

## Collections version $\leq 1.4$

### Programmation Orientée Objet



Jean-Christophe Routier  
Licence mention Informatique  
Université de Lille 1



# Premier regard sur les collections

- ▶ Une collection est un groupe d'objets (ses éléments).
- ▶ On trouve des collections de comportements différents (listes, ensembles, etc.)
- ▶ D'autres structures permettent de regrouper des objets sans être des collections : les "Map".
- ▶ On trouve (avec d'autres) ces types dans le paquetage

`java.util`

- ▶ Une interface `java.util.Collection` définit le contrat des collections.

# Méthodes principales de Collection

**boolean add(Object o)** *Ensures that this collection contains the specified element (optional operation).*

**boolean contains(Object o)** *Returns true if this collection contains the specified element.*

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

**Iterator 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.*

# List

- **List** = collection ordonnée d'objets

**ArrayList** pour accès direct

**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** quand nombreuses insertions et suppressions dans la liste

Quoi utiliser ?

# Quoi utiliser ?

Type	Get	Iteration	Insert	Remove
array	1430	3850	na	na
ArrayList	3070	12200	500	46850
LinkedList	16320	9110	110	60
Vector	4890	16250	550	46850

Quoi utiliser ?

# Méthodes complémentaires

```
Object get(int index)
```

```
boolean remove(int index)
```

```
int indexOf(Object element)
```

**ListIterator** pour parcours avant/arrière (`previous()`, `hasPrevious()`)

# Itérateur

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

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

**Object next()** *Returns the next element in the iteration.*

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

Les `Iterator` sont **fail-fast** : si, après que l'itérateur ait été créé, la collection attachée est modifiée autrement que par les `add` et `remove` de l'itérateur alors l'itérateur lance une `ConcurrentModificationException`.

Donc échec rapide et propre plutôt que de risquer l'incohérence.

# Usage

en java  $\leq 1.4$  : les collections **ne sont pas typées**

Elles contiennent des instances de la classe `Object`.

⇒ Il est donc indispensable de “downcaster” les objets récupérés avant de les manipuler.

```
Collection trashcan = new ArrayList();
```

```
trashcan.add(new Paper());           // upcast vers Object
```

```
trashcan.add(new Battery());        // automatique
```

```
Iterator it = trashcan.iterator();  // itérateur sur la collection
```

```
while(it.hasNext()) {
```

```
    Object o = it.next();           // récupération d'un élément
```

```
    (Recyclable) o).recycle();     // downcast (explicite) nécessaire
```

```
}
```



Créer une `List` typée

# Créer une `List` typée

Agréger une `List` et fournir les fonctions interfaces désirées en les redirigeant vers celle-ci.

Créer la classe `HobbitIterator` selon le même principe.

Si on veut une vraie *collection*,  $\Rightarrow$  implémenter `Collection` et fournir **toutes** les fonctions de l'interface.  
Pas possible ici (type de retour)...

```
public class Hobbit {
    private String name;
    public Hobbit(String name) {
        this.name = name;
    }
}

.....
import java.util.*;
public class HobbitList {
    private List myList = new ArrayList();
    public void add(Hobbit hobbit) {
        this.myList.add(hobbit);
    }
    public HobbitIterator iterator() {
        return new HobbitIterator(this.myList);
    }
    public Hobbit get(int index) {...}{
        return (Hobbit) this.myList.get(index);
    }
}
```

# Set

- **Set** collection d'objets sans répétition de *valeurs*

**HashSet** 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** ordonné 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)

# Map

“listes associatives”, dictionnaire, index, tables, etc.

groupe d’associations (Clé, Valeur)

Les “Map” ne sont pas des `Collections`.

$\implies$  pas d’itérateur.

**HashMap** 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** clés ordonnées

**API Doc** *This implementation provides guaranteed  $\log(n)$  time cost for the `containsKey`, `get`, `put` and `remove` operations.*

`Object get(Object key)` récupère la valeur associé à une clé

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

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

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

`Collection values()` renvoie la **collection** des valeurs

`Set keySet()` renvoie l'**ensemble** des valeurs

`Set entrySet()` renvoie l'**ensemble** des couples (clé,valeurs) (objets `Map.Entry`)

# “Parcours” d’une Map

```
Map table = ...; // associe Auteur (clé) à Livre
...
```

```
public void afficheMap() {
    Set lesCles = this.table.keySet();
    Iterator it_cle = lesCles.iterator();
    while (it_cle.hasNext()) {
        Auteur a = (Auteur) it_cle.next();
        S.o.p(a + " -> " + this.table.get(a));
    }
}
```

ou

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

Ca marche !

# Ca marche !

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

public class TestSetSimple {

    private Set s = new HashSet();
    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(Iterator it = this.s.iterator(); it.hasNext();) {
            System.out.println("value "+it.next());
        }
    }
    public static void main (String args[]) {
        TestSetSimple ts = new TestSetSimple();
        ts.fill();
        ts.dump();
    }
} // TestSetSimple
```

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

Damned !

# Damned !

```

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 s = new HashSet();
    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(Iterator it = this.s.iterator(); it.hasNext();) {
            System.out.println(it.next());
        }
    }
    public static void main (String args[]) {
        TestSet ts = new TestSet();
        ts.fill();
        ts.dump();
    }
} // TestSet

```

```

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

```

# Explications

- ▶ Les `HashSet` sont implémentés via une `HashMap` pour une plus grande efficacité.
- ▶ Dans les `HashMap`, le “*hashCode*<sup>1</sup>” de la clé est utilisé pour retrouver rapidement la clé (sans parcourir toute la structure).  
 ↪ par défaut la valeur de la référence.
- ▶ De plus la méthode `equals()` est utilisée pour gérer les collisions (2 clés avec même *hashCode*)

**donec** 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`

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

---

<sup>1</sup>produit à partir de l’objet par une fonction de hachage en un `int` “quasiment unique”



## Explications

```
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 s = new HashSet();
    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
+-----
```

## Explications

```

package essais;
public class ValueC implements Comparable {
    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 ValueD) && (this.i == ((ValueD) o).i);
    }
    public int hashCode() { return(i); }
    public int compareTo(Object o) {
        return i-((ValueC) o).i;
    }
}
...
package essais;
import java.util.*;
public class TestTreeSet {
    private Set s = new TreeSet();
    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
+-----

```



# Listes triées

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

- ▶ `Collections.sort(List list)`  
↪ utilisation de `compareTo`, les objets doivent être mutuellement “Comparable”.
- ▶ `Collections.sort(List list, Comparator comp)`

Interface **Comparator**  
pour relation d'ordre **totale**

- ▶ `int compare(Object o1, Object o2)`
- ▶ `equals(Object obj)`