Collections et tables de hachage

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1

Université de Lille 1 SCIENCES ET TECHNOLOGIES — UFR IEEA Formations en Informatique de Lille 1

java.util

Manipulation de structures de données complexes

où le cours d'**ASD** devient indispensable...

- collections : listes, ensembles + itérateurs
- tables de hachage

on trouve des types pour ces structures dans le paquetage

`java.util`

(avec d'autres : pile (`Stack`), file (`Queue`), `Vector`, etc.)

Premier regard sur les collections

- une collection est un regroupement d'objets (ses éléments).
- on trouve des collections de comportements différents (listes, ensembles, etc.)
- une interface `java.util.Collection<E>` définit le contrat des collections.
- à partir de java 1.5, les collections sont **typées**.
  `Collection<E>` où **E** représente le type des éléments de la collection.

Méthodes principales de `Collection<E>`

`boolean add(E e)` *Ensures that this collection contains the specified element (optional operation).*

`boolean contains(Object o)` *Returns true if this collection contains the specified element*, càd $\exists e$ `(o==null? e==null : o.equals(e))`
$\Longrightarrow$ explique la signature de la méthode `equals`

`boolean isEmpty()` *Returns true if this collection contains no elements.*

`Iterator<E> iterator()` *Returns an iterator over the elements in this collection.*

`boolean remove(Object o)` *Removes a single instance of the specified element from this collection, if it is present (optional operation).*

`int size()` *Returns the number of elements in this collection.*

+ `addAll`, `removeAll`, `toArray`, etc.

List<E>

- interface `List<E>` = collection ordonnée d'objets

La structure de données « liste »

- suite ordonnée d'éléments (i.e. il existe un suivant et un précédent)
- de taille non bornée
- qui supporte les opérations :
  - d'accès à la tête de la liste : le premier élément
  - d'accès au reste de la liste : tous les éléments sauf le premier
  - d'ajout en tête : ajoute un nouvel élément avant la tête
  - de test de la vacuité
- toutes les autres opérations sont une construites à partir de celles-ci
- c'est une structure de données intrinsèquement récursive

List<E>

- interface `List<E>` = collection ordonnée d'objets
  2 classes :

`ArrayList<E>` listes implantées avec un tableau

API Doc *The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires $O(n)$ time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation.*

`LinkedList<E>` listes (doublement) chaînées

List<E> : Méthodes complémentaires

Dans une liste les éléments sont ordonnés, la notion de position a un sens.

`add(int index, E element)` ajout de l'élément à l'index-ième position

`E get(int index)` fournit l'index-ième élément de la liste.
`IndexOutOfBoundsException` - si $(index < 0 \,||\, index >= size())$

`E remove(int index)` supprime l'index-ième élément de la liste. (même exception)

`int indexOf(Object element)` indice de la première occurence `element` dans la liste, -1 si absent

`ListIterator<E>` itérateur pour listes doublement chaînées

Quoi utiliser ?

Cf. cours ASD.

- `ArrayList` si ajout et accès “direct” (indicé)
- `LinkedList` si nombreuses insertions et suppressions dans la liste

La liste chaînée mutable

- ajouter un élément en tête : créer un nouvelle cellule, puis chaîner = mettre à jour le suivant
  en $\Theta(1)$
- suppression d'un élément : recherche de la cellule `current_cell` à supprimer en se souvenant de la cellule précédente `previous_cell`
  + suppression par 'déchaînage'
  `previous_cell["next"] = current_cell["next"]`
  en $\mathcal{O}(n) + \Theta(1) = \mathcal{O}(n)$

La liste implantée avec un tableau

- inconvénients :
  - pas très efficace sur les opérations courantes
  - le dépassement de capacité peut être résolu en copiant le contenu dans un tableau plus grand, en $\Theta(n)$
  - la concaténation est en $\Theta(n+m)$
  - necessité de disposer d'un espace supplémentaire en $\Theta(n)$ pour ces deux dernières opérations
- avantages :
  - l'accès au k-ième élément est en $\Theta(1)$
  - peut permettre l'implantation de fonctions de recherche ou de tri efficaces

Résumé des complexités des opérations sur les listes

	Tableau	Listes SC	Listes DC	avec sentinelle
inserer en tête	$\mathcal{O}(n)$	$\Theta(1)$	$\Theta(1)$	
chercher	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	
supprimer [1]	$\mathcal{O}(n)$	$\Theta(1)$	$\Theta(1)$	
accès au premier	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	
accès au dernier	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
accès au suivant	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	
accès au précédent	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	
inserer après/avant [1]	$\mathcal{O}(n)$	$\Theta(1)$	$\Theta(1)$	

1. une fois l'élément trouvé

Quoi utiliser ?

Cf. cours ASD.

- `ArrayList` si ajout et accès “direct” (indicé)
- `LinkedList` si nombreuses insertions et suppressions dans la liste

```
.../java/test$ java TestCollection2 20000 20000
*** insertion en tete LinkedList
20000 insertions ds LinkedList :  16 ms
*** insertion en tete ArrayList
20000 insertions dans ArrayList :  403 ms
```


Collections Listes Itérateurs Ensembles Tables Attention !!! Typage

Quoi utiliser ?

Cf. cours ASD.

- `ArrayList` si ajout et accès “direct” (indicé)
- `LinkedList` si nombreuses insertions et suppressions dans la liste

```
.../java/test$ java TestCollection2 20000 20000
*** insertion en tete LinkedList
20000 insertions ds LinkedList :  16 ms
*** insertion en tete ArrayList
20000 insertions dans ArrayList :  403 ms
```

```
*** remove LinkedList
20000 suppressions dans LinkedList :  8 ms
*** remove ArrayList
20000 suppressions dans ArrayList :  398 ms
```

Université Lille 1 - Licence mention Informatique Programmation Orientée Objet 7

Collections Listes Itérateurs Ensembles Tables Attention !!! Typage

Méthodologie

en cas de “non obligation” (ou de doute) sur le choix :
utiliser l’upcast vers l’interface associée à la collection
pour faciliter le changement de choix d’implémentation

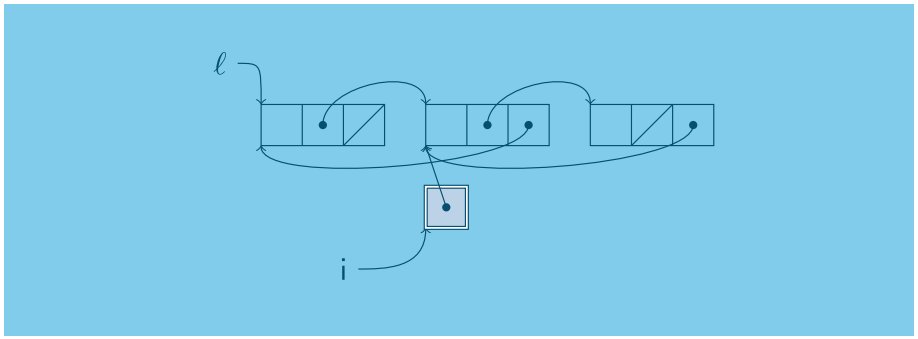
```
List<Livre> aList = new ArrayList<Livre>();
.
.  traitements avec uniquement des méthodes de l'interface List
.
```

si besoin ultérieurement on peut changer en :

```
List<Livre> aList = new LinkedList<Livre>();
.
.  mêmes traitements sans autre changement
.
```


Abstraction pour les parcours de listes

- un itérateur est une structure de donnée permettant le parcours
- opérations supportées :
  - avancer, reculer
  - est_en_fin, est_en_debut
  - valeur
  - inserer_apres, inserer_avant, supprimer



Itérateur

Pour parcourir les éléments d'une collection on utilise un **itérateur**. L'API Java définit une interface `java.util.Iterator<E>` (extraits) :

`boolean hasNext()` *Returns true if the iteration has more elements.*

`E next()` *Returns the next element in the iteration.*

`void remove()` *Removes from the underlying collection the last element returned by the iterator (optional operation).*

`ListIterator<E>` parcours avant/arrière (`previous()`, `hasPrevious()`) + `add(E e)`, `set(E e)`

Usage

```
Collection<Recyclable> trashcan = new ArrayList<Recyclable>();

trashcan.add(new Paper());              // upcast vers Recyclable
trashcan.add(new Battery());            // implicite

                                        // itérateur sur la collection
Iterator<Recyclable> it = trashcan.iterator();
while(it.hasNext()) {
   Recyclable ref = it.next();          // it.next() du type Recyclable
   ref.recycle();
}
```


Les `Iterator` sont **fail-fast** : si, après que l'itérateur ait été créé, la collection attachée est modifiée autrement que par un `remove` (ou `add`[1]) de l'itérateur alors l'itérateur lance une `ConcurrentModificationException`.
« Rupture » possible du contrat de l'itérateur.
Donc échec rapide et propre plutôt que de risquer l'incohérence.

```
List<Livre> l = ...;
for(int i = 0 ; i < 5; i++) {
   l.add(new Livre(...));
}
Iterator<Livre> itLivre = l.iterator();
Livre l = itLivre.next(); // ok
l.add(new Livre(...));    // modification de la liste
                          // ⟹ corruption de l'itérateur
l = itLivre.next();       // -> ConcurrentModificationException levée
```

[1] `ListIterator`

Attention

Il **ne faut pas** parcourir une liste en utilisant `get(int idx)`.
Il **faut** utiliser les itérateurs.

Attention

Il **ne faut pas** parcourir une liste en utilisant `get(int idx)`.
Il **faut** utiliser les itérateurs.

Pourquoi ne faut-il pas écrire :

```
List<...> l = ...;
for(int i = 0; i < l.size(); i ++) {
   utilisation de l.get(i)
}
```


Collections Listes Itérateurs Ensembles Tables Attention !!! Typage

Attention

Il **ne faut pas** parcourir une liste en utilisant `get(int idx)`.
Il **faut** utiliser les itérateurs.

Pourquoi ne faut-il pas écrire :

```
List<...> l = ...;
for(int i = 0; i < l.size(); i ++) {
   utilisation de l.get(i)
}
```

```
.../java/test$ java TestCollection 20000
*** parcours LinkedList avec itérateur
pacours 20000 éléments :  7 ms
*** parcours LinkedList avec get(i)
parcours 20000 éléments :  480 ms
```


Collections Listes Itérateurs Ensembles Tables Attention !!! Typage

Possibilité d'utiliser la syntaxe "à la *for-each*" pour itérer sur les collections :

```
for(Reclyclable r :  trashcan) {
   r.recycle();
}
```

NB : Cette syntaxe est possible sur les tableaux et toutes les classes qui implémentent l'interface `Iterable<T>`.

Collections Listes Itérateurs Ensembles Tables Attention !!! Typage

Iterable

L'interface `java.lang.Iterable<T>` est définie par la méthode :

```
public Iterator<T> iterator();
```

Les objets des classes qui implémentent cette méthode pourront être utilisés dans une boucle *for-each*.

```
public class Agence implements Iterable<Voiture> {
   private List<Voiture> lesVoitures;
   ...
   public Iterator<Voiture> iterator() {
      return this.lesVoitures.iterator();
   }
}

Agence agence = ...
for(Voiture v :  agence) {
   ...  utiliser v
}
```

Collection d'objets

- Les collections ne peuvent contenir que des objets.
  ↪ et donc pas de valeurs *primitives*
- `List<int>` **n'est pas** possible, il faut utiliser `List<Integer>`.

Depuis java 1.5, existe l'*autoboxing* ce qui signifie que les conversions

type primitif ↔ *classe associée*

sont gérées par le compilateur.
Ainsi on peut écrire :

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

l.add(12);              correspond à    l.add(new Integer(12));
int i = l.get(0);                       int i = l.get(0).intValue();
```


Set<E>

- interface `Set<E>` collection d'objets sans répétition de ***valeurs***
  2 classes :
  - `HashSet<E>` pour test appartenance rapide
    API Doc *This class offers constant time performance for the basic operations (`add`, `remove`, `contains` and `size`), assuming the hash function disperses the elements properly among the buckets.*
  - `TreeSet<E>` trié à partir d'une structure d'arbre (`SortedSet` : `first()`, `last()`)
    API Doc *This implementation provides guaranteed* $\log n$ *time cost for the basic operations (`add`, `remove` and `contains`).*
- `java.lang.Comparable` / `hashCode` et `equals`
  (cf. `TestSet.java`, `TestSetBis.java`, `TestTreeSet.java`)

On dispose de couples de données à ranger pour lesquels on souhaite faire les opérations

- **d'ajout,**
- **de recherche,**
- **(optionnellement de suppression).**

de manière très efficace

c'est-à-dire :

- aussi rapide qu'une liste pour ajouter
- aussi rapide qu'un tableau pour accéder

Une table de hachage est une structure de données dont le cahier des charges est le suivant :

- permet l'association d'une valeur à une clé
  dans l'exemple les valeurs sont des numéros de téléphone et les clés des noms
- permet un accès rapide à la valeur à partir de la clé (comme un tableau)
- permet l'insertion rapide (comme dans une liste)

Map<K,V>

"listes associatives", dictionnaire, index, tables, etc.

groupe d'associations (Clé,Valeur)

Les "Map" **ne sont pas** des `Collections`.
$\Longrightarrow$ pas d'itérateur.

`HashMap<K,V>` table de hachage, ajout et accès en temps constant
API Doc *This implementation provides constant-time performance for the basic operations (`get` and `put`), assuming the hash function disperses the elements properly among the buckets.*

`TreeMap<K,V>` en plus : clés triées
API Doc *This implementation provides guaranteed log(n) time cost for the `containsKey`, `get`, `put` and `remove` operations.*

`V get(K key)` récupère la valeur associée à une clé

`void put(K key, V value)` ajoute un couple (clé, valeur)

`V remove(Object key)` supprime le couple associé à la clé `key`

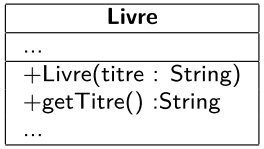
`boolean containsKey(Object key)` teste l'existence d'une clé (`equals`)

`boolean containsValue(Object value)` teste l'existence d'une valeur (`equals`)

`Collection<V> values()` renvoie la collection des valeurs

`Set<K> keySet()` renvoie l'ensemble des clés

`Set<Map.Entry<K,V>> entrySet()` renvoie l'ensemble des couples (clé,valeurs)
(objets `Map.Entry<K,V>`)




```
                                // associe un Auteur à un Livre
Map <Auteur,Livre> table = new HashMap <Auteur,Livre>();
Auteur auteur = new Auteur("Tolkien");
Livre livre1 = new Livre("Le Seigneur des Anneaux");
table.containsKey(auteur)               // vaut false
table.put(auteur,livre1);
S.o.p(table.get(auteur).getTitre());  // affiche le Seigneur des Anneaux
table.containsKey(auteur)               // vaut true
table.containsValue(livre1)             // vaut true
Livre livre2 = new Livre("Le Silmarillion");
table.put(auteur,livre2);
S.o.p(table.get(auteur).getTitre());  // affiche le Silmarillion
table.containsValue(livre1)             // vaut false
```


"Parcours" d'une Map (1)

pas d'itérateur "direct" $\Rightarrow$ itérer sur les clés

```
Map<Auteur,Livre> table = ...; // associe Auteur (clé) à Livre (valeur)
...
   public void afficheMap() {
      Set<Auteur> lesCles = this.table.keySet();
      Iterator<Auteur> it_cle = lesCles.iterator();
      while (it_cle.hasNext()) {
         Auteur a = it.next();
         S.o.p(a+" a ecrit "+ this.table.get(a));
      }
   }


   public void afficheMap() {
      for(Auteur a :  this.table.keySet()) {
         S.o.p(a+" a ecrit "+ this.table.get(a));
      }
   }
```

“Parcours” d’une Map (2)

ou en itérant sur les couples (“`Map.entry`”) :

```
public void afficheMap() {
   Set<Map.Entry<Auteur,Livre>> lesEntries = this.table.entrySet();
   Iterator<Map.Entry<Auteur,Livre>> it_entry = lesEntries.iterator();
   while (it_entry.hasNext()) {
      Map.Entry<Auteur,Livre> e = it_entry.next();
      S.o.p(e.getKey()+" a ecrit "+ e.getValue());
   }
}
```

```
public void afficheMap() {
   for(Map.Entry<Auteur,Livre> entry :  this.table.entrySet()) {
      S.o.p(entry.getKey()+" a ecrit "+ entry.getValue());
   }
}
```


Ca marche !

```
package essais;
import java.util.*;
public class TestMapSimple {
   private Map<Integer,String> m = new HashMap<Integer,String>();
   public void fill() {
      this.m.put(new Integer(1),"Integer :  1");
      this.m.put(new Integer(2),"Integer :  2");
      this.m.put(new Integer(1),"Integer :  1");
   }
   public void dump() {
      System.out.println("cle -> valeur");
      for(Integer key :  this.m.keySet()) {
         System.out.println(key+" -> "+this.m.get(key));
      }
   }

   public static void main (String args[]) {
      TestMapSimple tm = new TestMapSimple();
      tm.fill();
      tm.dump();
   }
} // TestMapSimple
```


Ca marche !

```
package essais;
import java.util.*;
public class TestMapSimple {
   private Map<Integer,String> m = new HashMap<Integer,String>();
   public void fill() {
      this.m.put(new Integer(1),"Integer :  1");
      this.m.put(new Integer(2),"Integer :  2");
      this.m.put(new Integer(1),"Integer :  1");
   }
   public void dump() {
      System.out.println("cle -> valeur");
      for(Integer key :  this.m.keySet()) {
         System.out.println(key+" -> "+this.m.get(key));
      }
   }

   public static void main (String args[]) {
      TestMapSimple tm = new TestMapSimple();
      tm.fill();
      tm.dump();
   }
} // TestMapSimple
```

```
+----------------
+ cle -> valeur
| 1 -> Integer :  1
| 2 -> Integer :  2
+----------------
```


Damned !

```
package essais;
import java.util.*;
class ValueB {
   private int i = 1;
   public ValueB(int i) { this.i = i; }
   public String toString() { return "value "+this.i; }
}

public class TestMap {
   private Map<ValueB,String> m = new HashMap<ValueB,String>();

   public void fill() {
      this.m.put(new ValueB(1),"valueB : 1");
      this.m.put(new ValueB(2),"valueB : 2");
      this.m.put(new ValueB(1),"valueB : 1");
   }
   public void dump() {...  }


   public static void main (String args[]) {
      TestMap tm = new TestMap();
      tm.fill();
      tm.dump();
   }
} // TestMap
```

Damned !

```
package essais;
import java.util.*;
class ValueB {
   private int i = 1;
   public ValueB(int i) { this.i = i; }
   public String toString() { return "value "+this.i; }
}

public class TestMap {
   private Map<ValueB,String> m = new HashMap<ValueB,String>();

   public void fill() {
      this.m.put(new ValueB(1),"valueB : 1");
      this.m.put(new ValueB(2),"valueB : 2");
      this.m.put(new ValueB(1),"valueB : 1");
   }
   public void dump() {...  }


   public static void main (String args[]) {
      TestMap tm = new TestMap();
      tm.fill();
      tm.dump();
   }
} // TestMap
```

```
+-------------------
+ cle -> valeur
| value 1 -> balueB : 1
| value 2 -> valueB : 2
| value 1 -> valueB : 1
+------------------
```


Résolution des collisions par chaînage

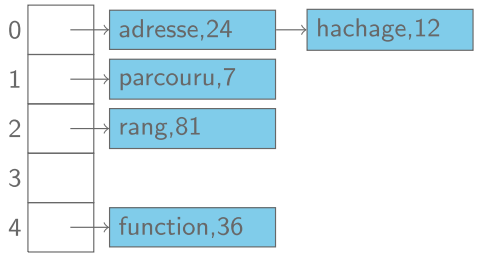
- la table a la capacité de grandir
- si une alvéole est déjà occupée, on ajoute ≪ dans la même alvéole ≫ le nouveau couple `<clé,valeur>`

Traitement des collisions

Si deux clés (non homonymes) aboutissent à la même adresse : il y a collision.

Schéma de principe

La table est un tableau de listes chaînées de couples.



k	v	$h(k)$
hachage	12	0
fonction	36	4
parcouru	7	1
rang	81	2
adresse	24	0

Calcul du hachage pour des objets quelconques

- il faut avoir une valeur unique pour chaque objet
- dans certains langages ce calcul est implicite (souvent en utilisant l'adresse mémoire où est rangé l'objet)
- mais attention, deux objets créés identiquement n'ont pas nécessairement même hash code,
  en Java par exemple il est nécessaire de redéfinir la méthode `hashCode`, en Python la méthode `__hash__`
- mais encore attention, il peut aussi être nécessaire de redéfinir l'égalite au sens logique des objets,
  en Java la méthode `equals`, en Python la méthode `__equals__`

(voir cours de POO pour les tables de hachage en Java)

En conclusion, il faut être en capacité

- de **calculer une adresse** à partir de la clé pour ranger
- de **tester l'égalité** entre deux clés pour le prédicat de présence

Explications

Dans les `HashMap`

- le "*hashCode*[2]" de la clé est utilisé pour retrouver rapidement la clé (sans parcourir toute la structure).
  ↪ par défaut la valeur de la référence.
- la méthode `equals()` est utilisée pour gérer les collisions (2 clés avec même *hashcode*)

donc pour que 2 objets soient considérés comme des clés identiques, **il faut** :

- qu'ils produisent le **même** hashcode
- qu'ils soient **égaux** du point de vue de `equals`

$\Longrightarrow$ définir des fonctions `hashCode()` (aïe !) et `equals(Object o)` adaptées pour les clés des `HashMap` (et donc valeurs des `HashSet`)

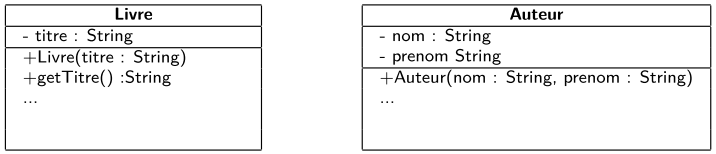
[2] `int` obtenu à partir de l'objet par une fonction de hachage


```
package essais;
import java.util.*;
class ValueD {
   private int i = 1;
   public ValueD(int i) { this.i = i; }
   public String toString() { return "value "+this.i; }

   public boolean equals(Object o) {
      return (o instanceof ValueD) && (this.i == ((ValueD) o).i);
   }
   public int hashCode() {
      return this.i;
   }
}

public class TestMapBis {
   private Map<ValueD,String> m = new HashMap<ValueD,String>();
   public void fill() {
      this.m.put(new ValueD(1),"valueD : 1");
      this.m.put(new ValueD(2),"valueB : 2");
      this.m.put(new ValueD(1),"valueD : 1");
   }
   public void dump() {...  }

   public static void main (String args[]) {
      TestMapBis tm = new TestMapBis();
      tm.fill();
      tm.dump();
   }
} // TestMap
```



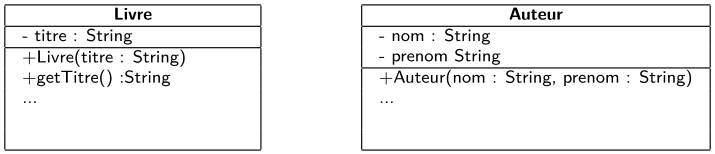
```
package essais;
import java.util.*;
class ValueD {
   private int i = 1;
   public ValueD(int i) { this.i = i; }
   public String toString() { return "value "+this.i; }

   public boolean equals(Object o) {
      return (o instanceof ValueD) && (this.i == ((ValueD) o).i);
   }
   public int hashCode() {
      return this.i;
   }
}

public class TestMapBis {
   private Map<ValueD,String> m = new HashMap<ValueD,String>();
   public void fill() {
      this.m.put(new ValueD(1),"valueD : 1");
      this.m.put(new ValueD(2),"valueB : 2");
      this.m.put(new ValueD(1),"valueD : 1");
   }
   public void dump() {...  }

   public static void main (String args[]) {
      TestMapBis tm = new TestMapBis();
      tm.fill();
      tm.dump();
   }
} // TestMap
```

```
+------------------
+ cle -> valeur
| value 1 -> valueD : 1
| value 2 -> valueD : 2
+------------------
```




```
Map <Auteur,Livre> table = new HashMap <Auteur,Livre>();   // associe un Auteur à un Livre
Auteur auteur = new Auteur("Tolkien","JRR");
Livre livre = new Livre("Le Seigneur des Anneaux");
table.put(auteur,livre);
Auteur secondAuteur = new Auteur("Tolkien","JRR");
```

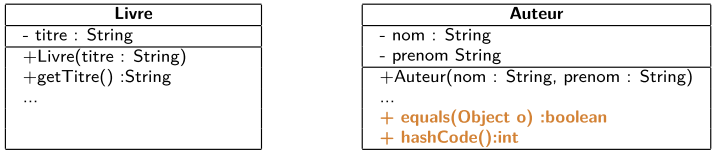

Collections Listes Itérateurs Ensembles Tables Attention !!! Typage




```
Map <Auteur,Livre> table = new HashMap <Auteur,Livre>();   // associe un Auteur à un Livre
Auteur auteur = new Auteur("Tolkien","JRR");
Livre livre = new Livre("Le Seigneur des Anneaux");
table.put(auteur,livre);
Auteur secondAuteur = new Auteur("Tolkien","JRR");
S.o.p(table.get(secondAuteur));                            // qu'est-ce qui est affiché ?
```

Université Lille 1 - Licence mention Informatique Programmation Orientée Objet 26

Collections Listes Itérateurs Ensembles Tables Attention !!! Typage



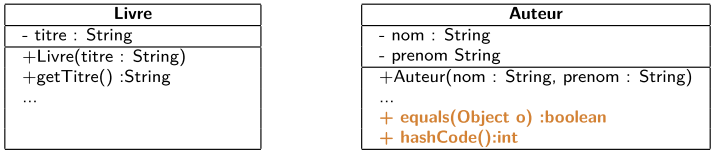

```
Map <Auteur,Livre> table = new HashMap <Auteur,Livre>();   // associe un Auteur à un Livre
Auteur auteur = new Auteur("Tolkien","JRR");
Livre livre = new Livre("Le Seigneur des Anneaux");
table.put(auteur,livre);
Auteur secondAuteur = new Auteur("Tolkien","JRR");
S.o.p(table.get(secondAuteur));                            // qu'est-ce qui est affiché ?
```

Réutiliser des méthodes hashCode existantes :

```
public class Auteur {
   ...
   public boolean equals(Object o) {
      if (o instanceof Auteur) {
         Auteur lAutre = (Auteur) o;
         return this.nom.equals(lAutre.nom) && this.prenom.equals(lAutre.prenom);
      } else return false;
   }
```

Université Lille 1 - Licence mention Informatique Programmation Orientée Objet 26

Collections Listes Itérateurs Ensembles Tables Attention !!! Typage




```
Map <Auteur,Livre> table = new HashMap <Auteur,Livre>();   // associe un Auteur à un Livre
Auteur auteur = new Auteur("Tolkien","JRR");
Livre livre = new Livre("Le Seigneur des Anneaux");
table.put(auteur,livre);
Auteur secondAuteur = new Auteur("Tolkien","JRR");
S.o.p(table.get(secondAuteur));                            // qu'est-ce qui est affiché ?
```

Réutiliser des méthodes hashCode existantes :

```
public class Auteur {
   ...
   public boolean equals(Object o) {
      if (o instanceof Auteur) {
         Auteur lAutre = (Auteur) o;
         return this.nom.equals(lAutre.nom) && this.prenom.equals(lAutre.prenom);
      } else return false;
   }
   public int hashCode() {
      return (this.nom+"@#"+this.prenom).hashCode(); // par exemple
   }
}
```

Université Lille 1 - Licence mention Informatique Programmation Orientée Objet 26

Livre
- titre : String
+Livre(titre : String) +getTitre() :String ... + equals(Object o):boolean + hashCode():int

Auteur
- nom : String - prenom String
+Auteur(nom : String, prenom : String) ... + equals(Object o) :boolean + hashCode():int

```
Map <Auteur,Livre> table = new HashMap <Auteur,Livre>();   // associe un Auteur à un Livre
Auteur auteur = new Auteur("Tolkien","JRR");
Livre livre = new Livre("Le Seigneur des Anneaux");
table.put(auteur,livre);
Auteur secondAuteur = new Auteur("Tolkien","JRR");
S.o.p(table.get(secondAuteur));                            // qu'est-ce qui est affiché ?
```

Réutiliser des méthodes `hashCode` existantes :

```
public class Auteur {
   ...
   public boolean equals(Object o) {
      if (o instanceof Auteur) {
         Auteur lAutre = (Auteur) o;
         return this.nom.equals(lAutre.nom) && this.prenom.equals(lAutre.prenom);
      } else return false;
   }
   public int hashCode() {
      return (this.nom+"@#"+this.prenom).hashCode(); // par exemple
   }
}
```

`equals` et `hashCode` devraient (doivent) être implémentées systématiquement.

Ca marche ! (ensembles)

```
package essais;
import java.util.*;

public class TestSetSimple {

   private Set<Integer> s = new HashSet<Integer>();
   public void fill() {
      this.s.add(new Integer(1));
      this.s.add(new Integer(2));
      this.s.add(new Integer(1));
   }
   public void dump() {
      for(Integer entier :  this.s) {
         System.out.println("value "+entier);
      }
   }
   public static void main (String args[]) {
      TestSetSimple ts = new TestSetSimple();
      ts.fill();
      ts.dump();
   }
} // TestSetSimple
```


Ca marche ! (ensembles)

```
package essais;
import java.util.*;

public class TestSetSimple {

   private Set<Integer> s = new HashSet<Integer>();
   public void fill() {
      this.s.add(new Integer(1));
      this.s.add(new Integer(2));
      this.s.add(new Integer(1));
   }
   public void dump() {
      for(Integer entier :  this.s) {
         System.out.println("value "+entier);
      }
   }
   public static void main (String args[]) {
      TestSetSimple ts = new TestSetSimple();
      ts.fill();
      ts.dump();
   }
} // TestSetSimple
```

```
+--------
| value 2
| value 1
+--------
```


Damned ! (Ensembles)

- Les `HashSet` sont implémentés via une `HashMap` (efficacité)

```
package essais;
public class ValueB {
   private int i = 1;
   public ValueB(int i) { this.i = i; }
   public String toString() { return "value "+i; }
}
...
package essais;
import java.util.*;
public class TestSet {
   private Set<ValueB> s = new HashSet<ValueB>();
   public void fill() {
      this.s.add(new ValueB(1));
      this.s.add(new ValueB(2)); this.s.add(new ValueB(1));
   }
   public void dump() {
      for(ValueB vb :  this.s) {
         System.out.println(vb);
      }
   }
   public static void main (String args[]) {
      TestSet ts = new TestSet();
      ts.fill();
      ts.dump();
   }
} // TestSet
```

Damned ! (Ensembles)

- Les `HashSet` sont implémentés via une `HashMap` (efficacité)

```
package essais;
public class ValueB {
   private int i = 1;
   public ValueB(int i) { this.i = i; }
   public String toString() { return "value "+i; }
}
...
package essais;
import java.util.*;
public class TestSet {
   private Set<ValueB> s = new HashSet<ValueB>();
   public void fill() {
      this.s.add(new ValueB(1));
      this.s.add(new ValueB(2)); this.s.add(new ValueB(1));
   }
   public void dump() {
      for(ValueB vb :  this.s) {
         System.out.println(vb);
      }
   }
   public static void main (String args[]) {
      TestSet ts = new TestSet();
      ts.fill();
      ts.dump();
   }
} // TestSet
```

```
+--------
| value 1
| value 2
| value 1
+--------
```



```
package essais;
public class ValueD {
   private int i = 1;
   public ValueD(int i) { this.i = i; }
   public boolean equals(Object o) {
      return (o instanceof ValueD) && (this.i == ((ValueD) o).i);
   }
   public int hashCode() { return this.i; }
   public String toString() { return "value "+this.i; }
}
...
package essais;
import java.util.*;
public class TestSetBis {
   private Set<ValueD> s = new HashSet<ValueD>();
   public void fill() {
      this.s.add(new ValueD(1)); this.s.add(new ValueD(2));
      this.s.add(new ValueD(1));
   }
   public void dump() { ...  }
   public static void main (String args[]) {
      TestSetBis ts = new TestSetBis();
      ts.fill(); ts.dump();
   }
} // TestSetBis
```



```
package essais;
public class ValueD {
   private int i = 1;
   public ValueD(int i) { this.i = i; }
   public boolean equals(Object o) {
      return (o instanceof ValueD) && (this.i == ((ValueD) o).i);
   }
   public int hashCode() { return this.i; }
   public String toString() { return "value "+this.i; }
}
...
package essais;
import java.util.*;
public class TestSetBis {
   private Set<ValueD> s = new HashSet<ValueD>();
   public void fill() {
      this.s.add(new ValueD(1)); this.s.add(new ValueD(2));
      this.s.add(new ValueD(1));
   }
   public void dump() { ...  }
   public static void main (String args[]) {
      TestSetBis ts = new TestSetBis();
      ts.fill(); ts.dump();
   }
} // TestSetBis
```

```
+--------
| value 2
| value 1
+--------
```


Ensembles triés

```
package essais;
public class ValueC implements Comparable<ValueC> {
   private int i = 1;
   public ValueC(int i) { this.i = i; }
   public String toString() { return "value "+this.i; }
   public boolean equals(Object o) {
      return (o instanceof ValueC) && (this.i == ((ValueC) o).i);
   }
   public int hashCode() { return this.i; }
   public int compareTo(ValueC vc) {
      return this.i-vc.i;
   }
}
...
package essais;
import java.util.*;
public class TestTreeSet {
   private Set<ValueC> s = new TreeSet<ValueC>();
   public void fill() {
      s.add(new ValueC(1)); s.add(new ValueC(2)); s.add(new ValueC(1));
   }
   public void dump() { ...  }
   public static void main (String args[]) {
      TestTreeSet ts = new TestTreeSet();
      ts.fill(); ts.dump();
   }
} // TestTreeSet
```

Ensembles triés

```
package essais;
public class ValueC implements Comparable<ValueC> {
   private int i = 1;
   public ValueC(int i) { this.i = i; }
   public String toString() { return "value "+this.i; }
   public boolean equals(Object o) {
      return (o instanceof ValueC) && (this.i == ((ValueC) o).i);
   }
   public int hashCode() { return this.i; }
   public int compareTo(ValueC vc) {
      return this.i-vc.i;
   }
}
...
package essais;
import java.util.*;
public class TestTreeSet {
   private Set<ValueC> s = new TreeSet<ValueC>();
   public void fill() {
      s.add(new ValueC(1)); s.add(new ValueC(2)); s.add(new ValueC(1));
   }
   public void dump() { ... }
   public static void main (String args[]) {
      TestTreeSet ts = new TestTreeSet();
      ts.fill(); ts.dump();
   }
} // TestTreeSet
```

```
+--------
| value 1
| value 2
+--------
```


"Problèmes" liés au typage

- `ArrayList<String>` est un sous-type de `Collection<String>`
- `Collection<String>` **n'est pas un sous-type** de `Collection<Object>`

"Problèmes" liés au typage

- `ArrayList<String>` est un sous-type de `Collection<String>`
- `Collection<String>` **n'est pas un sous-type** de `Collection<Object>`

Conséquence,

```
Collection<Hobbit> colHob = new ArrayList<Hobbit>();  // ok
Collection<Object> c = new ArrayList<Hobbit>();       // ne compile pas !!!
Collection<Object> c = colHob;                        // idem :  incompatible types
```

et donc :

```
public void dump(Collection<Object> c) {
   for (Object o :  c) {
      System.out.println(o);
   }
}
```

ne peut pas prendre pour paramètre autre chose que `Collection<Object>`.
`xxx.dump(new ArrayList<Hobbit>())` **ne compile pas** !

"Problèmes" liés au typage

- `ArrayList<String>` est un sous-type de `Collection<String>`
- `Collection<String>` **n'est pas un sous-type** de `Collection<Object>`

Conséquence,

```
Collection<Hobbit> colHob = new ArrayList<Hobbit>();  // ok
Collection<Object> c = new ArrayList<Hobbit>();       // ne compile pas !!!
Collection<Object> c = colHob;                        // idem :  incompatible types
```

et donc :

```
public void dump(Collection<Object> c) {
   for (Object o :  c) {
      System.out.println(o);
   }
}
```

ne peut pas prendre pour paramètre autre chose que `Collection<Object>`.
`xxx.dump(new ArrayList<Hobbit>())` **ne compile pas** !

- `Collection<Object>` ne signifie pas
  "*n'importe quelle collection pourvue qu'elle contienne des objets*"
  mais bien "*collection d'`Objects`*"

- Comment exprimer "*n'importe quelle collection*" ?
  càd le type qui réunit toutes les collections

Collections Listes Itérateurs Ensembles Tables Attention !!! Typage

- Comment exprimer "*n'importe quelle collection*" ?
  càd le type qui réunit toutes les collections

`Collection<?>` (collection d'*inconnus*, **?** = joker)

mais la seule garantie sur les éléments c'est que ce sont des `Objects` !

```
public void dump(Collection<?> c) {
   for (Object o :  c) {
      System.out.println(o);
   }
}
```

`xxx.dump(new ArrayList<Hobbit>())` est légal.

Collections Listes Itérateurs Ensembles Tables Attention !!! Typage

- Comment exprimer "*n'importe quelle collection*" ?
  càd le type qui réunit toutes les collections

`Collection<?>` (collection d'*inconnus*, **?** = joker)

mais la seule garantie sur les éléments c'est que ce sont des `Objects` !

```
public void dump(Collection<?> c) {
   for (Object o :  c) {
      System.out.println(o);
   }
}
```

`xxx.dump(new ArrayList<Hobbit>())` est légal.
Mais :

```
Collection<?> c = new ArrayList<Hobbit>();
c.add(new Hobbit(...));          // ne compile pas
```


Collections Listes Itérateurs Ensembles Tables Attention !!! Typage

```
public void recycleAll(Collection<Recyclable> c) {
   for (Recycable o :  c) {
      o.recycle();
   }
}
```

permet :

```
List<Recyclable> trashcan = new ArrayList<Recyclable>();
xxx.recycleAll(trashcan);
```

```
public void recycleAll(Collection<Recyclable> c) {
   for (Recycable o :  c) {
      o.recycle();
   }
}
```

permet :

```
List<Recyclable> trashcan = new ArrayList<Recyclable>();
xxx.recycleAll(trashcan);
```

mais pas :

```
List<Paper> paperBasket = new ArrayList<Paper>();
xxx.recycleAll(paperBasket); // ne compile pas, même raison
```


Comment exprimer :

une collection de *n'importe quoi du moment que c'est* `Recyclable`
càd du moment que c'est un **sous-type** de `Recyclable`

Comment exprimer :

une collection de *n'importe quoi du moment que c'est* `Recyclable`
càd du moment que c'est un **sous-type** de `Recyclable`

```
Collection<? extends Recyclable>
```


Comment exprimer :

une collection de *n'importe quoi du moment que c'est* `Recyclable`
càd du moment que c'est un **sous-type** de `Recyclable`

```
Collection<? extends Recyclable>
```

On a alors :

```
public void recycleAll(Collection<? extends Recyclable> c) {
   for (Recycable o :  c) {
      o.recycle();
   }
}
```

et alors `xxx.recycleAll(new ArrayList<Paper>())` est légal.

NB : Il existe **super** pour réclamer un type *plus général*.

Listes triées

méthode statique `sort` de la classe utilitaire `Collections`
(tri par fusion modifié $(\sim n \log n)$)

- `Collections.sort(List<T> list)`
  $\hookrightarrow$ utilisation de `compareTo`, les objets doivent être mutuellement "`Comparable`".
- `Collections.sort(List<T> list, Comparator<? super T> comp)`

Interface `Comparator<T>`
pour définir un opérateur de relation d'ordre **totale**

- `int compare(T o1, T o2)`