

Collections

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1



Université
Lille1
Sciences et Technologies

IEEA - FIL
Informatique

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<E>` définit le contrat des collections.
- ▶ A partir de java 1.5, les collections sont **typées**.
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 $\exists e \ (o==null? \ e==null : \ o.equals(e))$*
 \implies 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.*

List<E>

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

2 classes :

ArrayList<E> 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<E> 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

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).*

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()) {
    (it.next()).recycle();           // it.next() du type Recyclable
}
```

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

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

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

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  
}
```


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`.

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 itLivre = l.iterator();
Livre l = it.next();           // ok
l.add(new Livre(...));        // modification de la liste
                                // ==> corruption de l'itérateur
l = it.next();                 // -> ConcurrentModificationException levée
```

List<E> : Méthodes complémentaires

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

E `get(int index)` fournit l'index-ième élément de la liste.

`IndexOutOfBoundsException` - si ($index < 0$ || $index \geq size()$)

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

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

ListIterator<E> pour parcours avant/arrière
(méthodes `previous()`, `hasPrevious()`)

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>();
```

<code>l.add(12);</code>	<i>correspond à</i>	<code>l.add(new Integer(12));</code>
<code>int i = l.get(0);</code>		<code>int i = l.get(0).intValue();</code>

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)

Map<K, V>

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

groupe d'associations (Clé, Valeur)

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

⇒ pas d'itérateur.

HashMap<K, V> 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 associé à une clé

`void put(K key, V 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<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>`)

Livre
...
+Livre(titre : String)
+getTitre() :String
...

```

// 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”

```
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 manipulant 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 écrit "+ e.getValue());  
    }  
}
```

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

Ca marche !

Ca marche !

```

package essais;
import java.util.*;
public class TestMapSimple {
    public 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 !

Damned !

```

package essais;
import java.util.*;
class ValueB {
    public int i = 1;
    public ValueB(int i) { this.i = i; }
    public String toString() { return "value "+this.i; }
}
public class TestMap {
    public 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 -> valueB : 1
| value 2 -> valueB : 2
| value 1 -> valueB : 1
+-----+

```

Explications

- ▶ Dans les `HashMap`, le “*hashCode*¹” 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*)

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`

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

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

Explications

```

package essais;
import java.util.*;
class ValueD {
    public 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 {
    public 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
+-----+

```

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 !

Damned ! (Ensembles)

- Les HashSet sont implémentés via une HashMap pour une plus grande 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();
    }
}
```

```
+-----+
| 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

```

```

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

```


Damned !

Ensembles triés

```

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<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>`

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, soit :

```
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

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
```

```
public void recycleAll(Collection<Recyclable> c) {  
    for (Recyclable 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

Collection<? extends Recyclable>)

On a alors :

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

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

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

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
.
```

Listes triées

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

- ▶ `Collections.sort(List<T> list)`
↪ 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)`