

Programmation Par Objets

package java.util
utilitaires de Arrays
structures de données génériques

Utilitaires et collections Java

- Tableaux
 - intégrés dans le langage : «Objets» de taille fixe
 - type des éléments : primitif (homogène) ou objets (hétérogène)

- Bibliothèque (package) java.util
 - Arrays : utilitaires de manipulation de tableaux (égalité, tri, recherches)
 - Structures de données dynamiques (« collections »)
 - Set : ensembles
 - List : listes
 - Map : tables d'association
 - génériques depuis Java 5.0

Utilitaires sur les tableaux

- **Fonctionnalités offertes par la classe `java.util.Arrays`**
 - **sous la forme de fonctions static**
 - **définies pour des tableaux d'objets, mais surchargées pour tous les types primitifs**
- **Egalité de contenu**

```
boolean Arrays.equals(t1,t2) <=>
(t1.length == t2.length)
et (t1[i].equals(t2[i]), ∀ i<t1.length)
```
- **Rappel : la méthode `equals` de `Object` teste par défaut l'égalité physique (même instance au sens de `==`).**
- **il est possible de la redéfinir pour tester l'égalité logique d'objets.**
- **Exemple : égalité de contenu entre chaînes définie dans `String`**

Egalité de tableaux : exemple

```
public class Ouvrage {
    protected String titre, auteur;
    public Ouvrage(String tit, String aut) {
        titre=tit; auteur=aut;
    }
    public String toString() {
        return titre+" "+auteur;
    }
    public boolean equals(Object o) {
        return ( (o instanceof Ouvrage)
            && (auteur.equals(((Ouvrage)o).getAuteur())) //downcast
            && (titre.equals(((Ouvrage)o).getTitre())) );
    }
    ...}

```

Egalité de tableaux : exemple

```
Ouvrage t1[] = {  
    new Ouvrage("Germinal", "Zola"),  
    new Ouvrage("C", "Kernighan"),  
    new Ouvrage("Java", "Eckel") };
```

```
Ouvrage t2[] = {  
    new Ouvrage("Germinal", "Zola"),  
    new Ouvrage("C", "Kernighan"),  
    new Ouvrage("Java", "Eckel") };
```

- `t1==t2` **est** `false`
 - `<=> t1.equals(t2)` (égalité physique par défaut)
 - tout comme : `t1[i]==t2[i]`
- `Arrays.equals(t1,t2)` **est** `true` (égalité logique)
 - tout comme `t1[i].equals(t2[i])` (itérée sur les éléments)

Tri

```
void Arrays.sort(Object t[])
```

- Les éléments du tableau doivent offrir une relation d'ordre en implémentant la méthode `compareTo` spécifiée par l'interface `Comparable <T>` :

```
public interface java.lang.Comparable<T> {  
    int compareTo(T obj);  
    // < 0 si this < obj  
    // = 0 si this = obj  
    // > 0 si this > obj  
}
```

Tri : exemple

```
// sachant :
class String implements Comparable<String>

// ordre sur les ouvrages <=> ordre sur leur auteur :
public class Ouvrage implements Comparable<Ouvrage> {
    public int compareTo(Ouvrage obj) {
        return
            auteur.compareTo(obj.getAuteur());
    }...}

//application
Arrays.sort(t1);

// resultat
Java Eckel
C Kernighan
Germinal Zola
```

Ordres multiples

- La technique du `compareTo` par l'interface **Comparable** est dite «de l'ordre naturel» (méthode interne de l'objet).
- Elle pose problème si:
 - on veut trier des objets dont la classe n'implémente pas `Comparable`
 - ou si l'ordre naturel ne convient pas
 - ou encore si l'on désire ordonner les objets selon plusieurs critères.
- Une autre technique est offerte sous la forme de «callbacks» au travers «d'objets comparateurs» qui doivent implémenter l'interface **Comparator<T>**:

```
interface java.util.Comparator<T> {
    public int compare(T o1, T o2);
    // < 0 si o1 < o2
    // = 0 si o1 = o2
    // > 0 si o1 > o2
}
```

Ordres multiples

- Le tri de tableaux s'effectue alors en fournissant l'objet comparateur en paramètre :

```
<T> void Arrays.sort(T[] t, Comparator<T> c)
```

- Exemple

```
// nouvel ordre des ouvrages sur leur titre :
public class CompareurDeTitres
    implements Comparator<Ouvrage>{
    public int compare(Ouvrage o1, Ouvrage o2) {
        return o1.getTitre().compareTo(o2.getTitre());
    }
}

//application
Arrays.sort(t2, new CompareurDeTitres());
```

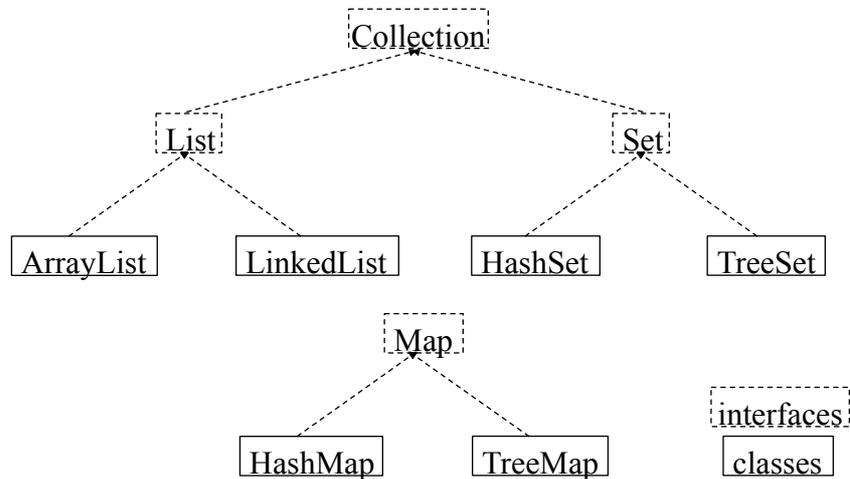
Recherches ordonnées

- recherche d'un objet dans un tableau ordonné par ordre naturel
`int Arrays.binarySearch(Object t[], Object obj)`
- recherche d'un objet dans un tableau ordonné par comparateur
`<T> int Arrays.binarySearch(T t[], T obj, Comparator<T> c)`
- Ces méthodes renvoient :
 - l'indice de l'élément s'il est trouvé
 - (- <indice d'insertion> - 1) sinon, avec:
 - <indice d'insertion> = 0 si `obj < t[0]`, `t.length` si `obj > t[t.length-1]`
 - `t[<indice d'insertion> - 1] < obj < t[<indice d'insertion>]`, sinon
- La recherche ordonnée sur un tableau non trié (selon l'ordre correspondant) est imprédictible!

```
Arrays.binarySearch(t1,new Ouvrage("Germinal","Zola"))=2
Arrays.binarySearch(t1,new Ouvrage("Parapente","AliGali"))=-1
Arrays.binarySearch(t1,new Ouvrage("IA","Nilsson"))=-3

Arrays.binarySearch(t2,new Ouvrage("Germinal","Zola"),
    new CompareurDeTitres())=1
```

Bibliothèque de structures de données



Collection

- Les collections sont des regroupements dynamiques d'objets.
- Les formes les plus courantes sont les listes (doublons d'éléments possibles) et les ensembles (sans doublons).
- Les collections contiennent des objets
- pour gérer des types primitifs utiliser les classes wrapper
- Avant 5.0 : le type des éléments est indifféremment Object
 - pas de contrôle statique de type
 - à contrôler dynamiquement « à la main » par casts
- Depuis 5.0 : les collections sont génériques:
 - paramétrées par le type des éléments
 - contrôle de type correspondant sur les opérations de manipulation.

Collection

- L'interface `Collection<E>` spécifie les fonctionnalités abstraites communes aux classes de collections :

```
public interface Collection<E>
    public boolean add(E o)
    public boolean remove(Object o)
    public boolean contains(Object o) //par equals()
    public int size()
    public void clear()
    public boolean isEmpty()
    public Object[] toArray() //inverse de Arrays.asList(t)
    public boolean equals(Object o)
```

- Il existe des versions itérées de `add`, `contains`, `remove` : suffixées par `All`

```
boolean addAll(Collection<? extends E> c)
boolean containsAll(Collection<?> c)
<=> c inclus dans this
```

Les listes : `java.util.List`

- Les listes sont des collections d'objets (avec doublons possibles) ordonnées de manière externe par indice de rangement.
- Elles sont issues d'une même interface `List<E>` qui ajoute à `Collection` les opérations d'accès direct indicé (`get(i)`, `add(i,x)`, `set(i,x)`, ...)
- Deux classes de listes :
 - listes chaînées: `LinkedList<E>`
plus performantes sur les opérations de mise à jour (ajout/retrait)
 - listes contiguës: `ArrayList<E>`
plus performantes sur les opérations d'accès indicé (voir aussi la classe `Vector` plus ancienne)

Les listes : java.util.List

```
public interface List<E> extends Collection<E>
    public void add(int index, E element)
        // sachant que add ajoute en queue
    public E get(int index)
        throws IndexOutOfBoundsException
    public E set(int index, E element)
        throws IndexOutOfBoundsException
    public E remove(int index)
        throws IndexOutOfBoundsException
    public int indexOf(Object o)
        throws ClassCastException
        // indice de la 1ere occurrence
        // -1 si !this.contains(o)
    public List<E> sublist(int fromIndex, int toIndex)
        throws IndexOutOfBoundsException
```

Exemple

```
import java.util.*;
public class Circuit {
    protected List<Porte>
        composants = new ArrayList<Porte>();
    public void brancher(Porte p) {
        composants.add(p);
    }
    public void remplacer(int i, Porte p) {
        composants.set(i,p);
    }
    public void run() {
        for (int i=0;i<composants.size();i++) {
            composants.get(i).run();
        }
    }
}
```

Itération sur les éléments

- Les collections offrent une interface d'itération sur leurs éléments

```
public interface Collection<E> extends Iterable<E> {  
    public Iterator<E> iterator();  
}
```

- Interface d'itération

```
public interface Iterator<E> {  
    public boolean hasNext()  
    public E next() throws NoSuchElementException  
}
```

- Exemple de la classe Circuit

```
public void run() {  
    Iterator<Porte> iter = composants.iterator();  
    while (iter.hasNext()) iter.next().run();  
    et même en 5.0, comme tout Iterable (tableaux, collections, ...):  
    for(Porte p : composants) p.run();  
}
```

Collections d'éléments de type primitif

- type primitif : utiliser les classes wrappers

```
ArrayList<Double> l = new ArrayList<Double>();
```

- en 5.0 l'"auto-boxing/unboxing" n'oblige pas à wrapper/dewrapper

```
double x;  
l.add(new Double(x)); // wrapping ou depuis 5.0 plus simplement:  
l.add(x); // "autoboxing" » (automatique)
```

```
// restitution : somme des elements  
double s=0.0;  
for(int i=0;i<l.size();i++)  
    // s=s+l.get(i).doubleValue(); // unwrapping, ou depuis 5.0 :  
    s=s+l.get(i); // "auto-unboxing" (automatique)
```

```
// par iteration "for each"  
for(double x : l) s=s+x;
```

Les ensembles : java.util.Set

- Tout comme les List, les Set implémentent l'interface Collection mais interdisent les doublons au sens de equals:
 $\forall o1, o2, !o1.equals(o2)$
 - Cette propriété est vérifiée lors des ajouts
 - mais elle n'est pas maintenue, la modification d'objets peut la rendre caduque : deux éléments peuvent devenir égaux («aliasing »)
- Contrairement aux List, l'ordre des éléments est interne et donc **pas d'accès direct** indiqué.
- Il existe principalement deux sortes de Set :
 - TreeSet
 - HashSet

HashSet / TreeSet

- HashSet<E>
 - Les éléments sont rangés de façon hachée par appel de leur méthode hashCode() (de Object, redéfinissable).
 - Les performances de HashSet sont meilleures que TreeSet.
 - Ensembles non ordonnés
- TreeSet<E>
 - Ou ensembles ordonnés, les éléments sont rangés dans un arbre binaire ordonné, ce qui assure des manipulations en $\log_2(n)$.
 - L'ordre des éléments est :
 - soit leur ordre naturel (compareTo de Comparable) si aucun Comparator n'est fourni au constructeur. La classe E des éléments doit alors implémenter Comparable<E>.
 - soit déterminé par un Comparator d'éléments fourni au constructeur:

```
public TreeSet<E>(Comparator<E> c)
```

Set : exemple

- Un ensemble d'ouvrages ordonné selon leur ordre naturel (par auteur)
- construit sur le tableau t1 (cf. chapitre Arrays)

```
TreeSet<Ouvrage> ensemble
    = new TreeSet<Ouvrage>(Arrays.asList(t1));
ensemble.add(new Ouvrage("Parapente", "Ali Gali"));
for(Ouvrage o : ensemble) System.out.println(o);

// resultats...
Parapente Ali Gali
Java Eckel
C Kernighan
Germinal Zola
```

Les utilitaires de la classe Collections

- Tout comme Arrays la classe Collections (à ne pas confondre avec l'interface Collection) offre les mêmes utilitaires (static) sur les List :
 - tri : sort sur ordre naturel ou par Comparator
 - recherche ordonnée : binarySearch sur ordre naturel ou par Comparator
- Ces méthodes ont les mêmes profils que dans Arrays mais sont paramétrées par des List
- Extraits (ordre naturel) :

```
public class Collections {
    public static void sort(List<E> list)
    public static int binarySearch(List<E> list, E x)
    ...}
```
- Ces opérations permettent de trier des listes de l'extérieur
- Contrairement aux TreeSet elles ne sont pas maintenues ordonnées en interne.

Les tables d'association : java.util.Map

- Elles permettent de maintenir des associations clé-valeur <K,V>:
 - chaque clé est unique
 - les clés et valeurs sont des objets (wrapper si types de bases)
- L'interface abstraite Map spécifie les opérations communes aux classes de tables d'association :

```
public interface Map<K,V> {  
    V put(K key, V value);  
    V get(Object key); //!this.containsKey(key)=>null  
    boolean containsValue(Object value)  
    boolean containsKey(Object key);  
    V remove(Object key);  
    Set<K> keySet() //l'ensemble des clés  
    Collection<V> values() // la liste des valeurs  
    ...}
```

HashMap / TreeMap

- Il existe principalement deux sortes de tables : les **HashMap** et les **TreeMap** qui implémentent l'interface Map.
- **HashMap<K,V>**
 - Tables de hachage
 - Elles utilisent la méthode hashCode() des objets clés.
 - l'ensemble des clés est un HashSet.
 - Les performances de HashMap sont meilleures que TreeMap mais pas d'ordre sur les clés
- **TreeMap<K,V>**
 - permet de gérer des tables ordonnées sur les clés.
 - l'ensemble des clés est un TreeSet (arbre binaire ordonné assurant un accès en $\log_2(n)$).
 - les clés doivent donc être ordonnables
 - soit en implémentant Comparable (ordre naturel)
 - soit en fournissant un Comparator au constructeur :
`public TreeMap<K,V>(Comparator<K> c)`

TreeMap : exemple

- Bibliothèque
 - table code-Ouvrage ordonnée par les codes (String)
 - TreeMap<String,Ouvrage>

```
public class NonDisponibleException extends Exception {}  
public class Ouvrage {  
    protected String titre, auteur;  
    protected boolean emprunte;  
    protected int compteur; // nombre d'emprunts  
    public int getCompteur() {return compteur;}  
    public void emprunter() throws NonDisponibleException {  
        if (emprunte) throw new NonDisponibleException();  
        else { emprunte=true; compteur++;}  
    }  
}
```

TreeMap : exemple

```
public class Bibliotheque {  
    protected Map<String,Ouvrage> ouvrages  
        = new TreeMap<String,Ouvrage>();  
    public void add(String code, Ouvrage o) {  
        ouvrages.put(code,o);  
    }  
    public int totalEmprunts() { // parcours des valeurs  
        int total=0;  
        Iterator<Ouvrage> iter = ouvrages.values().iterator();  
        while (iter.hasNext())  
            total=total+iter.next().getCompteur();  
        return total;  
    } //suite ...  
}
```

TreeMap : exemple

```
//suite ...
public void listing() {/"for each » sur les cles
    for(String code : ouvrages.keySet())
        System.out.println(code+": "+ouvrages.get(code));

    public void emprunter(String code)
        throws OuvrageInconnuException, NonDisponibleException {
        try {
            ouvrages.get(code).emprunter();
        } catch (NullPointerException ex) {
            throw new OuvrageInconnuException();
        } // NonDisponibleException : propagée (cf. throws)
    }
    public class OuvrageInconnuException extends Exception {}
```

TreeMap : exemple

```
// application (main)

Bibliotheque bib = new Bibliotheque();
bib.add("I101",new Ouvrage("C","Kernighan"));
bib.add("L202",new Ouvrage("Germinal","Zola"));
bib.add("S303",new Ouvrage("Parapente","Ali Gali"));
bib.add("I345",new Ouvrage("Java","Eckel"));
bib.listing();

/* resultat : ordre lexicographique des codes (String)
I101:C Kernighan
I345:Java Eckel
L202:Germinal Zola
S303:Parapente Ali Gali
*/
```

TreeMap : exemple

```
// application (suite)

String code; // ...

try {
    bib.emprunter(code);
} catch (OuvrageInconnuException ex) {
    System.out.println("ouvrage "+code+" inexistant");
} catch (NonDisponibleException ex) {
    System.out.println("ouvrage"+code+"non dispo");
}
```