HELHa
Haute École Louvain en Hainaut

Syllabus

Programmation Orientée Objet
Le Framework Collections

Bachelier informatique de gestion Mons

BLOC 2 - UE IG205

V. Altares, A. Colmant

Année académique 2019 - 2020

ÉCONOMIQUE

1. Le framework

Le framework collections est une architecture logicielle qui permet de manipuler des collections.

Les collections, aussi appelées « containers » sont utilisées pour stocker, récupérer et manipuler des ensembles de données.

A la différence des tableaux, les collections permettent de stocker un nombre *variable* d'objets de types homogènes ou hétérogènes.

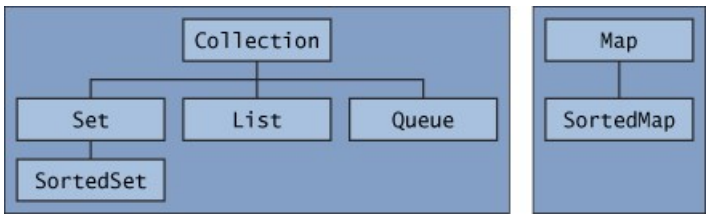
La taille d'une collection va donc s'adapter en fonction du nombre d'objets présents.

Le framework collections contient:

- des interfaces: types abstraits qui permettent de manipuler des collections indépendamment des détails de leur représentation,
- des implémentations, ce sont des classes qui implémentent des interfaces du framework,
- des algorithmes: des méthodes qui réalisent des opérations utiles telles que la recherche ou le tri.

Nous nous intéresserons ici à trois types de collections qui se trouvent dans le package *java.util:*

- **Les ensembles** qui contiennent des éléments *non dupliqués* dont l'accès reste très performant.
- **Les listes,** capables de contenir des objets de différents types accessibles *séquentiellement*.
- **Les maps,** tableaux associatifs qui permettent *d'associer* un objet clé à un autre objet valeur.



General-purpose Implementations

Interfaces	Implementations				
	Hash table	**Resizable array**	**Tree**	**Linked list**	**Hash table + Linked list**
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Le tableau ci-dessous est plus détaillé:

Interface	Classe	Description
List	Vector	tableau dynamique de taille variable
	Stack	mécanisme de pile de type LIFO
	ArrayList	tableau dynamique de taille variable
	LinkedList	liste chaînée utilisable comme une pile LIFO ou une file FIFO
Map	Hashtable	tableau associatif dont les clés ne peuvent être nulles
	HashMap	tableau associatif dont une clé et des valeurs peuvent être nulles
	WeakHashMap	tableau associatif où si une clé n'est plus référencée, la paire clé-valeur est ignorée
	TreeMap	tableau associatif dont les clés sont ordonnées par ordre croissant
Set	HashSet	ensemble associé à un map
	TreeSet	ensemble associé à un *TreeMap* et dont les objets sont en ordre croissant

Le diagramme suivant donne un aperçu général du framework:

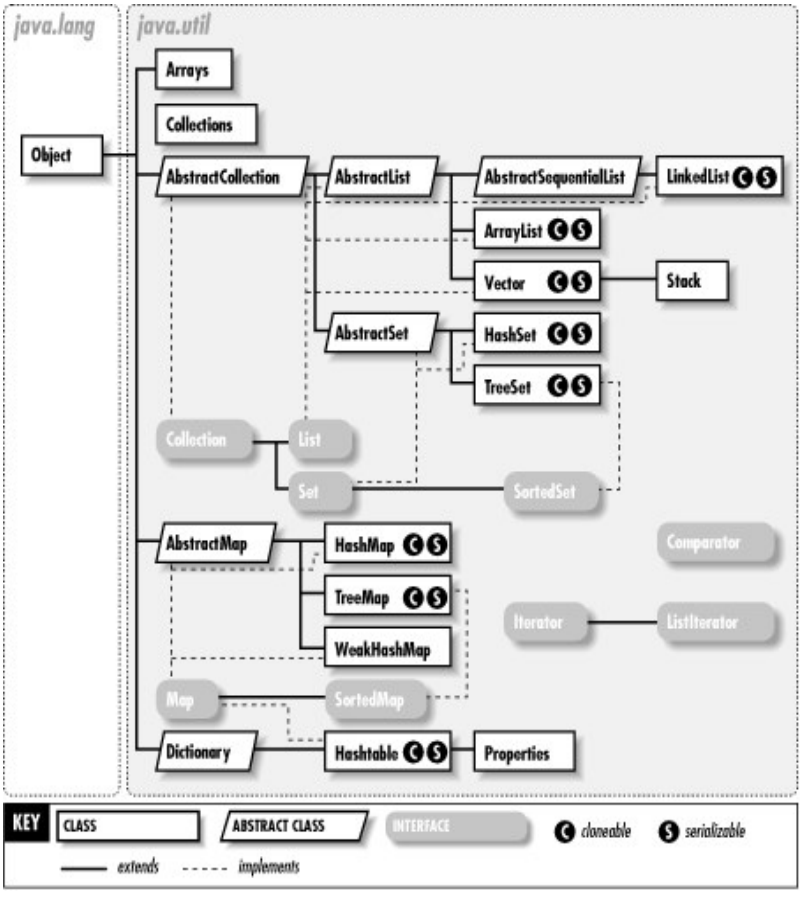
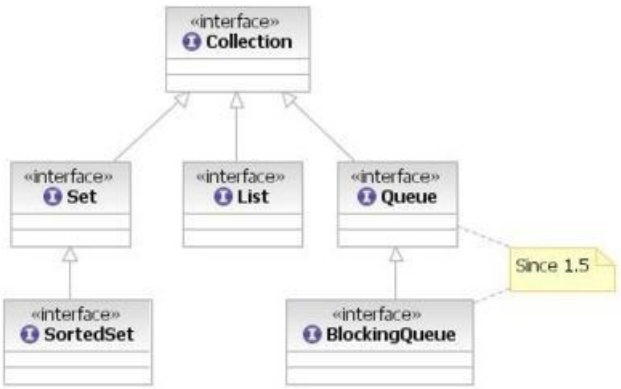


Schéma tiré du livre « Java Cookbook », O'Reilly.

2. L'interface Collection

L'interface *Collection* est à la base de la hiérarchie des collections Java.

L'interface *Collection* possède trois sous-interfaces importantes *List*, *Set* et *Queue*

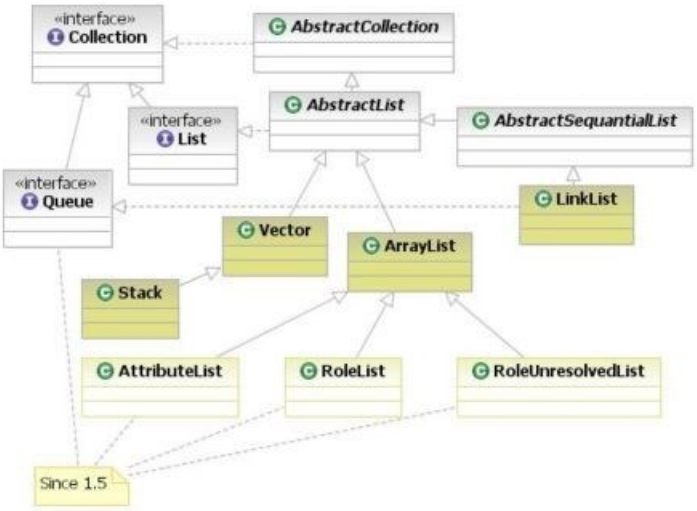


Les collections peuvent accepter des doublons comme dans le cas des listes, ou parfois les interdire comme pour les SortedSet.

Method Summary	
boolean	**add**(E e) Ensures that this collection contains the specified element (optional operation).
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this collection (optional operation).
void	**clear**() Removes all of the elements from this collection (optional operation).
boolean	**contains**(Object o) Returns true if this collection contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns true if this collection contains all of the elements in the specified collection.
boolean	**equals**(Object o) Compares the specified object with this collection for equality.
int	**hashCode**() Returns the hash code value for this collection.
boolean	**isEmpty**() Returns true if this collection contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this collection.
boolean	**remove**(Object o)
	Removes a single instance of the specified element from this collection, if it is present (optional operation).

boolean	**removeAll**(Collection<?> c) Removes all of this collection's elements that are also contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this collection.
Object[]	**toArray**() Returns an array containing all of the elements in this collection.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

Implémentations de l'interface Collection:



Les implémentations principales de l'interface *Collection* sont *AbstractCollection*, *AbstractList*, *AbstractSet*, *ArrayList*, *HashSet*, *LinkedHashSet*, *LinkedList*, *TreeSet* et *Vector*.

3. L'interface *List*

L'interface *List* représente une collection ordonnée, également connue sous le terme de séquence.

L'utilisateur de cette interface a un contrôle précis sur la position sur laquelle chaque élément sera inséré.

A l'image des tableaux, il est possible d'accéder aux éléments d'une liste par l'intermédiaire d'un indice. La recherche s'effectue de la même façon.

Les listes peuvent contenir des éléments doublons, ainsi que des valeurs *null*.

Les implémentations de l'interface *List* peuvent ajouter des restrictions supplémentaires sur les éléments qu'ils peuvent contenir, telles qu'interdire les éléments *null* ou de certains types.

Dans le cas ou des tentatives d'ajouter des éléments inappropriés seraient pratiquées, des exceptions non contrôlées pourraient être levées.

Les opérations sur un élément inapproprié, comme une vérification de présence, peuvent provoquer un comportement différent selon les implémentations comme la levée d'une exception ou un échec normal de l'opération (retour d'un booléen *false*).

Method Summary	
boolean	**add**(E e) Appends the specified element to the end of this list (optional operation).
void	**add**(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
boolean	**addAll**(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
boolean	**addAll**(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
void	**clear**() Removes all of the elements from this list (optional operation).
boolean	**contains**(Object o) Returns true if this list contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns true if this list contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this list for equality.
E	**get**(int index) Returns the element at the specified position in this list.
int	**hashCode**() Returns the hash code value for this list.
int	**indexOf**(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**() Returns true if this list contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this list in proper sequence.
int	**lastIndexOf**(Object o)

	Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	**listIterator**() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	**listIterator**(int index) Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position in this list.
E	**remove**(int index) Removes the element at the specified position in this list (optional operation).
boolean	**remove**(Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
boolean	**removeAll**(Collection<?> c) Removes from this list all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this list that are contained in the specified collection (optional operation).
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element (optional operation).
int	**size**() Returns the number of elements in this list.
List<E>	**subList**(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
Object[]	**toArray**() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

Il convient de remarquer la définition de la méthode equals de l'interface List:

boolean **equals**(Object o)

> Compares the specified object with this list for equality. Returns `true` if and only if the specified object is also a list, both lists have the same size, and all corresponding pairs of elements in the two lists are *equal*. (Two elements `e1` and `e2` are *equal* if `(e1==null ? e2==null : e1.equals(e2))`.)
>
> In other words, two lists are defined to be equal if they contain the same elements in the same order.

This definition ensures that the equals method works properly across different implementations of the `List` interface.

Implémentations de l'interface List:

L'interface *List* est une sous interface de l'interface racine *Collection*. Elle est implémentée notamment par les classes *AbstractList*, *ArrayList*, *LinkedList*, et *Vector*.

4. La classe *ArrayList*

La classe *ArrayList* permet de mettre en œuvre un tableau dont la taille peut varier dynamiquement.

La classe *ArrayList* implémente l'interface *List* qui elle-même étend *Collection*. De plus, elle est une extension de la classe abstraite *AbstractList*.

Les éléments d'un objet *ArrayList* peuvent être de n'importe quel type référence.

La valeur *null* est également admise dans cette liste.

Method Summary	
boolean	**add**(E e) Ensures that this collection contains the specified element (optional operation).
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this collection (optional operation).
void	**clear**() Removes all of the elements from this collection (optional operation).
boolean	**contains**(Object o) Returns true if this collection contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns true if this collection contains all of the elements in the specified collection.
boolean	**equals**(Object o) Compares the specified object with this collection for equality.
int	**hashCode**() Returns the hash code value for this collection.
boolean	**isEmpty**() Returns true if this collection contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this collection.
boolean	**remove**(Object o) Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	**removeAll**(Collection<?> c) Removes all of this collection's elements that are also contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this collection.

Object[]	**toArray**()
	Returns an array containing all of the elements in this collection.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

La création et l'initialisation d'un objet *ArrayList* dépend de trois constructeurs.

- Le premier permet d'instancier un objet vide avec une capacité initiale égale à 10.
- Le deuxième accepte comme argument une valeur destinée à définir cette capacité initiale.
- Le dernier initialise la liste avec les éléments d'une collection qui lui aura été spécifiée.

```
// par défaut : capacité initiale = 10
ArrayList liste = new ArrayList();
// capacité initiale de 50
ArrayList liste = new ArrayList(50);
// initialisé avec une
collection ArrayList
autreListe = new ArrayList();
...
ArrayList liste = new ArrayList(autreListe);
```

L'ajout d'éléments dans une collection *ArrayList* peut s'effectuer de plusieurs manières :

- ajout d'un élément,
  - liste.add(objet);
  - liste.add(23, objet);*//insertion à l'indice spécifié*

- ajout de tous les éléments d'une collection qui existe déjà :
  - liste.addAll(collection);
  - liste.addAll(23, collection)*//insertion à l'indice spécifié*

L'accès aux éléments d'une collection *ArrayList* s'effectue par l'intermédiaire de leur indice. Néanmoins, dans le cas d'ArrayList non génériques, l'objet retourné sera du type *Object*, la plupart du temps, il faudra le convertir.

```
String s = (String) liste.get(indice);
```

La taille de la liste est accessible à partir de la méthode *size()*.

```
int taille = liste.size();
```

Les méthodes *contains()* et *containsAll()* fournissent un moyen de vérification de présence d'un ou plusieurs éléments au sein d'une collection *ArrayList*.

```
// vérifie si la liste contient l'élément spécifié
boolean resultat = liste.contains(objet); //
vérifie si la liste contient tous //les éléments
de la collection spécifiée boolean resultat =
liste.containsAll(collection);
```

Les méthodes *indexOf()* et *lastIndexOf()* permettent de rechercher l'élément fourni, respectivement à partir du début et de la fin de la liste.

```
// recherche la première occurrence de l'objet spécifié
int indiceFirts = liste.indexOf(obj);
// recherche la dernière occurence de l'objet spécifié
int indiceLast = liste.lastIndexOf(obj);
```

Les méthodes *subList()* et *toArray()* sont capables d'extraire une partie ou la totalité des éléments d'une liste, dans respectivement un objet *List* ou un tableau.

```
// retourne une liste contenant les
éléments au sein de l'intervalle spécifié
List sousListe = liste.subList(1, 5);
// retourne tous les éléments de la liste dans un tableau
Object[] tableau = liste.toArray();
// retourne tous les éléments de la liste dans un tableau
//et affecte le type d'exécution du tableau spécifié au tableau retourné
String[] tabType;
Object[] tableau = liste.toArray(tabType);
```

La suppression d'un ou plusieurs éléments d'une liste s'accomplit grâce à plusieurs méthodes: *remove()*, *removeRange()*, *removeAll()* et *clear()*.

```
// supprime l'objet obj
liste.remove(obj);
// supprime l'objet à l'indice spécifié
liste.remove(2);
// supprime les objets compris dans l'intervalle spécifié ([,[)
liste.removeRange(1, 5);
// supprime tous les éléments
faisant // partie de la collection
spécifiée liste.removeAll(colection);
// supprime tous les éléments de la liste
liste.clear();
```

La méthode *set()* remplace un élément de la liste par un objet spécifié et retourne l'élément remplacé.

```
Object eltRemplace = liste.set(2, nouvelObjet);
```

Le parcours des éléments d'un objet *ArrayList* peut se faire en utilisant une boucle, ou en utilisant un itérateur sur les éléments de la liste.

```
for(int i = 0; i < liste.size(); i++){
   Object element = liste.get(i);
}
for(Object o :liste)
{    Object element =
o; }

// itérateur
Iterator iterateur = liste.iterator(); while(iterateur.hasNext())
{
   Object element = iterateur.next();
}
// itérateur de liste
```

```
ListIterator iterateur = liste.listIterator();
while(iterateur.hasNext()){
   Object element = iterateur.next();
}
```

L'objet *ListIterator* possède plusieurs méthodes supplémentaires très utiles dans la gestion des listes. Il est, par exemple, possible de parcourir une liste dans les deux directions.

```
// parcours du bas vers le haut
ListIterator iterateur = liste.listIterator();
while(iterateur.hasPrevious()){
   Object element = iterateur.previous();
}
// parcours du haut vers le bas
ListIterator iterateur = liste.listIterator();
while(iterateur.hasNext()){
   Object element = iterateur.next();
}
```

D'autre part, des méthodes permettent de retourner l'indice suivant (*nextIndex()*) ou précédent (*previousIndex()*), de supprimer (*remove()*), d'ajouter (*add()*) et de remplacer (*set()*) des éléments.

5. ArrayList génériques

JDK 1.5 permet de déclarer des classes, des interfaces et des méthodes génériques.

Plusieurs interfaces et classes dans l'API de Java sont modifiées en utilisant des types génériques.

Par exemple, la déclaration de l'interface java.lang.Comparable a évolué dans JDK 1.5.

L'interface comparable est redéfinie dans JDK 1.5 avec un type générique.

```
package java.lang;

public interface Comparable {
  public int compareTo(Object o)
}
```

```
package java.lang;

public interface Comparable<T> {
  public int compareTo(T o)
}
```

Ici, <T> représente *un type générique formel*, qui pourra être substitué par *un type concret réel* ultérieurement.

Cette substitution s'appelle *instantiation générique*.

Par convention, une majuscule simple telle que E ou T est utilisée pour dénoter un type générique formel.

Avant JDK 1.5, la déclaration illustrée dans le tableau ci-dessous (colonne de gauche) spécifie que c'est une référence de type Comparable et invoque la méthode compareTo pour comparer un objet Date avec un String. Le code compile parfaitement, mais une erreur d'exécution se produit parce qu'un String ne peut pas être comparé avec une date.

Le nouveau type générique découvre des erreurs possibles à la compilation.

```
Comparable c = new Date();
System.out.println(c.compareTo("red"));
```

```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

Dans la colonne de droite, la déclaration spécifie que c'est une référence de type Comparable <Date> dans JDK 1.5 et invoque la méthode compareTo pour comparer un objet Date avec un String. Une erreur compilation se produit, parce que l'argument passé à la méthode compareTo doit être du type Date. Comme les erreurs peuvent être découvertes à compilation plutôt qu'à l'exécution, le type générique rend le programme plus fiable.

Les types génériques doivent être des types de référence. Il n'est pas possible de substituer un type générique avec un type primitif comme int, double, char...

Le tableau ci-dessous montre les diagrammes de classe pour ArrayList avant JDK 1.5 et dans JDK 1.5, respectivement.

ArrayList est une classe générique dans JDK 1.5

java.util.ArrayList
+ArrayList() +add(o: Object): void +add(index: int, o: Object): void +clear(): void +contains(o: Object):boolean +get(index:int): Object +indexOf(o: Object): int +isEmpty(): boolean +lastIndexOf(o: Object): int +remove(o: Object): boolean +size(): int +remove(index: int): boolean +set(index: int, o: Object): Object

java.util.ArrayList<E>
+ArrayList() +add(o: E): void +add(index: int, o: E): void +clear(): void +contains(o: Object): boolean +get(index:int): E +indexOf(o: Object): int +isEmpty(): boolean +lastIndexOf(o: Object): int +remove(o: Object): boolean +size(): int +remove(index: int): boolean +set(index: int, o: E): E

Par exemple, la déclaration suivante crée une liste pour les String :

JAVA 1.5 et 1.6	JAVA1.8
ArrayList <String> liste = new ArrayList <String> ();	ArrayList <String> liste = new ArrayList < > (); Java 1.8 permet d'utiliser le « diamond » < > sans spécifier le type au moment de l'instanciation, il s'agit du mécanisme de « type inference »

Dans cette liste on ne peut ajouter que des String.

Par exemple,

```
list.add("Rouge");
```

Une erreur de compilation se produira lors d'une tentative d'ajout d'autre chose qu'un String.

Par exemple, la déclaration suivante est illégale maintenant parce que la liste peut contenir seulement des String :

```
list.add (new Integer(1));
```

Le transtypage n'est pas nécessaire pour récupérer une valeur d'une liste avec un type d'élément indiqué parce que le compilateur connaît déjà le type d'élément.

6. Les classes utilitaires

Pour être pleinement opérationnelles et permettre d'exécuter certaines opérations comme des comparaisons ou encore des itérations, les collections Java nécessitent les services de plusieurs classes et interfaces utilitaires.

Les classes et interfaces utilitaires peuvent être *Collections* pour l'ensemble des collections, *Enumeration* pour les ensembles et les listes, *Map.Entry* pour les maps, ...

A. La classe *Collections*

La classe *Collections* est une classe utilitaire du framework collections de Java.

Cette classe contient des méthodes statiques qui manipulent ou retournent des collections.

Toutes les méthodes de la classe *Collections* sont susceptibles de lancer une exception *NullPointerException* si les collections passées en argument ont la valeur *null*.

Tri

Le tri des collections *List* peut être mis en œuvre par les méthodes *sort()*, selon l'ordre naturel de éléments de la collection ou par rapport à un comparateur spécifié.

```
List <Point> liste = new ArrayList<Point>();
liste.add(new Point(1,2));
// ...
liste.add(new Point(1,6));
Comparator<Point> comp=new ComparateurPoint();
//la classe ComparateurPoint implémente Comparator<Point>
Collections.sort(liste, comp);
```

La méthode *reverseOrder()* retourne un objet *Comparator* qui impose un tri inverse à celui de l'ordre naturel d'une collection.

```
Collections.sort(liste,Collections.reverseOrder());
```

L'inversion complète (sans tri) de l'ordre des éléments d'une liste peut être obtenu par la méthode *reverse()*.

```
Collections.reverse(liste);
```

Recherche

Plusieurs méthodes *binarySearch()* permettent de rechercher, à l'aide d'un algorithme binaire, un objet spécifié au sein d'une collection de type *List* également donnée.

Pour que le résultat retourné par la méthode corresponde au résultat attendu, la liste doit être triée dans l'ordre ascendant.
Si nécessaire, un comparateur peut être spécifié afin d'ordonner la liste conformément à ses propres règles. Par défaut, l'ordre naturel est employé pour ce tri.

Si la liste contient plusieurs éléments égaux, alors il n'y a aucune garantie sur lequel des éléments sera trouvé. La valeur de retour de ces méthodes est du type *int*. Si l'objet spécifié est trouvé, la méthode retournera la position de l'élément. Dans le cas contraire, la valeur -1 - pointInsertion sera renvoyée. Si les éléments de la liste ne sont pas mutuellement comparables, alors l'exception *ClassCastException* sera levée par les méthodes.

```
List <String> liste = new ArrayList<String>();
liste.add("Premier");
// ...
liste.add("Sixième"); Collections.sort(liste); int res =
Collections.binarySearch(liste, "Quatrième");

List <Point> listePoint = new ArrayList<Point>();
listePoint.add(new Point(1,2));
// ...
listePoint.add(new Point(1,6));
Comparator<Point> comp=new ComparateurPoint(); //la classe
ComparateurPoint implémente Comparator<Point> int res =
Collections.binarySearch(listePoint,new Point(1,4),comp);
```

Quelques méthodes utiles

Les méthodes *min()* et *max()* retournent respectivement l'élément minimum ou maximum d'une collection spécifiée, pas nécessairement triée, selon l'ordre naturel de ses éléments ou d'un comparateur fourni.

```
Point pMin = Collections.min(liste);

Point pMin = Collections.min(liste, comp);

Point pMax = Collections.max(liste);

Point pMax = Collections.max(liste, comp);
```

Les méthodes *shuffle()* permutent aléatoirement la liste spécifiée en utilisant un algorithme aléatoire par défaut ou précisé par un objet *Random*.

```
Collections.shuffle(liste);

Collections.shuffle(liste, objRandom);
```

B. Arrays

La classe Arrays est une classe utilitaire qui contient des méthodes qui permettent de manipuler les tableaux Java.

Elle contient des méthodes statiques.

Toutes les méthodes de cette classe sont susceptibles de lever une exception *NullPointerException*, si le tableau sur lequel doit s'appliquer une opération, a la valeur *null*.

```
int[] tableau = null;
Arrays.uneMethodeStatique(tableau, args);
// L'exception NullPointerException est levée par la méthode
```

Les méthodes permettent notamment d'effectuer des recherches et des tris dans un tableau.

Tri

Les méthodes *sort()* sont capables de trier les éléments du tableau spécifié, selon un ordre ascendant.

Le tri peut être réalisé de deux manières. La première façon applique un tri selon l'ordre naturel des éléments, la seconde par rapport à un objet *Comparator* spécifié.

Premier cas: tous les éléments du tableau doivent impérativement implémenter l'interface *Comparable* et doivent être mutuellement comparable, c'est-à-dire, qu'une exception *ClassCastException* ne doit pas être levée lors de l'appel de la méthode *compareTo()*.

```
int res = e1.compareTo(e2);

Arrays.sort(tabObjets);
Arrays.sort(tabObjets, fromIndex,toIndex);
```

Deuxième cas: un comparateur est passé en argument avec le tableau, afin d'indiquer à la méthode *sort()* le mode de tri à utiliser sur ce tableau. Afin d'viter qu'une exception *ClassCastException* soit levée lors d'une tentative de comparaison, il est nécessaire que tous les éléments du tableau soient mutuellement comparables.

```
Arrays.sort(tabObjets, comparateur);
Arrays.sort(tabObjets, fromIndex, toIndex, comparateur);
```

Recherche

Les méthodes *binarySearch()* sont utilisées pour effectuer une recherche d'un élément spécifié dans un tableau cible.

De même que pour la classe Collections, le tableau doit être trié par la méthode *sort()* adéquate. En cas de réussite de la recherche, l'indice de l'élément dans le tableau est retourné, dans le cas contraire, la valeur *-1- pointInsertion* est retournée.

Comme pour le tri, la recherche peut être réalisée de deux manières. La première applique un tri selon l'ordre naturel des éléments, la seconde selon un objet *Comparator* spécifié.

Premier cas: tous les éléments du tableau doivent impérativement implémenter l'interface *Comparable* et doivent être mutuellement comparables.

```
Arrays.binarySearch(tabObjets, unObjet);
Arrays.sort(tabObjets, fromIndex,toIndex, unObjet);
```

Deuxième cas: un comparateur est passé en argument avec le tableau, afin d'indiquer à la méthode *sort()* le mode de tri à mettre en œuvre sur ce tableau. Il est nécessaire que tous les éléments du tableau soient mutuellement comparables, sans quoi une exception *ClassCastException* risquerait d'être levée lors d'une tentative de comparaison.

```
Arrays.sort(tabObjets, comparateur);
Arrays.sort(tabObjets, fromIndex, toIndex, unObjet , comparateur);
```

Quelques méthodes utiles

Les méthodes *fill()* affectent un valeur donnée à tous les éléments du tableau ou à une plage des éléments du tableau.

```
double tableau = new double[10];
Arrays.fill(tableau, 0.0);

String[] tableau = new String[100];
Arrays.fill(tableau, "");
```

Les méthodes *equals()* permettent de tester l'égalité entre deux tableaux de même type. Si les tableaux sont égaux, la méthode *equals()* retourne *true*, sinon elle renvoie *false*. Une égalité effective correspond à un nombre d'éléments équivalent et à une égalité stricte entre les paires d'éléments (tab1[1] = tab2[1], tab1[2] = tab2[2], etc..). Pour garantir un même ordre aux deux tableaux à comparer, il peut être opportun de les trier préalablement.

```
double[] tab1 = {0.1, 0.8, 0.9, 2.5, 1.1, 3.0};

double[] tab2 = {0.1, 0.8, 0.9, 2.5, 1.1, 3.0};

double[] tab3 = {3.0, 1.1, 2.5, 0.9, 0.8, 0.1};

boolean res = Arrays.equals(tab1, tab2); // retour true
boolean res = Arrays.equals(tab2, tab3); // retour false

Arrays.sort(tab3);

boolean res = Arrays.equals(tab2, tab3); // retour true
```

La méthode *asList()* constitue un pont entre les tableaux et les collections Java. En effet, cette méthode retourne une liste dont les éléments seront ceux du tableau spécifié en argument.

```
List liste = Arrays.asList(tableau);
```

7. Références :

- *Tutoriel sun/oracle: http://docs.oracle.com/javase/tutorial/collections/index.html*
- *WikiBook sur les collections : http://en.wikibooks.org/wiki/Java_Programming/Collection_Classes*