Chapitre 5

Collections & Génériques

Introduction

- Une collection est un objet qui contient un ensemble d'éléments (de type Object)
- Une collection ressemble à un tableau mais offre plus de fonctionnalités et de flexibilité que les tableaux.
- Il existe plusieurs types de collections qui diffèrent selon plusieurs critères :
  - L'acceptation de doublons
  - L'ordre
  - L'acceptation de la valeur null
  - La méthode de stockage et récupération des valeurs
  - etc…

Nature des collections

- Une collection normale :
  - Permet le stockage d'objets de classes différentes
  - Lors du stockage ils sont transformés en type Object.
  - Lors de la récupération, il faudrait les caster à leurs classes d'origine.
- Une collection générique :
  - Ne permet le stockage que d'objets de la même classe
  - Lors du stockage, ils conservent leurs types d'origine
  - Lors de la récupération, pas de besoin de cast
  - Les génériques n'acceptent pas les types primitifs

Nature des collections

- Exemple de collection non générique:

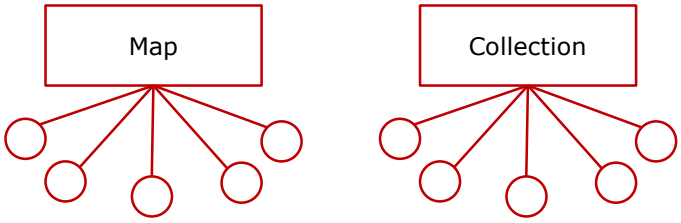
```
ArrayList list = new ArrayList();
list.add(0,new Etudiant("2014/354", "Ali", 15.5));
Object obj=list.get(0);
Etudiant e=(Etudiant)obj;
System.out.println(e.getNom());
```

- Exemple de collection générique

```
ArrayList< Etudiant > list = new ArrayList<>();
list.add(0, new Etudiant("2014/354", "Ali", 15.5));
Etudiant e=list.get(0);//pas besoin de cast
System.out.println(e.getNom());
```

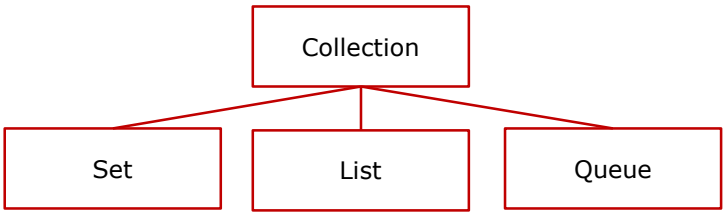
Hiérarchie de l'API collections

- Dans l'API collections il existe deux grandes familles de collections :
  - La famille d'interfaces et de classes héritant de l'interface Collection
  - La famille d'interfaces et de classes héritant de l'interface Map



Famille de l'interface Collection

- Se compose de trois sous-famille :
  - Sous-famille des ensembles : implémente l'interface Set
  - Sous-famille des listes : implémente l'interface List
  - Sous-famille des files : implémente l'interface Queue



Famille de l'interface Collection

- Quelques méthodes de l'interface collection :
  - int size() : retourne la taille de la collection.
  - boolean add(E e) : ajoute un nouvel élément à la collection.
  - boolean addAll(Collection c) : ajoute une collection d'éléments à la fois à la collection.
  - Object[] toArray() : retourne les objets de la collection dans un tableau.
  - void clear() : vide la collection.
  - boolean contains(Object o) : cherche un objet dans la collection.

Famille de l'interface Collection

  - boolean isEmpty() : teste si la collection est vide.
  - boolean remove(Object o) : supprime un objet de la collection.
  - boolean removeAll(Collection c) : supprime une collection d'elements de la collection.
- Remarque :
  - Nous retrouvons ces méthodes dans les interfaces Set, List et Queue qui héritent de l'interface collection et par conséquent, dans toutes les classes qui les implémentent.

Sous-famille des Set

- Est représentée par l'interface Set
- Set (un nom) veut dire Ensemble en français
- Un set est caractérisé par :
  - Il n'accepte pas de doublons
  - Même null il n'accepte qu'un seul

La famille des Set

- Classes implémentant l'interface Set:
  - AbstractSet, ConcurrentHashMap.KeySetView, ConcurrentSkipListSet, CopyOnWriteArraySet, EnumSet, JobStateReasons, LinkedHashSet, ReadOnlySetProperty, ReadOnlySetPropertyBase, ReadOnlySetWrapper, SetBinding, SetExpression, SetProperty, SetPropertyBase, SimpleSetProperty,
- Classes les plus connues:
  - HashSet : utilise un algorithme de hachage pour l'accès à ses éléments (pas d'ordre)
  - TreeSet : trie ses éléments pendant l'ajout

Exemple de Set

- Exemple avec HashSet :

```
import java.util.*;
public class SetExample {
    public static void main(String[] args) {
      Set set = new HashSet();
      set.add("un"); set.add("deux"); set.add("trois");
      set.add(4); set.add(5.0F);
      set.add("deux"); // dupliqué, non ajouté
      set.add(4); // dupliqué, non ajouté
      System.out.println(set);
    }
}
```

- Résultat du programme : [un, deux, 5.0, trois, 4]

Exemple de Set générique

- Exemple avec HashSet générique :

```
import java.util.*;
public class SetExample {
    public static void main(String[] args) {
      Set <String> set= new HashSet<>();
      set.add("un"); set.add("deux"); set.add("trois");
      set.add("deux"); // dupliqué, non ajouté
      set.add(4); // provoque une erreur de compilation
      System.out.println(set);
    }
}
```

Exemple de Set

- Exemple avec TreeSet générique :

```
import java.util.*;
public class SetExample {
    public static void main(String[] args) {
      Set set = new TreeSet();
      set.add("Bachir"); set.add("Omar"); set.add("Ali");
      set.add("Ali"); // dupliqué, non ajouté
      System.out.println(set);
    }
}
```

- Résultat du programme : [Ali, Bachir, Omar]
- Remarque : TreeSet ne permet pas des types hétérogènes pour pouvoir effectuer le tri automatique

Exemple de Set générique

- Exemple avec TreeSet générique :

```
import java.util.*;
public class SetExample {
    public static void main(String[] args) {
      Set<String> set = new TreeSet();
      set.add("Bachir"); set.add("Omar"); set.add("Ali");
      set.add("Ali"); // dupliqué, non ajouté
      System.out.println(set);
    }
}
```

- Résultat du programme : [Ali, Bachir, Omar]
- Remarque : TreeSet effectue le tri automatiquement

La famille des List

- Est représentée par l'interface List
- C'est une collection ordonnée d'éléments ou chaque élément est indexé par un indice.
- Elle accepte les doublons et les nulls

La famille des List

- Classes implémentant l'interface List :
  - AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, FilteredList, LinkedList, ListBinding, ListExpression, ListProperty, ListPropertyBase, ModifiableObservableListBase, ObservableListBase, ReadOnlyListProperty, ReadOnlyListPropertyBase, ReadOnlyListWrapper, RoleList, RoleUnresolvedList, SimpleListProperty, SortedList, Stack, TransformationList, Vector

La famille des List

- Classes les plus connues:
  - ArrayList : tableau redimensionnable non synchronisé
  - Vector : tableau redimensionnable synchronisé
  - Stack : Pile
  - SortedList : Liste Triée

La famille des List

- Quelques méthodes supplémentaires:
  - boolean add(int index, E e) : ajoute un nouvel élément au à la position index de la liste.
  - boolean addAll(int index, Collection c) : ajoute les éléments de la collection c au à la position index liste.
  - boolean remove(int index) : supprime l'objet à la position index de la liste.
  - E get(int index) : retourne l'objet à la position index.
  - E set(int index, E element) : remplace l'objet à la position index par l'objet E.

Un exemple de List : ArrayList

```
import java.util.*;
public class ListExample {
    public static void main(String[] args) {
     List list = new ArrayList();
     list.add("un"); list.add("deux"); list.add("trois");
     list.add(4); list.add(5.0F);
     list.add("deux"); // dupliqué, ajouté
     list.add(4); // dupliqué, ajouté
     System.out.println(list);
    }
}
```

- Résultat du programme : [un, deux, tois, 4, 5.0, deux, 4]

Un exemple de List : ArrayList générique

```
import java.util.*;
public class ListExample {
    public static void main(String[] args) {
     List<String> list = new ArrayList();
     list.add("un"); list.add("deux"); list.add("trois");
     list.add("deux"); // dupliqué, ajouté
     list.add(4); // provoque une erreur de compilation
     System.out.println(list);
    }
}
```

Un exemple de List : Stack

- Représente une PILE (LIFO)
- Méthodes :
  - boolean empty() : teste si la pile est vide.
  - Object push(Object item) : empile un objet en haut de la pile.
  - Object pop() : dépile un objet du haut de la pile
  - Object peek() : retourne une copie du sommet de la pile sans la dépiler.
  - int search(Object o) : recherche un objet sur la pile et renvoie sa position.

Un exemple de List : Stack

- Exemple :

```
import java.util.Stack;
public class Main {
   public static void main(String[] args) {
      Stack pile= new Stack();
      String[] noms={"Ali","Omar","Hassan","Samir"};
      for (String nom : noms) { pile.push(nom); }
      for (String nom : noms) {
        System.out.println(nom+" se trouve à la position : "+pile.search(nom));
      }
      System.out.println("L'élément au sommet est : "+pile.peek());
      System.out.println("Dépilement de : "+pile.pop());
      System.out.println("Dépilement de : "+pile.pop());
      System.out.println("L'élément au sommet est : "+pile.peek());
      System.out.println("La pile est-elle vide ? "+pile.empty());
   }
}
```

Un exemple de List : Stack

- Résultat de l'exécution :
  - Ali se trouve à la position : 4
  - Omar se trouve à la position : 3
  - Hassan se trouve à la position : 2
  - Samir se trouve à la position : 1
  - L'élément au sommet est : Samir
  - Dépilement de Samir
  - Dépilement de Hassan
  - L'élément au sommet est :Omar
  - La pile est-elle vide ? false

Un exemple de List : Stack générique

- Même exécution sauf pour la déclaration
- Stack<String> pile= new Stack();

La famille des Queue

- Représentent les files (FIFO)
- Elles implémentent l'interface Queue
- Quelques classes implémentant l'interface Queue :
  - AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, SynchronousQueue
- Classes les plus connues:
  - LinkedList: implémentée sous forme d'une liste chainée

L'interface Queue

- Quelques méthodes :
  - Boolean **add(**E e**)** : ajoute un élément à la fin de la file. Retourne true ou une exception.
  - boolean **offer(**E e**)** : ajoute un élément à la fin de la file. Retourne true ou false.
  - E **element()** : retourne la tête de la file sans l'enlever. génère une exception si vide.
  - E **peek()** : retourne la tête de la file sans l'enlever. Retourne null si vide.
  - E **poll()** : retourne et enlève la tête de la file (null si vide).
  - E **remove()** : retourne et enlève la tête de la file (exception si vide)

Exemple de Queue : LinkedList

```
import java.util.*;
public class Main {
   public static void main(String[] args) {
      Queue FileDattente = new LinkedList();//implémente aussi List
      FileDattente.add("Ali"); FileDattente.add("Omar"); FileDattente.add("Adil");
      FileDattente.add("Hassan"); FileDattente.add("Taha");
      System.out.println("Contenu File D'attente : " + FileDattente);
      System.out.println(" Retrait tête de la file : " + FileDattente.remove());
      System.out.println(" Contenu File D'attente : " + FileDattente);
      System.out.println(" Retrait tête de la file : " + FileDattente.poll());
      System.out.println(" Contenu File D'attente :" + FileDattente);
   }
}
```


Exemple de Queue : LinkedList

- Résultat de l'exécution :
  - File D'attente : [Ali, Omar, Adil, Hassan, Taha]
  - Enlevé de la file d'attente: Ali
  - nouvelle file d'attente : [Omar, Adil, Hassan, Taha]
  - Enlevé de la file d'attente: Omar
  - nouvelle file d'attente :[Adil, Hassan, Taha]

Queue générique

- Même exécution sauf pour la déclaration
- Queue<String> FileDattente = new LinkedList<>();

La famille des Map

- Sont représentés par l'interface Map
- Les Maps sont parfois appelés tableaux associatifs
- Un objet Map décrit des mapping des clés (keys) aux valeurs (values)
- les clés dupliqués ne sont pas permises
- Les mappings un-à-plusieurs ne sont pas permis aussi.

La famille des Map

- Classes implémentant l'interface Map :
  - AbstractMap, Attributes, AuthProvider, ClipboardContent, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, Headers, IdentityHashMap, LinkedHashMap, MapBinding, MapExpression, MapProperty, MapPropertyBase, MultiMapResult, PrinterStateReasons, Properties, Provider, ReadOnlyMapProperty, ReadOnlyMapPropertyBase, ReadOnlyMapWrapper, RenderingHints, ScriptObjectMirror, SimpleBindings, SimpleMapProperty, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

La famille des Map

- Classes les plus connues:
  - Hashtable : utilise une table de hachage pour lier les clés aux valeurs
  - HashMap : équivalente à Hashtable sauf qu'elle n'est pas synchronisée
  - Properties : les clés et les valeurs sont de type String

L'interface Map

- Quelques méthodes :
  - V put(K key, V value) : associe la clé K à la valeur V
  - V get(Object key) : récupère l'objet dont la clé est key
  - keySet() : Retourne un Set de toutes les clés du Map.
  - values() : Retourne une Collection de toutes les valeurs du Map.
  - entrySet() : Retourne un Set de paires clé-valeur.

L'interface Map : exemple

```
import java.util.*;
public class MapExample {
  public static void main(String args[]) {
    Map map = new HashMap();
    map.put("premier","1er");
    map.put("second", 2);
    map.put("troisième","3eme");
    map.put("troisième","III"); // écrase l'affectation précédente
    Set set1 = map.keySet(); // retourne un Set de clés
   Collection collection = map.values(); //retourne une Collection de valeurs
    Set set2 = map.entrySet(); // retourne un set de clé-valeurs
    System.out.println(set1 + "\n" + collection + "\n" + set2);
  }
}
```

L'interface Map : exemple

- Résultat du programme :
  - [second, premier, troisième]
  - [2, 1er, III]
  - [second=2, premier=1er, troisième=III]

Map générique

- Exemple:

```
import java.util.HashMap;
import java.util.Map;
public class MapGnerique {
  public static void main(String[] args) {
     Map<Integer, String> joueurs = new HashMap<>();
     String[] positions={"Gardien", "Défenseur droit", "Defenseur gauche", "Arrière
      droit", "Arrière gauche", "Milieu droit", "Milieu central", "Milieu gauche",
      "Avant-centre","Aillier droit","Aillier gauche"};
      // remplissage du Map
     for (int i = 1; i <= 11; i++) {   joueurs.put(i,positions[i]);  }
 // parcours du Map et affichage des clés et des valeurs
     for ( int clef : joueurs.keySet()) {
        System.out.println("Numero : " + clef + " , Position : " + joueurs.get(clef));
     }
  }
}
```

Parcours des collections et génériques

- Plusieurs moyens de parcours:
  - L'interface enumeration
  - L'interface Iterator
  - Interface ListIterator
  - Boucle for amélioré
- Les collections et les génériques sont parcourues de la même façon.

Parcours des collections

- L'interface Enumeration
  - Appartient au package java.util
  - Permet le parcours séquentiel de collections.
  - Définit 2 méthodes :
    - boolean **hasMoreElements**() : retourne true si l'énumeration contient encore un ou plusieurs elements
    - Object **nextElement**() : retourne l'objet suivant de l'énumération (lève une Exception *NoSuchElementException* si la fin de la collection est atteinte)

Parcours des collections

- Exemple :

```
import java.util.*;
class Main{
 public static void main (String args[]) {
  // Création et parcour d'un objet Vector
  Vector v = new Vector();
  v.add("chaine 1"); v.add("chaine 2"); v.add("chaine 3");
  for(Enumeration e = v.elements(); e.hasMoreElements(); ) {
     System.out.println(e.nextElement());     }
  // Création et parcour d'un objet Hashtable
  Hashtable h = new Hashtable();  h.put("jour", new Date());
  h.put(1,  "Bonjour"); h.put("deux",2);
  for (Enumeration e = h.keys() ; e.hasMoreElements() ; ){
   System.out.println(h.get(e.nextElement())); }
 } }
```


Parcours des collections

- L'interface Iterator
  - Appartient au package java.util
  - Similaire à Enumeration mais plus récente.
  - Offre en plus la possibilité de **supprimer** des éléments en cours d'énumération
  - Les noms de méthodes ont été raccourcis :
    - boolean **hasNext() :** retourne *true* si l'itérateur a encore des éléments
    - Object **next() :** retourne le prochain élément de l'itérateur
    - void **remove() :** supprime de la collection sous-jacente le dernier élément retourné par l'itérateur

Parcours des collections

- Exemple :

```
import java.util.ArrayList ;
public class TestIterator {
  public static void main(String[] args){
     ArrayList saisons= new ArrayList();
    saisons.add("Automne"); saisons.add("Hiver");
    saisons.add("Printemps"); saisons.add("Été");
     Iterator it = saisons.iterator();
     System.out.println("les saisons sont :");
     while (it.hasNext()) {
            String saison= (String)it.next();
            System.out.println(saison);
     }
   }
}
```


Parcours des collections

- L'interface ListIterator
  - Appartient au package java.util
  - Similaire à Iterator mais offre un parcours dans les 2 sens et permet modifier les éléments lors de ce parcours
  - Méthodes :
    - boolean **hasNext**() : retourne *true* si l'itérateur a encore des éléments en avant
    - Object **next**() : retourne le prochain élément en avant
    - int **nextIndex**() : retourne l'index du prochain élément à retourner next()

Parcours des collections

- boolean **hasPrevious**() : retourne *true* si l'itérateur a encore des éléments en arrière.
- Object **previous**() : retourne le prochain élément de l'itérateur en arrière
- int **previousIndex**() : retourne l'index du e l'élément à retourner par le prochain appel à previous()
- void **add**(Object o) : ajoute un élément à la liste.
- void **set**(Object o) : remplace le dernier element retourné par next() ou previous() par l'objet en paramètre.

Parcours des collections

- Exemple :

```
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;
public class ListIteratorExemple {
    public static void main(String a[]){
        List noms= new ArrayList();
        noms.add("Ali"); noms.add("Omar"); noms.add("Hassan");
        //Obtention de l'itérateur
        ListIterator litr=noms.listIterator();
        System.out.println("Parcours en avant:");
        while(litr.hasNext()){ System.out.println((String)litr.next()); }
        System.out.println("Parcours en arrière:");
        while(litr.hasPrevious()){
         System.out.println((String)litr.previous());
        }
    }
}
```

Parcours des collections

- La boucle For des collections :
  - Une itération Simplifiée à travers les collections
  - Plus courte, claire et sûre
  - Supprime les inconvénients de l'itérateur
- Exemple :

```
import java.util.ArrayList;
import java.util.List;
public class ForExemple {
    public static void main(String a[]){
        List noms= new ArrayList();
        noms.add("Ali"); noms.add("Omar"); noms.add("Hassan");
        System.out.println("Parcours avec For:");
        for(Object o : noms){
        System.out.println((String)o);
        }
    }
}
```


Tri des collections

- Certaines collections permettent le tri de leurs éléments
- Pour cela les interfaces **Comparable** et **Comparator** peuvent être utilisées.

« interface » **Comparable**
int compareTo(Object o)

« interface » **Comparator**
int compare(Object o1, Object o2)

L'interface Comparable

- Impose l'ordre naturel aux classes qui l'implémente :
  - Est composée de la méthode : int compareTo(Object o)
  - Les objets d'une classe implémentant cette interface sont triables.
- Ordre de tri naturel selon le type des éléments :
  - Les éléments String : ordre alphabétique
  - Les éléments Date : ordre chronologique
  - Les éléments Integer : ordre numérique
- Remarque : Les classes String, Date, et Integer implémentent l'interface Comparable

Comparable : un exemple

```
public class Etudiant implements Comparable{
 private String CNE;
 private String nom;
 private double moyenne;
// constructeur
 public Etudiant (String CNE, String nom, double moyenne) {
   this.CNE = CNE; this.nom = nom; this.moyenne = moyenne;
 }
//getters setters
 public String getNom() {  return nom;  }
 public void setNom(String nom) {  this.nom = nom; }
 public String getCNE() { return CNE;  }
public void setCNE( String CNE) {  this.CNE = CNE; }
```

Comparable : un exemple

```
public double getMoyenne () { return moyenne; }
public void setMoyenne (double moyenne) { this.moyenne = moyenne; }
@Override
public String toString() {
  return "CNE="+CNE+"Nom="+nom+"Moyenne="+moyenne;
 }
@Override
 public int compareTo(Object o) {
            Etudiant e2= (Etudiant) o;
           if (this.moyenne==e2.moyenne) return 0; // 0 égaux
           else if (this.moyenne<e2.moyenne)  return -1; // inférieure
                   else      return 1; // supérieure
 }
}
```


Comparable : un exemple

```
import java.util.*;
public class ComparableTest {
 public static void main(String[] args) {
  TreeSet etudiants = new TreeSet();
  etudiants.add(new Etudiant("2014/354", "Ali",15.5));
  etudiants.add(new Etudiant("2014/358", "Omar",12.5 ));
  etudiants.add(new Etudiant("2014/398", "Taha",13.6));
  etudiants.add(new Etudiant("2014/253", "Anass",14.3));
  Object[] tabEtudiants = etudiants.toArray();
  for(Object o: tabEtudiants ){
      System.out.println((Etudiant) o);
  }
 }
}
```

Attention : Lorsque des moyennes se répètent, le TreeSet ignore les doublons aussi.

Comparable : un exemple

- Résultat du programme :
  - CNE= 2014/358 Nom= Omar Moyenne= 12.5
  - CNE= 2014/398 Nom= Taha Moyenne= 13.6
  - CNE= 2014/253 Nom= Anass Moyenne= 14.3
  - CNE= 2014/354 Nom= Ali Moyenne= 15.5

Comparable générique

- Dans l'interface comparable générique, le besoin pour le cast est éliminé. Exemple :

```
public class Etudiant implements Comparable<Etudiant> {
 private String CNE;
 private String nom;
 private double moyenne;
      .
      .
      .
@Override
 public int compareTo(Etudiant e) {
          if (this.moyenne==e.moyenne) return 0; // 0 égaux
          else if (this.moyenne<e.moyenne)  return -1; // inférieure
                  else      return 1; // supérieure
 }
}
```

L'interface Comparator

- Est utilisée pour les objets qui n'implémente pas l'interface Comparable
- Est composée de la méthode :
  - int compare(Object o1, Object o2)
- Peut être passée à une méthode de tri
- Les objets à trier n'implémentent pas cette interface : c'est des classes externes qui le font.
- Dans l'exemple suivant nous ne touchons pas à la classe Etudiant mais d'autres classes implémentent Comparator

Comparator : exemple

```
import java.util.*;
public class CompMoyenne implements Comparator {
    @Override
    public int compare(Object o1, Object o2) {
        double m1=((Etudiant )o1).getMoyenne();
        double m2=((Etudiant )o2).getMoyenne();
         if (m1==m2) return 0;
          else if  (m1<m2)        return -1;
               else return 1;
    }
}
```

Comparator : exemple

```
import java.util.*;
public class CompNom implements Comparator {
      @Override
       public int compare(Object o1, Object o2) {
        String nom1=((Etudiant )o1).getNom();
        String nom2=((Etudiant )o2).getNom();
        return (nom1.compareTo(nom2));
}
}
```


Comparator : exemple

```
import java.util.*;
public class ComparableTest {
 public static void main(String[] args) {
  Comparator c = new CompNom();
  TreeSet etudiants = new TreeSet(c);
  etudiants.add(new Etudiant("2014/354", "Ali",15.5));
  etudiants.add(new Etudiant("2014/358", "Omar",12.5 ));
  etudiants.add(new Etudiant("2014/398", "Taha",13.6));
  etudiants.add(new Etudiant("2014/253", "Anass",14.3));
  Object[] tabEtudiants = etudiants.toArray();
  for(Object o: tabEtudiants ){
      System.out.println((Etudiant) o);
  }
 }
}
```

Comparator : exemple

- Résultat du programme :
  - CNE= 2014/354 Nom= Ali Moyenne= 15.5
  - CNE= 2014/253 Nom= Anass Moyenne= 14.3
  - CNE= 2014/358 Nom= Omar Moyenne= 12.5
  - CNE= 2014/398 Nom= Taha Moyenne= 13.6

Comparator générique

- Dans l'interface Comparator générique, le besoin pour le cast est éliminé. Exemple :

```
import java.util.*;
 public class CompMoyenne implements Comparator <Etudiant>{
            @Override
            public int compare(Etudiant e1, Etudiant e2) {
                double m1=e1.getMoyenne();
                double m2=e2.getMoyenne();
                if (m1==m2) return 0;
                  else if  (m1<m2)        return -1;
                       else return 1;
            }
}
```

Tri Avec ArrayList

- La classe ArrayList dispose d'une méthode sort :
  - Elle prend en paramètre un objet comparator
  - Si on lui passe null elle utilise plutôt l'interface comparable
- Exemple :

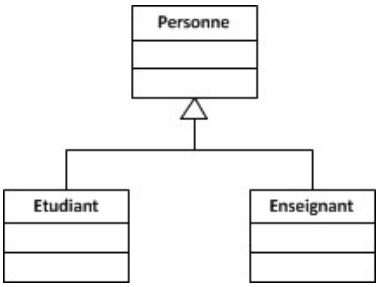
```
List list = new ArrayList();
list.add(new Etudiant("2014/354", "Ali", 15.5));
list.add(new Etudiant("2014/358", "Omar", 12.5));
list.add(new Etudiant("2014/398", "Taha", 13.6));
list.add(new Etudiant("2014/253", "Anass", 14.3));
```


Tri Avec ArrayList

```
list.sort(new CompNom());
 // ou list.sort(new CompMoyenne ());
// ou list.sort(null);
// dans ce cas elle prend comparable s'il existe
for (Object o : list) {
   Etudiant e =(Etudiant)o;
   System.out.println(e.toString());
}
```

Génériques dans le cas de l'héritage

- Soit les classes suivantes :



La sûreté dans les types

- Une fois le type d'objet d'un générique est précisé, le compilateur signale une erreur si un autre type est ajouté.
- Exemple :

```
public class SuretéDesTypes{
  public static void main(String[] args) {
   List<Enseignant> enseignants = new ArrayList<>();
   enseignants.add(new Enseignant()); // OK
   enseignants.add(new Etudiant()); // Erreur de Compilation!
 }}
```

L'invariance

- Dans la POO, un objet de la classe fille peut être affecté à un objet de la classe mère.
- Cependant, une collection générique d'objets filles ne peut pas être affectée à une collection d'objets mères.
- Exemple :
  - Personne p=new Etudiant() est légal
  - **personnes** = **etudiants** est illégal

Paramètre d'un générique

- Soit les deux fonctions suivantes qui permettent de parcourir un générique de type Etudiant (resp. Enseignant) :

```
public static void afficherNoms(List <Etudiant> liste) {
  for (int i=0; i < liste.size(); i++) {
    System.out.println(liste.get(i).getNom());
  }
 }
```

```
public static void afficherNoms(List <Enseignant> liste) {
  for (int i=0; i < liste.size(); i++) {
    System.out.println(liste.get(i).getNom());
  }
 }
```

Paramètre d'un générique

- Java permet de les unifier en une seule fonction à condition de ne pas mixer les types en ajoutant un paramètre « ? » représentant les deux classes.

```
public static void afficherNoms(List <? extends Personne> liste) {
  for (int i=0; i < liste.size(); i++) {
    System.out.println(liste.get(i).getNom());
  }
 }
```


Paramètre d'un générique

- Exemple d'appel :

```
public static void main(String[] args) {
  List<Enseignant> enseignants= new ArrayList<>();
  List<Etudiant> etudiants = new ArrayList<>();
 // on suppose que nous avons rempli les deux génériques
  afficherNoms (enseignants);
  afficherNoms (etudiants );
}
```

Paramètre d'un générique

- Remarque :
  - lors de l'appel de la fonction, le compilateur a accepté l'affectation du générique fils au générique parent mais il n'accepte pas l'ajout d'objets
- Exemple :

```
List<? extends Personne> liste2 = enseignants; //OK
liste2.add(new Enseignant ());//Erreur de Compilation!
```