



Les Collections

Java Collection Framework

Bachir Djafri

IBISC / Université d'Évry Val d'Essonne

bachir.djafri@univ-evry.fr

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

Plan

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

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
- ◆ La POO facilite le choix de SD adaptées : les collections + structures fournies avec les langages

Les interfaces de collection

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

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* (interface Java)

Interface et implémentation

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

- ◆ Pas de détails d'implémentation des méthodes
- ◆ Plusieurs 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

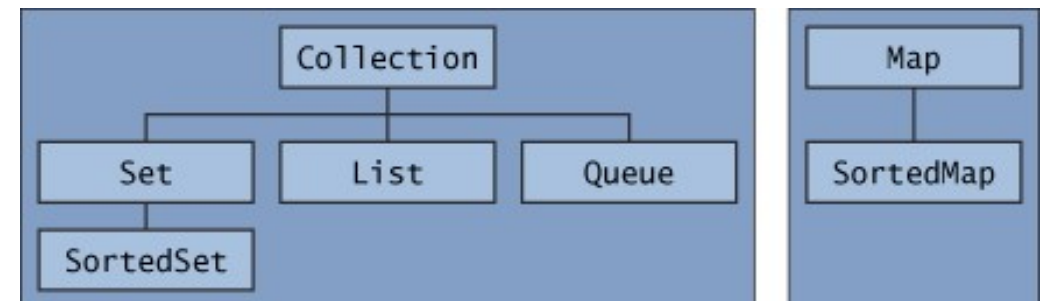
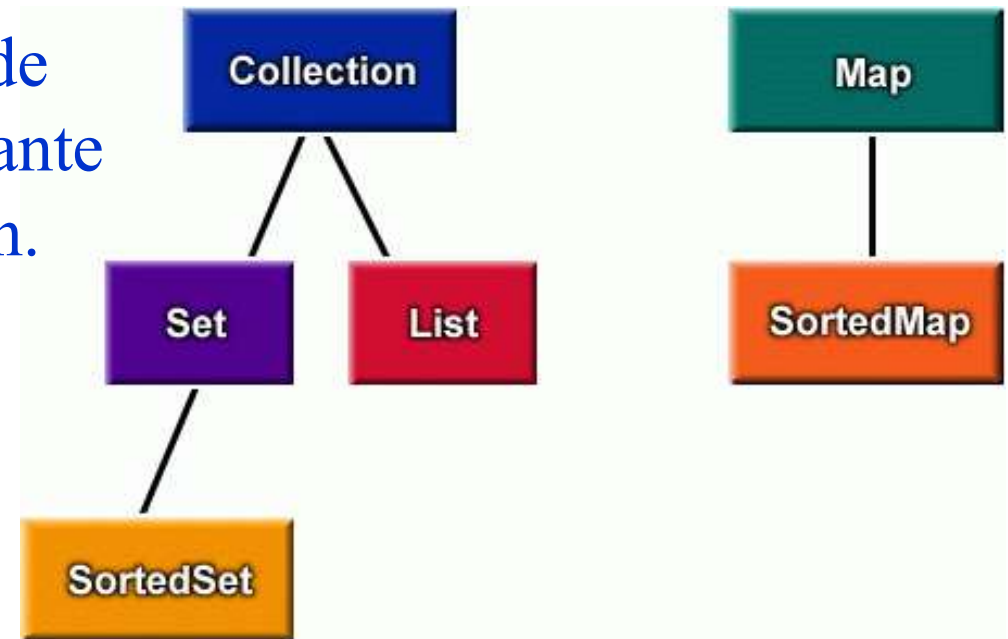
- ◆ Choix : tableau circulaire (collection bornée)
- ◆ L'interface de la méthode `add` doit alors pouvoir indiquer un échec de la méthode

```
void add(Object o) throws CollectionFullException;
```

- ◆ Problème
 - Impossible d'ajouter une gestion d'exception
(`add` est une méthode redéfinie/implémentée)
 - Définir deux interfaces ou déclencher des exceptions dans tous les cas ? Quelle interface ?
- ◆ Solution : les interfaces de collections (Framework Java)

Les interfaces de collections

- ♦ Manipulation des collections de données de manière indépendante (abstraite) de l'implémentation.
- ♦ Collections Java
 - 2 méthodes essentielles
 - `boolean add(Object o);`
 - `Iterator iterator();`
- ♦ L'**itérateur** : parcourir les éléments d'une collection (conteneur)
- ♦ Interfaces **génériques**



L'interface Collection

```
public interface Collection<E> extends Iterable<E>{  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element);  
    boolean remove(Object element);  
    Iterator<E> iterator();  
  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c);  
    boolean removeAll(Collection<?> c);  
    boolean retainAll(Collection<?> c);  
    void clear();  
  
    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  
}
```

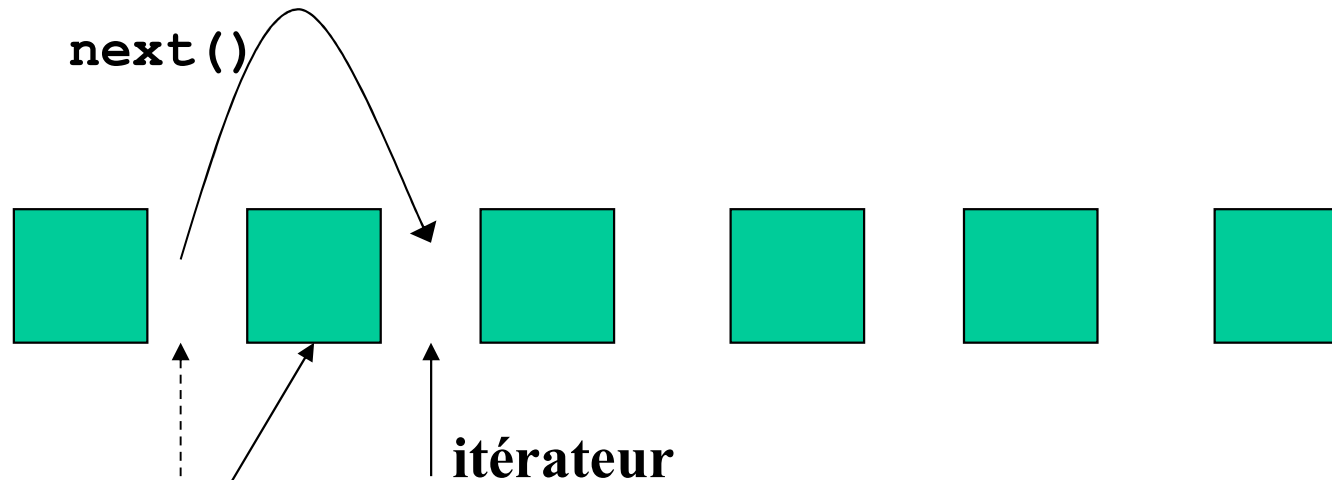
♦ Équivalente à l'interface *Enumeration*

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



Élément sélectionné/vu (renvoyé par la méthode **next()**)

◆ 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**

```
abstract 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 (\neq 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);  
    void add(int index, E element);  
    E remove(int index);  
    boolean addAll(int index, Collection<? extends E> c);  
  
    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();  
    void set(E o);  
    void add(E o);  
}
```

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

```
List<String> maListe =  
    new LinkedList<String>(); //ArrayList  
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 de 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 `hashCode()` de la classe `Object` doit être compatible avec la méthode `equals()`

Exemple d'Eléments de HashSet

```
public class Article{  
    private String nomArticle;  
    private String description;  
    private int codeArticle;  
    ...  
    public boolean equals(Object o){  
        return this.codeArticle==o.codeArticle;  
    }  
    public int hashCode(){  
        return codeArticle;  
    }  
}
```

Résumé des classes concrètes

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