Collection <E>

Cnam Paris
jean-michel Douin
14 Septembre 2005

Notes de cours java : les collections <E>

Les notes et les Travaux Pratiques sont disponibles en
http://jfod.cnam.fr/tp_cdi/{douin/}

Sommaire

- **Pourquoi, Objectifs**

- **Interface Collection<E>, Iterable<E> et Iterator<E>**
- **Classe AbtractCollection<E>**
- **Interface Set<E> et List<E>**
- **Classe AbtractList<E>**
- **Les concrètes Vector<E> et Stack<E>**
- **Interface SortedSet<E>**
- **Interface Map<K,V> et Map.Entry<K,V>**
- **Classes Collections et Arrays**

- **Le pattern Fabrique<T>**

Principale bibliographie

- **Le tutorial de Sun**

  http://java.sun.com/docs/books/tutorial/collections/

- **Introduction to the Collections Framework**

  http://developer.java.sun.com/developer/onlineTraining/collections/

- http://java.sun.com/j2se/1.5.0/docs/api/java/util/Collection.html
- http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html

Pourquoi

- **Organisation des données**
  - **Listes, tables, sacs, arbres, piles, files …**
  - **Données par centaines, milliers, millions ?**

- **Quel choix ?**
  - **En fonction de quels critères ?**
    - **Performance en temps d'exécution**
      - **lors de l'insertion, en lecture, en cas de modification ?**
    - **Performance en occupation mémoire**

- **Avant les collections, (avant Java-2)**
  - **Vector, Stack, Dictionary, Hashtable, Properties, BitSet (implémentations)**
  - **Enumeration (parcours)**

- **Hérite des STL (Standard Template Library) C++**

Les collections en java-2 : Objectifs

- **Reduces programming** effort by providing useful data structures and algorithms so you don't have to write them yourself.
- **Increases performance** by providing high-performance implementations of useful data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be easily tuned by switching implementations.
- **Provides interoperability** between unrelated APIs by establishing a common language to pass collections back and forth.
- **Reduces the effort required to learn** APIs by eliminating the need to learn multiple ad hoc collection APIs.
- **Reduces the effort required to design** and implement APIs by eliminating the need to produce ad hoc collections APIs.
- **Fosters software reuse** by providing a standard interface for collections and algorithms to manipulate them.

Ces objectifs seront-ils atteints ? … à suivre …

<E> comme généricité ou généralités…

- **Une collection d ’objets,**
- **si c’est une collection homogène**

  Le type devient un « paramètre de la classe »
  Le compilateur vérifie alors l’absence d’ambiguïtés
  C’est une analyse statique (et uniquement)
  Cela engendre une inférence de types à la compilation

- **sinon tout est Object ...**

Généricité / Généralités

```
public class Liste<T>{
  private ...
  public void add(T t){...;}
  public T first(){ return ...;}
  public T last(){ return ...;}
}

Liste <Integer> l = new Liste <Integer>();
…;
Integer i = l.first();

Liste <String> l1 = new Liste <String>();
…;
String s = l1.first();
Boolean b = l1.first(); <-- erreur de compilation
```

Généricité, syntaxe

- **Java.util.Collection<?>**

  Compatible avec n'importe quelle classe

```
public static void afficher(java.util.Collection<?> c){
   for( Object o : c)
     System.out.println(o);
  }
```

- **<? extends E>**

  Contrainte sur l'arbre d'héritage,

  E doit être la super classe de la classe inconnue

- **<? super E>**

  E doit être une sous classe de la classe inconnue

<T> T[] toArray(T[] a);

Notes : http://jfod.cnam.fr/tp_cdi/douin/JavaGeneric.pdf

Mixité permise, mais attention …

- **Avant** la version 1.5

```
List l = new ArrayList();
l.add(new Boolean(true));
l.add(new Integer(3));
```

- **Après**

```
List<Integer> l = new ArrayList<Integer>();
l.add(new Boolean(true));
l.add(new Integer(3));
```

← erreur de compilation

- **Mixité**

```
List l = new ArrayList<Integer>();
l.add(new Boolean(true));
l.add(new Integer(3));
```

← un message d'erreur et compilation effectuée !!!

Sommaire Collections en Java

- ***Quelles fonctionnalités ?***
- ***Quelles implémentations partielles ?***
- ***Quelles implémentations complètes ?***
- ***Quelles passerelles Collection <-> tableaux ?***

Les Collections en Java, paquetage java.util

- ***Quelles fonctionnalités ?***

  Quelles interfaces ?

  Collection<E>, Iterable<E>, Set<E>, SortedSet<E>, List<E>, Map<K,V>, SortedMap<K,V>, Comparator<T>, Comparable<T>

- ***Quelles implémentations partielles ?***

  Quelles classes incomplètes (dites abstraites) ?

  AbstractCollection<E>, AbstractSet<E>, AbstractList<E>, AbstractSequentialList<E>, AbstractMap<K,V>

- ***Quelles implémentations complètes ?***

  Quelles classes concrètes (toutes prêtes) ?

  LinkedList<E>, ArrayList<E>,

  TreeSet<E>, HashSet<E>,

  WeakHashMap<K,V>, HashMap<K,V>, TreeMap<K,V>

- ***Quelles passerelles ?***

  Collections et Arrays

Les Collections en Java : deux interfaces

- **interface Collection<T>**

  Pour les listes et les ensemble

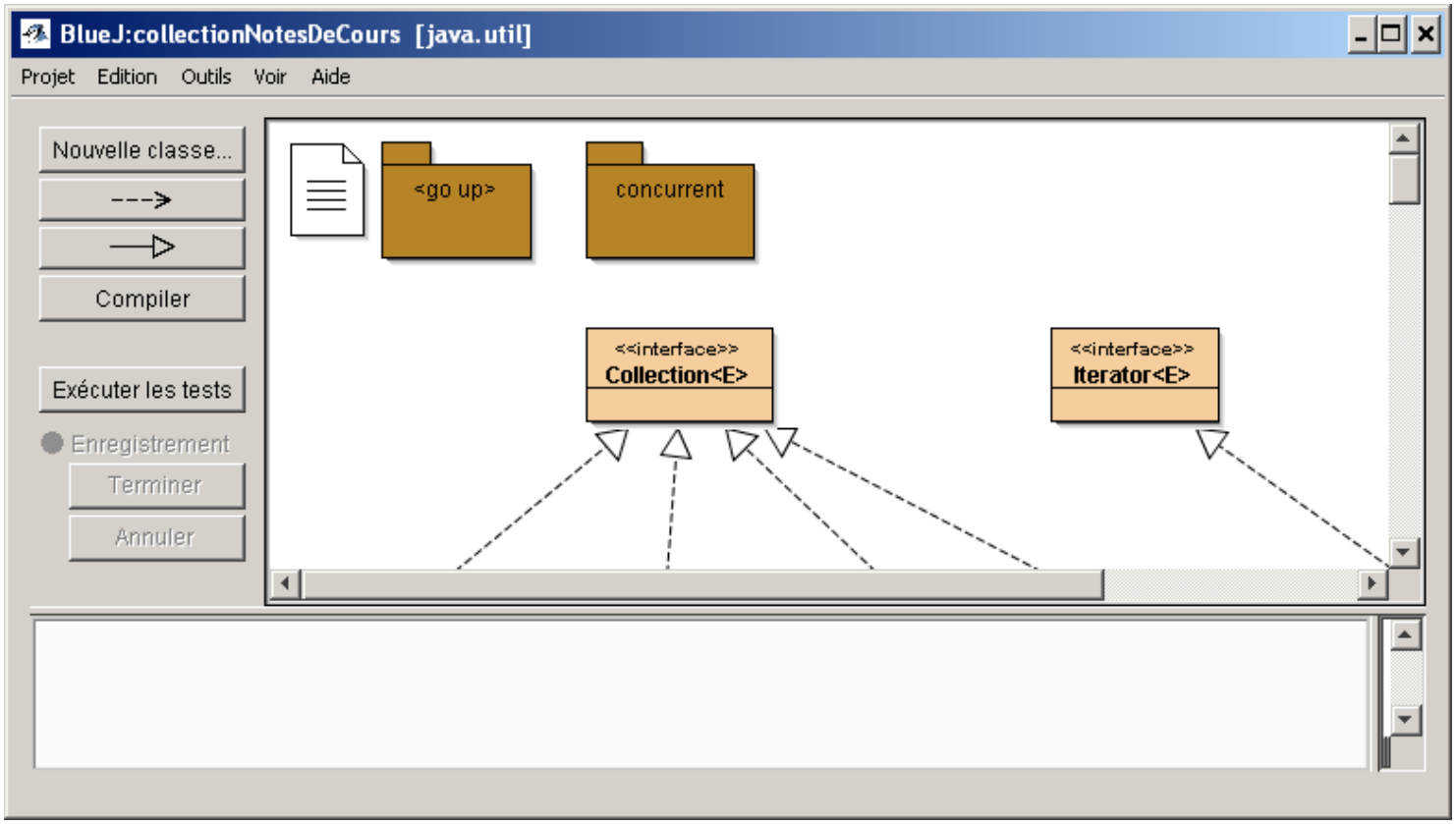
```
package java.util;
public interface Collection<E> extends Iterable<E>{
…
```

- **interface Map<K,V>**

  Pour les dictionnaires

```
package java.util;
public interface Map<K,V> {…
```

Interface java.util.Collection<E>



Principe une interface « Racine » et deux méthodes fondamentales :

boolean add(E o);

Iterator<E> iterator();

Interface java.util.Collection<E>

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

  // interrogation
  int size();
  boolean isEmpty();
  boolean contains(Object o);
  Iterator<E> iterator();
  Object[] toArray();
  <T> T[] toArray(T[] a);
```

Interface java.util.Collection<E> suite

```
// Modification Operations

    boolean add(E o);
    boolean remove(Object o);

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


    // Comparison and hashing
    boolean equals(Object o);
    int hashCode();
}
```

Iterable<T>

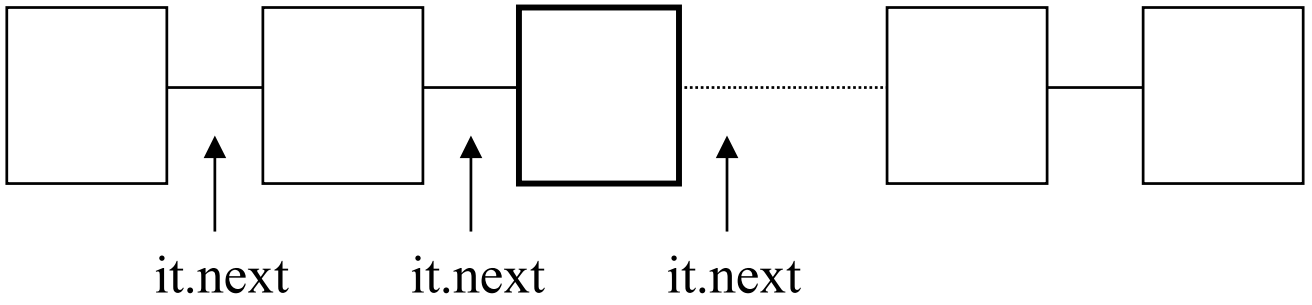
- ***public interface Collection<E> extends Iterable<E>***

- **Paquetage java.lang**

```
public interface Iterable<T>{

    Iterator<T> iterator();

}
```

Iterator<E>

```
paquetage java.util;
public interface Iterator<E>{
  E next();
  boolean hasNext();
  void remove();
}
```



java.util.Iterator<E> un usage

```
public static <T> void filtrer(    Collection<T> collection,
                                   Condition<T> condition){

  Iterator<T> it = collection.iterator();
  while (it.hasNext()) {
        T t = it.next();
        if (condition.isTrue(t)) {
            it.remove();
        }
  }
}

public interface Condition<T>{
  public boolean isTrue(T t);
}
```

boucle for (et Iterator)

- **Parcours d ’une Collection c**

exemple une Collection<Integer> c = new ….;

```
for( Integer i : c)
     System.out.println("  i = "  + i);
```

<==>

```
for(Iterator it = c.iterator(); it.hasNext();)
   System.out.println("  i = "  + it.next());
```

syntaxe

for(element e : collection)*

Collection : une classe avec iterator, (ou un tableau…voir les ajouts en 1.5)

Du bon usage de Iterator<E>

Quelques contraintes

- **au moins un appel de next doit précéder l'appel de remove**
- **cohérence vérifiée avec 2 itérateurs sur la même structure**

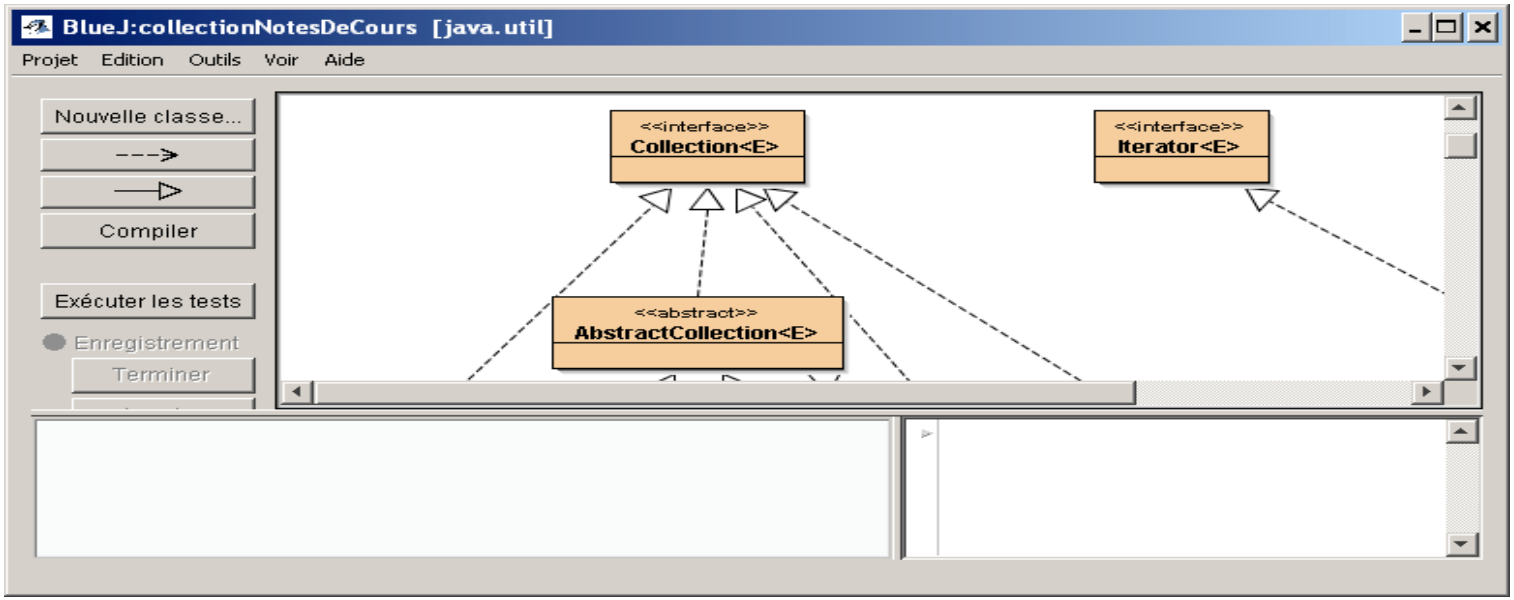
```
Collection<Integer> c = ..;
Iterator<Integer> it = c.iterator();
it.next();
it.remove();
it.remove();// → throw new IllegalStateException()

Iterator<Integer> it1 = c.iterator();
Iterator<Integer> it2 = c.iterator();
it1.next();it2.next();
it1.remove();
it2.next(); //→ throw new ConcurrentModificationException()
```

AbstractCollection<E>

- **AbstractCollection<E> implements Collection<E>**

Implémentations effectives de 13 méthodes sur 15 !

Première implémentation incomplète de Collection<E>



- **La classe incomplète : AbstractCollection<E>**

  Seules les méthodes :

  ```
  boolean add(E obj);
  Iterator<E> iterator();
  ```

  Ne sont pas développées, elles sont laissées à la responsabilité des sous classes

AbstractCollection, implémentation de containsAll

```
public boolean containsAll(Collection<?> c) {
  for( Object o : c)
    if( !contains(o)) return false

  return true;
 }
```

- *usage*

```
Collection<Integer> c  = ….
Collection<Integer> c1 = ….

if( c.containsAll(c1) …
```

AbstractCollection : la méthode contains

```
public boolean contains(Object o) {
  Iterator<E> e = iterator();
  if (o==null) {  // les éléments peuvent être « null »
     while (e.hasNext())
      if (e.next()==null)
          return true;
   } else {
     while (e.hasNext())
      if (o.equals(e.next()))
          return true;
  }
  return false;
  }
```

AbstractCollection : la méthode addAll

```
public boolean addAll(Collection<? extends E> c) {
  boolean modified = false;

  Iterator<? extends E> e = c.iterator();
  while (e.hasNext()) {
     if (add(e.next()))
        modified = true;
  }
  return modified;
  }
```

// rappel : add est laissée à la responsabilité des sous classes

```
 public boolean add(E o) {
   throw new UnsupportedOperationException();
   }
```

AbstractCollection : la méthode removeAll

```
public boolean removeAll(Collection<?> c) {
  boolean modified = false;
  Iterator<E> e = iterator();
  while (e.hasNext()) {
      if(c.contains(e.next())) {
        e.remove();
        modified = true;
      }
  }
  return modified;
 }
```

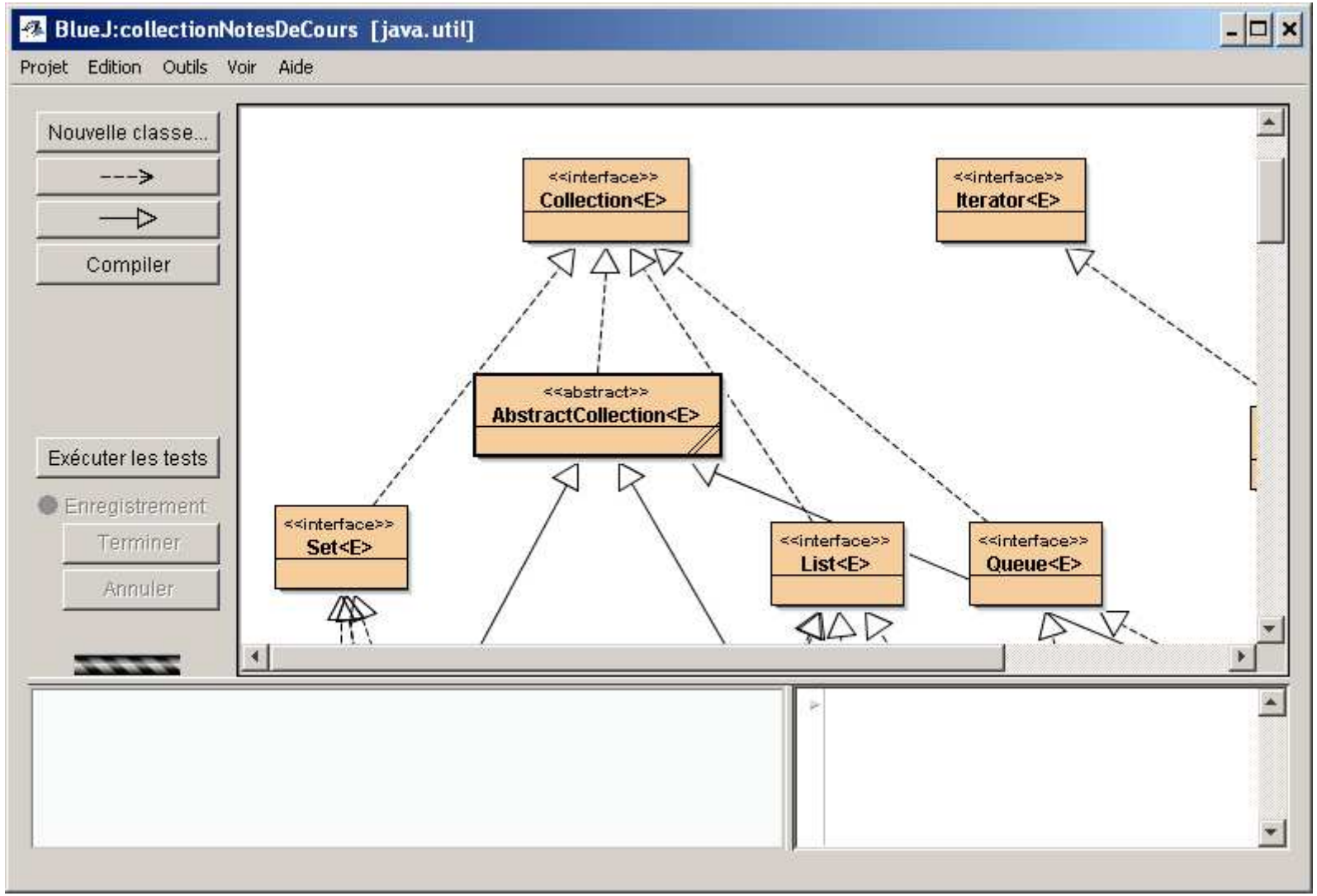
AbstractCollection : la méthode remove

```
public boolean remove(Object o) {
  Iterator<E> e = iterator();
  if (o==null) {
     while (e.hasNext())
       if (e.next()==null) {
          e.remove();
          return true;
       }
  } else {
     while (e.hasNext())
       if (o.equals(e.next())) {
          e.remove();
          return true;
       }
  }
  return false;
  }
```

Encore une : la méthode retainAll

```
public boolean retainAll(Collection<?> c) {
  boolean modified = false;
  Iterator<E> e = iterator();
  while (e.hasNext()) {
      if(!c.contains(e.next())) {
        e.remove();
        modified = true;
      }
  }
  return modified;
 }
```

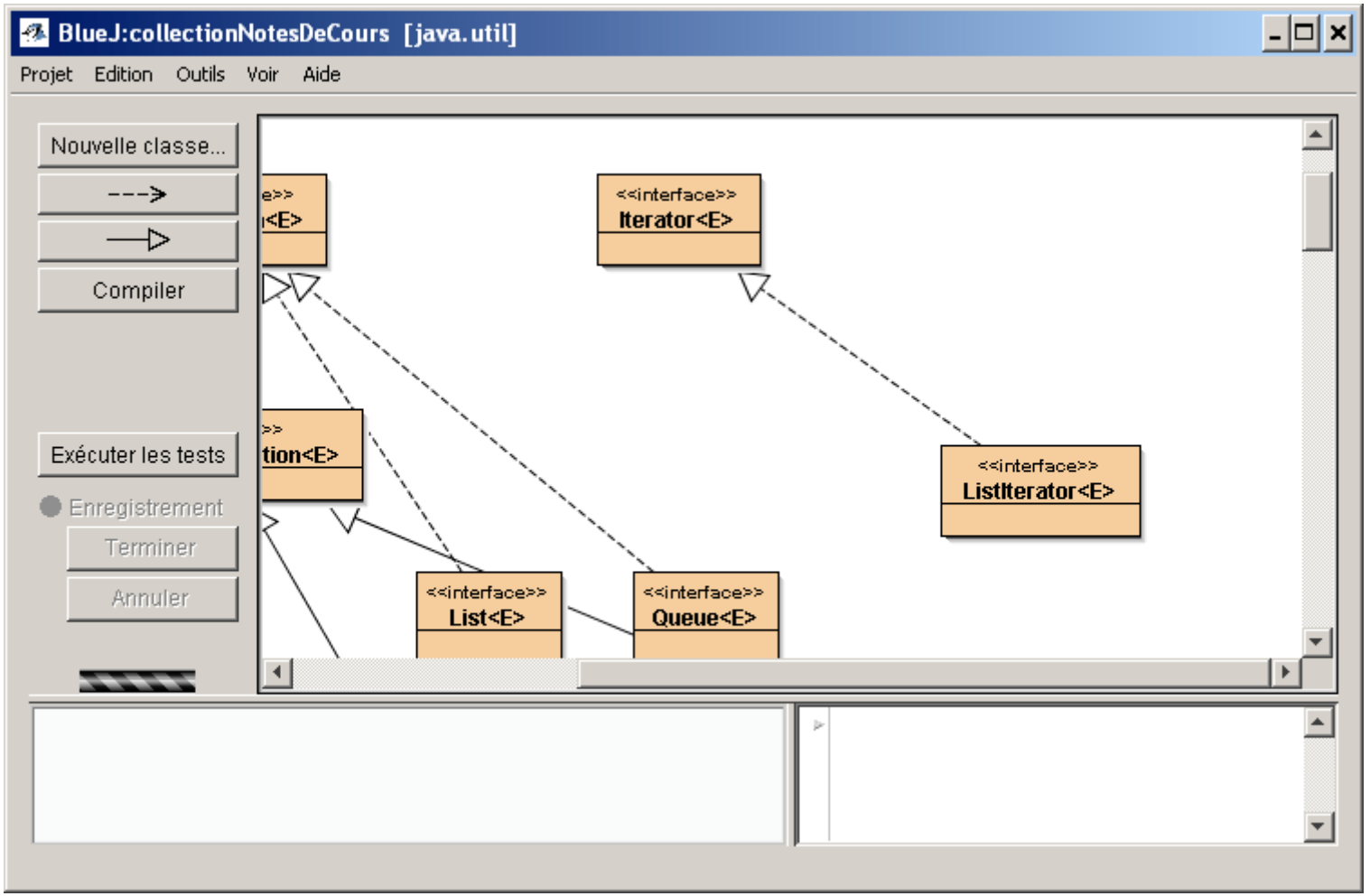
Interfaces List<E>, Set<E> et Queue<E>



List<E>

```
public interface List<E> extends Collection<E>{
  // …
   void add(int index, E element);
   boolean addAll(int index, Collection<? extends E> c);

   E get(int index);
   int indexOf(Object o);
   int lastIndexOf(Object o) ;

   ListIterator<E> listIterator();
   ListIterator<E> listIterator(int index);

   E set(int index, E element);
   List<E> subList(int fromIndex, int toIndex)
}
```

ListIterator<E> extends Iterator<E>

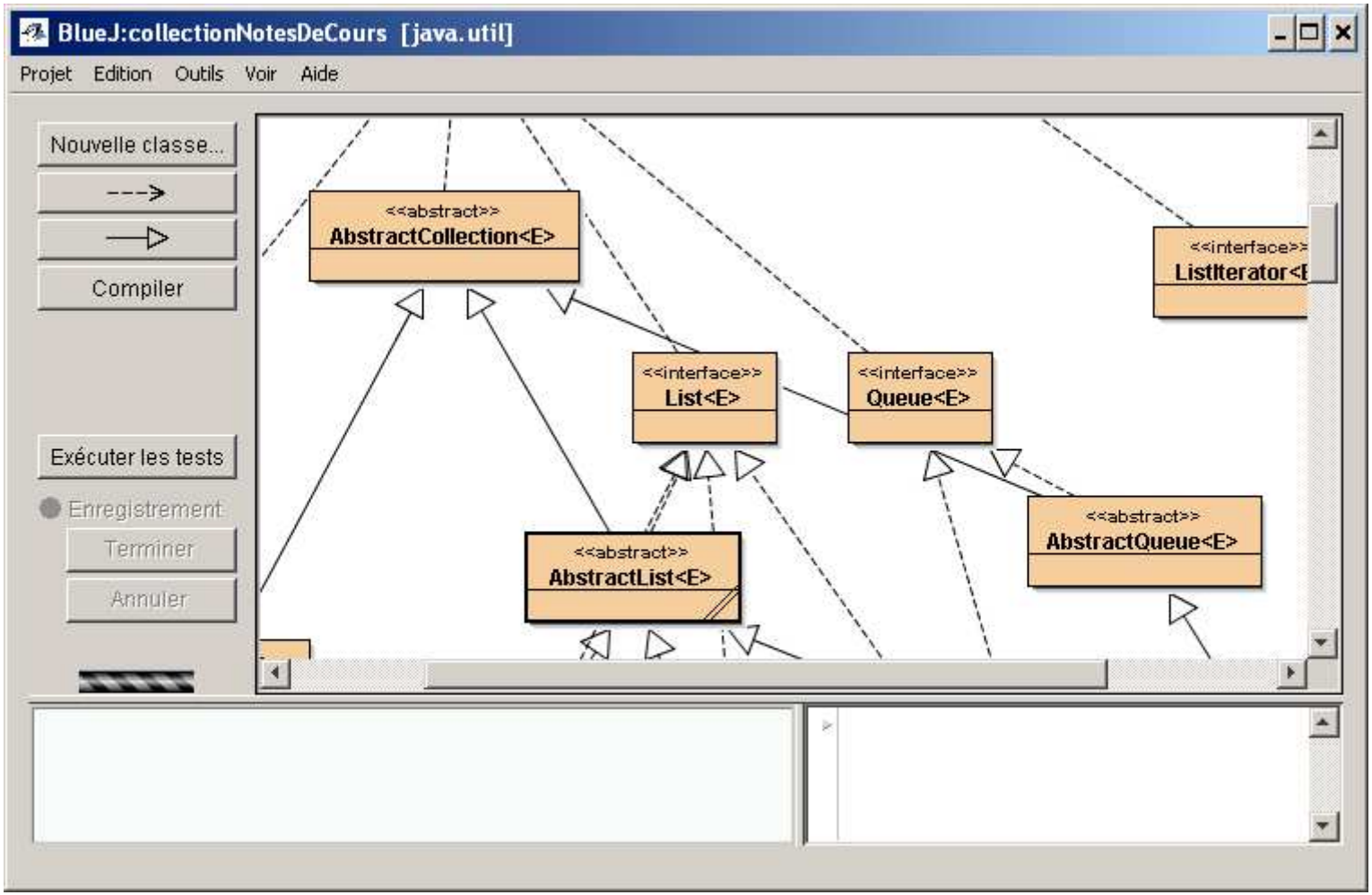


- **Parcours dans les 2 sens de la liste**
  - **next et previous**
  - **Méthode d'écriture : set(Object element)**

ListIterator<E>

```
public interface ListIterator<E> extends Iterator<E>{
   E next();
   boolean hasNext();

   E previous();
   boolean hasPrevious();

   int nextIndex();
   int previousIndex();

    void set(E o);
    void add(E o);

   void remove();
}
```

AbstractList<E>



- **AbtractList<E> et AbstractCollection<E> Même principe**
  - **add, set, get,**
  - **ListIterator iterator**

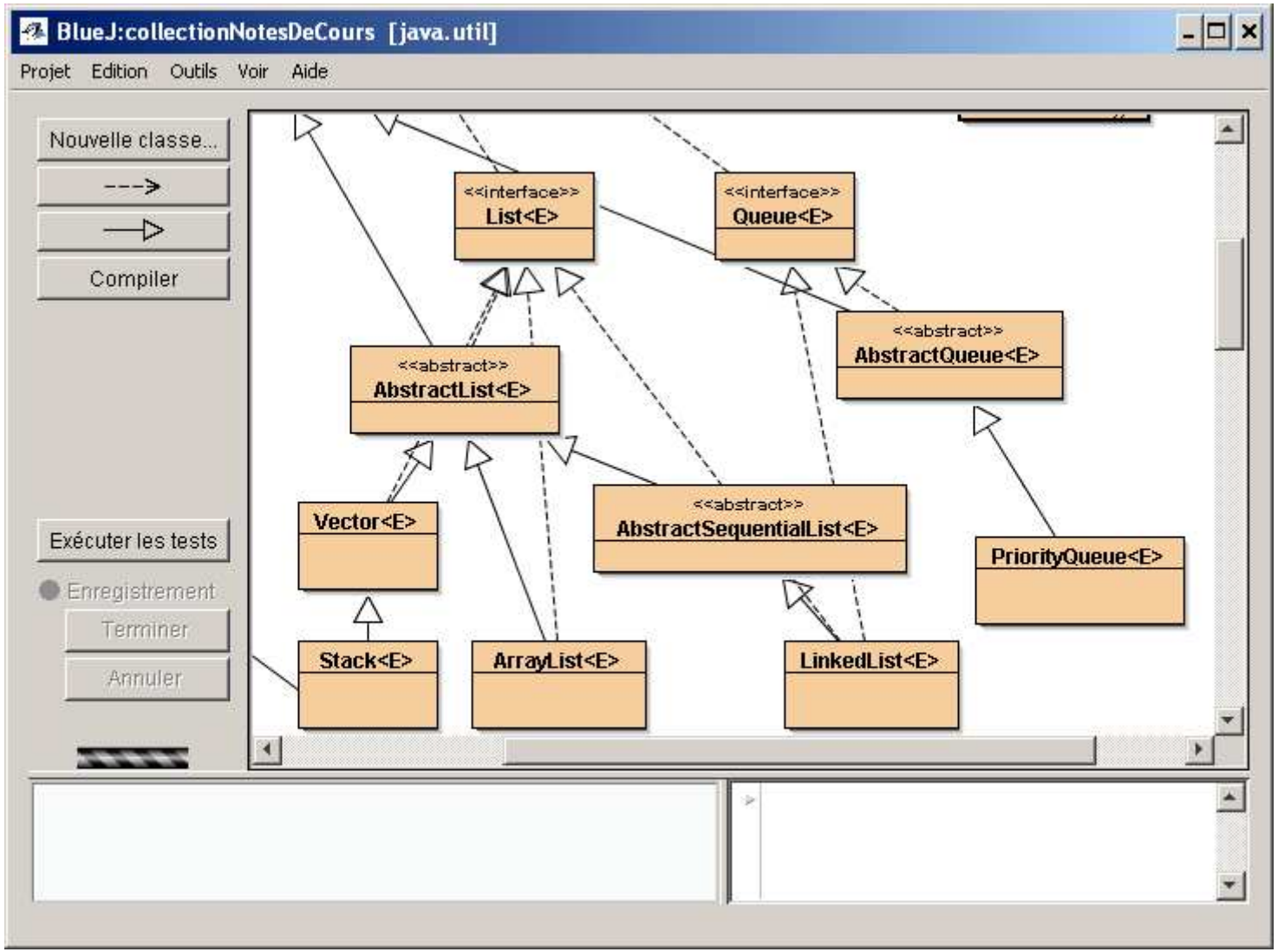
AbstractList : la méthode indexOf

```
public int indexOf(Object o) {
  ListIterator<E> e = listIterator();
  if (o==null) {
     while (e.hasNext())
       if (e.next()==null)
          return e.previousIndex();
  } else {
     while (e.hasNext())
       if (o.equals(e.next()))
          return e.previousIndex();
  }
  return -1;
 }
```

AbstractList : la méthode lastIndexOf

```
public int lastIndexOf(Object o) {
  ListIterator<E> e = listIterator(size());
  if (o==null) {
     while (e.hasPrevious())
       if (e.previous()==null)
          return e.nextIndex();
  } else {
     while (e.hasPrevious())
       if (o.equals(e.previous()))
          return e.nextIndex();
  }
  return -1;
 }
```

Les biens connues et concrètes Vector<E> et Stack<E>



Stack<E> hérite Vector<E> hérite de AbstractList<E> hérite AbstractCollection<E>

Autres classes concrètes

ArrayList<T>

LinkedList<T>

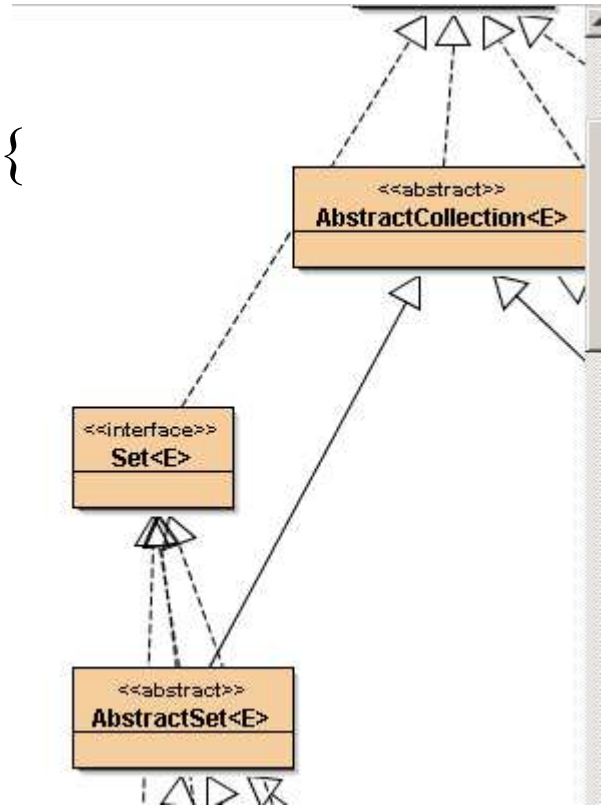
ArrayList, LinkedList : enfin un exemple concret

```
import java.util.*;
public class ListExample {
    public static void main(String args[]) {
      List<String> list = new ArrayList <String>();
      list.add("Bernardine"); list.add("Modestine"); list.add("Clementine");
      list.add("Justine");list.add("Clementine");
      System.out.println(list);
      System.out.println("2: " + list.get(2));
      System.out.println("0: " + list.get(0));

      LinkedList<String> queue = new LinkedList <String>();
      queue.addFirst("Bernardine"); queue.addFirst("Modestine");queue.addFirst("Justine");
      System.out.println(queue);
      queue.removeLast();
      queue.removeLast();
      System.out.println(queue);
   }
}
```

```
[Bernardine, Modestine, Clementine, Justine , Clementine]
2: Clementine
0: Bernardine
[Justine, Modestine, Bernardine]
[Justine]
```

Set et AbtractSet

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

  // les 16 méthodes


}
```



AbstractSet : la méthode equals

```
public boolean equals(Object o) {
  if (o == this)
     return true;

  if (!(o instanceof Set))
     return false;

  Collection c = (Collection) o;
  if (c.size() != size())
     return false;
  return containsAll(c);
  }
```

AbstractSet : la méthode hashCode

```
public int hashCode() {
  int h = 0;
  Iterator<E> i = iterator();
  while (i.hasNext()) {
     Object obj = i.next();
        if (obj != null)
           h = h + obj.hashCode();
     }
  return h;
  }
```

La somme de la valeur hashCode de chaque élément

L'interface SortedSet<E>

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

    Comparator<? super E> comparator();

    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);
    E first();
    E last();
}
```

un ensemble où l 'on utilise la relation d 'ordre des éléments

Ordre et relation

- **Interface Comparator<T>**

  Relation d'ordre de la structure de données

```
public interface Comparator<T>{
   int compare(T o1, T o2);
   boolean equals(Object o);
}
```

- **Interface Comparable<T>**

  Relation d'ordre entre chaque élément

```
public interface Comparable<T>{
   int compare(T o1);
}
```

Les concrètes

public class **TreeSet<E>** extends AbstractSet<E>implements **SortedSet**<E>,…

public class **HashSet<E>** extends AbstractSet<E>implements **Set**<E>,…

Les concrètes : un exemple

```
import java.util.*;
public class SetExample {
  public static void main(String args[]) {
    Set<String> set = new HashSet <String> ();
    set.add("Bernardine"); set.add("Mandarine");
    set.add("Modestine"); set.add("Justine");
    set.add("Mandarine");
    System.out.println(set);

    Set<String> sortedSet = new TreeSet <String> (set);
    System.out.println(sortedSet);
  }
}
```

```
[Modestine, Bernardine, Mandarine, Justine]
[Bernardine, Justine, Mandarine, Modestine]
```

Les concrètes bien connues

Elles implémentent Comparable<T>

Extrait de le documentation du j2sdk1.5

Interface Comparable

All Known Implementing Classes:

BigDecimal, BigInteger, Byte, ByteBuffer, Character, CharBuffer, Charset, CollationKey, Date, Double, DoubleBuffer, File, Float, FloatBuffer, IntBuffer, Integer, Long, LongBuffer, ObjectStreamField, Short, ShortBuffer, String, URI

Pour l'exemple : une classe Entier

```
public class Entier implements Comparable<Entier>{
  private int i;
  public Entier(int i){ this.i = i;}

  public int compareTo(Entier e){
     if (i < e.intValue()) return -1;
     else if (i == e.intValue()) return 0;
     else return 1;
  }

  public boolean equals(Object o){
     return this.compareTo((Entier)o) == 0;
  }
  public int intValue(){ return i;}
  public String toString(){ return Integer.toString( i);}
}
```

La relation d'ordre de la structure

```
public class OrdreCroissant implements Comparator<Entier>{

  public int compare(Entier e1, Entier e2){
    return e1.compareTo(e2);
  }

}

public class OrdreDecroissant implements Comparator<Entier>{

  public int compare(Entier e1, Entier e2){
    return -e1.compareTo(e2);
  }
}
```

Le test

```
public static void main(String[] args) {
   SortedSet<Entier> e = new TreeSet <Entier>(new OrdreCroissant());

    e.add(new Entier(8));
    for(int i=1; i< 10; i++){e.add(new Entier(i));}

    System.out.println(" e = " + e);
    System.out.println(" e.headSet(3) = " + e.headSet(new Entier(3)));
    System.out.println(" e.headSet(8) = " + e.headSet(new Entier(8)));
    System.out.println(" e.subSet(3,8) = " + e.subSet(new Entier(3),new
   Entier(8)));
    System.out.println(" e.tailSet(5) = " + e.tailSet(new Entier(5)));

    SortedSet<Entier>e1 = new TreeSet<Entier>(new OrdreDecroissant());
    e1.addAll(e);
    System.out.println(" e1 = "  + e1);
 }
```

e = [1, 2, 3, 4, 5, 6, 7, 8, 9]
e.headSet(3) = [1, 2]
e.headSet(8) = [1, 2, 3, 4, 5, 6, 7]
e.subSet(3,8) = [3, 4, 5, 6, 7]
e.tailSet(5) = [5, 6, 7, 8, 9]
e1 = [9,8,7,6,5,4,3,2,1]

l 'interface Queue<E>

- **public interface Queue<E>extends Collection<E>{**

- **une FIFO**

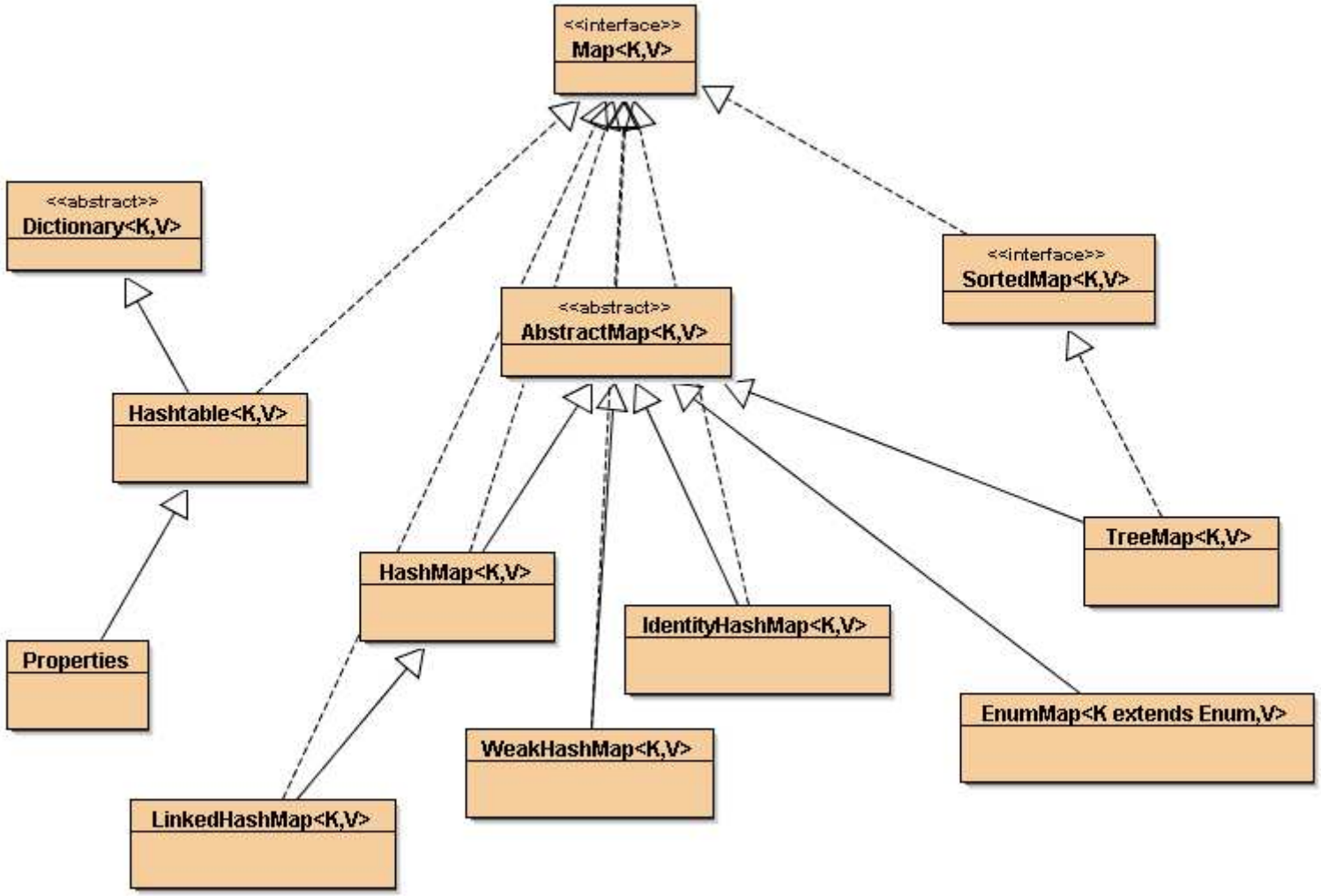
  peek, poll …

Interface Map<K,V>

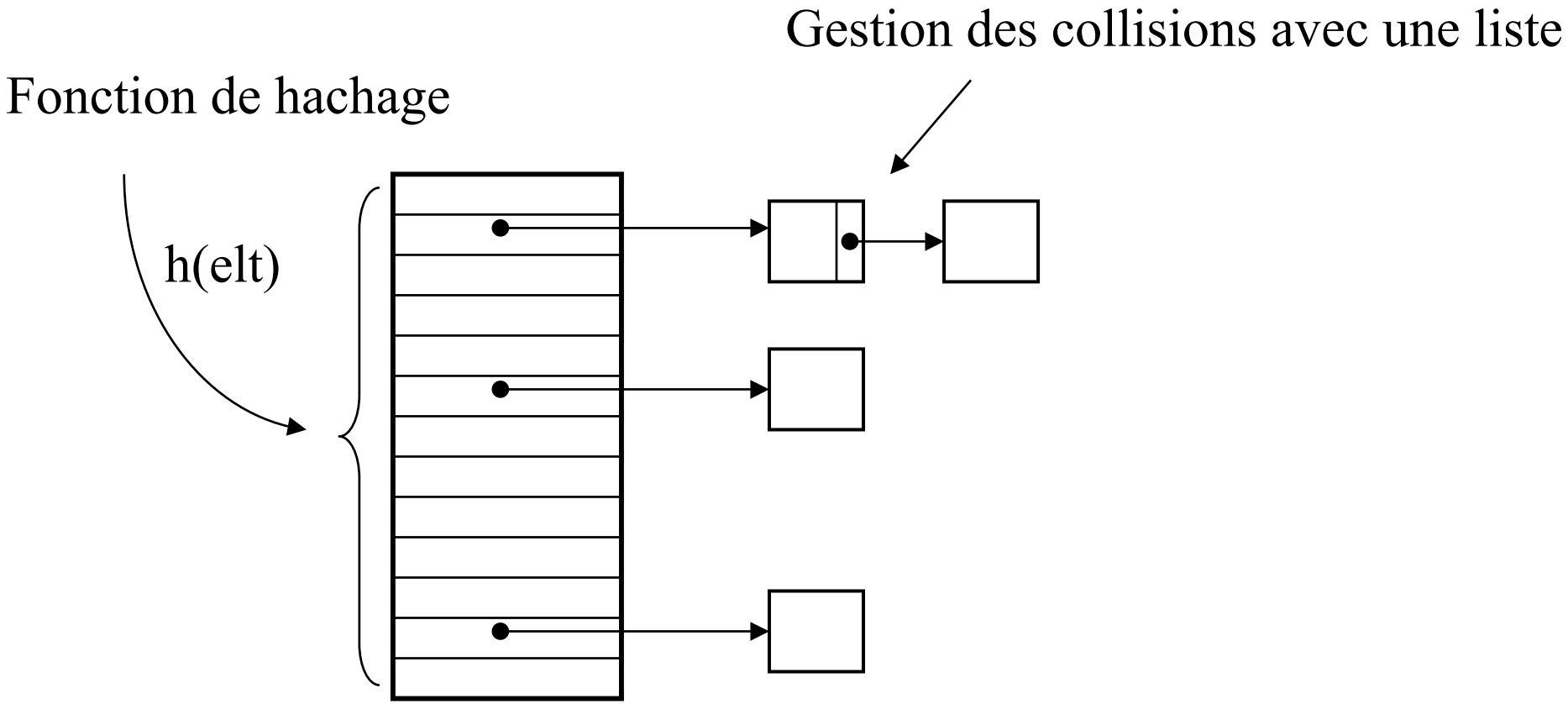
- **La 2ème interface Racine**
- **implémentée par les dictionnaires**
- **gestion de couples <Clé, Valeur>**
  la clé étant unique

```
interface Map<K,V>{

...
}
```

Adressage associatif, Hashtable



Une table de hachage



public class Hashtable<KEY,VALUE> ...

L ’interface Map<K,V>

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

   // Modification Operations
   V put(K key, V value);
   V remove(Object key);

    // Bulk Operations
   void putAll(Map<? extends K, ? extends V> t);
   void clear();

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

   // Comparison and hashing
    boolean equals(Object o);
    int hashCode();
```

L ’interface Map.Entry

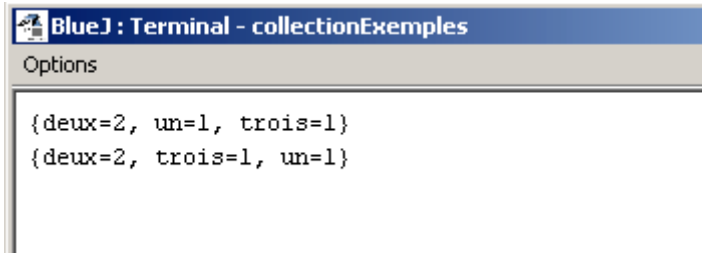
```
public interface Map<K,V>{

  // ...

interface Entry<K,V> {
 K getKey();

 V getValue();

 V setValue(V value);

 boolean equals(Object o);

 int hashCode();
}
```

Un exemple : fréquence des éléments d ’une liste

```
public Map<String,Integer> occurrence(Collection<String> c){
   Map<String,Integer> map = new HashMap<String,Integer>();
   Integer ONE = new Integer(1);

   for(String s : c){
      Integer occur = map.get(s);
      if (occur == null) {
         occur = ONE;
       }else{
           occur = new Integer(occur + 1);
       }
       map.put(s, occur);
    }
    return map;
 }
```

Un exemple : usage de occurrence

```
public void test(){
  List<String> l = new ArrayList<String>();
  l.add("un");l.add("deux");l.add("deux");l.add("trois");
  Map<String,Integer> map = occurrence(l);
  System.out.println(map);

  Map<String,Integer> sortedMap = new TreeMap<String,Integer>(map);
  System.out.println(sortedMap);
}
```



La suite

- **Interface SortedMap<K,V>**
- **TreeMap<K,V> implements SortedMap<K,V>**

  Relation d ’ordre sur les clés

- **et les classes**

  TreeMap

  WeakHashMap

  IdentityHashMap

  EnumHashMap

La classe Collections très utile

- **Class Collections {**

  // Read only : unmodifiable*Interface*

  static <T> Collection<T> unmodifiableCollection(Collection<? extends T> collection)

  static <T> List<T> unmodifiableList(List<? extends T list)

  …

  // Thread safe : synchronized*Interface*

  static <T> Collection<T> synchronizedCollection(Collection<T> collection)

  // Singleton

  singleton(T o)

  // Multiple copy

  // tri

  public static <T extends Comparable<? super T>> void sort(List<T> list)

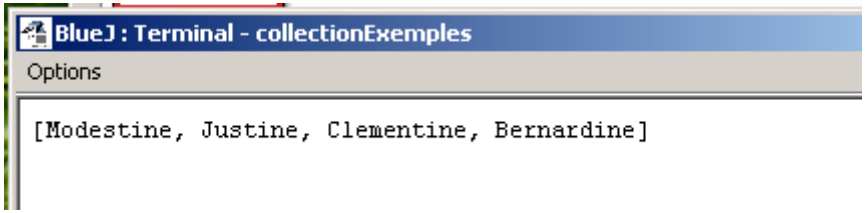
  public static <T> void sort(List<T> list, Comparator<? super T> c)

La méthode Collections.sort

```
 Object[] a = list.toArray();
 Arrays.sort(a, (Comparator)c);
 ListIterator i = list.listIterator();
 for (int j=0; j<a.length; j++) {
     i.next();
     i.set(a[j]);
 }
}
```

Un autre exemple d'utilisation

```
Comparator comparator = Collections.reverseOrder();
    Set reverseSet = new TreeSet(comparator);
    reverseSet.add("Bernardine");
    reverseSet.add("Justine");
   reverseSet.add("Clementine");
    reverseSet.add("Modestine");
    System.out.println(reverseSet);
```



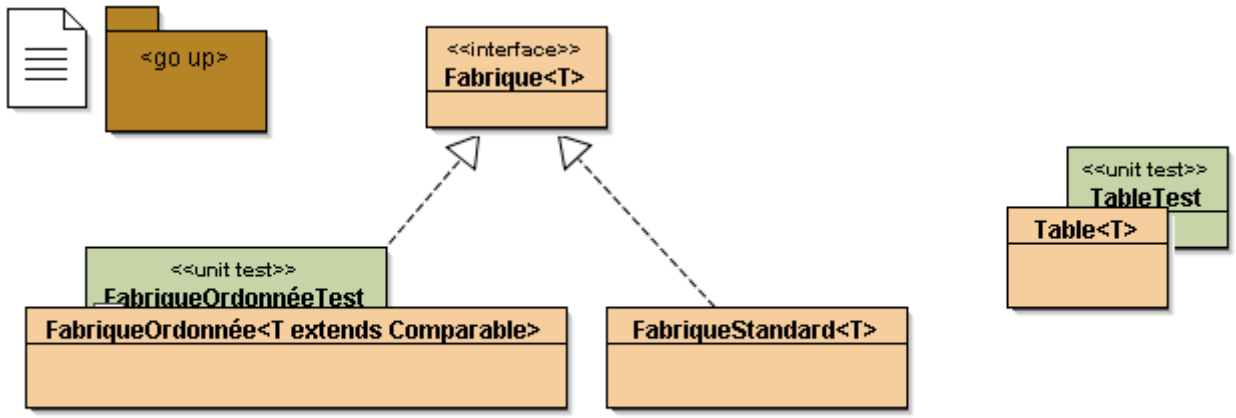
La classe Arrays

- **Rassemble des opérations sur les tableaux**

```
static void sort(int[] t);
…
static void sort(Object[] t, Comparator c)
boolean equals(int[] t, int[] t1);
…
int binarysearch(int[] t, int i);
…
static List asList(Object[] t);
…
```

Couplage faible …

- **le pattern fabrique : Choisir une implémentation par le client, à l ’exécution**
- **ici un ensemble qui implémente Set<T>**




```
import java.util.Set;
public interface Fabrique<T>{
  public Set<T> fabriquerUnEnsemble();
}
```

Le pattern Fabrique (1)

```
import java.util.TreeSet;
import java.util.Set;
public class FabriqueOrdonnée<T extends Comparable>
                    implements Fabrique<T>{

   public Set<T> fabriquerUnEnsemble(){
     return new TreeSet<T>();

   }
}
```

- **FabriqueOrdonnée : Une Fabrique dont les éléments possèdent une relation d'ordre**

Le pattern Fabrique (2)

```
import java.util.HashSet;
import java.util.Set;
public class FabriqueStandard<T> implements Fabrique<T>{
   public Set<T> fabriquerUnEnsemble(){
     return new HashSet<T>();
   }
}
```

```
import java.util.Set;
import cnam.tp.Ensemble;
public class MaFabrique<T> implements Fabrique<T>{
   public Set<T> fabriquerUnEnsemble(){
      return new Ensemble<T>();
   }
}
```

Le pattern Fabrique, le client : la classe Table

```
import java.util.Set;
public class Table<T>{
  private Set<T> set;

  public Table<T>(Fabrique<T> f){
      this.set = f.fabriquerUnEnsemble();
  }

  public void ajouter(T t){
    set.add(t);
  }
  public String toString(){
    return set.toString();
  }

  public boolean equals(Object o){
    if(! (o instanceof Table))
       throw new RuntimeException("mauvais usage de equals");
    return set.equals(((Table)o).set);
  }
}
```

Table, appel du constructeur

- **le pattern fabrique : Choisir une implémentation par le client, à l ’exécution**

- **Fabrique<String> fo = new FabriqueOrdonnée<String>());**
- **Table<String> t = new Table (fo);**

Ou bien

- **Table<String> t1 = new Table<String> (new FabriqueStandard<String>());**

Ou encore

- **Table<String> t2 = new Table <String> (new MaFabrique <String> ());**

Fabriquer une Discussion

- **Il reste à montrer que toutes ces fabriques fabriquent bien la même chose … ici un ensemble**

```
assertEquals("[a, f, w]", table2.toString());
assertEquals("[w, a, f]", table1.toString());
assertTrue(table1.equals(table2));
```

Conclusion

- **Lire, relire un tutoriel**
- **Utiliser Bluej**

  Outils puis Bibliothèque de classe
- **Les types primitifs sont-ils oubliés ?**

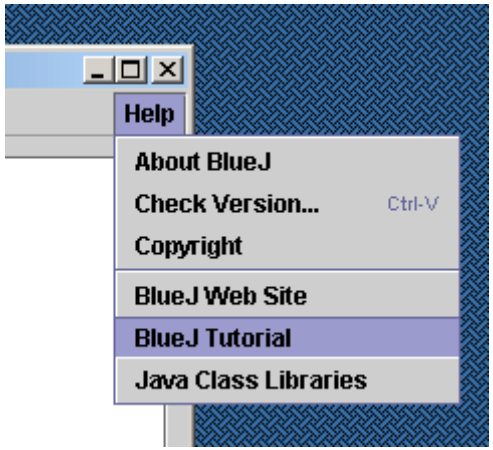
  http://pcj.sourceforge.net/ Primitive Collections for Java. De Søren Bak

  en 1.5 voir également l 'autoboxing
- **Ce support est une première approche**

  Collections : des utilitaires bien utiles

  Il en reste … WeakHashMap, … `java.util.concurrent`
- **…**

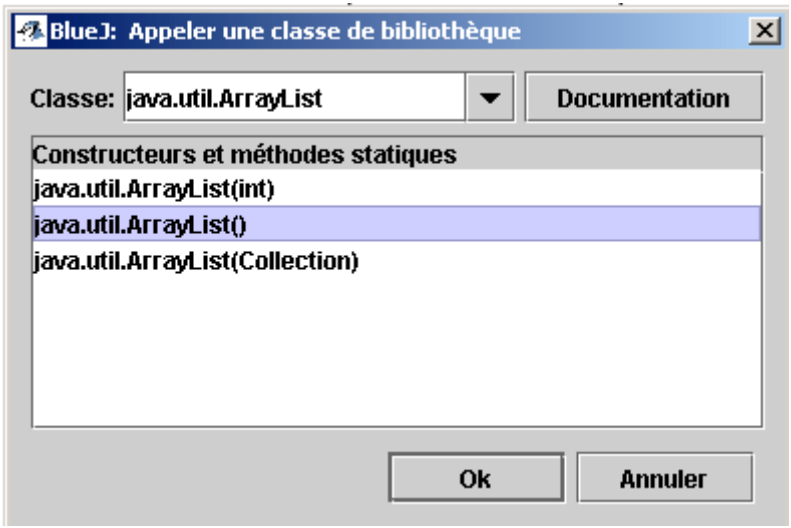
Documentation et tests unitaires

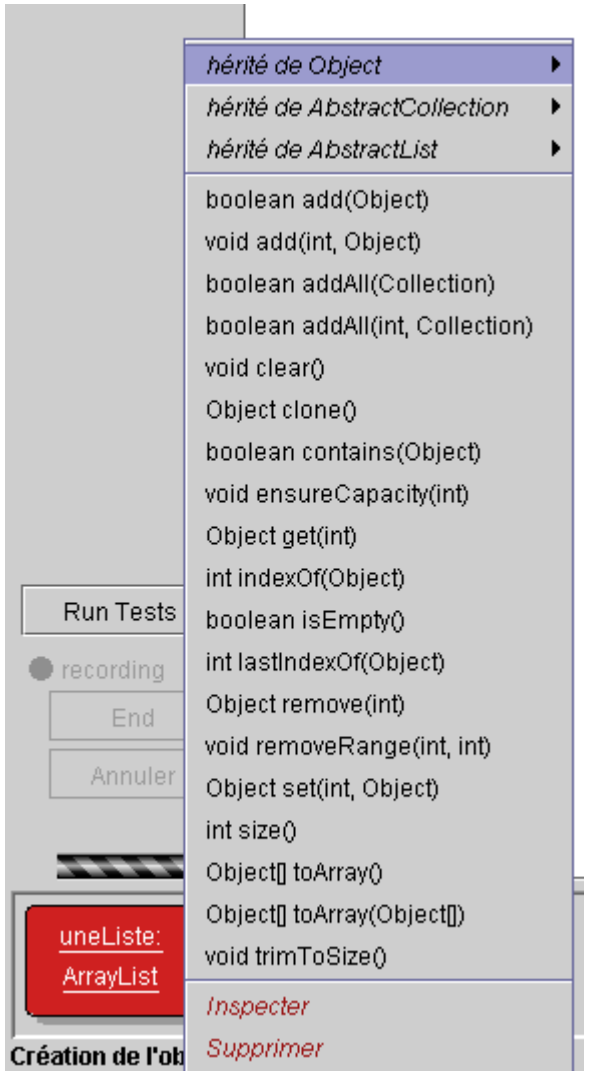


- **Documentation**
  - **Java API**
    - **item Java Class Libraries**
  - **Tutorial**
    - **item BlueJ Tutorial**

- **Tests unitaires**
  - **tutorial page 29, chapitre 9.6**

Approche BlueJ : test unitaire





De Tableaux en Collections

- **La classe java.util.Arrays, la méthode asList**

```
import java.util.Arrays;
……
public class CollectionDEntiers{
  private ArrayList<Integer> liste;
  …

  public void ajouter(Integer[] table){
    liste.addAll(Arrays.asList(table));
  }
```

De Collections en Tableaux

- **De la classe ArrayList**
- **public Object[] toArray(Object[] a)**

  Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array. If the collection fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this collection.

  String[] x = (String[]) v.toArray(new String[0]);

```
public Integer[] uneCopie(){
  return (Integer[])liste.toArray(new Integer[0]);
}
```

Itération et String (une collection de caractères ?)

- **La classe StringTokenizer**

```
String s = "un, deux; trois    quatre";

StringTokenizer st = new StringTokenizer(s);
while (st.hasMoreTokens())
  System.out.println(st.nextToken());
```

```
un,
deux;
trois
quatre
```

```
StringTokenizer st = new StringTokenizer(s, ", ;");
while (st.hasMoreTokens())
  System.out.println(st.nextToken());
```

```
un
deux
trois
quatre
```