



# Java Collections Framework

*Bachir Djafri*

*IBISC / Université d'Évry Val d'Essonne*

*bachir.djafri@ibisc.univ-evry.fr*

*<http://www.ibisc.univ-evry.fr/~djafri>*

## Plan

- ◆ Introduction
- ◆ Les interfaces de collection
- ◆ Les implémentations (collections concrètes)
- ◆ Les structures des collections
- ◆ Les algorithmes
- ◆ Les implémentations personnalisées
- ◆ *Références*

*Java Collections Framework*

*B. Djafri (2)*

## Les structures de données

- ◆ Encapsulation des données dans des classes
- ◆ Choix et organisation des SD selon le Pb posé
  - Recherche, tri, insertion, suppression, accès, ...
  - Structure en tableau, liste, arbre, autre ...
- ◆ Choix de la SD engendre des différences
  - Implémentation des méthodes
  - Performances
- ◆ Java facilite le choix de SD adaptées : les collections + structures fournies avec Java

*Java Collections Framework*

*B. Djafri (3)*

## Les interfaces de collection

- ◆ Collection (conteneur) = ensemble générique d'objets
  - Main de poker (collection de cartes), répertoire de mails, répertoire téléphonique, ...
- ◆ Les objets peuvent être soumis à des contraintes
  - Ordre (liste), entrées uniques (ensemble), ...
- ◆ Les tableaux, Properties, Hashtable et Vector : collections

*Java Collections Framework*

*B. Djafri (4)*

## Interface et implémentation

- ◆ Séparation de l'interface d'une collection de son implémentation (Abstraction)
  - Exemple : Queue de données (File d'attente)
- ◆ Interface = liste d'opérations (méthodes)
  - Ajouter, Supprimer, nombre d'éléments, ...
- ◆ Politique d'organisation des données
  - FIFO, LIFO, Triées, ...
- ◆ Utilisation des *interfaces* Java

## Interface et implémentation

```
public interface Queue {  
    void add(Object o);  
    Object remove();  
    int size();  
    ...  
}
```

- ◆ Pas de détails d'implémentation des méthodes
- ◆ Implémentations possibles : tableau circulaire, liste chaînée, ...

## Interface et implémentation

```
public class CircularArrayQueue  
    implements Queue {  
    public CircularArrayQueue(int  
        capacity){  
        // ...  
    }  
  
    public void add(Object o){ }  
    public Object remove(){ ...}  
    public int size(){ ... }  
  
    private Object[] elements;  
    private int head;  
    private int tail;  
}
```

```
public class LinkedListQueue  
    implements Queue {  
    public LinkedListQueue(){  
        // ...  
    }  
  
    public void add(Object o){ }  
    public Object remove(){ ...}  
    public int size(){ ... }  
  
    private Link head;  
    private Link tail;  
}
```

## Utilisation de la classe

- ◆ Pas besoin de connaître l'implémentation réelle
  - Le type interface permet le choix d'utilisation entre les deux implémentations

```
Queue clients = new CircularArrayQueue(101);  
Clients.add(new Client("Denis"));
```

Ou

```
Queue clients = new LinkedListQueue();  
Clients.add(new Client("Denis"));
```

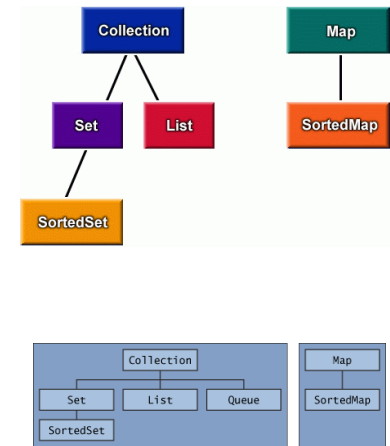
# implémentation

- ◆ Choix : tableau circulaire (collection bornée)
- ◆ L'interface de la méthode `add` doit alors pouvoir indiquer un échec de la méthode

```
void add(Object o) throws CollectionFullException;
```
- ◆ Problème
  - Impossible d'ajouter une gestion d'exception (add est une méthode surchargée)
  - Définir deux interfaces ou déclencher des exceptions dans tous les cas ?
- ◆ Solution : les interfaces de collection (Framework) Java

# Les interfaces de collection

- ◆ Comment manipuler des collections de données de manière indépendante (abstraite) de l'implémentation ?
- ◆ Collections Java
  - 2 méthodes essentielles
    - `boolean add(Object o);`
    - `Iterator iterator();`
- ◆ L'itérateur permet de parcourir les éléments d'un conteneur (collection)
- ◆ Interfaces génériques



## Ancienne interface Collection

```
public interface Collection {
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element); // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    boolean containsAll(Collection c);
    boolean addAll(Collection c); // Optional
    boolean removeAll(Collection c); // Optional
    boolean retainAll(Collection c); // Optional
    void clear(); // Optional

    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

## L'interface Collection

```
public interface Collection<E> extends Iterable<E>{
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element); //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c); //optional
    boolean retainAll(Collection<?> c); //optional
    void clear(); //optional

    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

## L'interface Iterable

```
public interface Iterable<T> {  
    Iterator<T> iterator();  
    // Returns an iterator over  
    // a set of elements of type T.  
}
```

♦ Objets Iterable : for-each

## L'interface Iterator

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove(); // Optional  
}
```

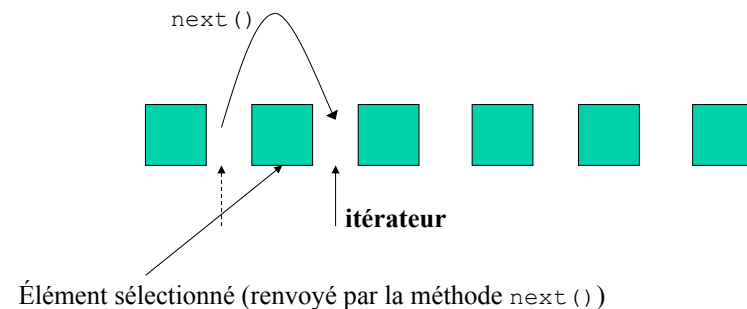
♦ Équivalente à l'interface Enumeration

## Exemple d'itérateur

```
Iterator<E> it = maCollection.iterator();  
while(it.hasNext()){  
    E o = it.next();  
    // utilisation de o ...  
}
```

```
for(Iterator<E> it=c.iterator(); it.hasNext();)  
{ if(!cond(it.next())) it.remove();  
}
```

## Progression d'un itérateur



♦ Attention !

- `it.remove(); it.remove();` // erreur
- `It.remove(); it.next(); it.remove();`

## Méthodes pratiques (1)

- ◆ Méthodes pouvant travailler sur tout type de collection

```
public static void printCollection(Collection<?> c){
    System.out.println("[");
    Iterator<?> it = c.iterator();
    while(it.hasNext()){
        System.out.print(it.next()+"");
    }
    System.out.println("]");
}
```

## Méthodes pratiques (2)

```
public static boolean addAll(
    Collection<?> to, Collection<?> from){
    Iterator<?> it = from.iterator();
    boolean modified = false;
    while(it.hasNext()){
        if(to.add(it.next())) modified=true;
    }
    return modified;
}
```

## La classe AbstractCollection

- ◆ Définit les méthodes add() et iterator() comme méthodes abstraites
- ◆ Donne une définition aux autres méthodes générales

```
public class AbstractCollection<E> implements Collection<E>{

    public boolean addAll(Collection<? extends E> from){
        Iterator<?> it = from.iterator();
        boolean modified = false;
        while(it.hasNext()){
            if(add(it.next())) modified = true;
        }
        return modified;
    }
    ...
}
```

## Les Listes

- ◆ Collection ordonnée (FIFO, LIFO, ...)
- ◆ La position des éléments de la liste est importante et « connue »
- ◆ Liste = séquence d'éléments
- ◆ Peut contenir des doublons (!= ensemble)
- ◆ Insertion/suppression au milieu d'une liste (à une position donnée)
- ◆ Accès via un indice (position de l'élément)

# L'interface List

```
public interface List<E> extends Collection<E> {

    E get(int index);
    E set(int index, E element); // Optional
    void add(int index, E element); // Optional
    E remove(int index); // Optional
    boolean addAll(int index, Collection<? extends E> c); // Optional

    int indexOf(Object o);
    int lastIndexOf(Object o);

    ListIterator<E> listIterator(); //renvoie un objet itérateur qui
    ListIterator<E> listIterator(int index); //implémente ListIterator

    List<E> subList(int from, int to);
}
```

# L'interface ListIterator

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove(); // Optional
    void set(E o); // Optional
    void add(E o); // Optional
}
```

## Exemple de listIterator

```
ListIterator<T> it =
    maListe.listIterator();
it.hasNext();
// vérifier que la liste n'est pas vide

T element = it.next();
//renvoie le 1er élément

It.set(nouvelElement);
// affecte une nouvelle valeur(objet)
// au 1er élément de la liste.
```

## Exemple de listIterator

- ♦ On peut avoir plusieurs itérateurs d'une même liste de données
  - problèmes de modification de la structure de la collection

```
ListIterator<T> it1 = maListe.listIterator();
ListIterator<T> it2 = maListe.listIterator();
// vérifier que la liste n'est pas vide

it1.next(); // renvoie le 1er élément
It1.remove();

It2.next(); // déclenche une exception
```

## Les classes concrètes (1)

### ◆ Les listes chaînées

- Pourquoi les listes chaînées ?
  - Collection ordonnée : la position des éléments est importante
- Les listes chaînées de Java : `LinkedList<E>`
  - Implémente l'interface `List`
  - Listes doublement chaînées
  - Utilise un itérateur de liste : objet d'une classe qui implémente la sous interface `ListIterator`
  - `ListIterator` : contient une méthode `add()`

## Exemple de LinkedList

```
LinkedList<String> maListe =  
    new LinkedList<String>();  
maListe.add(" Pascal ");  
maListe.add(" Sandrine ");  
maListe.add(" Denis ");  
  
Iterator<String> it = maListe.iterator();  
  
while(it.hasNext())  
    System.out.println(it.next());  
it.remove(); // supprime le dernier élément
```

## Les classes concrètes (2)

### ◆ Les listes tableau

- La liste tableau de Java : `ArrayList<E>`
  - Implémente l'interface `List`
  - Comparable à la classe **Vector** (non synchronisée)
  - Encapsule un tableau dynamique
  - Utilise les méthodes `set()` et `get()` au lieu de `setElementAt()` et `elementAt()`

## Les ensembles

- ◆ Collection sans doublons
- ◆ Les éléments ne sont pas ordonnés
- ◆ L'interface Java `Set` hérite de toutes les méthodes de l'interface `Collection` et n'ajoute aucune méthode.
- ◆ Restriction sur la duplication des éléments de la collection uniquement
- ◆ Plusieurs implémentations possibles : table de hachage, arbre, ...

## Les tables de hachage

- ◆ Pourquoi les tables de hachage ?
- ◆ Le code de hachage
- ◆ Structure d'une table de hachage
- ◆ Exemple
  - Objet Obj dont le code de hachage est 345
  - 101 paniers (ou seaux)
  - Obj sera placé dans le panier 42 ( $345 \% 101 = 42$ )
- ◆ Problème de collision de hachage

## La classe HashSet

- ◆ Implémente un ensemble (set) à partir d'une table de hachage
- ◆ Méthode `contains()` est redéfinie pour une recherche adaptée et plus rapide
- ◆ L'itérateur d'un set parcourt tous les éléments de tous les paniers un par un
- ◆ Code de hachage : entier renvoyé par la méthode `hashCode()` de la classe `Object`
- ◆ Exemple : la classe `String`
- ◆ La méthode `equals()` de la classe `Object` doit être compatible avec la méthode `hashCode()`

## Exemple de HashSet

```
class Article{
    private String nomArticle;
    private String description;
    private int codeArticle;
    ...
    public int hashCode() {
        return codeArticle;
    }
    // ...
}
```

## Les ensembles triés

- ◆ Collection sans doublons
- ◆ Les éléments sont triés selon un certain ordre (comparaison)
- ◆ L'interface `Java SortedSet` dérivée de l'interface `Set` ajoute des méthodes de « range-view »
- ◆ Restriction sur la duplication des éléments de la collection
- ◆ Implémentations possibles : arbre binaire, ...



## L'interface SortedSet

- ◆ Package java.util
- ◆ Dérive de l'interface Set (qui dérive elle-même de l'interface Collection)
- ◆ Implémentée par la classe TreeSet
- ◆ Éléments ordonnés selon l'ordre défini par la méthode compareTo ou par un comparateur
- ◆ Les éléments doivent implémenter l'interface Comparable.

## L'interface SortedSet

```
public interface SortedSet<E> extends Set<E>{

    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? Super E> comparator();
}
```

## Les Arbres

- ◆ Classe TreeSet
  - implémente les arbres binaire
  - collection triée (ensemble trié)
  - insertion dans n'importe quel ordre
  - Présentation des éléments selon un ordre croissant
- ◆ Le tri est assuré par une structure de données en arbre (arbre binaire)
- ◆ Les objets TreeSet trient automatiquement leurs éléments

## Exemple d'Arbre

```
TreeSet<String> arbre = new
    TreeSet<String>();

arbre.add(" Gilles ");
arbre.add(" Sandrine ");
arbre.add(" Denis ");

Iterator<String> it = arbre.iterator();
while(it.hasNext())
    System.out.println(it.next());
```

## Comparaison d'objets

- ◆ Dans quel ordre trier les éléments ?
- ◆ Insertion d'éléments (objets) implémentant l'interface Comparable
- ◆ Interface Comparable
  - `public interface Comparable<T>`
  - Une seule méthode : `int compareTo(T o);`
  - Résultat : 0 si =, <0 si inf. et >0 si sup.
  - Exemple la classe String (ordre alphabétique)

## Exemple d'objets Comparables

```
class Article implements Comparable<Article>{  
    ...  
    public int compareTo(Article a){  
        // Article a = (Article) o;  
        return codeArticle - a.codeArticle;  
    }  
    ...  
}
```

## Interface Comparable

- ◆ Package `java.lang`
- ◆ Classes implémentant l'interface :
  - `BigDecimal`, `BigInteger`, `Byte`, `ByteBuffer`, `Character`, `CharBuffer`, `Charset`, `CollationKey`, `Date`, `Double`, `DoubleBuffer`, `File`, `Float`, `FloatBuffer`, `IntBuffer`, `Integer`, `Long`, `LongBuffer`, `ObjectStreamField`, `Short`, `ShortBuffer`, `String`, `URI`, ...

## Les objets Comparator

- ◆ Objets de classes implémentant l'interface Comparator
- ◆ L'interface Comparator
  - `public interface Comparator<T>`
  - Une méthode : `int compare(T a, T b)`
  - Résultat : 0 si =, <0 si inf. et >0 si sup.
  - Une autre méthode : `boolean equals(Object o)`
- ◆ Utilisation : passés en paramètre au constructeur de l'arbre `TreeSet` par exemple.

## Exemple de comparateur

```
class ArticleComparator implements
    Comparator<Article>{
    ...
    public int compare(Article A, Article B){
        // Article A = (Article)a;
        // Article B = (Article)b;
        String da = A.getDescription();
        String db = B.getDescription();
        return da.compareTo(db);
    }
    ...
}
```

## Utilisation de comparateurs

```
ArticleComparator c = new
    ArticleComparateur<Article>();
TreeSet<Article> trieParDesc = new
    TreeSet<Article>(c);
```

- ◆ c est appelé « objet de fonction »
  - Objet sans état
  - Emplacement pour la méthode compare

## Utilisation de comparateurs (2)

```
TreeSet<Article> trieParDesc = new
    TreeSet<Article>(
        new Comparator<Article>() {
            public int compare(Article a, Article b){
                // Article A = (Article)a;
                // Article B = (Article)b;
                String da = a.getDescription();
                String db = b.getDescription();
                return da.compareTo(db);
            }
        }
    );
```

## La classe TreeSet

- ◆ Package java.util
- ◆ Dérive de la classe AbstractSet<E>
- ◆ implémente les interfaces  
SortedSet, Cloneable, Serializable, Iterable<E>
- ◆ Assure le tri de ses éléments (ordre croissant)
- ◆ Les éléments doivent implémenter l'interface Comparable.

# Les cartes (Map)

- ◆ Ensemble/collection de paires (clé, valeur)
- ◆ Permettent une recherche rapide (selon certaines informations)
- ◆ Recherche des éléments à partir de leurs clés
- ◆ Les clés ne peuvent être dupliquées
- ◆ Une valeur au plus par clé
- ◆ Exemple (Banque)
  - (Client Cl, Compte cpt)
- ◆ Deux implémentations concrètes en Java
  - HashMap (Carte sans tri) & TreeMap (carte triée)

# L'interface Map

```
public interface Map<K, V> {
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    void putAll(Map<? extends K, ? extends V> m);
    void clear();

    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

# L'interface SortedMap

```
public interface SortedMap<K, V> extends Map<K, V>{
    Comparator<? super K> comparator();
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

# Implémentations concrètes

- ◆ HashMap : répartition des clés en plusieurs catégories
- ◆ TreeMap : ordre total pour organiser les clés dans un arbre de recherche
- ◆ Fonctions de hachage appliquées aux clés seulement
- ◆ Choix entre les 2 implémentations ?
  - Cartes de hachage légèrement plus rapides

# La classe HashMap

- ◆ Package java.util
- ◆ Dérive de AbstractMap
- ◆ implémente Map
- ◆ Implémentation basée sur une table de hachage
- ◆ Équivalente à la classe Hashtable (synchronisée)
- ◆ Permet des clés et des valeurs `null` (contrairement aux Hashtables)
- ◆ Éléments non ordonnés

# Exemple de HashMap

```
HashMap<String, Client> clients = new  
    HashMap<String, Client>();  
Client unClient = new Client("Denis" , "Dupont");  
clients.put("123-4567" , unClient);
```

## ◆ Recherche

```
String cle = "123-4567" ;  
Client c = clients.get(cle);
```

# Les vues des cartes (1)

- ◆ Vue = objet d'une classe implémentant l'interface Collection ou une de ses sous interfaces
- ◆ Trois vues possibles pour une carte
  - L'ensemble des clés
  - L'ensemble des valeurs
  - L'ensemble des paires (clé/valeur)

# Les vues des cartes (2)

- ◆ Méthodes permettant d'obtenir les 3 vues
  - **public** Set<K> keySet();
  - **public** Collection<V> values();
  - **public** Set<Map.Entry<K,V>> entrySet();
- ◆ Les élément de entrySet : objet d'une classe qui implémente l'interface interne Map.Entry
- ◆ keySet : objet d'une classe qui implémente l'interface Set (pas de doublons)

## Utilisation des vues (1)

### ◆ Énumération des clés d'une carte

```
Set<String> cles = carte.keySet();
Iterator<String> it = cles.iterator();
while(it.hasNext()){
    String k = it.next();
    // utilisation de l'objet k ...
}
```

## Utilisation des vues (2)

### ◆ Énumération des clés/valeurs d'une carte

```
Set<Map.Entry<K,V>> entrees = clients.entrySet();
Iterator<Map.Entry<K,V>> it = entrees.iterator();
while(it.hasNext()){
    Map.Entry<Map.Entry<K,V>> entree = it.next();
    K cle = entree.getKey();
    V valeur = entree.getValue();
    // utilisation des objets cle et valeur ...
}
```

## Classes et interfaces utiles

- ◆ Interface java.util.Map
- ◆ Interface java.util.SortedMap
- ◆ Classe interne java.util.Map.Entry
- ◆ Classe java.util.HashMap (éléments non triés)
- ◆ Classe java.util.TreeMap (éléments triés)

## Les vues et les emballages

- ◆ Pourquoi les vues ?
  - Problème de synchronisation
  - Mécanisme qui génère des vues synchronisées pour toutes les interfaces
- ◆ Obtenir des objets de classes qui implémentent Collection et Map en utilisant les **vues**
- ◆ Exemple : la méthode `keySet` des classes de cartes
- ◆ Les vues manipulent directement les cartes d'origine

## Exemple de vue synchronisée

```
HashMap hashMap = new HashMap();  
// création d'un objet carte non synchronisé  
  
map = Collections.synchronizedMap(hashMap);  
// création d'un vue synchronisée de l'objet  
// carte.  
  
map = Collections.synchronizedMap(new HashMap());  
// éviter l'utilisation de la HashMap non synchronisée
```

## La classe Collections (1)

- ◆ 19 méthodes statiques qui manipulent ou renvoient des Collections (vues ou emballages).
- ◆ 6 méthodes pour obtenir des Collections synchronisées (une par interface) : vues ou emballages
  - Collection synchronizedCollection(Collection c);
  - Set synchronizedSet(Set s);
  - List synchronizedList(List list);
  - Map synchronizedMap(Map m);
  - SortedSet synchronizedSortedSet(SortedSet s);
  - SortedMap synchronizedSortedMap(SortedMap m);

## Les vues non modifiables

- ◆ 6 méthodes pour obtenir des Collections non modifiables (une par interface) :

- Collection unmodifiableCollection(Collection c);
- Set unmodifiableSet(Set s);
- List unmodifiableList(List list);
- Map unmodifiableMap(Map m);
- SortedSet unmodifiableSortedSet(SortedSet s);
- SortedMap unmodifiableSortedMap(SortedMap m);

## Exemple

```
List clients = new LinkedList();  
...  
uneMethodeDeLecture(Collections.unmodifiableList(clients));  
  
// attention !  
List listeNonModifiable =  
    Collections.unmodifiableList(new LinkedList());  
  
// la liste reste vide
```

# Les sous ensembles

## ◆ Vues de sous ensembles

- `List` groupe = `clients.subList(7, 13);`
- 1er indice inclusif, 2nd exclusif


# Hiérarchie de interfaces

- ◆ `interface java.util.Collection`
  - `interface java.util.List`
  - `interface java.util.Set`
    - `interface java.util.SortedSet`
- ◆ `interface java.util.Comparator`
- ◆ `interface java.util.Enumeration`
- ◆ `interface java.util.Iterator`
  - `interface java.util.ListIterator`
- ◆ `interface java.util.Map`
  - `interface java.util.SortedMap`
- ◆ `interface java.util.Map.Entry`

# Hiérarchie des classes

- ◆ `class java.util.AbstractCollection` (implements `java.util.Collection`)
  - `class java.util.AbstractList` (implements `java.util.List`)
    - `class java.util.AbstractSequentialList`
      - `class java.util.LinkedList` (implements `java.lang.Cloneable`, `java.util.List`, `java.io.Serializable`)
    - `class java.util.ArrayList` (implements `java.lang.Cloneable`, `java.util.List`, `java.util.RandomAccess`, `java.io.Serializable`)
    - `class java.util.Vector` (implements `java.lang.Cloneable`, `java.util.List`, `java.util.RandomAccess`, `java.io.Serializable`)
      - `class java.util.Stack`
  - `class java.util.AbstractSet` (implements `java.util.Set`)
    - `class java.util.HashSet` (implements `java.lang.Cloneable`, `java.io.Serializable`, `java.util.Set`)
      - `class java.util.LinkedHashSet` (implements `java.lang.Cloneable`, `java.io.Serializable`, `java.util.Set`)
    - `class java.util.TreeSet` (implements `java.lang.Cloneable`, `java.io.Serializable`, `java.util.SortedSet`)
- ◆ `class java.util.AbstractMap` (implements `java.util.Map`)
  - `class java.util.HashMap` (implements `java.lang.Cloneable`, `java.util.Map`, `java.io.Serializable`)
    - `class java.util.LinkedHashMap`
  - `class java.util.IdentityHashMap` (implements `java.lang.Cloneable`, `java.util.Map`, `java.io.Serializable`)
  - `class java.util.TreeMap` (implements `java.lang.Cloneable`, `java.io.Serializable`, `java.util.SortedMap`)
  - `class java.util.WeakHashMap` (implements `java.util.Map`)
  - `class java.util.GregorianCalendar`
- ◆ `class java.util.Collections`
- ◆ `class java.util.Dictionary`
  - `class java.util.Hashtable` (implements `java.lang.Cloneable`, `java.util.Map`, `java.io.Serializable`)
    - `class java.util.Properties`
- ◆ `class java.util.StringTokenizer` (implements `java.util.Enumeration`)

# Résumé des classes concrètes

		Implementations			
		Hash Table	Resizable Array	Balanced Tree	Linked List
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	