NFP121, Cnam/Paris
Cours 5-1
les Collections

Patrons Template method, Iterator et Factory Method

jean-michel Douin, douin au cnam point fr
version : 24 Octobre 2017

Notes de cours

Sommaire pour les Patrons

- **Selon la classification habituelle**
  - **Créateurs**
    - **Abstract Factory, Builder, Factory Method Prototype Singleton**
  - **Structurels**
    - **Adapter Bridge Composite Decorator Facade Flyweight Proxy**
  - **Comportementaux**

    Chain of Responsability. Command Interpreter Iterator
    Mediator Memento Observer State
    Strategy Template Method Visitor

Les patrons déjà vus …

- **Adapter**
  - **Adapte l'interface d'une classe conforme aux souhaits du client**
- **Proxy**
  - **Fournit un mandataire au client afin de contrôler/vérifier ses accès**
- **Observer**
  - **Notification d'un changement d'état d'une instance aux observateurs inscrits**
- **Strategy**
  - **Une stratégie concrète est choisie à la volée en fonction du contexte**

Sommaire pour les collections (1/2)

- **Pourquoi ? Quels objectifs ?**

- **Quelles interfaces ?**
- *Quelles implémentations, mêmes incomplètes*
- **Quelles implémentations concrètes, toutes prêtes**

- **Quels utilitaires ?**

Sommaire pour les collections (2/2)

- **Interface Collection<E>, Iterable<E> et Iterator<E>**
- *Classe AbstractCollection<E>*

- **Interface Set<E>,SortedSet<E> et List<E>**
- *Classes AbstractList<E> et AbstractSet<E>*

- **Les concrètes Vector<E>, Stack<E>, ArrayList<E>, TreeSet<E> ..**

- **Interface Map<K,V> et Map.Entry<K,V>**
- *Classe AbstractMap<K,V>*

- **Les concrètes HashMap<K,V>, TreeMap<K,V> ..**

- **Classes Collections et Arrays**
- **Le patron Fabrique<T>**

Principale bibliographie

- **Introduction to the Collections Framework**
  - **http://docs.oracle.com/javase/tutorial/collections/intro/**

- 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

Les collections, 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)**

- ***Un héritage des STL (Standard Template Library) de 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.

- http://docs.oracle.com/javase/tutorial/collections/intro/

<G> comme généricité

- **Une collection d ’objets**

- **Si l’on souhaite 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)**
    - **une inférence de types est effectuée à la compilation**
    - **Des contraintes sur l’arbre d’héritage peuvent être précisées**

- **sinon tout est Object ...**
  - **Une collection d’Objects**
    - **Une collection hétérogène**

Généricité / Généralités

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

```
Liste <Integer> l = new Liste <Integer>();
Integer i = l.first(); <-- vérification statique : ok
```

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

Généricité,
afin de pouvoir lire la documentation d'Oracle

- **java.util.Collection<?>**
  - **? comme Classe Inconnue « compatible avec n'importe quelle classe »**
    - **Avec des contraintes d'utilisation…**

- **Exemple :**

```
public static void afficher(java.util.Collection<?> c){
    Iterator<Object> it = c.iterator();
    while(it.hasNext()){
      System.out.println(it.next());
    }
  }
```

Généricité,
afin de pouvoir lire la documentation d'Oracle

- **Contraintes sur l'arbre d'héritage**

- **<? extends E>**
  - **E doit être une super classe de la classe inconnue**
- **<? super E>**
  - **E doit être une sous classe de la classe inconnue**
- **<? extends Comparable<E>>**
  - **La classe doit implémenter l'interface Comparable**
- **<? extends Comparable<? super E>>**
  - **Une des super classes de la classe « inconnue » doit implémenter l'interface Comparable**
- **<? extends Comparable<E> & Serializable>**
  - **Une des …. Doit implémenter l'interface Comparable<E> et l'interface Serializable**
- **public <T> T[] toArray(T[] a)**
  - **T est un paramètre générique de la méthode**

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)); ***← erreur de compilation***
  - l.add(new Integer(3));

- **Mixité**
  - List l = new ArrayList<Integer>();
  - l.add(new Boolean(true)); ***← un message d'alerte et compilation effectuée***
  - l.add(new Integer(3));

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>, Queue<E>, SortedMap<K,V>, Comparator<E>, Comparable<E>

- ***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 (dites toutes prêtes) ?**
  - **LinkedList**<E>, **ArrayList**<E>, PriorityQueue<E>, …
  - **TreeSet**<E>, **HashSet**<E>,
  - WeakHashMap<K,V>, HashMap<K,V>, TreeMap<K,V>, …

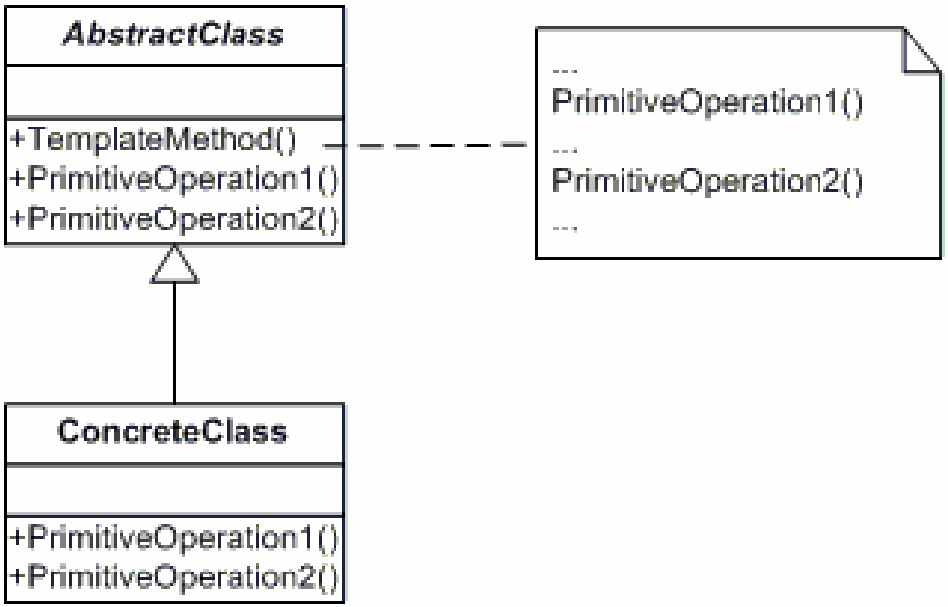
- ***Quelles passerelles ?***
  - Collections et Arrays

Les fondations : deux patrons avant tout

- **Template Method**
  - **Laisser la réalisation de certaines méthodes aux sous-classes**
  - Largement utilisé :
    - AbstractCollection<E>, AbstractSet<E>, AbstractList<E>, AbstractSequentialList<E>, AbstractMap<K,V> …

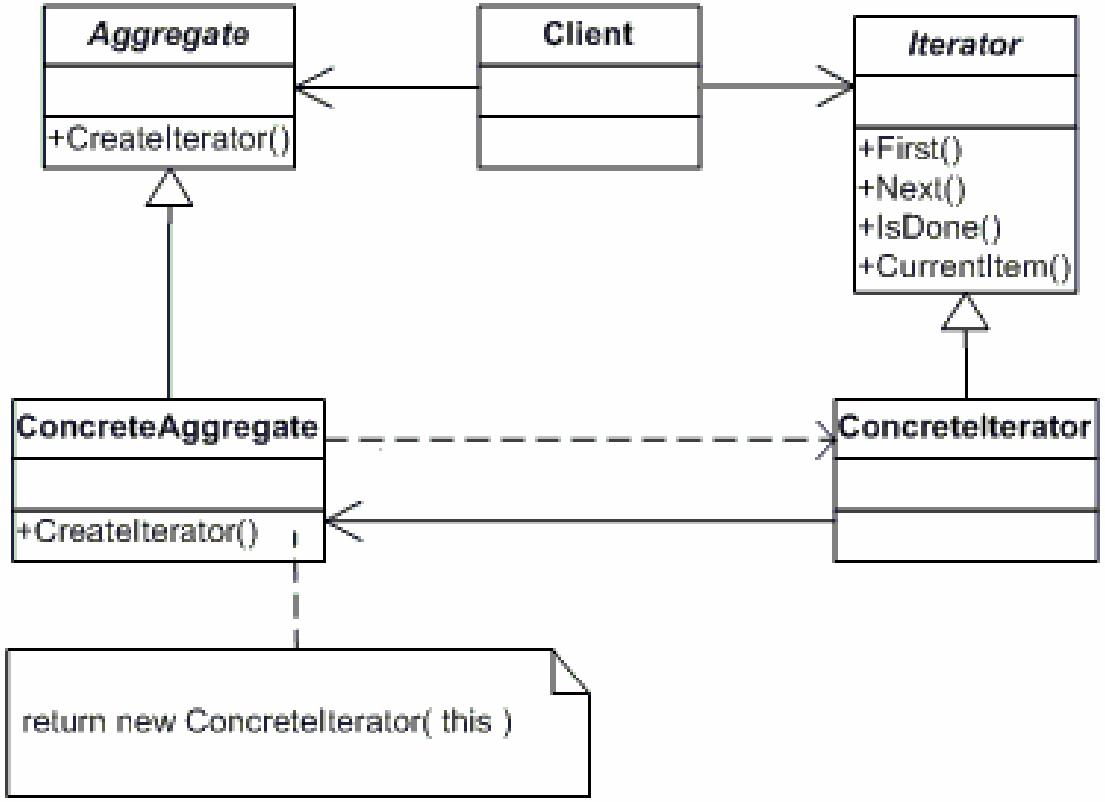
- **Iterator**
  - **Parcourir une collection sans se soucier de son implémentation**
    - Chaque collection est « Iterable<E> » et propose donc un itérateur

Template Method



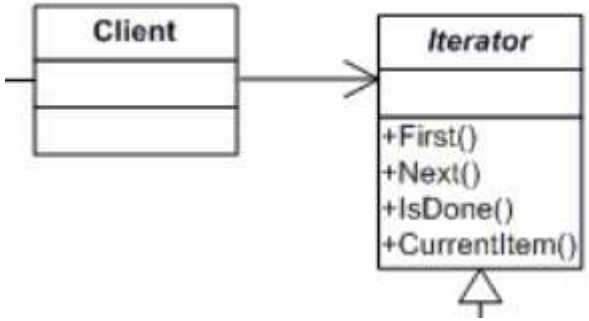
- **Laisser aux sous-classes de grandes initiatives …**
  - **Bonne idée …**

Iterator



- **Ou comment parcourir une structure quelle que soit son implémentation**

java.util.Iterator<E>



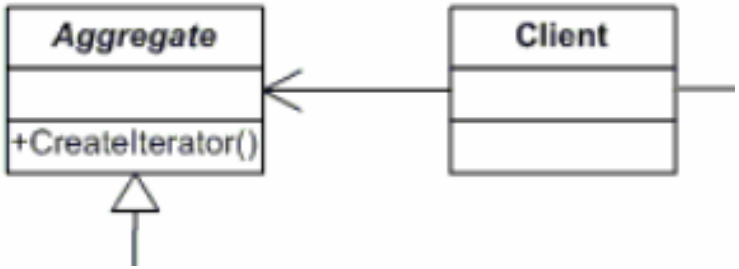

```
public interface Iterator<E>{
  E next();
  boolean hasNext();
  void remove();
}
```

paquetage java.util

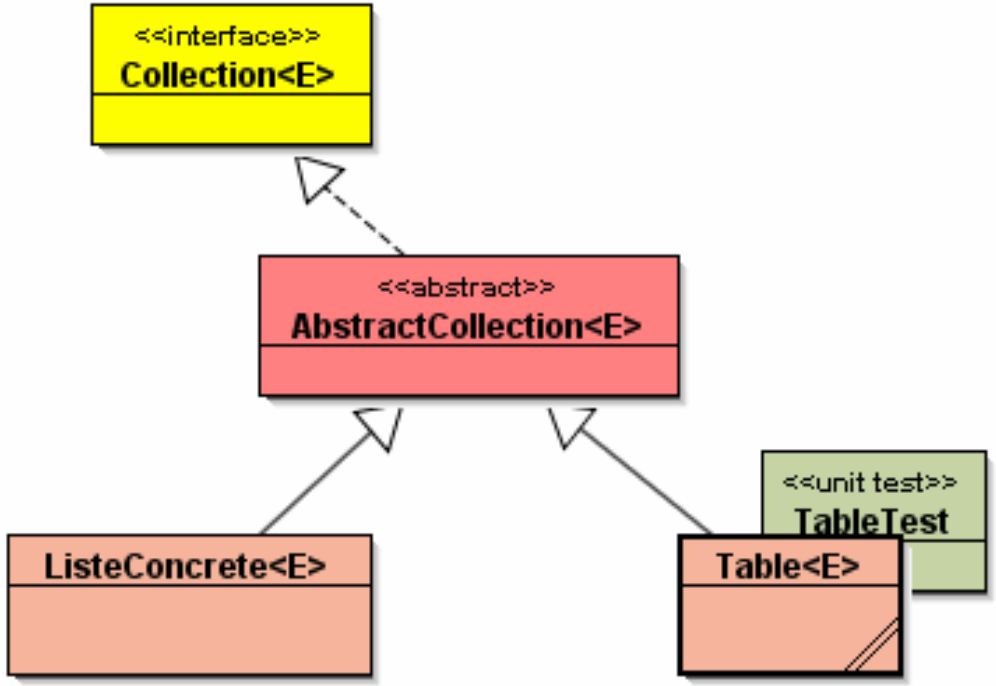
java.util.Iterator<E>

```
public interface Collection<E>{
  //…
  Iterator<E> iterator();

  // add, remove, contains…
}
```

paquetage java.util



Template Method + Iterator : un exemple

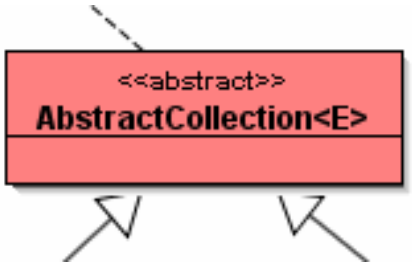


- **Collection :** une interface
- **AbstractCollection,** une implémentation partielle **:**
  - **Avec plusieurs méthodes concrètes : taille, retirer, contient,…**
  - **Et deux méthodes abstraites : iterator, ajouter**
- **ListeConcrète, Table :** deux implémentations complètes

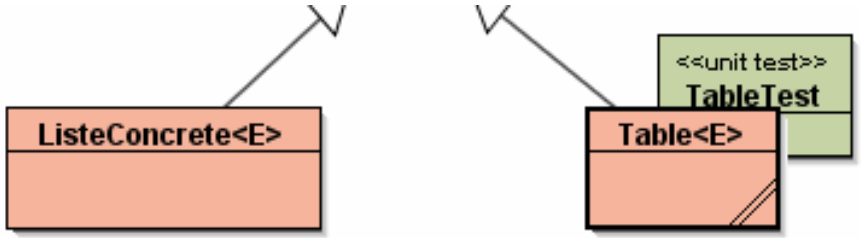
Un exemple … une démonstration

- **Une interface Collection**

- **Une classe abstraite AbstractCollection**
  - **Usage du patron TemplateMethod + Iterator**

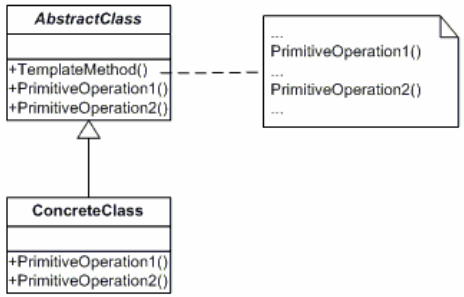


- **Plusieurs classes concrètes**

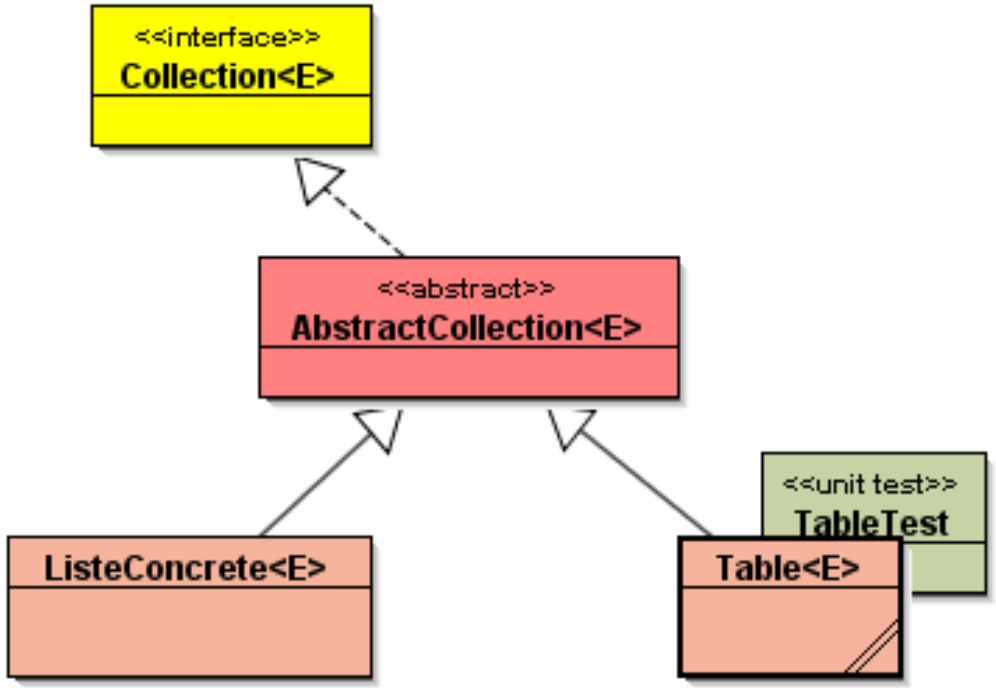


Template Method : Démonstration

```
public interface  Collection<E> {
     boolean ajouter(E e);
     boolean ajouter(Collection<E> e);
     int  taille();
     boolean retirer(E e);
     Iterator<E> iterator();
}

public abstract AbstractCollection<E> implements Collection<E>{
     public int taille(){
       // complète
     }
     public boolean ajouter(Collection<E> e);
      // complète
     }
     public boolean retirer(E e){
       // complète
     }
     public abstract boolean ajouter(E e);    // cf. template Method
     public abstract Iterator<E> iterator();
}

public ListeConcrete<E> extends AbstractCollection<E> {
     public boolean ajouter(E e){complète}
     public Iterator<E> iterator(){complète}
}
```

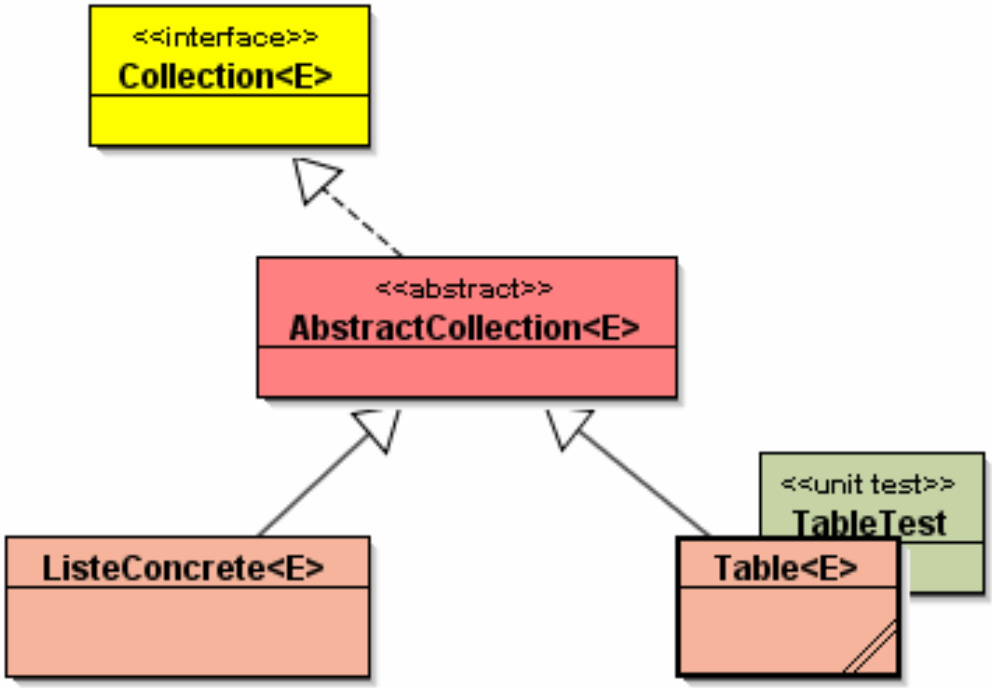


Démonstration



– Le projet au complet ici
http://jfod.cnam.fr/NFP121/cours5_collections_exemples.jar

Iterator : un exemple



- **Table<E>**
  - **Avec une table bornée comme contenu**
  - **Seules deux méthodes seront à implémenter :** ajouter et iterator

  - Le projet au complet ici http://jfod.cnam.fr/NFP121/cours5_collections_exemples.jar

Iterator : Démonstration

```
public class Table<E> extends AbstractCollection<E> {
  private final int TAILLE_MAX;
  private Object[] contenu;
  private int courant;

  public Table(int capacite){
    this.TAILLE_MAX = capacite;
    this.contenu = new Object[TAILLE_MAX];
    this.courant = 0;
  }

  public boolean ajouter(E e){       // Seules les méthodes ajouter(E e) et iterator()
    if(courant <= TAILLE_MAX){                                        // sont à écrire
      contenu[courant] =  e;
      courant++;
      return true;
    }
    return false;
  }


 public Iterator<E> iterator(){
    return new TableIterator<E>();       // classe interne, page suivante
  }
```

Iterator : Démonstration

```
// une classe interne et membre, accès aux champs de l'instance englobante

private class TableIterator<E> implements Iterator<E>{
     // partiel : il manque le traitement des cas d'erreur, voir page suivante
  private int index = 0;

  public boolean hasNext(){
    return index < courant;
  }

  public E next(){
    Object obj = contenu[index];
    index++;
    return (E)obj;
  }

  public void remove(){
   // supprimer le courant, soit contenu[index-1]
      if(index<courant)
        System.arraycopy(contenu,index,contenu,index-1, courant-1);
      courant--;
      index--;
  }
}
```

Iterator : Démonstration

Précaution prise : **Tout appel de remove doit être précédé d'au moins un appel de next … Une exception si l'élément n'existe pas, mais il manque une exception lorsque nous avons deux modifications en même temps de la même collection avec deux itérateurs différents…**

```
private class TableIterator<E> implements Iterator<E>{  //
    private int index = 0;
    private boolean nextOk; // next() a été effectué au moins une fois avant remove

    public boolean hasNext(){
      return index < courant;
    }

    public E next(){
      if(index>=courant)throw new NoSuchElementException();
      Object obj = contenu[index];
      index++;
      nextOk = true;
      return (E)obj;
    }

    public void remove(){
      if(nextOk){ // supprimer le courant, soit contenu[index-1]
        nextOk = false;
        if(index<courant)
          System.arraycopy(contenu,index,contenu,index-1, courant-1);
        courant--;
        index--;
      }else
        throw new IllegalStateException();
    }
  }
```

Sommaire suite

Deux interfaces

- **java.util.Collection<E>**

- **java.util.Map<K,V>**

Les Collections en Java : deux interfaces

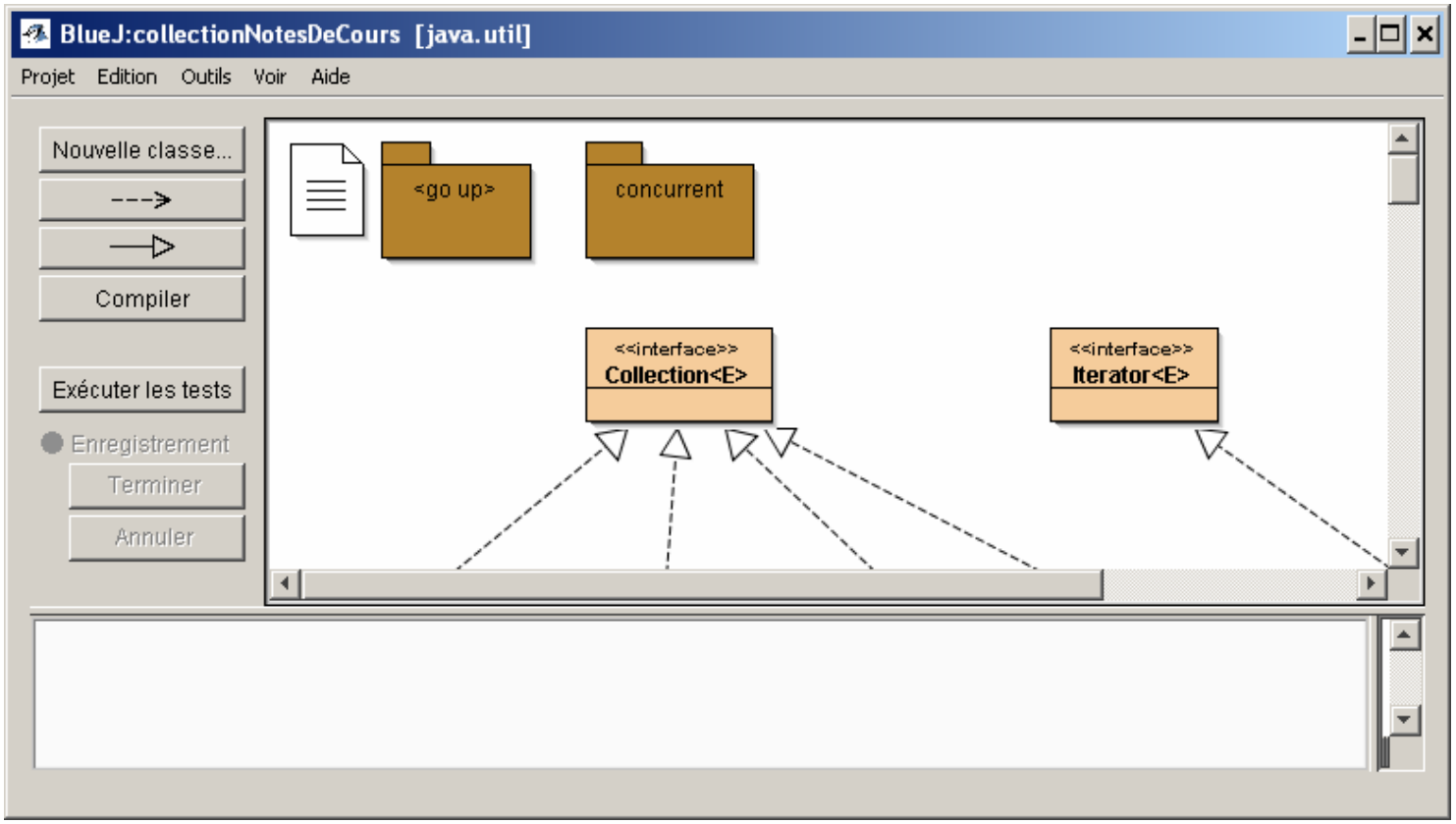
- **interface Collection<T>**
  - **Pour les listes,les ensembles, les files, les piles…**

  – **package java.util;**
  – **public interface Collection<E> extends Iterable<E>{**
    - **…**

- **interface Map<K,V>**
  - **Pour les dictionnaires,**
    – **gestion de couples <Key, Value>**

  – **package java.util;**
  – **public interface Map<K,V> {**
    - **public interface Entry<K,V>{**
      – **…**

interface java.util.Collection<E> extends Iterable<E>



- **Une interface pour toutes les Collections (au sens large)**

interface java.util.Collection<E>

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

```
// interrogation
  int size();
  boolean isEmpty();
  boolean containsAll(Collection<?> c);
  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 addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();


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

java.lang.Iterable<T>

Cf. Le patron Iterator

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

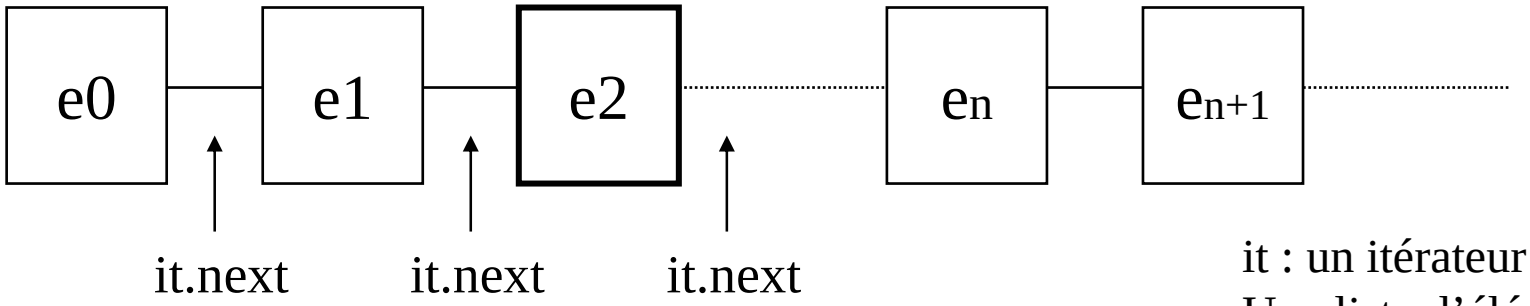
- **Avec**

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

Iterator<E> ? : le parcours d'une collection

java.util.Iterator<E>

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



it : un itérateur concret
Une liste d'élément

java.util.Iterator<E> : un usage

```
public static <T> void filtrer(   Iterable<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);
}
```

Démonstration

- **Discussion**

boucle foreach (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 qui implémente Iterable, (ou un tableau…)

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()
```

Parcours dans l'ordre ? Mais lequel ?

- **Extrait de la documentation pour next :**

  Returns the next element in the iteration.

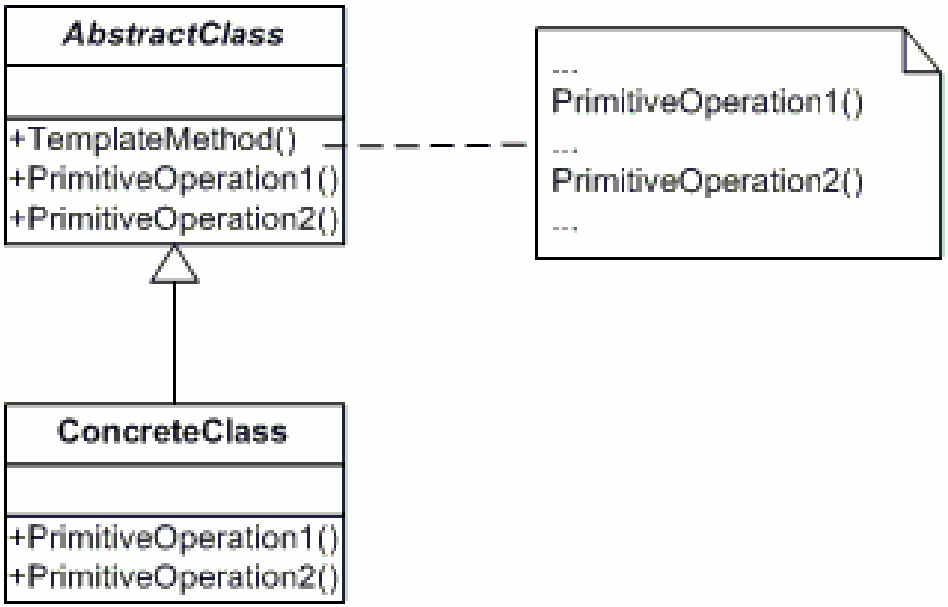
- **Une liste, une table**
- **Un arbre binaire ?**
  - **Prefixé, infixé … ?**
- **Un dictionnaire ?**
- **… etc**

Lire la documentation …

AbstractCollection<E>

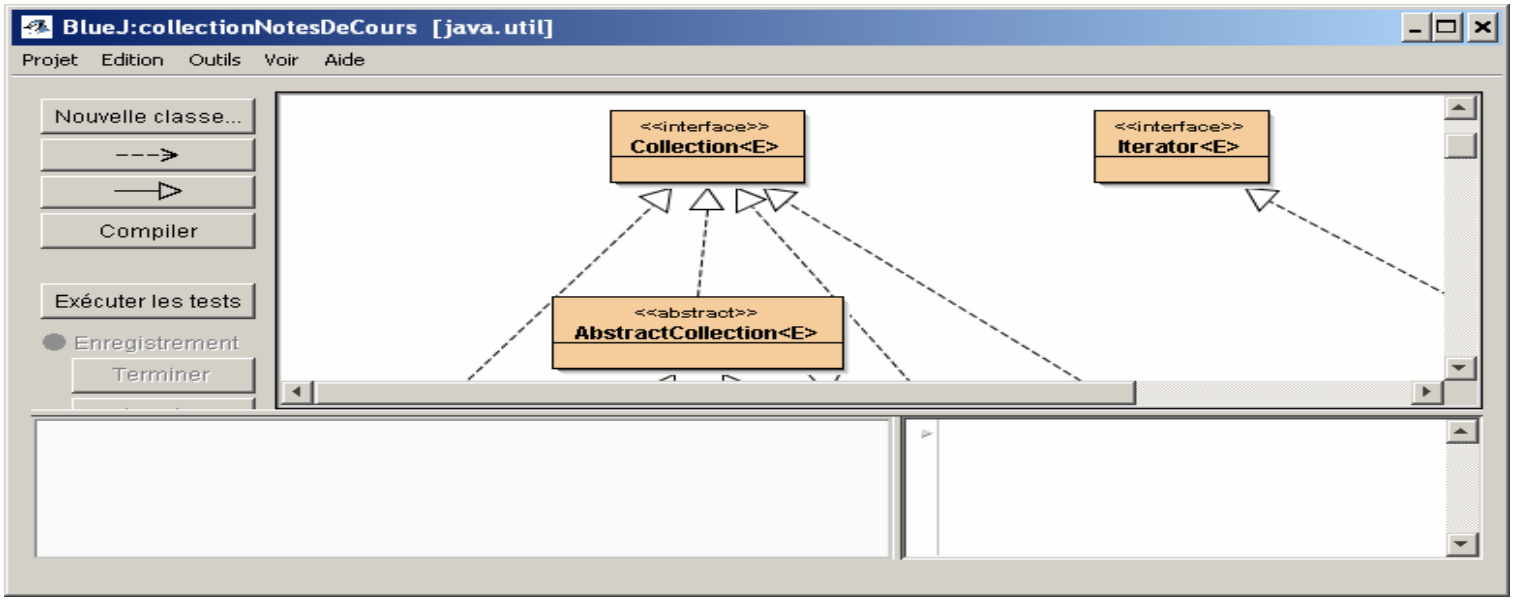
- **AbstractCollection<E> implements Collection<E>**
  - Implémentations effectives de 13 méthodes sur 15 !
  - Usage du patron Template Method

Template Method



- **Laisser aux sous-classes certaines implémentations …**

Première implémentation incomplète de Collection<E>



- **La classe incomplète : AbstractCollection<E>**
  - **Seules les méthodes :**

  - **boolean add(E obj);**
  - **Iterator<E> iterator();**

  - **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) ...
```

Note sur l'usage de <?>, compatible avec Object … contains ? page suivante

AbstractCollection : la méthode contains

```
public boolean contains(Object o) {
  Iterator<E> e = iterator();
  if (o==null) {                    // si l'élément est égal à « null »
     while (e.hasNext())
       if (e.next()==null)          // alors usage de ==
          return true;
   } else {
     while (e.hasNext())
       if (o.equals(e.next()))      // sinon usage de equals
          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, à redéfinir donc

```
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())) {          // appel de contains, déjà vu
       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())) {                // = equals
          e.remove();
          return true;
       }
  }
  return false;
  }
```

Encore une : la méthode retainAll

c.retainAll(c1),

ne conserve que les éléments de c1 également présents dans la collection c

```
Collection<Integer> c = new ArrayList<Integer>();
c.add(1);c.add(3);c.add(4);c.add(7);

Collection<Integer> c1 = new Vector<Integer>();
c1.add(3);c1.add(2);c1.add(4);c1.add(8);

asserTrue(c.retainAll(c1));
asserEquals(c.toString(), "[3, 4]");
```

En exercice …

AbstractCollection : 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;
 }
```

javadoc

- http://docs.oracle.com/javase/6/docs/api/java/util/package-summary.html

- **Iterator<E>**
- **Collection<E>**
- **AbstractCollection<E>**

Patron Template Method

- **Discussion**

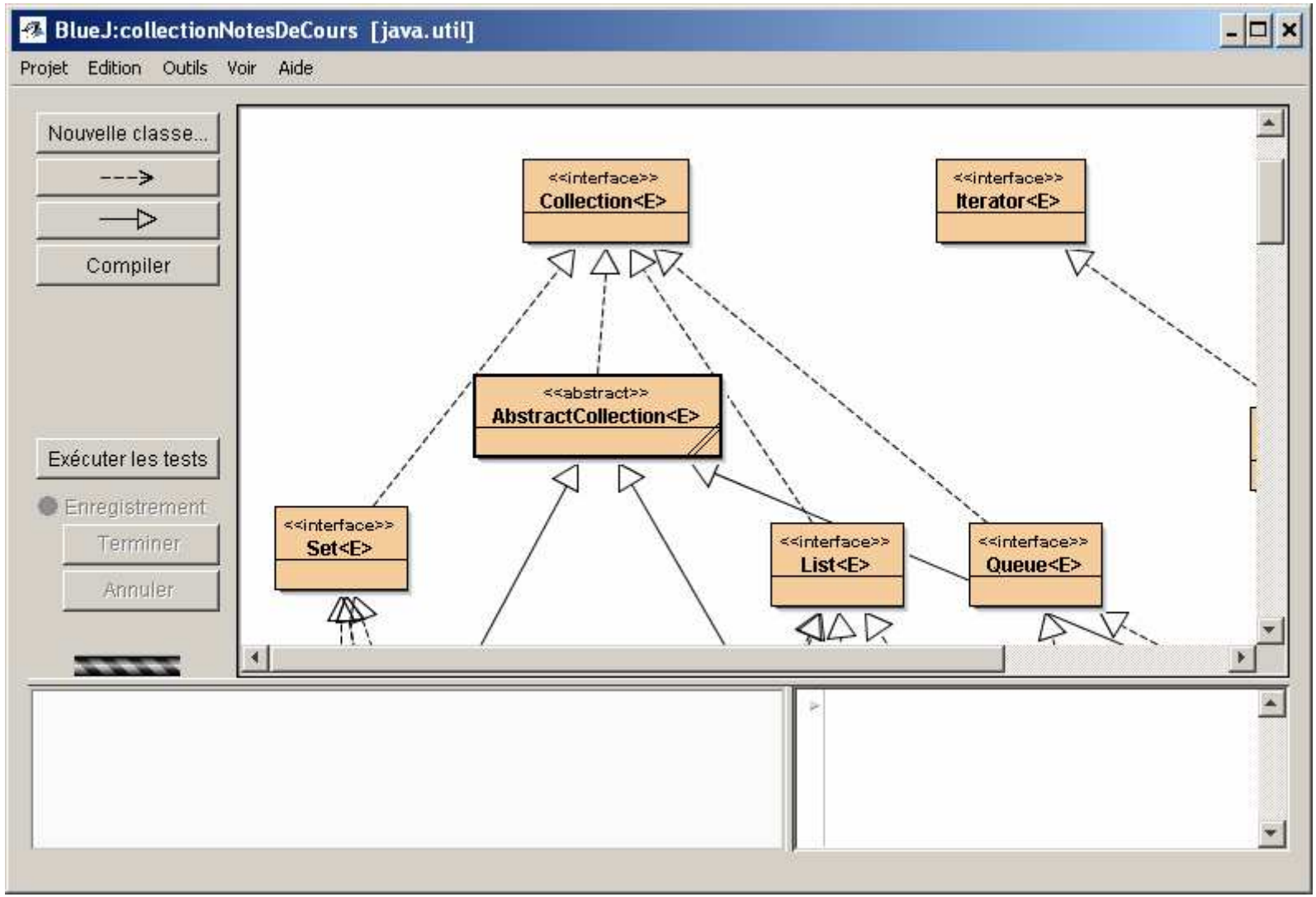
  – **Grand usage de ce patron dans l'implémentation des Collections en Java**

  Une implémentation concrète devra proposer (au moins) ces deux méthodes

  – **add // ajout**

  – **iterator() // retrait et parcours effectifs**

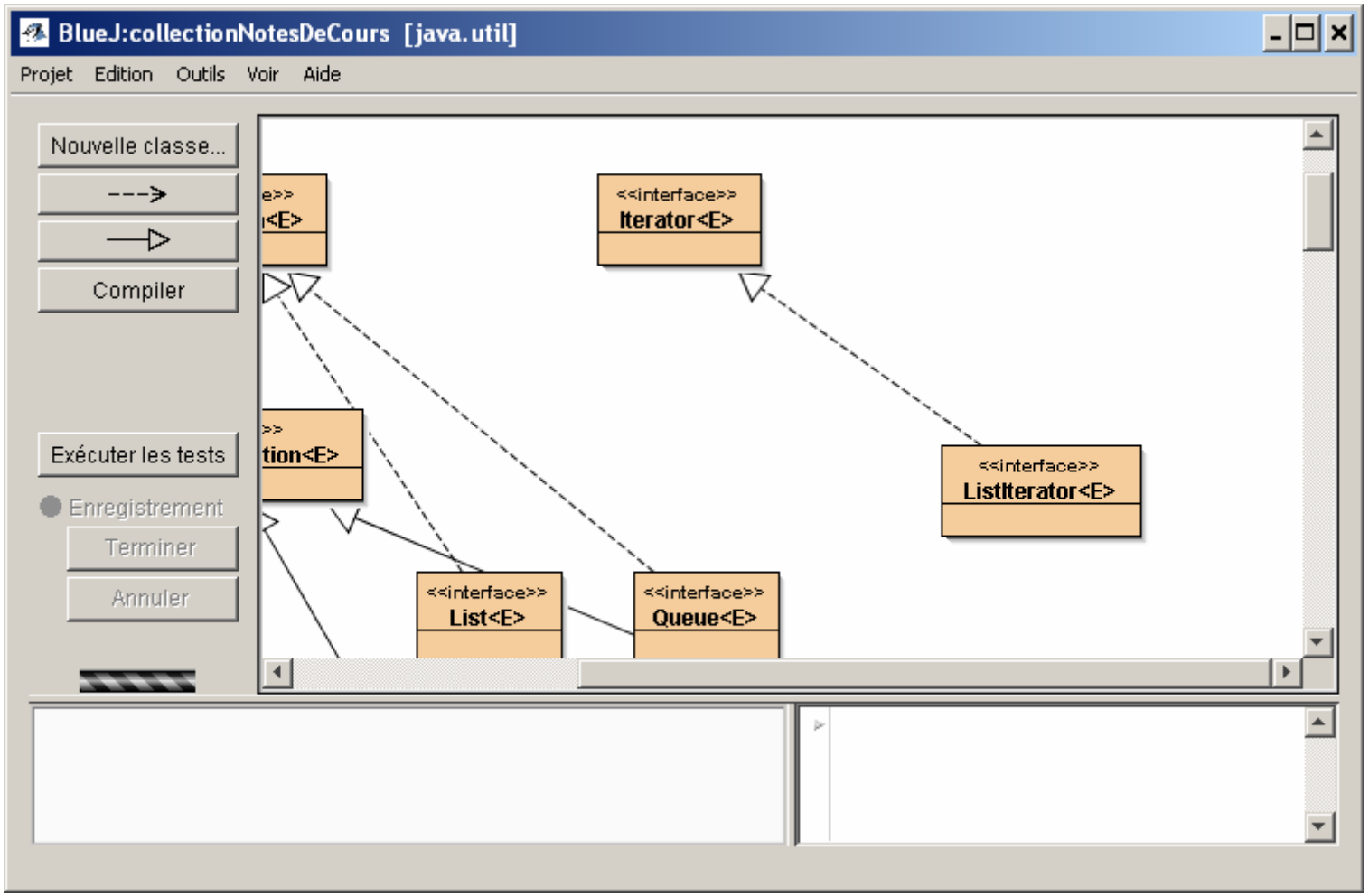
Interfaces List<E>, Set<E> et Queue<E>



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)
}
```

Iterator<E> extends ListIterator<E>

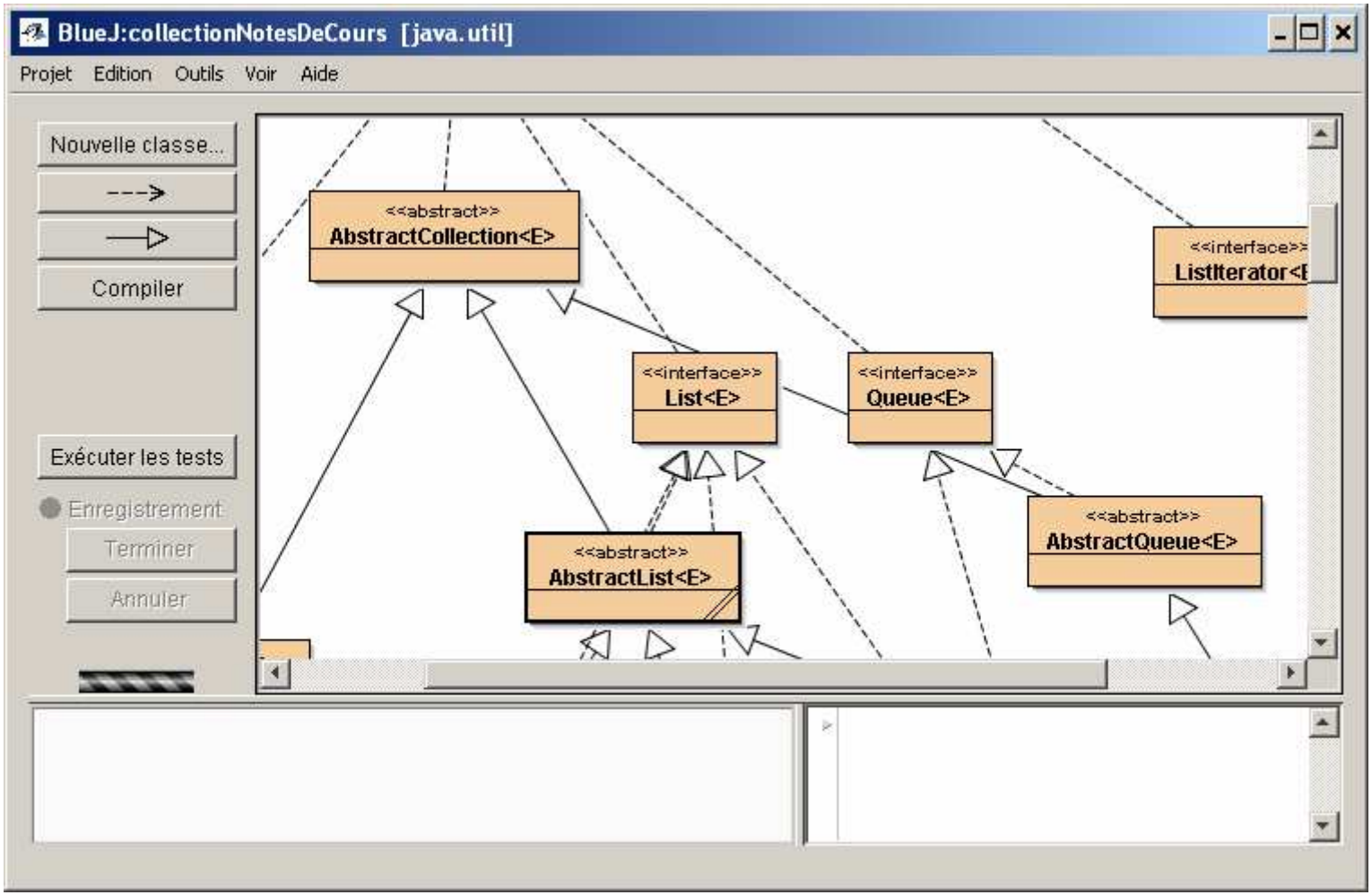


- **Parcours dans les 2 sens de la liste**
  - **next et previous**
  - **Méthode d'écriture : set(Object element)**

ListIterator<E>

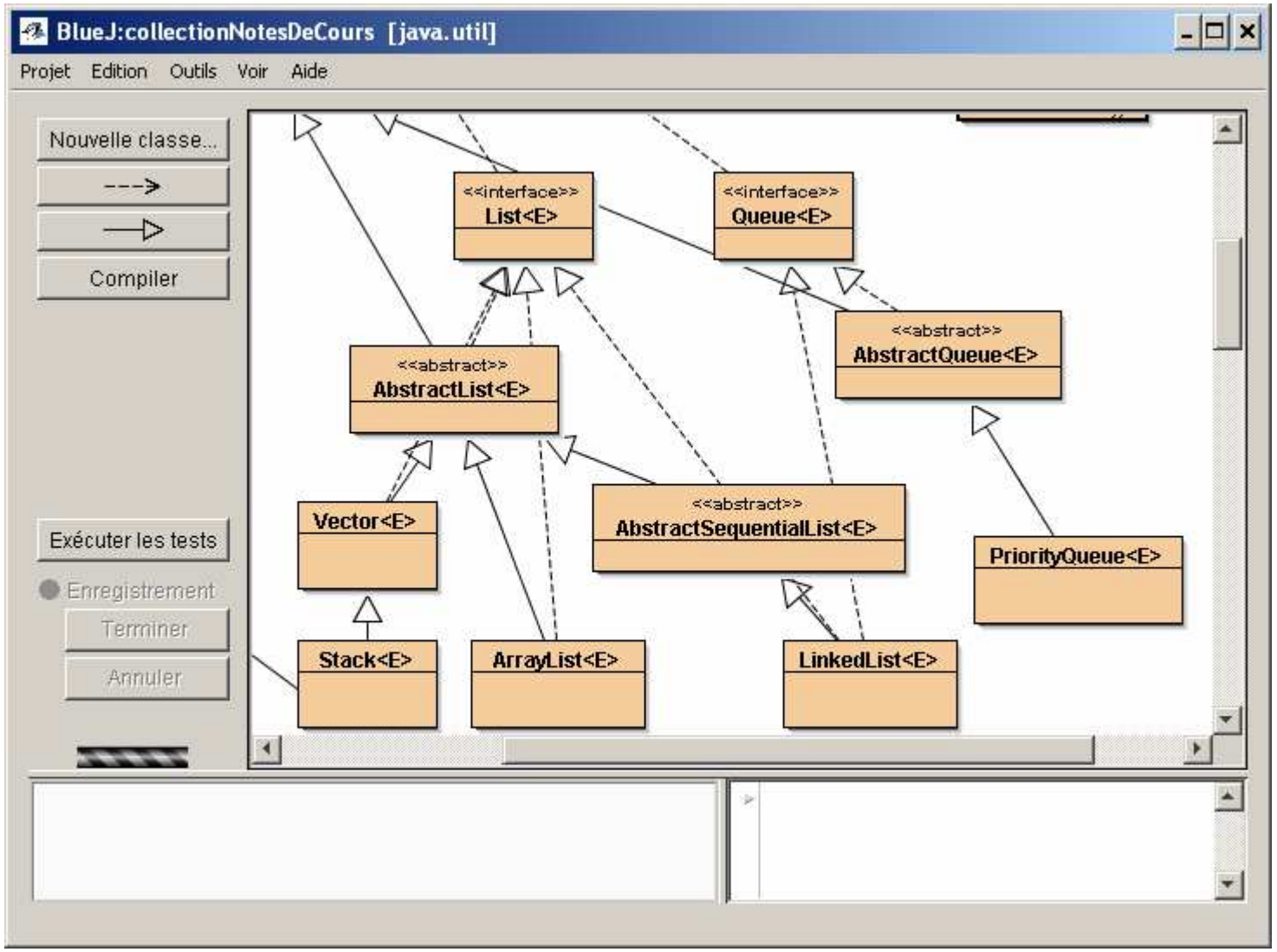
```
public interface ListIterator<E> extends Iterator<E>{
   E next();
   boolean hasNext();

   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**

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>

Ensemble<T> en tp ?

...

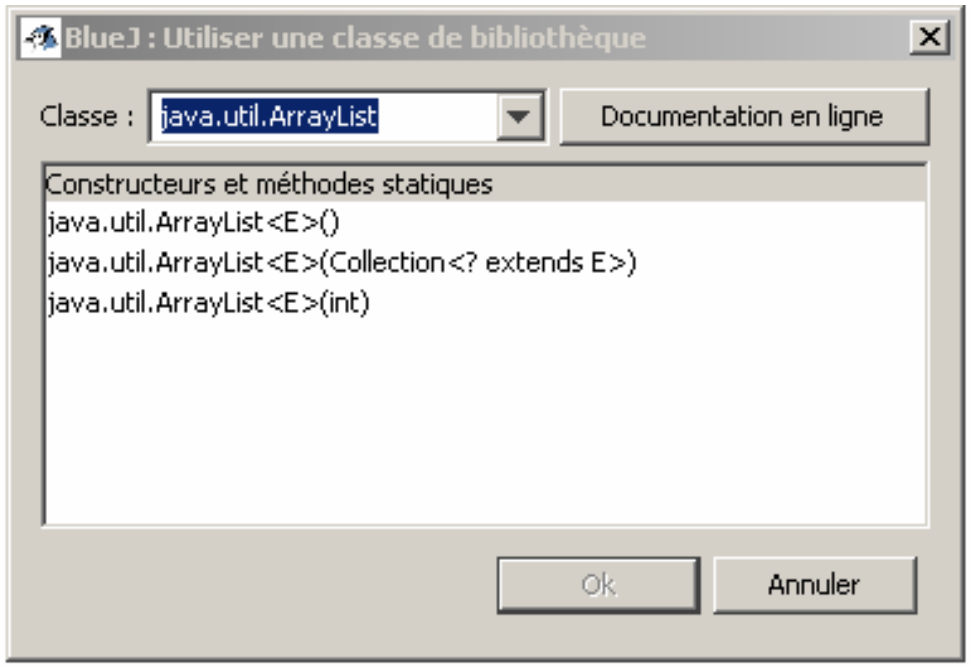
ArrayList, LinkedList :
enfin un exemple concret

```
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]
```

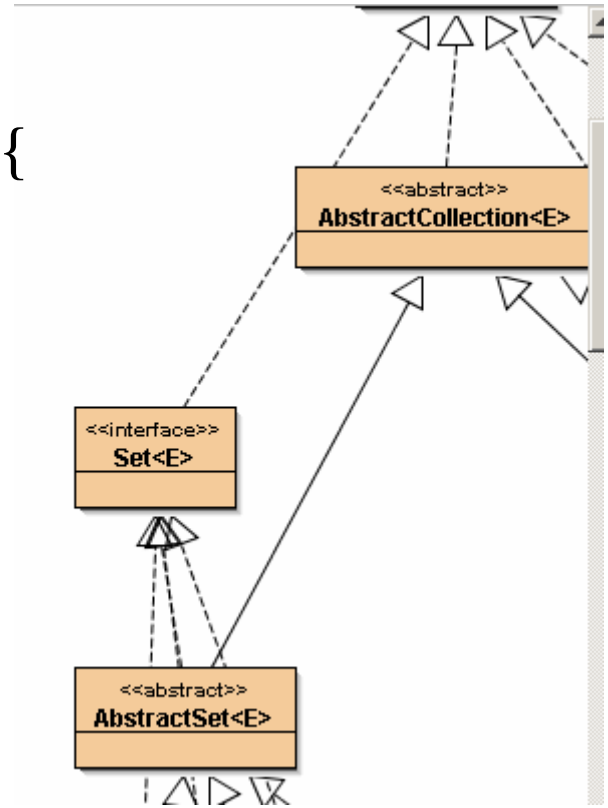
Démonstration

- **Bluej**
  - **Outils/ Utiliser une classe de la bibliothèque**



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

Relation d'ordre

- **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 NavigableSet<E>, Cloneable, Serializable
```

A NavigableSet implementation based on a TreeMap. The elements are ordered using their natural ordering,

public class **TreeSet<E>** extends AbstractSet<E> implements NavigableSet<E> ..

avec interface NavigableSet<E> extends SortedSet<E>

Attention ici la définition de TreeSet devrait être

TreeSet<E extends Comparable<E>>

ou **TreeSet<E extends Comparable<? super E>>**

```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

This class implements the Set interface, backed by a hash table (actually a HashMap instance).

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);        // Relation d'ordre ?
    System.out.println(sortedSet);
  }
}
```

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

Comparable concrètement

java.lang

Interface Comparable<T>

Type Parameters:

T - the type of objects that this object may be compared to

All Known Subinterfaces:

Delayed, Name, RunnableScheduledFuture<V>, ScheduledFuture<V>

All Known Implementing Classes:

Authenticator.RequestorType, BigDecimal, BigInteger, Boolean, Byte, ByteBuffer, Calendar, Character, CharBuffer, Charset, ClientInfoStatus, CollationKey, Component.BaselineResizeBehavior, CompositeName, CompoundName, Date, Date, Desktop.Action, Diagnostic.Kind, Dialog.ModalExclusionType, Dialog.ModalityType, Double, DoubleBuffer, DropMode, ElementKind, ElementType, Enum, File, Float, FloatBuffer, Formatter.BigDecimalLayoutForm, FormSubmitEvent.MethodType, GregorianCalendar, GroupLayout.Alignment, IntBuffer, Integer, JavaFileObject.Kind, JTable.PrintMode, KeyRep.Type, LayoutStyle.ComponentPlacement, LdapName, Long, LongBuffer, MappedByteBuffer, MemoryType, MessageContext.Scope, Modifier, MultipleGradientPaint.ColorSpaceType, MultipleGradientPaint.CycleMethod, NestingKind, Normalizer.Form, ObjectName, ObjectStreamField, Proxy.Type, Rdn, Resource.AuthenticationType, RetentionPolicy, RoundingMode, RowFilter.ComparisonType, RowIdLifetime, RowSorterEvent.Type, Service.Mode, Short, ShortBuffer, SOAPBinding.ParameterStyle, SOAPBinding.Style, SOAPBinding.Use, SortOrder, SourceVersion, SSLEngineResult.HandshakeStatus, SSLEngineResult.Status, StandardLocation, String, SwingWorker.StateValue, Thread.State, Time, Timestamp, TimeUnit, TrayIcon.MessageType, TypeKind, URI, UUID, WebParam.Mode, XmlAccessOrder, XmlAccessType, XmlNsForm

- **String est bien là …**

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 hashCode(){ return new Integer(i).hashCode();}
  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);
  }
}
```

OrdreDesCroissants ici https://www.youtube.com/watch?v=xily6acWFoQ

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]

Lecture … rappels c.f. début de cours

- **Mais que veut dire :**
  - **public class ListeOrdonnée<E extends Comparable<E>>**

  Et

  - **public class ListeOrdonnée<E extends Comparable<? super E>>**
  - *C'est le cours de la semaine prochaine…*

Démonstration/discussion

l'interface Queue<E>

java.util

Interface Queue<E>

Type Parameters:

E - the type of elements held in this collection

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Subinterfaces:

BlockingDeque<E>, BlockingQueue<E>, Deque<E>

All Known Implementing Classes:

AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, PriorityBlockingQueue, PriorityQueue, SynchronousQueue

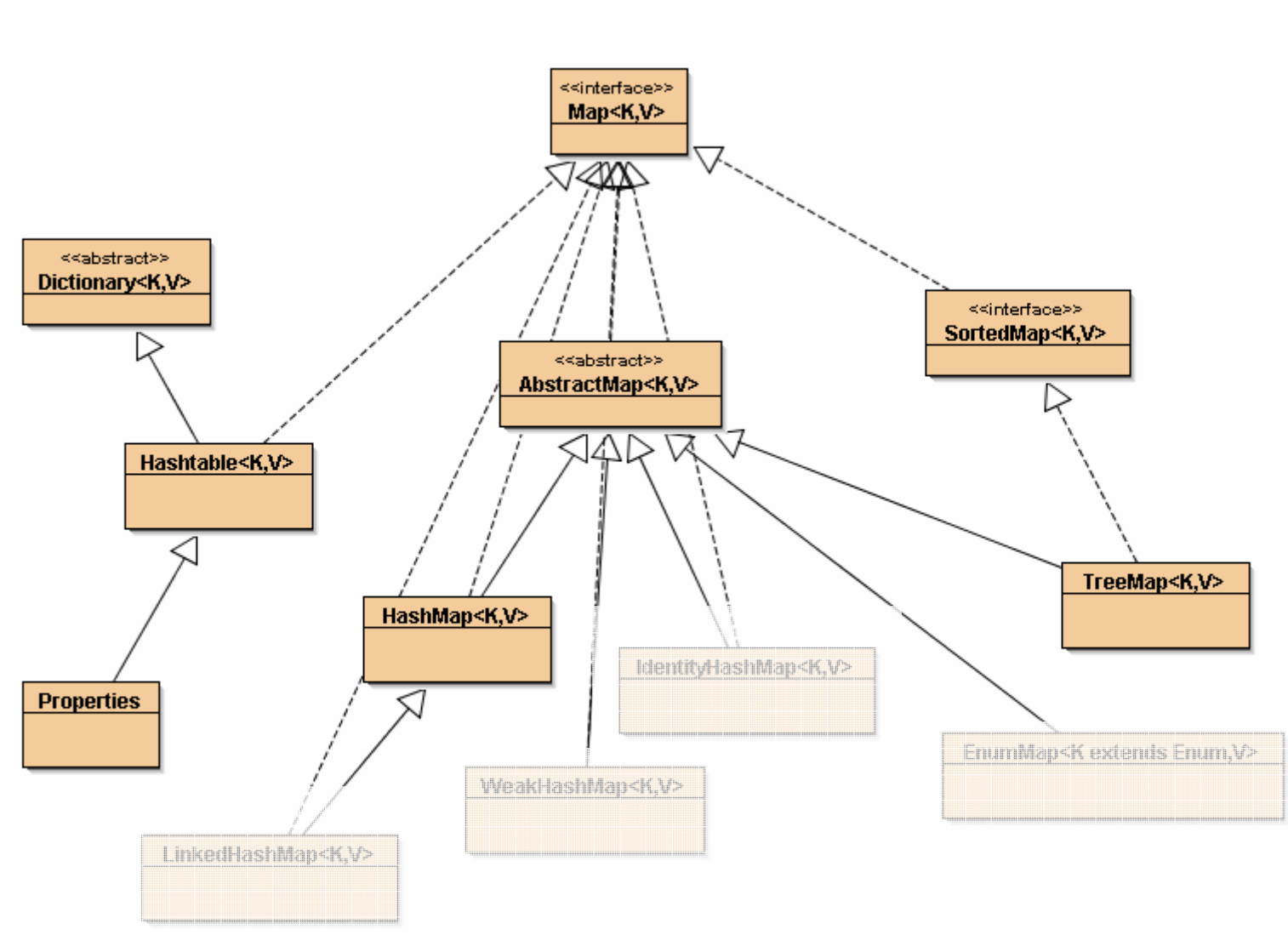
- **peek, poll, …**
- **BlockingQueue**
  - **FIFO avec lecture bloquante si pas d'éléments, schéma producteur/consommateur, bien utile en accès concurrent**
- **PriorityQueue**

Interface Map<K,V>

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

```
interface Map<K,V>{

   interface Entry<K,V>{
    ....
   }
...
}
```

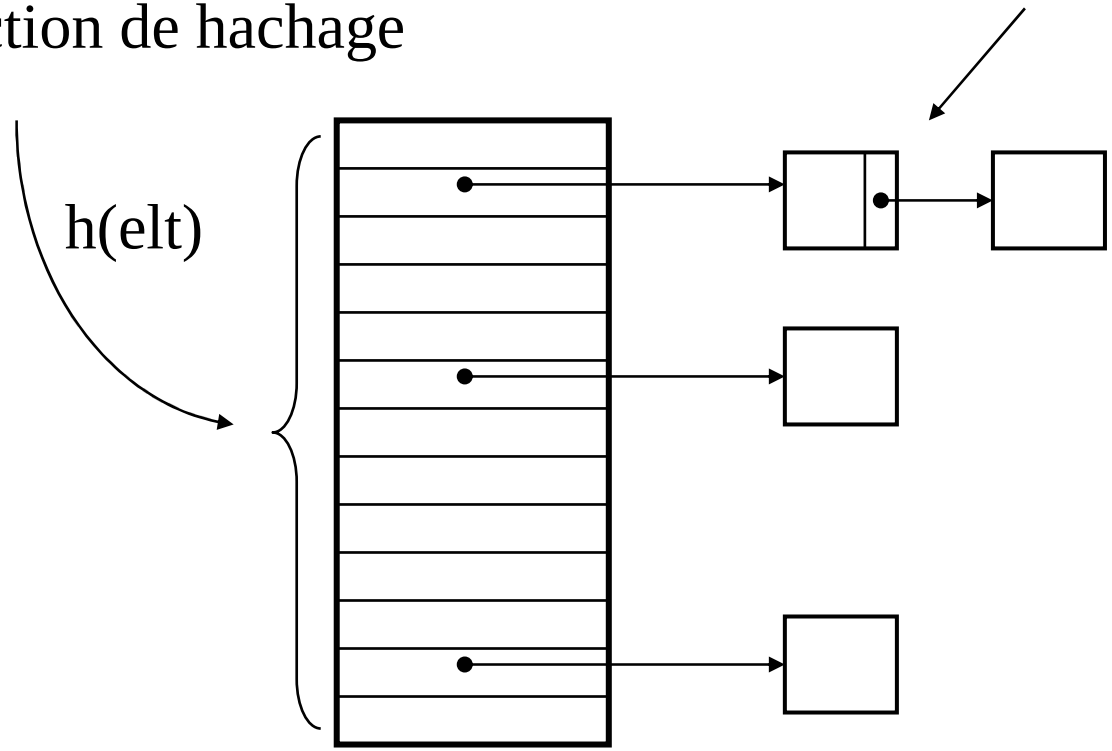
Adressage associatif, Hashtable



Une table de hachage

Ici gestion des collisions avec une liste

Fonction de hachage



public class Hashtable<KEY,VALUE> ...

Extrait de la javadoc : Generally, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the time cost to look up an entry (which is reflected in most Hashtable operations, including get and put).

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<K,V>

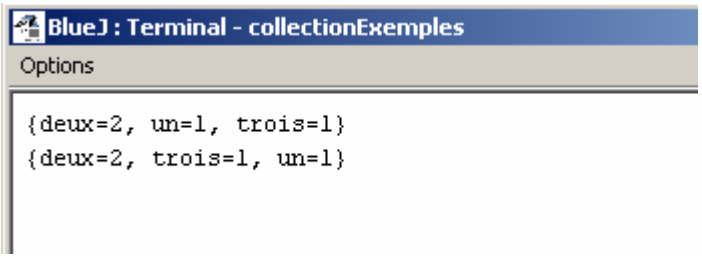
```
public interface Map<K,V>{

  // ...

 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>();

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

Un exemple : usage de occurrence

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


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



Parcours d'une *Map*, avec des *Map.Entry*

```
Map<String,Integer> map = new HashMap<String,Integer>();
…

 for(Map.Entry<String,Integer> m : map.entrySet()){
   System.out.println(m.getKey() + " , " + m.getValue());
  }

}
```

La suite …

- **Interface SortedMap<K,V>**
- **TreeMap<K,V> implements SortedMap<K,V>**
  - **Relation d ’ordre sur les clés**

- **et les classes**
  - **WeakHashMap**
  - **IdentityHashMap**
  - **EnumHashMap**
  - **???, un autre support**

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***
  - **// Singleton**
  - **// 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)**

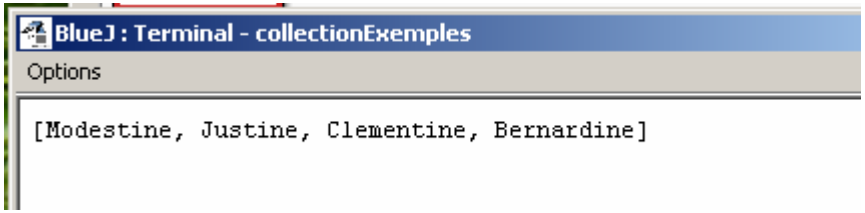
La méthode Collections.sort

Un usage …

```
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);
```

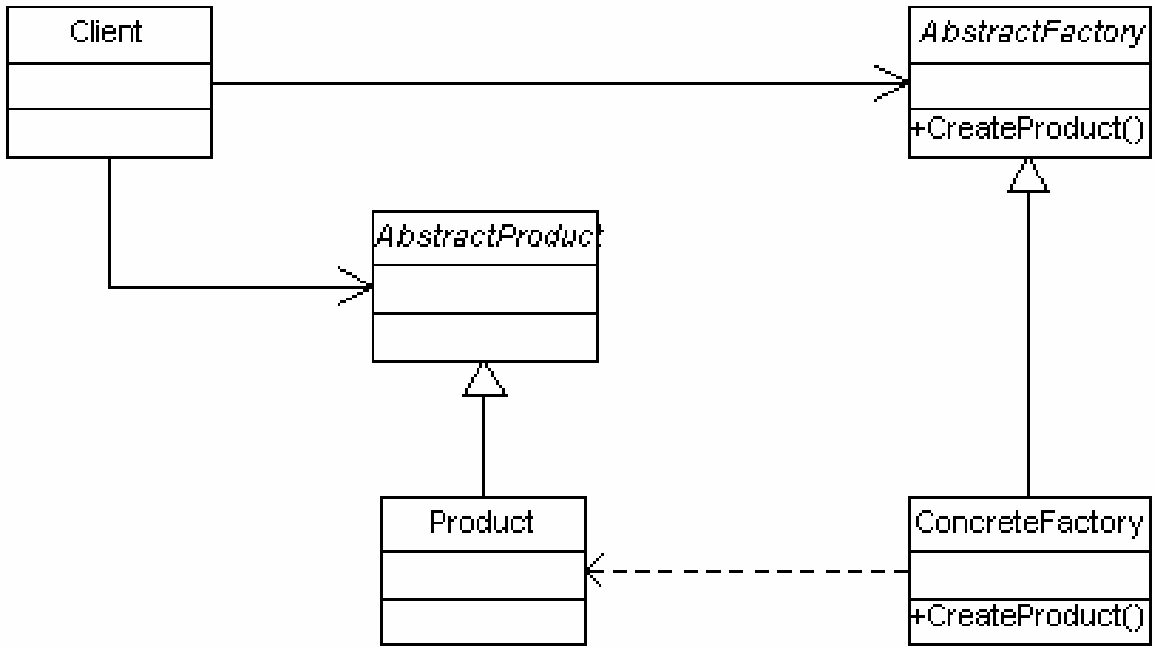


La classe Arrays

Rassemble des opérations sur les tableaux

```
static void sort(int[] t);
…
static <T> void sort(T[] t, Comparator<? super T> c)
boolean equals(int[] t, int[] t1);
…
int binarysearch(int[] t, int i);
…
static <T> List<T> asList(T… a);


…
```

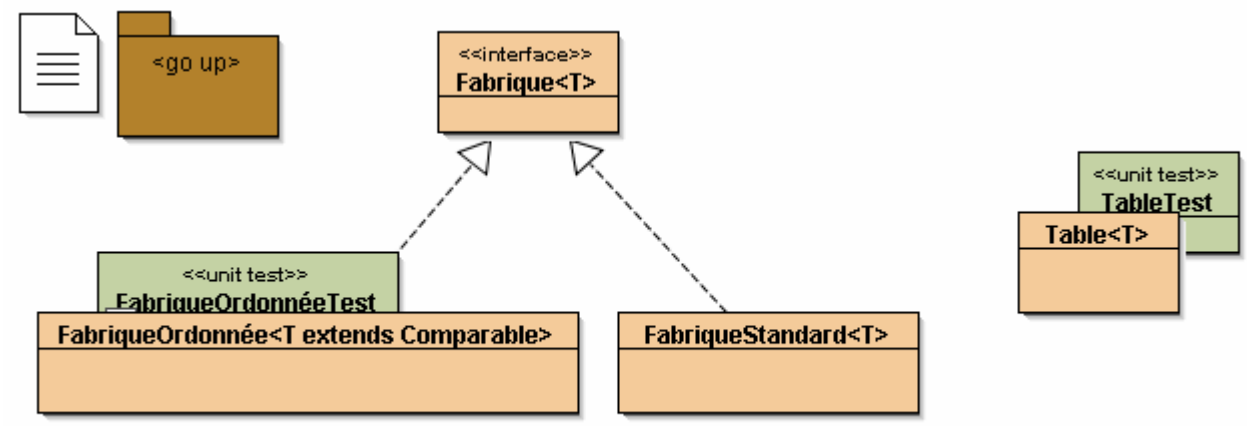
Le patron Fabrique, Factory method



- **L'implémentation est choisie par le client à l'exécution**
- **Comme « un constructeur » d'une classe au choix**

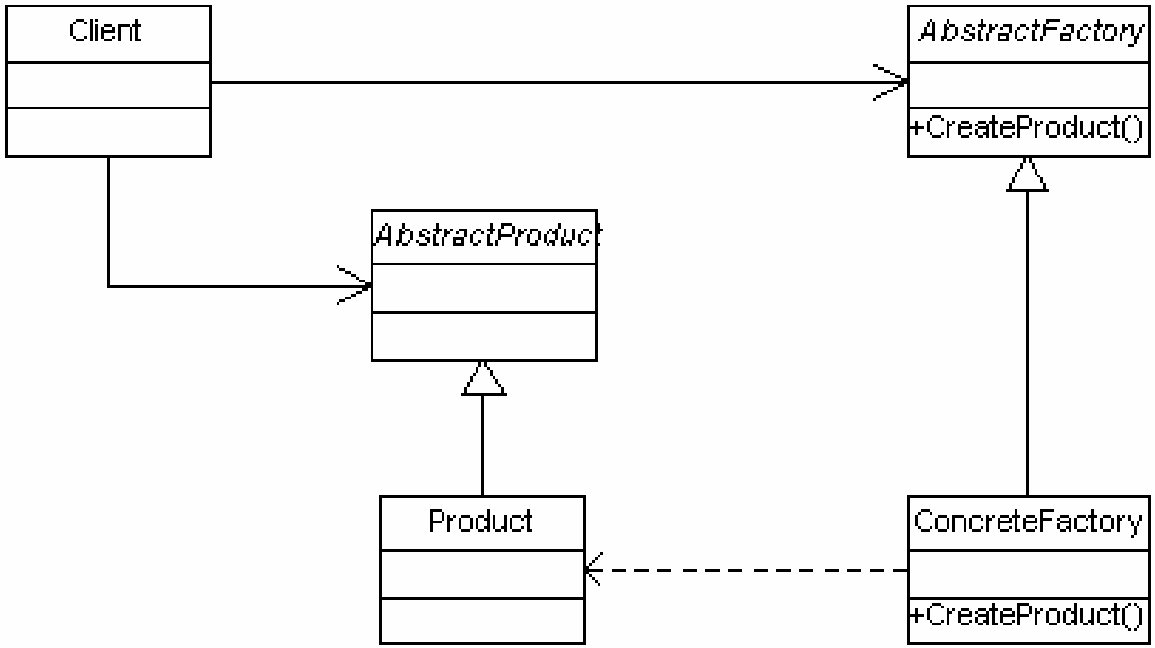
Patron fabrique

- **le pattern fabrique :**
  - **ici un ensemble qui implémente Set<T>**




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

Le patron Fabrique, Factory method



- public interface Fabrique<T>{
- public Set<T> **fabriquerUnEnsemble**();
- }

- Avec AbstractProduct Set
- Et CreateProduct **fabriquerUnEnsemble**

Le pattern Fabrique (1)

```
import java.util.TreeSet;
import java.util.Set;
public class FabriqueOrdonnée<T extends Comparable<T>>
                    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;
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){      // une injection de dépendance,
                                       // par l'usage du patron fabrique
      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 :**
  - **Choix d'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]", t2.toString());
assertEquals("[w, a, f]", t1.toString());
assertTrue(t1.equals(t2));
```

Conclusion

- **Lire, relire un tutoriel**

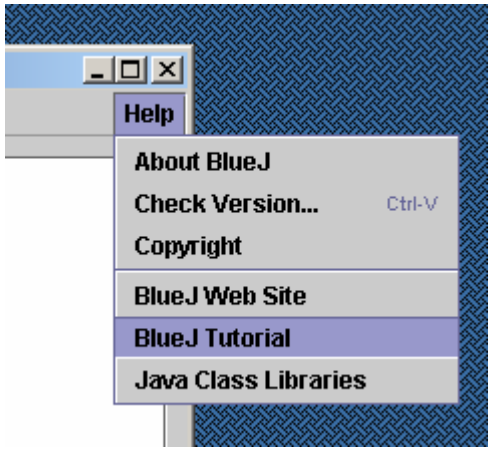
- **Utiliser Bluej**
  - **Outils puis Bibliothèque de classe**

- **Les types primitifs sont-ils oubliés ?**
  - **http://pcj.sourceforge.net/ Primitive Collections for Java. De Søren Bak**
  - **Depuis 1.5 voir également l 'autoboxing**

- **Ce support est une première approche !**
  - **Collections : des utilitaires bien utiles**
  - **Il en reste … WeakHashMap, … java.util.concurrent**

- **…**

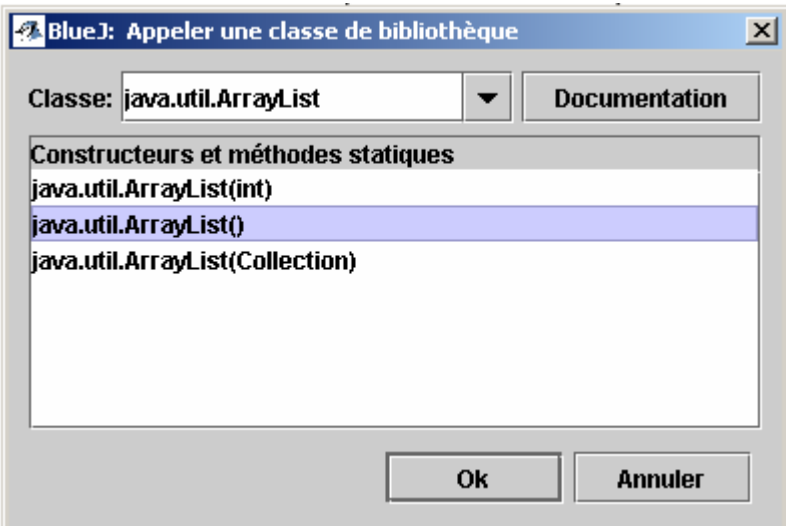
Documentation et tests unitaires

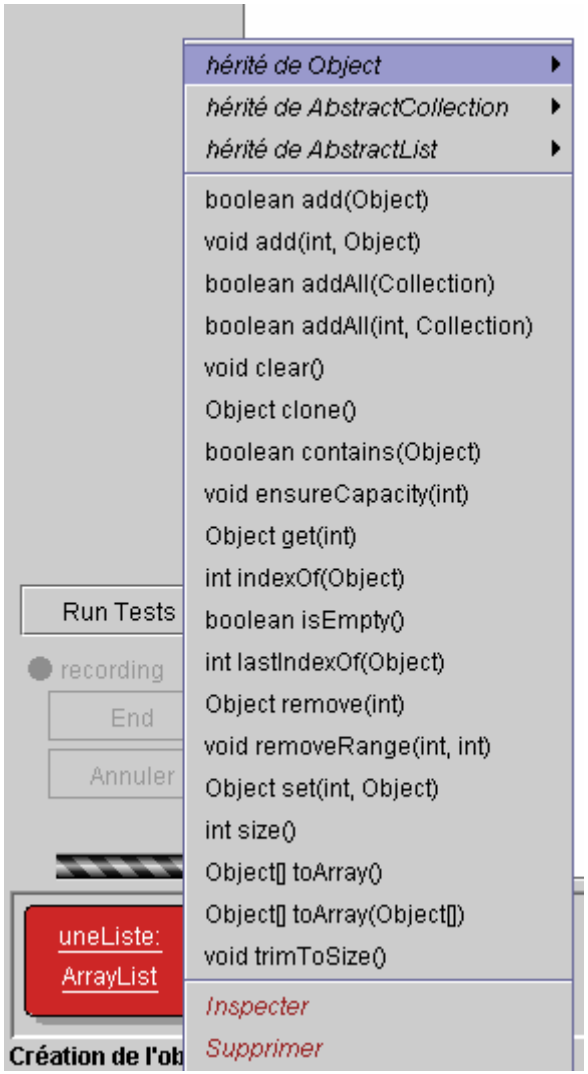


- **Documentation**
  - Java API
  - item Java Class Libraries
  - Tutorial
  - item BlueJ Tutorial

- **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,
extait du tutorial de Sun

```
public static <T> List<T> asList(T[] a) {
  return new MyArrayList<T>(a);
}

private static class MyArrayList<T> extends AbstractList<T>{
  private final T[] a;
  MyArrayList(T[] array) { a = array; }
  public T get(int index) { return a[index]; }
  public T set(int index, T element) {
    T oldValue = a[index];
    a[index] = element;
    return oldValue;
   }
  public int size() { return a.length; }
}
```

démonstration

De Collections en Tableaux

- **De la classe ArrayList**
  - **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.**

- **public <T> T[] toArray(T[] a)**
  - **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
```

Itération et fichier (une collection de caractères ?)

- **La classe Scanner**
  - **Analyseur lexical, typé et prêt à l'emploi**

```
Scanner sc = new Scanner(res.getText());
assertEquals("--> est attendu ???","-->",sc.next());
try{
   int leNombre = sc.nextInt();
   assertEquals(" occurrence  erroné ???",36,leNombre);
}catch(InputMismatchException ime){
   fail("--> N, N : un entier est attendu ???");
}
```

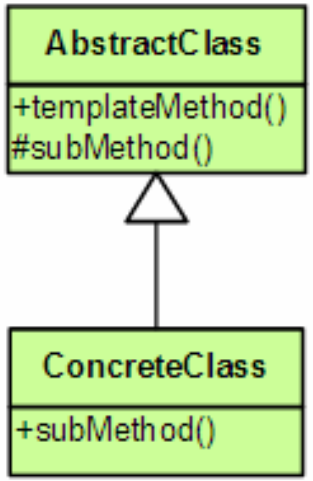
En rappel, extrait de

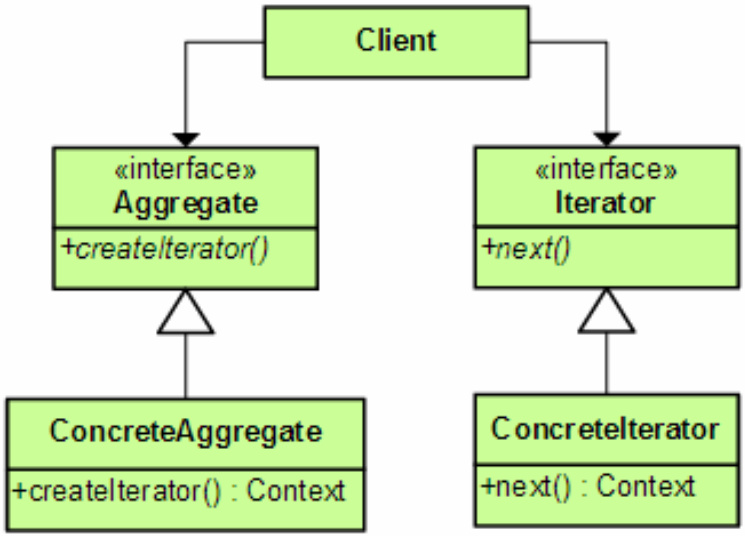
http://www.mcdonaldland.info/files/designpatterns/designpatternscard.pdf

Template Method

Type: Behavioral

What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.





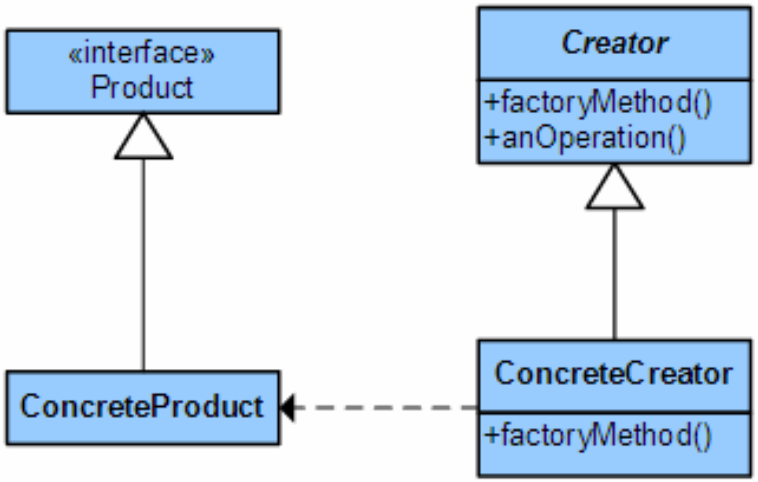
Iterator

Type: Behavioral

What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

En rappel, extrait de
http://www.mcdonaldland.info/files/designpatterns/designpatternscard.pdf

Factory Method

Type: Creational

What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.