

Chapitre 6:

Les classes de base

Object, Collections, String, Math, Graphics, etc...

Plan du chapitre

- La classe Object
- Les collections
 - La classe ArrayList
 - La classe Vector
- Les classes Wrappers
- Les classes String et StringBuffer
- La classe Math
- La classe Graphics

La classe Object

- La classe **Object** est la classe mère dont héritent toutes les autres classes
- Toute classe définie en Java, que ce soit par l'utilisateur ou dans un package de l'API, hérite de la classe **Object**
- Ainsi, toute classe hérite les méthodes de la classe **Object**
 - **public final Class getClass()** renvoie le type de l'objet
 - nom de la classe,
 - nom de ses ancêtres,
 - structure.
 - **void finalize()** utilisée par le Garbage Collector
 - **boolean equals(Object o)** compare le contenu des références (à redéfinir)
 - **Object clone()** : crée une copie de l'objet. (à redéfinir)
 - **String toString()** renvoie une String décrivant l'objet. (à redéfinir)

La classe Object

Utilisation des méthodes toString, equals et getClass

```
class Personne { // dérive de Object par défaut
    private String nom ;
    public Personne(String nom) { this.nom = nom ; }
    public String toString() { return "Classe: " + getClass().getName()
        + " Objet : " + nom;}
    boolean equals(Personne p) { return p.nom.equals(this.nom) ; }
}
```

Exemple d'utilisation

```
Personne p1 = new Personne("Ahmed") ;
Personne p2 = new Personne("Ahmed") ;
System.out.println(p1.toString()) ;
System.out.println(p2.toString()) ;
System.out.println(p1 == p2) ;
System.out.println(p1.equals(p2)) ;
```

Résultat d'affichage

```
Classe: Personne Objet : Ahmed
Classe: Personne Objet : Ahmed
false
true
```

La classe Object

- Certains traitements sont si génériques qu'ils peuvent s'appliquer à tous les objets
- Par exemple, une liste peut contenir n'importe quel type d'objet et les compter, les trier, etc.
- Donc il faut écrire la classe **Liste** pour manipuler des objets de la classe **Object**
- On pourra alors stocker dans la liste n'importe quel type d'objet
- Il existe en Java plusieurs classes qui de ce type, permettant de stocker, trier, retrouver... efficacement des objets : les **Collections**

Les collections

- Une collection est un objet regroupant un ensemble d'objets et permettant de :
 - stocker et retrouver des données.
 - transmettre des données dans un emballage unique.
- Exemple :
 - un dessin est un ensemble de formes.
 - un répertoire téléphonique est un ensemble de pairs : nom, numéro de téléphone.
- Java propose une architecture unifiée pour les collections :
 - un ensemble d'interfaces, un ensemble d'implémentation.
 - Des algorithmes d'accès, de tri, de recherche.
- Une interface définit une liste de service.

Les collections

```
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();  
    // 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[]);  
}
```

Les collections

```
boolean contains(Object o)
```

```
Returns true if this collection contains the specified element.  
More formally, returns true if and only if this collection  
contains at least one element e such that  
(o==null ? e==null : o.equals(e)).
```

Parameters:

```
o - element whose presence in this collection is to be tested.
```


Returns:

```
true if this collection contains the specified element
```

- La javadoc définit comment réaliser ces services
- L'implémentation de l'interface Collection pour **contains** doit utiliser la méthode **equals** comme critère.
- Les objets stockés doivent appartenir à une classe où la méthode **equals** est correctement défini.

Les collections

- Si la classe Cercle, n'a pas redéfini la méthode `equals`, c'est la méthode définie dans la classe `Object` qui sera appelée.



```
// Dans la classe Object.  
public boolean equals(Object o) {  
    return this == o;  
}
```

```
// une référence sur une Collection.  
c.clear();  
Cercle a = new Cercle (0,0,5);  
c.add(a);  
Cercle b = new Cercle (0,0,5);  
System.out.println(c.contains(a)); // true  
System.out.println(c.contains(b)); // false
```

Les collections

- Il existe 2 sortes de collection :
 - public interface `List` extends `Collection`
 - 2 éléments identiques peuvent coexister.
 - public interface `Set` extends `Collection`
 - pas d'éléments dupliqués
- Dans les deux cas les méthodes `add` et `remove` ont le même comportement. Elles sont contenues dans l'interface *Collection*
 - La méthode boolean `add(Object o)`
 - La collection contiendra l'objet o.
 - Retourne `true` si la collection a été modifiée.
 - La méthode boolean `remove(Object o)`
 - retire un objet e tel que `o.equals(e)`
 - Retourne `true` si la collection a été modifiée.

Iterator

- Permet de gérer les éléments d'une collection
- Toute collection peut fournir son « Iterator »
- Un itérateur permet le parcours séquentiel d'une collection sans autre précision sur l'ordre de parcours.
- Il existe une interface :

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove // optional.  
}
```

- Contient 3 méthodes essentielles:
 - hasNext(): *boolean* ➔ Indique s'il reste des éléments après l'élément en cours
 - next(): *Object* ➔ Fournit l'élément suivant de la collection
 - Remove(): *void* ➔ Supprime l'élément en cours

Les collections

- Deux principales implémentations de l'interface « Set »
 - HashSet (Set) : rapide mais n'offre aucune garantie en termes de l'ordre
 - TreeSet (SortedSet) : moins rapide mais contient une structure permettant d'ordonner les éléments
 - A n'utiliser que si la collection doit être triée ou doit pouvoir être parcourue dans un certain ordre
- Deux principales implémentations de l'interface « List »
 - **ArrayList et Vector** : particulièrement rapides et sont les plus utilisés
 - LinkedList : plus lent; à utiliser pour
 - Ajouter des éléments au début de la liste
 - Supprimer des éléments au milieu de la liste

La classe `ArrayList`

- un `ArrayList` se comporte comme un tableau
 - il contient plusieurs objets (de la classe `Object` uniquement)
 - ne peut pas contenir des types primitifs
 - il accède à ses éléments à l'aide d'un index
 - il grossit automatiquement
 - quand plus de place pour contenir de nouveaux objets
 - il existe des méthodes pour ajouter ou enlever un élément
 - il est possible d'indiquer la taille initiale dans le constructeur
 - Il y a 2 constructeurs :
 - `ArrayList()`
 - `ArrayList(int initialCapacity)`

La classe `ArrayList`

- Modification d'éléments
 - Il y a deux manières d'ajouter un élément
 - à la fin d'un `ArrayList` avec la méthode
`boolean add(Object newElement)`
 - à une position donnée
`void add(int index, Object newElement)`
 - pour remplacer un objet à une position donnée
`Object set(int index, Object newElement)`
- Accès aux éléments
 - il n'y a pas d'indexation comme pour les tableaux
 - il faut utiliser la méthode spécialisée `Object get(int index)`
 - exemple :

```
ArrayList uneListe = new ArrayList();
uneListe.add(new Point());
uneListe[0].display(); // interdit !
uneListe.get(0).display(); // ok
```

La classe `ArrayList`

- pour tester le contenu, il existe la méthode `boolean isEmpty()`
- pour connaître le nombre d'éléments dans la liste, il faut utiliser la méthode

```
int size()
```

– exemple :

```
if (!uneListe.isEmpty()) {  
    for (int i=0; i<uneListe.size(); i++){  
        System.out.println(uneListe.get(i));  
    }  
}
```

- pour recopier une liste dans un tableau, il faut utiliser la méthode

```
Object[] toArray()
```

– exemple :

```
ArrayList uneListe = new ArrayList();  
...  
Object[] tab = new Object[uneListe.size()];  
tab = uneListe.toArray();
```

La classe `ArrayList`

- Suppression d'éléments
 - pour supprimer un élément à une position donnée, il faut utiliser la méthode
`Object remove(int index)`
- Recherche d'éléments
 - pour savoir si un objet est présent ou non dans une liste, il faut utiliser la méthode
`boolean contains(Object obj)`
 - pour connaître la position d'un élément dans une liste, on peut utiliser deux méthodes
 - pour avoir la *première occurrence*, il faut utiliser
`int indexOf(Object obj)`
 - pour avoir la *dernière occurrence*, il faut utiliser
`int lastIndexOf(Object obj)`

La classe **Vector**

- Classe située aussi dans le package `java.util`
- **Vector** permet de réaliser des listes d'éléments quelconques pourvu qu'ils héritent de la classe `Object`.
- Les objets de la classe **Vector** représentent des tableaux à taille variable, c.à.d. il n'y a pas de limite au nombre d'objets qu'il peut contenir
- constructeurs :
 - `Vector()` crée un vecteur vide
 - `Vector(int nombre)` crée un vecteur vide de capacité précisé.
- attributs :
 - `elementCount` nombre d'éléments du vecteur

La classe Vector

- **isEmpty()** retourne true si le vecteur est vide
- **size()** retourne le nombre d'éléments du vecteur
- **addElement(Objet)** ajoute un élément à la fin du vecteur
- **insertElementAt(Objet, int position)** ajoute un élément à la position spécifiée
- **contains(Objet)** retourne true s'il l'Objet se trouve dans le vecteur
- **elementAt(int position)** retourne l'élément à la position spécifiée
- **indexOf(Objet)** retourne la position de la première occurrence de l'Objet dans le vecteur
- **removeElementAt(int position)** supprime l'élément à la position spécifiée

La classe Vector

```
import java.util.*;
```

```
Vector v = new Vector() ;  
v.addElement("1");  
v.addElement("2");  
v.addElement("3");  
System.out.println(v.toString());
```

Résultat : [1, 2, 3]

```
v.insertElementAt("abc",2); //insertion en 3eme position  
System.out.println(v.toString());
```

Résultat : [1, 2, abc, 3]

```
v.setElementAt("isimm",2); // changement d'un élément  
System.out.println(v.toString());
```

Résultat : [1, 2, isimm, 3]

La classe Vector

- Parcours d'un Vector :

```
for(int i=0 ; i < v.size(); i++)  
    System.out.println(v.elementAt(i));
```

- Autres méthodes utiles

```
int num ;  
num = v.indexOf("2") ;      // indice d'un élément  
v.removeElement("2") ;     // retrait d'un élément  
System.out.println(v.lastElement( )) ;
```

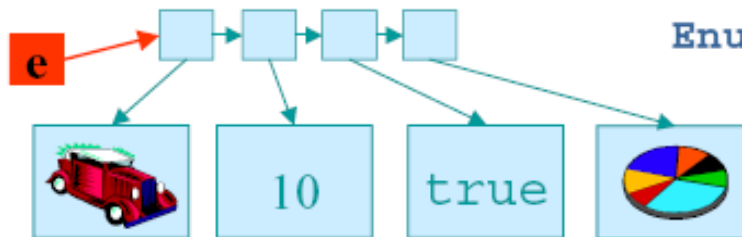
- Un élément retiré d'un Vector doit être converti vers le type désiré

```
import java.util.* ;  
class B{.....}
```

```
B b1;  
Vector v = new Vector( ) ;  
v.addElement(new B(..)); //ajout d'un objet de type B  
v.addElement(new B(..)); // ajout d'un objet de type B  
b1 = (B) v.elementAt(1); // conversion en B
```

L'interface Iterator

- Interface permettant de créer un **itérateur** pour accéder séquentiellement à chaque élément d'une collection
 - implémentée p.ex. par Vector
 - ne permet qu'une utilisation par instance d'Enumeration
- Utilisation avec Vector
 - itérateur obtenu par appel à la méthode `elements()`
 - `hasMoreElements()` indique s'il reste des éléments
 - `nextElement()` rend l'élément suivant



```
Enumeration e = v.elements();  
e.nextElement();
```



L'interface Iterator

- Afficher tous les éléments d'un vecteur

```
Vector v = new Vector();  
v.addElement("Un petit peu de texte");  
v.addElement(new Integer(10));  
Enumeration e = v.elements();  
while(e.hasMoreElements()) {  
    final Object o = e.nextElement();  
    System.out.println(o.toString());  
} // while
```

- Résultat:

```
Un petit peu de texte  
10
```

Les classes Wrappers

- Problème : les types de base (int, float, double, boolean... ne sont pas des objets)
- On ne peut pas les stocker tels qu'ils sont dans les collections
- C'est à cela que servent les classes enveloppes (Wrappers)
- Permettent de représenter des types de base sous forme d'objets
- Les classes Wrapper permettent en particulier de récupérer les valeurs **minimum** et **maximum** du type de base correspondant
- Les conversions entre types de base et chaînes de caractères sont possibles via des objets Wrappers.

Les classes Wrappers

- `int entier => Integer n`
`Integer n = new Integer(entier);`
`entier = n. intValue();`
- `double => Double, boolean => Boolean, char => Character, ...`
`intValue(), doubleValue() ...`
- Conversion d'une chaîne de caractère vers un entier :
 - `Integer.parseInt(String s)`
 - `Integer(String s).intValue()`
 - `Integer.valueOf(String s).intValue()`
- Conversion d'un entier en chaîne de caractère :
 - `Integer(int i).toString()`
 - `String.valueOf(int i)`
- `Integer.MIN_VALUE, Integer.MAX_VALUE, ...`

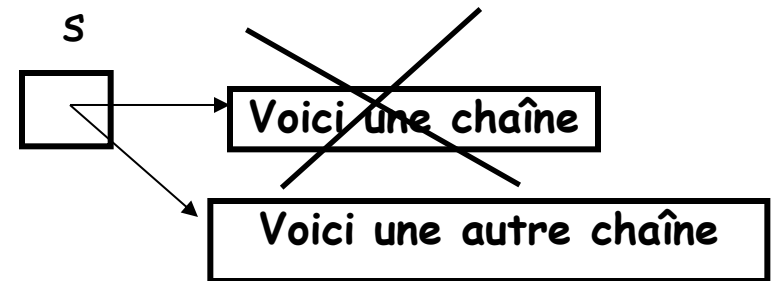
La classe String

- Les chaînes de caractères sont gérées à l'aide de la classe `String`
- Les méthodes :
 - `String ()`; crée une chaîne vide
 - `String (String)`; crée une chaîne à partir d'une autre chaîne
 - `char charAt (int n)`; fournit le n ième caractère de la chaîne
 - `int compareTo (String s)`; compare la chaîne de l'objet et la chaîne s
 - `String concat(String s)`; concatène la chaîne de l'objet et la chaîne s
 - `int length ()`; longueur de la chaîne
 - `String toLowerCase ()`; fournit une nouvelle chaîne convertie en minuscules
 - `String toUpperCase ()`; fournit une nouvelle chaîne convertie en majuscules
 - `int indexOf (int c)`; indice du caractère c dans la chaîne
 - `String replace (char c1, char c2)`; crée une nouvelle chaîne en remplaçant le caractère c1 par c2
 - etc.

La classe String

- La classe **String** décrit des objets qui contiennent une chaîne de caractères **constante**

```
String s ;  
s = "voici une chaîne";  
s = "voici une autre chaîne";
```



- C'est aussi la seule classe qui dispose d'opérateurs supplémentaires : **+** et **+=** pour la concaténation de chaînes de caractères.

```
int nombre = 3 ;  
String message = "Mon nombre est " + nombre ;
```

La classe StringBuffer

- La classe **StringBuffer** est une classe Java qui permet de manipuler des chaînes de caractères **modifiables** (mutables), contrairement à **String** qui est **immuable**.
- **StringBuffer** est une classe utilisée pour construire et modifier des chaînes de caractères **de manière efficace**, surtout lorsque ces modifications sont nombreuses.
- Classe **java.lang.StringBuffer** :
 - Un objet de la classe **StringBuffer** se caractérise par deux tailles, qui sont retournées par les méthodes :
 - **int length()** : qui retourne le nombre de caractères exacte du contenu de la chaîne de caractère.
 - **int capacity()** : qui retourne la taille actuelle du tableau interne

La classe StringBuffer

- Quelques méthodes supplémentaires:
 - `StringBuffer append(p)` : ajoute **p** en fin de chaîne (**p** est n'importe quel type de base)
 - `StringBuffer insert(int offset, p)` : même chose, mais en insérant **p** à l'emplacement indiqué par **offset**.
 - `StringBuffer reverse()` : inversion des caractères de la chaîne.
 - `delete(int start, int end)` : supprimer une partie.

La classe StringBuffer

- Exemple 1:

```
StringBuffer s = new StringBuffer("Bonjour");  
s.append(" tout le monde");  
System.out.println(s);  
System.out.println("length : " + s.length());  
System.out.println("capacity : " + s.capacity());
```

- Résultat :

```
Bonjour tout le monde  
length : 21  
capacity : 23
```

- Exemple 2 :

```
s.insert(2, "ISIMM");  
System.out.println(s);
```

- Résultat :

```
BoISIMMnjour tout le monde
```

- Exemple 3 :

```
System.out.println(s.reverse());
```

- Résultat :

```
ednom el tuot ruojnMMISIoB
```

La classe Math

- Cette classe **final** rassemble l'ensemble des méthodes de calcul mathématique Java.
- Classe “vide”
- Ne permet pas de créer des instances;
- Ne contient que des méthodes et des variables *de classe* (déclarées avec le mot-clé *static*), donc qui existent dès l'activation de la classe, sans même qu'une instance soit créée.
- Fonctions mathématiques
 - sin, cos, abs, max, log, etc.
- Constantes mathématiques
 - e, PI

La classe Math

- `public static double asin (double a)` Renvoie l'arc sinus (\sin^{-1}) du nombre a,
- `public static double acos (double a)` Renvoie l'arc cosinus (\cos^{-1}) du nombre a
- `public static double atan (double a)` Renvoie l'arc tangente (\tan^{-1}) du nombre a
- `public static double atan2 (double a, double b)` Renvoie l'arc tangente (\tan^{-1}) de a/b,
- `public static double exp (double a)` et `public static double log (double a)`: Ces méthodes permettent de calculer l'exponentielle ou le logarithme népérien du nombre a.
- `public static double sqrt (double a)` Renvoie la racine carrée du nombre a.
- `public static double pow (double a, double b)` Renvoie le nombre a à la puissance b (a^b).
- `public static double random ()` Renvoie un nombre aléatoire compris entre 0. inclus et 1. exclu.

La classe Graphics

- La classe **Graphics** est une classe abstraite contenant des informations permettant de dessiner des formes géométriques ou des textes
- Les principales informations :
 - le composant sur lequel il faut dessiner
 - la couleur du dessin
 - la fonte utilisée
- La classe **Graphics** définit de nombreuses méthodes pour tracer du texte ou des formes géométriques

La classe Graphics

- Quelques méthodes :

- `drawLine (int, int, int, int)` : trace une ligne droite entre deux points
- `drawRect (int, int, int, int)` : trace un rectangle
- `drawOval (int, int, int, int)` : trace un ovale dans le rectangle définit
- `drawString (String, int, int)` : trace une chaîne de caractères
- `fillRect (int, int, int, int)` : remplit un rectangle
- `getColor ()` : fournit la couleur courante
- `getFont ()` : fournit la fonte courante
- `setColor (Color)` : change la couleur courante
- `setFont (Font)` : change la fonte courante

- Des exemples :

```
g.setColor (Color.red); // couleur courante = rouge
g.drawString ("Bonjour", 5 , 14); // Bonjour en x=5 et y=14
```