
les Collections

Patrons : Template method, Iterator, Singleton et Factory method

jean-michel Douin, douin au cnam point fr
version : 23 Septembre 2014

Notes de cours

Sommaire pour les Patrons

- **Classification habituelle**

- **Créateurs**

- Abstract Factory, Builder, Factory Method, Prototype, Singleton

- **Structurels**

- Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy

- **Comportementaux**

- Chain of Responsibility, 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**

Sommaire pour les collections

- **Pourquoi ? Quels objectifs ?**
- **Quelles interfaces ?**
- *Quelles classes abstraites ?*
- **Quelles classes concrètes ?**
- **Quels utilitaires ?**

Principale bibliographie

- **Le tutorial de Sun/Oracle**

- <http://docs.oracle.com/javase/tutorial/collections/>

- **Introduction to the Collections Framework**

- <http://docs.oracle.com/javase/7/docs/technotes/guides/collections/overview.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)
- *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.

Préambule

- **Généricité en java**

- **Syntaxe, usage**
- **<E>**
- **<? extends Etudiant>**
- **...**

<E> comme généricité (ici des généralités...)

- Une collection d'objets
- si l'on souhaite une collection homogène <E>
 - 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'Object
 - 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 de Sun

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

- **? Joker** « compatible avec n'importe quelle classe »

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

```
package java.util;  
public class Collection<E>{  
  
    public boolean containsAll(Collection<?> c) {... }  
}
```

Généricité,

afin de pouvoir lire la documentation de Sun

- **Contraintes sur l'arbre d'héritage**
- **<? extends E>**
 - E doit être une super classe de la classe associée au joker (?)
- **<? super E>**
 - E doit être une sous classe de la classe associée au joker (?)
- **<? extends Comparable<E>>**
 - La classe associée au joker (?) doit implémenter l'interface Comparable
- **<? extends Comparable<? super E>>**
 - Une des super classes de la classe associée au joker (?) doit implémenter l'interface Comparable
- **<? extends Comparable<E> & Serializable>**
 - Une des Doit implémenter l'interface Comparable<E> et l'interface java.io.Serializable

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, cf. paquetage java.util, ici partiel

- **Quelles fonctionnalités ?**

- Quelles interfaces ?
- Collection<E>, Iterable<E>, Set<E>, SortedSet<E>, List<E>, Map<K,V>, 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 (toutes prêtes) ?
- LinkedList<E>, ArrayList<E>, Queue<E>, PriorityQueue<E>
- TreeSet<E>, HashSet<E>,
- WeakHashMap<K,V>, HashMap<K,V>, TreeMap<K,V>

- **Quelles passerelles ?**

- Collections et Arrays

Mais quelques patrons avant tout

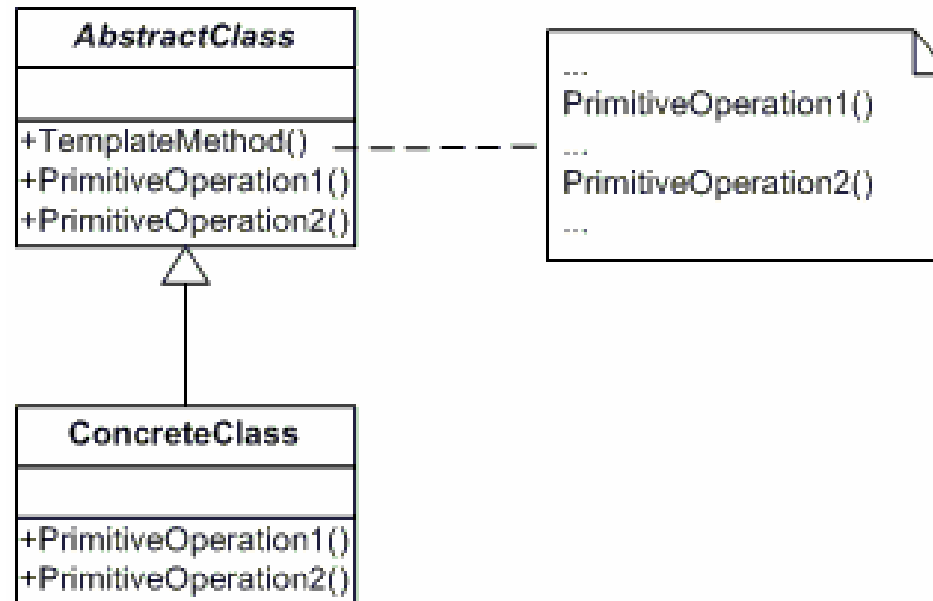
- **Template Method**

- Laisser la réalisation de certaines méthodes aux sous-classes
- Utilisé pour `AbstractCollection<E>`, `AbstractSet<E>`, `AbstractList<E>`, `AbstractSequentialList<E>`, `AbstractMap<K,V>`

- **Iterator**

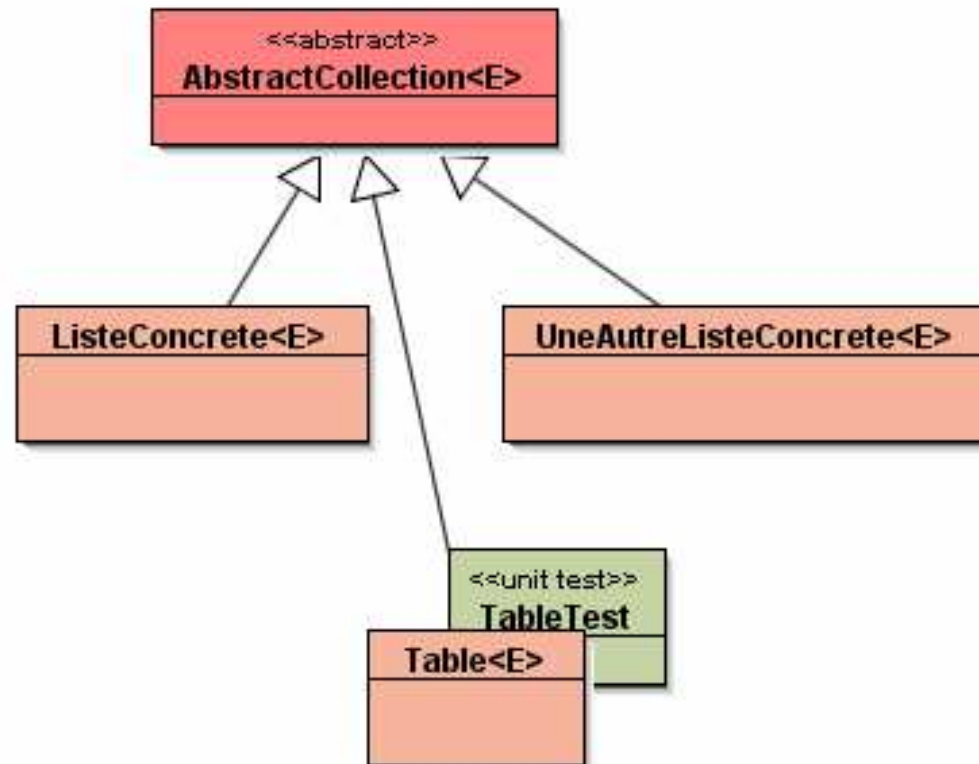
- Parcourir une collection sans se soucier de son implémentation
- Chaque collection est « `Iterable<E>` » et propose donc un itérateur

Template Method



- Laisser aux sous-classes de grandes initiatives ...

Template Method : un exemple



- **AbstractCollection :**
 - Deux méthodes concrètes : taille, retirer
 - Deux méthodes abstraites : iterator, ajouter

Template Method : Démonstration

```
interface Collection<E> implements Iterable<E>{
    boolean ajouter(E e);
    void taille();
    boolean retirer(E e);
}

public abstract AbstractCollection<E> implements Collection<E>{

    public void taille(){

    }

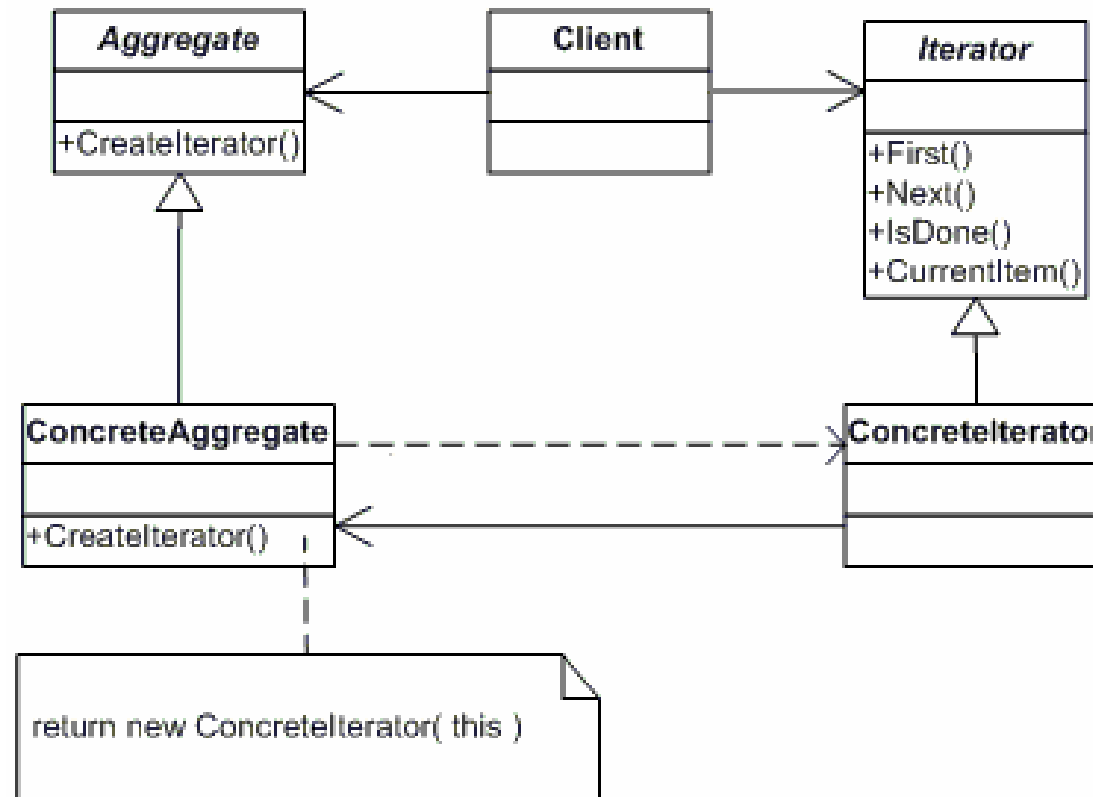
    public boolean retirer(E e){

    }

}

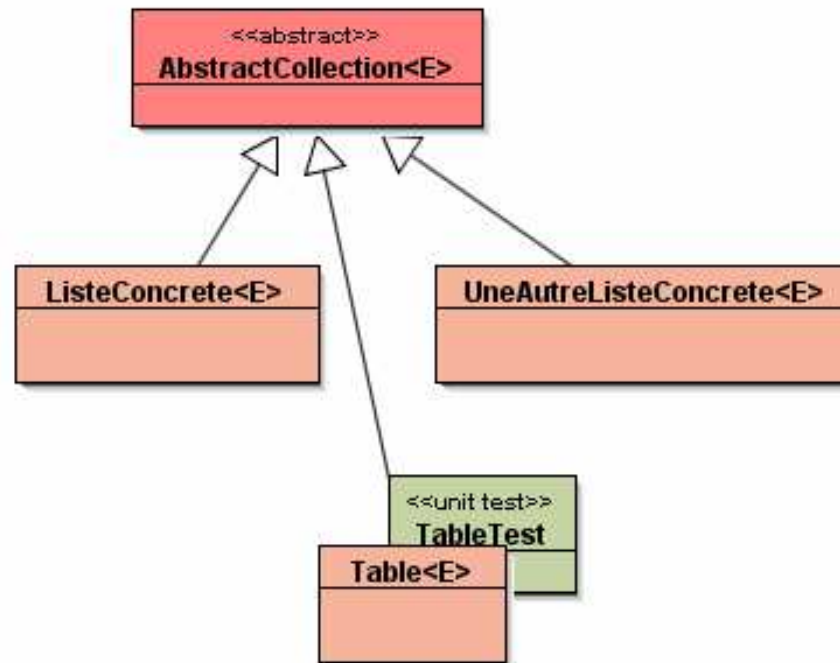
public ListeConcrete<E> extends AbstractCollection<E> {
    public boolean ajouter(E e){ ... }
    public Iterator<E> iterator(){ ... }
}
```

Iterator



- Ou comment parcourir une structure quelque soit son implémentation

Iterator : un exemple



- **Table<E>**
 - Avec `E[]` comme contenu
 - deux méthodes à implémenter : ajouter et iterator

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){

        if(courant <= TAILLE_MAX){
            contenu[courant] = e;
            courant++;
            return true;
        }
        return false;
    }

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

Iterator : Démonstration

```
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;
    }

    @SuppressWarnings("unchecked")
    public E next(){
        Object obj = contenu[index];
        index++;
        nextOk = true;
        return (E)obj;
    }

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

Les Collections en Java : deux interfaces

- **interface Collection<T>**

- Pour les listes et les ensembles

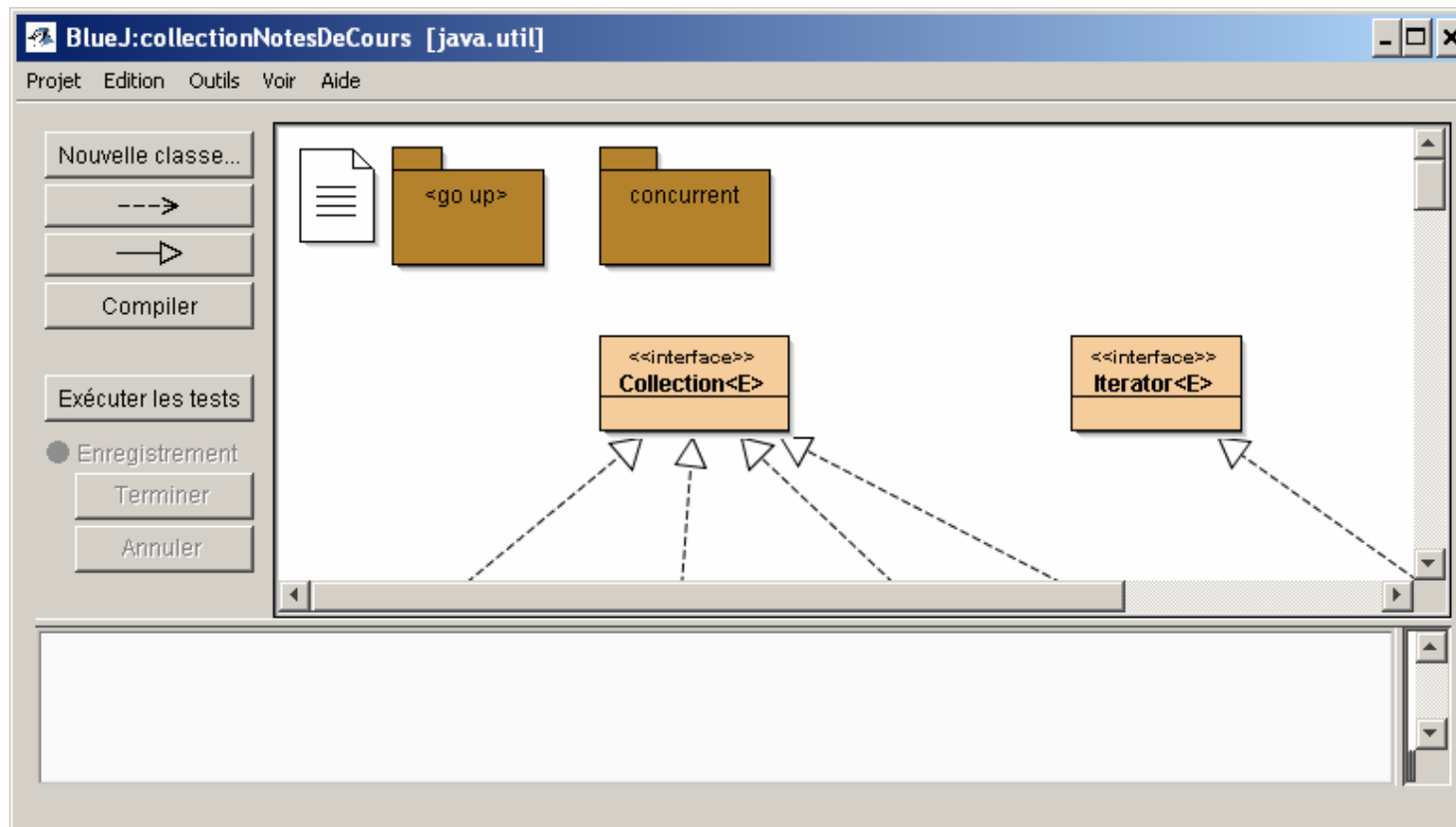
- package java.util;
- public interface Collection<E> extends Iterable<E>{
- ...

- **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();

}

java.lang.Iterable<T>

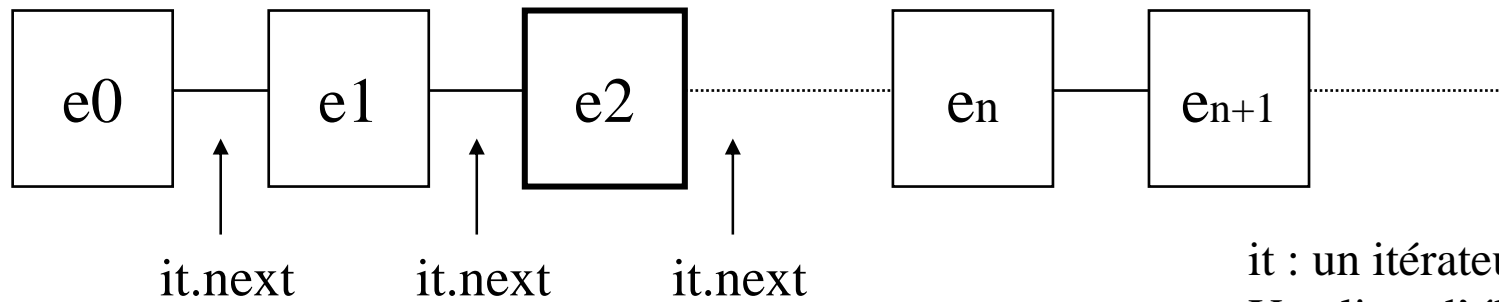
- Le patron Iterator
- *public interface Collection<E> extends Iterable<E>*

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

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);
```

```
}
```

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 ?

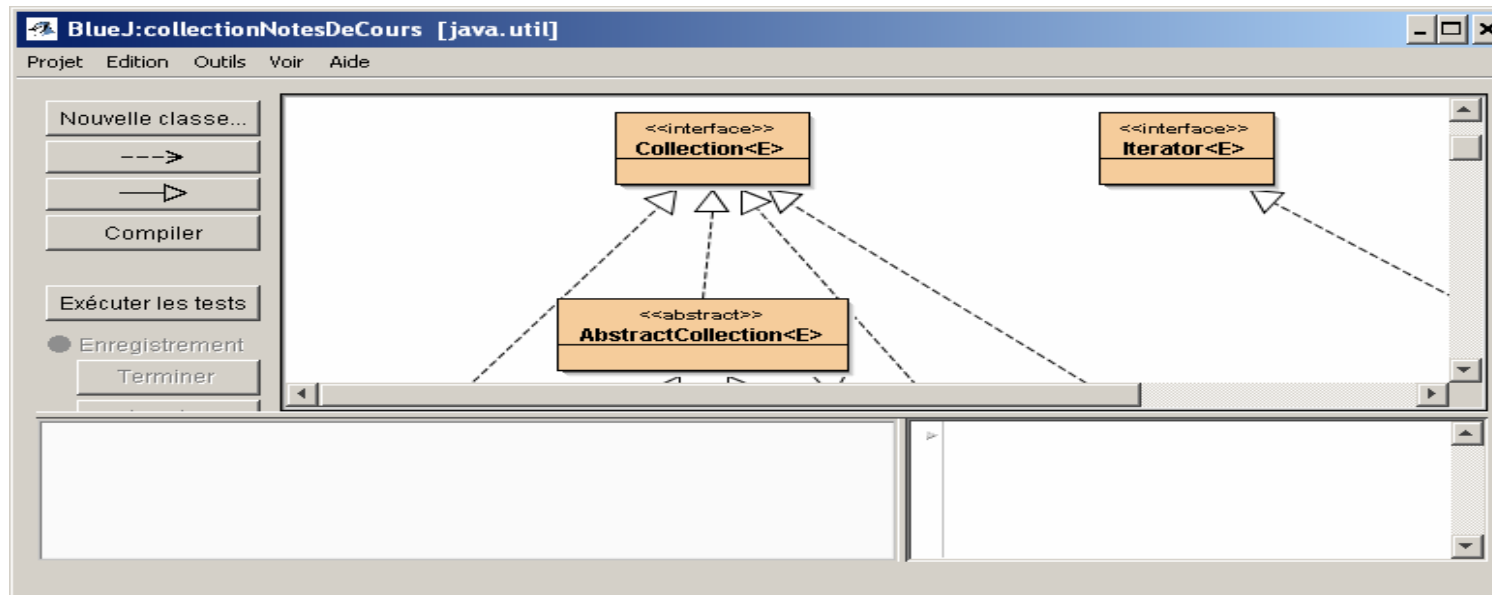
- **Extrait de la documentation pour next :**
Returns the next element in the iteration.
- **Une liste, une table**
- **Un arbre binaire ?**
 - Prefixé, infixé ... ?
- **Un dictionnaire ?**
- **... etc**

Lire la documentation ...

AbstractCollection<E>

- **AbstractCollection<E> implements Collection<E>**
 - Implémentations effectives de 13 méthodes sur 15 !

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) ...

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

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);
```

```
assertTrue(c.retainAll(c1));  
assertEquals(c.toString(), "[3, 4]");
```

En exercice ... à rendre..., vous avez 5 minutes ...

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://java.sun.com/javase/6/docs/api/java/util/package-summary.html>

Interfaces:

- **Iterable<E>**
- **Iterator<E>**
- **Collection<E>**

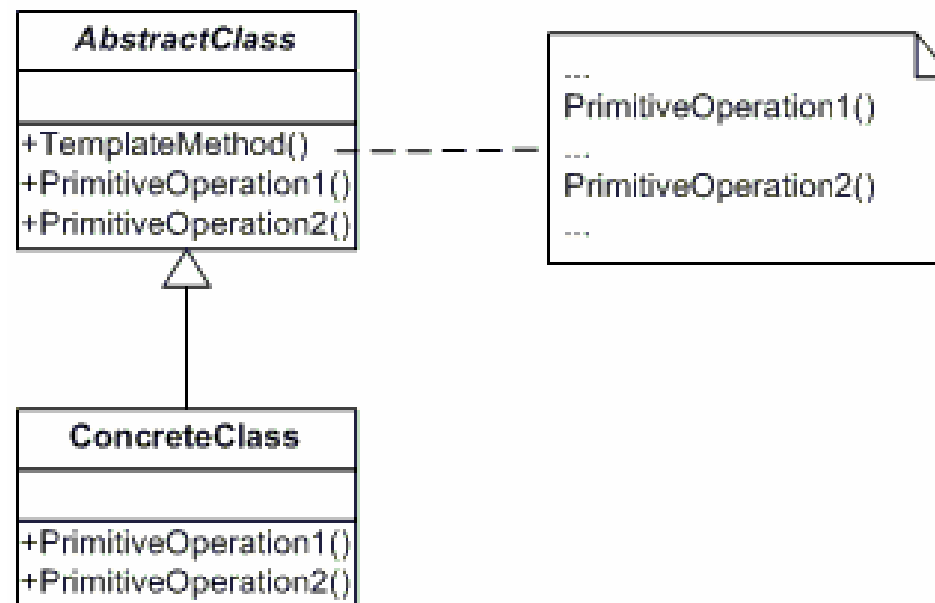
Classes incomplètes

- **AbstractCollection<E>**

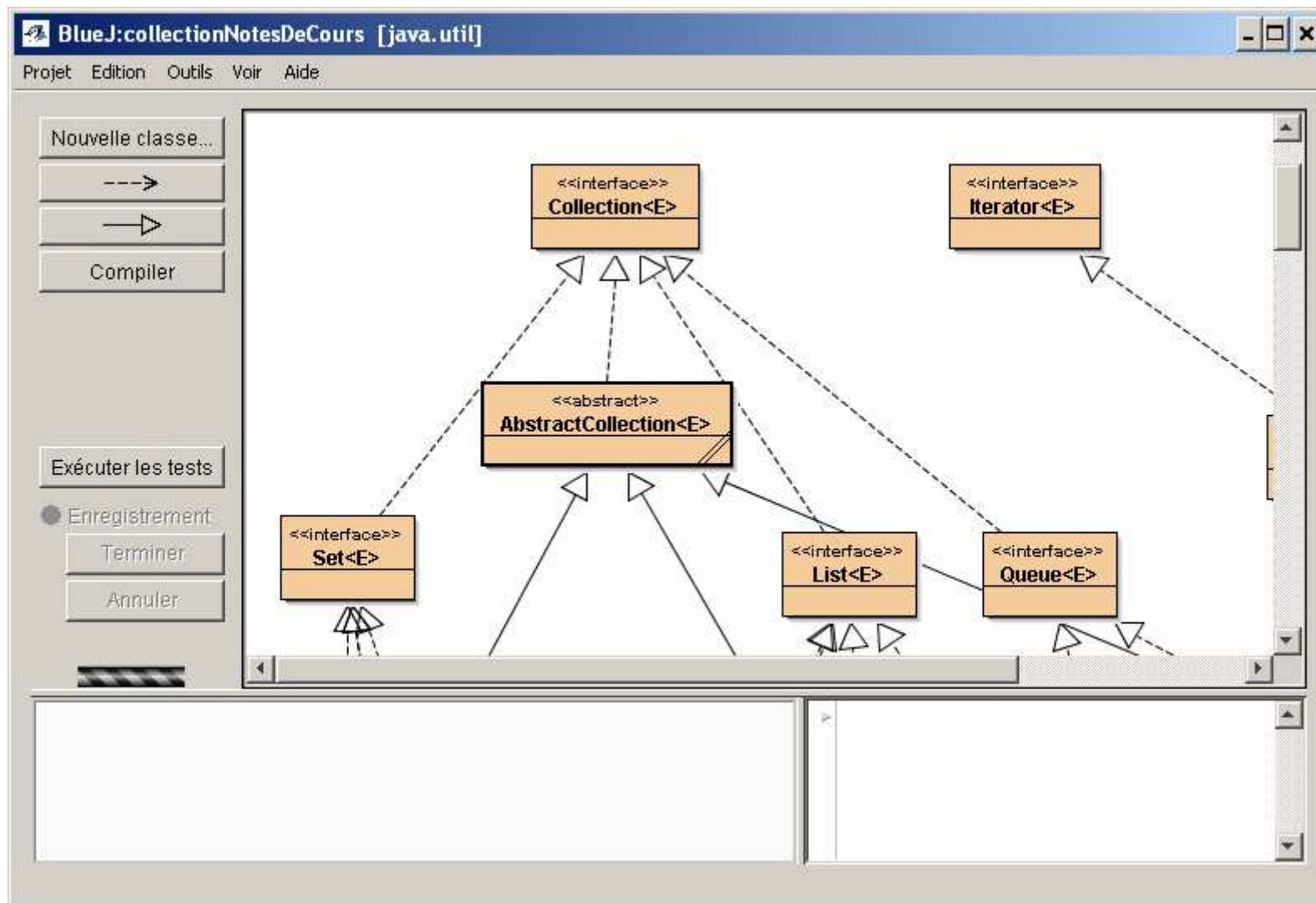
Patron Template Method

- **Discussion**

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



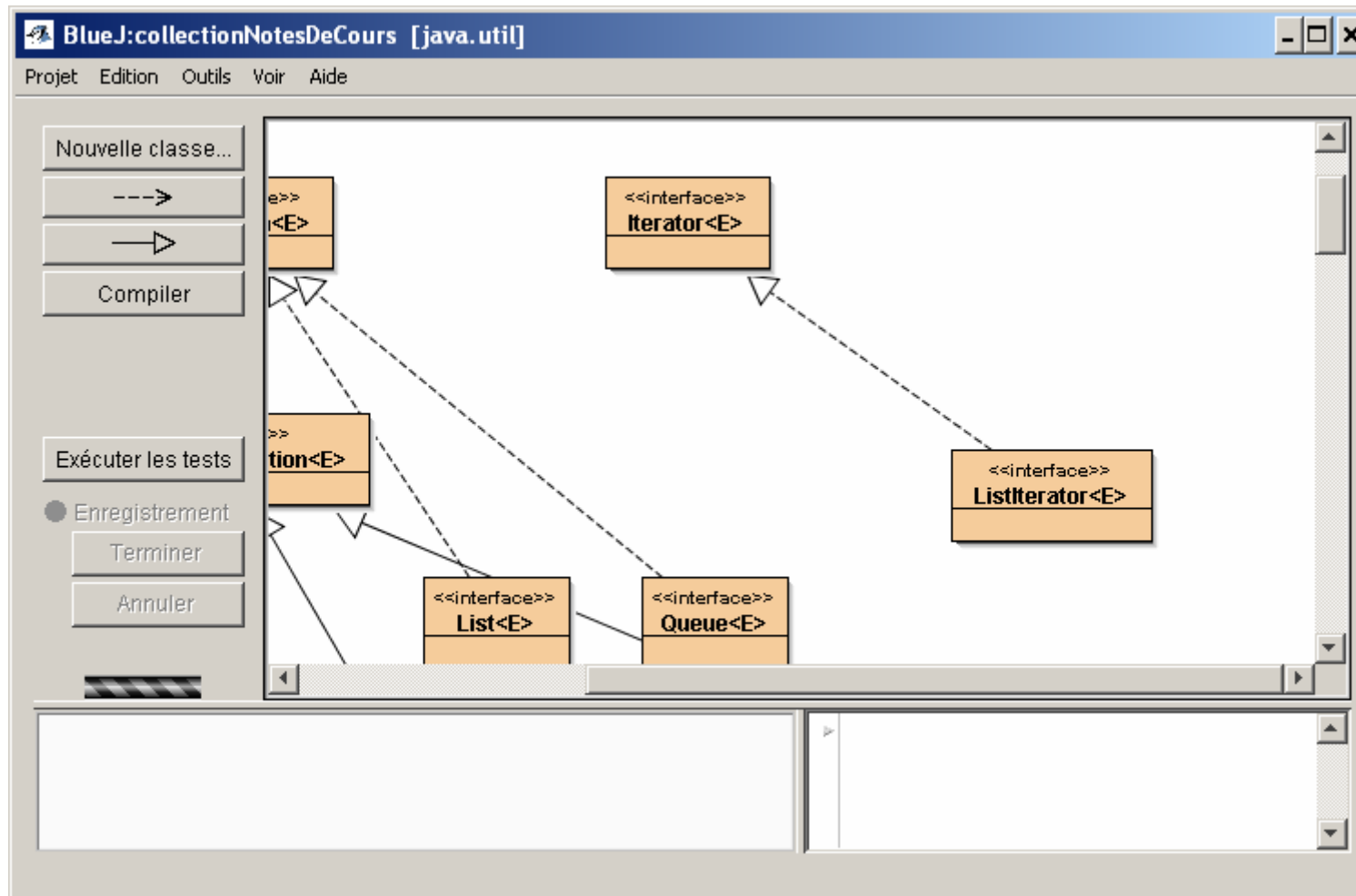
Interfaces List<E>, Set<E> et Queue<E>



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>

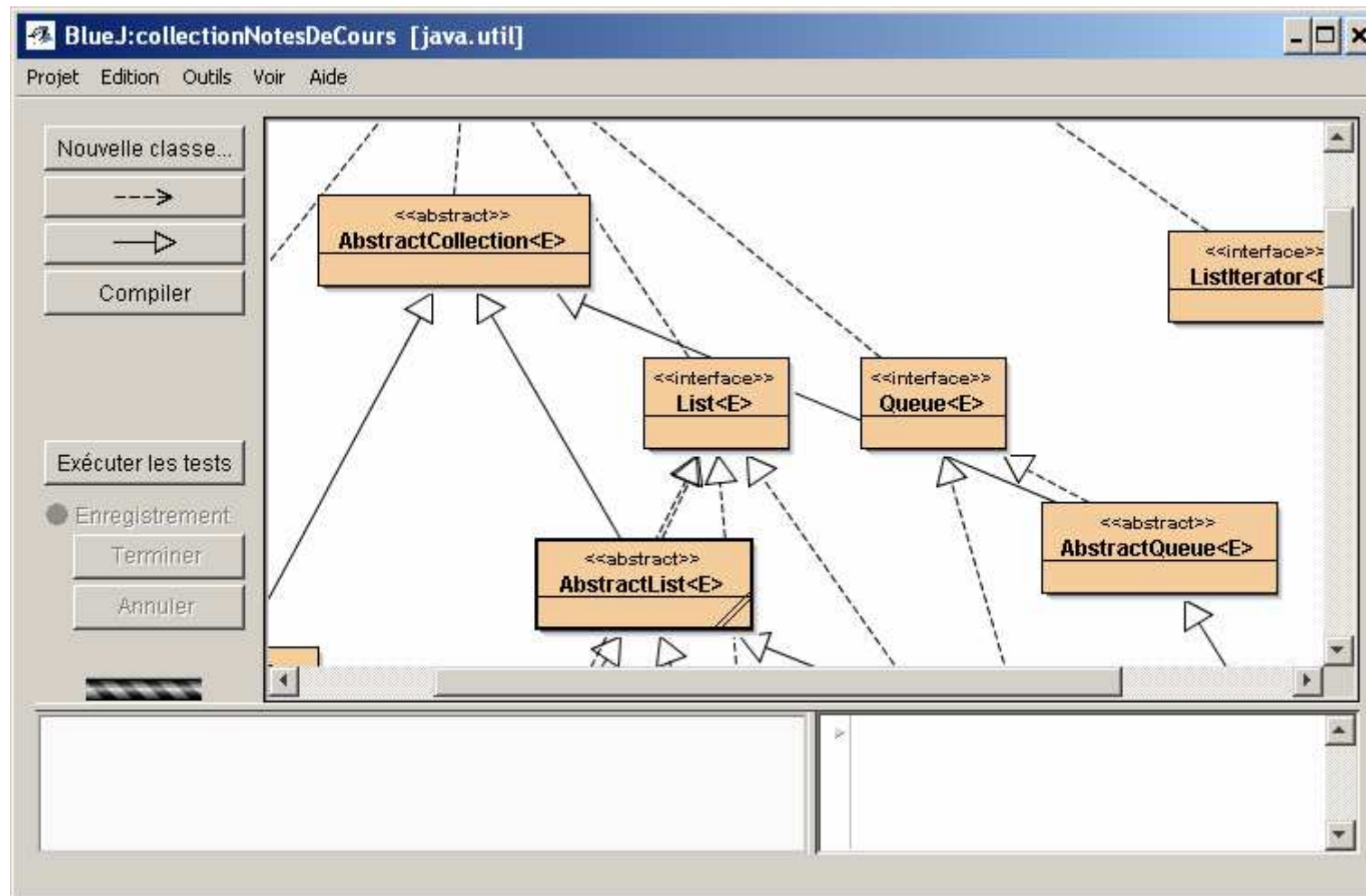


- **Parcours dans les 2 sens de la liste**
 - next et previous
 - Méthode d'écriture : set(Object element)

ListIterator<E>

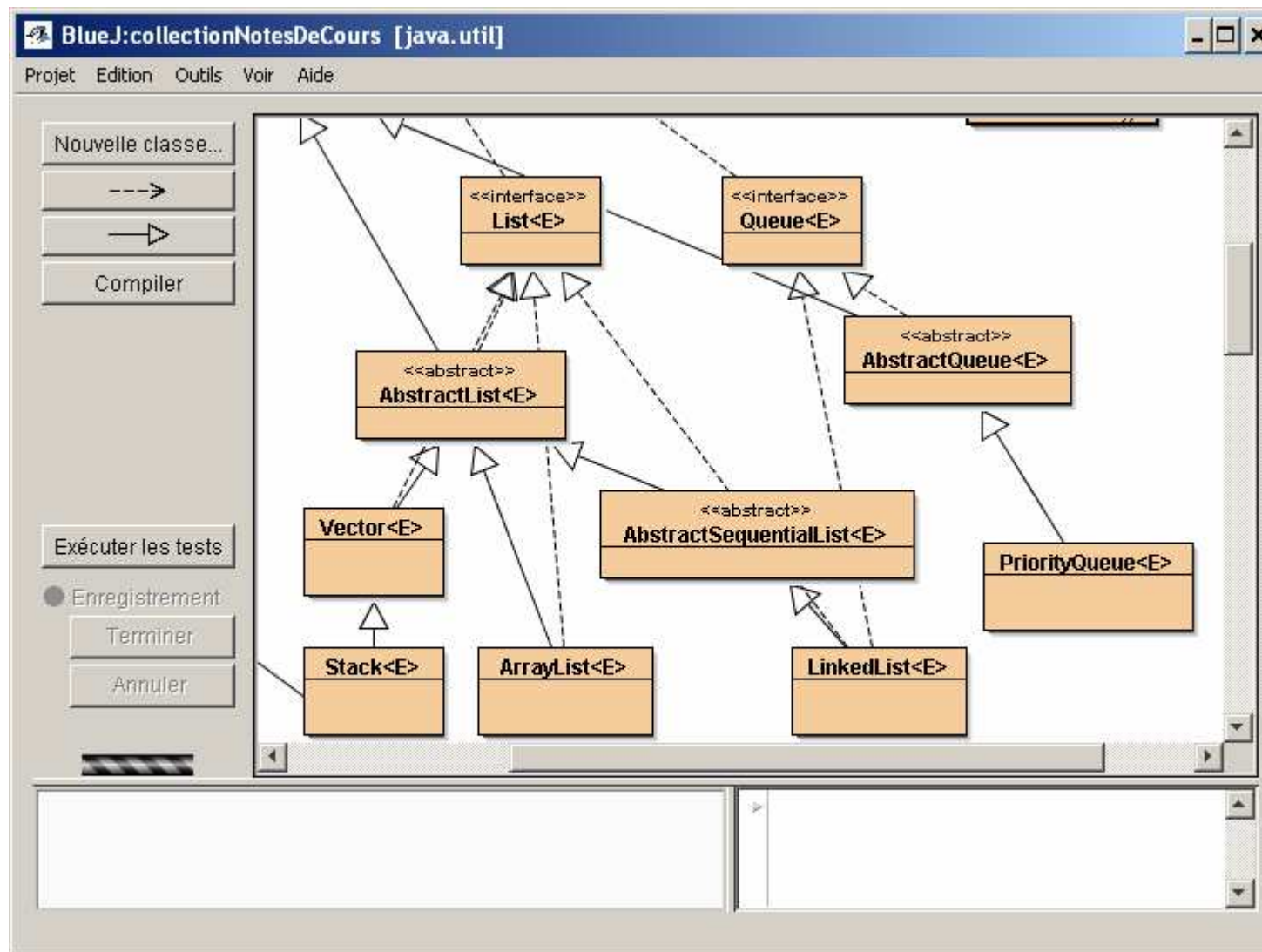
```
public interface ListIterator<E> extends Iterator<E>{  
    E next();  
    boolean hasNext();  
  
    E previous();  
    boolean hasPrevious();  
  
    int nextIndex();  
    int previousIndex();  
  
    void set(E o);  
    void add(E o);  
  
    void remove();  
}
```


AbstractList<E>



- **AbstractList<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

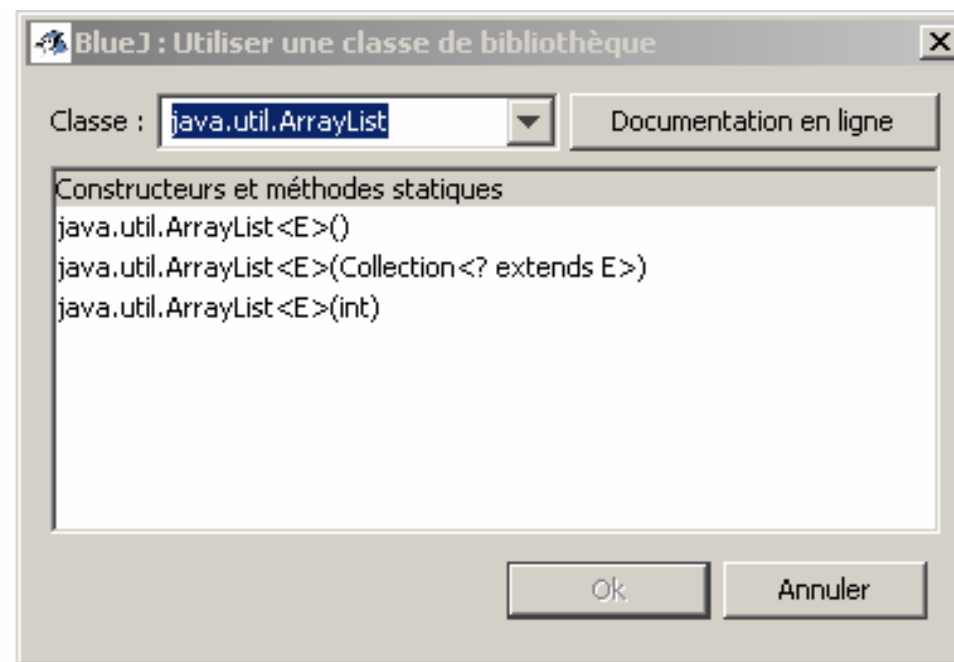
```
import java.util.*;
public class ListExample {
    public static void main(String args[]) {
        List<String> list = new ArrayList <String>();
        list.add("Bernardine"); list.add("Modestine"); list.add("Clementine");
        list.add("Justine");list.add("Clementine");
        System.out.println(list);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



Un patron comme intermède ...

Singleton
-instance : Singleton
-Singleton() +Instance() : Singleton

- **Le Patron Singleton**
 - garantir une et une seule instance d'une classe

Singleton, une idée

```
public class Singleton{
    // -----

    // ----- ci-dessous lignes propres au Singleton
    private static Singleton instance = null;

    public static Singleton getInstance(){
        if(instance==null)
            instance = new Singleton();

        return instance;
    }

    private Singleton(){
    }
}
```

Singleton : un usage

```
public class Singleton{
    // ----- ci-dessous lignes propres à la classe
    // dont on ne veut qu'une seule instance
    private Collection<Integer> value = new ArrayList<Integer>();

    public Collection<Integer> getValue(){return value;}

    // ----- ci-dessous lignes propres au Singleton
    private static Singleton instance = null;

    public static Singleton getInstance(){
        if(instance==null)
            instance = new Singleton();

        return instance;
    }
    private Singleton(){
    }
}
```

Attention version sans accès concurrent ... NotThreadSafe....

Singleton : un test

```
Singleton s1 = Singleton.getInstance();
```

```
assertNotNull(s1);
```

```
Singleton s2 = Singleton.getInstance();
```

```
assertSame(s2, s1);
```

```
Singleton.getInstance().getValue().add(5);
```

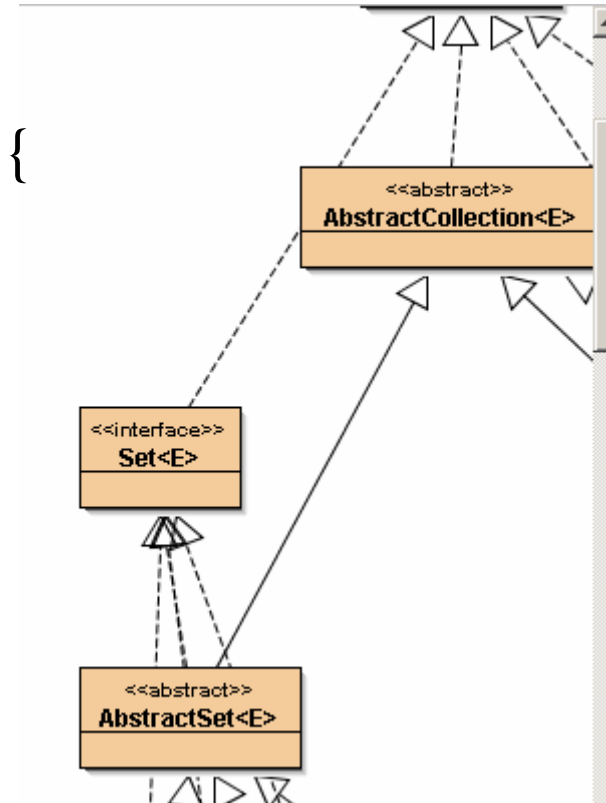
Fin de l'intermède

La suite des collections

- Interfaces pour les ensembles
 - *Set<T>*
 - *SortedSet<T extends Comparable<T>>*
- Interfaces pour un dictionnaire
 - *Map<K,V>*
 - *Map.Entry<K,V>* , interface imbriquée

Set et AbstractSet

```
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 **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);    // 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](#), [URL](#), [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 super.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);  
    }  
  
}
```

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>>**

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, utile en accès concurrent

Interface Map<K,V>

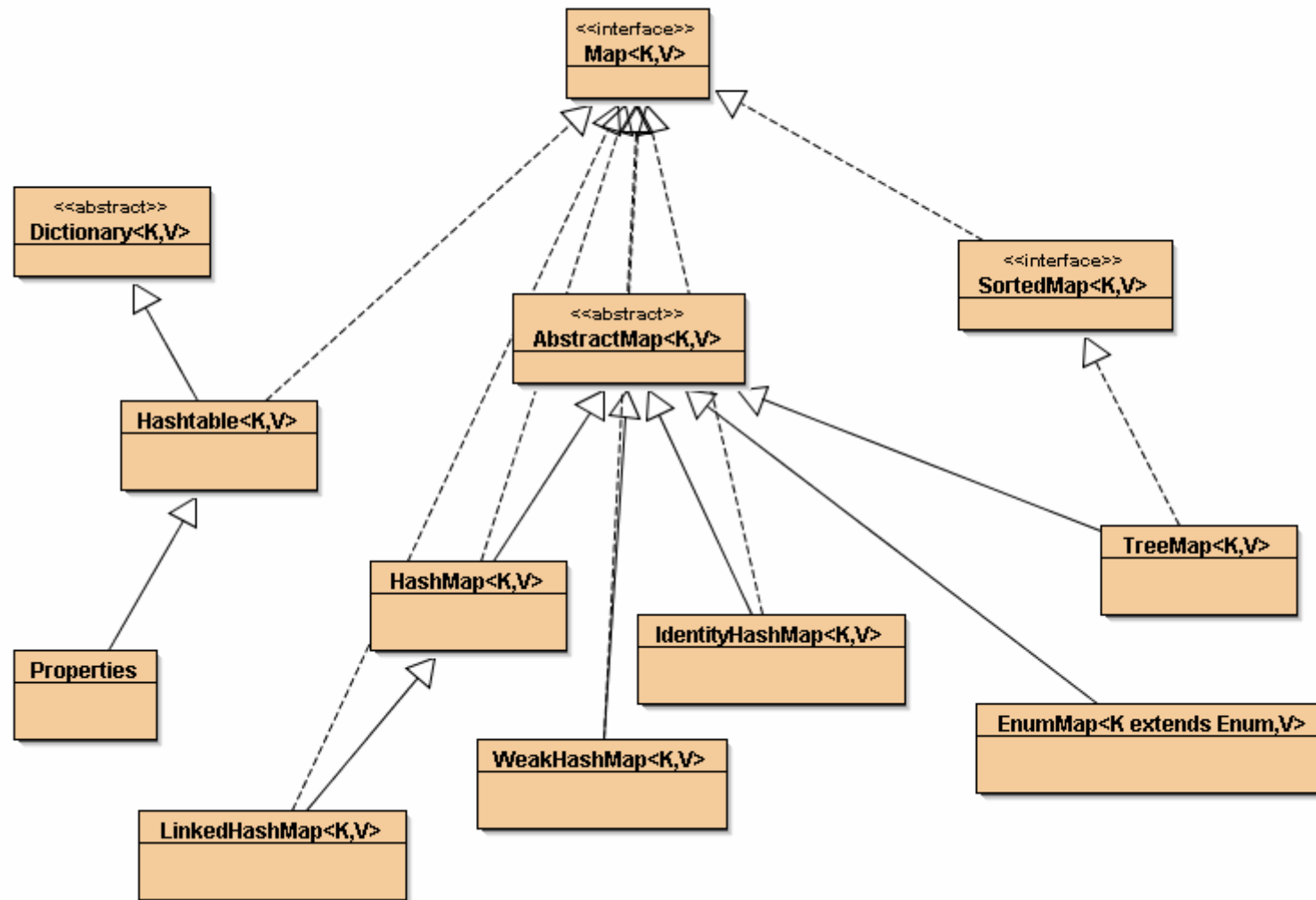
- **La 2ème interface Racine**
- **implémentée par les dictionnaires**
- **gestion de couples <Key, Value>**
 - la clé étant unique

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

```
...
```

```
}
```

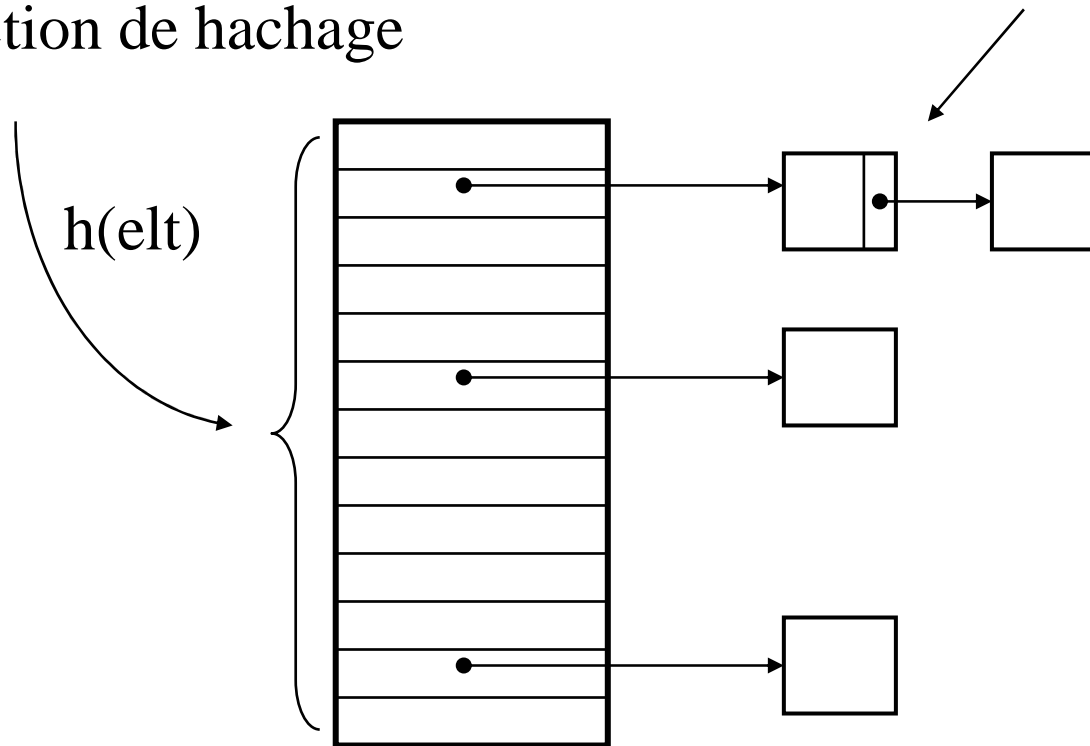
Adressage associatif, Hashtable



Une table de hachage

Fonction de hachage

Gestion des collisions avec une liste



```
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<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

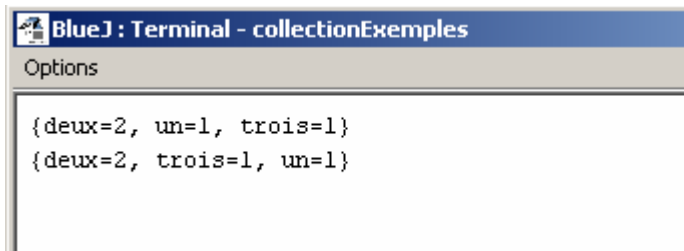
```
public Map<String,Integer> occurrence(Collection<String> c){
    Map<String,Integer> map = new HashMap<String,Integer>();

    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(l);
    System.out.println(map);

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



BlueJ : Terminal - collectionExemples

Options

```
{deux=2, un=1, trois=1}
{deux=2, trois=1, un=1}
```

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**
 - TreeMap
 - WeakHashMap
 - IdentityHashMap
 - EnumHashMap

La classe Collections : très utile

- Class Collections {

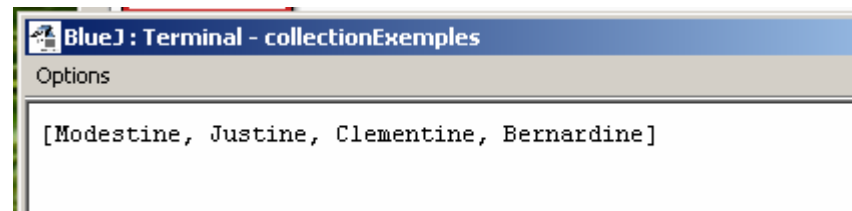
- // Read only : *unmodifiableInterface*
- static <T> Collection<T> unmodifiableCollection(Collection<? extends T> collection)
- static <T> List<T> unmodifiableList(List<? extends T> list)
- ...
-
- // Thread safe : *synchronizedInterface*
- static <T> Collection<T> synchronizedCollection(Collection<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);
```

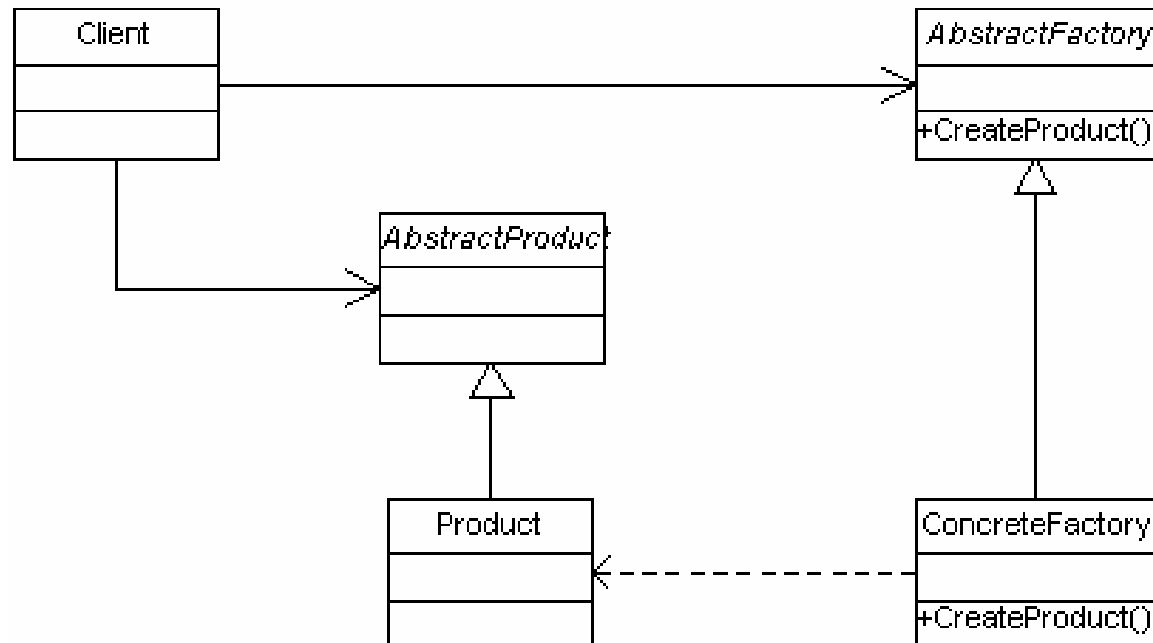


La classe Arrays

Rassemble des opérations sur les tableaux

```
static void sort(int[] t);  
...  
static <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);  
...  
...
```

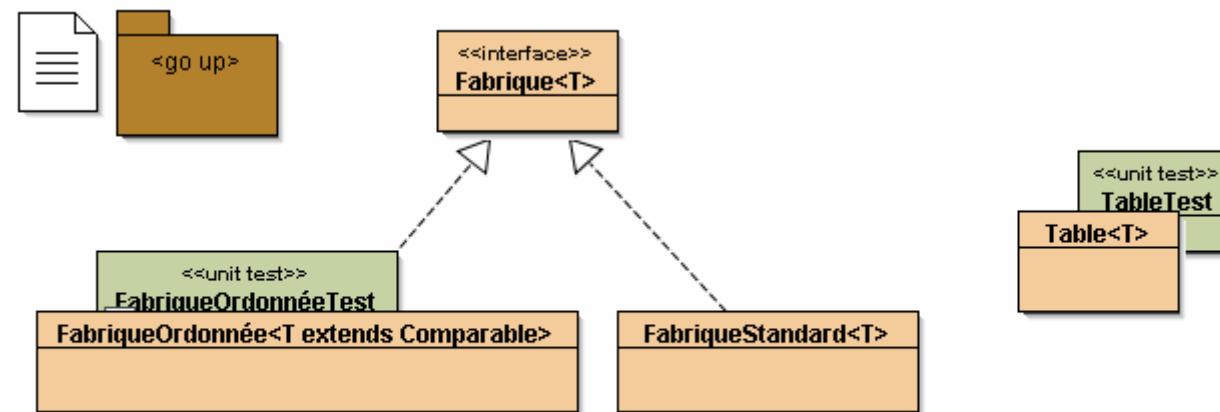
Le patron Fabrique, Factory method



- L'implémentation est choisie par le client à l'exécution
- « 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 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){        // injection de dépendance et
                                            // 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));
```

Avertissement

- **Les 3 prochaines diapositives sont-elles bien utiles ?**
- **Discussion de l'usage de l'introspection au sein du patron fabrique**
- *Utile/inutile ...*

Fabrique générique et introspection

- **La classe Class est générique mais**

```
public class Fabrication<T> {  
    public T fabriquer(){  
        return T.class.newInstance(); // erreur de compil  
        return new T();                // erreur de compil  
    }  
}
```

- **Mais avons nous ?**

```
public T getInstance(Class<T>, int id)
```

Classe Class générique

- `Class<?> c11 = Integer.class;`
- `Class<? extends Number> c1 = Integer.class; // yes`

```
public < T extends Number> T[] toArray(int n, Class<T> type){  
    T[] res = (T[])Array.newInstance(type,n);  
    return res;  
}
```

```
Integer[] t = toArray(4, Integer.class);    // satisfaisant
```

La Fabrication revisitée

```
public class Fabrication<T> {  
  
    public T fabriquer(Class<T> type) throws Exception{  
        return type.newInstance();  
    }  
}
```

- Usage :

```
Integer i = new Fabrication<Integer>().fabriquer(Integer.class);  
Number n = new Fabrication<Integer>().fabriquer(Integer.class);
```

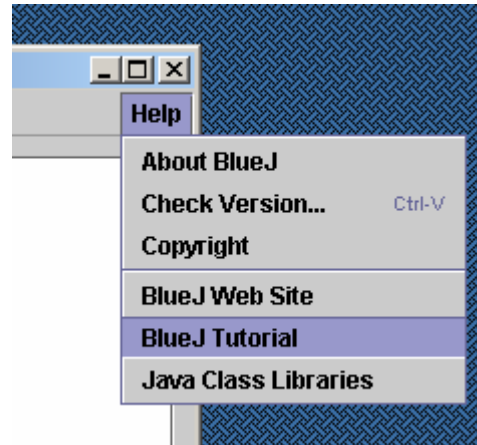
Exercice : Set<Integer> = new Fabrication<.....>()

Utile/inutile ...

Conclusion

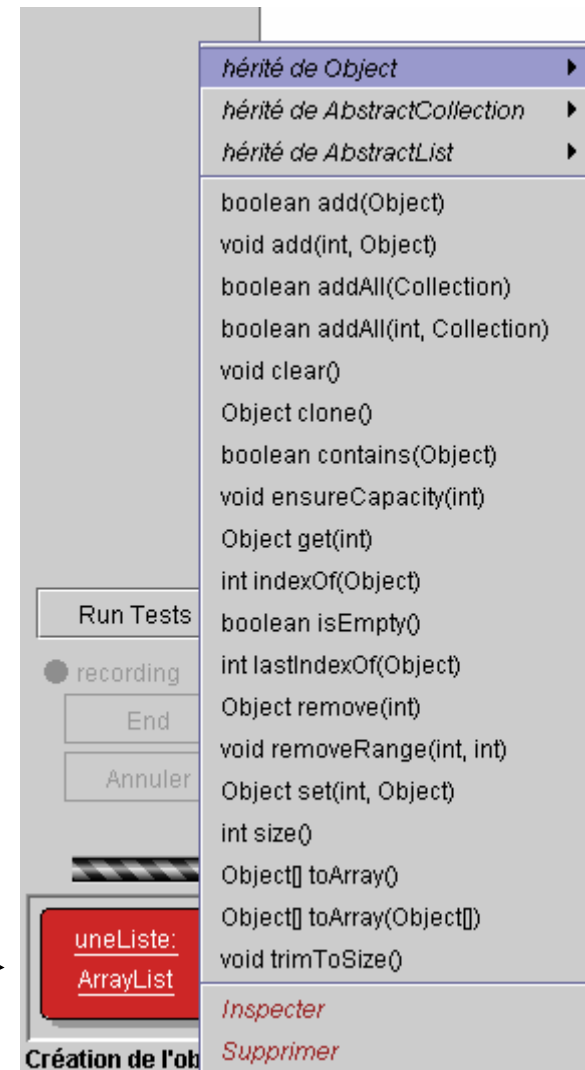
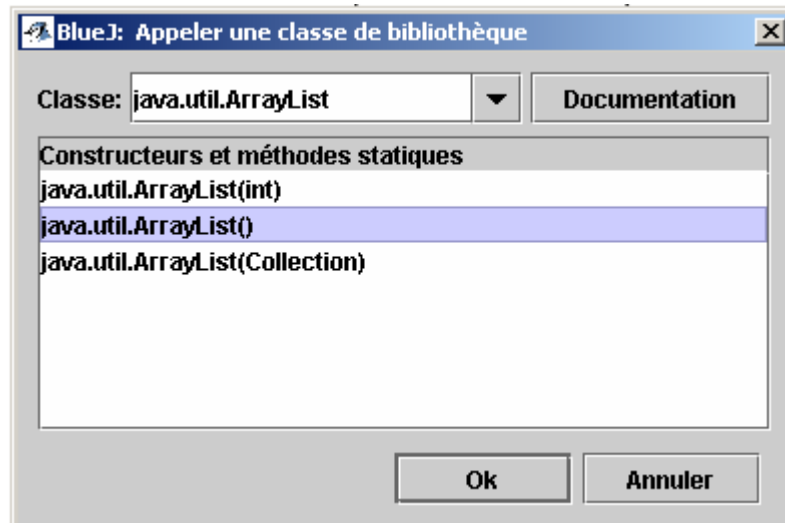
- Lire, relire un tutoriel
- Utiliser Bluej
 - Outils puis Bibliothèque de classe
- Les types primitifs sont-ils oubliés ?
 - <http://pcj.sourceforge.net/> Primitive Collections for Java. De Søren Bak
 - Depuis 1.5 voir également l'autoboxing
- Ce support est une première approche !
 - Collections : 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, extrait 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 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 ???");
}
```