Java, les objets : tout de suite !
Classes abstraites et interfaces

Cnam Paris
jean-michel Douin, douin@cnam.fr
Version du 3Avril 2003

Notes de cours associées au chapitre 10

Ce support accompagne, référence le livre de David J. Barnes & Michael Kölling
Objects First with Java A Practical Introduction using BlueJ
Pearson Education, 2003 ISBN 0-13-044929-6.

Sommaire

- **Classes abstraites**

  ou classes incomplètes

- **Interfaces**

  ou le “protocole” qu’une classe doit respecter

- **Un exemple complet : IntSet de Liskov**

  page 156, Program Development in Java

  Barbara Liskov Addison Wesley

  ou Le pattern décorateur

- **Les collections en Java en extra**

  Interface Collection

  La classe abstraite AbstractCollection

Classe abstraite ou incomplète

- **Syntaxe habituelle d 'une classe, pas de restrictions**
- **sauf que certaines implémentations de méthodes sont laissées à la responsabilité des sous-classes**

```
public abstract class Nom{
…
  public abstract void m();
}
```

- **note : pas de création d 'instance possible ...**

Interface

- **Interface comme types**

  précise le « type » d 'une classe, permet un test du bon type à l 'exécution

  exemple : l 'interface Cloneable

- **Interface comme spécification…**

```
public interface TableI{
  public void ajouter(Object o);
  public Object retirer(Object o);
 }
```

IntSet de liskov

- **Un ensemble d 'entiers, fonctionnalités minimales attendues**

```
insert(int x);
remove(int x);



boolean isIn(int x);
int choose() une exception si l 'ensemble est vide;
int size();
boolean subset(IntSet s);
```

Parcours, égalité et invariant

- Parcours de cet ensemble ?

  Iterator iterator();

```
package java.util;
public interface Iterator{
    public boolean hasNext();
    public Object next();
    public void remove();
}
```

- Egalité de deux ensembles ?

  boolean equals(Object o);

- Invariant de représentation ?

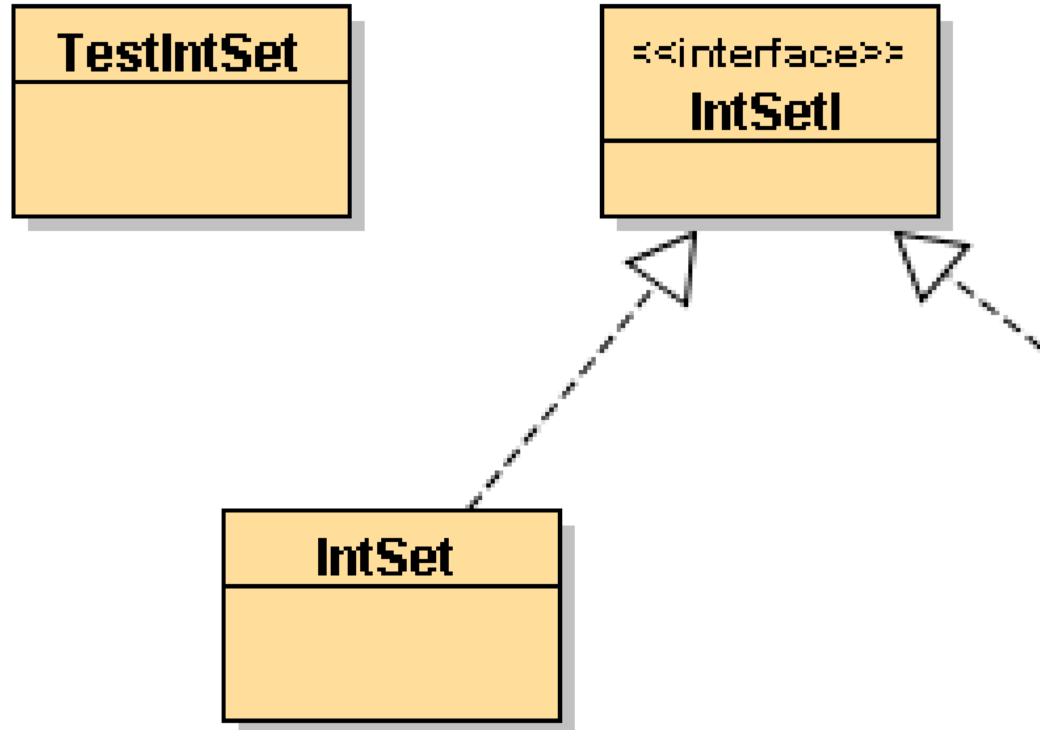
  Ou invariant de classe

  boolean repOk();

L 'interface IntSetI

```
import java.util.Iterator;
public interface IntSetI{

  public void insert(int x);
  public void remove(int x);

  public boolean isIn(int x);
  public int choose() throws EmptyException;
  public int size();
  public Iterator iterator();
  public boolean subset(IntSetI s);

  public boolean repOk();
}
```

Une implémentation possible : IntSet



La classe IntSet

```
import java.util.Iterator;
import java.util.Vector;
public class IntSet implements IntSetI {
  protected Vector els;

  public IntSet() { els = new Vector();}

  public boolean isIn(int x) {
    return els.indexOf(new Integer(x)) >= 0;
  }

  public void insert(int x) {
    if (!isIn(x)) els.add(new Integer(x));
  }

  public void remove(int x) {
     els.remove(new Integer(x));
  }
```

choose, size, subset et iterator

```
public int choose() throws EmptyException{
  if (size() == 0) throw new EmptyException();
  return ((Integer)els.lastElement()).intValue();
}

public int size() {return els.size(); }

public boolean subset(IntSetI s) {
  if (s == null)
    return false;
  int i = 0;
  while (i < els.size()) {
    if (!s.isIn(((Integer) els.get(i)).intValue()))
      return false;
    i++;
  }
  return true;
}

public Iterator iterator() {return els.iterator();}
```

equals, toString et repOk

```
 public boolean equals(Object o) {
  if (o instanceof IntSet) {
    IntSet s = (IntSet) o;
    return this.subset(s) && s.subset(this);
  }
  return false;
}

public String toString() {
  return this.els.toString().replace('[','{').replace(']','}');
}

 public boolean repOk() {
  if (els == null) return false;
  for (int i = 0; i < els.size(); i++) {
    Object x = els.get(i);
    if (!(x instanceof Integer))
      return false;
    for (int j = i + 1; j < els.size(); j++)
      if (x.equals(els.get(j)))
        return false;
  }
  return true;
}
```

Un extrait de la classe TestIntSet

```
import java.util.Iterator;

public class TestIntSet{
  public static void main(String[] args) throws IOException{
    IntSetI s1 = new IntSet(); assert(s1 != null);
    IntSetI s2 = new IntSet(); assert(s2 != null);

    s1.insert(4);s1.insert(3);s1.insert(2);s1.insert(2);
    s2.insert(2);s2.insert(4);s2.insert(3);assert(s2.size() == 3);
    s2.insert(9);s2.insert(11);s2.insert(2002);
    System.out.println("s1 : " + s1 + "s2 : " + s2);

    assert s1.repOk() && s2.repOk();
    System.out.println("s1.subset(s1) : " + s1.subset(s1));
    assert(s1.subset(s1) == true);
    System.out.println("s1.subset(s2) : " + s1.subset(s2));
    etc ...
```

Pause : Quelle conception de classes ?

- **Comment développer une série de tests unitaires de la classe IntSet ?**
- **Comment assurer la réutilisation de cette série de tests pour une autre classe ?**

  Une autre classe qui implémente la même interface
- **Comment ajouter une nouvelle série de tests ?**
- **Comment cumuler les séries de tests ?**

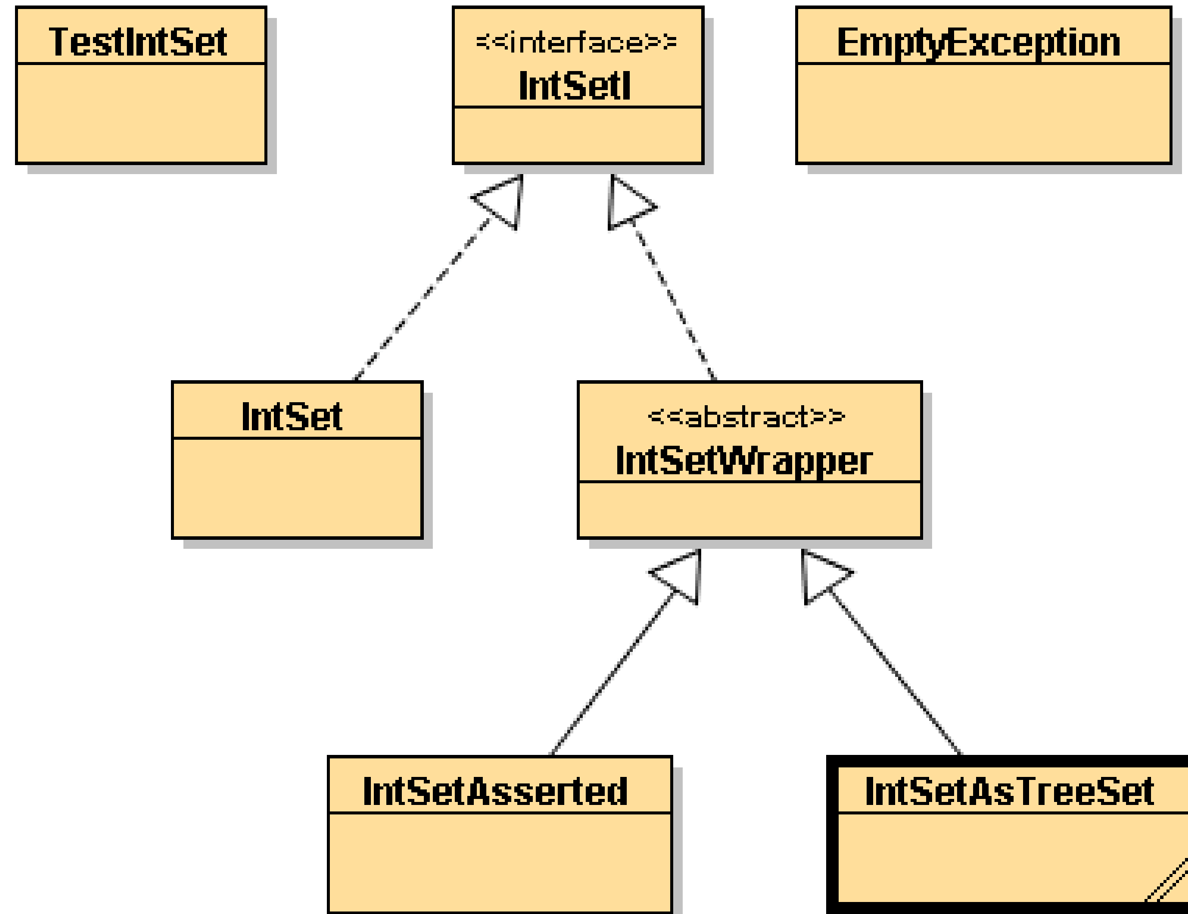
Une solution : le Pattern Décorateur

- **DECORATEUR, GoF page 203**

- **INTENTION**

  « Attache dynamiquement des responsabilités supplémentaires à un objet.
  Les décorateurs fournissent une alternative souple à la dérivation, pour étendre les fonctionnalités »

Pattern décorateur et IntSet



La classe abstraite IntSetWrapper

- **Cette classe implémente l 'interface IntSetI**
- **Contient une donnée d 'instance de type IntSetI et Propose un constructeur de ce type**

```
public abstract class IntSetWrapper implements IntSetI{
   protected IntSetI s;
   public IntSetWrapper(IntSetI s){this.s = s;}
```

- **L 'implémentation des méthodes est déléguée à l 'instance de type IntSetI**

```
  public void insert(int x){s.insert(x);}

  public void remove(int x){s.remove(x);}
```

La classe IntSetWrapper

```
import java.util.Iterator;
public abstract class IntSetWrapper implements IntSetI{
    protected IntSetI s;
    public IntSetWrapper(IntSetI s){this.s = s;}

    public void insert(int x){s.insert(x);}

    public void remove(int x){s.remove(x);}

    public boolean isIn(int x){return s.isIn(x);}

    public int choose() throws EmptyException{ return s.choose();}

    public int size(){ return s.size();}

    public Iterator iterator(){return s.iterator();}

    public boolean subset(IntSetI s){return s.subset(s);}
    public boolean repOk(){return s.repOk();}
}
```

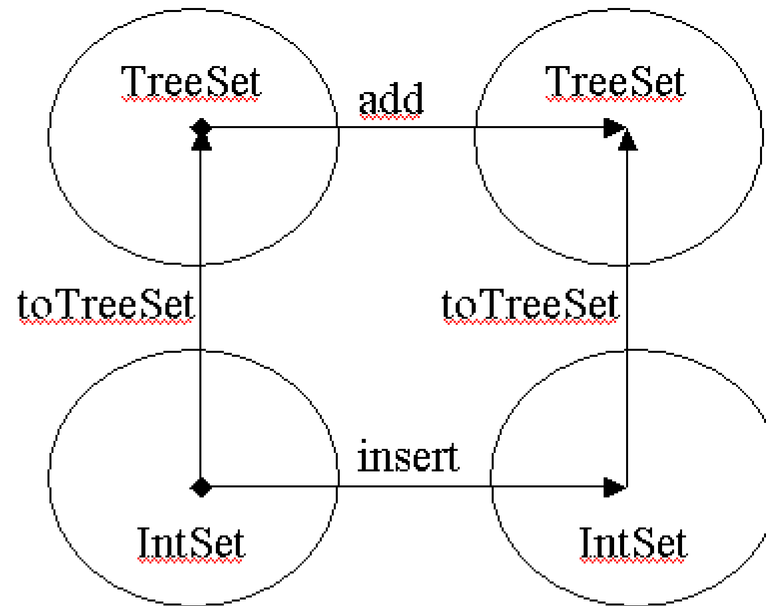
La classe IntSetAsserted

```
public class IntSetAsserted extends IntSetWrapper{

  public IntSetAsserted(IntSetI set){ super(set);}

  public void insert(int x){
    assert super.repOk();
    boolean present = super.isIn(x);
    int taille = super.size();
    super.insert(x);
    assert present && taille == super.size() ||
           !present && taille+1 == super.size();
    assert super.repOk();
  }
etc ...
```

Un premier test …

```
IntSetI s4 = new IntSetAsserted(new IntSet());

s4.insert(4);

 etc...
```

Comment comparer deux implémentations ?



toTreeSet(s).add(e) ‘==’ toTreeSet(s.insert(e))

s est une instance de la classe IntSet, **ts** une instance de la classe TreeSet et e est une instance de la classe Integer.

La classe IntSetAsTreeSet

```
public class IntSetAsTreeSet extends IntSetWrapper{

   public IntSetAsTreeSet(IntSetI s){
     super(s);
   }

   public static TreeSet toTreeSet( IntSetI s) {
     TreeSet ts = new TreeSet();
     for(Iterator it = s.iterator();it.hasNext();){
       ts.add(it.next());
     }
     return ts;
   }

   public void insert( int x){
     assert s.repOk();
     TreeSet ts = toTreeSet(s);
     super.insert(x);
     ts.add(new Integer(x));
     assert s.repOk();
     assert toTreeSet(s).equals(ts);
   }
```

Un deuxième test …

```
IntSetI s5 = new IntSetAsTreeSet(new IntSet());
s5.insert(4);
 etc...
```

Un troisième test = premier(deuxième(original))

```
IntSetI s6 = new IntSetAsserted(
                new IntSetAsTreeSet(new IntSet())
             );
s6.insert(4);

etc...
```

imbrication des tests …

voir www.junit.org

Voir une utilisation du décorateur : InputStream, et FilterInputStream

Un quatrième test = deuxième(premier(original))

```
IntSetI s6 = new IntSetAsTreeSet (
                new IntSetAsserted(new IntSet())
             );
s6.insert(4);

etc...
```

Seconde Partie en extra

- **Les collections en Java**

  Interface, classes abstraites un résumé

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/

Pourquoi ?

- **Organisation des données**

  Listes, tables, sacs, arbres, …
  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 …

Sommaire Collections en Java

- ***Quelles fonctionnalités ?***
- ***Quelles implémentations partielles ?***
- ***Quelles implémentations complètes ?***
- ***Quels algorithmes ?***

Les Collections en Java, paquetage java.util

- ***Quelles fonctionnalités ?***

  Quelles interfaces ?

  Collection, Set, SortedSet, List, Map, SortedMap,Comparator, Comparable

- ***Quelles implémentations partielles ?***

  Quelles classes abstraites ou incomplètes ?

  AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList, AbstractMap

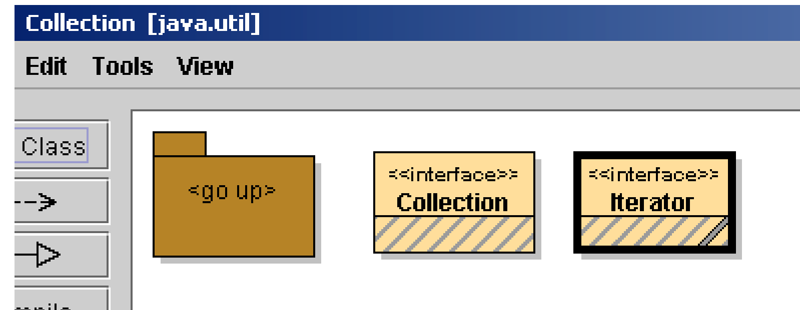
- ***Quelles implémentations complètes ?***

  Quelles classes concrètes toutes prêtes ?

  HashSet, TreeSet, LinkedList,ArrayList, WeakHashMap, HashMap, TreeMap

- ***Quels algorithmes ?***

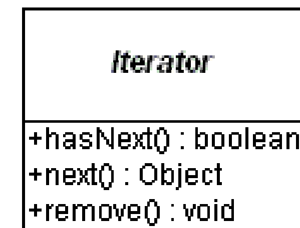
Interface java.util.Collection



2 méthodes fondamentales

- boolean add(Object obj);
- Iterator iterator();

```
public interface Iterator{
  Object next();
  boolean hasNext();
  void remove();
}
```



java.util.Iterator

```
Collection collection = ...;
Iterator iterator = collection.iterator();
while (iterator.hasNext()) {
        Object element = iterator.next();
        if (condition(element)) {
            iterator.remove();
        }
}
```

Attention :

remove engendre de nouvelles contraintes

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

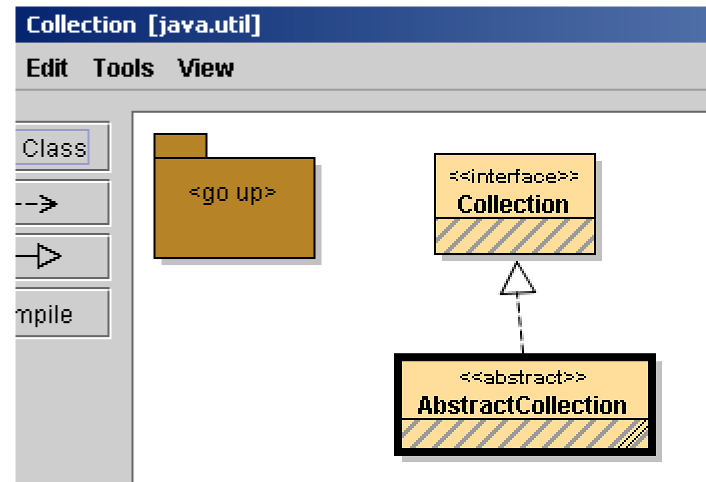
Du bon usage de Iterator

```
Collection collection = ...;

Iterator it = collection.iterator();
it.next();
it.remove();
it.remove();// → throw new IllegalStateException()

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

Java.util.Collection

Collection
+add(element : Object) : boolean +addAll(collection : Collection) : boolean +clear() : void +contains(element : Object) : boolean +containsAll(collection : Collection) : boolean +equals(object : Object) : boolean +hashCode() : int +iterator() : Iterator +remove(element : Object) : boolean +removeAll(collection : Collection) : boolean +retainAll(collection : Collection) : boolean +size() : int +toArray() : Object[ ] +toArray(array : Object[ ]) : Object[ ]

Première implémentation incomplète



- **La classe abstraite AbstractCollection**

  Les méthodes :

```
boolean add(Object obj);
Iterator iterator();
```

sont laissées à la responsabilité des sous classes

AbstractCollection, implémentations de containsAll

```
public boolean containsAll(Collection c) {
  Iterator e = c.iterator();
  while (e.hasNext())
     if(!contains(e.next()))
       return false;

   return true;
 }
```

AbstractCollection : la méthode contains

```
public boolean contains(Object o) {
  Iterator e = iterator();
  if (o==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 removeAll

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

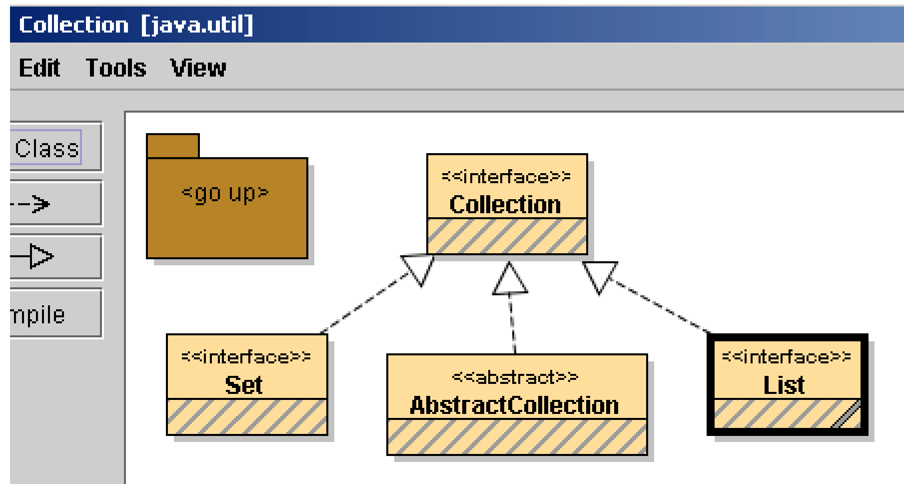
AbstractCollection : la méthode remove

```
public boolean remove(Object o) {
  Iterator e = 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 = iterator();
  while (e.hasNext()) {
     if(!c.contains(e.next())) {
      e.remove();
      modified = true;
     }
  }
  return modified;
 }
```

Interface Set et List



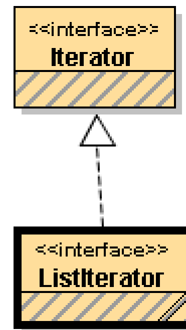
Set

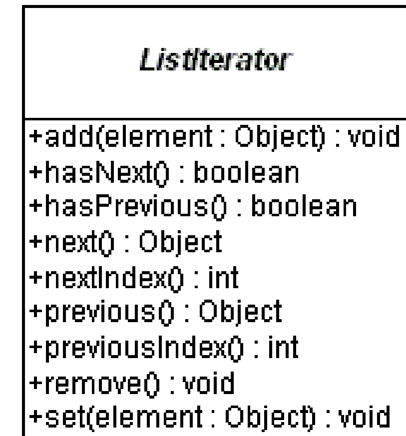
+add(element : Object) : boolean
+addAll(collection : Collection) : boolean
+clear() : void
+contains(element : Object) : boolean
+containsAll(collection : Collection) : boolean
+equals(object : Object) : boolean
+hashCode() : int
+iterator() : Iterator
+remove(element : Object) : boolean
+removeAll(collection : Collection) : boolean
+retainAll(collection : Collection) : boolean
+size() : int
+toArray() : Object[]
+toArray(array : Object[]) : Object[]

List

+add(element : Object) : boolean
+add(index : int, element : Object) : void
+addAll(collection : Collection) : boolean
+addAll(index : int, collection : Collection) : boolean
+clear() : void
+contains(element : Object) : boolean
+containsAll(collection : Collection) : boolean
+equals(object : Object) : boolean
+get(index : int) : Object
+hashCode() : int
+indexOf(element : Object) : int
+iterator() : Iterator
+lastIndexOf(element : Object) : int
+listIterator() : ListIterator
+listIterator(startIndex : int) : ListIterator
+remove(element : Object) : boolean
+remove(index : int) : Object
+removeAll(collection : Collection) : boolean
+retainAll(collection : Collection) : boolean
+set(index : int, element : Object) : Object
+size() : int
+subList(fromIndex : int, toIndex : int) : List
+toArray() : Object[]
+toArray(array : Object[]) : Object[]

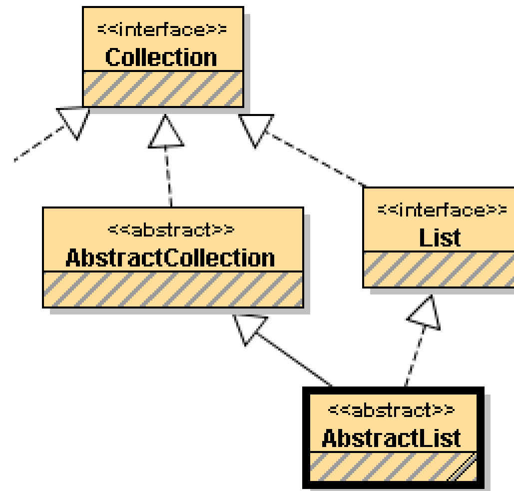
Iterator extends ListIterator





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

AbstractList



- **AbtractList et AbstractCollection   Même principe**
  - **add**
  - **ListIterator iterator**

AbstractList : la méthode indexOf

```
public int indexOf(Object o) {
  ListIterator 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 : Ll méthode lastIndexOf

```
public int lastIndexOf(Object o) {
  ListIterator 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;
 }
```

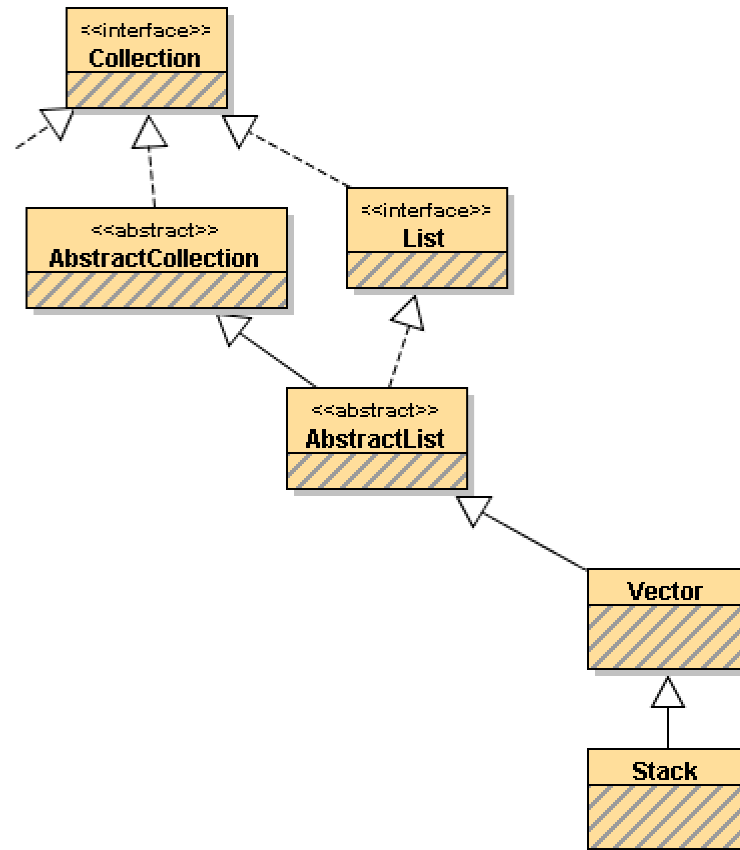
AbstractList : la méthode iterator et la classe interne

```
public Iterator iterator() {return listIterator(0);}

public ListIterator listIterator(final int index) {
    checkForComodification();
    if (index<0 || index>size)
        throw new IndexOutOfBoundsException(
            "Index: "+index+", Size: "+size);
    return new ListIterator() {
        private ListIterator i = l.listIterator(index+offset);

        public boolean hasNext() { return nextIndex() < size;}

        public Object next() {
            if (hasNext())return i.next();
            else throw new NoSuchElementException();
        }
        public boolean hasPrevious() { return previousIndex() >= 0;}
```

La classe interne , suite

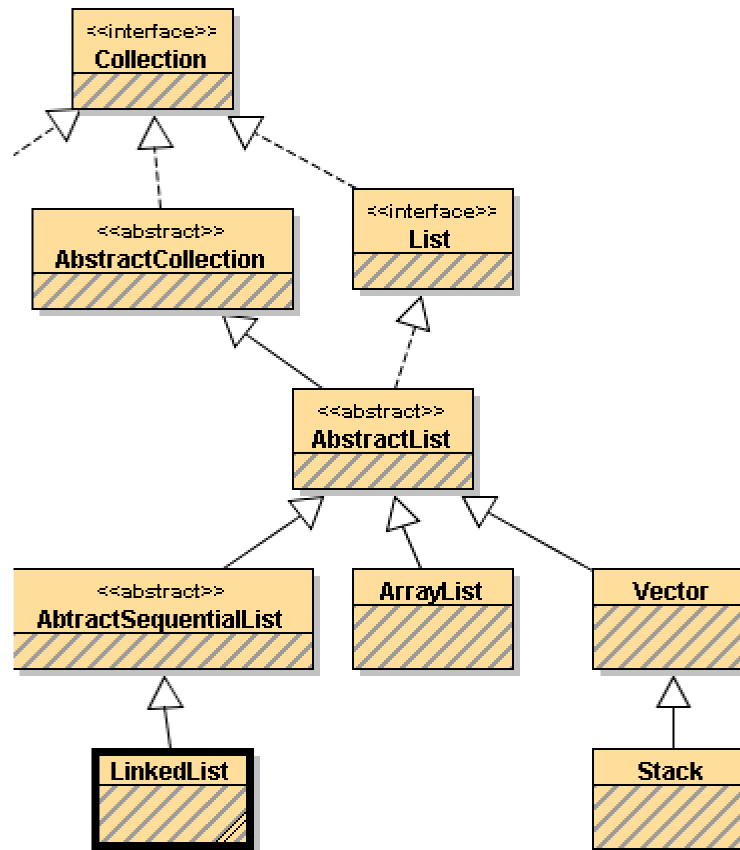
```
public Object previous() {
  if (hasPrevious()) return i.previous();
    else throw new NoSuchElementException();
}

public int nextIndex() { return i.nextIndex() - offset;}

public int previousIndex() { return i.previousIndex() - offset;}

public void remove() {
  i.remove();
  expectedModCount = l.modCount;
  size--;
  modCount++;
}

public void set(Object o) { i.set(o);}

public void add(Object o) {
  i.add(o);
  expectedModCount = l.modCount;
  size++;
  modCount++;
}
};}
```

Les biens connues et concrètes Vector et Stack



Stack hérite Vector hérite de AbstractList !

Les classes concrètes ArrayList et LinkedList



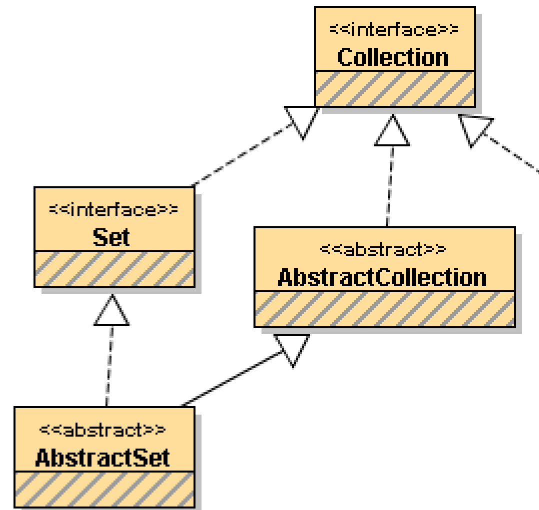
LinkedList
+addFirst(element : Object) : void +addLast(element : Object) : void +getFirst() : Object +getLast() : Object +removeFirst() : Object +removeLast() : Object

ArrayList, LinkedList : enfin un exemple concret

```
import java.util.*;
public class ListExample {
    public static void main(String args[]) {
      List list = new ArrayList();
      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 queue = new LinkedList();
      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



Set
+add(element : Object) : boolean
+addAll(collection : Collection) : boolean
+clear() : void
+contains(element : Object) : boolean
+containsAll(collection : Collection) : boolean
+equals(object : Object) : boolean
+hashCode() : int
+iterator() : Iterator
+remove(element : Object) : boolean
+removeAll(collection : Collection) : boolean
+retainAll(collection : Collection) : boolean
+size() : int
+toArray() : Object[ ]
+toArray(array : Object[ ]) : Object[ ]

AbstractSet : la méthode equals

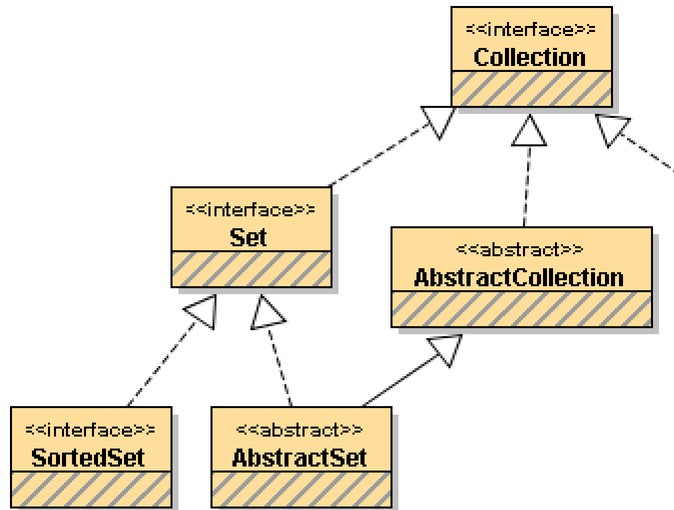
```
public boolean equals(Object o) {
  if (o == this)
     return true;

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

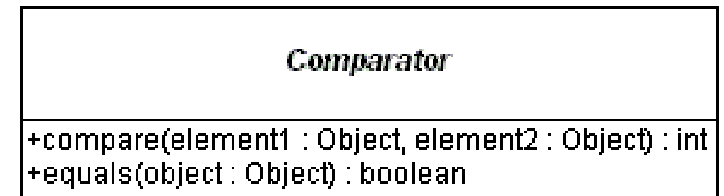
AbstractSet : la méthode hashCode

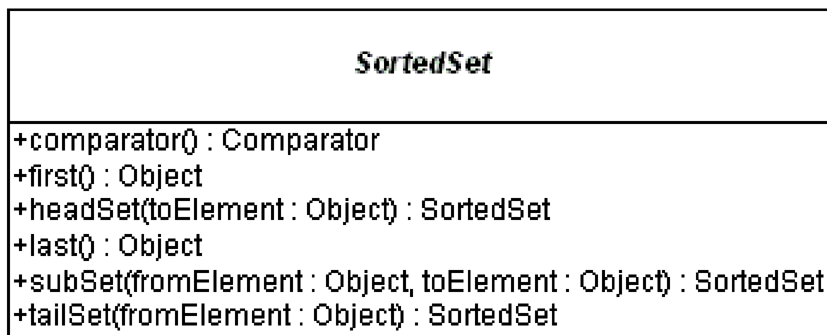
```
public int hashCode() {
  int h = 0;
  Iterator i = iterator();
  while (i.hasNext()) {
     Object obj = i.next();
        if (obj != null)
           h = h + obj.hashCode();
     }
  return h;
  }
```

La somme de la valeur hashCode de chaque élément

L'interface SortedSet





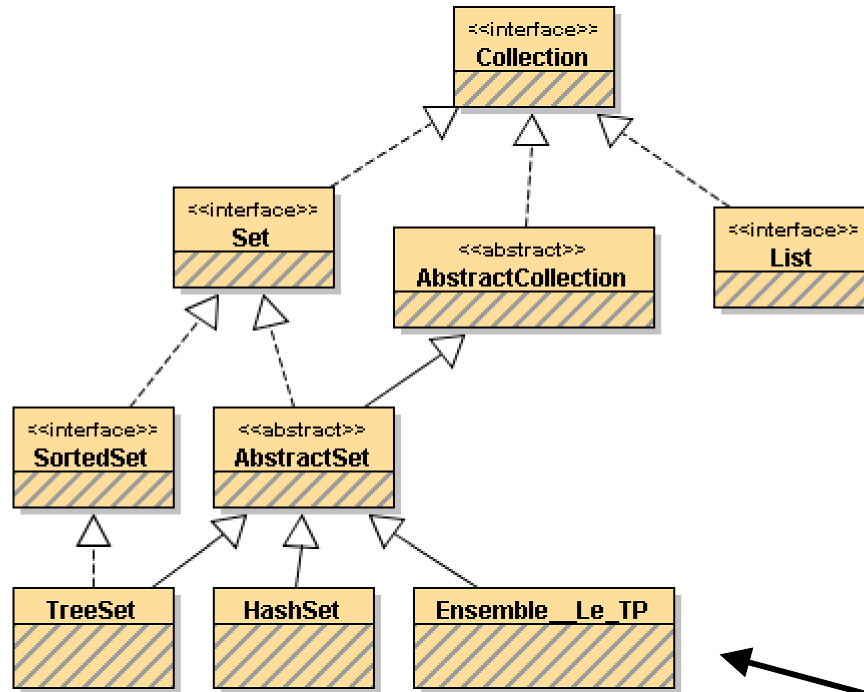


Ordre et relation

- **Interface Comparator**
  Relation d’ordre de la structure de données

- **Interface Comparable**
  Relation d’ordre entre chaque élément

Les concrètes en fin



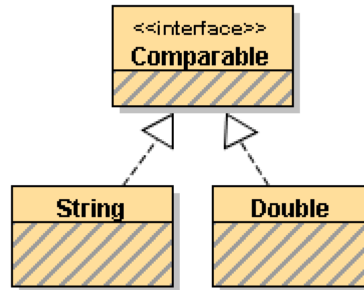
http://jfod.cnam.fr/tp_cdi/Tp5/Tp5.html

Les concrètes : un exemple

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


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

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

Les concrètes bien connues



Extrait de le documentation du j2sdk1.4

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{
  private int i;
  public Entier(int i){ this.i = i;}

  public int compareTo(Object o){
    if (o instanceof Entier)
      if (i < ((Entier)o).intValue()) return -1;
      else if (i == ((Entier)o).intValue()) return 0;
           else return 1;
    else throw new ClassCastException();
}

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

La relation d'ordre de la structure

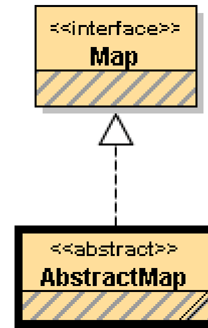
```
public class Croissant implements Comparator{
public int compare(Object o1, Object o2){
  if (o1 instanceof Comparable && o2 instanceof Comparable)
     return ((Comparable)o1).compareTo(o2);
   else throw new ClassCastException();
}
}

public class Decroissant implements Comparator{
public int compare(Object o1, Object o2){
  if (o1 instanceof Comparable && o2 instanceof Comparable)
      return -((Comparable)o1).compareTo(o2);
   else throw new ClassCastException();
}
}
```

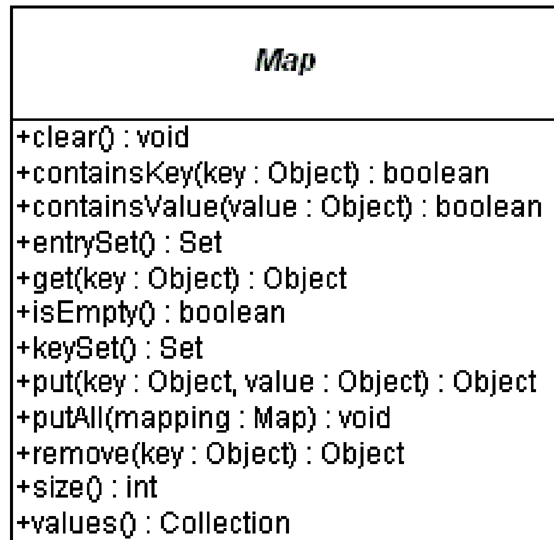
Le test

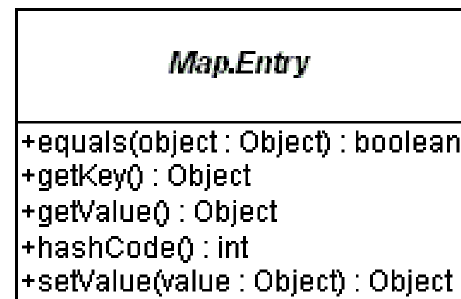
```
public static void main(String[] args) {
  SortedSet e = new TreeSet(new Croissant());

   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  e1 = new TreeSet(new Decroissant());
   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]
```

Adressage associatif, Hashtable



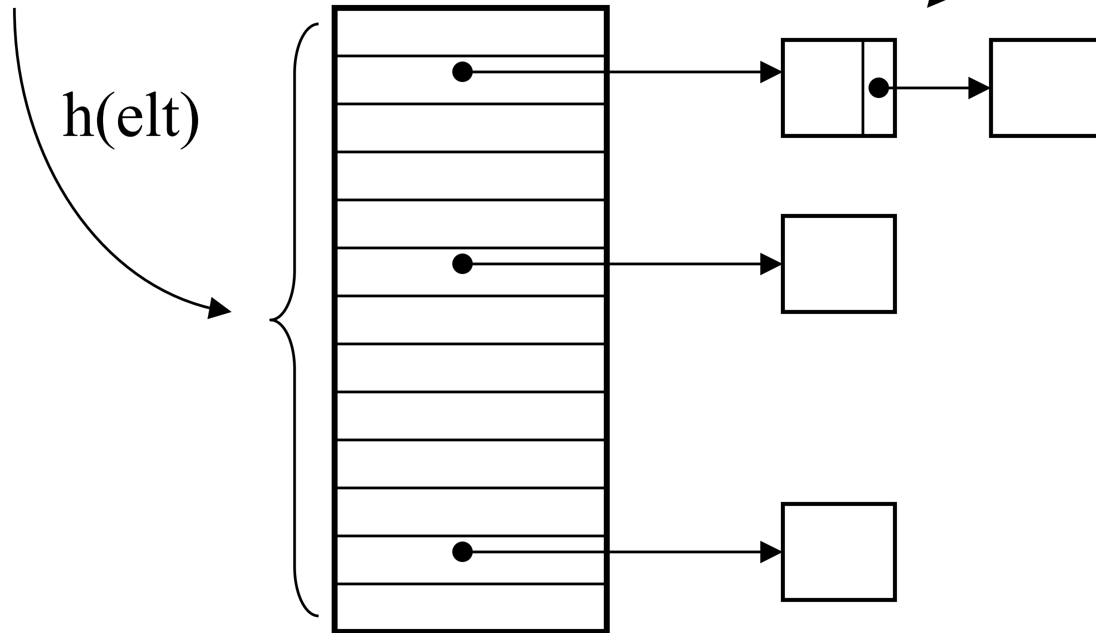




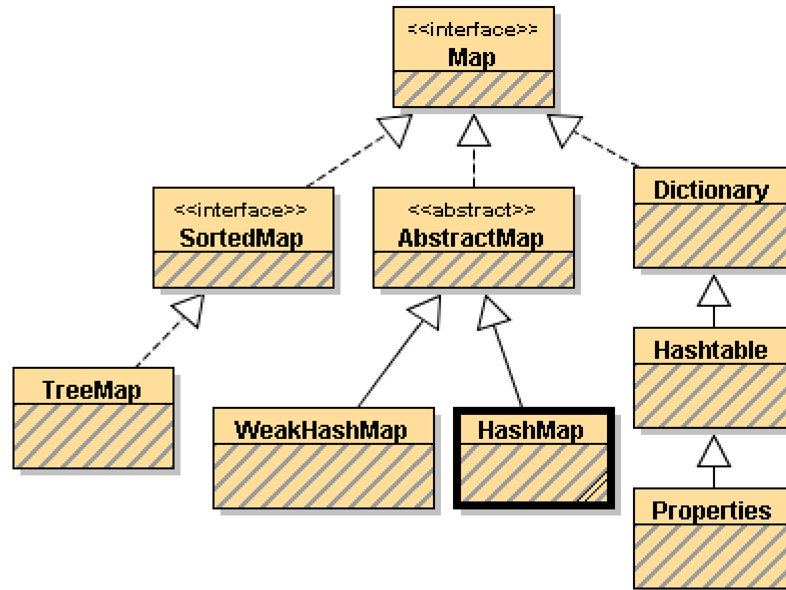
Une table de hachage

Gestion des collisions avec une liste

Fonction de hachage



Au complet



Les biens connues Dictionary, Hashtable, Properties

Un exemple : fréquence des éléments de args …

```
import java.util.*;
public class MapExample {
  public static void main(String args[]) {
   Map map = new HashMap();
   Integer ONE = new Integer(1);
   for (int i=0, n=args.length; i<n; i++) {
       String key = args[i];
       Integer frequency = (Integer)map.get(key);
       if (frequency == null) {
           frequency = ONE;
       } else {
           int value = frequency.intValue();
           frequency = new Integer(value + 1);
       }
       map.put(key, frequency);
   }

   System.out.println(map);
   Map sortedMap = new TreeMap(map);
   System.out.println(sortedMap);
} }
```

La classe Collections

- **Class Collections {**

  // Read only : unmodifiable*Interface*

  Collection unmodifiableCollection(Collection collection)

  List unmodifiableList(List list)

  …

  // Thread safe : synchronized*Interface*

  Collection synchronizedCollection(Collection collection)

  List synchronizedList(List list)

  // Singleton

  // Multiple copy

  // tri

  public static void sort(List list, Comparator c)

La méthode Collections.sort

```
 public static void sort(List list,Comparator c){
  Object a[] = list.toArray();
  Arrays.sort(a, 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);
```

Résumé, synthèse