Chapitre 3

Les collections (concepts avancés)

L'objectif de ce cours est de découvrir et savoir utiliser les classes JAVA de définition des collections.

1. PRESENTATION 2

1.1. L'INTERFACE ITERABLE<T> 2
1.2. L'INTERFACE COLLECTION<E> 2
1.3. LA CLASSE ABSTRACTCOLLECTION 2
1.4. L'INTERFACE LIST (ET ABSTRACTLIST) 5
1.5. LA CLASSE ABSTRACTSET ET L'INTERFACE SET 5
1.6. L'INTERDEPENDANCE DES "COLLECTION" (INTERFACE) 6

2. LES COLLECTIONS ET LES TABLEAUX : LA CLASSE ARRAYS 8

3. L' « AUTO-BOXING » DES COLLECTIONS 10

4. LA RECHERCHE DANS UNE COLLECTION NON POLYMORPHE 11

5. LA RECHERCHE DANS UNE COLLECTION POLYMORPHE 13

6. LE TRI D'UNE COLLECTION : INTERFACE COMPARABLE 15

7. LES MULTI-TRI D'UNE COLLECTION : INTERFACE COMPARATOR 18

8. LA CLASSE VECTOR 21

9. LA CLASSE HASHTABLE 23

9.1. PRESENTATION 23
9.2. EXEMPLE (VOIR EXEMPLEHASHTABLE.JAVA) 23
9.3. LES METHODES DE LA CLASSE 25

10. LA CLASSE HASHSET 26

10.1. PRESENTATION 26
10.2. EXEMPLE (VOIR EXEMPLEHASHSET.JAVA) 27
10.3. LES METHODES DE LA CLASSE 29
10.4. EXEMPLE2 (VOIR REDONDANCE.JAVA) 30

1. Présentation

Les "collections" et autres classes permettant de stocker des éléments sont nombreuses en JAVA. Elles sont structurées en classe, classe abstraite et interface.

On trouve les classes de collection dans le package : **java.util.***

1.1. L'interface Iterable<T>

Cette interface est dans le package java.lang car pas uniquement propre aux collections.

Cette interface contient la méthode :

```
Iterator<T> iterator()
```

qui **retourne un objet** dont la classe d'appartenance (pas nécessairement connue par l'appelant) implémente l'interface :
Iterator<E>
Cette interface définit les 2 méthodes :

```
boolean hasNext()
E next()
```

Un itérateur est un objet qui permet grâce à ces deux méthodes d'itérer une structure informatique.
Ce qui est le cas de toutes les collections qui par définition contiennent des éléments qu'il est possible de parcourir grâce à un itérateur.

C'est pour cela, comme on va le voir ci-dessous, que toutes les collections implémentent l'interface **Iterable<T>**.

Cet itérateur est utilisé dans la boucle for énumérative.

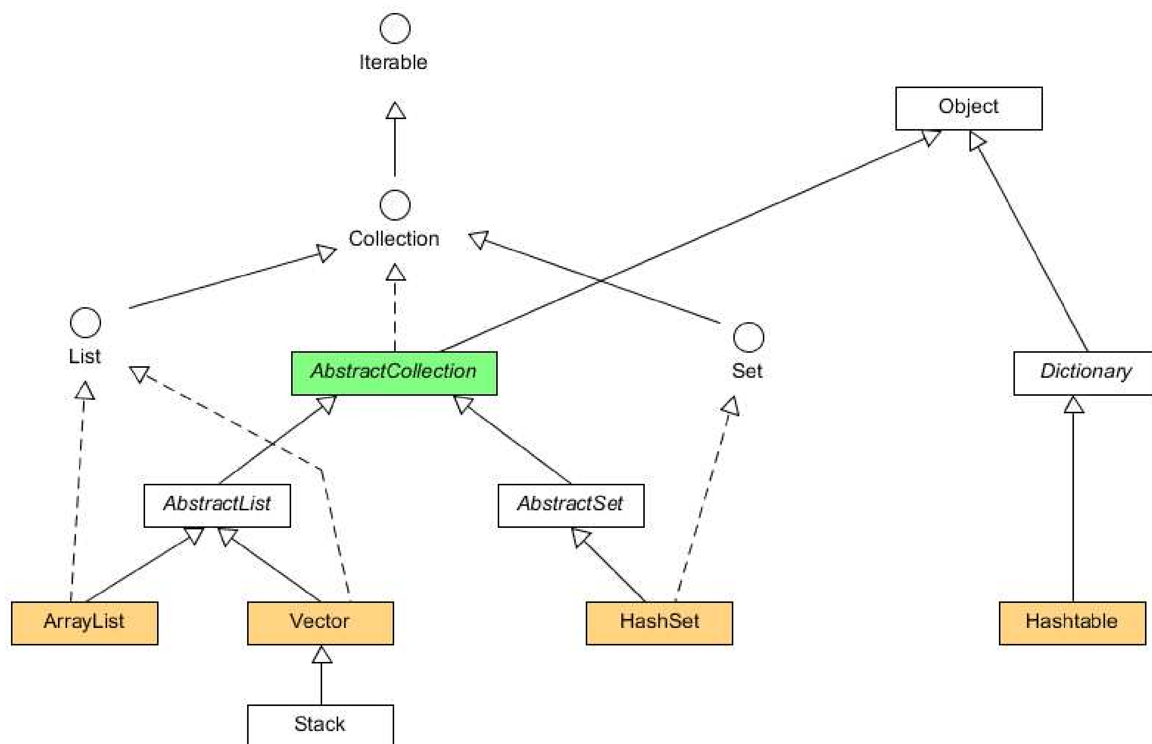
1.2. L'interface Collection<E>

Cette interface hérite d' Iterable<T>.
Elle définit toutes les autres opérations qu'une collection devrait implémenter : add, contains, clear, equals, isEmpty, iterator, remove,size, ….

Toutes les classes de collection doivent implémenter cette interface.

Mais au lieu que les classes collections implémentent directement cette interface, Java a préféré de créer une classe intermédiaire abstraite, **AbstractCollection**, qui implémente par défaut cette interface.
Ainsi les classes de collection héritent de cette classe abstraite au lieu d'implémenter directement cette interface.

1.3. La classe AbstractCollection



Cette classe abstraite est un squelette d'implémentation de l'interface Collection afin de minimiser l'effort pour implémenter cette interface.

Elle contient deux méthodes abstraites : size et iterator

Les autres méthodes sont implémentées et retourne par défaut l'exception : UnsupportedOperationException.

Ceci permet de ne pas être obligé d'implémenter toutes les méthodes de l'interface dans la conception d'un nouveau type de collection.

Voir sur le site http://jacques.laforgue.free.fr l'exemple **Exemple28_AbstractCollection**

```
import java.util.*;

public class Exemple28
{
    public static void main(String... args)
    {
        System.out.println("Execution de Exemple28");

        // Création de MaCollection
        MaCollection<Integer> c = new MaCollection<Integer>(100);

        c.add(100);
        c.add(200);
```

```
          c.add(10);

          System.out.println(c);
          System.out.println("Avec un iterateur:");
          for(Integer e:c)System.out.println(e);
      }
  }

  // Implémentation de MaCollection
  //  qui hérite de la classe abstraite qui ne contient que deux méthodes abstraites : size et iterator. Les autres sont implémentées et retourne par défaut l'exception : UnsupportedOperationException.
  //
  // Cela permet de ne pas implémenter toutes les méthodes de l'interface Collection
  //
  class MaCollection<E> extends AbstractCollection<E> implements Iterator<E>
  {
      private int current;
      private int nb;
      private E[] tab;

      public MaCollection(int capacity)
      {
          // tab=new E[capacity] provoque l'erreur de compilation :
          //    error: generic array creation
          // Il faut faire :
          tab=(E[])(new Object[capacity]);
      }

      // ---------- Implémentation des méthodes abstraites -------------
      //            de AbstractCollection
      //

      // Méthode qui retourne le nombre d'éléments utiles de la collection
      public int size()
      {
          return nb;
      }


      public Iterator<E> iterator()
      {
          current = 0;
          return this;
      }

      // ---------- Implémentation des méthodes de l'interface  -------------
      //              Iterator

      // Il reste encore des éléments à parcourir
      public boolean hasNext()
      {
          return (current<nb);
      }

      // Retourne l'élément courant et passe au suivant
      public E next()
      {
```

```
        E e = tab[current];
        current++;
        return e;
    }

    // ---------- Surcharge des méthodes de la classe  -------------
    //              AbstractCollection

    // Ajouter un élément
    public boolean add(E e)
    {
        if (nb==tab.length) return false;
        tab[nb]=e;
        nb++;
        return true;
    }

    // Conversion en chaine de la collection
    public String toString()
    {
        String s="";
        for(int i=0;i<nb;i++) s=s + tab[i] + " ";
        return s;
    }
}
```

1.4. L'interface List (et AbstractList)

Si on compare l'interface List et l'interface Collection, on s'aperçoit que cette interface définit des méthodes supplémentaires basées sur la notion d' « **index** ».

Cela signifie qu'une « liste » est bien une « collection » dans le sens où on peut ajouter, supprimer (par itération uniquement), rechercher un élément (Contains), … mais en plus les éléments sont **indexés par un entier**.

D'où les méthodes :
- add(index,e) permettant l'insertion
- get(i) qui retourne l'élément se trouvant à l'index i
- remove(i) qui supprime l'élément se trouvant à l'index i
- ….

Il y a aussi en plus un ListIterator qui permet une itération dans les 2 sens (previous()).

Ainsi il existe une classe AbstractList (pour les mêmes raisons que AbstractCollection) de laquelle dérivent les classes importantes de collection :
- ArrayList
- Vector

1.5. La classe AbstractSet et l'interface Set

En informatique un « ensemble » est une structure de données qui ressemble beaucoup à une collection mais pour laquelle les éléments ne peuvent pas exister en double. La notion d'index est également inutile car les éléments d'un ensemble ne sont pas nécessairement indexés.

Par contre il faut pouvoir les parcourir.

C'est donc une sorte de sous-ensemble des propriétés d'une collection.

Ainsi la classe AbstractSet hérite de AbstractCollection et implémente l'interface Set.

Remarque : il n'y a pas de différence entre l'interface Set et Collection. Il n'y a pas de nouvelles propriétés. La différence est d'autre conceptuelle dans l'implémentation de ces méthodes et donc dans leurs rôles (les commentaires des méthodes sont différents).

1.6. L'interdépendance des "Collection" (interface)

Prenons la méthode addAll de l'interface Collection :

```
boolean addAll(Collection< ? extends E> c)
```

Cette méthode ajoute tous les éléments d'une collection passée en paramètre à une autre collection ;
Cela signifie que toutes les collections peuvent se copier les éléments entre elles ;

Exemple :

```
Vector<String> v = new Vector<String>();
        v.add("TOTO");
        v.add("TATA");
        v.add("TUTU");
        v.add("TUTU");
        v.add("TATA");
        v.add("TOTO");

        ArrayList<String> liste = new ArrayList<String>();

        liste.add("PREMIER");
        liste.addAll(v);

        for(String s:liste)System.out.println(s);
```

Egalement par exemple tester si tous les éléments d'un Vector sont dans un ArrayList :

```
         if (liste.containsAll(v))
            System.out.println("VRAI");
```

Attention à la compatibilité des types des éléments des collections quand on les utilise.

```
Vector v1 = new Vector();  // Vector de Object
        v1.add(12);
        v1.add("TATA");
        liste.addAll(v1);  // Ceci fonctionne

        for(String s:liste)System.out.println(s);
```

L'exécution de la boucle for :

```
Exception in thread "main" java.lang.ClassCastException:
java.lang.Integer cannot be cast to java.lang.String
   at Exemple29.main(Exemple29.java:29)
```

On peut aussi passer par un ensemble pour supprimer les éléments redondants :

```
// Ajouter une liste dans un ensemble
        System.out.println("Ajouter une liste dans un ensemble");
        HashSet<String> ensemble = new HashSet<String>();
        ensemble.addAll(liste2);

        System.out.println("\nAVANT:");
        for(String s:liste2)System.out.println(s);

        System.out.println("\nAPRES:");
        for(String s:ensemble)System.out.println(s);
        System.out.println("Les elements redondants sont supprimes !!");

        liste2 = new ArrayList(ensemble);
        for(String s:liste2)System.out.println(s);
```

Les exemples de code précédents sont dans :

Voir sur le site http://jacques.laforgue.free.fr l'exemple **Exemple29_InterCollections**

Exécution :

```
Execution de Exemple29
Les elements de liste (1)
PREMIER
TOTO
TATA
TUTU
TUTU
TATA
TOTO
Rechercher tous les elements de v dans liste
VRAI
Les elements de liste (2)
PREMIER
TOTO
TATA
TUTU
TUTU
TATA
TOTO
java.lang.ClassCastException:  java.lang.Integer  cannot  be  cast  to
java.lang.String
 at Exemple29.main(Exemple29.java:41)
Ajouter une liste dans un ensemble

AVANT:
PREMIER
TOTO
TATA
TUTU
TUTU
TATA
TOTO
```

```
APRES:
TOTO
TUTU
PREMIER
TATA
Les elements redondants sont supprimes !!
TOTO
TUTU
PREMIER
TATA
```

2. Les collections et les tableaux : la classe Arrays

Il existe un fort lien entre les collections et les tableaux java :

- la plupart des collections sont en fin de compte implémentées en tableau (pas de problème de performance)
- la vision de certains programmeurs est toujours celles de tableau
- Java s'interface avec d'autres langages qui impose l'utilisation des tableaux
- des standards de communication réseau n'utilisent que les tableaux pour communiquer des collections de données

La classe **Arrays** ne contient que des méthodes statics.

Créer un ArrayList à partir d'un tableau : utilisé la méthode : Arrays.**asList**

```
String tab[] = new String[10];
tab[0]="TOTO";
tab[1]="TATA";
tab[3]="TUTU";

List<String> l = Arrays.asList(tab);
ArrayList<String> liste = new ArrayList<String>(l);

System.out.println("Les elements de liste (1)");
for(String s:liste)System.out.println(s);
```

Exécution :

Execution de Exemple30
Les elements de liste (1)
TOTO
TATA
null
TUTU
null
null
null
null
null
null

Attention : tous les éléments du tableau sont mis dans le ArrayList et donc toutes les valeurs « non initialisées » aussi.

Afficher les tableaux : **Arrays.toString( )**

```
System.out.println(Arrays.toString(tab));
```

Exécution :

```
[TOTO, TATA, null, TUTU, null, null, null, null, null, null]
```

Trier les tableaux : **Arrays.sort( )**

```
Exception in thread "main" java.lang.NullPointerException
```

Attention : il ne faut pas d'élément à null

```
        tab = new String[10];
        tab[0]="TOTO";
        tab[1]="TATA";
        tab[2]="TUTU";
        tab[3]="ABBE";
        System.out.println(Arrays.toString(tab));

        Arrays.sort(tab,0,4);  // trie de 0 à 3 inclus (4 exclus)
        System.out.println("Tableau trie : ");
        System.out.println(Arrays.toString(tab));
```

Exécution :

```
[TOTO, TATA, TUTU, ABBE, null, null, null, null, null, null]
Tableau trie :
[ABBE, TATA, TOTO, TUTU, null, null, null, null, null, null]
```

Remplir un tableau : **Arrays.fill( )**

```
        String[] tab2 = tab.clone();
        Arrays.fill(tab2,"XXX");
        System.out.println("tab2: "+Arrays.toString(tab2));
        System.out.println("tab : "+Arrays.toString(tab));
```

Execution :

```
tab2: [XXX, XXX, XXX, XXX, XXX, XXX, XXX, XXX, XXX, XXX]
tab : [ABBE, TATA, TOTO, TUTU, null, null, null, null, null, null]
```

Egalité entre tableau : **Arrays.equals( )**

```
        if (Arrays.equals(tab,tab2))
            System.out.println("tab egual a tab2");
        else
            System.out.println("tab different de tab2");

        String[] tab3 = tab.clone(); // pour la suite
                if (Arrays.equals(tab,tab3))
            System.out.println("tab egual a tab3");
        else
            System.out.println("tab different de tab3");
```

Exécution :

```
tab different de tab2
tab egual a tab3
```

Extraction de collextion : **Arrays.copyOfRange( )**

```
          String[] tab4 = Arrays.copyOfRange(tab3,0,4);
          System.out.println("tab4 : "+Arrays.toString(tab4));
```

Exécution :

```
tab4 : [ABBE, TATA, TOTO, TUTU]
```

Rechercher un elément : **Arrays.binarySearch( )**

```
          int index = Arrays.binarySearch(tab4,"TOTO");
          System.out.println("Recherche de TOTO : "+index);

          index = Arrays.binarySearch(tab4,"X");
          System.out.println("Recherche de X : "+index);
```

Exécution :

```
Recherche de TOTO : 2
Recherche de X : -5
```

3. L' « auto-boxing » des collections

Depuis la version 1.5 de Java, un arrayList permet de gérer indirectement les types primitifs, à travers l'utilisation des classes classes Integer, Double, ….. car depuis cette version il existe un mécanisme appelé l'autoboxing qui permet une conversion automatique entre les types primitifs et les types objets équivalents.

Voir sur le site http://jacques.laforgue.free.fr l'exemple **Exemple11_Collections**

Exemple 11 du site :

```
import java.util.*;

public class Exemple11
{
    public static void main(String... a_args)
    {
        Terminal.ecrireStringln("Exemple 11");

        Terminal.ecrireStringln("-----------------------------------");
        // Un array list contenant les types primitifs
        //
        ArrayList liste1 = new ArrayList();
        liste1.add( 123 );
        liste1.add( 23.45 );
        liste1.add( true );
        liste1.add( "TOTO" );
        liste1.add( "ENCORE" );
```

```
        Terminal.ecrireStringln(liste1.toString());

        Terminal.ecrireStringln("-----------------------------------");
        /// int n = liste1.get(0);
        // Exemple11.java:18: incompatible types
        // found    : java.lang.Object
        // required: int

        Integer i = (Integer) liste1.get(0);
        Double  d = (Double) liste1.get(1);
        Boolean b = (Boolean) liste1.get(2);
        String  s1 = (String) liste1.get(3);
        String  s2 = (String) liste1.get(4);
        Terminal.ecrireStringln("" + i);
        Terminal.ecrireStringln("" + d);
        Terminal.ecrireStringln("" + b);
        Terminal.ecrireStringln(s1);
        Terminal.ecrireStringln(s2);

        Terminal.ecrireStringln("-----------------------------------");
        // Le plus courant le ArrayList contient toujours le même type
        // Exemple un tableau d'entier :

        ArrayList<Integer> tabint = new ArrayList<Integer>();

        int n =456;
        tabint.add(123);
        tabint.add(n);
        tabint.add(2);

        int x = tabint.get(0);  //  Ici l'affectation est acceptée
        Terminal.ecrireStringln("x = " + x);

    }
}


Exécution :

Exemple 11
-----------------------------------
[123, 23.45, true, TOTO, ENCORE]
-----------------------------------
123
23.45
true
TOTO
ENCORE
-----------------------------------
x = 123
```

4. La recherche dans une collection non polymorphe

Il existe deux méthodes de recherche :

- la méthode **contains** de l’interface Collection
- la méthode **indexOf** de l’interface List

Ces méthodes utilisent la méthode : boolean <T>.***equals***.(<T>)
où <T> est le type de l'élément de la collection.

Nous n'avons pas besoin ici de passer par une interface pour réaliser le traitement car :
- la méthode equals est une méthode de Object
- <T> hérite de Object,
- la méthode contains (et indexOf) est basé sur Object
- la classe <T> surcharge la méthode equals

Exemple (suite de l'exemple 11)

Cet exemple démontre la différence entre avec ou sans l'implémentation de la méthode equals.

```
         Terminal.ecrireStringln("Utilisation de contains, role de
equals");
          ArrayList<Bidule> listeBidule = new ArrayList<Bidule>();

          Bidule bidule1 = new Bidule(20);

          listeBidule.add(new Bidule(10));
          listeBidule.add(bidule1);
          listeBidule.add(new Bidule(30));
          listeBidule.add(new Bidule(40));

          //Executer les lignes qui suit sans la méthode equals de Bidule
puis avec
          //
          if (listeBidule.contains(bidule1))
              Terminal.ecrireStringln("bidule1 de 20 trouve");
          else
              Terminal.ecrireStringln("bidule1 de 20 non trouve");

          Bidule bidule2 = new Bidule(20);
          if (listeBidule.contains(bidule2))
              Terminal.ecrireStringln("bidule2 de 20 trouve");
          else
              Terminal.ecrireStringln("bidule2 de 20 non trouve");


  avec


  class Bidule
  {
      int x;
      public Bidule(int x){this.x=x;}

      public String toString()
      {
          return ""+x;
      }

      //public boolean equals(Bidule o)  {return x==o.x;}
      // !!Piège ==> ce n'est pas la vraie méthode equals

      public boolean equals(Object o)
      {
```

```
        return x==((Bidule)o).x;
    }
}



Exécution avec ou sans la méthode equals :

Sans :
Utilisation de contains, role de equals
bidule1 de 20 trouve
bidule2 de 20 non trouve

Avec :
Utilisation de contains, role de equals
bidule1 de 20 trouve
bidule2 de 20 trouve
```

Si la méthode equals n'est pas implémentée alors c'est la méthode equals de Object qui est utilisée. Et cette méthode teste l'égalité des références entre les objets.

Il est donc primordial de définir systématiquement la méthode **equals** dans nos classes surtout si les objets de ses classes peuvent être dans une collection.

La méthode indexOf marche comme contains mais retourne l'indice de l'élément trouvé sinon -1.
Cette méthode ne peut être utilisée que pour les collections qui sont des List.

5. La recherche dans une collection polymorphe

La recherche dans une collections polymorphe nécessite de réaliser plusieurs points :

- créer une classe abstraite qui est le type d'élément de la collection
- d'écrire la méthode réelle equals dans la classe abstraite
- la méthode equals utilise soit des attributs de la classe abstraite, soit une méthode abstraite qui est implémentée par les classes réelles qui retourne le critère de comparaison

Voir sur le site http://jacques.laforgue.free.fr
l'exemple **Exemple31_CollectionPolymorphe**

```
import java.util.*;

public class Exemple31
{
    public static void main(String... args)
    {
        System.out.println("Execution de Exemple31");

        // Création de la liste
```

```
          ArrayList<Element> liste = new ArrayList<Element>();

          liste.add(new  Individu("LAFONT","Pierre","23  rue  de  la  pomme
TOULOUSE 31130"));
          liste.add(new Individu("ABBE","Paul","12 av de la poste MONTEAU
21345"));
          liste.add(new Voiture("Peugeot","AS 234 FG"));
          liste.add(new Voiture("Citroen","DF 456 GH"));
          System.out.println( Arrays.toString(liste.toArray()) );

          // Rechecrhe d'une Voiture
          Voiture v = new Voiture("","AS 234 FG");
          int index = liste.indexOf(v);
          if (index!=-1) System.out.println( liste.get(index).toString() );

          // Recherche d'un Individu
          Individu i = new Individu("ABBE","Paul","");
          index = liste.indexOf(i);
          if (index!=-1) System.out.println( liste.get(index).toString() );

          // Recherche par un objet de recherche dédié
          System.out.println(  liste.indexOf(new  ElementIndex("AS  234  FG"))
);  // 2
          System.out.println(  liste.indexOf(new  ElementIndex("ABBE  Paul"))
);       // 1


      }
  }

  abstract class Element
  {
      abstract public String getIdent();

      public boolean equals(Object o)
      {
          String e1 = this.getIdent();
          String e2 = ((Element)o).getIdent();
          return( e1.equals(e2) );
      }
  }

  class Individu extends Element
  {
      private String nom;
      private String prenom;
      private String adresse;
      public Individu(String nom,String prenom,String adresse)
      {
          this.nom=nom;
          this.prenom=prenom;
          this.adresse=adresse;
      }

      public String getIdent()
      {
          return nom+" "+prenom;
      }

      public String toString()
      {
```

```
        return(nom+" "+prenom+" "+adresse);
    }
}

class Voiture extends Element
{
    private String marque;
    private String plaque;
    public Voiture(String marque,String plaque)
    {
        this.marque=marque;
        this.plaque=plaque;
    }

    public String getIdent()
    {
        return plaque;
    }

    public String toString()
    {
        return(marque+" "+plaque);
    }
}

class ElementIndex  extends Element
{
    private String value;
    public ElementIndex(String value)
    {this.value=value;}

    public String getIdent(){return value;}
}
```

Exécution :

```
[LAFONT Pierre 23 rue de la pomme TOULOUSE 31130, ABBE Paul 12 av de la poste MONTEAU 21345, Peugeot AS 234 FG, Citroen DF 456 GH]
Peugeot AS 234 FG
ABBE Paul 12 av de la poste MONTEAU 21345
2
1
```

L'objet qui est passé en paramètre de la méthode indexOf doit être soit une Voiture soit un Individu. Cela peut être gênant car les constructeurs ne sont pas bien adaptés.
On préfère souvent créé artificiellement une classe qui sert de critère de recherche.
Dans l'exemple : la classe ElementIndex.

Remarque :

```
System.out.println( Arrays.toString(liste.toArray()) );
```

Cette instruction affiche le contenu de tous les éléments d'un ArrayList

6. Le tri d'une collection : interface Comparable

Pour trier une collection il faut 2 choses :
- être une classe qui implémente l'interface **List**
- la classe d'appartenance des éléments implémente l'interface **Comparable**

L'interface Comparable contient une méthode unique :

```
public int compareTo(Object lp)
```

retourne -1 si this est inférieur à lp
retourne 0 si this est égal à lp
retourn +1 si this est supérieur à lp

Le traitement de tri est dans la classe Collections. C'est une méthode static :

```
public static void sort(List<T> list)
```

Exemple :

```
import java.util.*;

public class TrierCollection
{
    static public void main(String args[])
    {
        // Cas d'une liste de String
        //
        ArrayList<String> liste = new ArrayList<String>();
        liste.add("ZADE Martine");
        liste.add("DUPONT Patrick");
        liste.add("LAFONT Pierre");
        liste.add("LAFARGUE Claude");
        liste.add("AMONGA N'Gunma");

        Terminal.ecrireStringln("-------- Tri de la liste -----");
        Collections.sort(liste);

        for(String s:liste)
            Terminal.ecrireStringln(s);

        // Cas d'une liste de Livre
        ArrayList<Livre> liste2 = new ArrayList<Livre>();
        liste2.add(new Livre("SF","DUNE"));
        liste2.add(new Livre("ROMAN","KGB"));
        liste2.add(new Livre("ROMAN","A MORT"));
        liste2.add(new Livre("SF","ALARME"));

        Terminal.ecrireStringln("-------- Tri de la liste -----");
        Collections.sort(liste2);

        for(Livre l:liste2)
            Terminal.ecrireStringln(""+l);

    }
}

class Livre implements Comparable
{
    String titre;
    String genre;
    public Livre(String genre, String titre)
    {
        this.titre=titre;
        this.genre=genre;
    }
```

```
    public int compareTo(Object livre)
    {
        Livre l1 = this;
        Livre l2 = (Livre)livre;

        if (l1.genre.compareTo(l2.genre)<0)
            return(-1);
        else if (l1.genre.compareTo(l2.genre)>0)
            return(1);
        else if (l1.titre.compareTo(l2.titre)<0)
            return(-1);
        else if (l1.titre.compareTo(l2.titre)>0)
            return(1);
        else
            return(0);
    }

    public String toString()
    {
        return genre+" / "+titre;
    }
}

Résultat de l'exécution :
java TrierCollection
-------- Tri de la liste -----
AMONGA N'Gunma
DUPONT Patrick
LAFARGUE Claude
LAFONT Pierre
ZADE Martine
-------- Tri de la liste -----
ROMAN / A MORT
ROMAN / KGB
SF / ALARME
SF / DUNE
```

Remarque : On peut aussi coder la classe Livre de la manière suivante :

```
class Livre implements Comparable<Livre>
{
    String titre;
    String genre;
    public Livre(String genre, String titre)
    {
        this.titre=titre;
        this.genre=genre;
    }

    public int compareTo(Livre livre)
    {
        Livre l1 = this;
        Livre l2 = livre;

        if (l1.genre.compareTo(l2.genre)<0)
            return(-1);
        else if (l1.genre.compareTo(l2.genre)>0)
            return(1);
        else if (l1.titre.compareTo(l2.titre)<0)
            return(-1);
```

```
          else if (l1.titre.compareTo(l2.titre)>0)
              return(1);
          else
              return(0);
      }

      public String toString()
      {
          return genre+" / "+titre;
      }
  }
```

Comme pour la recherche dans une collection polymorphe, le tri d'une collection polymorphe nécessite d'implémenter la méthode **compareTo** dans la **classe abstraite**.

Suite de l'exemple 31 précédent :

Dans la méthode main :

```
          // Trier la collection
          Collections.sort(liste);
          System.out.println( Arrays.toString(liste.toArray()) );
```

Dans la classe abstraite Element :

```
      public int compareTo(Object o)
      {
          String e1 = this.getIdent();
          String e2 = ((Element)o).getIdent();
          return( e1.compareTo(e2) );
      }
```

Exécution :

```
  [ABBE Paul 12 av de la poste MONTEAU 21345, Peugeot AS 234 FG, Citroen DF
456 GH, LAFONT Pierre 23 rue de la pomme TOULOUSE 31130]
```

7. Les multi-tri d'une Collection : interface Comparator

Pour trier une collection en utilisant des critères de tri différents, il faut 2 choses :

- créer une classe qui implémente l'interface Comparator
- utiliser la méthode

```
  static Collections.sort(List<T> list, Comparator<? super T> c)
```

en lui passant en paramètre une instance de cette classe

L'interface Comparator contient la méthode:

```
  public int compare(T o1, T o2)
```

Le traitement de tri est dans la classe Collection. C'est une méthode static qui prend en paramètre l'objet qui est le critère de tri

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

Voir sur le site http://jacques.laforgue.free.fr
l'exemple **Exemple12_SortCollection**

```
import java.util.*;

//Classe de définition d'une collection de livre gérés dans une bibliothèque
// Une collection de livre est caractérisée par :
//
//   - le tableau de livre
//
class Livres
{
        ArrayList<Livre> array;

    public Livres()
    {
                array = new ArrayList<Livre>();
    }
}

// Classe de définition des livres
class Livre
{
    String titre;
    String auteur;
    String ident;

    public Livre(String ident,String titre, String auteur)
    {
        this.titre  = titre;
        this.auteur = auteur;
        this.ident  = ident;
    }

    public String getTitre() {return titre;}
    public String getAuteur() {return auteur;}
    public String getIdent() {return ident;}

    public String toString()
    {
        return ident+"    "+titre+"      "+auteur;
    }
}

class CompLivreTitre implements Comparator<Livre>
{
    public int compare(Livre l1,Livre l2)
    {
        String s1 = l1.getTitre();
        String s2 = l2.getTitre();
        return( s1.compareTo(s2) );
    }
}
```

```
class CompLivreAuteur implements Comparator<Livre>
{
    public int compare(Livre l1,Livre l2)
    {
        String s1 = l1.getAuteur();
        String s2 = l2.getAuteur();
        return( s1.compareTo(s2) );
    }
}

class CompLivreIdent implements Comparator<Livre>
{
    public int compare(Livre l1,Livre l2)
    {
        String s1 = l1.getIdent();
        String s2 = l2.getIdent();
        return( s1.compareTo(s2) );
    }
}


public class Exemple12
{
    public static void main(String... a_args)
    {
        Terminal.ecrireStringln("Exemple 12");

        Livres meslivres = new Livres();

        Livre l1 = new Livre("2012/01/002","Cavernes d'acier (Les)","Asimov Isaac");
        Livre l2 = new Livre("2012/01/001","Fleuve de l'éternité (Le)","Farmer Philip José");
        Livre l3 = new Livre("2012/01/003","Dune","Herbert Frank");
        Livre l4 = new Livre("2012/01/004","Robot","Asimov Isaac");
        Livre l5 = new Livre("2012/01/005","Dieux du fleuve (Les)","Farmer Philip José");

        meslivres.array.add(l1);
        meslivres.array.add(l2);
        meslivres.array.add(l3);
        meslivres.array.add(l4);
        meslivres.array.add(l5);

        for(Livre l:meslivres.array)Terminal.ecrireStringln(l+"");

        // Tri par Titre
        Terminal.ecrireStringln("------ Tri par Titre ---------------------------------------");
        Collections.sort(meslivres.array,new CompLivreTitre());
        for(Livre l:meslivres.array)Terminal.ecrireStringln(l+"");

        // Tri par Auteur
        Terminal.ecrireStringln("----- tri par Auteur -----------------------------------------");
        Collections.sort(meslivres.array,new CompLivreAuteur());
        for(Livre l:meslivres.array)Terminal.ecrireStringln(l+"");

        // Tri par Ident
```

```
          Terminal.ecrireStringln("---- tri par Ident ---------------------
---------------------");
          Collections.sort(meslivres.array,new CompLivreIdent());
          for(Livre l:meslivres.array)Terminal.ecrireStringln(l+"");

      }
  }
```

```
Exécution :

Exemple 12
2012/01/002    Cavernes d'acier (Les)      Asimov Isaac
2012/01/001    Fleuve de l'éternité (Le)      Farmer Philip José
2012/01/003    Dune      Herbert Frank
2012/01/004    Robot      Asimov Isaac
2012/01/005    Dieux du fleuve (Les)      Farmer Philip José
------ Tri par Titre ----------------------------------------
2012/01/002    Cavernes d'acier (Les)      Asimov Isaac
2012/01/005    Dieux du fleuve (Les)      Farmer Philip José
2012/01/003    Dune      Herbert Frank
2012/01/001    Fleuve de l'éternité (Le)      Farmer Philip José
2012/01/004    Robot      Asimov Isaac
----- tri par Auteur -----------------------------------------
2012/01/002    Cavernes d'acier (Les)      Asimov Isaac
2012/01/004    Robot      Asimov Isaac
2012/01/005    Dieux du fleuve (Les)      Farmer Philip José
2012/01/001    Fleuve de l'éternité (Le)      Farmer Philip José
2012/01/003    Dune      Herbert Frank
---- tri par Ident ------------------------------------------
2012/01/001    Fleuve de l'éternité (Le)      Farmer Philip José
2012/01/002    Cavernes d'acier (Les)      Asimov Isaac
2012/01/003    Dune      Herbert Frank
2012/01/004    Robot      Asimov Isaac
2012/01/005    Dieux du fleuve (Les)      Farmer Philip José
```

<Commentaire durant le cours>

8. La classe Vector

La classe Vector est dans le package java.util.

java.util

Class Vector<E>

```
java.lang.Object
  └java.util.AbstractCollection<E>
      └java.util.AbstractList<E>
          └java.util.Vector<E>
```

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

Stack

Cette classe est également une gestion d'un tableau dynamique. Il n'y a pas trop de différence entre ArrayList et Vector **sauf que Vector est synchronized alors que ArrayList ne l'est pas.**

Remarque : pour synchroniser un ArrayList :

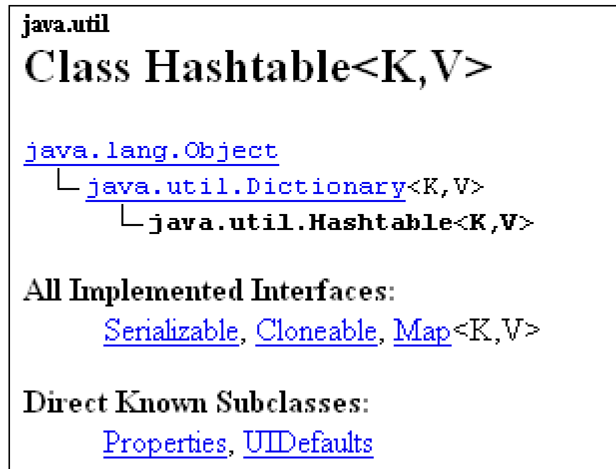
```
List synchronizedList = Collections.synchronizedCollections(newArrayList());
```

La classe java.util.Vector est une classe héritée de Java 1. Elle n'est conservée dans l'API actuelle que pour des raisons de compatiblité ascendante et elle ne devrait pas être utilisée dans les nouveaux programmes. Dans tous les cas, il est préférable d'utiliser un ArrayList.

9. La classe Hashtable

9.1. Présentation

Cette classe est dans le package java.util



java.util

Class Hashtable<K,V>

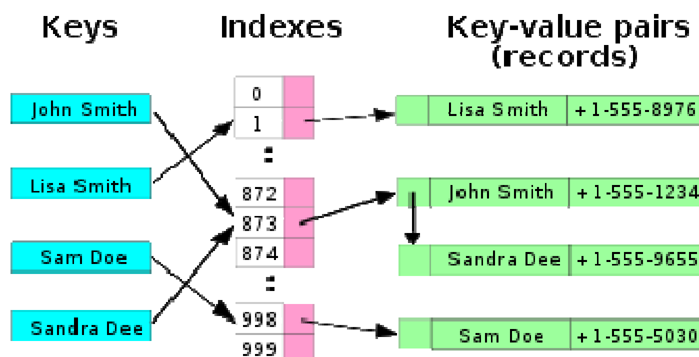
java.lang.Object
└ java.util.Dictionary<K,V>
　└ **java.util.Hashtable<K,V>**

All Implemented Interfaces:
　Serializable, Cloneable, Map<K,V>

Direct Known Subclasses:
　Properties, UIDefaults

La Hashtable est une collection de couple. Chaque couple est composé d'une clef et d'une valeur.

L'objectif est d'accéder à la valeur en utilisant la clef et non un indice comme cela est le cas pour ArrayList.



9.2. Exemple (voir ExempleHashtable.java)

```
import java.util.*;

public class ExempleHashtable
{
    public static void main(String[] args)
    {
        Hashtable<String,Couleur> colormap;
        colormap = new Hashtable<String,Couleur>();

        colormap.put("black", new Couleur(0,0,0));
        colormap.put("white", new Couleur(255,255,255));
        colormap.put("blue", new Couleur(0,0,255));
        colormap.put("navajo white", new Couleur(255,222,173));
```

```
            Terminal.ecrireStringln("--1-- Recherche d'une valeur");
            // Il est inéressant de voir cette exécution avec et sans la
            // méthode equals de la classe Couleur.
            // Sans la méthode 'equals', la couleur n'est pas trouvée
            //
            Couleur c1 = new Couleur(0,0,255);
            if (colormap.contains(c1))
                Terminal.ecrireStringln(c1 + " est dans la colormap");
            else
                Terminal.ecrireStringln(c1 + " n est pas dans la colormap");

            Terminal.ecrireStringln("--2-- Parcours des valeurs (enum)");
            Enumeration<Couleur> enumCoul = colormap.elements();
            while ( enumCoul.hasMoreElements())
                {
                    Couleur c = enumCoul.nextElement();
                    Terminal.ecrireStringln(c.toString());
                }
            Terminal.ecrireStringln("--3--      Parcours      des      valeurs
(collection)");
            for(Couleur c : colormap.values())
                Terminal.ecrireStringln(c.toString());

            Terminal.ecrireStringln("--4-- Parcours des valeurs (key)");
            Enumeration<String> keys = colormap.keys();
            while ( keys.hasMoreElements())
                {
                    String key = keys.nextElement();
                    Terminal.ecrireStringln(key + " : "+colormap.get(key));
                }

        }
    }

    class Couleur
    {
        int r,v,b;
        public Couleur(int r,int v,int b)
        {
            this.r=r;
            this.v=v;
            this.b=b;
        }
        public String toString()
        {
            return "["+r + " " + v + " " + b+"]";
        }
        public boolean equals(Object o)
        {
            Couleur c = (Couleur)o;
            return ((r ==c.r) && (v==c.v)&&(b==c.b));
        }
    }
```

Résultat de l'exécution :

```
java ExempleHashtable
--1-- Recherche d'une valeur
[0 0 255] est dans la colormap
--2-- Parcours des valeurs (enum)
[0 0 255]
```

```
[255 255 255]
[255 222 173]
[0 0 0]
--3-- Parcours des valeurs (collection)
[0 0 255]
[255 255 255]
[255 222 173]
[0 0 0]
--4-- Parcours des valeurs (key)
blue : [0 0 255]
white : [255 255 255]
navajo white : [255 222 173]
black : [0 0 0]
```

9.3. Les méthodes de la classe

Constructor Summary
Hashtable()   Constructs a new, empty hashtable with a default initial capacity (11) and load factor (0.75).
Hashtable(int initialCapacity)   Constructs a new, empty hashtable with the specified initial capacity and default load factor (0.75).
Hashtable(int initialCapacity, float loadFactor)   Constructs a new, empty hashtable with the specified initial capacity and the specified load factor.
Hashtable(Map<? extends K,? extends V> t)   Constructs a new hashtable with the same mappings as the given Map.

Method Summary	
void	**clear**() Clears this hashtable so that it contains no keys.
Object	**clone**() Creates a shallow copy of this hashtable.
boolean	**contains**(Object value) Tests if some key maps into the specified value in this hashtable.
boolean	**containsKey**(Object key) Tests if the specified object is a key in this hashtable.
boolean	**containsValue**(Object value) Returns true if this hashtable maps one or more keys to this value.
Enumeration<V>	**elements**() Returns an enumeration of the values in this hashtable.
Set<Map.Entry<K,V>>	**entrySet**() Returns a Set view of the mappings contained in this map.
boolean	**equals**(Object o)

	Compares the specified Object with this Map for equality, as per the definition in the Map interface.
V	**get**(Object key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
int	**hashCode**() Returns the hash code value for this Map as per the definition in the Map interface.
boolean	**isEmpty**() Tests if this hashtable maps no keys to values.
Enumeration<K>	**keys**() Returns an enumeration of the keys in this hashtable.
Set<K>	**keySet**() Returns a Set view of the keys contained in this map.
V	**put**(K key, V value) Maps the specified key to the specified value in this hashtable.
void	**putAll**(Map<? extends K,? extends V> t) Copies all of the mappings from the specified map to this hashtable.
protected void	**rehash**() Increases the capacity of and internally reorganizes this hashtable, in order to accommodate and access its entries more efficiently.
V	**remove**(Object key) Removes the key (and its corresponding value) from this hashtable.
int	**size**() Returns the number of keys in this hashtable.
String	**toString**() Returns a string representation of this Hashtable object in the form of a set of entries, enclosed in braces and separated by the ASCII characters ", " (comma and space).
Collection<V>	**values**() Returns a Collection view of the values contained in this map.

10. La classe HashSet

10.1. Présentation

Cette classe est dans le package java.util.

java.util

Class HashSet<E>

```
java.lang.Object
  └ java.util.AbstractCollection<E>
      └ java.util.AbstractSet<E>
          └ java.util.HashSet<E>
```

Type Parameters:

E - the type of elements maintained by this set

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, Set<E>

Direct Known Subclasses:

JobStateReasons, LinkedHashSet

La classe HashSet permet de gérer une collection sous la forme d'un ensemble. Cela veut dire que si on ajoute un élément déjà existant alors la collection n'est pas modifiée.

Par choix d'implémentation, les éléments sont rangés dans une table de hachage, permettant ainsi un accès plus rapide à l'élément.

10.2. Exemple (voir ExempleHashSet.java)

```
import java.util.*;

public class ExempleHashSet
{
    public static void main(String[] args)
    {
        HashSet<String> setChaine;
        setChaine = new HashSet<String>();

        setChaine.add("un");
        setChaine.add("trois");
        setChaine.add("deux");
        setChaine.add("quatre");
        setChaine.add("quatre");

        Terminal.ecrireStringln("--1-- Affichage de setChaine");
        for(String s:setChaine)
            Terminal.ecrireStringln(s);



        HashSet<Bidule2> setBidule2;
        setBidule2 = new HashSet<Bidule2>();

        setBidule2.add(new Bidule2(1));
        setBidule2.add(new Bidule2(3));
        setBidule2.add(new Bidule2(2));
        setBidule2.add(new Bidule2(4));
        setBidule2.add(new Bidule2(4));
```

```
        Terminal.ecrireStringln("--2-- Affichage de setBidule2");
        for(Bidule2 b:setBidule2)
            Terminal.ecrireStringln(b.toString());


        // Avec un set de Truc2 alors que Truc2 n'a pas defini
        //   sa méthode equals et sa méthode hashCode
        HashSet<Truc2> setTruc2;
        setTruc2 = new HashSet<Truc2>();

        setTruc2.add(new Truc2("un"));
        setTruc2.add(new Truc2("trois"));
        setTruc2.add(new Truc2("deux"));
        setTruc2.add(new Truc2("quatre"));
        setTruc2.add(new Truc2("quatre"));

        Terminal.ecrireStringln("--3-- Affichage de setTruc2");
        for(Truc2 t:setTruc2)
            Terminal.ecrireStringln(t.toString());


        // Rechercher un bidule
        Terminal.ecrireStringln("--4-- Recherche d'un bidule");
        if (setBidule2.contains(new Bidule2(4)))
            Terminal.ecrireStringln("Bidule 4 trouvé");
        else
            Terminal.ecrireStringln("Bidule 4 non trouvé");


        // Rechercher un truc
        Terminal.ecrireStringln("--5-- Recherche d'un truc");
        if (setTruc2.contains(new Truc2("quatre")))
            Terminal.ecrireStringln("Truc quatre trouvé");
        else
            Terminal.ecrireStringln("Truc quatre non trouvé");

    }
}

class Bidule2
{
    int x;
    public Bidule2(int x){this.x=x;}
    public String toString()
    {
        return ""+x;
    }

    // Il faut definir equals et hashCode
    //

    public boolean equals(Object o)
    {
        return x==((Bidule2)o).x;
    }

    public int hashCode()  // <=== IMPORTANT
    {
        return x;
    }
}
```

```
class Truc2
{
    String  a;
    public Truc2(String a){this.a = new String(a);}
    public String toString()
    {
        return ""+a;
    }
}

Résultat de l'exécution :
java ExempleHashSet
--1-- Affichage de setChaine
deux
trois
quatre
un
--2-- Affichage de setBidule2
1
2
3
4
--3-- Affichage de setTruc2
deux
quatre
un
trois
quatre
--4-- Recherche d'un bidule
Bidule 4 trouvé
--5-- Recherche d'un truc
Truc quatre non trouvé
```

10.3. Les méthodes de la classe

Constructor Summary
`HashSet()` Constructs a new, empty set; the backing `HashMap` instance has default initial capacity (16) and load factor (0.75).
`HashSet(Collection<? extends E> c)` Constructs a new set containing the elements in the specified collection.
`HashSet(int initialCapacity)` Constructs a new, empty set; the backing `HashMap` instance has the specified initial capacity and default load factor (0.75).
`HashSet(int initialCapacity, float loadFactor)` Constructs a new, empty set; the backing `HashMap` instance has the specified initial capacity and the specified load factor.

Method Summary	
`boolean`	`add(E e)` Adds the specified element to this set if it is not already present.

void	**clear**() Removes all of the elements from this set.
Object	**clone**() Returns a shallow copy of this `HashSet` instance: the elements themselves are not cloned.
boolean	**contains**(Object o) Returns `true` if this set contains the specified element.
boolean	**isEmpty**() Returns `true` if this set contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this set.
boolean	**remove**(Object o) Removes the specified element from this set if it is present.
int	**size**() Returns the number of elements in this set (its cardinality).

10.4. Exemple2 (voir Redondance.java)

Par exemple pour éliminer les éléments redondants d'un ArrayList

```
import java.util.*;

// Comment supprimer les redondances dans un ArrayList<String> avec
HashSet
//
public class Redondance
{
    public static void main(String[] args)
    {
        ArrayList<String> liste = new ArrayList<String>();
        liste.add("un");
        liste.add("un");
        liste.add("deux");
        liste.add("trois");
        liste.add("quatre");
        liste.add("un");
        liste.add("deux");

        Terminal.ecrireStringln("---------------");
        for(String s:liste)
            Terminal.ecrireStringln(s);

        HashSet<String> set = new HashSet<String>(liste);
        liste = new ArrayList<String>(set);

        Terminal.ecrireStringln("---------------");
        for(String s:liste)
            Terminal.ecrireStringln(s);

    }
}

Exécution :
java Redondance
---------------
un
```

```
un
deux
trois
quatre
un
deux
---------------
deux
trois
quatre
un
```

Par contre l'ordre des éléments n'est pas conservé.