

Programmation Orientée Objet

Collections

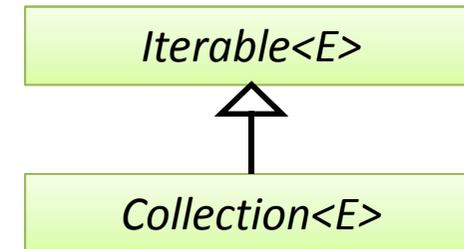
Frédéric Mallet

<http://deptinfo.unice.fr/~fmallet/>

Objectifs

- ❑ Réutilisation
 - Utilisation des types génériques
- ❑ Structures de contrôle
 - for-each & Iterable
- ❑ Structures de données
 - Collections
 - Tables d'association

java.util.Collection



□ Groupe d'éléments

- **capacité** extensible
- **taille** dynamique
- Les collections sont des objets

□ Les méthodes

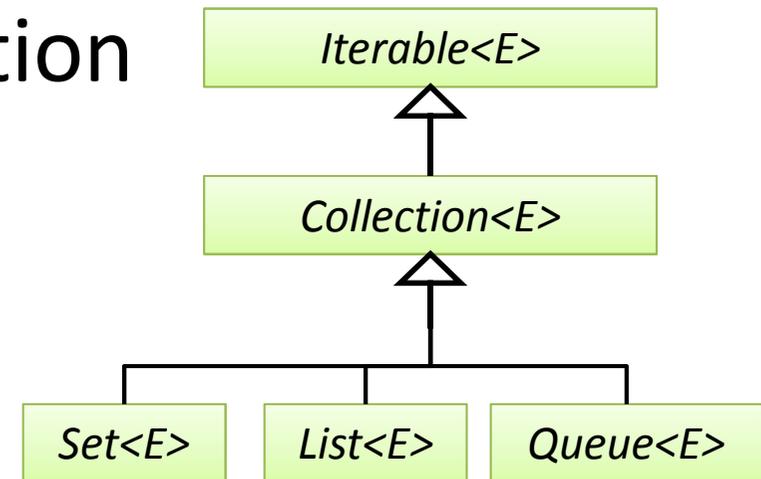
```
[*] boolean add(E)
[*] boolean addAll(Collection<?>)
[*] void clear()
    boolean contains(E)
    boolean containsAll(Collection<?>)
    boolean isEmpty()
[*] boolean remove(E)
[*] boolean removeAll(Collection<?>)
[*] boolean retainAll(Collection<?>)
    int size()
```

java.util.Collection

- ❑ Groupe d'éléments
 - capacité extensible
 - taille dynamique
 - Les collections sont des objets

- ❑ Différentes interfaces

- Ensemble
- Liste
 - Ensemble ordonné, accès par indice
- Files (double sens)
- Table association (clé/valeur)



`java.util.Set`

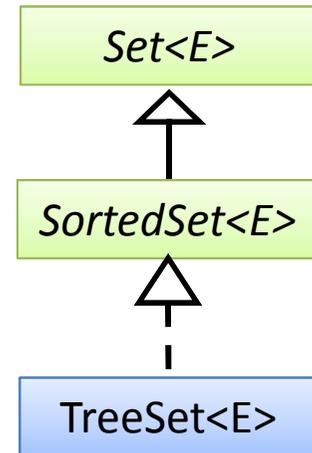
`java.util.List`

`java.util.Queue (Deque)`

`java.util.Map`

Classes concrètes: **Set**

- Ensemble `java.util.Set` `java.util.SortedSet`
 - Table de hachage : `java.util.HashSet`
 - Arbre balancé : `java.util.TreeSet`
 - Hachage + liste chaînée : `java.util.LinkedHashSet`



□ Attention:

- Les classes concrètes garantissent qu'on n'ajoute pas un objet qui est déjà présent (au sens de la méthode `equals`)
- => Aucune garantie avec des objets modifiables
- `HashSet` utilise les méthodes **`equals`** et **`hashCode`** de la classe **`Object`**
 - Lorsque deux objets sont égaux (au sens de **`equals`**), leur méthode **`hashCode`** doit renvoyer le même code !

Classes concrètes : List

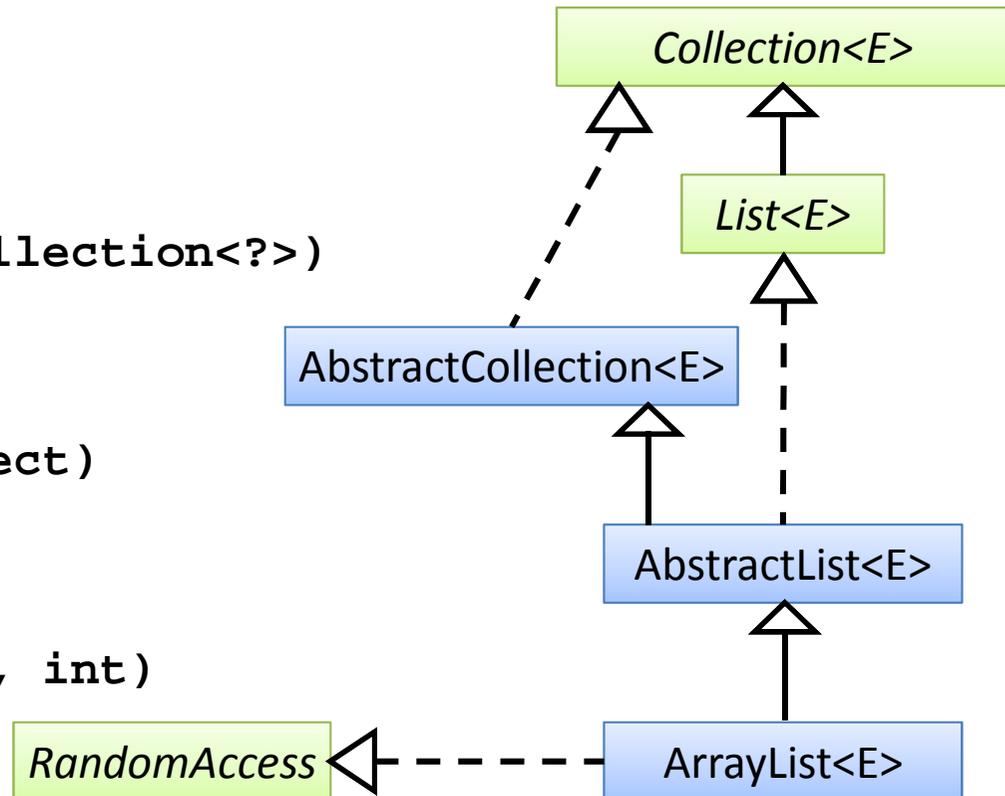
☐ Liste

`java.util.List`

- Tableau à capacité variable : `java.util.ArrayList`
- Liste chaînée : `java.util.LinkedList`

☐ Les méthodes

```
[*] void add(int, E)
[*] void addAll(int, Collection<?>)
    E get(int)
    int indexOf(Object)
    int lastIndexOf(Object)
[*] void remove(int)
[*] E set(int, E)
    List<E> subList(int, int)
```

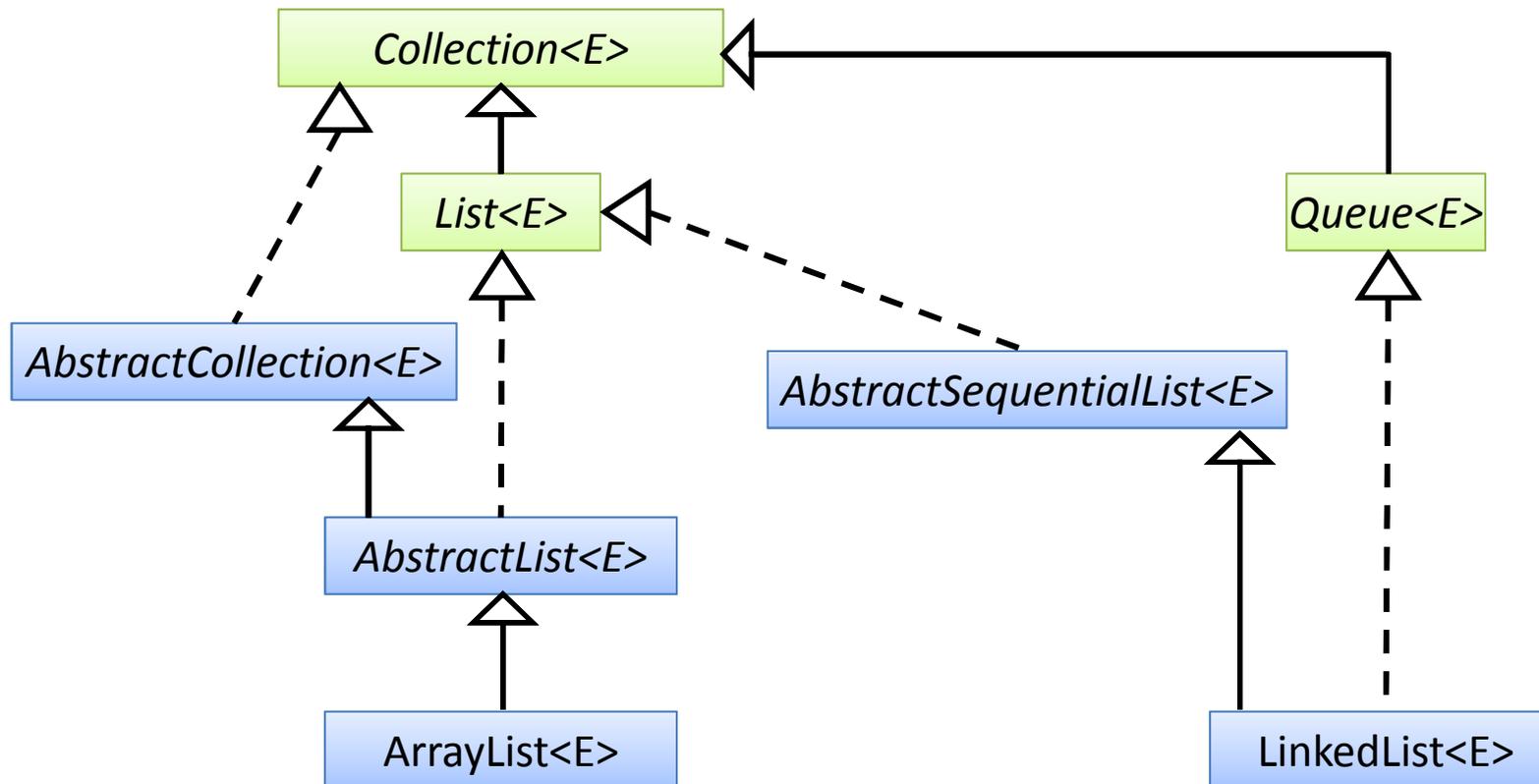


Classes concrètes : Files

□ File

`java.util.Queue` `java.util.Deque`

- Tableau à capacité variable : `java.util.ArrayDeque`
- Liste chaînée : `java.util.LinkedList`



Collections : syntaxe

- ❑ Déclaration (référence)
 - `java.util.ArrayList liste1;`
- ❑ Construction d'une liste (pas de ses éléments)
 - `liste1 = new java.util.ArrayList();`
- ❑ Affectation de collections (pas de copie globale)
 - `java.util.ArrayList liste2 = liste1;`
- ❑ Taille (dynamique)
 - `liste1.size() → 0`

Opérations

❑ Ajout (toutes les classes héritent de Object)

- `liste1.add(new Rectangle());`
- `liste1.add(new ComplexeCartesien(0,1));`

❑ Accès en lecture (par indice)

- `Object o1 = liste1.get(0);`
- ~~`o1.aire();`~~
- `Object o2 = liste1.get(1);`
- ~~`((Rectangle)o2).aire();`~~ ClassCastException

❑ Accès en écriture (par indice)

- `liste1.set(0, new Carré(10));`
- `liste1.set(1, o1);`

Paquetage

❑ Classes regroupées par thèmes

- java.util, java.awt, java.lang, ...

❑ La directive import

- Avant le bloc de déclaration de classe
- Exemple:

```
import java.util.ArrayList;  
class Toto {  
    ArrayList liste1 = new ArrayList();  
}
```

Collections génériques : syntaxe

❑ Déclaration

(référence)

- `ArrayList<IComplexe> complexes;`

❑ Construction

- `complexes = new ArrayList<IComplexe>();`

❑ Ajout

- `complexes.add(new ComplexeCartesien(0,1));`
- ~~`complexes.add(new Rectangle(10,20));`~~

❑ Accès en lecture (par indice)

- `IComplexe c1 = complexes.get(0);`
- `c1.reelle();`



« interface »
IComplexe

+ *reelle()* : double
+ *imaginaire()* : double

Parcours des éléments

```
ArrayList complexes = new ArrayList(); ...
```

❑ Par indice (seulement pour les listes)

```
for(int i = 0; i < complexes.size(); i++) {  
    IComplexe c = (IComplexe) complexes.get(i);  
    c.reelle();  
}
```

❑ For-each : itérateurs

```
for(Object c : complexes) {  
    ((IComplexe)c).reelle();  
}
```

« interface »
IComplexe

+ *reelle()* : double
+ *imaginaire()* : double

Parcours des éléments

```
ArrayList<IComplexe> complexes; ...
```

❑ Par indice (seulement pour les listes)

```
for(int i = 0; i < complexes.size(); i++) {  
    IComplexe c = complexes.get(i);  
    c.reelle();  
}
```

❑ For-each : itérateurs

```
for(IComplexe c : complexes) {  
    c.reelle();  
}
```

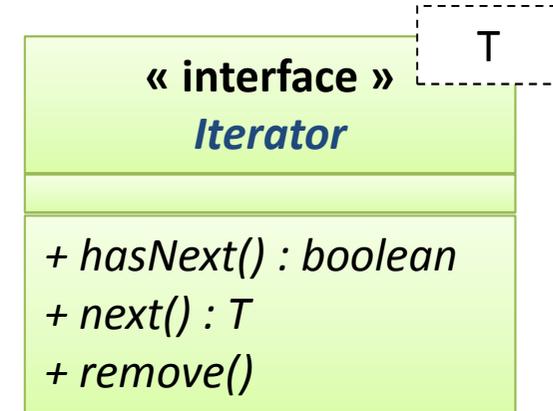
« interface »
IComplexe

+ *reelle()* : double
+ *imaginaire()* : double

Parcours des éléments : **Iterator**

□ Interface `java.util.Iterator`

```
interface Iterator<T> {
    boolean hasNext();
    T next();
    void remove();
}
```



□ Usage

```
Collection<IComplexe> complexes = ...;
Iterator<IComplexe> it = complexes.iterator();
while(it.hasNext()) {
    IComplexe c = it.next();
    c.reelle();
}
```

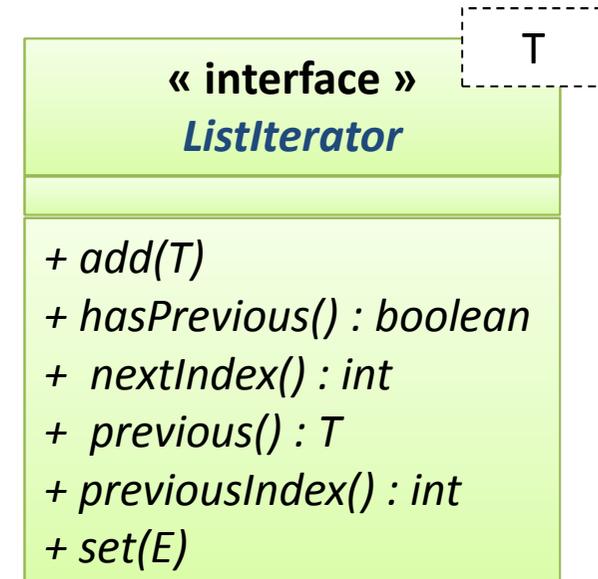
Parcours des éléments : **Iterator**

- ❑ Méthode `next()`
 - `NoSuchElementException` si `hasNext()` renvoie `false`
- ❑ Méthode `remove()`
 - Optionnelle
 - `NoSuchOperationException` si pas supportée
 - Enlève le dernier élément énuméré
 - `IllegalStateException` si `next()` n'a pas été appelé
- ❑ La collection ne doit pas être modifiée pendant l'itération
 - `ConcurrentModificationException` si on essaye d'enlever ou d'ajouter des éléments pendant l'itération

Parcours des éléments : **ListIterator**

- ❑ Seulement avec des **List** (pas avec toutes les collections)
- ❑ Interface `java.util.ListIterator`

```
interface ListIterator<T> extends Iterator<T> {
    add(T);
    boolean hasPrevious();
    int nextIndex();
    T previous();
    int previousIndex();
    set(E);
}
```



Parcours des éléments

Interface `java.lang.Iterable`

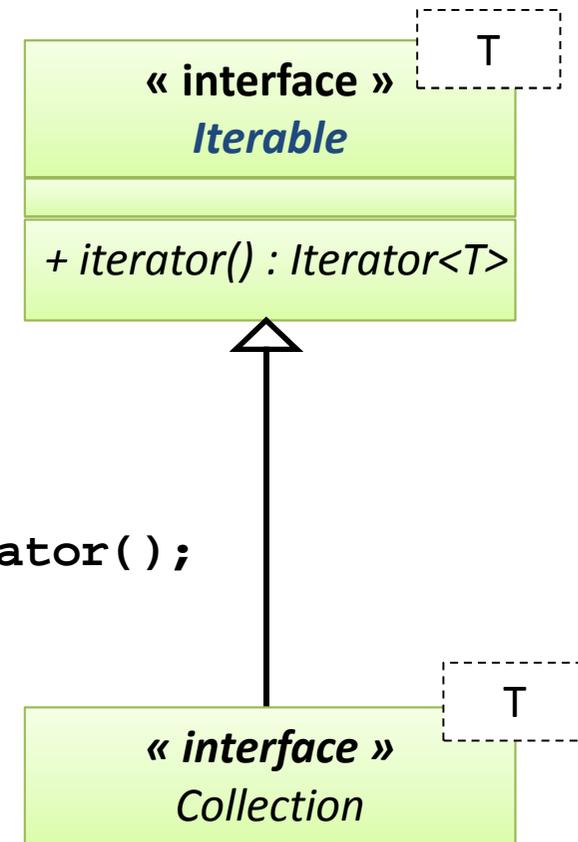
```
interface Iterable<T> {
    Iterator<T> iterator();
}
```

Usage

```
Collection<IComplexe> complexes = ...;
Iterator<IComplexe> it = complexes.iterator();
while(it.hasNext()) {
    IComplexe c = it.next();
    c.reelle();
}
```



```
for(IComplexe c : complexes) c.reelle();
```



Auto (un)boxing

□ Avec les types primitifs ?

- ~~ArrayList<int> listeEntiers;~~
- ArrayList<Integer> listeEntiers;

□ Ajout d'éléments

- listeEntiers.add(new Integer(12));  Auto boxing
-

□ Lecture

- int v = listeEntiers.get(0).intValue();  Auto unboxing
-

Surcharge de remove

☐ Attention il y a donc deux méthodes remove

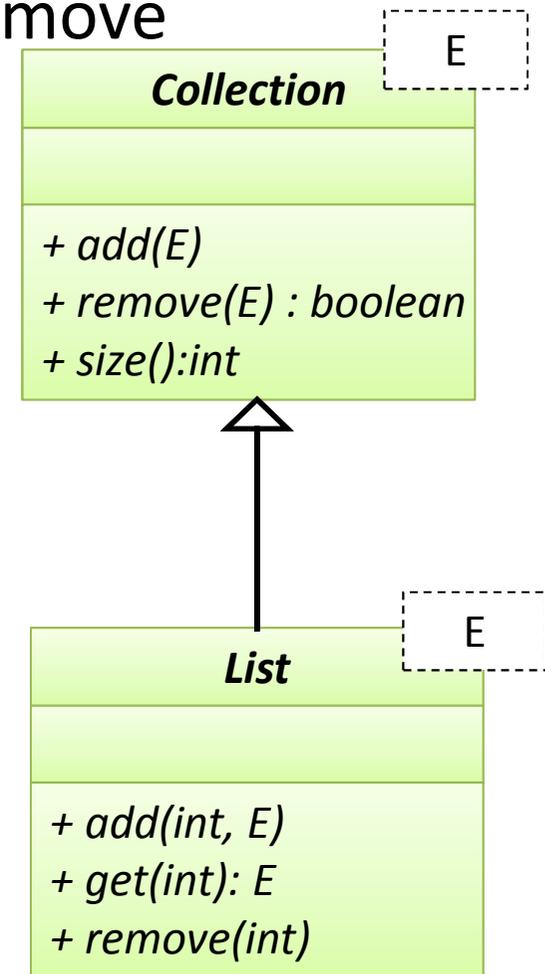
- `void remove(int index);`
- `boolean remove(E element);`

☐ Avec les listes d'entiers : $E \rightarrow \text{Integer}$

- `void remove(int index);`
- `boolean remove(Integer element);`

☐ Usage

- `liste.remove(12);`
- `liste.remove(new Integer(12));`



Classe utilitaire : `java.util.Collections`

□ Manipulations courantes de tableaux

- Recherche:

```
int binarySearch(List<T> li, T val);  
T min(Collection<T> c);  
T max(Collection<T> c);
```
- Copy:

```
void copy(List<T> l1, List<T> l2);
```
- Comparaison:

```
boolean disjoint(List<T> l1, List<T> l2);
```
- Remplissage:

```
void fill(List<T> li, T val);
```
- Divers:

```
void replaceAll(List<T> li, T old, T new);  
void reverse(List<T> li);  
void rotate(List<T> li, int distance);  
void shuffle(List<T> li);
```
- Tri:

```
void sort(List<T extends Comparable> li)
```

TRI ET RECHERCHE DANS UNE COLLECTION

Example

□ Tri et affichage d'une liste de chaînes de caractères

```
List<String> liste = new ArrayList<String>();  
liste.add("Pierre Jacques");  
liste.add("Pierre Paul");  
liste.add("Jacques Pierre");  
liste.add("Paul Jacques");  
System.out.println("Avant tri : " + liste);  
Collections.sort(liste);  
System.out.println("Après tri : " + liste);
```

```
Avant tri : [Pierre Jacques, Pierre Paul, Jacques Pierre, Paul Jacques]  
Après tri : [Jacques Pierre, Paul Jacques, Pierre Jacques, Pierre Paul]
```

String est un Comparable <String>!

[java.lang.ClassCastException:](#)

[Complexe cannot be cast to java.lang.Comparable](#)

Interface Comparable<T>

- Impose un ordre total entre les objets qui réalisent cette interface
 - Ordre naturel de la classe !
 - Pas nécessairement consistant avec equals mais c'est fortement recommandé

```
int compareTo(T element);
```

□ Règles:

- $\text{sgn}(x.\text{compareTo}(y)) == -\text{sgn}(y.\text{compareTo}(x))$
- Transitivité:
 - $x.\text{compareTo}(y) > 0 \ \&\& \ y.\text{compareTo}(z) > 0 \Rightarrow x.\text{compareTo}(z) > 0$
- $x.\text{compareTo}(y) == 0 \Rightarrow$
 $\text{sgn}(x.\text{compareTo}(z)) == \text{sgn}(y.\text{compareTo}(z))$

Exemple

□ Ordre lexicographique sur les nombres complexes

```
class Complexe implements Comparable<Complexe> {  
    double reelle, imaginaire;  
  
    @Override  
    public int compareTo(Complexe c) {  
        if (this.reelle==c.reelle)  
            return (int)Math.round(this.imaginaire-c.imaginaire);  
        return (int)Math.round(this.reelle-c.reelle);  
    }  
}
```

Avant: [1, 1+i, i]

Après: [i, 1, 1+i]

Interface Comparator<T>

□ Réalise un ordre

- Lorsqu'on ne veut pas utiliser l'ordre naturel
- Lorsque les éléments ne réalisent pas Comparable et qu'on ne peut/veut pas modifier
 - Exemple:
 - il n'y a pas d'ordre naturel sur les nombres complexes
 - Deux ordres « non » naturels: lexicographique ou produit

□ Signature

```
interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

Un ordre pour les complexes

□ Ordre lexicographique

```
class OrdreLexicographique implements Comparator<Complexe> {

    @Override
    public int compare(Complexe c1, Complexe c2) {
        if (c1.reelle==c2.reelle)
            return (int)Math.round(c1.imaginaire-c2.imaginaire);
        return (int)Math.round(c1.reelle-c2.reelle);
    }
}
```

□ Usage

```
Complexe[] tc = { ... };
Arrays.sort(tc, new OrdreLexicographique());
```

```
Avant:[1, 1+i, i]
Après:[i, 1, 1+i]
```

Recherche dichotomique

□ Deux méthodes

- `static <T> int binarySearch(List l, T e);`
 - Calcule l'indice de l'élément `e` dans la liste `l`.
 - Suppose que la liste est triée dans l'ordre naturel de `T`
 - Les éléments de `l` doivent être `Comparable<T>`
- `static <T> int binarySearch(List l, T e, Comparator<T> c);`
 - Idem à l'autre sauf que l'ordre est donné en paramètre `c`

□ Usage (T -> Complexe)

- `List<Complexe> liste = ...`
- `Collections.sort(liste, new OrdreLexicographique());`
- `Collections.binarySearch(liste, new Complexe(0,1),
new OrdreLexicographique());`

Collections: JDK 1.1

□ JDK 1.1

- `java.util.Vector` => `java.util.ArrayList`
- `java.util.Enumeration` => `java.util.Iterator`
- `java.util.Hashtable` => `java.util.HashMap`
- Thread-safe
 `List<T> Collections.synchronizedList(List<T> l)`

□ Types génériques

- Java 5 (JDK 1.5)

Liste vers tableau

□ Deux méthodes possibles

▪ `Object[] toArray()`

```
List<String> liste = new ArrayList<String>();  
liste.add("premier");  
Object[] t = liste.toArray();  
String[] tS = (String[])liste.toArray();
```

▪ `<T> T[] toArray(T[] a)`

```
String[] t = liste.toArray(new String[0]);  
t.length → 1
```

Méthodes optionnelles

- ❑ Certaines méthodes de l'interface Collection sont dites optionnelles
 - add, addAll, clear, remove, removeAll, retainAll
 - Elles renvoient parfois `java.lang.UnsupportedOperationException`
 - Exemples:
 - Dans une liste modifiable on ne veut pas pouvoir enlever des éléments

Convention

- ❑ Une interface ne peut pas imposer la forme des constructeurs (comme le feraient des constructeurs)
- ❑ Par convention, les classes qui réalisent l'interface Collection doivent avoir au moins 2 constructeurs
 - Un constructeur sans paramètre
 - Un constructeur avec en paramètre une Collection dont les éléments sont de type compatible
- ❑ Exemple
 - `ArrayList<E>()`
 - `ArrayList<E>(Collection<? extends E> c)`

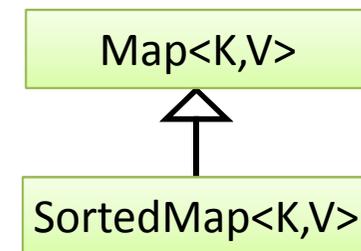
TABLES D'ASSOCIATION (MAP)

Classes concrètes

- ❑ Table d'association `java.util.Map`, `java.util.SortedMap`
 - Table de hachage : `java.util.HashMap`
 - Arbre équilibré : `java.util.TreeMap`
 - Hachage + liste chaînée : `java.util.LinkedHashMap`

- ❑ Utilisation
 - Association 1 clé => 1 valeur
 - Les clés sont uniques (au sens de equals)

- ❑ Exemples:
 - `HashMap<Integer, String> map;`
 - `HashMap<String, String> dictionnaire;`
 - `HashMap<String, Complexe> variables;`



HashMap<K,V>

□ Propriétés

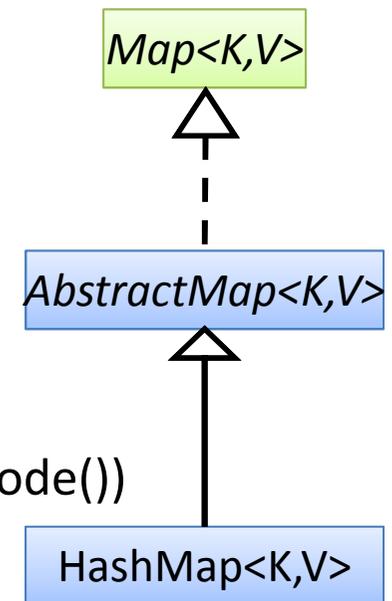
- Accès à une valeur en temps constant
 - Temps d'accès similaire à un tableau
 - Mais K n'est pas nécessairement un entier (cf. hashCode())

□ Example

```
HashMap<String, Integer> variables = new HashMap<String, Integer>();
variables.put("x", 12);
variables.put("y", 24);
System.out.println("Avant:" + variables);
variables.put("x", variables.get("x") + variables.get("y"));
System.out.println("Après:" + variables);
```

Avant:{y=24, x=12}

Après:{y=24, x=36}



Interface Map<K,V>

□ Les méthodes

```
[*] void clear()
    boolean containsKey(Object)
    boolean containsValue(Object)
    V get(Object)
    boolean isEmpty();
    Set<K> keySet();
    Collection<V> values();
    Set<Map.Entry<K,V>> entrySet();

[*] V put(K, V)
[*] void putAll(Map<? Extends K, ? extends V>)
[*] V remove(Object)
    int size()
```

Attention

- ❑ Les clés doivent être des objets immutables
 - Le fonctionnement n'est pas garanti si on modifie une clé
 - Pour enlever une clé
 - Il faut d'abord enlever l'ancienne clé
 - Puis ajouter la nouvelle

- ❑ Convention
 - Les classes qui réalisent l'interface Map doivent avoir au moins deux constructeurs
 - 1 par défaut
 - 1 qui prend en paramètre un Map avec des types compatibles
 - Impossible d'imposer cette règle de typage avec une interface !

Parcours des éléments d'une Map

□ Deux parcours possibles

- Par les clés (cf. méthode `keySet()`)
- Par les valeurs (cf. méthode `entrySet()`)
- On manipule alors des ensembles (non ordonnés)
 - Avec leurs itérateurs
 - L'ordre de parcours n'est pas garanti constant
 - => sinon il faut utiliser `TreeMap<K,V>` qui est une `SortedMap`

TreeMap<K,V>

- ❑ Table d'association avec des éléments ordonnés
 - Soit des Comparable
 - Soit avec Comparator
- ❑ Structure d'arbre (binaire)
 - Accès en $O(\log_2(n))$ où n est le nombre d'éléments
 - $\log(n)$ est une borne supérieure de la profondeur de l'arbre

