Cours Java Avancé Partie 2

Mme Nouira Sana

Cours Java Avancé - Les classe de base et Les Collections

Mme Nouira Sana

La classe Object

- **La classe Object** est la classe mère de toutes les classes, elle contient des méthodes que la plupart des classes Java doivent redéfinir
- La classe **Object** est la racine unique de l'arbre des classes. Toutes les classes héritent donc des méthodes de la classe Object (par exemple toString( ), equals( ), finalize( )…).
- Quelques méthodes qu'on peut soit utiliser telles quelles soit redéfinir :
  - **toString()** : retourne un String contenant le nom de la classe concernée et l'adresse de l'objet en hexadécimal (précédée @)
  - **equals()** : retourne un booléen qui compare les adresses de deux objets
  - **finalize()** : méthode d'instance qui ne possède pas de paramètres , n'a pas de type de retour (retourne void). Elle est invoquée avant la destruction automatique d'un objet ; elle est à re-définir.
  - **clone() :** instancie un nouvel objet, qui est une copie de this, uniquement si la classe de l'objet implémente l'interface Clonable

EXEMPLE

```
class Personne extends Object {    // dérive de Object par défaut
        private String nom ;

public Personne(String nom) { this.nom = nom ; }
@override
public String toString() {
        return "Classe : " + getClass().getName() + " Objet : " + nom ;
}
@override
boolean equals(Personne p) { return p.nom.equals(nom) ; }
}

Personne p1 = new Personne("Jean Dupond") ;
Personne p2 = new Personne("Jean Dupond") ;

System.out.println( p1 ) ;
// Affiche : Classe : Personne Objet : Jean Dupond

if ( p1 == p2 ) … // Faux, les références sont différentes
if (p1.equals(p2)) … // Vrai, comparaison sémantique
```

Classe Class

- Il existe un objet Class pour toute classe utilisée. Il permet d'accéder aux informations sur les classes.
- Au niveau de la classe Object on trouve une méthode publique **getClass**
  - qui renvoie un objet de type Class et permet d'accéder aux informations sur cette classe classes via les méthodes de Class
  - cette méthode **Class getClass()** peut être utilisée sur tout objet pour déterminer sa nature (nom de sa classe, nom de ses ancêtres, structure).
  - **getClass() :** Cette méthode retourne un objet, instance d'une classe particulière appelée Class.

- **getName()** retourne le nom (String) de la classe
- **getSuperclass()** renvoie des informations (de type Class) sur la super-classe de la classe
- **toString()** = getName() + "class" ou "interface" selon le cas
- **newInstance()** produit une nouvelle instance du type de l'objet.

Les chaînes de caractères

- **La classe java.lang.String**
- **La classe java.lang.StringBuffer**

  - La classe **StringBuffer** permet de représenter des chaînes de caractères de taille variable.

  - Contrairement à la classe **String**, il n'est pas possible d'utiliser l'opérateur + avec les objets de cette classe.

  - Lorsqu'un objet du type **StringBuffer** est construit, un tableau de caractères est alloué dans l'objet.
    - Sa taille initiale est de 16, mais si l'on ajoute plus de 16 caractères dans l'objet, un nouveau tableau, plus grand est alloué (les caractères sont recopiés de l'ancien tableau vers le nouveau et l'ancien tableau est détruit).

- **La classe java.util.StringTokenizer**

  - Cette classe permet de construire des objets qui savent découper des chaînes de caractères en sous-chaînes (C'est un objet ayant un caractère utilitaire => package java.util).

- Lors de la construction du **StringTokenizer**, il faut préciser la chaîne à découper et, dans une seconde chaîne, le ou les caractères qui doivent être utilisés pour découper cette chaîne :

```
void AfficheParMots(String texte) {
   StringTokenizer st = new  StringTokenizer(texte, ",:") ;

   while ( st.hasMoreTokens() ) {
         String mot = st.nextToken() ;
         System.out.println(mot) ;
   }
}
…
AfficheParMots("Lundi,Mardi:Mercredi;Jeudi");
// Affiche :
Lundi
Mardi
Mercredi;Jeudi
```

Les classes génériques

- **Un type générique** est une classe ou une interface qui prend en paramètre un type.
  - **Syntaxe : *className* <*type*>**

EXEMPLE

```
public class Box <T> {
   private T obj;
   public Box(){ }
   public Box (T obj)
     {  this.obj = obj;}
  public T get() {
        return obj; }
  public void set(T obj) {
        this.obj = obj;}
}

public class Pair <T1,T2>{
private T1 obj1;
private T2 obj2;
public Pair(){ }
public Pair (T1 obj1,T2 obj2)
{  this.obj1 = obj1;
this.obj2 = obj2;}
public T1 getType1() {
return obj1; }
public T2 getType2() {
return obj2; }
public void setType1(T1 obj1) {
this.obj1 = obj1;}
public void setType2(T2 obj2) {
this.obj2 = obj2;}
}
```

```
public class Point {
    double x, y;
Point (double x, double y) {this.x=x ; this.x=y;}
// méthode de translation
void translater (double dx, double dy){ x += dx;  y +=
dy; }
// méthode de distance par rapport à l'origine
double distance() { double dist; // variable locale
dist = Math.sqrt(x*x+y*y); return dist; }
}
public class Genérique {
   public static void main(String[] args) {
    Box<Integer> intBox = new Box<>();
      // ou new Box<Integer>();
    Box<String> strBox = new Box<>();
     // ou new Box<String>();
    Box<Point> ptBox = new Box<>();
     // ou new Box<Point>();
   intBox.set(5);
   strBox.set("Hello");
   ptBox.set(new Point(1,3));
   System.out.println(ptBox.get());
    // On peut créer sans le type (ça sera object)
   Box b = new Box(); // Raw type
   b.set(8); // Autoboxing int => Integer => Object
            //On peut passer plus qu'un type générique.
    Pair <String, Integer>  p1 = new
                              Pair<>("Even", 8);
    // après la définition de la classe Pair
   }
}
```

```
Output - genérique (run)
run:
5
Hello
x : 3.0 y : 0.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

- Pour définir un paramètre générique qui inclut une classe donnée ainsi que toutes ses sous-classes, on utilise la syntaxe :

  < T extends type >

  Exemple :

  ```
  public class ShapeBox<T extends Polygon> {
          private T shape; // Polygon or sub-class of Polygon }
  ```

- La formulation "extends type" (extends ou implements) doit être interprétée comme : La classe type ainsi que toutes ses sous-classes ou toute classe qui implémente l'interface type

- Le symbole '?' peut être utilisé comme caractère joker (wildcard) dans différentes situations en lien avec les classes et interfaces génériques : type d'un paramètre, d'un champ, d'une variable locale... Il représente un type (inconnu) quelconque.

  Exemple : List<?> : une liste d'éléments de n'importe quel type

  - <? extends type> accepte la classe type ainsi que toutes ses sous classes.
    - Cette notation est également valable pour toutes les classes qui implémentent l'interface type en remplaçant « extends » par « implements »

  <? super type> accepte la classe type ainsi que toutes les classes parentes de type.

  Exemple : List<? super Integer> list ; // accepte Integer, Number ou Object

Cours Java Avancé -

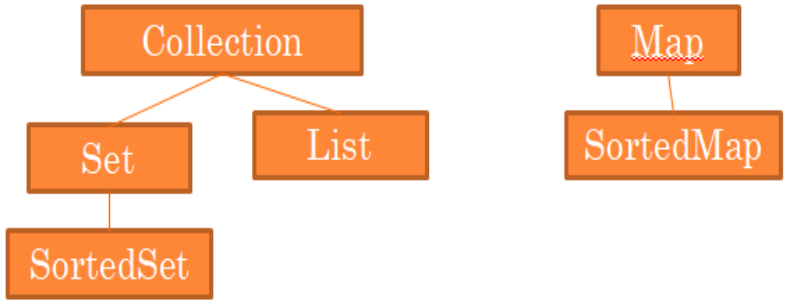
Les Collections

Mme Nouira Sana

Interface Collection

- Java dispose dans le package « java.util » des classes permettant de manipuler les principales structures de données à savoir :
  - les vecteurs dynamiques, les listes chainées, les ensembles, les queues et les tables associatives.
  - Ces structures ont évoluées au fil des versions du java jusqu'au stade que toutes ces structures arrivent à implémenter la même interface appelée **Collection**, et que chaque structure complète avec ses fonctionnalités propres.
- Depuis Java 5, les collections sont manipulées par le biais de classes génériques implémentant l'interface Collection <T>.
  - T représente le type des éléments de la collection.
  - Les interfaces collections frameWork de Java sont réparties en deux groupes : Collection et Map.

- Les structures de données implémentant ces interfaces sont :
  - Les vecteurs dynamiques : classes ArrayList et Vector
  - Les listes chainées : classe LinkedList
  - Les ensembles : classes HashSet et TreeSet
  - Les queues avec priorité : classe PriorityQueue
    - (à partir de java 5)
  - Les queues à double entrée : classe ArrayDecque
    - (à partir de java 6)



Interface Collection

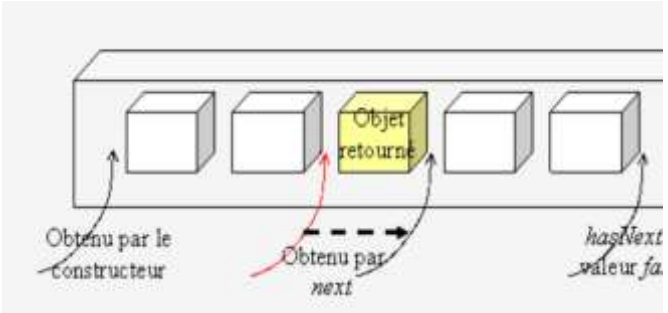
```
public interface Collection {
      // Basic Operations
       int size();
       boolean isEmpty();
       boolean contains(Object element);
       boolean    add(Object    element);    //
Optional
       boolean   remove(Object   element);   //
Optional
       Iterator iterator();
       int hashCode();
   boolean equals(Object element);
     // Bulk Operations
     boolean containsAll(Collection c);
     boolean addAll(Collection c); // Optional
     boolean   removeAll(Collection   c);   //
     Optional
     boolean   retainAll(Collection   c);   //
     Optional
     void clear(); // Optional
     // Array Operations
     Object[] toArray();
     Object[] toArray(Object a[]);
     }
```

L'INTERFACE ITERATOR

- De nombreuses classes conteneurs implémentent les méthodes de l'interface *Collection* : *Vector*, *Stack*, *ArrayList*, *LinkedList*, *HashSet*, *TreeSet*,
- **Chacune de ces classes doit implémenter la méthode *Iterator iterator()* qui retourne un itérateur sur l'ensembles des éléments**
- **Un itérateur est un objet qui implémente l'interface *Iterator*, et qui a pour but de permettre le parcours des éléments d'un conteneur, sans avoir besoin de savoir de quelle nature est ce conteneur**
- Cet outil est utilisé pour parcourir une collection.
- L'interface Iterator contient ce qui suit:

```
public interface Iterator {
    BOOLEAN HASNEXT(); //RETOURNE TRUE SI L'ITÉRATEUR N'EST PAS ARRIVÉ EN FIN DE L'ENSEMBLE ET FALSE SINON.
    OBJECT NEXT(); PERMET D'AVANCER L'ITÉRATEUR, ET QUI RETOURNE L'ÉLÉMENT QUI A ÉTÉ SAUTÉ. LÈVE UNE EXCEPTION NOSUCHELEMENTEXCEPTION SI L'ITÉRATEUR N'A PAS D'OBJET QUI LE SUIT.
    VOID REMOVE(); // SUPPRIME L'OBJET QUI VIENT D'ÊTRE OBTENU PAR NEXT. CETTE MÉTHODE EST FACULTATIVE
}
```

Exemple : Collection collection = new ...;
Iterator iterator = collection.iterator();
while (iterator.hasNext()) {
    Object element = iterator.next();
    if (removalCheck(element)) {
    iterator.remove();
}}



Les conteneurs

- De nombreuses classes conteneurs implémentent les méthodes de l'interface *Collection* : *Vector*, *Stack*, *ArrayList*, *LinkedList*, *HashSet*, *TreeSet*,
- **La classe java.util.Vector**

```
Vector vec = new Vector() ;

for (int i=0 ; i<10 ; i++) {
    Integer element = new Integer(i) ;
    vec.addElement(element) ;              // Ajout en fin de Vecteur
}
// => 0 1 2 3 4 5 6 7 8 9
…
Integer i = new Integer(15) ;
vec.insertElementAt(i,5) ;                 // Insertion à la position indiquée
// => 0 1 2 3 4 15 5 6 7 8 9
…
vec.removeElementAt(0) ;                   // Suppression de l'élément indiqué
// => 1 2 3 4 15 5 6 7 8 9
…
Integer j = (Integer)vec.elementAt(6) ;
// j contient une référence sur l'objet Integer contenant 5
…
vec.removeAllElements() ;                  // Suppression de tous les éléments
// =>
```

Interface Set

- C'est une interface identique à celle de Collection.
- **Set** est ensemble ne contenant que des valeurs et ces valeurs ne sont pas dupliquées. Une liste d'éléments uniques
  - Par exemple l'ensemble A = {1,2,4,8}. Set hérite donc de Collection, mais n'autorise pas
- La duplication. Interface **SortedSet** est un Set trié.
- Les deux classes permettant l'implémentation de cette interface sont :
  - **TreeSet**: les éléments sont rangés de manière ascendante.
  - **HashSet**: les éléments sont rangés suivant une méthode de hachage.

EXEMPLE SET (VOIR EXEMPLE ENSEMBLE NETBEANS)

```
import java.util.*;
public class SetExample {
public static void main(String args[]) {
    Set set = new HashSet(); // Une table de
    Hachage
    set.add("Bernadine");
    set.add("Elizabeth");
    set.add("Gene");
    set.add("Elizabeth");
    set.add("Clara");
    //parcours d'une set element par element
    Iterator it = set.iterator();
        while(it.hasNext())
          System.out.println(it.next());
    //retour l'objet set
    System.out.println(set);
    Set sortedSet = new TreeSet(set); // Un Set
    trié
    System.out.println(sortedSet);
    }
}
[Bernadine, Elizabeth, Gene, Clara]
[Bernadine, Clara, Elizabeth, Gene]
```

Interface List

- **L'interface List** hérite aussi de collection, mais autorise la duplication.
  - Servent à stocker des objets sans condition particulière sur la façon de les stocker
- Dans cette interface, un système d'indexation a été introduit pour permettre l'accès (rapide) aux éléments de la liste.
  - Ils acceptent toutes les valeurs, même les valeurs NULL
- List est une collection ordonnée.
  - Elle permet la duplication des éléments.
  - L'interface est renforcée par des méthodes permettant d'ajouter ou de retirer des éléments se trouvant à une position donnée.
  - Elle permet aussi de travailler sur des sous listes.
    - On utilise le plus souvent des **ArrayList** sauf s'il y a insertion d'élément(s) au milieu de la liste.
    - Dans ce cas il est préférable d'utiliser une **LinkedList** pour éviter ainsi les décalages.

Interface List

```
public interface List extends Collection {
    // Positional Access
    Object get(int index);
    Object set(int index, Object element); //
    Optional
    void add(int index, Object element); //
    Optional
    Object remove(int index); // Optional
    boolean addAll(int index, Collection c);
    // Optional
    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);
    // Iteration
    ListIterator listIterator();
    ListIterator listIterator(int index);
    // Range-view
    List subList(int fromIndex, int toIndex);
}
```

- Pour parcourir une liste, il a été défini un itérateur spécialement pour la liste.

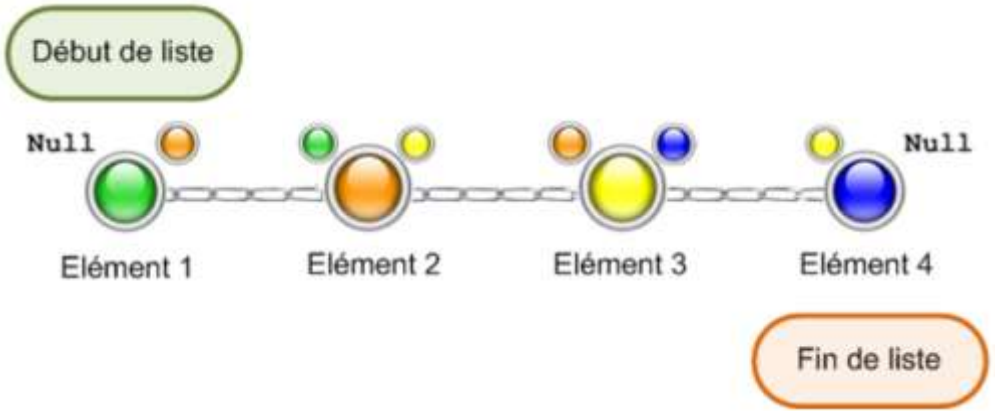
```
public interface ListIterator extends Iterator {
    boolean hasNext();
    Object next();
    boolean hasPrevious();
    Object previous();
    int nextIndex();
    int previousIndex();
    void remove(); // Optional
    void set(Object o); // Optional
    void add(Object o); // Optional }
```

- permet donc de parcourir la liste dans les deux directions et de modifier un élément (set) ou d'ajouter un nouveau élément.

Exemple :

```
List list = new ...;
ListIterator iterator = list.listIterator(list.size());
while (iterator.hasPrevious()) {
        Object element = iterator.previous();
        // traitement d'un élément
}
```

- **Classe LinkedList :**
  - Une liste chaînée (LinkedList en anglais) est une liste dont chaque élément est lié aux éléments adjacents par une référence à ces derniers.
  - Chaque élément contient une référence à l'élément précédent et à l'élément suivant, exceptés le premier, dont l'élément précédent vaut Null et le dernier, dont l'élément suivant vaut également Null



- **classe ArrayList**
  - est un de ces objets qui n'ont pas de taille limite et qui, en plus, acceptent n'importe quel type de données, y compris NULL
  - Mettre tout ce que nous voulons dans un ArrayList

- **Comparaison ArrayList LinkedList**
- ArrayList sont rapides en lecture, même avec un gros volume d'objets.
- Elles sont cependant plus lentes si vous devez ajouter ou supprimer des données en milieu de liste.

- Pour résumer
  - si vous effectuez beaucoup de lectures sans vous soucier de l'ordre des éléments, optez pour une ArrayList
  - si vous insérez beaucoup de données au milieu de la liste, optez pour une LinkedList

Exemple List

```
import java.util.*;
public class ListExample {
public static void main(String args[]) {
    List list = new ArrayList();
    list.add("Bernadine");   list.add("Elizabeth");
     list.add("Gene");                  list.add("Elizabeth");
    list.add("Clara");
    System.out.println(list);
    System.out.println("2: " + list.get(2));
    System.out.println("0: " + list.get(0));
    LinkedList queue = new LinkedList();
    queue.addFirst("Bernadine");
     queue.addFirst("Elizabeth"); queue.addFirst("Gene");
    queue.addFirst("Elizabeth"); queue.addFirst("Clara");
    System.out.println(queue); queue.removeLast();
    queue.removeLast();
    System.out.println(queue);}
}
Bernadine, Elizabeth, Gene, Elizabeth, Clara]
2: Gene
0: Bernadine
[Clara, Elizabeth, Gene, Elizabeth, Bernadine]
[Clara, Elizabeth, Gene]
```

EXEMPLE CLASS COLLECTION1

```
Collection <String> col =  new ArrayList<String>() ;
// ajout des elements a cette collection
            col.add("un") ; col.add("deux") ; col.add("trois") ;
// test d'appartenance de "deux"
            boolean b1 = col.contains("deux") ;
            System.out.println(b1) ;  // affiche true
 // test d'appartenance de "DEUX"
            boolean b2 = col.contains("DEUX") ;
            System.out.println(b2) ;  // affiche false

// parcourir les elements de la collection avec  un iterateur
            Iterator<String> it = col.iterator() ;
            while (it.hasNext()) {
                        String element = it.next() ;
                              // retourne un objet de type String
                        System.out.println(element) ; }

// balayer les elements de la collection avec un for each
            for (String element : col) {
                        System.out.println(element) ; }

// balayer les éléments de la collection avec forEach
//forEach (consumer<? super String>)void
            col.forEach(System.out::println);
            // Ici, la syntaxe "::<méthode>" permet de définir
              //  une référence sur méthode
}}
```

true
false

un
deux
trois

un
deux
trois

un
deux
trois

Exemple ListeChainée

```
LinkedList <String> lin = new LinkedList();
   System.out.print("liste en A:  ");
                  afficher(lin);
   lin.add("a");lin.add("b");// ajout en fin de liste
   System.out.print("liste en B:  ");
                 afficher(lin);
   ListIterator <String> it = lin.listIterator();
   it.next(); // on se place sur le premier élément
   it.add("c"); it.add("b"); // on ajoute deux éléments
   System.out.print("liste en C:  ");
                 afficher(lin);
   it = lin.listIterator();
   it.next(); // on progresse d'un élément
   it.add("b"); it.add("d"); // on ajoute deux éléments
   System.out.print("liste en D:  ");
                afficher(lin);
   it = lin.listIterator(lin.size()); // on se place en fin de liste
   while (it.hasPrevious()) { // on recherche le dernier b
    String ch = it.previous();
    if (ch.equals("b")) it.remove();// on le supprime
    break;    }
     System.out.print("liste en E:  ");
                              afficher(lin);
     it = lin.listIterator();
   it.next(); it.next(); // on se place sur le deuxième élément
   it.set("x"); // on le remplace par "x"
   System.out.print("liste en F:  ");
                   afficher(lin);
     }
}
```

```
public static void afficher(LinkedList
li){
      ListIterator <String> iter =
li.listIterator();
      while(iter.hasNext())
System.out.print(iter.next()+"  ");

 // avec forEach
      li.forEach(System.out::print);
        System.out.println();
}
```

- liste en A:
- liste en B: a b ab
- liste en C: a c b b acbb
- liste en D: a b d c b b abdcbb
- liste en E: a b d c b abdcb
- liste en F: a x d c b axdcb

Interface Map

- **Map** est un groupe de paires contenant une clé et une valeur associée à cette clé.
- Elles sont particulières, car elles fonctionnent avec un système clé - valeur pour ranger et retrouver les objets qu'elles contiennent.
  - C'est un ensemble de paires, contenant une clé et une valeur (en réalité, nous pouvons associer plusieurs valeurs. Dans ce cas la, nous sommes en présence d'une multimap …).
  - Deux clés ne peuvent être égales au sens de equals.

- **L'interface interne Entry** permet de manipuler les éléments d'une paire comme suit:

```
public interface Entry {
Object getKey();
Object getValue();
Object setValue(Object value);
}
```

INTERFACE MAP

```
public interface Map {
        // Basic Operations
        Object put(Object key, Object value);
        Object get(Object key);
        Object remove(Object key);
        boolean containsKey(Object key);
        boolean containsValue(Object value);
        int size();
        boolean isEmpty();
        // Bulk Operations
        void putAll(Map t);
        void clear();
        // Collection Views
        public Set keySet();
        public Collection values();
        public Set entrySet();
        // Interface for entrySet elements
        public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
```

}} values retourne les valeurs sous la forme d'une Collection.

EXEMPLE MAP (VOIR EXEMPLE TABLEAUASSOCIATIF)

```
import java.util.HashMap;  import java.util.Map;  import java.util.Set;
public class TablesAssociative {
public static void main(String[] args) {
        Map <String,String> m = new HashMap();
        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 intial: "+m);

        // retrouver la valeur associée à la clé "f"
        String ch = m.get("f");
        System.out.println("valeur associée à f est: "+ch);

        // ensemble des valeurs (Collection et non Set)
        Collection <String> valeurs = m.values();
        System.out.println("liste des valeurs initiales: "+ valeurs);
        valeurs.remove("30"); // on supprime la valeur 30 par la vue associée
        System.out.println("liste des valeurs après suppression: "+ valeurs);

        // ensemble des clés (ici Set)
        Set <String> cles = m.keySet();
        System.out.println("liste des clés initiales: "+cles);
        cles.remove("p"); // on supprime la clé p par la vue associée
        System.out.println("liste des clés après suppression: "+cles);
        System.out.println("Map après les deux suppressions: "+m);

        // modification de la valeur associée à la clé "x"
        String old = m.put("x","25");
        if (old!=null) System.out.println("valeur ancienne pour la clé x = " + old);
        System.out.println("Map après modification: "+m);
        System.out.println("liste des valeurs après modification: "+ valeurs);
    }
}
```

Map intial: {p=50, c=10, f=20, g=60, x=40, k=30}

valeur associée à f est: 20

liste des valeurs initiales: [50, 10, 20, 60, 40, 30]

liste des valeurs après suppression: [50, 10, 20, 60, 40]

liste des clés initiales: [p, c, f, g, x]

liste des clés après suppression: [c, f, g, x]

Map après les deux suppressions: {c=10, f=20, g=60, x=40

Map après modification: {c=10, f=20, g=60, x=25}

liste des valeurs après modification: [10, 20, 60, 25]

Interface- Classe Implémentant les interfaces

Interface	Table de hachage	Tableau de taille variable	Arbre balancé	Liste chainée
Set	hashSet		TreeSet	
List		ArrayList		linkedList
Map	hashMap		TreeMap	

Algorithmes manipulant les collections

- **Trier:**
  - sort(List list) ; trie une liste.
  - sort(List list,Comparator comp) ; trie une liste en utilisant un comparateur.
- **Mélanger:**
  - shuffle(List liste) ; mélange les éléments de manière aléatoire.
- **Manipuler**:
  - reverse(List liste) ; inverse les éléments de la liste.
  - fill (List liste, Object element) ; initialise les éléments de la liste avec element.
  - copy(List dest, List src) ; copy une liste src dans une liste dest.
- **Rechercher:**
  - binarySearch(List list, Object element) ; une recherche binaire d'un élément.
  - binarySearch(List list, Object element, Comparator comp) ; une recherche binaire d'un élément en utilisant un comparateur.
  - Des algorithmes qui s'appliquent sur toutes les collections: effectuer des recherches extrêmes:
  - min, max … min (Collection) max (Collection)

Interface **Queue**

- A partir de Java5 : La classe LinkedList a intégrée les nouvelles méthodes de Queue
- L'interface Queue dérivée de Collection destinée à la gestion des files d'attente (ou queues).
- Il s'agit des structures qui permettent d'introduire un nouvel élément si la queue n'est pas pleine et prélever le premier élément de la queue :
  - Méthode **offer** au lieu d'add pour introduire nouvel élément. Elle ne renvoie pas une exception comme add si la queue est pleine mais plutôt « false »
  - Méthode **pull** pour prélever un élément d'une façon destructive avec renvoi de null si la queue est vide,
  - et méthode **peek** d'une façon non destructive.

Classes implémentant Queue : PriorityQueue

- La classe PriorityQueue, introduite par Java 5, permet de choisir une relation d'ordre.
- **Le type d'éléments doit implémenter l'interface Comparable ou être doté d'un comparateur approprié.**

- Les éléments de la queue sont alors ordonnés par cette relation d'ordre et le prélèvement d'un élément porte sur le « premier » élément au sens de cette relation (on parle du « plus prioritaire »).

Les interfaces de comparaison : Comparator et Comparable

- Java propose l'interface **Comparable** qui doit être implémentée par une classe si nous voulons utiliser les méthodes de tri d'Arrays ou Collections.
- C'est un algorithme extrêmement rapide et stable
  - Il est utilisé pour trier la liste en utilisant l'ordre naturel du type

    interface Comparable
    {int compareTo(Object obj)}

  - L'interface Comparable possède une seule méthode **int compareTo(T obj)**
    - Elle retourne : un entier négatif si l'objet qui fait l'appel est plus petit que obj
    - nul (égal) si ils sont identiques,
    - ou a positif (supérieur) si il est plus grand

- Dans le cas d'une classe qui n'implante pas la classe Comparable, ou bien si il faut spécifier un autre ordre alors il faut implementer :
- L'interface **Comparator** propose une méthode int ***compare(Object o1, Object o2)***
  - Celle-ci retourne un entier négatif, nul ou positif si le premier objet est respectivement inférieur, égal ou supérieur au deuxième
  - Elle permet de comparer deux élément de la collection
  - Pour trié il faut passer une instance de cette classe à la méthode sort()

```
public class Employee implements Comparable<Employee>{
  private int id;     private String name;    private int age;
    private long salary;
public Employee(int id, String name, int age, int salary) {
        this.id = id;        this.name = name;
        this.age = age;        this.salary = salary;  }
    public int getId() {
        return id;    }
    public String getName() {
        return name;    }
    public int getAge() {
        return age;    }
    public long getSalary() {
        return salary;    }
   @Override
    public String toString() {
        return "[id=" + this.id + ", name=" + this.name + ", age=" + this.age + ",
salary=" +   this.salary + "]";  }
  @Override
    public int compareTo(Employee emp) {
        return (this.id - emp.id);}

 // Différentes implémentations de Comparator dans la classe Employee.
// Comparator pour trier la liste ou le tableau des employés en fonction du salaire
     public static Comparator<Employee> SalaryComparator = new
Comparator<Employee>() {
        @Override
        public int compare(Employee e1, Employee e2) {
            return (int) (e1.getSalary() - e2.getSalary()); }};
  // Comparator pour trier la liste des employés ou le tableau en fonction de l'âge
    public static Comparator<Employee> AgeComparator = new
Comparator<Employee>() {
        @Override
        public int compare(Employee e1, Employee e2) {
          return e1.getAge() - e2.getAge(); }}
    // Comparator pour trier la liste des employés ou le tableau en fonction du nom
     public static Comparator<Employee> NameComparator = new
Comparator<Employee>() {
        @Override
        public int compare(Employee e1, Employee e2) {
            return e1.getName().compareTo(e2.getName()); }};
```

EXEMPLE COMPARATORCOMPARABLE

```
public class ComparatorComparable {


    public static void main(String[] args) {
      //sort primitives array like int array
       int[] intArr = {5,9,1,10};
       Arrays.sort(intArr);
       System.out.println(Arrays.toString(intArr));
       //sorting String array
       String[] strArr = {"A", "C", "B", "Z", "E"};
       Arrays.sort(strArr);
       System.out.println(Arrays.toString(strArr));
       //sorting list of objects of Wrapper classes
       List<String> strList = new ArrayList<String>();
       strList.add("A");
       strList.add("C");
       strList.add("B");
       strList.add("Z");
       strList.add("E");
       Collections.sort(strList);
       for(String str: strList) System.out.print(" "+str);
       //sorting object array
      Employee[] empArr = new Employee[4];
      empArr[0] = new Employee(10, "Med", 25, 10000);
      empArr[1] = new Employee(20, "Ali", 29, 20000);
      empArr[2] = new Employee(5, "Leila", 35, 5000);
      empArr[3] = new Employee(1, "Papa", 32, 50000);

      //trie  des employes array  en utilisant l'implementation
  //de l'interface Comparable
      Arrays.sort(empArr);
      System.out.println("Liste trié par défaut des employés:\n"+Arrays.toString(er

  //sort employees array en utilisant Comparator  en fonction du salaire
Arrays.sort(empArr, Employee.SalaryComparator);
System.out.println("Liste  des employés triée en fonction du
salaire:\n"+Arrays.toString(empArr));

//sort employees array en utilisantComparator en fonction de l'age
Arrays.sort(empArr, Employee.AgeComparator);
System.out.println("Liste  des employés triée en fonction de l'
Age:\n"+Arrays.toString(empArr));

//sort employees array en utilisantComparator en fonction du nom
Arrays.sort(empArr, Employee.NameComparator);
System.out.println("Liste  des employés triée en fonction du
Nom:\n"+Arrays.toString(empArr));
    }

}
```

Interface Deque

- Java 6 a introduit l'interface Deque, dérivée de Queue, destinée à gérer des files d'attente à **double entrées**.
- On peut réaliser l'une des opérations suivantes à l'une des deux extrémités : **ajouter un élément, examiner un élément et supprimer un élément**
  - Pour ces 6 possibilités (3 actions, 2 extrémités) il existe deux méthodes :
    - une déclenchant une exception si l'opération échoue (queue pleine ou vide),
    - l'autre renvoyant une valeur spéciale (false pour l'ajout, null pour la suppression ou l'examen)
- Les méthodes de l'interface Queue restent utilisables sachant que celles d'ajout agissent sur la queue, tandis que celles de suppression ou d'examen agissent sur la tête.

	Exception	Valeur spéciale
Ajout	addFirst (e) addLast (e)	offerFirst () offerLast ()
Examen	getFirst () getLast()	peekFirst () peekLast ()
Suppression	removeFirst () removeLast ()	pollFirst () pollLast ()

Classe implémentant Deque : ArrayDeque

- A partir de Java 6
- La classe ArrayDeque implémente **une queue à double entrée sous forme d'un tableau**
  - (élements contigus en mémoire) redimensionable à volonté (comme ArrayList).
- Hormis les constructeurs, les méthodes spécifique de cette implémentation sont :
  - descendingIterator,
  - removeFirstOccurrence
  - et removeLastOccurrence.

Les classes Wrapper

- Les classes Wrapper sont des classes qui permettent de représenter les types de base sous forme d'objets :
  - **Boolean** pour **boolean**
  - **Integer** pour **int**
  - **Float** pour **float**
  - **Double** pour **double**
  - **Long** pour **long**
  - **Character** pour **char**

- Voir exemples

Les énumérations

- **Les énumérations :** permettent de rassembler un ensemble fixe de constantes (static et final)
- Une énumération peut être définie à l'intérieur d'une classe ou dans une unité de compilation séparée (comme toute classe Java).
- Une énumération ne peut pas être définie dans une méthode.
- Il est possible d'ajouter des champs, des constructeurs et des méthodes dans une classe de type enum.

Manipulation des enumerations

```
public class TestEcolor {
    public enum Ecolor  {RED, BLUE, GREEN, YELLOW};
    public static void main(String[] args) {

            Ecolor color = Ecolor.BLUE;
            System.out.println(color);

/*Pour tous les types énumérés, deux méthodes statiques values() et valueOf() sont implicitement déclarées*/
    //values() retourne un tableau des constantes d'énumération
            for (Ecolor col : Ecolor.values())
            System.out.print(col + " ");
/* valueof()convertit une chaîne de caractères en un élément du type énuméré ou lève l'exception
    IllegalArgumentException si impossible*/
            Ecolor yColor = Ecolor.valueOf("YELLOW");
            System.out.println(yColor);
/* name() retourne un String correspondant à la représentation textuelle d'une variable de type énuméré (semblable
    à toString()*/
            String colName = yColor.name();
/*ordinal() retourne un int correspondant à la position de l'élément dans la  déclaration d'une variable de type  énuméré
(la numérotation commence à zéro)*/
            int pos = yColor.ordinal(); // 3 for YELLOW
            System.out.println(pos);
    }
}
```