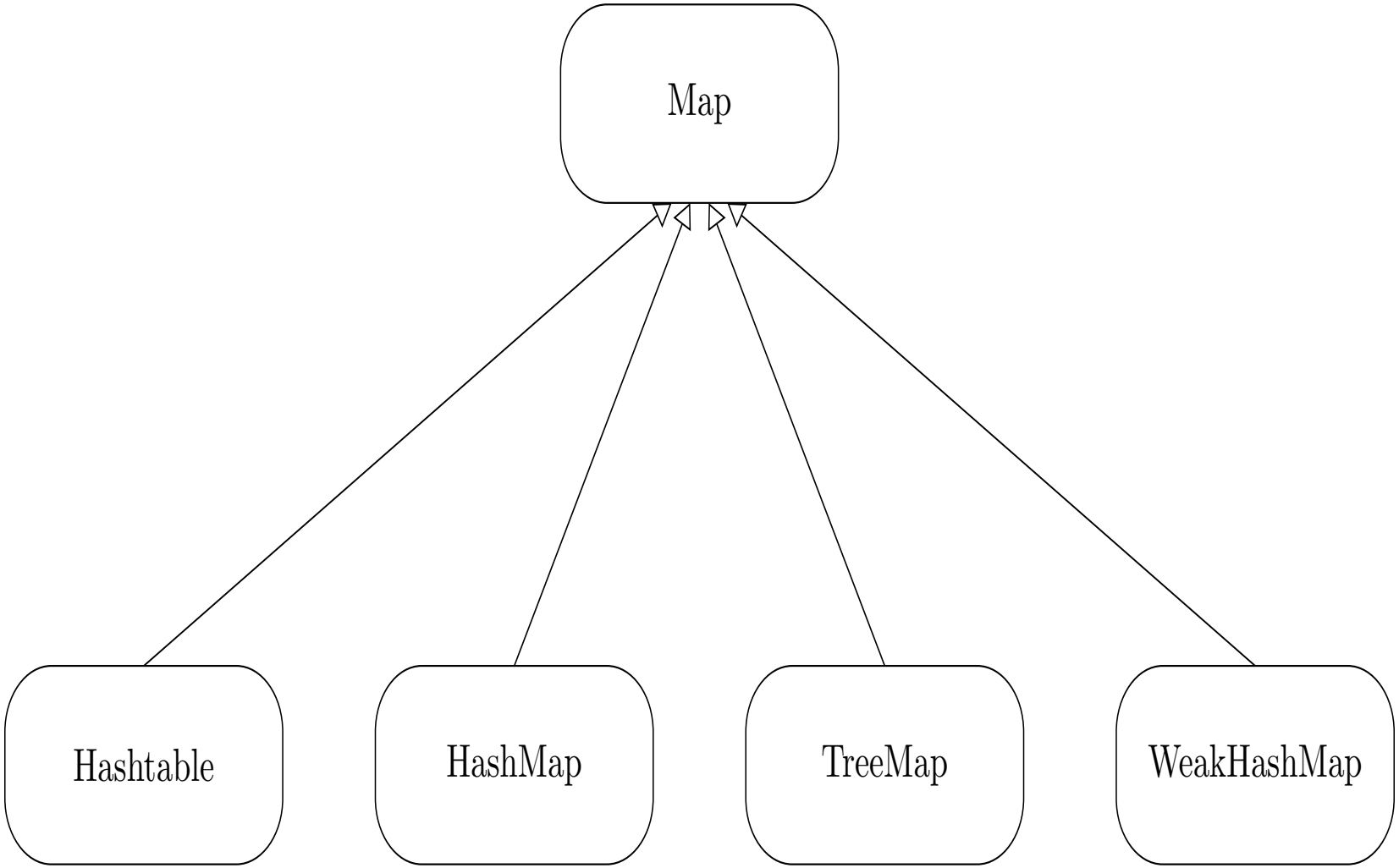
- Possibilité de parcourir ce type de collection avec un objet `Iterator`
- Les éléments enregistrés sont automatiquement triés

Les Set

`TreeSet` : exemple

```
import java.util.TreeSet;
import java.util.Iterator;
public class Test {
  public static void main(String[] args) {
    TreeSet ts = new TreeSet();
    ts.add(5);
    ts.add(8);
    ts.add(2);
    Iterator it = ts.iterator();
    while(it.hasNext())
      System.out.println(it.next());
  }
}
// L'affichage sera fait dans l'ordre
```

Les Map



Les Map

`Hashtable`

- `Hashtable` fonctionne avec un couple (clé,valeur)
- Elle utilise une table de hachage
- Elle n'accepte pas la valeur `null`
- La clé doit être unique
- Pour la parcourir, on utilise l'objet `Enumeration`

Les Map

`Hashtable` : exemple

```
import java.util.Enumeration;
public class Test {
  public static void main(String[] args) {
    Hashtable ht = new Hashtable();
    ht.put(1, "Java");
    ht.put(2, "PHP");
    ht.put(10, "C++");
    ht.put(17, "Pascal");
    Enumeration e = ht.elements();
    while(e.hasMoreElements())
      System.out.println(e.nextElement());
  }
}
```

Les Map

Autres méthodes de la classe `Hashtable`

- `isEmpty()` retourne `true` si l'objet est vide, `false` sinon.
- `contains(value)` retourne `true` si la valeur existe dans la `Hashtable`, `false` sinon.
- `containsKey(key)` retourne `true` si la clé existe dans la `Hashtable`, `false` sinon.
- `elements()` retourne une énumération des éléments de l'objet
- `keys()` retourne la liste des clés sous forme d'énumération

Les Map

`HashMap`

- `HashMap` fonctionne aussi avec un couple (clé,valeur)
- Elle utilise aussi une table de hachage
- `HashMap` accepte la valeur `null`
- La clé doit être unique
- Pour la parcourir, on utilise un objet `Set`

Les Map

`HashMap` : exemple

```
import java.util.HashMap;
public class Test {
  public static void main(String[] args) {
    HashMap<Integer, String> hm = new HashMap();
    hm.put(1, "Java");
    hm.put(2, "PHP");
    hm.put(10, "C++");
    hm.put(17, null);
    Set s = hm.entrySet();
    Iterator it = s.iterator();
    while(it.hasNext())
      System.out.println(it.next());
  }
}
```

Les Map

Pour afficher la clé et la valeur, on peut utiliser l'`Entry`

```
import java.util.HashMap;
public class Test {
  public static void main(String[] args) {
    HashMap<Integer, String> hm = new HashMap();
    hm.put(1, "Java");
    hm.put(2, "PHP");
    hm.put(10, "C++");
    hm.put(17, null);
    for (Entry<Integer, String> entry : hm.entrySet()) {
      System.out.print(entry.getKey() +  " " + entry.getValue()
        );
    }
  }
}
```

Les Map

Étant donné ce dictionnaire

```
HashMap<String, Integer> repetition = new HashMap();
repetition.put("Java",2);
repetition.put("PHP",5);
repetition.put("C++",1);
repetition.put("HTML",4);
```

Exercice 1

Écrire un programme Java qui permet de répéter l'affichage de chaque clé de ce dictionnaire selon la valeur associée

Résultat attendu (l'ordre n'a pas d'importance) :

```
JavaJava PHPPHPPHPPHPPHP C++ HTMLHTMLHTMLHTML
```

Les Map

Exercice 2 : Étant donnée la liste suivante :

```
List list = Arrays.asList(2,5,"Bonjour",true,'c',"3"
   ,"b",false,10);
```

Écrire un programme Java qui permet de stocker dans un dictionnaire (`Map`) les types contenus dans la liste `list` ainsi que le nombre d'éléments de cette liste appartenant à chaque type.

Résultat attendu :

```
Integer=3
Character=1
String=3
Boolean=2
```

Les Map

Une solution possible

```
HashMap<String, Integer> compteur = new HashMap<>();
for(Object elt : list) {
  String type = elt.getClass().getSimpleName();
  if (compteur.containsKey(type)){
    compteur.put(type, compteur.get(type)+1);
  }
  else {
    compteur.put(type, 1);
  }
}
for (Entry<String, Integer> entry : compteur.entrySet()) {
  System.out.println(entry.getKey() +  " " + entry.getValue());
}
```

Remarques : méthodes utiles

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.ListIterator;
public class Test {
   public static void main(String[] args) {
      List<String> lettres = new ArrayList<String>();
      lettres.add("d");
      lettres.add("b");
      lettres.add("a");
      lettres.add("c");
      Collections.sort(lettres); // pour trier la liste
      System.out.println(lettres);
      Collections.shuffle(lettres); // pour desordonner la liste
      System.out.println(lettres);
      List<String> sub = lettres.subList(1, 2); // extraire une sous-
         liste
      System.out.println(sub);
      Collections.reverse(sub); // pour trier la liste dans le sens
         decroissant
      System.out.println(sub);
   }
}
```

Remarques

`ArrayList` vs `LinkedList`

- `ArrayList` est plus rapide pour l'opération de recherche (`get`)
- `LinkedList` est plus rapide pour des opérations d'insertion et de suppression
- `LinkedList` utilise un chaînage double (deux pointeurs) d'où une consommation de mémoire plus élevée.

Remarques

`ArrayList` vs `LinkedList`

- `ArrayList` est plus rapide pour l'opération de recherche (`get`)
- `LinkedList` est plus rapide pour des opérations d'insertion et de suppression
- `LinkedList` utilise un chaînage double (deux pointeurs) d'où une consommation de mémoire plus élevée.

Remarques

- `Map` à utiliser lorsqu'on veut rechercher ou accéder à une valeur via une clé de recherche
- `Set` à utiliser si on n'accepte pas de doublons dans la collection
- `List` accepte les doublons permet l'accès à un élément via son indice et les éléments sont insérés dans l'ordre (pas forcément triés)