

# Cours 6 : Les collections

**Généralités**

**Tableaux**

**Collections**

**Maps**

**Iterator**

**Vues et ponts entre structures**

# Cours 6 : Les collections

## Généralités

## Tableaux

## Collections

## Maps

## Iterator

## Vues et ponts entre structures

# Structures de données

- ▶ En Java, 3 sortes de structures de données :

# Structures de données

- ▶ En Java, 3 sortes de structures de données :
  - ▶ Les **tableaux**
    - ▶ Taille fixe, accès direct aux éléments.

# Structures de données

- ▶ En Java, 3 sortes de structures de données :
  - ▶ Les **tableaux**
    - ▶ Taille fixe, accès direct aux éléments.
  - ▶ Les **collections**
    - ▶ Structure modifiable, différent algorithmes de stockage.

# Structures de données

- ▶ En Java, 3 sortes de structures de données :
  - ▶ Les **tableaux**
    - ▶ Taille fixe, accès direct aux éléments.
  - ▶ Les **collections**
    - ▶ Structure modifiable, différent algorithmes de stockage.
  - ▶ Les **Maps**
    - ▶ Structure modifiable, stockage de couples CLEF → VALEUR.

# Contrats

- ▶ Les algorithmes existant sur les collections supposent que les objets stockés respectent un certain **contrat**.
- ▶ Il faut implémenter correctement `equals`, `hashCode`, `compareTo`....
- ▶ Sinon, n'importe quoi et/ou exceptions !

# Cours 6 : Les collections

Généralités

**Tableaux**

Collections

Maps

Iterator

Vues et ponts entre structures



# Tableaux

- ▶ Taille fixe définie à l'initialisation.
- ▶ Toujours mutable.

# Arrays

- ▶ Classe `Arrays` (avec un `s`).
- ▶ Méthodes statiques utilitaires sur les tableaux.
  - ▶ Remplissage (`fill`).
  - ▶ Tri, égalité, recherche binaire.
  - ▶ Conversion en liste, `toString`...

## copies

- `System.arraycopy` (memcpyy comme en C)
- `Arrays.copyOf()` (créé et recopie un tableau + grand ou + petit)
- `Object.clone()` duplique mais on utilise plutôt `Arrays.copyOf(array, array.length)`

`toString`, `deepToString`, `equals`, `hashCode`, etc

# Cours 6 : Les collections

Généralités

Tableaux

**Collections**

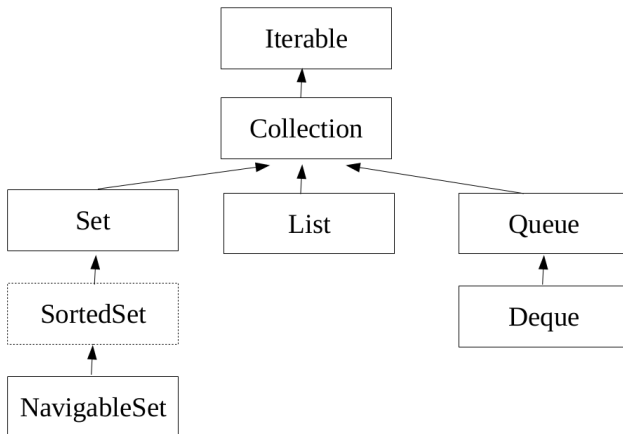
Maps

Iterator

Vues et ponts entre structures

# Collections

- ▶ Hiérarchie d'interfaces.
- ▶ Permet de prendre en paramètre ou en type de retour une collection d'objets **sans préciser comment ils sont stockés en mémoire**.



# Collections - Définitions abstraites

- ▶ **Collection** : ensemble de données.
- ▶ **Set** : ensemble de données **sans doublon**.
- ▶ **SortedSet** : ensemble de données sans doublon et **trié**.
- ▶ **NavigableSet** : ensemble de données sans doublon, trié et **avec précédent/suivant**.
- ▶ **List** : liste indexée ou séquentielle.
- ▶ **Queue** : file (FIFO).
- ▶ **Deque** : queue avec IO sur chaque côté.

# Collections paramétrées

- ▶ Les collections sont **homogènes** : contiennent des éléments qui ont le **même type** (mais pas forcément la même classe).
- ▶ Les collections sont donc **paramétrées** par une variable de type, souvent nommée E (pour Element).
- ▶ Si on veut stocker des éléments de type différent, on utilise le super-type commun (au pire, Object donc).

# Interface Collection

- ▶ Interface parente des collections (pas des Maps !).
  - ▶ Méthodes d'ajout d'un ou plusieurs objets.
  - ▶ Test d'appartenance.
  - ▶ Suppression.
  - ▶ Conversion en tableau.
  - ▶ Taille.
  - ▶ Demande d'itérateur.
- ▶ **Complexité variable** selon le type de collection !
- ▶ Certaines opérations sont optionnelles (UnsupportedOperationException levée).



# Mutabilité

- ▶ Les collections sont mutables par défaut.
  - ▶ `add(E)`, `remove(Object)`, `removeIf`, `clear`.
- ▶ Les opérations de mutation peuvent lever des `UnsupportedOperationException` pour représenter des **collections non-mutables**.
- ▶ **Vue** via `Collections.unmodifiableCollection()`
  - ▶ Par ex., plutôt que renvoyer une copie défensive, faire une vue non modifiable

```
1 List<String> list = new ArrayList<>() ;
2 List<String> list2 = Collections.unmodifiableList(list) ;
3 list2.add("hello") ; // UnsupportedOperationException
4 list.add("hello") ; // ok
5 list.add("hello2") ;
6 list2.size() ; // 2 : c'est une vue !
```

# Recherche

- ▶ On recherche un élément dans une collection via `boolean contains(Object)`.
- ▶ Complexité ?

# Recherche

- ▶ On recherche un élément dans une collection via `boolean contains(Object)`.
- ▶ Complexité ?
- ▶ Dépend de la structure de données !
  - ▶ HashSet en  $O(1)$ .
  - ▶ TreeSet en  $O(\ln n)$ .
  - ▶ ArrayList en  $O(n)$ .

# Object

- ▶ Pourquoi dans la JDK pour Set ou List on a par exemple :
    - ▶ `boolean contains(Object o)`
    - ▶ `boolean remove(Object o)`
- et pas des E ?

# Object

- ▶ Pourquoi dans la JDK pour Set ou List on a par exemple :
  - ▶ `boolean contains(Object o)`
  - ▶ `boolean remove(Object o)`et pas des E ?
- ▶ `contains` et `remove` utilisent `equals`.
- ▶ `equals` peut renvoyer true pour des types différents
  - ▶ Par ex., une `ArrayList` et une `LinkedList` peuvent avoir **le même contenu** (et `equals` renvoie true) et pourtant de type différent !

# List

- ▶ Liste d'éléments indexé conservant l'ordre d'insertion.
  - ▶ Type les noms des majors de promo chaque année.
- ▶ Des méthodes supplémentaires par rapport à Collection :
  - ▶ `E get(int index), E set(int index, E e)`
  - ▶ `int indexOf(E e), int lastIndexOf(E e)`
  - ▶ `void sort(Comparator<? super E>) (Java 8).`

# List

- ▶ Implémentations :
  - ▶ ArrayList : **tableau dynamique**. Ajout fin en  $O(1)$ , début en  $O(n)$ , accès en  $O(1)$ .
    - ▶ Tableau avec taille initiale (10) puis agrandi au besoin.
  - ▶ LinkedList : **liste doublement chaînée**. Ajout fin/début en  $O(1)$ , accès en  $O(n)$ .
    - ▶ Utilise plus de mémoire. Temps de parcours plus long.
- ▶ Interface List dangereuse niveau complexité!!

# List init

Depuis Java 9, possibilité de créer des listes **non mutables** plus facilement avec `List.of()`

```
1 List<String> l = List.of("salut", "c'est", "cool");
```



# List init

Depuis Java 9, possibilité de créer des listes **non mutables** plus facilement avec `List.of()`

```
1 List<String> l = List.of("salut", "c'est", "cool");
```

Identique à

```
1 List<String> l = Arrays.asList("salut", "c'est", "cool");  
2 l = Collections.unmodifiableList(l);
```

# Liste ou tableau ?

Si on connaît le nombre d'éléments, on utilise plutôt un tableau car c'est plus efficace

Mais un tableau de type paramétré est unsafe

Il est habituelle dans du code utilisateur d'utiliser `List<Set<String>>` au lieu de `Set<String>[]`

Il est aussi possible de voir un tableau comme une liste (cf plus tard)

# Set

- ▶ Représente un ensemble d'éléments **sans doublon**.
- ▶ Différentes implémentations :
  - ▶ HashSet : **Table de hachage**, ensemble sans ordre, add/remove en  $O(1)$ .
    - ▶ Attention au hashCode>equals !
    - ▶ Grossit en fonction du remplissage (défaut 75%).
  - ▶ TreeSet : **Arbre rouge/noir**, ordre selon un comparateur, add/remove en  $O(\ln n)$ .
  - ▶ LinkedHashSet : **Table de hachage+liste chaînée** sur les entrées, préserve l'ordre d'insertion, add/remove en  $O(1)$ . Évite le chaos sur l'ordre d'HashSet sans le coût supplémentaire de TreeSet.
    - ▶ Plus de mémoire utilisé que HashSet.

# Set

- ▶ Représente un ensemble d'éléments **sans doublon**.
- ▶ Différentes implémentations :
  - ▶ HashSet : **Table de hachage**, ensemble sans ordre, add/remove en  $O(1)$ .
    - ▶ Attention au hashCode>equals !
    - ▶ Grossit en fonction du remplissage (défaut 75%).
  - ▶ TreeSet : **Arbre rouge/noir**, ordre selon un comparateur, add/remove en  $O(\ln n)$ .
  - ▶ LinkedHashSet : **Table de hachage+liste chaînée** sur les entrées, préserve l'ordre d'insertion, add/remove en  $O(1)$ . Évite le chaos sur l'ordre d'HashSet sans le coût supplémentaire de TreeSet.
    - ▶ Plus de mémoire utilisé que HashSet.
- ▶ Objet algorithmique complexe.
- ▶ Choisir avec soin l'implémentation (et la comprendre...).

# Set - complexités

- Parcours classé par ordre de vitesse pour beaucoup d'éléments.

	size	clear	add	remove	contains	parcours
HashSet	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	3
LinkedHashSet	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	1
TreeSet	$O(1)$	$O(1)$	$O(\ln n)$	$O(\ln n)$	$O(\ln n)$	2

# Files / Queues

- ▶ Interface Queue (Java 5).
- ▶ FIFO : insertion en fin, suppression en début.

# Deque

- ▶ Depuis Java 6.
- ▶ Hérite de Queue.
- ▶ Interface pour piles/files.
- ▶ Insertion en début ou fin.
- ▶ Suppression en début ou fin.

# Queue et Deque - méthodes

- ▶ Des méthodes différentes selon 2 sémantiques :
  - ▶ Lever une exception si vide ou plein.
  - ▶ Valeur de retour null/faux si vide ou plein.



# Queue et Deque - méthodes

- ▶ Des méthodes différentes selon 2 sémantiques :
  - ▶ Lever une exception si vide ou plein.
  - ▶ Valeur de retour null/faux si vide ou plein.
- ▶ Demander un élément sans le retirer :
  - ▶ `peek()` ou `element()` et `getFirst()` ou `peekFirst()`.
- ▶ Ajouter en queue :
  - ▶ `add(e)` ou `offer(e)` et `addLast(e)` ou `offerLast()`.
- ▶ Retirer en tête :
  - ▶ `remove()` ou `poll()` et `removeFirst()` ou `pollFirst()`

# Queue/Deque - Implémentations

- ▶ `PriorityQueue` (Queue).
  - ▶ ABR codé dans un tableau.
  - ▶ Ordre entre éléments ( $O(\ln n)$  pour poll et offer).
- ▶ `ConcurrentLinkedQueue` (Queue).
  - ▶ Autorise les accès concurrents.
- ▶ `LinkedList` (Queue et Deque).
- ▶ `ArrayDeque` (Deque).

# Classe Collections

- ▶ Classe Collections (avec un s).
- ▶ **Méthodes statiques utiles** sur les collections.
  - ▶ Recherche binaire dans une collection triée.
  - ▶ Tri, mélange, min, max...
  - ▶ Listes à un seul élément non mutable.
  - ▶ Vues non mutables d'une liste.

# Cours 6 : Les collections

Généralités

Tableaux

Collections

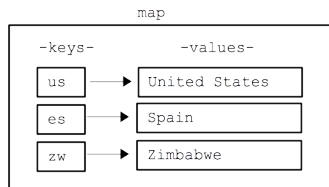
**Maps**

Iterator

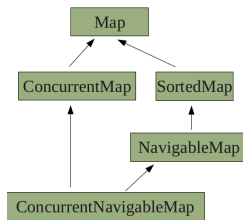
Vues et ponts entre structures

# Map

- ▶ Aussi appelé “table associative”, “dictionnaire”...
- ▶ Deux éléments :
  - ▶ Un élément clef.
  - ▶ Un élément valeur.
- ▶ **Pas de doublon** sur les clefs.
- ▶ **Doublon possible** sur les valeurs.



# Maps - Définitions abstraites



- Map : association sans relation d'ordre.
- SortedMap : association avec **clefs triées**.
- NavigableMap : association avec clefs triées et **suivant/précédent**.
- ConcurrentMap : association avec accès concurrent.

# Map - méthodes

- ▶ `V put(K,V)` : insère un couple clef/valeur, supprime le couple précédent avec la même clef, renvoie l'ancienne valeur ou null.
  - ▶ `V putIfAbsent(K,V)` : n'ajoute pas si le couple existe déjà (Java 8).
- ▶ `V get(Object key)` : renvoie la valeur correspondant à la clef ou null si pas de couple correspondant à la clef.
  - ▶ Prend un `Object` car uniquement selon le `equals`
  - ▶ `V getOrDefault(Object key, V defaultvalue)` : renvoie une valeur par défaut plutôt que null (Java 8).
- ▶ Faisable en groupé (`putAll`, `replaceAll`).

# Implémentations

- ▶ **HashMap** : **table de hachage** sur les clefs
  - ▶ Pas d'ordre sur les couples
  - ▶ Accès/ajout/suppression en  $O(1)$
- ▶ **LinkedHashMap** : **Table de hachage sur les clefs + liste doublement chaînée**
  - ▶ Couples ordonnés par ordre d'insertion
  - ▶ Accès/ajout/suppression en  $O(1)$
- ▶ **TreeMap** : **arbre rouge/noir**
  - ▶ Couples triés suivant un ordre de comparaison donné
  - ▶ Accès/insertion/suppression en  $O(\ln n)$



# Exemple

```
1 private static Map<String,String> map = new HashMap<>() ;
2 static {
3     map.put("France","Paris") ;
4     map.put("Allemagne","Berlin") ;
5 }
6 static String getCapital(String country) {
7     String resp = map.get(country) ;
8     if (resp==null)
9         throw new UnknownCountryException(country) ;
10    return resp ;
11 }
12 static void listKnownCapital() {
13     Set<Map.Entry<String,String>> entries = map.entrySet() ;
14     for(Map.Entry<String,String> entry :entries) {
15         System.out.println(entry.getKey()+ " has for capital "+ entry.getValue()) ;
16     }
17 }
```

# Cours 6 : Les collections

Généralités

Tableaux

Collections

Maps

**Iterator**

Vues et ponts entre structures

# Iterator

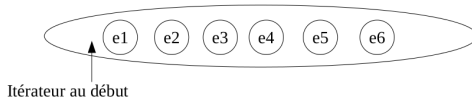
- ▶ Interface `Collection` a une méthode :
  - ▶ `Iterator<E> iterator()`
- ▶ Renvoie un objet qui implémente l'interface `Iterator`.
- ▶ Sorte de curseur pour **parcourir une collection élément par élément**.

# Iterator

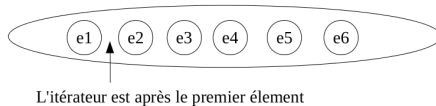
- ▶ Interface `Collection` a une méthode :
  - ▶ `Iterator<E> iterator()`
- ▶ Renvoie un objet qui implémente l'interface `Iterator`.
- ▶ Sorte de curseur pour **parcourir une collection élément par élément**.
- ▶ Permet de parcourir les éléments de la collection :
  - ▶ `boolean hasNext()` : reste des éléments à parcourir ?
  - ▶ `Object next()` : retourne l'élément suivant de la liste (et avance). (Exception si pas d'elt).
  - ▶ `void remove()` : supprime de la collection le dernier élément envoyé par `next` (donc pas possible de faire 2 `remove` de suite sans `next` entre).
- ▶ Parcourir une collection avec un itérateur et la modifier (ajout/supp) (sans utiliser `remove`) est interdit : lève une `ConcurrentModificationException`.

# Iterator - utilisation

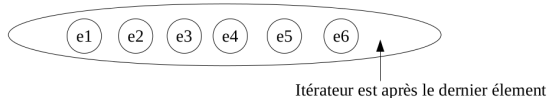
A la création :



Après un appel à `iterator.next()` :



Si `hasNext()` renvoie `false` :



# Iterator - intérêt

- ▶ Pas toujours possible d'effectuer un parcours d'une collection (Set, Queue...)
- ▶ Complexité pour certaines.
- ▶ L'itérator garantie un parcours en  $O(n)$ .

# Iterator - utilisation

- Plus efficace pour parcourir une collection sans accès direct !

```
1 private static int sum3(List<Integer> l) {  
2     int sum=0 ;  
3     Iterator<Integer> it = l.iterator() ;  
4     while(it.hasNext()) {  
5         sum += it.next() ;  
6     }  
7     return sum ;  
8 }
```

# Iterator - suppression

```
1    LinkedList<String> l = ...  
2    for(String s : l) {  
3        if(s.length() % 2 == 0){  
4            l.remove(s);  
5        }  
6    }
```

OK?



# Iterator - suppression

```
1  LinkedList<String> l = ...
2  for(String s : l) {
3      if(s.length() % 2 == 0){
4          l.remove(s);
5      }
6  }
```

OK?

► Marche pas ! Et mauvaise complexité !

```
1  LinkedList<String> l = ...
2  Iterator<String> it = l.iterator();
3  while(it.hasNext()) {
4      String s = it.next();
5      if(s.length()%2 == 0) {
6          it.remove();
7      }
8  }
```

Mieux

# Iterable

- ▶ Si une classe implémente l'interface `Iterable` : elle est capable de fournir un `Iterator`.
- ▶ `Collection` implémente `Iterable`.
- ▶ Les éléments `Iterable` (et les tableaux) peuvent être utilisés dans un `foreach`.

# Iterable

```
1 public class Firm {  
2     private final LinkedList<Employee> employees = new LinkedList<>();  
3  
4     public static void main(String[] args) {  
5         Firm f = new Firm();  
6         for(Employee e : f) { //impossible  
7             System.out.println(e);  
8         }  
9     }  
10 }
```

# Iterable

```
1 public class Firm {
2     private final LinkedList<Employee> employees = new LinkedList<>();
3
4     public static void main(String[] args) {
5         Firm f = new Firm();
6         for(Employee e : f) { //impossible
7             System.out.println(e);
8         }
9     }
10 }
```

```
1 public class Firm implements Iterable<Employee>{
2     private final LinkedList<Employee> employees = new LinkedList<>();
3
4     @Override
5     public Iterator<Employee> iterator() {
6         return employees.iterator();
7     }
8
9     public static void main(String[] args) {
10         Firm f = new Firm();
11         for(Employee e : f) { //ok !
12             System.out.println(e);
13         }
14     }
15 }
```

# ListIterator

- ▶ Version spécialisée d'Iterator (en hérite).
- ▶ Ajoute les méthodes add (avant le curseur) et set (du dernier next),
- ▶ et le parcours à l'envers hasNext() previous().

```
1  ListIterator<String> li = l.listIterator(l.size()); //demarre à la fin
2  while(li.hasPrevious()) {
3      String s = li.previous();
4  }
```

# Cours 6 : Les collections

Généralités

Tableaux

Collections

Maps

Iterator

**Vues et ponts entre structures**

# Ponts

- ▶ Deux types de méthodes de conversions :
  - ▶ **Copier** les données d'une structure vers une autre.
  - ▶ **Voir** une structure comme une autre.
    - ▶ Les données restent dans la structure initiale et sont **vues** dans la nouvelle structure.

# Ponts via copie

- ▶ De **collections** vers **tableaux** :
  - ▶ `Object[] toArray()` dans l'interface `Collection`.
- ▶ **Collections** vers **collections** :
  - ▶ Toutes les collections ont un constructeur qui prend une `Collection< ? extends E>`.
  - ▶ `addAll(Collection< ? extends E>)`.
  - ▶ **ATTENTION** copie des **références** !
- ▶ **Tableaux** vers **collections** :
  - ▶ `<T> Collections.addAll(Collection< ? super T>, T... array)`.
- ▶ **Tableaux** vers **tableaux** :
  - ▶ `Arrays.copyOf` (totalité ou début)
  - ▶ `Arrays.copyOfRange` (intervalle)
  - ▶ `System.arraycopy` (plus précis)



# Ponts via vue

- ▶ Vue d'une **liste** :
  - ▶ `AbstractList`
  - ▶ `AbstractSequentialList`
- ▶ **Tableau** vers **liste** :
  - ▶ `<T> List<T> Arrays.asList(T...array).`
- ▶ **Liste** vers **liste** :
  - ▶ `l.subList(int start, int end)`

# Ponts via vue - exemple

- ▶ Pas de méthode `contains` ou `shuffle` pour les tableaux.
- ▶ On utilise une vue !

```
1 List<String> list = Arrays.asList(args) ;  
2 list.contains("miage") ; //miage dans args ?  
3 Collections.shuffle(list) ; //args shuffled !
```

# Ponts via vue - maps

- ▶ Map vers l'ensemble des clefs :
  - ▶ `Set<K> map.keySet()`.
- ▶ Map vers collection de valeurs :
  - ▶ `Collection<V> map.values()`.
- ▶ Map vers couples clef/valeur :
  - ▶ `Set<Map.Entry<K,V>> map.entrySet()`.

# Map.Entry

- ▶ Interface interne de Map.
- ▶ Représente des couples clef/valeur mutables.
- ▶ Opérations :
  - ▶ K getKey().
  - ▶ V getValue().
  - ▶ V setValue(V value).

```
1 HashMap<String, Integer> map = ...  
2 for(Map.Entry<String,Integer> entry : map.entrySet()) {  
3     System.out.println("key " + entry.getKey());  
4     System.out.println("value " + entry.getValue());  
5 }
```

# Conclusion

- ▶ Le choix de la structure de données est important.
- ▶ Selon l'algorithme à implémenter :
  - ▶ Choisir l'interface (List, Set, Queue, Map...).
  - ▶ Choisir l'implémentation.
- ▶ Si hésitation entre deux implémentations, faire des tests avec de larges données.