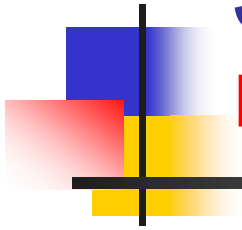
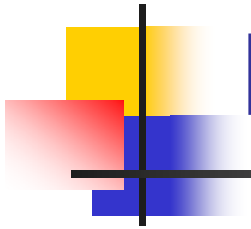


Java

Licence Professionnelle CISI, 2009-2010

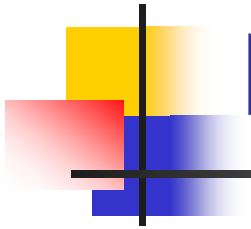
Cours 8 : Les collections

Inspiré du livre de Claude Delannoy :
Programmer en Java, Ed. Eyrolles



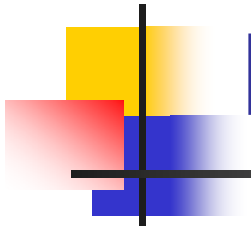
Les collections

- Qu’est-ce que c’est ?
  - Une collection représente un groupe d'objets, connu par ses éléments
  - Certaines collections acceptent les doublons, d'autres pas
  - Certaines sont ordonnées (listes chaînées, vecteurs dynamiques), d'autres pas (comme les ensembles)
  - Dans les collections ordonnées, on pourra à tout moment parler
    - du premier élément
    - du dernier élément
    - du ième élément



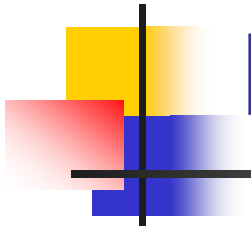
Les collections

- Utilisation de la méthode compareTo
  - Plusieurs collections nécessitent de comparer leurs éléments
  - Pour cela elles ont besoin de redéfinir la méthode compareTo de l'interface Comparable
    - Public int compareTo(Object o)
  - Celle-ci doit comparer l'objet courant this à l'objet o et renvoyer un entier : nul (en cas d'égalité), négatif (si l'objet courant est inférieur à o) et supérieur à zéro si l'objet courant est supérieur à o



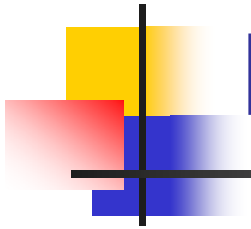
Les collections

- Utilisation de la méthode compareTo
  - Remarques
    - Notez bien le type Object de o. Dans le corps de la méthode, on sera souvent amené à le convertir dans un type précis
    - Faites bien attention que la méthode compareTo définisse convenablement une relation d'ordre
    - Si on oublie d'indiquer que la classe de vos éléments implémente l'interface Comparable, leur méthode compareTo ne sera pas appelée car les méthodes comparent des objets de type Comparable



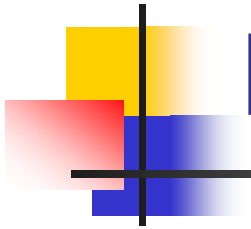
Les collections

- Utilisation d'un objet comparateur
  - Il se peut que la démarche précédente (utilisation de compareTo) ne convienne pas
  - Ce sera notamment le cas lorsque :
    - Les éléments sont des objets d'une classe existante qui n'implémente pas l'interface Comparable
    - On a besoin de définir plusieurs ordres sur une même collection
  - Il est alors possible de définir l'ordre souhaité, non plus dans la classe des éléments, mais :
    - soit lors de la construction de la collection
    - soit lors de l'appel d'un algorithme
  - Pour ce faire, on fournit en argument (du constructeur ou de l'algorithme) un objet qu'on nomme un comparateur



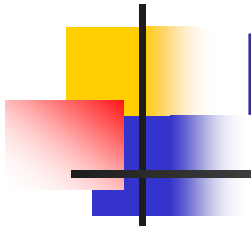
Les collections

- Utilisation d'un objet comparateur (suite)
  - Il s'agit en fait d'un objet d'un type implémentant l'interface Comparator qui comporte une seule méthode
    - Public int compare (Object o1, object o2)
  - Celle-ci doit donc cette fois-ci comparer les objets o1 et o2 reçus en argument et renvoyer un entier : négatif si o1 est inférieur à o2…



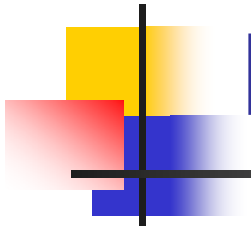
Les collections

- Égalité d'éléments d'une collection
  - Toutes les collections nécessitent de définir l'égalité de deux éléments
  - Ce besoin est évident dans le cas des ensembles HashSet et TreeSet dans lesquels un même élément ne peut apparaître qu'une seule fois
    - ➔ il faut donc pouvoir le tester
  - Cela existe également pour d'autres collections qui emploient la méthode remove qui supprime un objet d'une valeur donnée
    - ➔ cette valeur nécessite d'être testée par l'égalité avec le paramètre d'entrée



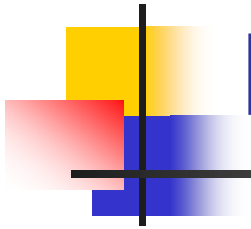
Les collections

- Égalité d'éléments d'une collection (suite)
  - Cette égalité est définie en recourant à la méthode equals de l'objet
  - Ainsi, là encore, pour des méthodes de type String, File…, les choses seront naturelles puisque la méthode equals se base réellement sur la valeur des objets
  - En revanche, pour les autres, il faut se souvenir que, par défaut, leur méthode equals est celle héritée de la classe Object
  - Elle se base simplement sur les références : deux objets différents apparaîtront toujours comme non égaux (même s'ils contiennent exactement les mêmes valeurs)
    - Pour obtenir un comportement plus satisfaisant, il faudra alors redéfinir la méthode equals de façon appropriée



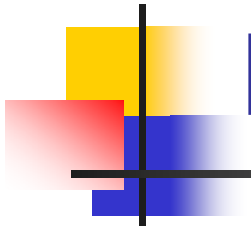
Les collections

- Égalité d'éléments d'une collection (suite)
  - Remarque
    - En pratique, on peut être amené à définir dans une même classe les méthodes compareTo et equals
    - Il faut alors naturellement prendre garde à ce qu'elles soient compatibles entre elles
    - Notamment, il est nécessaire que compareTo fournisse 0 si et seulement si equals fournit true



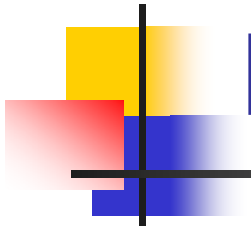
Les collections

- Les itérateurs et leurs méthodes
  - Les itérateurs sont des objets qui permettent de « parcourir » un par un les différents éléments d'une collection
  - Ils ressemblent à des pointeurs (tels que ceux de C ou de C++) sans en avoir exactement les mêmes propriétés
  - Il existe deux types d'itérateurs
    - Monodirectionnels
      - Le parcours de la collection se fait d'un début vers une fin; on ne passe qu'une seule fois sur chacun des éléments
    - Bidirectionnels
      - La parcours peut se faire dans les deux sens ; on peut avancer et reculer à sa guise dans la collection



Les itérateurs

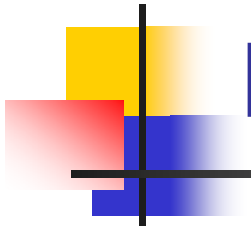
- Les itérateurs monodirectionnels
  - Chaque classe collection dispose d'une méthode nommée iterator fournissant un itérateur monodirectionnel, c.à.d un objet d'une classe implémentant l'interface Iterator
  - Associé à une collection donnée, il possède les propriétés suivantes :
    - Il désigne la position de l'élément courant
    - On peut obtenir l'objet désigné par un itérateur en appelant la méthode next de l'itérateur, ce qui, en outre, avance l'itérateur d'une position
    - La méthode hasNext de l'itérateur permet de savoir si l'itérateur est ou non en fin de collection



Les itérateurs

- Les itérateurs monodirectionnels
  - Canevas de parcours d'une collection
    - On pourra parcourir tous les éléments d'une collection c, en appliquant ce canevas

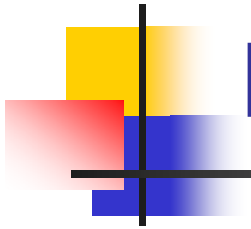
```
Iterator iter = c.iterator(); //la méthode iterator renvoie un objet
                              //implémentant l'interface Iterator et
                              //désignant initialement le premier
                              //élément s'il existe
while (iter.hasNext())
{
    Object o = iter.next();  //fournit l'élément désigné par
                             //l'itérateur et avance l'itérateur à la
                             //position suivante. En général, on convertira
                             //o dans le type effectif des éléments et on
                             //utilisera cet objet courant
}
```



Les itérateurs mondirectionnels

- Remarques
  1. La classe Iterator dispose d'un constructeur recevant un argument entier représentant une position dans la collection
     - Par exemple si c est une collection, l'instruction :

       ListeIterator it = c.listeIterator(5);
     - Crée l'itérateur it et l'initialise à ce qu'il désigne le sixième élément de la collection (le premier élément portant le numéro 0)
  2. Si l'on souhaite parcourir plusieurs fois une même collection, il suffit de réinitialiser l'itérateur en appelant à nouveau la méthode listeIterator

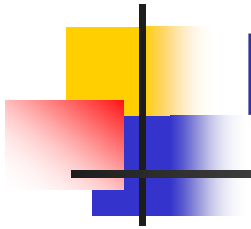


Les itérateurs mondirectionnels

- La méthode remove de l'interface Iterator
  - L'interface Iterator prévoit la méthode remove qui supprime de la collection, le dernier objet renvoyé par next
  - Exemple :
    - supprimer de la collection c tous les objets vérifiant une condition

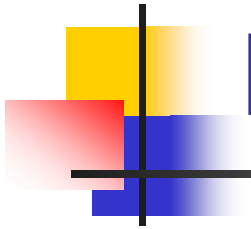
```
Iterator iter = c.iterator();
while (c.iter.hasNext())
{
    Object o = iter.next();
    if (condition) iter.remove();
}
```

  - Notez bien que remove ne travaille pas directement avec la position courante de l'itérateur mais bien avec la dernière référence envoyée par next



Les itérateurs bidirectionnels

- Les itérateurs bidirectionnels : l'interface ListeIterator
  - Certaines collections (listes chaînées, vecteurs dynamiques) peuvent, par nature, être parcourues dans les deux sens
  - Elles disposent d'une méthode nommée listIterator qui fournit un itérateur bidirectionnel
  - Il dispose bien sûr des méthodes : next, hasNext, et remove héritées de Iterator
  - Mais dispose d'autres méthodes comme
    - previous et hasPrevious, complémentaires de next et hastNext
    - add et set



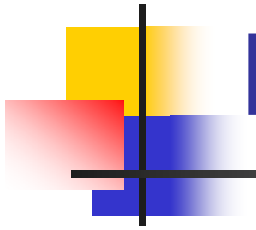
Les itérateurs bidirectionnels

- L'interface ListeIterator
  - Exemple d'utilisation

```
iter = l.listeIterator(l.size); // position courante : fin de liste
while(iter.hasPrevious())
{
    Object o = iter.previous()
    //utilisation de l'objet courant o
}
```

  - Autre exemple d'utilisation

```
iter = l.listeIterator(l.size);
Object elem;
while(iter.hasNext())
{
    elem = iter.next();
    elem= iter.previous(); // annule l'action précédente
}
```

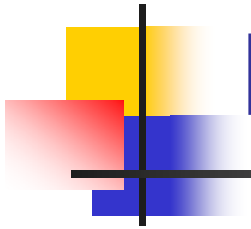


Les itérateurs bidirectionnels

- Méthode add
  - L'interface ListeIterator prévoit une méthode add qui ajoute un élément à la position courante de l'itérateur
  - Si ce dernier se trouve en fin de collection, l'ajout se fait tout naturellement en fin de collection
- Exemple

```
ListeIterator it = c.listeIterator();
it.next(); //premier élément = élément courant
it.next(); //deuxième élément = élément courant
it.add(item); //ajoute elem à la position courante, c.à.d
   entre le premier et le deuxième élément
```

- Remarque
  - add déplace la position courante



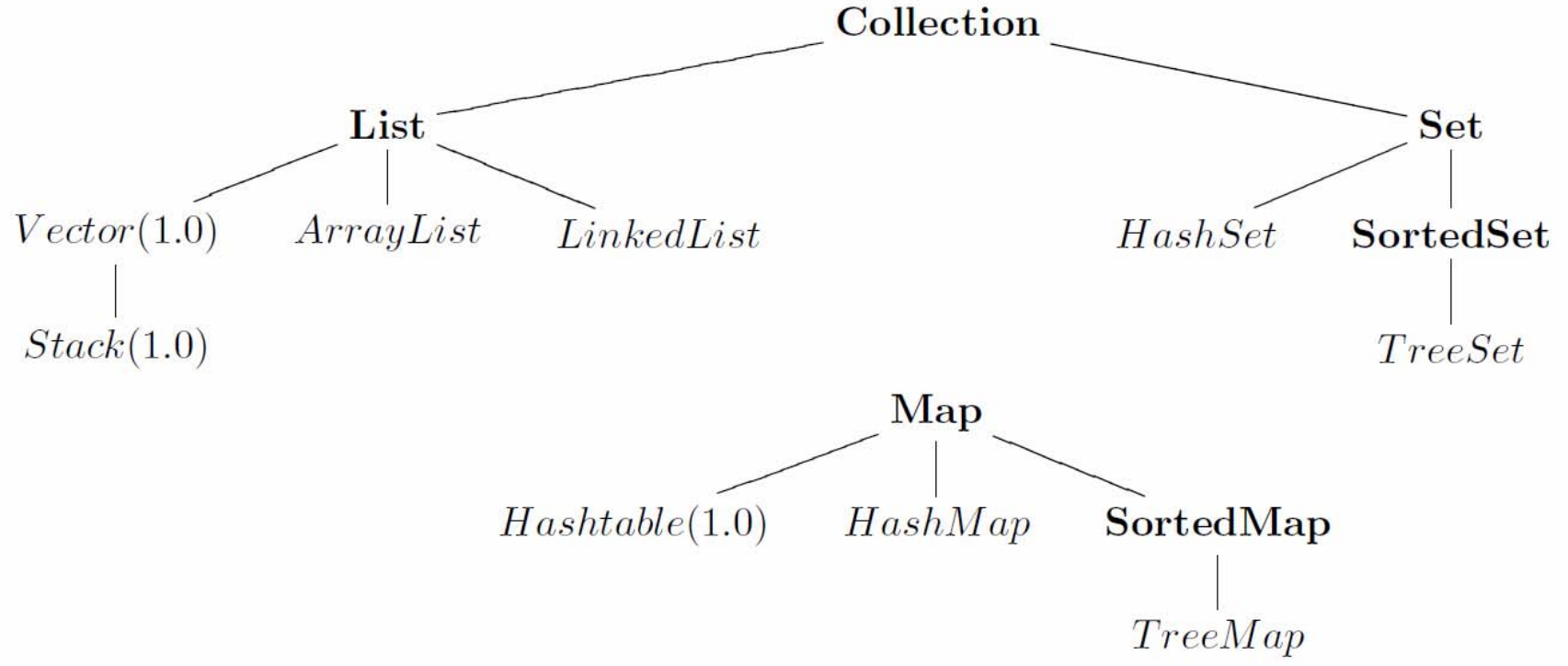
Les itérateurs bidirectionnels

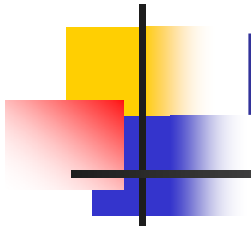
- Méthode set(elem)
  - Remplace par elem l'élément courant
  - La position courante n'est pas modifiée
- Exemple :
  - Remplacement par null tous les éléments d'une collection c vérifiant une condition

```
ListeIterator it = c.listeIterator();
while(it.hasNext())
{
    Object o = it.next();
     if(condition) it.set(null);
}
```

Les collections

- Opérations communes à toutes les collections
  - Les collections étudiées ici implémentent toutes l'interface Collection. Elle appartient au package java.util :

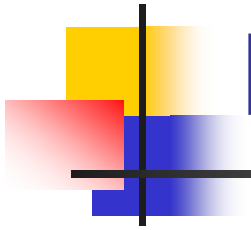




Les collections

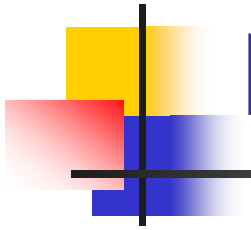
- Opérations communes à toutes les collections
  1. Toute classe collection C dispose d'un constructeur sans argument

     C c = new C(); // crée une collection vide
  2. Toute collection dispose des méthodes suivantes
     - add(elem) //fournit true lorsque l'ajout s'est bien réalisé
     - remove(elem) // supprime l'élément elem en fournissant la valeur true
     - addAll(ca) //ajoute à la collection c tous les éléments de la collection ca
     - removeAll(ca) //supprime de la collection c tout élément apparaissant égal à un des éléments de la collection ca
     - retainAll(ca) // supprime de la collection c tout élément qui n'apparaît pas égal à un des éléments de la collection ca (on ne conserve donc dans c que les éléments présents dans ca)



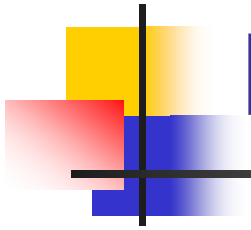
Les collections

3. Autres méthodes
   - size //fournit la taille d'une collection
   - isEmpty //teste si elle est vide
   - clear //supprime tous les éléments
   - contains(elem) //permet de savoir si la collection contient un élément
   - toString () //pour décrire le contenu d'une collection



Les collections

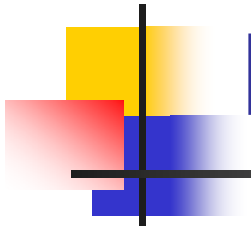
- Les listes chaînées : classe LinkedList
  - La classe LinkedList permet de manipuler des listes « doublement chaînées »
  - A chaque élément de la collection, on associe (de façon totalement transparente pour le programmeur) deux informations supplémentaires qui ne sont autres que les références à l'élément précédent et au suivant
  - Une telle collection peut ainsi être parcourue à l'aide d'un itérateur bidirectionnel de type ListIterator



LinkedList

- Opérations usuelles
  - Construction et parcours
    - LinkedList l1 = new LinkedList(); //crée une liste vide
    - LinkedList l2 = new LinkedList(c); //crée une liste formée de tous les éléments de la collection c
    - On peut utiliser toutes les méthodes vues sur les collections
      - next, previous, hasNext, hasPrevious
  - Ajout d'un élément

```
LinkedList l;
…
ListIerator iter = l.listIterator(); //iter désigne initialement le début de la liste
iter.add(item);
```



LinkedList

- Ajout d'un élément : autre exemple

```
while(iter.hasNext())
{…
   if(…)l.add(elem); //déconseillé : itérateur en cours d'utilisation
   else l.addFirst(elem); //idem
   iter.next();
}
```

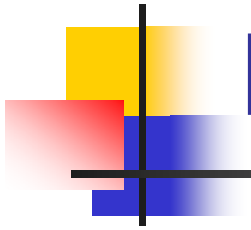
- Suppression d'un élément
  - méthode ListIterator :
    - remove //qui supprime le dernier élément renvoyé soit par next, soit par previous
  - Méthodes spécifiques
    - RemoveFirst, removeLast

- Exemple 1 : illustrant ListIterator

```
import java.util.* ;
public class Liste1
{ public static void main (String args[])
  { LinkedList l = new LinkedList() ;

    System.out.print ("Liste en A : ") ;
     affiche (l);
    l.add ("a") ; l.add ("b") ;  // ajouts en fin
     de liste
    System.out.print ("Liste en B : ") ;
     affiche (l);

    ListIterator it = l.listIterator () ;
    it.next() ; // on se place sur le premier
     élément
    it.add ("c") ; it.add ("b") ;  // et on ajoute
     deux éléments
    System.out.print ("Liste en C : ") ;
     affiche (l);

    it = l.listIterator() ;
    it.next() ; // on progresse d'un element
    it.add ("b") ; it.add ("d") ;     // et on
     ajoute deux éléments
    System.out.print ("Liste en D : ") ;
     affiche (l) ;

    it = l.listIterator (l.size()) ;   // on se place
     en fin de liste
    while (it.hasPrevious())     // on
     recherche le dernier b
    { String ch = (String) it.previous() ;
      if (ch.equals ("b"))
      { it.remove() ; // et on le supprime
        break ;}
    }
    System.out.print ("Liste en E : ") ;
     affiche (l) ;
    it = l.listIterator() ;
    it.next() ; it.next() ;   // on se place sur le
     2eme element
    it.set ("x") ; // qu'on remplace par "x"
    System.out.print ("Liste en F : ") ;
     affiche (l) ;
  }
  public static void affiche (LinkedList l)
  { ListIterator iter = l.listIterator ();
    while (iter.hasNext()) System.out.print
     (iter.next() + " ") ;
    System.out.println () ;
  }}
```

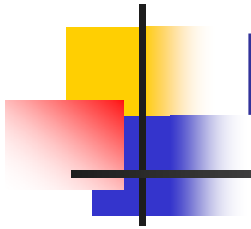
- Exemple 2 : pour afficher à l'envers

```
import java.util.* ;
public class Liste2
{ public static void main (String
    args[])
  { LinkedList l = new LinkedList() ;
     /* on ajoute a la liste tous les
    mots lus au clavier */
    System.out.println ("Donnez un
    suite de mots (vide pour finir)") ;
    while (true)
    { String ch = Lire.S() ;
      if (ch.length() == 0) break ;
      l.add (ch) ;
    }
    System.out.println ("Liste des
      mots a l'endroit :") ;
    ListIterator iter = l.listIterator() ;
    while (iter.hasNext())
    System.out.print (iter.next() + "
    ") ;
    System.out.println () ;
    System.out.println ("Liste des
    mots a l'envers :") ;
    iter = l.listIterator(l.size()) ;  //
    iterateur en fin de liste
    while (iter.hasPrevious())
    System.out.print (iter.previous()
    + " ") ;
    System.out.println () ;
  }
}
```



Exercice

- Énoncé
  - Soient deux listes d'entiers triées
  - Réaliser l'interclassement de ces deux listes dans une des deux listes
  - Faites le d'abord pour des entiers avec des comparaisons classiques élément par élément
  - Prévoyez le cas où une liste entière est inférieure à une autre : redéfinir la méthode compareTo
  - Faites l'interclassement pour des objets de la classe Object nécessitant de redéfinir la méthode CompareTo



Les vecteurs dynamiques

- Classe ArrayList
  - Offre des fonctionnalités d'accès rapide comparables à celles d'un tableau d'objets
  - Bien qu'elle implémente elle aussi l'interface List, sa mise en œuvre est différente pour permettre un accès rapide à un élément de rang donné
  - Cette classe offre plus de souplesse que les tableaux d'objets dans la mesure où sa taille peut varier dynamiquement au fil de l'exécution

Les vecteurs dynamiques

classe ArrayList

- Opérations usuelles
  - Comme toute construction, un vecteur dynamique peut être construit vide ou à partir d'une autre collection
    - ArrayList v1 = new ArrayList(); //vecteur dynamique vide
    - ArrayList v2 = new ArrayList (c); //vecteur dynamique contenant tous les éléments de la collection c
  - Ajout d'un élément
    - add(elem) : ajoute l'élément en fin de vecteur
    - add(i,elem) : ajoute un élément en rang i, le ième élément et tous les suivants sont décalés

Les vecteurs dynamiques

classe ArrayList

- Suppression d'un élément
  - Méthode spécifique : remove
    - permettant de supprimer un élément de rang donné

```
ArrayList v;
…
Object o =v.remove(3); // supprime le troisième
   élément de v qu'on obtient dans o
```

  - Autre méthode spécifique : removeRange
    - ArrayList v;
    - …
    - v.removeRange(3,8); // supprime les éléments de rang 3 à 8 de v

Les vecteurs dynamiques

classe ArrayList

- Accès aux éléments : get(i)

```
For (int i=0; i<v.size();i++)
{
   //utilisation de v.get(i);
}
```

- Remplacement d'une valeur : set(i,elem)

```
For (int i=0; i<v.size();i++)
{
   if(condition) set(i,null);
}
```

Les vecteurs dynamiques

classe ArrayList

- Remarque
  - ArrayList implémente également l'interface List et à ce titre dispose d'un itérateur bidirectionnel qu'on peut théoriquement utiliser pour parcourir les éléments d'un vecteur
  - Toutefois, cette possibilité fait double emploi avec l'accès direct get ou set auquel on recourra le plus souvent
  - A titre d'exemple :

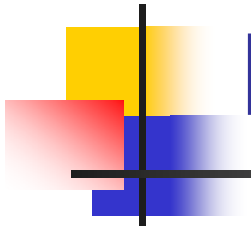
```
ListIterator it = v.listIterator();
while(it.hasNext())
{
   Object o = it.next()
   //utilisation de o
}
```

- Exemple 1 : crée un vecteur de 10 objets de type Integer

```
import java.util.* ;
public class Array1
{ public static void main (String args[])
  { ArrayList v = new ArrayList () ;
    System.out.println ("En A : taille de v = "
     + v.size() ) ;
      /* on ajoute 10 objets de type Integer */
    for (int i=0 ; i<10 ; i++) v.add (new
     Integer(i)) ;
    System.out.println ("En B : taille de v = "
     + v.size() ) ;
     /* affichage du contenu, par acces
    direct (get) a chaque élément */
    System.out.println ("En B : contenu de v
     = ") ;
  for (int i = 0 ; i<v.size() ; i++)
       System.out.print (v.get(i) + " ") ;
    System.out.println () ;
     /* suppression des éléments de position
    donnée */
v.remove (3) ;
    v.remove (5) ;
    v.remove (5) ;
    System.out.println ("En C : contenu de v
     = " + v) ;
     /* ajout d'éléments a une position
    donnee */
    v.add (2, new Integer (100)) ;
    v.add (2, new Integer (200)) ;
    System.out.println ("En D : contenu de v
     = " + v) ;
     /* modification d'éléments de position
    donnee */
    v.set (2, new Integer (1000)) ;  //
     modification élément de rang 2
    v.set (5, new Integer (2000)) ;  //
     modification élément de rang 5
    System.out.println ("En D : contenu de v
     = " + v) ;
  }
}
```

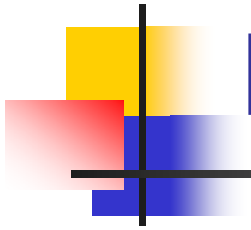
- Exemple 2 : crée un vecteur hétérogène avec des éléments Integer et String. On remplace par set tous les éléments String…

```
import java.util.* ;
public class Array2
{ public static void main (String args[])
  { ArrayList v = new ArrayList () ;
    /* on introduit 10 elements de type
    Integer */
    for (int i=0 ; i<10 ; i++) v.add (new
    Integer(i)) ;
    /* puis 4 elements de type String */
    v.add (2, "AAA") ;  // en position 2
    v.add (4, "BBB") ;  // en position 4
    v.add (8, "CCC") ;  // en position 8
    v.add (5, "DDD") ;  // en position 5
    System.out.println ("En I   : contenu
    de v = " + v) ;
/* on remplace tous les objets de type
    chaine par la reference null */
    for (int i = 0 ; i<v.size() ; i++)
     if (v.get(i) instanceof String) v.set (i,
    null) ;
    System.out.println ("En II  : contenu
    de v = " + v) ;
/* on cree une nouvelle collection
    (ici une liste) contenant deux
    */
     /* elements : une reference a
    un objet Integer (5), une
    reference null */
    LinkedList l = new LinkedList () ;
    l.add (new Integer (5)) ; l.add
    (null) ;
    v.removeAll (l) ;
    System.out.println ("En III :
    contenu de v = " + v) ;
  }
}
```



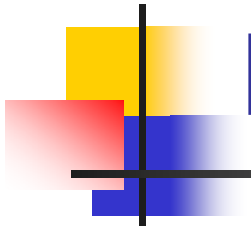
Exercice

- Énoncé
  - Programmer le triangle de Pascal en utilisant un ArrayList



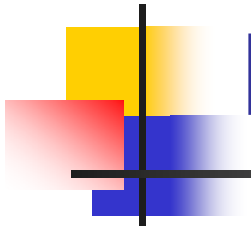
Les ensembles

- Généralités
  - Deux classes implémentent la notion d'ensemble :
    - HasSet et TreeSet
  - Rappelons que, théoriquement, un ensemble est une collection non ordonnée d'éléments
    - aucun élément ne pouvant apparaître plusieurs fois dans un même ensemble
  - Chaque fois qu'on introduit un nouvel élément dans une collection de type HashSet ou TreeSet, il est nécessaire de s'assurer qu'il n'y figure pas déjà
  - Nous avons dit que dans ce cas de collection, il faut se préoccuper des méthodes equals ou compareTo
    - Si on ne le fait pas, il faut accepter que deux objets de références différentes ne soient jamais identiques, quelles que soient leurs valeurs !



Les ensembles

- Généralités (suite)
  - Bien que les ensembles ne soient pas ordonnés, une certaine organisation des données y est nécessaire
  - Deux démarches ont été employées par les concepteurs des collections, d'où l'existence de deux classes différentes :
    - HashSet
      - qui recourt à une technique de hachage
    - TreeSet
      - qui utilise un arbre pour ordonner complètement les données



Les ensembles

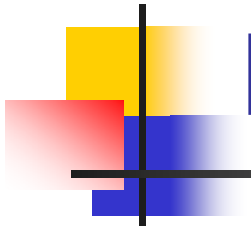
- Opérations usuelles
  - Construction et parcours
    - Comme toute collection, un ensemble peut être construit vide ou à partir d'une autre collection
    - HashSet

      HashSet e1 = new HashSet();

      HashSet e1 = new HashSet(c);
    - TreeSet

      TreeSet e1 = new TreeSet();

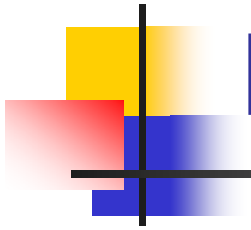
      TreeSet e1 = new TreeSet(c);



Les ensembles

- Opérations usuelles
  - Construction et parcours
    - Les deux classes HashSet et TreeSet disposent de la méthode iterator prévue dans l'interface Collection
    - Elle fournit un itérateur monodirectionnel (Iterator) permettant de parcourir les différents éléments de la collection

```
HashSet e //ou  TreeSet e;
…
Iterator it = e.iterator();
while(it.hasNext())'
    Object o = it.next();
    //utilisation de o
}
```

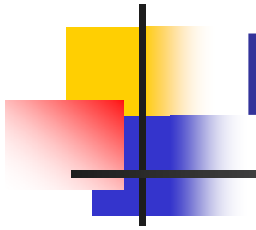


Les ensembles

- Ajout d'un élément

```
HashSet e; Object elem;
…
boolean existe =e.add(elem);
if(existe) System.out.println(elem + "existe déjà");
else
   System.out.println(elem + "a été ajouté");
}
```

- Remarque
  - add :
    - ajoute un élément en fin d'ensemble. En effet, les ensembles sont organisés au niveau de leur implémentation



Les ensembles

- Suppression d'un élément
  - remove(o) qui renvoie true si l 'élément est trouvé

```
TreeSet e; Object o;
…
boolean trouve =e.remove(o);
if(trouve) System.out.println(o + "a été supprimé");
else
    System.out.println(o + "n'existe pas");
}
```

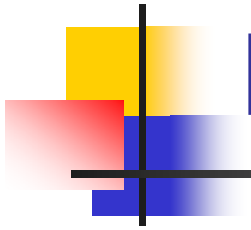
  - Par ailleurs la méthode remove de l'itérateur permet de supprimer l'élément courant(le dernier renvoyé par next) s'il existe

```
TreeSet e;
…
Iterator it = e.iterator();
It.next();it.next(); //deuxième élément = élément courant
It.remove(); //supprime le deuxième élément
}
```

  - Enfin la méthode contains permet de tester l'existence d'un objet

- Exemple 1 : ensemble d'éléments de type Integer

```
import java.util.* ;
public class Ens1
{ public static void main (String args[])
  { int t[] = {2, 5, -6, 2, -8, 9, 5} ;
    HashSet ens = new HashSet() ;
      /* on ajoute des objets de type
    Integer */
    for (int i=0 ; i< t.length ; i++)
    { boolean ajoute = ens.add (new
    Integer (t[i])) ;
     if (ajoute) System.out.println ("  On
    ajoute " + t[i]) ;
           else System.out.println ("  " +
    t[i] + " est deja present") ;
    }
    System.out.print ("Ensemble en A =
    ") ; affiche (ens) ;
      /* on supprime un eventuel objet
    de valeur Integer(5) */
    Integer cinq = new Integer (5) ;
    boolean enleve = ens.remove (cinq)
    ;
    if (enleve) System.out.println ("  On a
     supprime 5") ;
    System.out.print ("Ensemble en B =
     ") ; affiche (ens) ;
      /* on teste la presence de
     Integer(5) */
    boolean existe = ens.contains (cinq) ;
    if (!existe) System.out.println ("  On
     ne trouve pas 5") ;
  }
  public static void affiche (HashSet
     ens)
  { Iterator iter = ens.iterator () ;
    while (iter.hasNext())
    { System.out.print (iter.next() + " ") ;
    }
    System.out.println () ;
  }
}
```



Les ensembles

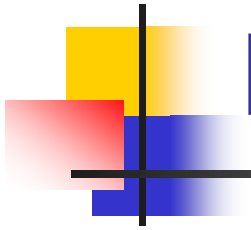
- Opérations ensemblistes
  - Les méthodes removeAll, addAll et retainAll, applicables à toutes les collections, vont prendre un intérêt tout particulier avec les ensembles où elles vont bénéficier de l'efficacité de l'accès à une valeur donnée
  - Ainsi, si e1 et e2 sont deux ensembles
    - e1.addAll(e2) place dans e1 tous les éléments présents dans e2
      - Après exécution, la réunion de e1 et de e2 se trouve dans e1
    - e1.retainAll(e2) garde dans e1 ce qui appartient à e2
      - Après exécution, on obtient l'intersection de e1 et de e2 dans e1
    - e1.removeAll(e2) supprime de e1 tout ce qui appartient à e2
      - Après exécution, on obtient le complémentaire par rapport à e1 dans e1

- Exemple 2 : EnsOp

```
import java.util.* ;
public class EnsOp
{ public static void main (String
    args[])
  { int t1[] = {2, 5, 6, 8, 9} ;
int t2[] = {3, 6, 7, 9} ;
    HashSet e1 = new HashSet(),
    e2=new HashSet();
      /* on ajoute des objets de type
    Integer */
    for (int i=0 ; i< t1.length ;
    i++)e1.add(new Integer(t1[i]));
    for (int i=0 ; i< t2.length ;
    i++)e2.add(new Integer(t2[i]));
     System.out.println ("e1 = " +
    e1);System.out.println ("e2 = " +
    e2);
//réunion de e1 et e2 dans u1
    HashSet u1 = new
    HashSet();copie(u1,e1);//copie e1
    dans u1
u1.addAll(e2);
System.out.println ("u1 = " + u1);
//intersection de e1 et e2 dans i1
    HashSet i1 = new
     HashSet();copie(i1,e1);
    i1.retainAll(e2);
    System.out.println ("i1 = " + i1);
  }
  public static void copie
    (HashSet but, HashSet source)
  { Iterator iter = source.iterator () ;
    while (iter.hasNext())
    { but.add(iter.next()) ;
    }
  }

}
```

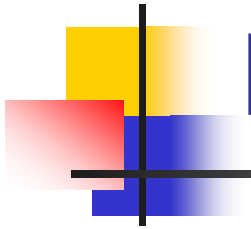
- Exemple 3 : Ens2 : lettres et voyelles présentes dans un texte

```
import java.util.* ;
public class Ens2
{ public static void main (String args[])
  { String phrase = "je me figure ce zouave qui joue" ;
    String voy = "aeiouy" ;

    HashSet lettres = new HashSet() ;
    for (int i=0 ; i<phrase.length() ; i++)
      lettres.add (phrase.substring(i, i+1)) ;
     System.out.println ("lettres presentes : " + lettres) ;

    HashSet voyelles = new HashSet() ;
    for (int i=0 ; i<voy.length() ; i++)
      voyelles.add (voy.substring (i, i+1)) ;
    lettres.removeAll (voyelles) ;
    System.out.println ("lettres sans les voyelles : " + lettres)
    ;
  }
}
```



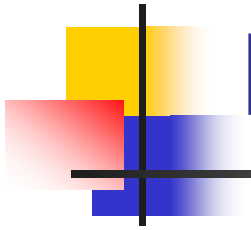
Les ensembles

- Les ensembles HashSet
  - Jusqu'ici, on a considéré que des ensembles dont les éléments étaient d'un String ou primitif pour lesquels, nous n'avions pas à nous préoccuper des détails d'implémentation
  - Dès que l'on cherche à utiliser des éléments d'un autre type, il est nécessaire de connaître quelques conséquences de la manière dont les ensembles sont effectivement implémentés



Les ensembles HashSet

- Dans le cas de HashSet, on doit définir convenablement
  - equals :
    - qui sert à définir l'appartenance d'un élément à l'ensemble
  - hashCode :
    - dont nous allons voir comment elle est exploitée pour ordonnancer les éléments d'un ensemble, ce qui nous amène à parler de table de hachage

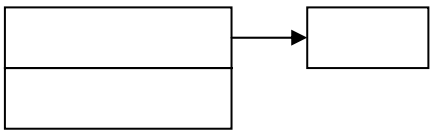


Les ensembles HashSet

- Notion de table de hachage
  - C’est une organisation des données qui permet de retrouver facilement un élément de valeur donnée
  - Pour cela, on utilise une fonction de hachage qui à la valeur d’un élément (existant ou recherché) associe un entier
  - Un même entier peut correspondre à plusieurs valeurs différentes
  - En revanche, deux éléments de même valeur doivent toujours fournir le même code de hachage

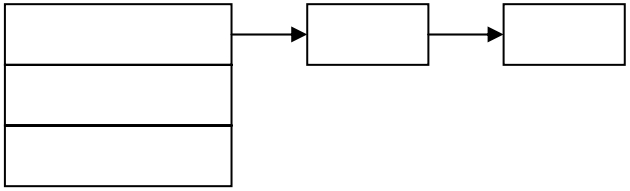
Les ensembles HashSet

- Notion de table de hachage (suite)
  - Pour organiser les éléments de la collection, on va constituer un tableau de N listes chaînées (nommées souvent seaux)

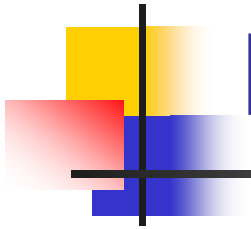


Seau de rang 0 : 1 élément

Seau de rang 0 : 1 élément



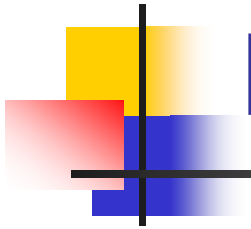
Seau de rang i : 2 éléments



Les ensembles HashSet

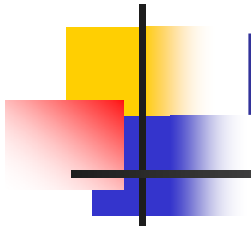
- Notion de table de hachage (suite)
  - Initialement les seaux sont vides
  - A chaque ajout d'un élément à la collection, on lui attribuera un emplacement dans un des seaux dont le rang i (dans le tableau des seaux) est défini en fonction de son code de hachage code de la manière suivante :

    i = code %N
  - S'il existe déjà des éléments dans le seau, le nouvel élément est ajouté à la fin de la liste chaînée correspondante
  - Le choix de la valeur initiale de N sera fait en fonction du nombre d'éléments prévus pour la collection
  - Pour retrouver un élément de la collection, on détermine son code
  - La formule i= code%N fournit un numéro i de seau dans lequel l'élément est susceptible de se trouver
  - Il ne reste plus qu'à parcourir les différents du seau…
  - Notez qu'on recourt à la méthode equals que pour les éléments du seau i



Les ensembles HashSet

- La méthode hashcode
  - Elle est utilisée pour calculer le code de hachage d'un objet
  - Les classes String, File et les classes enveloppes définissent une méthode de hashcode utilisant la valeur effective des objets
  - C'est pour cela que nous avons constitué sans problème des HashSet d'éléments de ce type
  - En revanche, les autres classes ne redéfinissent pas hashCode et l'on recourt à la méthode hashCode héritée de la classe Object, laquelle se contente d'utiliser comme « valeur » la simple adresse des objets
  - Dans ces conditions, des éléments différents auront toujours des codes différents (bof !)



Les ensembles HashSet

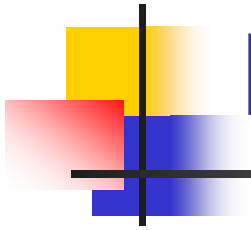
- La méthode hashcode (suite)
  - Si l'on souhaite pouvoir définir une égalité des éléments basée sur leur valeur effective, il va donc falloir définir dans la classe correspondante une méthode hashcode

    Int hascode()
  - Elle doit fournir le code de hachage correspondant à la valeur de l'objet
  - Dans la définition de cette fonction, il ne faudra pas oublier que le code de hachage doit être compatible avec equals : deux objets égaux par equals doivent fournir le même code

- Exemple 2 : EnsPt1 : utilise un ensemble d'objets de type Point. Elle redéfinit equals et hashCode. On a choisi un déterminant simple pour le hascode : somme des deux coordonnées

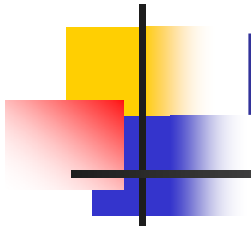
```
import java.util.* ;
public class EnsPt1
{ public static void main (String args[])
  { Point p1 = new Point (1, 3), p2 = new
     Point (2, 2) ;
    Point p3 = new Point (4, 5), p4 = new
     Point (1, 8) ;
    Point p[] = {p1, p2, p1, p3, p4, p3} ;
    HashSet ens = new HashSet () ;
    for (int i=0 ; i<p.length ; i++)
    { System.out.print ("le point ") ;
     p[i].affiche() ;
      boolean ajoute = ens.add (p[i]) ;
      if (ajoute) System.out.println (" a ete
     ajoute") ;
            else System.out.println ("est deja
     present") ;
      System.out.print ("ensemble = " ) ;
     affiche(ens) ;
    }
  }
-
```

```
public static void affiche (HashSet ens)
  { Iterator iter = ens.iterator() ;
    while (iter.hasNext())
    { Point p = (Point)iter.next() ;
      p.affiche() ;}
      System.out.println () ;}
}
class Point
{ Point (int x, int y) { this.x = x ; this.y = y
    ; }
  public int hashCode ()
  { return x+y ; }

  public boolean equals (Object pp)
  { Point p = (Point) pp ;
    return ((this.x == p.x) & (this.y == p.y))
    ;}
  public void affiche ()
  { System.out.print ("[" + x + " " + y + "] ")
    ;}
  private int x, y ;
}
```



Les ensembles

- Les ensembles TreeSet
  - La classe TreeSet propose une autre organisation que celle choisie par HashSet, sous forme d'arbre binaire qui ordonne les objets
  - On utilise, cette fois, la relation d'ordre usuelle induite par la méthode compareTo
  - Dans TreeSet, equals n'intervient pas du tout



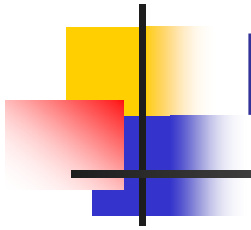
Les ensembles TreeSet

- Exemple
  - On va reprendre l'exemple EnsPt1 avec la classe Point en utilisant la classe TreeSet à la place de HashSet
  - On modifie la classe Point en supprimant les fonctions hashCode et equals en lui implémentant l'interface Comparable en redéfinissant compareTo
  - La comparaison se fait de manière lexicographique : on compare les abscisses et si elles ont égales, on compare les ordonnées

Exemple 2 : EnsPt2 :

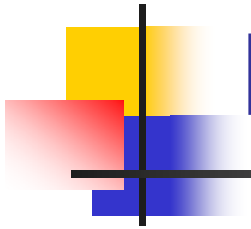
```
import java.util.* ;
public class EnsPt2
{ public static void main (String
    args[])
  { Point p1 = new Point (1, 3), p2 =
    new Point (2, 2) ;
   Point p3 = new Point (4, 5), p4 =
    new Point (1, 8) ;
   Point p[] = {p1, p2, p1, p3, p4, p3} ;
   TreeSet ens = new TreeSet () ;
   for (int i=0 ; i<p.length ; i++)
   { System.out.print ("le point ") ;
    p[i].affiche() ;
     boolean ajoute = ens.add (p[i]) ;
     if (ajoute) System.out.println (" a
    ete ajoute") ;
          else System.out.println ("est
    deja present") ;
     System.out.print ("ensemble = " ) ;
    affiche(ens) ;
   }
 }
```

```
class Point implements Comparable   //
    ne pas oublier implements ....
{ Point (int x, int y) { this.x = x ; this.y = y
    ; }
  public int compareTo (Object pp)
  { Point p = (Point) pp ;    // egalite si
    coordonnees egales
   if (this.x < p.x) return -1 ;
    else if (this.x > p.x) return 1 ;
     else if (this.y < p.y) return -1 ;
      else if (this.y > p.y) return 1 ;
       else return 0 ;
  }
  public void affiche ()
  { System.out.print ("[" + x + " " + y + "] ") ;
  }
  private int x, y ;
}
```



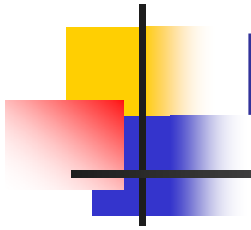
Les algorithmes

- La classe Collections
  - Elle fournit, sous forme de méthodes statiques, des méthodes utilitaires, générales applicables aux collections, notamment :
    - Recherche de maximum ou de minimum,
    - Tri et mélange aléatoire
    - Recherche binaire, copie, …
  - Ces méthodes disposent d'arguments d'un type Interface Collection ou List



La classe Collections

- Principales méthodes
  - Recherche de maximum ou de minimum
    - Ces algorithmes s'appliquent à des collections quelconques (implémentant l'interface Collection)
    - Ils utilisent une relation d'ordre définie classiquement
      - Soit par la méthode compareTo des éléments (il faut qu'ils implémentent l'interface Comparable)
      - Soit en fournissant un comparateur en argument de l'algorithme



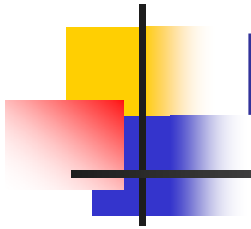
La classe Collections

- Recherche de maximum ou de minimum
  - Exemple :
    - Recherche du maximum des objets de type Point d'une liste l
    - La classe Point implémente ici l'interface Comparable et définit compareTo en se basant uniquement sur les abscisses des points
    - L'appel :
      - Collections.max(l) recherche le plus grand élément de l
    - Par ailleurs, on effectue un second appel de la forme :
      - Collections.max (l, new Comparator(){…}
    - On y fournit en second argument un comparateur anonyme, c.à.d un objet implémentant l'interface Comparator et définissant une méthode compare

- Exemple : MaxMin :

```
import java.util.* ;
public class MaxMin
{ public static void main (String args[])
  { Point p1 = new Point (1, 3) ; Point p2 = new Point (2, 1)
    ;
    Point p3 = new Point (5, 2) ; Point p4 = new Point (3, 2)
    ;
    LinkedList l = new LinkedList() ;
    l.add (p1) ; l.add (p2) ; l.add (p3) ; l.add (p4) ;

      /* max de l, suivant l'ordre defini par compareTo de
      Point */
    Point pMax1 = (Point)Collections.max (l) ;   // max
      suivant compareTo
    System.out.print ("Max suivant compareTo =  ") ;
      pMax1.affiche() ;
    System.out.println () ;

      /* max de l, suivant l'ordre defini par un comparateur
      anonyme */
    Point pMax2 = (Point)Collections.max (l, new
      Comparator()
                { public int compare (Object o1, Object o2)
                  { Point p1 = (Point) o1 ; Point p2 = (Point)
    o2 ;
                    if (p1.y < p2.y) return -1 ;
                      else if (p1.y == p2.y) return 0 ;
                        else return 1 ;
                  }
                }) ;
    System.out.print ("Max suivant comparator = " ) ;
      pMax2.affiche() ;
  }
}
```

```
class Point implements
    Comparable
{ Point (int x, int y) { this.x = x ;
    this.y = y ; }
  public void affiche ()
  { System.out.print ("[" + x + " " +
    y + "] ") ;
  }
  public int compareTo (Object pp)
  { Point p = (Point) pp ;
    if (this.x < p.x) return -1 ;
      else if (this.x == p.x) return 0 ;
        else return 1 ;
  }
  public int x, y ;    // public ici, pour
    simplifier les choses
}
```

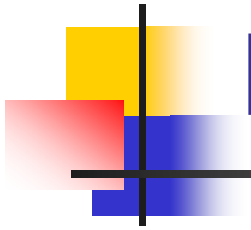


La classe Collections

- Tris et mélange
  - La classe Collections dispose de méthodes sort qui réalisent des algorithmes de tri des éléments d'une collection qui doit, cette fois implémenter l'interface List
  - Ses éléments sont réorganisés de façon à respecter l'ordre induit soit par compareTo, soit par un comparateur

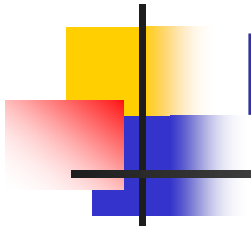
Exemple : Tri1 :

```
import java.util.* ;
public class Tri1
{ public static void main (String args[])
  { int nb[] = {4, 9, 2, 3, 8, 1, 3, 5} ;
    ArrayList t = new ArrayList() ;
    for (int i=0 ; i<nb.length ; i++) t.add (new Integer(nb[i])) ;

    System.out.println ("t initial      = " + t) ;
    Collections.sort (t) ;
    System.out.println ("t trie         = " + t) ;
    Collections.shuffle (t) ;
    System.out.println ("t melange      = " + t) ;
    Collections.sort (t, Collections.reverseOrder()) ;
    System.out.println ("t trie inverse = " + t) ;
  }
}
```



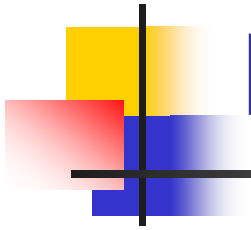
Les tables associatives

- Généralités
  - Une table associative permet de conserver une information associant deux parties nommées : clé et valeur
  - Exemples
    - Dictionnaire : à un mot, on associe une définition
    - Annuaire : à un nom, on associe un numéro de téléphone



Les tables associatives

- Implémentation
  - Comme pour les ensembles, l'intérêt des tables associatives est de pouvoir y retrouver rapidement une clé donnée pour en obtenir l'information associée
  - On retrouve les deux types d'organisation vus pour les ensembles
    - Table de hachage :
      - classe HashMap
    - Arbre binaire :
      - classe TreeMap
  - Dans les deux cas, seule la clé sera utilisée pour ordonnancer les l'informations

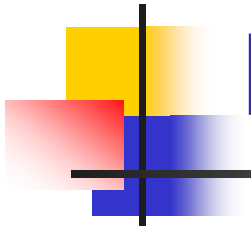


HashMap et TreeMap

- Présentation générale
  - Comme nous l'avons signalé, les classes HashMap et TreeMap n'implémentent pas l'interface Collection mais une interface nommée Map car leurs éléments ne sont pas des objets mais des paires d'objets
  - Ajout d'objet

```
HashMap m = new HashMap(); // table vide
…
m.put("m", new Integer(3)); // ajoute 3 associé à la clé m
```

  - Si la clé fournie à put existe déjà, la valeur associée remplace l'ancienne



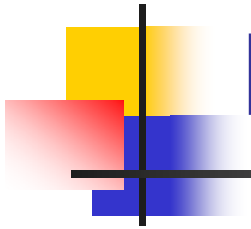
HashMap et TreeMap

- Recherche d'information : get

```
Object o = get("x"); // fournit la valeur associée à la clé "x"
If(o==null) System.out.println ("aucune valeur associée à la
   clé "x");
```

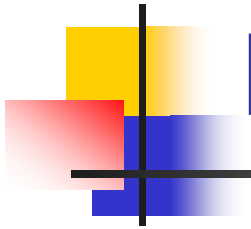
- La méthode containsKey permet de savoir si une clé donnée est présente
- Suppression d'information : remove

```
Object cle = "x";
Object val = remove(cle); // supprime l'élément (clé +
   valeur) de la clé "x";
if (val !=null) System.out.println(" on a supprimé l'élément
   de clé " + cle + " et de valeur " + val);
else System.out.println(" la clé " + cle + " n'existe pas ");
```



HashMap et TreeMap

- Parcours d'une table
  - En théorie, HashMap et TreeMap ne disposent pas d'itérateurs, mais à l'aide d'une méthode nommée entrySet, on peut voir une table comme un ensemble de paires
  - Une paire est un élément de type Map.Entry
  - Les méthodes getKey et getValue permettent de récupérer respectivement la clé et la valeur



HashMap et TreeMap

- Parcours d'une table
  - Voici un canevas de parcours d'une table utilisant ces possibilités

```
HashMap m;
…
Set entrees = m.entrySet(); //entrees est un ensemble de paires
iterator iter =entrees.iterator(); //itérateur sur les paires
while(iter.hasNext()) //boucle sur les paires
{
   Map.Entry entree =(Map.Entry)iter.next; //paire courante
   Object cle = entree.getKey(); //Clé de la partie courante
   Object valeur = entree.getValue(); //valeur de la paire courante
…
}
```

- Exemple : Map1

```
import java.util.* ;
public class Map1
{ public static void main (String args[])
  { TreeMap m = new TreeMap() ;
    m.put ("c", "10") ; m.put ("f", "20") ; m.put ("k",
      "30") ;
    m.put ("x", "40") ; m.put ("p", "50") ; m.put ("g",
      "60") ;
    System.out.println ("map initial :                    "
      + m) ;

    // retrouver la valeur associee a la cle "f"
    String ch = (String)m.get("f") ;
    System.out.println ("valeur associee a f :
      " + ch) ;

    // ensemble des valeurs
    Collection valeurs = m.values () ;  // attention, ici
      Collection, pas Set
    System.out.println ("liste des valeurs initiales :
      " + valeurs) ;
    valeurs.remove ("30") ; // on supprime la valeur
      "30" par la vue associee
    System.out.println ("liste des valeurs apres sup
      :       " + valeurs) ;

// ensemble des cles
    Set cles = m.keySet () ;
    System.out.println ("liste des cles
      initiales :          " + cles) ;
    cles.remove ("p") ;     // on supprime la
      cle "p" par la vue associee
    System.out.println ("liste des cles apres
      sup :          " + cles) ;

// modification de la valeur associee a la
     clé x
    String old = (String)m.put("x", "25") ;
    if (old != null)
    System.out.println ("valeur associee a x
      avant modif :    " + old) ;
    System.out.println ("map apres modif de
      x :              " + m) ;
    System.out.println ("liste des valeurs
      apres modif de x : " + valeurs) ;
    // On parcoure les entree (Map.Entry)
      du map jusqu'a trouver la valeur 20
```

```
// et on supprime l'element correspondant (suppose exister)
    Set entrees = m.entrySet () ;
    Iterator iter = entrees.iterator() ;
    while (iter.hasNext())
    { Map.Entry entree = (Map.Entry)iter.next() ;
      String valeur = (String)entree.getValue() ;
      if (valeur.equals ("20"))
        { System.out.println ("valeur 20 " + "trouvee en cle "+
     (String)entree.getKey()) ;
          iter.remove() ; // suppression sur la vue associee
          break ;
        }
    }
    System.out.println ("map apres sup element suivant 20 :   " + m) ;
    // on supprime l'element de cle "f"
    m.remove ("f") ;
    System.out.println ("map apres suppression f :            " + m) ;
    System.out.println ("liste des cles apres suppression f : " + cles) ;
    System.out.println ("liste des valeurs apres supp de f  : " + valeurs) ;
    Lire.S() ;
  }
}
```