

## Console

Résumé des infos utiles .....	3
Introduction à Java .....	17
Les variables et les opérateurs .....	18
Lire les entrées clavier .....	22
Les conditions .....	23
Les boucles .....	25
Les tableaux .....	27
Les méthodes de classe .....	28
Arguments de la ligne de commande .....	31
La première classe .....	32
L'héritage .....	36
Modéliser des objets grâce à UML .....	39
Les packages .....	40
Les classes abstraites et les interfaces .....	41
Les exceptions .....	52
Les énumérations .....	57
Les collections d'objets .....	59
La généricité en Java .....	65
Le flux d'entrée / sortie .....	69
Java et la réfléxivité .....	95

## Interfaces Graphiques

Les interfaces graphiques .....	98
Le fil rouge : une animation .....	103
Positionner des boutons .....	106
Interagir avec des boutons .....	110
Exécuter des tâches simultanément .....	123
Les champs de formulaire .....	134
Les menus et boîtes de dialogue .....	141

Conteneurs, sliders et barres de progression .....	148
Les interfaces des tableaux .....	153
Mieux structurer son code : Le pattern MVC.....	158
Le drag'n drop.....	167
Mieux gérer les interactions avec les composants.....	170

### **Base de données**

JDBC : la porte d'accès aux bases de données .....	175
Fouiller dans sa base de données.....	179
Limiter le nombre de connexions.....	188

### **Annexe**

Annexe : La documentation Java.....	189
Annexe : Les annotations .....	196
Annexe : Les tests unitaires.....	202
Annexe : Le debugging en Java.....	214
Annexe : Les mises-à-jour Java au fil des années .....	216

## Résumé des infos utiles

- La JVM fait fonctionner les programmes Java, précompilés en byte code
- Fichiers contenant du code source **.java**; fichiers précompilés **.class**
- Tous les programmes Java sont composé d'au moins une classe
- Il y a une seule méthode main active par projet : `public static void main(String[] args)`
- **Commentaires** : `//` ou `/* */`
- **Variables**
  - variable de type **primitif** : byte, short, int, long, float, double, boolean, char
    - on peut utiliser des `_` dans l'initialisation des types numériques : `int nb = 32_000;`
    - nombre en format hexadécimal : `int entier = 0x14; (= 20)`
    - nombre en format binaire : `int entier = 0b1111_1111; (= 255)`
    - changer le type de variable (cast) : `(int)x; (double)x; Integer.parseInt(s); ...`
    - afficher 2 décimales : `String.format("%.2f", floatVar)`
  - variable **complexe** : String + tous les objets
    - méthodes utiles pour les objets String :  
`length(), charAt(i), contains("..."), endsWith("..."), startsWith("..."),  
indexOf("..."), indexOf("...", i), lastIndexOf("..."), lastIndexOf("...", i),  
split("..."), substring(i), substring(i, j), replace("...", "..."), replaceFirst("...", "..."),  
equals(s), equalsIgnoreCase(s), compareTo(s) (-1 < 0 = ; +1 >), compareToIgnoreCase(s),  
toLowerCase(), toUpperCase(), toCharArray(), trim(), isEmpty()`
    - formater les Strings :  

```
System.out.printf("Float var = %f,  
                  integer var = %d,  
                  string var = %s\n",  
                  floatVar, intVar, stringVar);  
  
System.out.println(String.format("Float var = %.2f,  
                                 integer var = %02d,  
                                 string var = %7s",  
                                 floatVar, intVar, stringVar));
```
- Convention de **nommage** :
  - noms de classe : commencent par une majuscule
  - noms de variables : commencent par une minuscule
  - si les noms de variables sont composés de plusieurs mots : premier mot minuscule + les autres mots commencent par une majuscule
  - sans accentuation
  - nom du package (ensemble dossiers et sous-dossiers permettant de ranger les classes) :
    - écrits entièrement en minuscules (a-z, A-Z, 0-9)
    - peuvent contenir des points
    - doit commencer par com, edu, gov, mil, net, org ou fr / en / ...
    - ne doit contenir aucun mot clé Java, sauf s'il est suivi par un `_`

- **Opérateurs :**
  - +, -, /, \*, %
  - "+" sert aussi à concaténer des Strings : `System.out.println("Le résultat est = " + resultat);`
  - opérations mathématiques, class **Math** :
   
`Math.random() ([0, 1[]), Math.abs(f), Math.ceil(d), Math.floor(d), Math.exp(d), Math.pow(x, y),  
Math.log(d), Math.log10(d), Math.max(x, y), Math.min(x, y), Math.sqrt(d), Math.sin(d), ...`
- L'entrée et la sortie **I/O** :
  - écrire à l'écran : `System.out.println("...");`
  - lire du clavier, en utilisant la classe **Scanner** :
 

```
Scanner sc = new Scanner(System.in);

System.out.print("Veuillez saisir un mot: ");
String str = sc.nextLine();

System.out.print("Veuillez saisir un numero: ");
int ab = sc.nextInt();                                // nextDouble(), nextFloat(), ...
sc.nextLine();           // repositionner la tête de lecture apres avoir lu un entier, double ...
```
- **Conditions :**
  - if else; if else if else;
  - switch : `switch (note) { case 0 : ..; break; case 1 : ..; break; default: ..; }`
  - condition ternaire : `(x < y ) ? y : x;`
- **Boucles :**
  - while : `while (...) { ... }`
  - do while : `do { ... } while ( ...);`
  - for : `for (int i = 0; i < 10; i++) { ... }`
  - for : `for ( String word : words) {...}` // words = array de String
- **Tableaux :**
  - de int : `int[] tableauEntiers = {7, 1, 5, 2, 3};`
  - de int (sans initialisation): `int[] tableauEntiersSI = new int[6];`
  - de double : `double[] tableauDoubles = {0.0, 0.1, 0.2};`
  - de char : `char[] tableauCaracteres = {'a', 'b', 'c'};`
  - de String : `String[] words = {"bonjour", "allo", "ici"};`
  - multidimensionnel : `int premiersNombres[][] = {{0,2,4}, {1,3,5,7,9}};`
- Utilisation de tableaux :
  - accès : `tableauEntiers[1];`
  - taille : `tableauEntiers.length;`
  - passer comme paramètre à une méthode :

```

public static void printArray(int[] array)
{
    for (int i = 0; i < array.length; i++)
        System.out.print(array[i] + " ");
}

```

- méthodes de la classe Arrays :
  - **Arrays.sort(tableauEntiers);**
  - **Arrays.binarySearch(tableauEntiers, 2);** // => 1
  - **Arrays.toString(tableauEntiers);** // => [1, 2, 3, 5, 7]
  - **Arrays.fill(tableauEntiers, 0)** // => [0, 0, 0, 0, 0]
  - **Arrays.equals(tableauEntiers, tableauEntiersSI)** // => false
  - **Arrays.asList(words).contains("allo");** // marche que sur les objets !!!

- Les **méthodes** en Java :
  - définies dans une classe
  - ne peuvent pas être imbriquées
  - l'on peut utiliser les méthodes prédéfinies ou créer nos propres méthodes
    - **portée de la méthode** : public, protected, private
    - **static** (optionnel)
    - **type de retour** : void / type primitif / objets
    - **nom** de la méthode
    - **arguments** de la méthode (pas de limite)
    - instruction **return** à l'intérieur de la méthode

```

public static double arrondi(double A, int B)
{
    return (double) ( (int) (A * Math.pow(10, B) + .5)) / Math.pow(10, B);
}

```

- La **surcharge des méthodes**  
 Conserver le nom et le type de retour d'une méthode et  
**changer la liste ou le type des paramètres.**

```

static void parcourirTableau(String[] tab)
{
    for(String str : tab)
        System.out.println(str);
}

static void parcourirTableau(int[] tab)
{
    for(int str : tab)
        System.out.println(str);
}

```

- Les **classes**
  - une classe permet de définir des **objets**
  - un nouvel objet est instance grâce au mot clé **new**
  - permettent d'encapsuler des données et du code
    - principe de l'encapsulation : protéger les données de l'extérieur
  - ont des variables : **d'instance** (attributs), **de classe** (communes à toutes les classes)
    - il est recommandé de déclarer les variables d'instance private
    - il faut les déclarer **protected** pour qu'elles soient visibles dans les classes filles
    - une portée par défaut = visibilité uniquement dans le package courant
    - les variables de classes doivent être déclarées **static**
  - ont des méthodes : **constructeurs, accesseurs et mutateurs, d'instance**
    - les constructeurs ont le même nom que la classe et n'ont pas de type de retour
    - un objet peut avoir plusieurs constructeurs (la même méthode, mais surchargé)
    - l'ordre des paramètres passés dans le constructeur doit être respecté
    - les accesseurs (`getX`) permettent d'accéder aux variables d'instance
    - les mutateurs (`setX`) permettent de modifier les variables d'instance
  - dans une classe, on accède aux variables de celle-ci grâce au mot clé **this**  
(référence à l'objet courant)
- L'héritage
  - créer des classes héritées (classes dérivées, **classes filles**) de **classes mères**
  - toutes les classes en Java héritent automatiquement de la classe Object
  - une classe peut hériter d'une seule classe
  - une classe hérite d'une autre classe par le biais du mot clé **extends**

```
class Capitale extends Ville {}
```
  - la classe fille hérite de toutes les propriétés et méthodes déclarées *public* et *protected*
  - l'on peut **redéfinir** une méthode héritée (changer son code) => **polymorphisme !!**
    - une méthode polymorphe a un **squelette identique** à la méthode de base, mais traite les choses différemment
    - si une méthode de la classe mère n'est pas redéfinie dans la classe fille, à l'appel de celle-ci avec un objet fille, c'est la méthode de la classe mère qui sera invoquée
  - si aucun constructeur n'est défini dans une classe fille, la JVM appellera automatiquement le constructeur de la classe mère

- l'on peut utiliser le comportement d'une classe mère par le biais du mot clé **super**

```
public Capitale(String nom, int hab, String pays, String monument)
{
    super(nom, hab, pays);
    this.monument = monument;
}
```

- les méthodes de type **final** ne peuvent pas être redéfinies dans une classe fille

```
public final int maMethode()
{ // méthode ne pouvant pas être redéfinie }
```

- les classes de type **final** ne peuvent pas être héritées

```
public final class FinalClass
{ // classe ne pouvant pas être héritée }
```

- méthodes souvent redéfinies lors d'un héritage : `toString()`, `equals(Object)`, `hashCode()`

- Les **classes abstraites**

- une classe qui **ne peut pas être instanciée**
- servent à définir une *superclasse* : pour créer un nouveau type d'objets
- une classe est définie comme abstraite avec le mot clé **abstract**

```
abstract class Animal {}
```

- contient la même chose qu'une classe normale
- peut contenir en plus des **méthodes abstraites** : méthodes qui **n'ont pas de corps** et qui doivent forcément être **redéfinies dans les classes fille**

```
abstract class Animal           // une classe abstraite = ne peut pas être instancié
{
    abstract void manger();     // une méthode abstraite = n'a pas de corps
}
```

- Les **interfaces**

- sont des **classes 100% abstraites**
- toutes ses méthodes sont abstraites : n'ont pas de corps ! (*sans le mot abstract*)
- une interface est définie avec le mot clé **interface**

```
public interface I1
{
    public void A();           // méthode abstraite
    public String B();         // méthode abstraite
}
```

- une interface s'implémente dans une classe en utilisant le mot clé **implements**
- une classe peut implémenter un **nombre illimité d'interfaces**
- il faut **redéfinir toutes les méthodes de l'interface** dans la classe

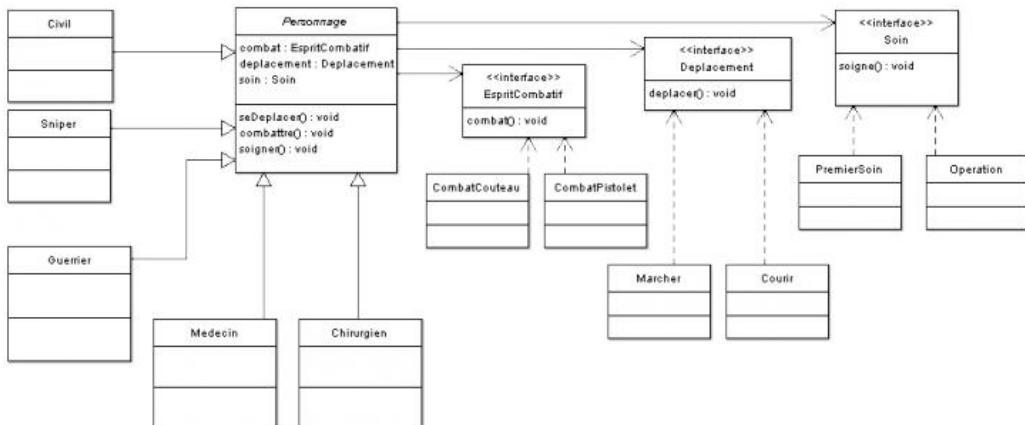
```
public class X implements I1, I2
{
    public void A()
    {
        { ... }
    }
    public String B()
    {
        { ... }
    }
    public void C()
    {
        { ... }
    }
    public String D()
    {
        { ... }
    }
}
```

- Une classe qui hérite d'une autre classe et qui implémente des interfaces doit respecter **l'ordre des déclarations** : l'expression d'héritage avant l'expression d'implémentation

```
public class Chien extends Canin implements Rintintin
```

- Le **pattern strategy**

- permet de rendre la hiérarchie des classes plus souple
- son principe de base : **isoler ce qui varie dans le programme et l'encapsuler**
- préfère la **composition ("a un")** à l'héritage ("est un")



- Le **pattern decorator**

- ajouter des fonctionnalités à un objet de façon dynamique (sans modifier le code source)
- l'invocation des méthodes se fait en allant jusqu'au dernier élément et remonte ensuite la pile des invocations

```
Patisserie pat = new CoucheChocolat(
    new CoucheCaramel(
        new CoucheBiscuit(
            new Gateau()))));
```

- Les exceptions

- un événement que la JVM ne sait pas gérer ( == une erreur)
- lorsqu'un programme rencontre une exception, son exécution est interrompue
- le principe de gestion d'exceptions
  - repérer un morceau de code qui pourrait générer une exception
  - **capturer** l'exception correspondante : avec un bloc **try{...} catch{...}**
  - traiter l'exception correspondante
- lorsqu'une ligne de code lève une exception, l'instruction dans le bloc *try* est interrompue et le programme se rend dans le bloc *catch correspondant à l'exception levée*
- le **paramètre de la clause catch** indique le type d'exception qui doit être capturé; son objet associé peut servir à préciser le message d'erreur grâce à l'appel de la méthode **getMessage()** ou **printStackTrace()**
- la clause **finally** permet d'exécuter du code quoi qu'il arrive (s'il y a eu une exception ou pas). Peut être utile pour fermer un fichier, une connexion à une base de données, ...

```
try
{
    System.out.println(j/i); // code susceptible de lever une exception
}
catch (ArithmeticException ex)
{
    // code exécuté lorsqu'une exception de type ArithmeticException est levé
    System.out.println("Division par 0! Erreur : " + ex.getMessage());
}
finally
{
    // code exécuté quoi qu'il arrive (s'il y a une exception qui est levé ou non)
    System.out.println("action faite systématiquement");
}
```

- Il est possible de catcher plusieurs exceptions dans l'instruction catch()

```
catch (NombreHabitantException | NomVilleException ex) {...}
```

- la superclasse qui gère les exceptions : *Exception*
- pour créer une classe d'exception personnalisée : elle doit hériter de la classe *Exception*

```
class NombreHabitantException extends Exception
{
    public NombreHabitantException() // constructeur par défaut
    {
        System.out.println("Instancier la Ville avec un nb négatif !");
    }
}

public Ville(String nom, int nb) throws NombreHabitantException
{
    if (nb < 0) { throw new NombreHabitantException(); }
    else { ... }
}
```

- **Énumérations** : une classe contenant une liste de sous-objets
  - se déclare avec le mot clé `enum`
  - chaque élément d'une énumération est un objet à part entière
  - peut contenir des méthodes

```
public enum Langage
{
    JAVA ("Langage JAVA"),
    C ("Langage C"),
    PHP ("Langage PHP");
```

```
private String name = "";
```

```
Langage(String name) // constructeur
    { this.name = name; }
```

```
}
```

```
// main
for (Langage lang : Langage.values() )
{
    if ( Langage.JAVA.equals(lang) )
        System.out.println("J'aime le : " + lang);
    else
        System.out.println(lang);
}
```

```
// le résultat :
J'aime le : Langage JAVA
Langage C
Langage PHP
```

- **Collections** : stocker un nombre variable d'objets, sans taille prédéfinie
  - **LinkedList** (`List`) : chaque élément est lié aux éléments adjacents par une référence

Méthodes utiles :

```
size(), add(x), get(i), remove(i), remove((Object)x), contains(x),
indexOf(x), lastIndexOf(), clear(), isEmpty(), toString(),
toArray(), iterator()
```

```
List l = new LinkedList();
l.add(12);
l.add("toto ! !");
l.add(12.20f);
for (int i = 0; i < l.size(); i++)
    System.out.println("Élément à l'index " + i + " = " + l.get(i));
```

- o **ArrayList** (List) : pas de taille limite; accepte n'importe quel type de données

```
ArrayList al = new ArrayList();
```

Contient les mêmes méthodes que les objets LinkedList.

Il est possible de restreindre le type de données acceptés par la liste.

```
List<String> list = new ArrayList<String>();
list.add("sravan");
list.add("vasu");
list.add("raki");

Iterator it = list.iterator();
while ( it.hasNext() )
    System.out.println(it.next());
```

**Conversion** d'un objet List vers un array :

```
String[] names = list.toArray(new String[list.size()]);
Integer[] numbers = listI.toArray(new Integer[listI.size()])
```

**Conversion** d'un array des Integer vers un objet List :

```
Integer[] tableIntP = {7, 0, 5, 1, 9, 2, 3, 4, 5, 6, 7, 8, 9};
List<Integer> al = Arrays.asList(tableIntP);
```

Contrairement aux LinkedList, les **ArrayList** sont :

- **rapides en lecture**, même avec un gros volume d'objets
- **lentes** lors de l'**ajout** ou de la **suppression** des éléments **en milieu de liste**.

- o **HashSet** (Set) : n'autorise pas les doublons

Contient les mêmes méthodes que les objets de type List.

```
HashSet hs = new HashSet();
hs.add("toto");
hs.add(12);
hs.add('d');
Iterator it = hs.iterator();
while ( it.hasNext() )
    System.out.println(it.next());
```

- o **HashMap** (Map) : collections qui fonctionnent avec un système clé - valeur

La clé sert à identifier une entrée; elle est unique. La valeur peut être associé à plusieurs clés. Accepte la valeur null.

Leur points faibles : la taille des données à stocker, ils ne sont pas Thread Safe  
(=> class *ConcurrentHashMap*).

Méthodes utiles :

**put(key, value)**, **putIfAbsent(key, value)**, **get(key)** (=> value), **size()**,  
**containsKey(Object)**, **containsValue(Object)**, **remove(key)** (=> delete key and value),  
**keySet()**, **values()**, **entrySet()**, **clear()**, **isEmpty()**

```

HashMap ex = new HashMap();
HashMap<Integer, String> ex = new HashMap<Integer, String> ();
hm.put(1, "printemps");
hm.put(10, "été");
hm.put(12, "automne");
hm.put(45, "hiver");

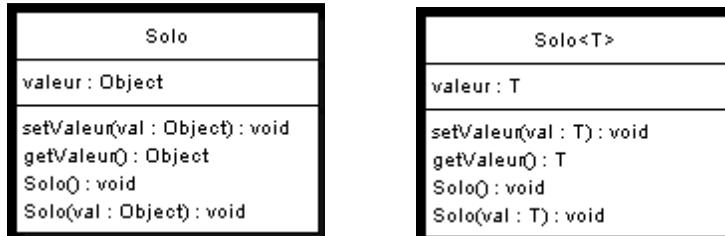
for ( Object key : hm.keySet() )
{ System.out.println(key);}

for ( Object value : hm.values() )
{ System.out.println(value);}

for ( Map.Entry<Integer, String> entry : hm.entrySet() )
{
    Integer key = entry.getKey();
    String value = entry.getValue();
    System.out.println(key + " , " + value);
}

```

- La **généricité**
  - concept permettant de ne pas spécifier de type précis pour une classe, une collection ou une méthode afin d'avoir du code réutilisable
  - exemple d'une classe sans et avec généricité



- le type <T> de l'attribut sera défini lors de l'instanciation de la classe

```

public class Solo<T>
{
    private T valeur;           // variable d'instance; type inconnu
    public Solo()               // constructeur par défaut
        { this.valeur = null; }
    public Solo(T val)          // constructeur avec paramètre de type inconnu
        { this.valeur = val; }
}

```

```

// main
Solo<Integer> valI = new Solo<Integer>();
Solo<String> valS = new Solo<String>("TOTOTOTO");
Solo<Float> valF = new Solo<Float>(12.2f);
Solo<Double> valD = new Solo<Double>(12.202568);

```

- cette classe fonctionne donc avec **tous les types de données**
- une fois instancié avec un type, l'objet ne pourra travailler qu'avec ce type de données
- l'on peut coupler les collections avec la généricité

```
List<String> listeString= new ArrayList<String>();
```

```
List<Float> listeString= new ArrayList<Float>();
```

- le **wildcard <?>** permet d'indiquer que n'importe quel type peut être traité et accepté
- ```
ArrayList<?> list;
```
- Il y a néanmoins une restriction : lors de l'utilisation du wildcard, ladite collection est rendue en lecture seule.
- pour élargir le champ d'acceptation d'une collection générique: utiliser le mot clé **extends**  
L'instruction **<? extends MaClass>** autorise toutes les collections de classes ayant pour supertype MaClasse.

Exemple : List n'acceptant que des instances de Voiture ou de ses classes filles

```
public static void main(String[] args)
{
    List<? extends Voiture> listVSP = new ArrayList<VoitureSansPermis>();
    afficher(listVSP);
}
```

- Les **flux d'entrée / sortie**

- les classes traitant des entrées / sorties se trouvent dans le package **java.io**
  - pour l'échange de données entre le programme et une autre source, Java emploie un stream (un flux). Celui-ci joue le rôle de médiateur entre la source des données et sa destination.
  - toute opération sur les entrées / sorties doit suivre le schéma :
- ouverture, lecture / écriture, fermeture du flux**

- l'objet **File**

```
File fileW = new File("test.txt");
```

- l'objet **FileWriter** : pour écrire dans des fichiers

```
try ( FileWriter fw = new FileWriter(fileW) )           // try-with-resources
{
    fw.write("Bonjour à tous, amis Zéros !\n");
    fw.write("\tComment allez-vous ? \n");
}
catch (FileNotFoundException | IOException e)
{
    System.out.println("Catch error : " + e.getMessage());
}
```

- l'objet **BufferedReader** : pour lire dans les fichiers

```

try (
    FileInputStream fis = new FileInputStream(fileW);
    InputStreamReader isr = new InputStreamReader(fis, Charset.forName("UTF-8"));
    BufferedReader br = new BufferedReader(isr); )
{
    while ( (line = br.readLine()) != null )
    {
        String[] words = line.split("\\s");
        for ( String word : words)
            System.out.println(word);      // pour int : println(Integer.parseInt(word))
    }
}
catch (FileNotFoundException | IOException e)
{ System.out.println("Catch error : " + e.getMessage()); }

```

- l'objet **Scanner** : pour lire dans les fichiers

```

try ( Scanner sc = new Scanner(fileW) )
{
    while ( sc.hasNext() )
    {
        String word = sc.next();        // pour int : int nb = Integer.parseInt(sc.next());
        System.out.println(word);
    }
}
catch (FileNotFoundException | IOException e)
{
    System.out.println("Catch error : " + e.getMessage());
}

```

- les objets **ObjectInputStream** et **ObjectOutputStream** : pour **sérialiser les objets**

Pour qu'un objet soit sérialisable il doit implémenter l'interface **Serializable** (interface sans rien à redéfinir; une interface marqueur)

```

import java.io.Serializable;
public class Game implements Serializable
{ ... }

// main
ObjectInputStream ois;
ObjectOutputStream oos;

try
{
    oos = new ObjectOutputStream(
        new BufferedOutputStream(
            new FileOutputStream(
                new File("game.txt"))));
}

```

```

oos.writeObject(new Game("Assassin Creed", "Aventure", 45.69));
oos.writeObject(new Game("Assassin Creed", "Aventure", 45.69));
oos.close();

ois = new ObjectInputStream(
    new BufferedInputStream(
        new FileInputStream(
            new File("game.txt"))));
try
{
    System.out.println(Affichage des jeux :);
    System.out.println("*****\n");
    System.out.println(((Game)ois.readObject()).toString());
    System.out.println(((Game)ois.readObject()).toString());
    System.out.println(((Game)ois.readObject()).toString());
}
catch (ClassNotFoundException e)
{ e.printStackTrace(); }

ois.close();                                // ne pas oublier de fermer le flux !
}
catch (FileNotFoundException | IOException e)
{ e.printStackTrace(); }

```

- La **réflexivité** ou l'**introspection**

- découvrir de façon dynamique des informations relatives à une classe ou à un objet
- un moyen de connaître toutes les informations concernant une classe donnée

```

Class c1 = String.class;
Class c2 = new String().getClass();

String name = new String().getClass().getSimpleName();
System.out.println(" La superclasse de String : " + c1.getSuperclass());
// Ses interfaces : c1.getInterfaces();
// Ses méthodes : c1.getMethods();
// Ses attributs : c1.getDeclaredFields();
// Ses constructeurs : c1.getConstructors();

```



## Introduction à Java

- **Fonctionnement des programmes Java :**
  - La JVM (Java Virtual Machine) est le cœur de Java.
  - Elle fait fonctionner vos programmes Java, précompilés en byte code. Le byte code est un code intermédiaire entre celui de votre programme et celui que votre machine peut comprendre. Votre machine NE PEUT PAS comprendre le byte code, elle a besoin de la JVM.
  - Les fichiers contenant le code source de vos programmes Java ont l'extension **.java**.
  - Les fichiers précompilés correspondant à vos codes source Java ont l'extension **.class**.
  - Un programme Java, codé sous Windows, peut être précompilé sous Mac et enfin exécuté sous Linux.
- **Composition d'un programme Java :**
  - Tous les programmes Java sont composés d'au moins une classe.
  - Le point de départ de tout programme Java est la méthode

```
public static void main(String[] args)
```
- On peut **afficher des messages** dans la console grâce à ces instructions :
  - `System.out.println()`, qui affiche un message avec un saut de ligne à la fin ;
  - `System.out.print()`, qui affiche un message sans saut de ligne.
- **Commentaires :**
  - simples, sur une seule ligne : avec **//**
  - complexes, sur multiples lignes : avec **/\*** et **\*/**
- Présentation **Eclipse** :
  - File : créer de nouveau projets java / les enregistrer / les exporter
  - Raccourcis :
    - ALT + SHIFT + N = nouveau projet
    - CTRL + S = enregistrer le fichier courant; CTRL + SHIFT + S = tout sauvegarder
    - CTRL + W = fermer le fichier courant; CTRL + SHIFT + W = tous les fermer
  - Edit : commandes "copier", "coller", etc
  - Window : configurer Eclipse selon nos besoins
  - Barre d'outils :
    - 1 : nouveau général
    - 2 : enregistrer
    - 3 : imprimer
    - 4 : exécuter la classe ou le projet spécifié
    - 5 : créer un nouveau projet
    - 6 : créer une nouvelle classe

## Les variables et les opérateurs

- Une **variable** est un élément qui stocke des informations de toute sorte en mémoire : des chiffres, des résultats de calcul, des tableaux, des renseignements fournis par l'utilisateur...

- Une déclaration de variable se fait comme ceci :

```
<Type de la variable> <Nom de la variable> ;
```

- **Types de variables** :

- primitives (simples)

- **byte** (1 octet) : peut contenir les entiers entre -128 et +127
    - **short** (2 octets) : contient les entiers compris entre -32768 et +32767
    - **int** (4 octets) : va de  $-2 \cdot 10^9$  à  $2 \cdot 10^9$  (9 zéros)
    - **long** (8 octets) : va de  $-9 \cdot 10^{18}$  à  $9 \cdot 10^{18}$  (18 zéros)  
Lors de l'initialisation, il faut ajouter un "**L**" à la fin du nombre
    - **float** (4 octets) est utilisé pour les nombres avec une virgule flottante  
Lors de l'initialisation, il faut ajouter un "**f**" à la fin du nombre
    - **double** (8 octets) : identique à float; il contient plus de chiffres derrière la virgule et il n'a pas de suffixe; il faut ajouter la lettre "**d**" pour la déclaration
    - **booléen** : il peut contenir deux variables **true** / **false**
    - **char** : un caractère stocké entre apostrophes

```
byte temperature;
temperature = 64;

short vitesseMax;
vitesseMax = 32000;

int temperatureSoleil;
temperatureSoleil = 15600000;           // la température est exprimée en kelvins

long anneeLumiere;
anneeLumiere = 9460700000000000L;

float pi;
pi = 3.141592653f;

float nombre;
nombre = 2.0f;

double division;
division = 0.3334d;

boolean question;
question = true;

char caractere;
caractere = 'A';

int entier = 32;
float pi = 3.1416f;
char carac = 'z';

int nbre1 = 2, nbre2 = 3, nbre3 = 0;
```

- o complexes (objets)
  - **String** : permet de gérer les chaînes de caractères

```
//Première méthode de déclaration
String phrase;
phrase = "Titi et Grosminet";

//Deuxième méthode de déclaration
String str = new String();
str = "Une autre chaîne de caractères";

//Troisième méthode de déclaration
String string = "Une autre chaîne";

//Quatrième méthode de déclaration
String chaine = new String("Et une de plus !");
```

- **Conventions de nommage :**
  - o tous les noms de **classes** doivent commencer par une **majuscule**
  - o tous les noms de **variables** doivent commencer par une **minuscule**
  - o si le nom d'une variable est composé de **plusieurs mots**, le premier commence par une **minuscule**, le ou les autres par une **majuscule**, et ce, sans séparation
  - o tout ceci sans accentuation !
- **Les opérateurs arithmétiques :**
  - o « **+** » : permet **d'additionner** deux variables numériques (mais aussi de **concaténer** des chaînes de caractères ; ne vous inquiétez pas, on aura l'occasion d'y revenir).
  - o « **-** » : permet de **soustraire** deux variables numériques.
  - o « **\*** » : permet de **multiplier** deux variables numériques.
  - o « **/** » : permet de **diviser** deux variables numériques
  - o « **%** » : permet de renvoyer le reste de la division entière de deux variables de type numérique ; cet opérateur s'appelle le **modulo**.

```
int nbre1, nbre2, nbre3; //Déclaration des variables
nbre1 = 1 + 3;           //nbre1 = 4
nbre2 = 2 * 6;           //nbre2 = 12
nbre3 = nbre2 / nbre1;   //nbre3 = 3
nbre1 = 5 % 2;           //nbre1 = 1, car 5 = 2 * 2 + 1
nbre2 = 99 % 8;          //nbre2 = 3, car 99 = 8 * 12 + 3
nbre3 = 6 % 3;           //là, nbre3 = 0, car il n'y a pas de reste
nbre1 = nbre1 + 1;
nbre1 += 1;               // raccourci incrémentation
nbre1++;
++nbre1;
```

```

nbre1 = nbre1 - 1;           // raccourcis soustraction
nbre1 -= 1;
nbre1--;
--nbre1;

nbre1 = nbre1 * 2;          // raccourcis multiplication et division
nbre1 *= 2;
nbre1 = nbre1 / 2;
nbre1 /= 2;

```

On ne peut faire du **traitement arithmétique que sur des variables de même type** sous peine de perdre de la précision lors du calcul. On ne divise pas un int par un float, ou pire, par un char !

**Concaténation** (mélanger du texte brut avec des variables) :

```

double nbre1 = 10, nbre2 = 3;
int resultat = (int)(nbre1 / nbre2);
System.out.println("Le résultat est = " + resultat);

```

- Les **conversions**, ou "cast"

- les variables de type double contiennent plus d'informations que les variables de type int
- au besoin, l'on serait amenés à convertir des variables

```

int i = 123;
float j = (float)i;

int i = 123;
double j = (double)i;

double i = 1.23;
double j = 2.9999999;
int k = (int)i;           // k vaut 1
k = (int)j;               // k vaut 2

double nbre1 = 10, nbre2 = 3;
int resultat = (int)(nbre1 / nbre2);
System.out.println("Le résultat est = " + resultat);    // =3

int nbre1 = 3, nbre2 = 2;
double resultat = nbre1 / nbre2;
System.out.println("Le résultat est = " + resultat);    // =1

int nbre1 = 3, nbre2 = 2;
double resultat = (double)(nbre1 / nbre2);
System.out.println("Le résultat est = " + resultat);    // =1

```

En Java, comme dans d'autres langages d'ailleurs, il y a la notion de **priorité d'opération**. L'**affectation**, le **calcul**, le **cast**, le **test**, l'**incrémentation**... toutes ces choses sont des opérations ! Et Java les fait dans un certain ordre, il y a des priorités.

Dans le cas qui nous intéresse, il y a trois opérations : un **calcul**, un **cast** sur le résultat de l'opération et une **affectation** dans la variable *resultat*. Eh bien, Java exécute notre ligne dans cet ordre ! Il fait le **calcul** (ici 3/2), il **caste** le résultat en double, puis il l'**affecte** dans notre variable *resultat*.

Pour avoir un résultat correct, il faudrait caster chaque nombre avant de faire l'opération :

```
int nbre1 = 3, nbre2 = 2;
double resultat = (double)(nbre1) / (double)(nbre2);
System.out.println("Le résultat est = " + resultat); // = 1.5
```

**Transformer** l'argument d'un type donné, par exemple **int en String, et inversement** :

```
int i = 12;
String j = new String();
j = j.valueOf(i); // j = "12" (String)
int k = Integer.valueOf(j).intValue(); // k = 12 (int)

String j = String.valueOf(i);
int k = Integer.parseInt(j);
j += "3"; // j = "123"
k += 3; // k = 15
```

- Le **formatage des nombres**

Le langage Java est en perpétuelle évolution. Depuis la version 7 de Java, vous avez la possibilité de formater vos variables de types numériques avec un séparateur, l'**underscore** (\_), ce qui peut s'avérer très pratique pour de grands nombres qui peuvent être difficiles à lire. Voici quelques exemples :

```
double nombre = 1000000000000d; // cast en d
double nombre = 1_____000_____000_____000d; // le nombre d'underscores compte pas

int entier = 32_000;
double monDouble = 12_34_56_78_89_10d; // cast en d
double monDouble2 = 1234_5678_8910d; // cast en d
```

Les underscore doivent être **placés entre deux caractères numériques** : ils ne peuvent donc pas être utilisés en début ou en fin de déclaration ni avant ou après un séparateur de décimal. Ainsi, ces déclarations ne sont pas valides :

```
double d = 123_.159;
int entier = _123;
int entier2 = 123_;
```

Avant Java 7, il était possible de déclarer des expressions numériques en **hexadécimal**, en utilisant le préfixe « **0x** » :

```
int entier = 255; // peut s'écrire « int entier = 0xFF; »
int entier = 20; // peut s'écrire « int entier = 0x14; »
int entier = 5112; // peut s'écrire « int entier = 0x13_F8; »
```

Depuis java 7, vous avez aussi la possibilité d'utiliser la notation **binnaire**, en utilisant le préfixe « **0b** » :

```
int entier = 0b1111_1111; // équivalent à : « int entier = 255; »
int entier = 0b1000_0000_0000; // équivalent à : « int entier = 2048; »
int entier = 0b100000000000; // équivalent à : « int entier = 2048; »
```

## Lire les entrées clavier

- Vous pouvez saisir des informations et les stocker dans des variables afin de pouvoir les utiliser *a posteriori*.
- Pour que Java puisse lire ce que vous tapez au clavier, l'on utilise un objet de type **Scanner**. Cet objet peut prendre différents paramètres, mais généralement l'on utilisera un seul : celui qui correspond à l'entrée standard en Java (System.in)

```
import java.util.Scanner; // faut indiquer à Java où se trouve la classe Scanner  
Scanner sc = new Scanner(System.in);
```

Nous devons importer la classe Scanner grâce à l'instruction **import**. La classe que nous voulons se trouve dans le package `java.util`. Un **package** est **un ensemble de classes**. En fait, c'est un ensemble de dossiers et de sous-dossiers contenant une ou plusieurs classes.

Les classes qui se trouvent dans les packages autres que `java.lang` (package **automatiquement importé** par Java, on y trouve entre autres la classe `System`) sont à importer à la main dans vos classes Java pour pouvoir vous en servir.

En Eclipse, la façon pour importer la classe `java.util.Scanner` est très commode. Mais on peut aussi le faire manuellement :

```
import java.util.Scanner; // ceci importe la classe Scanner du package java.util  
import java.util.*; // ceci importe toutes les classes du package java.util
```

- Voici l'instruction pour récupérer ce que vous avez saisi pour ensuite l'afficher :

```
Scanner sc = new Scanner(System.in);  
  
System.out.println("Veuillez saisir un mot :");  
String str = sc.nextLine();  
System.out.println("Vous avez saisi : " + str);  
  
System.out.println("Veuillez saisir un nombre :");  
int str = sc.nextInt();  
System.out.println("Vous avez saisi le nombre : " + str);  
  
System.out.println("Saisissez une lettre :");  
Scanner sc = new Scanner(System.in);  
String str = sc.nextLine();  
char carac = str.charAt(0); // renvoie le premier caractère  
System.out.println("Vous avez saisi le caractère : " + carac);
```

Si on invoque la méthode **nextLine()** après avoir invoqué une méthode comme **nextInt()**, **nextDouble()**, ..., celle-ci ne vous invitera pas à saisir une chaîne de caractères : elle videra la ligne commencée par les autres instructions. En effet, celles-ci **ne reposent pas la tête de lecture**, l'instruction `nextLine()` le fait à leur place. Pour pallier ce problème, il suffit de vider la ligne après les instructions ne les faisant pas automatiquement :

```
Scanner sc = new Scanner(System.in);  
System.out.println("Saisissez un entier : ");  
int i = sc.nextInt();  
System.out.println("Saisissez une chaîne : ");  
sc.nextLine(); // on vide la ligne avant d'en lire une autre  
String str = sc.nextLine();  
System.out.println("FIN ! ");
```

## Les conditions

- La lecture et l'exécution se font de façon **séquentielle**, c'est-à-dire ligne par ligne. Avec les **conditions**, nous allons pouvoir **gérer différents cas de figure** sans pour autant lire tout le code.
- Les opérateurs logiques : ==, !=, <, <=, >, >=, && (and), || (or), ? (l'opérateur ternaire)
- Condition if else :

```
int i = 10;

if (i < 0)
    System.out.println("le nombre est négatif");
else
    System.out.println("le nombre est positif");
```

Condition il else if else :

```
int i = 0;

if (i < 0)
    System.out.println("Ce nombre est négatif !");
else if (i > 0)
    System.out.println("Ce nombre est positif !");
else
    System.out.println("Ce nombre est nul !");
```

Conditions multiples

```
int i = 58;

if (i < 100 && i > 50)
    System.out.println("Le nombre est bien dans l'intervalle.");
else
    System.out.println("Le nombre n'est pas dans l'intervalle.");
```

La structure **switch** : pour éviter des else if à répétition et pour alléger un peu le code

```
int note = 10;      // on imagine que la note maximale est 20

switch (note)
{
    case 0:
        System.out.println("Ouch !");
        break;
    case 10:
        System.out.println("Vous avez juste la moyenne.");
        break;
    case 20:
        System.out.println("Parfait !");
        break;
    default:
        System.out.println("Il faut davantage travailler.");
}
```

Depuis la version 7 de Java, l'instruction **switch** accepte les objets de type **String** en paramètre

```
String chaine = "Bonjour";  
  
switch(chaine)  
{  
    case "Bonjour":  
        System.out.println("Bonjour monsieur !");  
        break;  
    case "Bonsoir":  
        System.out.println("Bonsoir monsieur !");  
        break;  
    default:  
        System.out.println("Bonjoir ! :p");  
}
```

La condition ternaire

```
int x = 10, y = 20;  
int max = (x < y) ? y : x; // maintenant, max vaut 20
```

Si un bloc d'instructions contient plus d'une ligne, vous devez l'entourer d'accolades afin de bien délimiter le début et la fin

## Les boucles

Le rôle des **boucles** est de **répéter un certain nombre de fois les mêmes opérations**. Une boucle s'exécute tant qu'une condition est remplie.

La boucle **while** :

```
int a = 1, b = 15;
while (a < b)
{
    System.out.println("coucou " + a + " fois !!");
    a++; // sans cette instruction, le programme ne s'arrête jamais (boucle infinie)
}
```

Priorité d'opérateurs :

```
int a = 1, b = 15;
while (a++ < b)
    System.out.println("coucou " +a+ " fois !!"); // 2, 3, 4, .. 15

int a = 1, b = 15;
while (++a < b)
    System.out.println("coucou " +a+ " fois !!"); // 2, 3, ..., 14
```

**1er cas** : l'opérateur « < » a la priorité sur l'opérateur d'incrémentation « ++ ». La boucle while teste la condition et ensuite incrémenté la variable a.

**2eme cas** : l'opérateur d'incrémentation est prioritaire sur l'opérateur d'inégalité, c'est-à-dire que la boucle incrémenté la variable a, et ce n'est qu'après l'avoir fait qu'elle teste la condition !

La boucle **do while** : s'exécute au moins une fois, la phase de test de la condition se fait à la fin

```
String prenom = new String();
char reponse = ' ';
Scanner sc = new Scanner(System.in);

do
{
    System.out.println("Donnez un prénom : ");
    prenom = sc.nextLine();
    System.out.println("Bonjour " +prenom+, ", comment vas-tu ?");

    do
    {
        System.out.println("Voulez-vous réessayer ? (O/N)");
        reponse = sc.nextLine().charAt(0);
    } while (reponse != 'O' && reponse != 'N');

} while (reponse == 'O');

System.out.println("Au revoir...");
```

La boucle **for**

```
for (int i = 1; i <= 10; i++)
{
    System.out.println("Voici la ligne " + i);
}

for (int i = 10; i >= 0; i--)
    System.out.println("Il reste " + i + " ligne(s) à écrire");

for (int i = 0, j = 2; (i < 10 && j <= 6); i++, j+=2)
{
    System.out.println("i = " + i + ", j = " + j);      // i=0,1,2; j=2,4,6
}
```

## Les tableaux

- Un tableau est une **variable** contenant **plusieurs données d'un même type**. On peut lui affecter plusieurs valeurs ordonnées séquentiellement qu'on pourra appeler au moyen d'un indice (ou d'un compteur). Il suffit d'introduire l'emplacement du contenu désiré dans la variable tableau pour la sortir, travailler avec, l'afficher...
- Tableau à une dimension :

```
<type du tableau> <nom du tableau> [] = { <contenu du tableau> };  
  
int tableauEntier[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
double tableauDouble[] = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};  
char tableauCaractere[] = {'a','b','c','d','e','f','g'};  
String tableauChaine[] = {"chaine1", "chaine2", "chaine3" , "chaine4"};  
  
int tableauEntier[] = new int[6];  
int[] tableauEntier2 = new int[6];
```

- Tableau multidimensionnel :

```
int premiersNombres[][] = { {0,2,4,6,8}, {1,3,5,7,9} };  
// premiersNombres[0][0] correspondra au premier élément de la 1ere ligne  
// premiersNombres[1][0] correspondra au premier élément de la 2eme ligne
```

Vous pouvez ajouter autant de dimensions à votre tableau que vous le souhaitez, ceci en cumulant des crochets à la déclaration.

- Utiliser et rechercher dans un tableau :

- un tableau débute toujours à l'indice 0
- afficher le contenu d'un tableau

```
char tableauCaractere[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g'};  
for (int i = 0; i < tableauCaractere.length; i++) {  
    System.out.println("À l'emplacement " + i +" du tableau nous avons ");  
    System.out.println(tableauCaractere[i]);  
}
```

```
String tab[] = {"toto", "titi", "tutu", "tete", "tata"};  
for (String mot : tab)  
    System.out.println(mot);
```

```
String tab[][]={{"toto", "titi", "tutu", "tete"}, {"1", "2", "3", "4"}};  
int i = 0, j = 0;  
for (String ligne[] : tab) {  
    i = 0;  
    for (String item : ligne) {  
        System.out.println("La valeur de la nouvelle boucle est : " + item);  
        System.out.println("La valeur du tableau à l'indice [" + j + "][" + i + "] est : " + tab[j][i]);  
        i++;  
    }  
    j++;  
}
```

## Les méthodes de classe

- Une méthode est un **morceau de code réutilisable** qui effectue une action bien définie. Les méthodes **se définissent dans une classe**. Les méthodes **ne peuvent pas être imbriquées**; elles se déclarent les unes après les autres.

- **Méthodes utiles :**

- concernant les chaînes de caractères :

```
String chaine = new String("COUCOU TOUT LE MONDE");
String chaine2 = chaine.toLowerCase();           // => "coucou tout le monde !"
```

```
String chaine = new String("coucou coucou");
String chaine2 = chaine.toUpperCase();           // => "COUCOU COUCOU"
```

```
String chaine = new String("coucou ! ");
int longueur = chaine.length();                  // => 9
```

```
String str1 = new String("coucou"), str2 = new String("toutou");
if ( ! str1.equals(str2) )
    System.out.println("Les deux chaînes sont différentes !");
else
    System.out.println("Les deux chaînes sont identiques !");
```

```
String nbre = new String("1234567");
char carac = nbre.charAt(4);                     // => '5'
```

```
String chaine = new String("la paix niche");
String chaine2 = chaine.substring(3, 13);         // => "paix niche"
// Elle prend deux entiers en arguments :
// - le premier définit le premier caractère (inclus) de la sous-chaîne à extraire,
// - le second correspond au dernier caractère (exclu) à extraire.
```

```
String mot = new String("anticonstitutionnellement");
int n = 0;
n = mot.indexOf('t');                           // => 2
n = mot.indexOf("ti");                          // => 2
n = mot.indexOf('x');                           // => -1
n = mot.lastIndexOf('t');                      // => 24
n = mot.lastIndexOf("ti");                     // => 12
```

- concernant les opérations mathématiques (classe Math) :

```
double x = Math.random();                      // nombre aléatoire entre 0 et 1, comme 0.01356
double sin = Math.sin(120);                    // sinus
double cos = Math.cos(120);                    // cosinus
double tan = Math.tan(120);                    // tangente
double abs = Math.abs(-120.25);               // valeur absolue (retourne le nombre sans le signe)
double d = 2;
double exp = Math.pow(d, 2);                   // La fonction exposant (d élevée au carré)
```

- **Créer ses propres méthodes :**

- exemple :

```
public static double arrondi(double A, int B) {
    return (double) ( (int) (A * Math.pow(10, B) + .5)) / Math.pow(10, B);
}
```

Décortiquons un peu cela :

- tout d'abord, il y a le mot clé **public**, qui définit la **portée de la méthode**
- ensuite, il y a **static** : méthodes faisant partie de la classe principale (main) statiques
- juste après, il y a le type de **retour de la méthode** (ici **double**)
- vient ensuite le **nom de la méthode**
- puis arrivent les **arguments de la méthode**
- finalement, il y a une **instruction return** à l'intérieur de la méthode

- Les méthodes ne sont **pas limitées en nombre de paramètres**.

- Il existe **trois types de méthodes** :

- qui **ne renvoient rien** : pas d'instruction return; elles sont de type void
- qui **retournent des types primitifs** (double, int, etc)
- qui **retournent des objets** (String, ...)

- méthode affichant un tableau

```
static void parcourirTableau(String[] tab)
{
    for(String str : tab)
        System.out.println(str);
}
```

```
static String toString(String[] tab)
{
    System.out.println("Méthode toString() !\n-----");
    String retour = "";

    for(String str : tab)
        retour += str + "\n";

    return retour;
}
```

```
public static void main(String[] args)
{
    String[] tab = {"toto", "tata", "titi", "tete"};
    parcourirTableau(tab);
    System.out.println(toString(tab));
}
```

- **La surcharge des méthodes :**

- La surcharge de méthode consiste à **garder le nom d'une méthode et son type de retour** (donc un type de traitement à faire) et à **changer la liste ou le type de ses paramètres**.
- Dans le cas qui nous intéresse, nous voulons que notre méthode parcourirTableau puisse parcourir n'importe quel type de tableau.
- Surcharger cette méthode afin qu'elle puisse aussi travailler avec des int

```
static void parcourirTableau(String[] tab)
{
    for(String str : tab)
        System.out.println(str);
}

static void parcourirTableau(int[] tab)
{
    for(int str : tab)
        System.out.println(str);
}

static void parcourirTableau(String[][] tab)
{
    for(String tab2[] : tab)
    {
        for(String str : tab2)
            System.out.println(str);
    }
}

static void parcourirTableau(String[][][] tab, int i)
{
    for(String tab2[] : tab)
    {
        for(String str : tab2)
            System.out.println(str);
    }
}
```

Utilisation :

```
String[] tabStr      = {"toto", "titi", "tata"};
int[] tabInt         = {1, 2, 3, 4};
String[][] tabStr2  = {{"1", "2", "3", "4"}, {"toto", "titi", "tata"}};

parcourirTableau(tabStr); // la méthode avec un tableau de String sera invoquée
parcourirTableau(tabInt); // la méthode avec un tableau d'int sera invoquée
parcourirTableau(tabStr2); // la méthode avec un tableau de String à 2 dim invoquée
```

## Arguments de la ligne de commande

Nous allons essayer de récupérer les paramètres fournis par la commande suivante :

```
java MaClasse.java param1 param2 ...
```

Ce que l'on peut récupérer facilement en Java :

```
public class MaClasseTest
{
    public static void main(String[] args)
    {
        System.out.println("Argument 1 : " + args[0] + ", argument 2 : " + args[1]);
    }
}
```

Ce qui affiche tranquillement

```
Argument 1 : param1, argument 2 : param2
```

C'est plutôt pas mal, mais on peut mieux faire.

Par exemple en vérifiant qu'on a bien le bon nombre d'arguments souhaités.

```
public class MaClasseTest
{
    public static void main(String[] args)
    {
        if ( args.length != 2 )
        {
            System.out.println("Usage : <progname> <param1> <param2>");
            exit(0);
        }

        for ( String s : args )
            System.out.println("Argument : " + s);
    }
}
```

Mais tout cela oblige une **notion d'ordre** qui n'est pas forcément la bienvenue.

Pour corriger cela, on peut faire précéder nos paramètres par un flag ou option, qui nous permet de les différencier.

La commande bash devient alors :

```
java MaClasse.java -p1 param1 -p2 param2 ...
```

## La première classe

- Une **classe permet de définir des objets.**

```
public class Ville {}
```

Ils ont des **variables** et des **méthodes**; ils permettent d'encapsuler des données et du code.

Les différents **types de variables** dans un objet :

- les **variables d'instance (attributs)** → variables décrivant l'objet (sa carte d'identité)
- les **variables de classe** → variables communes à toutes les instances de classe
- les **variables locales**

Les différents **types de méthodes** dans un objet :

- les **constructeurs** → méthodes servant à créer des objets ;
- les **accesseurs et mutateurs** → méthodes servant à accéder aux données des objets ;
- les **méthodes d'instance** → méthodes servant à la gestion des objets.

- Le ou les **constructeurs** d'une classe doivent porter le même nom que la classe et n'ont pas de type de retour.

```
public class Ville
{
    String nomVille;                      // stocke le nom de notre ville
    String nomPays;                        // stocke le nom du pays de notre ville
    int nbreHabitants;                    // stocke le nombre d'habitants de notre ville

    public Ville()                         // constructeur par défaut (sans paramètres)
    {
        System.out.println("Création d'une ville !");
        nomVille = "Inconnu";
        nomPays = "Inconnu";
        nbreHabitants = 0;
    }

    public Ville(String nom, int nbre, String pays) // constructeur avec paramètres
    {
        System.out.println("Création d'une ville avec des paramètres !");
        nomVille = nom;
        nomPays = pays;
        nbreHabitants = nbre;
    }
}
```

Un **constructeur** est une méthode d'instance qui va se charger de créer un objet et, le cas échéant, d'initialiser ses variables de classe ! Cette méthode a pour rôle de signaler à la JVM qu'il faut réserver de la mémoire pour notre futur objet et donc, par extension, d'en réserver pour toutes ses variables.

Un objet peut avoir **plusieurs constructeurs**. Il s'agit de la même méthode, mais **surchargée** ! Pour surcharger une méthode, il faut conserver son nom et son nom de retour, mais changer le nombre ou le type des paramètres.

On **instancie un nouvel objet** grâce au mot clé **new**.

- L'**ordre des paramètres passés dans le constructeur doit être respecté**.

```
// initialisation sans paramètres
Ville ville = new Ville();

// l'ordre est respecté -> aucun souci
Ville ville1 = new Ville("Marseille", 123456789, "France");

// erreur dans l'ordre des paramètres -> erreur de compilation au final
Ville ville2 = new Ville(12456, "France", "Lille");
```

- Il est recommandé de **déclarer ses variables d'instance private**, pour les protéger d'une mauvaise utilisation par le programmeur.

```
public class Ville
{
    private String nomVille;
    private String nomPays;
    private int nbreHabitants;
}
```

La **portée** détermine qui peut faire appel à une classe, une méthode ou une variable.

Un attribut déclaré **public** peut être appelé depuis n'importe quel endroit du programme. Un attribut déclaré **private** ne pourra être appelée que depuis l'intérieur de la classe où il se trouve. Un attribut déclaré **protected** pourra être appelée depuis l'intérieur de la classe où il se trouve, et depuis l'intérieur de ses classes héritées.

Si un attribut n'a pas de portée associée, on dit qu'il a une **portée par défaut**, ce qui signifie qu'il sera visible dans toutes les classes du package courant.

- On crée des **accesseurs et mutateurs** (méthodes **getters** et **setters**) pour permettre une modification sûre des variables d'instance.

Un accesseur est une méthode qui va nous permettre d'accéder aux variables de nos objets en lecture, et un mutateur nous permettra d'en faire de même en écriture. Grâce aux accesseurs, vous pourrez afficher les variables des objets, et grâce aux mutateurs, vous pourrez les modifier.

```
public class Ville
{
    //***** ACCESSEURS *****
    public String getNom()           // retourne le nom de la ville
    {
        return nomVille;
    }

    public String getNomPays()        // retourne le nom du pays
    {
        return nomPays;
    }

    public int getNombreHabitants()   // retourne le nombre d'habitants
    {
        return nbreHabitants;
    }
}
```

```

//***** MUTATEURS *****
public void setNom(String pNom)           // définit le nom de la ville
{
    nomVille = pNom;
}

public void setNomPays(String pPays)        // définit le nom du pays
{
    nomPays = pPays;
}

public void setNombreHabitants(int nbre)    // définit le nombre d'habitants
{
    nbreHabitants = nbre;
}

```

Ces méthodes sont publiques. Les accesseurs sont du même type que la variable qu'ils doivent retourner. Les mutateurs sont de type void (ne retournent aucune valeur).

**Convention de nommage :** les accesseurs commencent par **get** et les mutateurs par **set**.

- Dans une classe, on accède aux variables de celle-ci grâce au mot clé **this**.

```

public String comparer(Ville v1)
{
    String str = new String();
    if ( v1.getNombreHabitants() > this.nbreHabitants )
        str = v1.getNom() + " est une ville plus peuplée que " + this.nomVille;
    else
        str = this.nomVille + " est une ville plus peuplée que " + v1.getNom();
    return str;
}

```

Le mot clé **this** fait référence à l'objet courant.

Dans l'exemple :

```

Ville v1 = new Ville("Lyon", 654, "France");
Ville v2 = new Ville("Lille", 123, "France");
v1.comparer(v2);

```

- pour accéder à la variable *nbreHabitants* de l'**objet v2** il suffit d'utiliser la syntaxe **v2.getNombreHabitants()** (référence à l'attribut *nbreHabitants* de l'objet v2)
  - mais l'**objet v1**, lui, est l'appelant de cette méthode. pour se servir de ses propres variables on utilise alors **this.nbreHabitants**, ce qui a pour effet de faire appel à la variable *nbreHabitants* de l'objet exécutant la méthode **comparer(Ville v)**
- Une **variable de classe** est une variable devant être déclarée **static**. Les **méthodes** n'utilisant que des variables de classe doivent elles aussi être déclarées **static**.

Dans cet exemple, nous allons compter le nombre d'instances de la classe Ville, en utilisant une variable de classe déclarée publique et une variable de classe déclarée privée. L'accesseur de la variable de classe déclarée privée est aussi déclaré static : ceci est une règle !

```

public class Ville {
    public static int nbreInstances = 0;           // variable publique qui comptera les instances
    private static int nbreInstancesBis = 0; // variable privée qui comptera les instances

    public Ville(){           // on incrémente nos variables à chaque appel aux constructeurs
        nbreInstances++;
        nbreInstancesBis++;
    }

    public Ville(String pNom, int pNbre, String pPays)
    {
        nbreInstances++;
        nbreInstancesBis++;
    }

    public static int getNombreInstancesBis() // getter pour la variable de classe privée
    {
        return nbreInstancesBis;
    }
}

Ville v1 = new Ville();
System.out.println("Le nombre d'instances classe Ville est : " + Ville.nbreInstances);           // 1
System.out.println("Le nombre d'instances classe Ville est : " + Ville.getNombreInstancesBis()); // 1

Ville v2 = new Ville("Marseille", 1236, "France");
System.out.println("Le nombre d'instances classe Ville est : " + Ville.nbreInstances);           // 2
System.out.println("Le nombre d'instances classe Ville est : " + Ville.getNombreInstancesBis()); // 2

Ville v3 = new Ville("Rio", 321654, "Brésil");
System.out.println("Le nombre d'instances classe Ville est : " + Ville.nbreInstances);           // 3
System.out.println("Le nombre d'instances classe Ville est : " + Ville.getNombreInstancesBis()); // 3

```

- Le principe d'**encapsulation**

Nous avons construit notre premier objet « Ville ». Cependant, nous avons fait plus que ça : nous avons créé un objet dont les **variables** sont **protégées de l'extérieur**. En effet, depuis l'extérieur de la classe, elles ne sont accessibles que via les accesseurs et mutateurs que nous avons défini. C'est le principe d'encapsulation !

Lorsqu'on procède de la sorte, on s'assure que le **fonctionnement interne à l'objet est intégré**, car toute modification d'une donnée de l'objet est maîtrisée. Nous avons développé des méthodes qui s'assurent qu'on ne modifie pas n'importe comment les variables.

## L'héritage

- La notion d'héritage est l'un des fondements de la programmation orientée objet. Grâce à elle, nous pourrons créer des classes héritées (aussi appelées des classes dérivées) de nos classes mères (aussi appelées des classes de base). Nous pourrons créer autant de classes dérivées que nous le souhaitons. De plus, nous pourrons nous servir d'une classe dérivée comme d'une classe de base pour élaborer encore une autre classe dérivée.

La classe fille hérite de toutes les propriétés et méthodes public et protected de la classe mère. Les méthodes et les propriétés private d'une classe mère ne sont pas accessibles dans la classe fille.

On peut redéfinir une méthode héritée, c'est-à-dire qu'on peut changer tout son code.

Limitation Java : une classe ne peut hériter que d'une seule classe. Lors d'un héritage multiple, la JVM ne saura pas quelle méthode utiliser; alors, plutôt que de forcer le programmeur à gérer les cas d'erreur, les concepteurs du langage ont préféré de l'interdire.

- Une classe hérite d'une autre classe par le biais du mot clé extends.

```
class Capitale extends Ville {}
```

- Si aucun constructeur n'est défini dans une classe fille, la JVM en créera un et appellera automatiquement le constructeur de la classe mère.

En revanche, dès que vous avez créé un constructeur, n'importe lequel, la JVM ne crée plus le constructeur par défaut.

```
public class Capitale extends Ville
{
    public Capitale()           // constructeur propre à la classe dérivée
    {
        this.nomVille = "toto"; // ce code s'exécute seulement si cet attribut n'est pas
                               // déclaré privé dans la classe mère
    }
}
```

- On peut utiliser le comportement d'une classe mère par le biais du mot clé super.

```
class Capitale extends Ville
{
    private String monument;

    public Capitale()           // constructeur par défaut
    {
        super();                // ce mot clé appelle le constructeur de la classe mère
        monument = "aucun";
    }

    public Capitale(String nom, int hab, String pays, String monument){
        super(nom, hab, pays);
        this.monument = monument;
    }
}
```

- Grâce à l'héritage et au **polymorphisme**, nous pouvons utiliser la covariance des variables.
- Le concept de polymorphisme complète parfaitement celui de l'héritage. Il permet de **manipuler des objets sans vraiment connaître leur type**.

Exemple :

Nous créons un tableau de villes contenant des villes et des capitales (nous avons le droit de faire ça, car les objets Capitale sont aussi des objets Ville).

Ensuite on affiche leur contenu avec la méthode `toString()` qui fait bien son travail ! Elle appelle la méthode `toString()` appartenant à la classe Ville pour les objets de type Ville, et elle appelle la méthode `toString()` appartenant à la classe Capitale pour les objets de type Capitale.

**Attention** à ne pas confondre la surcharge avec une méthode polymorphe :

- une méthode surchargée diffère de la méthode originale par le nombre ou le type des paramètres qu'elle prend en entrée.
- **une méthode polymorphe a un squelette identique à la méthode de base**, mais traite les choses différemment. Cette méthode se trouve dans une autre classe et donc, par extension, dans une autre instance de cette autre classe.

```
public class Ville
{
    public String toString()
    {
        return "\t"+this.nomVille+" est une ville de "+this.nomPays+",\n        elle comporte : "+this.nbreHabitant+" => catégorie : "+this.categorie;
    }
}

public class Capitale extends Ville
{
    public String toString()
    {
        return super.toString() + "\n \t ==> monument" + this.monument;
    }
}

// main
Ville[] tableau = new Ville[4]; // définition d'un tableau de villes null

String[] tab = {"Marseille", "lille", "paris", "nantes"};
int[] tab2 = {123456, 78456, 654987, 75832165};

for(int i = 0; i < 4; i++){
    if (i < 2 )
        { Ville v1 = new Ville(tab[i], tab2[i], "france"); tableau[i] = v1; }
    else
        { Capitale C = new Capitale(tab[i], tab2[i], "france", "Eiffel"); tableau[i] = C; }
}

for ( Object obj : tableau )
    { System.out.println(obj.toString()+"\n"); }
```

### Précisions :

- Toutes les classes en Java héritent automatiquement de la classe *Object*.
- Si vous ne **redéfinissez pas** ou ne « **polymorphez** » pas la méthode d'une classe mère dans une classe fille (exemple de `toString()`), à l'appel de celle-ci avec un objet fille, c'est la méthode de la classe mère qui sera invoquée (JVM remontera dans l'hierarchie d'héritage jusqu'à arriver à la classe *Object*).
- `System.out.println(v);` appellera automatiquement `System.out.println(v.toString());`
- Il existe deux autres méthodes qui sont très souvent redéfinies (à coté de la méthode `toString()`)
  - `public boolean equals(Object o)` : qui permet de vérifier si un objet est égal à un autre
  - `public int hashCode()` : qui attribue un code de hashage à un objet (un identifiant)Eclipse vous permet de générer automatiquement ces deux méthodes, via le menu  
**Source/Generate hashCode and equals**
- Vous ne pouvez pas hériter d'une classe déclarée final.
- Il existe encore des **méthodes** de type **final**. Une méthode déclarée final n'est pas redéfinissable

```
public final int maMethode(){  
    // méthode ne pouvant pas être surchargée  
}
```
- Il existe aussi des **classes** déclarées **final**. Ces classes sont immuables, c'est à dire que l'on pourra pas hériter un objet d'une classe déclarée final.

```
public final class FinalClass {  
    // classe ne pouvant pas être héritée  
}
```

## Modéliser des objets grâce à UML

- **UML** vous permet de représenter les liens entre vos classes.

Le sigle UML signifie **Unified Modeling Language**, que l'on peut traduire par « langage de modélisation unifié ». Il ne s'agit pas d'un langage de programmation, mais plutôt d'une **méthode de modélisation**.

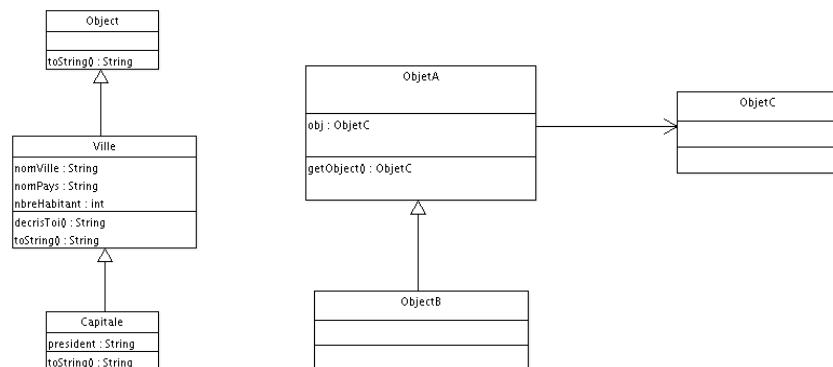
Lorsque vous programmez en orienté objet, il vous sera sans doute utile de pouvoir schématiser vos classes, leur hiérarchie, leurs dépendances, leur architecture, etc. L'idée est de pouvoir, d'un simple coup d'œil, vous représenter le fonctionnement de votre logiciel.

Avec UML, vous pouvez modéliser toutes les étapes du développement d'une application informatique, de sa conception à la mise en route, **grâce à des diagrammes**.

Avec ces outils, vous pouvez réaliser les différents diagrammes qu'UML vous propose :

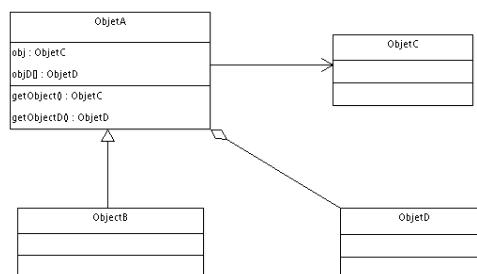
- o le **diagramme de use case** (cas d'utilisation) permet de déterminer les différents cas d'utilisation d'un programme informatique ;
- o le **diagramme de classes** permet de modéliser des classes (leurs attributs et leurs méthodes) ainsi que les interactions entre elles ;
- o les **diagrammes de séquences** permettent de visualiser le déroulement d'une application dans un contexte donné ;

- Exemple de diagrammes de classes :



Représenter l'héritage avec une flèche signifiant « est un » : Ville → Objet; capitale → ville.

Représenter l'appartenance avec une flèche signifiant « a un » : ObjetA → ObjetC



Représenter la composition avec une flèche signifiant « est composé de plusieurs » :

ObjetA ← ObjetD

## Les packages

- Un **package** est un ensemble de dossiers et de sous-dossiers, permettant de ranger nos classes. Charger un package nous permet d'utiliser les classes qu'il contient.  
L'un des avantages des packages est que nous allons y **gagner en lisibilité** dans notre package par défaut, mais aussi que les classes mises dans un package sont **plus facilement transportables** d'une application à l'autre. Pour cela, il vous suffit d'inclure le dossier de votre package dans un projet et d'y importer les classes qui vous intéressent !
- Le nom du package est soumis à une **convention de nommage** :
  - ceux-ci doivent être écrits entièrement en **minuscules** (de a à z, de 0 à 9) et peuvent contenir des points (.)
  - tout package **doit commencer par** *com, edu, gov, mil, net, org* ou les deux lettres identifiant un pays(*fr, en, ...*)
  - aucun **mot clé** Java ne doit être présent dans le nom, sauf s'il est suivi par un underscore (ex : *com.sdz.package\_*)
- Si vous voulez utiliser un mot clé Java dans le nom de votre package, vous devez le faire suivre d'un underscore (« \_ »).
- Les classes déclarées **public** sont **visibles depuis l'extérieur du package** qui les contient.
- Les classes n'ayant pas été déclarées public ne sont pas visibles depuis l'extérieur du package qui les contient.
- Si une classe déclarée public dans son package a une variable d'un type ayant une portée default, cette dernière ne pourra pas être modifiée depuis l'extérieur de son package.

## Les classes abstraites et les interfaces

- Derrière ces deux notions se cache la manière dont Java vous permet de structurer votre programme.

Vos programmes Java regorgeront de classes, avec de l'héritage, des dépendances, de la composition... Afin de bien structurer vos programmes (on parle d'**architecture logicielle**), vous allez vous creuser les méninges pour savoir où ranger des comportements d'objets (dans la classe mère ?, dans la classe fille ?, ... ).

Une **classe abstraite** est une classe qui **ne peut pas être instancié** ! Ces classes servent à définir une *superclasse* : elles servent essentiellement à créer un nouveau type d'objets (un modèle pour des classes filles).

Les classes abstraites sont à utiliser lorsqu'une classe mère ne doit pas être instanciée.

Une classe est définie comme abstraite avec le mot clé **abstract**.

```
abstract class Animal {}
```

Une telle classe peut contenir la même chose qu'une classe normale. Ses enfants pourront utiliser tous ses éléments déclarés (attributs et méthodes déclarés public ou protected).

Cependant, ce type de classe permet de définir des **méthodes abstraites** qui présentent une particularité : elle **n'ont pas de corps** !

En voici un exemple :

```
abstract class Animal
{
    abstract void manger(); // une méthode abstraite
}
```

Pourquoi on dit « méthode abstraite » : difficile de voir ce que cette méthode sait faire.

**Une méthode abstraite ne peut exister que dans une classe abstraite.** Si, dans une classe il y a une méthode déclarée abstraite, vous devez déclarer cette classe comme étant abstraite.

Mais, **une classe abstraite n'est pas obligée de contenir de méthode abstraite**.

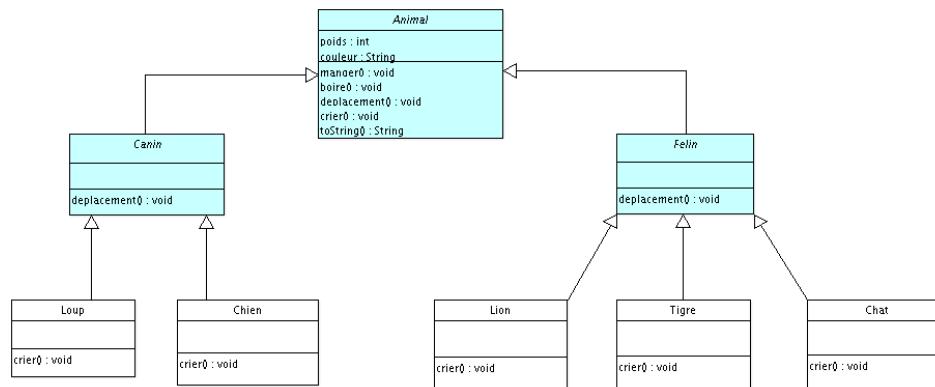
Exemple d'utilisation : une superclasse Animal de qui héritent les classes Loup, Chien, Lion, ...

```
public class Test
{
    public static void main(String args[])
    {
        Animal loup = new Loup(); // classe fille de la classe Animal
        Animal chien = new Chien(); // classe fille de la classe Animal
        loup.manger();
        chien.crier();
    }
}
```

Nous avons pas instancié notre classe abstraite; nous avons instancié un objet Loup que nous avons mis dans un objet de type Animal (de même pour l'instanciation de la classe Chien).

L'instance se crée seulement avec le mot clé **new** !

Un exemple complet :



```
abstract class Animal
{
    protected String couleur;
    protected int poids;

    abstract void déplacement();
    abstract void crier();

    protected void manger()
    {
        System.out.println("Je mange de la viande.");
    }

    protected void boire()
    {
        System.out.println("Je bois de l'eau.");
    }

    public String toString()
    {
        return "Je suis un objet de la classe " + this.getClass() + ",\n        je suis " + this.couleur + ", je pèse " + this.poids;
    }
}

public abstract class Canin extends Animal
{
    void déplacement() // définition la méthode abstraite
    {
        System.out.println("Je me déplace en meute.");
    }
}

public class Loup extends Canin
{
    public Loup() {} // constructeur par défaut

    public Loup(String couleur, int poids) // constructeur avec des paramètres
    {
        this.couleur = couleur;
        this.poids = poids;
    }

    void crier() // définition la méthode abstraite
    {
        System.out.println("Je hurle à la Lune en faisant ouhouh !");
    }
}
```

```
// utilisation des classes
public class Test
{
    public static void main(String[] args)
    {
        Loup l = new Loup("Gris bleuté", 20);
        l.boire();                                // Je bois de l'eau.
        l.manger();                               // Je mange de la viande.
        l.deplacement();                         // Je me déplace en meute.
        l.crier();                                // Je hurle à la Lune en faisant ouhou !
        System.out.println(l.toString());          // Je suis un objet de la class Loup,
  // je suis Gris bleuté, je pèse 20
    }
}
```

Dans cet exemple nous avons un objet Loup :

- à l'appel de la méthode **boire()** : l'objet appelle la méthode de la classe **Animal**.
  - à l'appel de la méthode **manger()** : l'objet appelle la méthode de la classe **Animal**.
  - à l'appel de la méthode **toString()** : l'objet appelle la méthode de la classe **Animal**.
  - à l'appel de la méthode **deplacement()** : la méthode de la classe **Canin** est invoquée
  - à l'appel de la méthode **crier()** : la méthode de la classe **Loup** est appelée.
- Une **interface** est une classe **100 % abstraite**. Elle permet de définir un nouveau supertype et à utiliser le polymorphisme. On peut même en ajouter autant que l'on veut dans une seule classe.  
Aucune méthode d'une interface n'a de corps. Les méthodes d'interfaces sont publiques et abstraites (sans spécifier le mot clé **abstract**).  
Une interface s'implémente dans une classe en utilisant le mot clé **implements**.  
Vous devez redéfinir toutes les méthodes de l'interface (ou des interfaces) dans votre classe.

Exemple :

```
public interface I1
{
    public void A();
    public String B();
}

public class X implements I1, I2
{
    public void A()
    {
        ...
    }

    public String B()
    {
        ...
    }
}

public interface I2
{
    public void C();
    public String D();
}
```

Exemple complet pour créer une nouvelle classe Chien, qui lui permettra de faire le beau, faire léchouille, faire des câlins :

```
public interface Rintintin
{
    public void faireCalin();
    public void faireLechouille();
    public void faireLeBeau();
}

public class Chien extends Canin implements Rintintin
{
    public Chien() {}

    public Chien(String couleur, int poids)
    {
        this.couleur = couleur;
        this.poids = poids;
    }

    void crier()
    {
        System.out.println("J'aboie sans raison !");
    }

    public void faireCalin()
    {
        System.out.println("Je te fais un GROS CÂLIN");
    }

    public void faireLeBeau()
    {
        System.out.println("Je fais le beau !");
    }

    public void faireLechouille()
    {
        System.out.println("Je fais de grosses léchouilles...");
    }
}
```

**Attention** : l'ordre des déclarations est primordial. Vous devez mettre l'expression d'héritage avant l'expression d'implémentation, sinon votre code ne compilera pas.

```
public class Test
{
    public static void main(String[] args)
    {
        // les méthodes d'un chien
        Chien c = new Chien("Gris bleuté", 20);
```

```

        c.boire();
        c.manger();
        c.deplacement();
        c.crier();
        System.out.println(c.toString());

        System.out.println("-----");
        // les méthodes de l'interface
        c.faireCalin();
        c.faireLeBeau();
        c.faireLechouille();

        System.out.println("-----");
        // utilisons le polymorphisme de notre interface
        Rintintin r = new Chien();
        r.faireLeBeau();
        r.faireCalin();
        r.faireLechouille();
    }
}

```

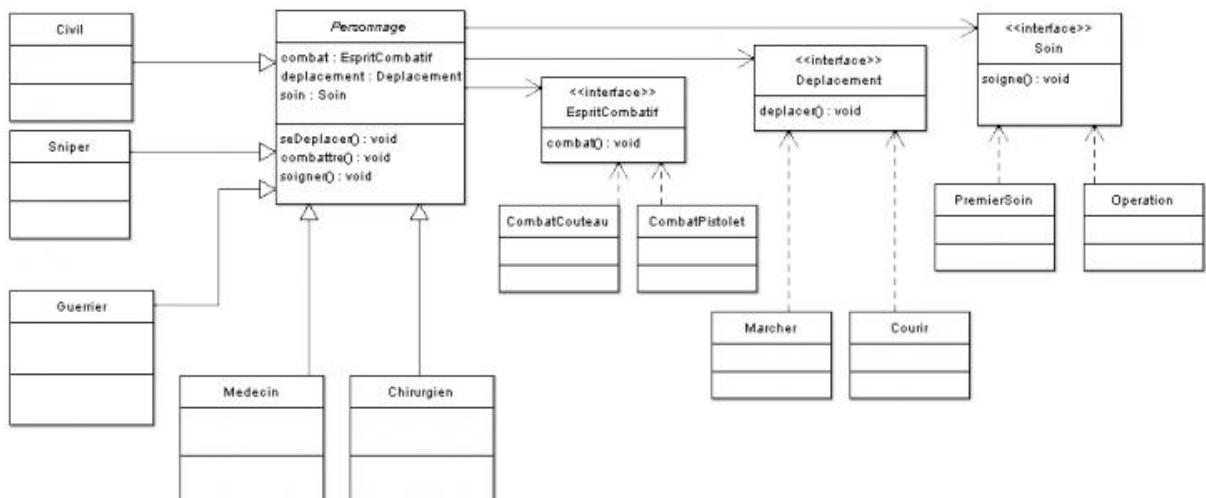
- Le **pattern strategy** (technique de programmation) vous permet de rendre une hiérarchie de classes plus souple.

Un **design pattern** est un patron de conception, une façon de construire une hiérarchie des classes permettant de répondre à un problème.

Nous aborderons le **pattern strategy**, qui va nous permettre de remédier à la limite de l'héritage. En effet, même si l'héritage offre beaucoup de possibilités, il a ses limites.

Le principe de base de ce pattern est le suivant : « **isolez ce qui varie dans votre programme et encapsulez-le !** ». Préférez encapsuler des comportements plutôt que de les mettre d'office dans l'objet concerné.

**Exemple :** un jeu de combat dont le comportement des personnages n'arête pas de changer



### La classe Personnage a

- comme attributs :
  - un objet de type **EspritCombatif** (interface)
    - ❖ comportements possibles :  
classes fille **Pacifiste**, **CombatPistolet**, **CombatCouteau**
  - un objet de type **Deplacement** (interface)
    - ❖ comportements possibles :  
classes fille **Marcher**, **Courir**
  - un objet de type **Soin** (interface)
    - ❖ comportements possibles :  
classes fille **PremierSoin**, **Operation**, **AucunSoin**
- comme méthodes :
  - seDeplacer()
  - combattre()
  - soigner()

### **Organisation du code :**

- package <com.sdz.comportement>
  - **AucunSoin.java**
  - **CombatCouteau.java**
  - **CombatPistolet.java**
  - **Courrir.java**
  - **Deplacement.java**
  - **EspritCombatif.java**
  - **Marcher.java**
  - **Operation.java**
  - **Pacifiste.java**
  - **PremierSoin.java**
  - **Soin.java**
- package
  - **Personnage.java** (abstraite)
  - **Civil.java** : classe fille de la classe Personnage
  - **Sniper.java** : classe fille de la classe Personnage
  - **Guerrier.java** : classe fille de la classe Personnage
  - **Medecin** : classe fille de la classe Personnage
  - **Chirurgien** : classe fille de la classe Personnage

### Code :

#### Les interfaces :

```
package com.sdz.comportement;

public interface EspritCombatif
{
    public void combat();
}
```

```
package com.sdz.comportement;

public interface Soin
{
    public void soigne();
}
```

```
package com.sdz.comportement;

public interface Deplacement
{
    public void deplacer();
}
```

#### Les classes de comportement :

```
package com.sdz.comportement;

public class Pacifiste implements EspritCombatif
{
    public void combat()
        { System.out.println("Je ne combats pas !"); }
}
```

```
package com.sdz.comportement;

public class CombatPistolet implements EspritCombatif
{
    public void combat()
        { System.out.println("Je combats au pitolet !"); }
}
```

```
package com.sdz.comportement;

public class CombatCouteau implements EspritCombatif
{
    public void combat()
        { System.out.println("Je me bats au couteau !"); }
}
```

```
package com.sdz.comportement;

public class Marcher implements Deplacement
{
    public void deplacer()
        { System.out.println("Je me déplace en marchant."); }
}
```

```
package com.sdz.comportement;

public class Courir implements Deplacement
{
    public void deplacer()
        { System.out.println("Je me déplace en courant."); }
}
```

```
package com.sdz.comportement;

public class PremierSoin implements Soin
{
    public void soigne()
        { System.out.println("Je donne les premiers soins."); }
}
```

```
package com.sdz.comportement;

public class Operation implements Soin
{
    public void soigne()
        { System.out.println("Je pratique des opérations !"); }
}
```

```
package com.sdz.comportement;

public class AucunSoin implements Soin
{
    public void soigne()
        { System.out.println("Je ne donne AUCUN soin !"); }
}
```

La superclasse **Personnage** (abstraite)

```
import com.sdz.comportement.*;

public abstract class Personnage
{
    // les instances de comportement
    protected EspritCombatif espritCombatif = new Pacifiste();
    protected Soin soin = new AucunSoin();
    protected Deplacement deplacement = new Marcher();

    // constructeurs
    public Personnage(){}
    public Personnage(EspritCombatif ec, Soin soin, Deplacement dep)
    {
        this.espritCombatif = ec;
        this.soin = soin;
        this.deplacement = dep;
    }

    //***** SETTERS *****

    public void setEspritCombatif(EspritCombatif ec) //redéfinit le comportement combat
    { this.espritCombatif = ec; }

    public void setSoin(Soin soin) // redéfinit le comportement de soin
    { this.soin = soin; }

    public void setDeplacement(Deplacement dep) // redéfinit le comportement déplacement
    { this.deplacement = dep; }

    //***** ACTIONS *****

    public void seDeplacer() // méthode de déplacement de personnage
    { deplacement.deplacer(); } //on utilise les objets de déplacement de façon polymorphe

    public void combattre() // méthode que les combattants utilisent
    { espritCombatif.combat(); } // on utilise les objets de combat de façon polymorphe

    public void soigner() // méthode de soin
    { soin.soigne(); } // on utilise les objets de soin de façon polymorphe
}
```

### **Les classes filles**

```
import com.sdz.comportement.*;

public class Guerrier extends Personnage
{
    public Guerrier()
        { this.espritCombatif = new CombatPistolet(); }

    public Guerrier(EspritCombatif esprit, Soin soin, Deplacement dep)
        { super(esprit, soin, dep); }
}
```

```
import com.sdz.comportement.*;

public class Civil extends Personnage
{
    public Civil() {}

    public Civil(EspritCombatif esprit, Soin soin, Deplacement dep)
        { super(esprit, soin, dep); }
}
```

```
import com.sdz.comportement.*;

public class Medecin extends Personnage
{
    public Medecin()
        { this.soin = new PremierSoin(); }

    public Medecin(EspritCombatif esprit, Soin soin, Deplacement dep)
        { super(esprit, soin, dep); }
}
```

### **Exemple d'utilisation**

```
class Test
{
    public static void main(String[] args)
    {
        Personnage[] tPers = {new Guerrier(), new Civil(), new Medecin()};

        for (int i = 0; i < tPers.length; i++)
        {
            System.out.println("\nInstance de " + tPers[i].getClass().getName());
            tPers[i].combattre();
            tPers[i].seDeplacer();
            tPers[i].soigner();
        }
    }
}
```

Résultat :

```
Problems @ Javadoc Declaration Console X
<terminated> Test (1) [Java Application] C:\Program Files\Java
Instance de Guerrier
*****
Je combats au pitolet !
Je me déplace en marchant.
Je ne donne AUCUN soin !

Instance de Civil
*****
Je ne combats pas !
Je me déplace en marchant.
Je ne donne AUCUN soin !

Instance de Medecin
*****
Je ne combats pas !
Je me déplace en marchant.
Je donne les premiers soins.
```

Exemple de changement nécessaire :

S'il faut changer dynamiquement le comportement de Guerrier en lui donnant la possibilité d'effectuer des petites opérations chirurgicales, on peut tout simplement changer le code de test :

```
import com.sdz.comportement.*;

class Test
{
    public static void main(String[] args)
    {
        Personnage pers = new Guerrier();
        pers.soigner();
        pers.setSoin(new Operation());
        pers.soigner();
    }
}
```

Nul besoin de modifier l'architecture ou le code des classes !

## Les exceptions

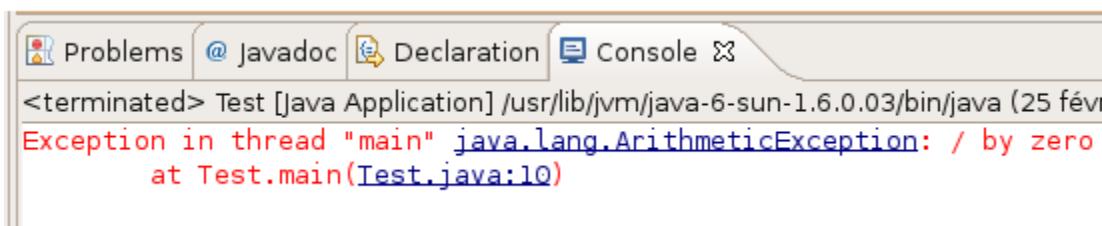
- Lorsqu'un événement que la JVM ne sait pas gérer apparaît, une exception est levée (exemple : division par zéro). **Une exception correspond donc à une erreur.**

Lorsqu'un programme rencontre une exception, l'exécution de celui-ci est interrompue !

Le fait de gérer les exceptions s'appelle aussi "la capture d'exception". Le principe consiste :

- à repérer un morceau de code qui pourrait générer une exception,
- de capturer l'exception correspondante
- et enfin de la traiter, c'est-à-dire d'afficher un message personnalisé et de continuer l'exécution.

Exemple d'exception : la division par zéro (une exception de type *ArithmeticException*)



A screenshot of an IDE interface showing the 'Console' tab. The output window displays the following text:  
<terminated> Test [Java Application] /usr/lib/jvm/java-6-sun-1.6.0.03/bin/java (25 février 2013)  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at Test.main(Test.java:10)

L'instruction qui permet de capturer des exceptions est le bloc `try {...} catch {}`.

Lorsqu'une ligne de code lève une exception, l'instruction dans le bloc try est interrompue et le programme se rend dans le bloc catch correspondant à l'exception levée.

```
public static void main(String[] args)
{
    int j = 20, i = 0;

    try
    {
        System.out.println(j/i); // code susceptible de lever une exception
    }
    catch (ArithmeticException ex)
    {
        // code exécuté lorsqu'une exception de type ArithmeticException est levé
        System.out.println("Division par zéro ! Erreur : " + ex.getMessage());
    }

    System.out.println("coucou toi !");
}
```

Le résultat visible à l'écran :

```
Division par zéro ! Erreur : /by zero
coucou toi !
```

Le paramètre de la clause catch permet de connaître le type d'exception qui doit être capturé. Et l'objet (ici: ex) peut servir à préciser notre message grâce à l'appel de la méthode `getMessage()`.

**Lorsque nous capturons une exception**, le code présent dans le bloc `catch(){...}` est exécuté, mais le **programme suit son cours** !

Si une exception est levée dans le bloc try, les instructions figurant dans le bloc catch seront exécutées pour autant que celui-ci capture la bonne exception levée.

La clause **finally** associée au bloc try {} catch {} :

```
try
{
    System.out.println(" => " + (1/0));
}
catch (ClassCastException e)
{
    e.printStackTrace();
}
finally
{
    // code exécuté quoi qu'il arrive (s'il y a une exception qui est levé ou non)
    System.out.println("action faite systématiquement");
}
```

Cela est surtout utilisé lorsque vous devez vous assurer d'avoir fermé un fichier, clos votre connexion à une base de données ou un socket (une connexion réseau).

- La superclasse qui gère les exceptions s'appelle **Exception**.  
Dans cette classe sont répertoriés tous les cas d'erreur.
- Vous pouvez créer une classe d'exception personnalisée : faites-lui hériter de la classe **Exception**.

La procédure :

- créer une classe héritant de la classe **Exception** : ex *NombreHabitantsException*
- renvoyer l'exception levée à notre classe
- gérer celle-ci dans notre classe

Pour faire tout cela, il faut connaître les mots clés :

- **throws** : ce mot clé permet de signaler à la JVM qu'un morceau de code, une méthode, une classe ... est potentiellement dangereux et qu'il faut utiliser un bloc try{}catch{}. Il est suivi du nom de la classe qui va gérer l'exception
- **throw** : celui-ci permet de lever une exception manuellement en instanciant un objet type **Exception** (ou un objet hérité). Exemple :

```
class NombreHabitantException extends Exception
{
    public NombreHabitantException()                      // constructeur par défaut
    {
        System.out.println("Vous essayez d'instancier une classe Ville
                            avec un nombre d'habitants négatif !");
    }
    public NombreHabitantException(int nb)                // constructeur avec un paramètre
    {
        System.out.println("Instanciation avec un nombre d'habitants négatif !");
        System.out.println("\t => " + nb);
    }
}
```

Utiliser cette exception dans le constructeur de la classe Ville pour signaler la présence d'une erreur lorsqu'on essaye d'associer un nombre négatif à l'attribut *nombreHabitants*, et ainsi refuser l'instanciation de l'objet :

```
public Ville(String nom, int nb, String pays) throws NombreHabitantException
{
    if (nb < 0)
        throw new NombreHabitantException(nb);
    else
    {
        nomVille = nom;
        nomPays = pays;
        nbreHabitant = nb;
        this.setCategorie();

        nbreInstance++;
        nbreInstanceBis++;
    }
}
```

Une instanciation lancée par le biais de l'instruction **throw** doit être déclarée avec **throws** avant !

**throws NombreHabitantException** nous indique que si une erreur est capturée, celle-ci sera traitée en tant qu'objet de la classe NombreHabitantException, ce qui nous renseigne sur le type de l'erreur en question. Elle indique aussi à la JVM que le constructeur de notre objet Ville est potentiellement dangereux et qu'il faudra gérer les exceptions possibles.

Si la condition **if(nbre < 0)** est remplie, **throw new NombreHabitantException();** instancie la classe NombreHabitantException. Par conséquent, si un nombre d'habitants est négatif, l'exception est levée.

Vu les changements dans le constructeur, il vous faudra gérer les exceptions qui pourraient survenir dans l'instruction d'instanciation d'une nouvelle classe Ville avec un bloc **try{...}catch{}**.

```
Ville v = null; // objet créé à l'extérieur du bloc try{}catch{}

try
{
    v = new Ville("Rennes", 12000, "France");
}
catch (NombreHabitantException e)
{}
finally
{
    if ( v == null ) // éviter d'utiliser un objet null lorsque l'exception est levé
        v = new Ville();
}

System.out.println(v.toString());
```

- Vous pouvez définir plusieurs risques d'exceptions sur une même méthode. Il suffit de séparer les déclarations par une virgule. Vous pouvez ajouter autant de blocs catch que vous le voulez à la suite d'un bloc try, mais respectez l'ordre : du plus pertinent au moins pertinent.

Exemple :

```
public class NomVilleException extends Exception
{
    public NomVilleException(String message)
        { super(message); }
}

public Ville(String nom, int nb, String pays) throws
  NombreHabitantException, NomVilleException
{
    if(nb < 0)
        throw new NombreHabitantException(nb);

    if(nom.length() < 3)
        throw new NomVilleException("nom ville < 3 caractères ! nom = " + nom);
    else
    {
        nomVille = nom;
        nomPays = pays;
        nbreHabitant = nb;
        this.setCategorie();

        nbreInstance++;
        nbreInstanceBis++;
    }
}

// main
Ville v = null;

try
{ v = new Ville("Re", 12000, "France"); }
catch (NombreHabitantException e1)           // gestion de l'exception sur le nombre d'habitants
{ e1.printStackTrace(); }
catch (NomVilleException e2)                 // gestion de l'exception sur le nom de la ville
{ System.out.println(e2.getMessage()); }
finally
{
    if(v == null)
        v = new Ville();
}

System.out.println(v.toString());
```

- Le **multi-catch** depuis Java 7

Java 7 permet de catcher plusieurs exceptions dans l'instruction `catch{}`. Ceci se fait grâce à l'opérateur « | » qui permet d'informer la JVM que le bloc de code est susceptible d'engendrer plusieurs types d'exception. C'est vraiment simple à utiliser et cela vous permet d'avoir un code plus compact.

```
public static void main(String[] args)
{
    Ville v = null;

    try
    { v = new Ville("Re", 12000, "France"); }
    catch (NombreHabitantException | NomVilleException ex) // plusieurs exceptions diff
        { System.out.println(ex.getMessage()); }
    finally
    {
        if(v == null)
            v = new Ville();
    }

    System.out.println(v.toString());
}
```

## Les énumérations

- Une **énumération** est une **classe** contenant une **liste de sous-objets**.

Notion nouvelle depuis Java 5. Ce sont des structures qui définissent une liste de valeurs possibles. Cela vous permet de créer des types de données personnalisés.

Une énumération se déclare comme une classe, mais en remplaçant le mot-clé *class* par **enum**. Autre différence : les énumérations héritent de la classe *java.lang.Enum*.

Exemple : une structure de données qui encapsule quatre "objets"

```
public enum Langage
{
    JAVA,
    C,
    CPlus,
    PHP;
}
```

```
// main
for (Langage lang : Langage.values() )
{
    if ( Langage.JAVA.equals(lang) )
        System.out.println("J'aime le : " + lang);
    else
        System.out.println(lang);
}
```

```
// le résultat
J'aime le : JAVA
C
CPlus
PHP
```

- Chaque élément d'une énumération est un objet à part entière.
- Vous pouvez compléter les comportements des objets d'une énumération avec des **méthodes**.

```
public enum Langage
{
    JAVA ("Langage JAVA"),                                // objets directement construits
    C      ("Langage C"),
    CPlus ("Langage C++"),
    PHP   ("Langage PHP");

    private String name = "";

    Langage(String name)                                // constructeur
    {
        this.name = name;
    }

    public String toString()
    {
        return name;
    }
}
```

Pas de déclaration de portée pour le constructeur, pour une raison simple ; il est toujours considéré comme **private** afin de préserver les valeurs définies dans l'enum. Vous noterez par ailleurs que les données formant notre énumération sont directement construites dans la classe.

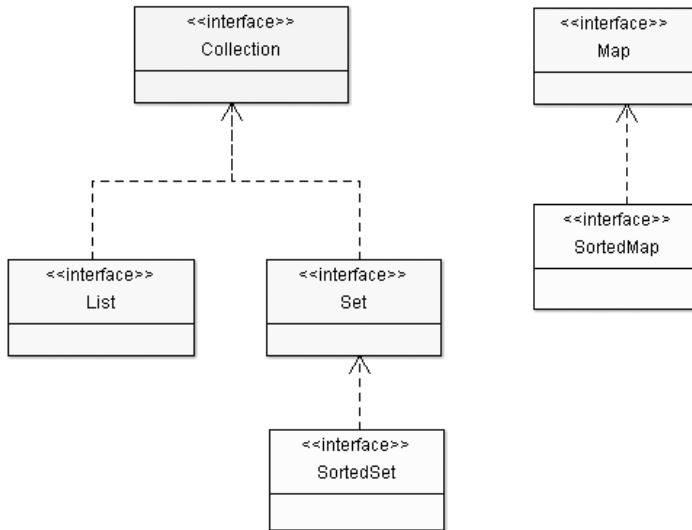
Le résultat du teste précédent :

```
J'aime le : Langage JAVA  
Langage C  
Langage C++  
Langage PHP
```

## Les collections d'objets

- Une **collection** permet de stocker un **nombre variable d'objets** (sans taille prédéfinie).
- Il y a principalement trois types de collection : les **List**, les **Set** et les **Map**, tous stockés dans le package `java.util`. Chaque type a ses avantages et ses inconvénients.

La hiérarchie d'**interfaces** composant ce qu'on appelle les collections :



Les interfaces `List` et `Set` implémentent directement l'interface `Collection`. L'interface `Map` gravite autour de cette hiérarchie, tout en faisant partie des collections Java.

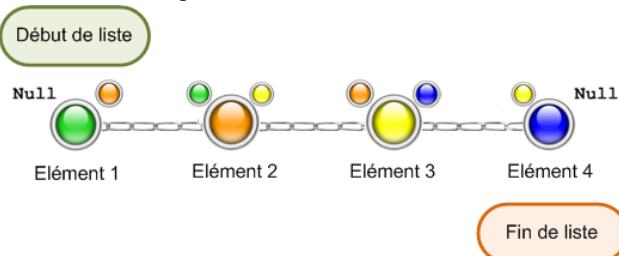
- Les objets de type **List** servent à stocker des objets **sans condition particulière** sur la façon de les stocker. Ils acceptent toutes les valeurs, même les valeurs null.

Les objets appartenant à la catégorie `List` sont, pour simplifier, **des tableaux extensibles** à volonté. On y trouve les objets `Vector`, `LinkedList` et `ArrayList`.

Vous pouvez y insérer autant d'éléments que vous le souhaitez sans craindre de dépasser la taille de votre tableau. Ils fonctionnent tous de la même manière : vous pouvez récupérer les éléments de la liste via leurs **indices**. De plus, les `List` contiennent des **objets**. Je vous propose de voir deux objets de ce type qui, je pense, vous seront très utiles.

### L'objet `LinkedList`

Une liste chaînée (`LinkedList`) 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.



### Fonctionnement de la LinkedList

```
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;

public class Test
{
    public static void main(String[] args)
    {
        List l = new LinkedList();
        l.add(12);
        l.add("toto ! !");
        l.add(12.20f);

        for (int i = 0; i < l.size(); i++)
            System.out.println("Élément à l'index " + i + " = " + l.get(i));
    }
}
```

Ce genre d'objets implémentent l'interface **Iterator**, utile pour lister les **LinkedList**.

Un **itérateur** est un **objet** qui a pour rôle de **parcourir une collection**.

```
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;

public class Test
{
    public static void main(String[] args)
    {
        List l = new LinkedList();
        l.add(12);
        l.add("toto ! !");
        l.add(12.20f);

        for(int i = 0; i < l.size(); i++)
            System.out.println("Élément à l'index " + i + " = " + l.get(i));

        System.out.println("\n \tParcours avec un itérateur ");
        System.out.println("-----");

        ListIterator li = l.listIterator();
        while ( li.hasNext() )
            System.out.println(li.next());
    }
}
```

Les deux manières de procéder sont analogues !

**Attention** : vu que tous les éléments contiennent une référence à l'élément suivant, de telles listes risquent de devenir particulièrement lourdes en grandissant ! Cependant, elles sont **adaptées lorsqu'il faut beaucoup manipuler une collection** en supprimant ou en ajoutant des objets en milieu de liste. Elles sont donc à utiliser avec précaution.

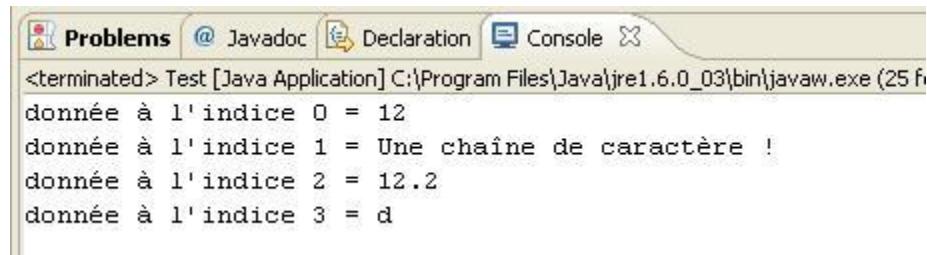
### L'objet ArrayList

Voici un objet bien pratique. 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 ! Nous pouvons mettre tout ce que nous voulons dans un ArrayList, voici un morceau de code qui le prouve :

```
import java.util.ArrayList;

public class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(12);
        al.add("Une chaîne de caractères !");
        al.add(12.20f);
        al.add('d');

        for (int i = 0; i < al.size(); i++)
            System.out.println("donnée à l'indice " + i + " = " + al.get(i));
    }
}
```



Il existe tout un panel de méthodes fournies avec cet objet :

- **add()** permet d'ajouter un élément ;
- **get(int index)** retourne l'élément à l'indice demandé ;
- **remove(int index)** efface l'entrée à l'indice demandé ;
- **isEmpty()** renvoie « vrai » si l'objet est vide ;
- **removeAll()** efface tout le contenu de l'objet ;
- **contains(Object element)** retourne vrai si l'élément passé en paramètre est dans la liste

Contrairement aux LinkedList, les 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 à l'extrême, si vous effectuez beaucoup de lectures sans vous soucier de l'ordre des éléments, optez pour une ArrayList ; en revanche, si vous insérez beaucoup de données au milieu de la liste, optez pour une Linkedlist.

- Un **Set** est une collection un peu plus restrictive, car elle **n'autorise pas deux fois la même valeur** (le même objet), ce qui est pratique pour une liste d'éléments uniques, par exemple.

Par exemple, elle n'accepte qu'une seule fois null, car deux valeurs null sont considérées comme un doublon. On trouve parmi les Set les objets **HashSet**, **TreeSet**, **LinkedHashSet**... Certains Set sont plus restrictifs : il en existe qui n'acceptent pas null, certains types d'objets, ....

Les Set sont particulièrement adaptés pour manipuler une grande quantité de données. Cependant, les performances de ceux-ci peuvent être amoindries en insertion.

Généralement, on opte pour un HashSet, car il est plus performant en temps d'accès, mais si vous avez besoin que votre collection soit **constamment triée**, optez pour un **TreeSet**.

### L'objet HashSet

C'est sans nul doute la plus utilisée des implémentations de l'interface Set. On peut parcourir ce type de collection avec un objet Iterator ou extraire de cet objet un tableau d'Object :

```
import java.util.HashSet;
import java.util.Iterator;

public class Test
{
    public static void main(String[] args)
    {
        HashSet hs = new HashSet();
        hs.add("toto");
        hs.add(12);
        hs.add('d');

        Iterator it = hs.iterator();
        while ( it.hasNext() )
            System.out.println(it.next());

        System.out.println("\nParcours avec un tableau d'objet");
        System.out.println("-----");

        Object[] obj = hs.toArray();
        for ( Object o : obj )
            System.out.println(o);
    }
}
```

La liste des méthodes que l'on trouve dans cet objet :

- add() ajoute un élément ;
- contains(Object value) retourne « vrai » si l'objet contient value ;
- isEmpty() retourne « vrai » si l'objet est vide ;
- iterator() renvoie un objet de type Iterator ;
- remove(Object o) retire l'objet o de la collection ;
- toArray() retourne un tableau d'Object.

- Les **Map** sont des collections qui fonctionnent avec un **système clé - valeur** pour ranger et retrouver les objets qu'elles contiennent.

On y trouve les objets **Hashtable**, **HashMap**, **TreeMap**, **WeakHashMap**...

La clé, qui sert à identifier une entrée dans notre collection, est unique. La valeur, au contraire, peut être associée à plusieurs clés.

Ces objets ont comme point faible majeur leur rapport conflictuel avec la taille des données à stocker. En effet, plus vous aurez de valeurs à mettre dans un objet Map, plus celles-ci seront lentes et lourdes.

### L'objet Hashtable

On parcourt les « tables de hachage » grâce aux clés qu'il contient en recourant à la classe *Enumeration*. L'objet *Enumeration* contient notre *Hashtable* et permet de le parcourir très simplement.

Exemple : le code suivant insère les quatre saisons avec des clés qui ne se suivent pas, et notre énumération récupère seulement les valeurs :

```
import java.util.Enumeration;
import java.util.Hashtable;

public class Test
{
    public static void main(String[] args)
    {
        Hashtable ht = new Hashtable();
        ht.put(1, "printemps");
        ht.put(10, "été");
        ht.put(12, "automne");
        ht.put(45, "hiver");

        Enumeration e = ht.elements();
        while ( e.hasMoreElements() )
            System.out.println(e.nextElement());
    }
}
```

Cet objet offre tout un panel de méthodes utiles :

- isEmpty() retourne « vrai » si l'objet est vide ;
- contains(Object value) retourne « vrai » si la valeur est présente. Identique à containsValue(Object value) ;
- containsKey(Object key) retourne « vrai » si la clé passée en paramètre est présente dans la *Hashtable* ;
- put(Object key, Object value) ajoute le couple key - value dans l'objet ;
- elements() retourne une énumération des éléments de l'objet ;
- keys() retourne la liste des clés sous forme d'énumération.

De plus, il faut savoir qu'un objet Hashtable **n'accepte pas la valeur null** et qu'il est **Thread Safe**, c'est-à-dire qu'il est utilisable dans plusieurs threads simultanément (cela signifie que **plusieurs éléments de votre programme peuvent l'utiliser simultanément**) sans qu'il y ait un risque de conflit de données.

### L'objet HashMap

Cet objet ne diffère que très peu de la Hashtable:

- il accepte la valeur null ;
- il n'est pas Thread Safe.

En fait, les deux objets de type Map sont, à peu de choses près, équivalents.

- **A retenir :**

- Les **Collection** stockent des objets alors que les **Map** stockent un couple clé - valeur.
- Si vous voulez rechercher ou accéder à une valeur via une clé de recherche, optez pour une collection de type **Map**.
- Si vous insérez fréquemment des données en milieu de liste, utilisez une **LinkedList**.
- Si vous avez une grande quantité de données à traiter, tournez-vous vers une liste de type **Set**.

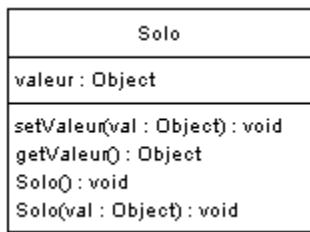
## La généréricité en Java

- La **généricité** est un concept très utile pour développer des objets travaillant avec plusieurs types de données. Son principe est de faire des classes qui n'acceptent qu'un certain type d'objets ou de données de façon **dynamique**.

La généréricité permet de réutiliser sans risque le polymorphisme avec les collections. Cela confère plus de robustesse à votre code.

Vous passerez donc moins de temps à développer des classes traitant de façon identique des données différentes.

**Exemple** de classe **sans généréricité** (attribut Objet, faut faire des *cast* pour changer de type) :



**Exemple** de classe **avec généréricité** (attribut de type T) :

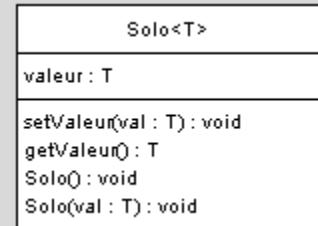
```
public class Solo<T>
{
    private T valeur;           // variable d'instance

    public Solo()               // constructeur par défaut
    { this.valeur = null; }

    public Solo(T val)         // constructeur avec paramètre inconnu pour l'instant
    { this.valeur = val; }

    public void setValeur(T val) // définit la valeur avec le paramètre
    { this.valeur = val; }

    public T getValeur()        // retourne la valeur déjà « castée » par la signature de la méthode
    { return this.valeur; }
}
```



Le type de donnée **T** n'est pas encore défini. Il sera défini lors de l'instanciation de la classe.

Cette classe fonctionne donc avec tous les types de données :

```
public static void main(String[] args)
{
    Solo<Integer> valI = new Solo<Integer>();
    Solo<String> valS = new Solo<String>("TOTOTOTO");
    Solo<Float> valF = new Solo<Float>(12.2f);
    Solo<Double> valD = new Solo<Double>(12.202568);
}
```

Une fois instancié avec un type, l'objet ne pourra travailler qu'avec le type de données spécifié.

```
public static void main(String[] args)
{
    // exemple correct
    Solo<Integer> val = new Solo<Integer>(12);
    int nbre = val.getValeur();

    // exemples incorrects
    Solo<Integer> val = new Solo<Integer>("toto"); // mettre String à la place d'un entier
    val.setValeur(12.2f); // mettre un float à la place d'un entier
}
```

Si l'objet ne reçoit pas le bon type d'argument, il y a un conflit entre le type de données passé à votre instance lors de sa création et le type de données que vous essayez d'utiliser dans celle-ci.

La généricité peut être multiple.

```
public class Duo<T, S>
{
    private T valeur1;
    private S valeur2;
    ...
}

public static void main(String[] args)
{
    Duo<String, Boolean> dual = new Duo<String, Boolean>("toto", true);
    System.out.println("val1=" + dual.getValeur1() + ", val2=" + dual.getValeur2());

    Duo<Double, Character> dual2 = new Duo<Double, Character>(12.2585, 'C');
    System.out.println("val1=" + dual2.getValeur1() + ", val2=" + dual2.getValeur2());
}
```

- L'**autoboxing** est une fonctionnalité du langage permettant **de transformer automatiquement un type primitif en classe wrapper** (on appelle ça le **boxing**) **et inversement**, c'est-à-dire une classe wrapper en type primitif (on appelle ça l'**unboxing**).

Les classes **wrapper** sont des classes enveloppes qui ajoutent les méthodes de la classe Object au types primitifs, ainsi que des méthodes permettant de *caster* leurs valeurs.

```
int i = new Integer(12);           // équivalent à int i = 12
double d = new Double(12.2586);   // équivalent à double d = 12.2586
Double D = 12.0;
Character C = 'C';

al = new ArrayList();
// avant Java 5 il fallait faire al.add(new Integer(12))
// depuis Java 5 il suffit de faire
al.add(12);
```

**Conclusion** : plus besoin de faire le cast new Integer(...), new Double(...) pour les types primitifs.

- Vous pouvez **coupler les collections avec la généricité** !

```

System.out.println("Liste de String");
System.out.println("-----");
List<String> listeString= new ArrayList<String>();
listeString.add("Une chaîne");
listeString.add("Une autre");
listeString.add("Encore une autre");
listeString.add("Allez, une dernière");
for (String str : listeString)
    System.out.println(str);

System.out.println("\nListe de float");
System.out.println("-----");
List<Float> listeFloat = new ArrayList<Float>();
listeFloat.add(12.25f);
listeFloat.add(15.25f);
listeFloat.add(2.25f);
listeFloat.add(128764.25f);
for (float f : listeFloat)
    System.out.println(f);

```

- Le **wildcard (?)** permet d'indiquer que n'importe quel type peut être traité et donc accepté !

Le fait de **déclarer une collection avec le wildcard**, comme ceci :

```
ArrayList<?> list;
```

... revient à indiquer que notre **collection accepte n'importe quel type d'objet**.

Cependant, il y a une **restriction**. Dès que le wildcard (?) est utilisé, cela revient à rendre ladite collection en **lecture seule** ! Le wildcard signifie « tout objet », et dès l'utilisation de celui-ci, la JVM verrouillera la compilation du programme afin de prévenir les risques d'erreurs.

- Vous pouvez élargir le champ d'acceptation d'une collection générique grâce au mot-clé **extends**  
L'instruction **<? extends MaClasse>** autorise toutes les collections de classes ayant pour supertype MaClasse.

Exemple : List n'acceptant que des instances de Voiture ou de ses **classes filles**

```

public static void main(String[] args)
{
    List<? extends Voiture> listVoitureSP = new ArrayList<VoitureSansPermis>();
    afficher(listVoitureSP);
}

static void afficher(ArrayList<? extends Voiture> list) // méthode générique !
{
    for(Voiture v : list)
        System.out.println(v.toString());
}

```

Exemple : type d'utilisation de wildcard qui fonctionne à merveille pour la lecture

```
public static void main(String[] args)
{
    List<Voiture> listVoiture = new ArrayList<Voiture>();
    listVoiture.add(new Voiture());
    listVoiture.add(new Voiture());

    List<VoitureSansPermis> listVoitureSP = new ArrayList<VoitureSansPermis>();
    listVoitureSP.add(new VoitureSansPermis());
    listVoitureSP.add(new VoitureSansPermis());

    affiche(listVoiture);
    affiche(listVoitureSP);
}

// cette méthode accepte aussi bien les collections de Voiture que les collection de VoitureSansPermis
static void affiche(List<? extends Voiture> list)
{
    for (Voiture v : list)
        System.out.print(v.toString());
}
```

- L'instruction `<? super MaClasse>` autorise toutes les collections de classes ayant pour type `MaClasse` et tous ses supertypes !

```
public static void main(String[] args)
{
    List<Voiture> listVoiture = new ArrayList<Voiture>();
    listVoiture.add(new Voiture());
    listVoiture.add(new Voiture());

    List<Object> listVoitureSP = new ArrayList<Object>();
    listVoitureSP.add(new Object());
    listVoitureSP.add(new Object());

    affiche(listVoiture);
}

// cette méthode accepte les collections de Voiture et les collections d'Object
static void affiche(List<? super Voiture> list)
{
    for (Object v : list)
        System.out.print(v.toString());
}
```

- Pour ce genre de cas, les méthodes génériques sont particulièrement adaptées et permettent d'utiliser le polymorphisme dans toute sa splendeur !

## Le flux d'entrée / sortie

- Les classes traitant des entrées/sorties se trouvent dans le package **java.io**.

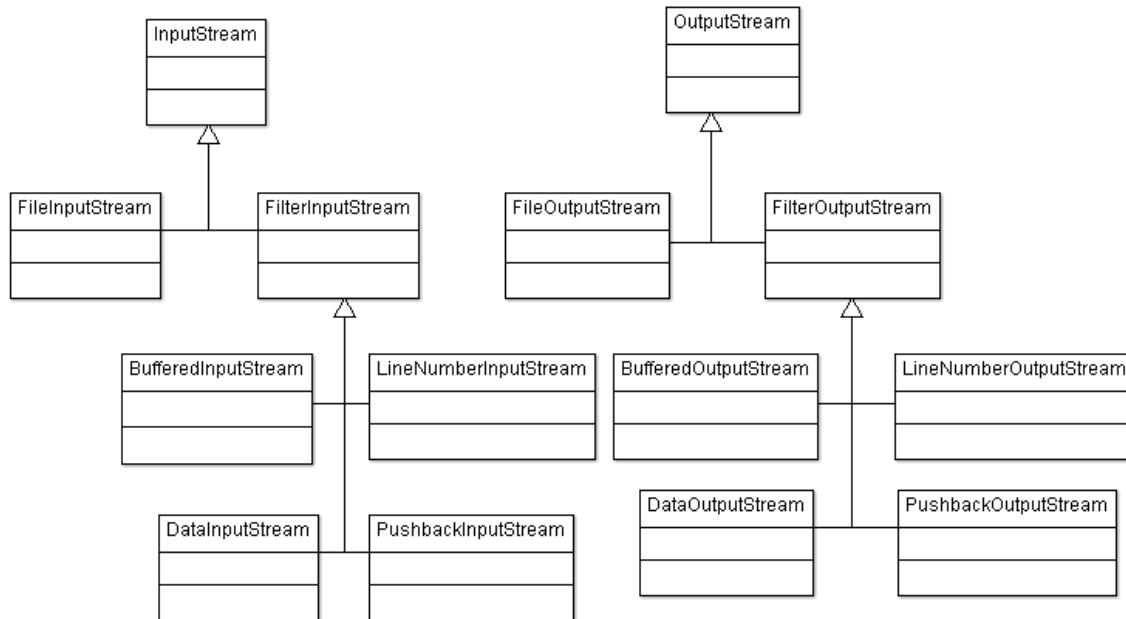
Une **entrée/sortie** en Java consiste en un **échange de données** entre le programme et une autre source, par exemple la mémoire, un fichier, le programme lui-même...

Pour réaliser cela, Java emploie ce qu'on appelle un **stream** (qui signifie « **flux** »). Celui-ci joue le rôle de médiateur entre la source des données et sa destination. Java met à notre disposition toute une panoplie d'objets permettant de communiquer de la sorte.

Toute opération sur les entrées/sorties doit suivre le schéma suivant :  
**ouverture, lecture, fermeture du flux.**

Java a décomposé les objets traitant des flux en deux catégories :

- les objets travaillant avec des flux d'entrée (**in**), pour la lecture de flux ;
- les objets travaillant avec des flux de sortie (**out**), pour l'écriture de flux.



- L'objet **File** :

```

import java.io.File; // package à importer afin d'utiliser l'objet File
public class Main
{
    public static void main(String[] args)
    {
        File f = new File("test.txt"); // création de l'objet File
        System.out.println("Chemin absolu du fichier : " + f.getAbsolutePath());
        System.out.println("Nom du fichier : " + f.getName());
        System.out.println("Est-ce qu'il existe ? " + f.exists());
        System.out.println("Est-ce un répertoire ? " + f.isDirectory());
        System.out.println("Est-ce un fichier ? " + f.isFile());
        System.out.println("Affichage des lecteurs à la racine du PC : ");

        for (File file : f.listRoots() )
        {
            System.out.println(file.getAbsolutePath());
            try
            {
                int i = 1;
                for ( File nom : file.listFiles() ) // la liste des fichiers et répertoires
                {
                    // s'il s'agit d'un dossier, on ajoute un "/"
                    System.out.print("\t\t" + ((nom.isDirectory()) ? nom.getName() + "/" : nom.getName()));
                    if ( (i % 4) == 0 ) { System.out.print("\n"); }
                    i++;
                }
                System.out.println("\n");
            }
            catch (NullPointerException e) {} // exception levé s'il n'y a pas de sous-fichier !
        }
    }
}

```

Le résultat :

```

<terminated> Main (1) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (13 août 2012 13:40:53)
Chemin absolu du fichier : C:\workspace\File\test.txt
Nom du fichier : test.txt
Est-ce qu'il existe ? true
Est-ce un répertoire ? false
Est-ce un fichier ? true
Affichage des lecteurs à la racine du PC :
C:\$Recycle.Bin/.rnd          Apps/      autoexec.bat
BACKUP.DD/config.sys          dell/      dell.
Documents and Settings/Drivers/  eula.1028.txt
eula.1033.txt                 eula.1036.txt   eula.1040.txt
eula.1042.txt                 eula.1049.txt   eula.2052.txt
glassfish3/globdata.ini        hiberfil.sys
install.ini                   install.res.1028.dll  install.res.1
install.res.1036.dll           install.res.1040.dll  insta
install.res.1049.dll           install.res.2052.dll  insta
10.SYS/LandparkIP/log2.txt    log2.txt      logPe
mcdbp.log/MSDOS.SYS           MSDOS.SYS      MSCache/
Partage/PerfLogs/              PerfLogs/      Program Files/

```

Vous pouvez aussi effacer le fichier grâce la méthode **delete()**, créer des répertoires avec la méthode **mkdir()**, etc.

- Les classes **FileInputStream** (pour lire dans un fichier) et **FileOutputStream** (pour écrire dans un fichier) sont héritées des classes **InputStream** (classes gérant les flux d'entrée) et **OutputStream** (classes gérant les flux de sortie), présentes dans le package java.io.

```

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class Main
{
    public static void main(String[] args)
    {
        // déclarer les objets en dehors du bloc try/catch
        FileInputStream fis = null;
        FileOutputStream fos = null;

        try {
            // On instancie nos objets : fis va lire le fichier et fos va écrire dans le nouveau !
            fis = new FileInputStream(new File("test.txt"));
            fos = new FileOutputStream(new File("test2.txt"));

            // on crée un tableau de byte pour indiquer le nombre de bytes lus à chaque tour de boucle
            byte[] buf = new byte[8];
            // variable de type int pour y affecter le résultat de la lecture; vaut -1 quand c'est fini
            int n = 0;

            // tant que l'affectation dans la variable est possible, on boucle
            // lorsque la lecture du fichier est terminée, l'affectation n'est plus possible! On sort de la boucle
            while ( (n = fis.read(buf)) >= 0)
            {
                // on écrit dans notre deuxième fichier avec l'objet adéquat
                fos.write(buf);

                // on affiche ce qu'a lu notre boucle au format byte et au format char
                for ( byte bit : buf )
                    { System.out.print("\t" + bit + "(" + (char) bit + ")"); }
                System.out.println("");

                // réinitialise le buffer à vide au cas où les derniers byte lus ne soient pas un multiple de 8
                // ça permet d'avoir un buffer vierge à chaque lecture et d'éviter les doublons en fin de fichier
                buf = new byte[8];
            }
            System.out.println("Copie terminée !");
        }
        catch (FileNotFoundException e) // si l'objet FileInputStream ne trouve aucun fichier
        { e.printStackTrace(); }
        catch (IOException e) // lors d'une erreur d'écriture ou de lecture
        { e.printStackTrace(); }
    }
}

```

```

finally
{
    // on ferme nos flux de données dans un bloc finally pour s'assurer que ces instructions
    // seront exécutées dans tous les cas même si une exception est levée !
    try
    {
        if ( fis != null )
            fis.close();
    }
    catch (IOException e)
        { e.printStackTrace(); }

    try
    {
        if ( fos != null)
            fos.close();
    }
    catch (IOException e)
        { e.printStackTrace(); }
    }
}
}

```

La façon dont on travaille avec des flux doit respecter la logique suivante :

- ouverture de flux ;
- lecture/écriture de flux ;
- fermeture de flux.

La gestion des flux peut engendrer la levée d'**exceptions** : *FileNotFoundException, IOException*

Le bloc **finally** permet de s'assurer que nos objets ont bien fermé leurs liens avec leurs fichiers respectifs, ceci afin de permettre à Java de détruire ces objets pour ainsi libérer un peu de mémoire à l'ordinateur.

Le code précédent lit 8 bits à la fois (en Java 1 byte = 1 octet = 8 bits), correspondant au code binaire UNICODE1 (les caractères de a à z, de A à Z, de 0 à 9, ...). Ce codage n'inclut pas les caractères accentués.

- Les objets **FilterInputStream** et **FilterOutputStream**

Ce sont des classes abstraites. Elles définissent un comportement global pour leurs classes filles qui, elles, permettent d'ajouter des fonctionnalités aux flux d'entrée / sortie

Il y a quatre classes filles héritant de **FilterInputStream** (de même pour FilterOutputStream) :

- **DataInputStream** : offre la possibilité de lire directement des types primitifs (double, char, int) grâce à des méthodes comme readDouble(), readInt(), ...
- **BufferedInputStream** : permet d'avoir un tampon à disposition dans la lecture du flux. En gros, les données vont tout d'abord remplir le tampon, et dès que celui-ci est plein, le programme accède aux données
- **PushBackInputStream** : permet de remettre un octet déjà lu dans le flux entrant
- **LineNumberInputStream** : permet de récupérer le numéro de la ligne lue à un instant T

Ces classes prennent en paramètre une instance dérivant des classes **InputStream** ou respectivement, de **OutputStream**.

```
FileInputStream fis = new FileInputStream(new File("toto.txt"));
DataInputStream dis = new DataInputStream(fis);
BufferedInputStream bis = new BufferedInputStream(dis);
// ou en condensé :
BufferedInputStream bis = new BufferedInputStream(
    new DataInputStream(
        new FileInputStream(
            new File("toto.txt"))));
```

Comparaison de temps lecture entre le **FileInputStream** et le **BufferedInputStream** :

```
import java.io.File;
import java.io.FileInputStream;
import java.io.BufferedInputStream;
import java.io.FileOutputStream;
import java.io	BufferedOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class Main
{
    public static void main(String[] args)
    {
        FileInputStream fis;
        FileOutputStream fos;
        BufferedInputStream bis;
        BufferedOutputStream bos;

        try
        {
            fis = new FileInputStream(new File("test.txt"));
            fos = new FileOutputStream(new File("test2.txt"));
            bis = new BufferedInputStream(new FileInputStream(new File("test.txt")));
            bos = new BufferedOutputStream(new FileOutputStream(new File("test3.txt")));
        }
```

```

byte[] buf = new byte[8];
long startTime = System.currentTimeMillis(); // on récupère le temps du système

while ( fis.read(buf) != -1 )
{ fos.write(buf); }

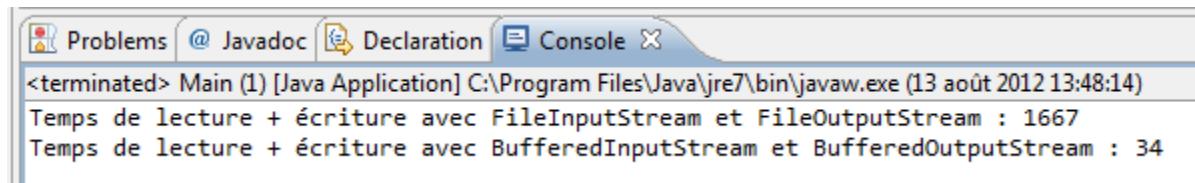
// on affiche le temps d'exécution
System.out.println("Temps de lecture + écriture avec FIS et FOS : "
+ (System.currentTimeMillis() - startTime));

// on réinitialise
startTime = System.currentTimeMillis();
while ( bis.read(buf) != -1 )
{ bos.write(buf); }

// on réaffiche
System.out.println("Temps de lecture + écriture avec BIS et BOS : "
+ (System.currentTimeMillis() - startTime));

// on ferme nos flux de données
fis.close();
bis.close();
fos.close();
bos.close();
}
catch (FileNotFoundException e)
{ e.printStackTrace(); }
catch (IOException e)
{ e.printStackTrace(); }
}
}

```



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The output window displays the execution results of the Java application. It starts with the application's title and path, followed by two lines of text indicating the time taken for different I/O operations.

```

Problems @ Javadoc Declaration Console
<terminated> Main (1) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (13 août 2012 13:48:14)
Temps de lecture + écriture avec FileInputStream et FileOutputStream : 1667
Temps de lecture + écriture avec BufferedInputStream et BufferedOutputStream : 34

```

- Les objets **DataInputStream** et **DataOutputStream**

Ceux-ci ont des méthodes de lecture pour chaque type primitif : il faut cependant que le fichier soit généré par le biais d'un DataOutputStream pour que les méthodes fonctionnent correctement

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class Main
{
    public static void main(String[] args)
    {
        DataInputStream dis;
        DataOutputStream dos;

        try
        {
            dos = new DataOutputStream(
                new BufferedOutputStream(
                    new FileOutputStream(
                        new File("sdz.txt"))));

            // écrire chaque type primitif
            dos.writeBoolean(true);
            dos.writeByte(100);
            dos.writeChar('C');
            dos.writeDouble(12.05);
            dos.writeFloat(100.52f);
            dos.writeInt(1024);
            dos.writeLong(123456789654321L);
            dos.writeShort(2);
            dos.close();

            // récupère maintenant les données
            dis = new DataInputStream(
                new BufferedInputStream(
                    new FileInputStream(
                        new File("sdz.txt"))));

            System.out.println(dis.readBoolean());
            System.out.println(dis.readByte());
            System.out.println(dis.readChar());
        }
    }
}
```

```
        System.out.println(dis.readDouble());
        System.out.println(dis.readFloat());
        System.out.println(dis.readInt());
        System.out.println(dis.readLong());
        System.out.println(dis.readShort());
    }
    catch (FileNotFoundException e)
    { e.printStackTrace(); }
    catch (IOException e)
    { e.printStackTrace(); }
}
}
```

- Les objets **ObjectInputStream** et **ObjectOutputStream**

Lorsqu'on veut écrire des objets dans des fichiers on appelle ça de la "**sérialisation**" : c'est le nom que porte l'action de sauvegarder des objets

Pour qu'un objet soit sérialisable, il doit implémenter l'interface **Serializable**. Cette interface n'a pas de méthode à redéfinir : c'est ce qu'on appelle une « interface marqueur ». Rien qu'en implémentant cette interface dans un objet, Java sait que cet objet peut être sérialisé.

Si vous n'implémentez pas cette interface dans vos objets, ceux-ci ne pourront pas être sérialisés. En revanche, si une superclasse implémente l'interface Serializable, ses enfants seront considérés comme sérialisables.

Objets à sérialiser :

```
import java.io.Serializable;

public class Game implements Serializable
{
    private String nom, style;
    private double prix;

    public Game(String nom, String style, double prix)
    {
        this.nom = nom;
        this.style = style;
        this.prix = prix;
    }

    public String toString()
    {
        return "Nom du jeu : " + this.nom + "\n"
               "Style de jeu : " + this.style + "\n"
               "Prix du jeu : " + this.prix + "\n";
    }
}
```

Sérialiser ces objets dans un fichier et désérialiser ces objets afin de les réutiliser :

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
```

```

public class Main
{
    public static void main(String[] args)
    {
        ObjectInputStream ois;
        ObjectOutputStream oos;

        try
        {
            // écrire chaque objet Game dans le fichier
            oos = new ObjectOutputStream(
                new BufferedOutputStream(
                    new FileOutputStream(
                        new File("game.txt"))));

            oos.writeObject(new Game("Assassin Creed", "Aventure", 45.69));
            oos.writeObject(new Game("Tomb Raider", "Plateforme", 23.45));
            oos.writeObject(new Game("Tetris", "Stratégie", 2.50));
            oos.close();                                // ne pas oublier de fermer le flux !

            // on récupère maintenant les données
            ois = new ObjectInputStream(
                new BufferedInputStream(
                    new FileInputStream(
                        new File("game.txt"))));

            try
            {
                System.out.println("Affichage des jeux :");
                System.out.println("*****\n");
                System.out.println(((Game)ois.readObject()).toString());
                System.out.println(((Game)ois.readObject()).toString());
                System.out.println(((Game)ois.readObject()).toString());
            }
            catch (ClassNotFoundException e)
            {
                e.printStackTrace();
            }

            ois.close();                                // ne pas oublier de fermer le flux !
        }
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}

```

Le résultat :

```
Affichage des jeux :  
*****  
Nom du jeu : Assassin Creed  
Style de jeu : Aventure  
Prix du jeu : 45.69  
  
Nom du jeu : Tomb Raider  
Style de jeu : Plateforme  
Prix du jeu : 23.45  
  
Nom du jeu : Tetris  
Style de jeu : Stratégie  
Prix du jeu : 2.5
```

Si un objet sérialisable comporte un objet d'instance non sérialisable, une **exception** sera levée lorsque vous voudrez sauvegarder votre objet.

```
public class Game implements Serializable  
{  
    private String nom, style;  
    private double prix;  
    private Notice notice; // objet Notice (non sérialisable)  
    ...  
}
```

L'une des solutions consiste à rendre l'objet d'instance sérialisable, l'autre à le déclarer **transient** afin qu'il soit ignoré à la sérialisation.

```
public class Game implements Serializable  
{  
    private String nom, style;  
    private double prix;  
  
    private transient Notice notice; // maintenant, cette variable ne sera pas sérialisée;  
                                    // elle sera tout bonnement ignorée  
    ...  
}
```

**Attention** à ne pas inclure cette variable dans la méthode `toString()` si elle n'est pas sérialisable.

- Les classes **CharArrayWriter / CharArraReader** et **StringWriter / StringReader**

Ces deux types jouent quasiment le même rôle. De plus, ils ont les mêmes méthodes que leur classe mère. Ces deux objets n'ajoutent donc aucune nouvelle fonctionnalité à leur objet mère.

Leur principale fonction est de permettre d'écrire un flux de caractères dans un buffer adaptatif : un emplacement en mémoire qui peut changer de taille selon les besoins.

```
import java.io.CharArrayReader;
import java.io.CharArrayWriter;
import java.io.IOException;

public class Main
{
    public static void main(String[] args)
    {
        CharArrayWriter caw = new CharArrayWriter();
        CharArrayReader car;

        try
        {
            caw.write("Coucou les Zéros");      // appel à la méthode toString de notre objet
            System.out.println(caw);
            caw.close();          //caw.close() n'a aucun effet sur le flux; Seul caw.reset() peut tout effacer

            // on passe un tableau de caractères à l'objet qui va lire le tampon
            car = new CharArrayReader(caw.toCharArray());

            int i;
            String str = "";                  // on remet tous les caractères lus dans un String

            while ( ( i = car.read() ) != -1 )
                str += (char) i;

            System.out.println(str);

        }
        catch (IOException e)
        { e.printStackTrace(); }
    }
}
```

Même comportement pour StringReader :

```
StringWriter sw = new StringWriter();
StringReader sr;
...
sr = new StringReader(sw.toString());
...
```

- Les classes **FileWriter / FileReader** et **PrintWriter / PrintReader**

Les objets travaillant avec des flux utilisent des **flux binaires**.

La conséquence est que même si vous ne mettez que des caractères dans un fichier et que vous le sauvegardez, les objets étudiés précédemment traiteront votre fichier de la même façon que s'il contenait des données binaires !

Ces deux objets, présents dans le package **java.io**, servent à lire et écrire des données dans un fichier **texte**.

```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.FileNotFoundException;

public class Main
{
    public static void main(String[] args)
    {
        File file = new File("testFileWriter.txt");
        FileWriter fw;
        FileReader fr;

        String str = "Bonjour à tous, amis Zéros !\n\tComment allez-vous ? \n";

        try
        {
            fw = new FileWriter(file);           // création de l'objet
            fw.write(str);                    // écrire la chaîne
            fw.close();                      // fermer le flux

            fr = new FileReader(file);         // création de l'objet de lecture

            str = "";
            int i = 0;
            while( (i = fr.read()) != -1 )    // lecture des données
                str += (char)i;

            System.out.println(str);          // affichage
            fr.close();                      // fermer le flux
        }
        catch (FileNotFoundException e)
        { e.printStackTrace(); }
        catch (IOException e)
        { e.printStackTrace(); }
    }
}
```

- **java.nio**

Depuis le JDK 1.4, un nouveau package a vu le jour, visant à améliorer les performances des flux, buffers, réseau, etc., traités par java.io : java.nio; nio signifie "New I/O".

Il permet de lire les données d'une façon différente. Ces objets traitent les données par **blocs de données** : la lecture est donc accélérée !

Tout repose sur deux objets de ce nouveau package : les **channels** et les **buffers**.

Les channels sont en fait des flux, tout comme dans l'ancien package, mais ils sont amenés à travailler avec un buffer dont vous définissez la taille.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.BufferedInputStream;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.FileChannel;
import java.io.IOException;
import java.io.FileNotFoundException;

public class Main
{
    public static void main(String[] args)
    {
        FileInputStream fis;
        BufferedInputStream bis;
        FileChannel fc;

        try
        {
            fis = new FileInputStream(new File("test.txt"));
            bis = new BufferedInputStream(fis);

            long time = System.currentTimeMillis(); // démarrage du chrono

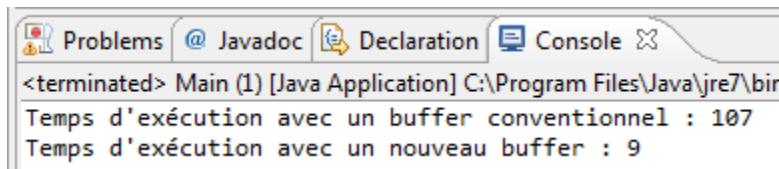
            while ( bis.read() != -1); // lecture totale
            System.out.println("Temps exéc buffer :" + (System.currentTimeMillis() - time));

            fis = new FileInputStream(new File("test.txt")); // nouveau flux de fichier
            fc = fis.getChannel(); // on récupère le canal
            int size = (int)fc.size(); // on en déduit la taille

            ByteBuffer bBuff = ByteBuffer.allocate(size); // buffer correspondant à la taille du fichier

            time = System.currentTimeMillis(); // démarrage du chrono
            fc.read(bBuff); // démarrage de la lecture
            bBuff.flip(); // on prépare à la lecture avec l'appel à flip
            System.out.println("Temps exéc buffer :" + (System.currentTimeMillis() - time));
        }
    }
}
```

```
//Puisque nous avons utilisé un buffer de byte afin de récupérer les données  
//Nous pouvons utiliser un tableau de byte; la méthode array retourne un tableau de byte  
byte[] tabByte = bBuff.array();  
}  
catch (FileNotFoundException e)  
{ e.printStackTrace(); }  
catch (IOException e)  
{ e.printStackTrace(); }  
}  
}  
}
```



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The output window displays the following text:  
<terminated> Main (1) [Java Application] C:\Program Files\Java\jre7\bin  
Temps d'exécution avec un buffer conventionnel : 107  
Temps d'exécution avec un nouveau buffer : 9

L'utilisation de *buffers* permet une nette amélioration des performances en lecture et en écriture de fichiers.

Il y a un buffer par type primitif pour la lecture sur le channel :  
IntBuffer, CharBuffer, ShortBuffer, ByteBuffer, DoubleBuffer, FloatBuffer, LongBuffer

- **Gestion des exceptions sur les flux**

Avec l'arrivée de Java 7, quelques nouveautés ont vu le jour pour la gestion des exceptions sur les flux. Contrairement à la **gestion de la mémoire** (vos variables, vos classes, etc.) qui est **déléguée au garbage collector** (ramasse miette), plusieurs types de **ressources doivent être gérées manuellement**. Les flux sur des fichiers en font parti mais, d'un point de vue plus général, toutes les ressources que vous devez fermer manuellement (**flux réseaux, connexions à une base de données, ...**).

Pour ce genre de flux, vous avez vu qu'il vous faut déclarer une variable en dehors d'un bloc try{...}catch{...} afin qu'elle soit accessible dans les autres blocs d'instructions.

Java 7 initie ce qu'on appelle vulgairement le « **try-with-resources** ». Ceci vous permet de déclarer les ressources utilisées directement dans le bloc try(...), ces dernières seront automatiquement fermées à la fin du bloc d'instructions ! Ainsi, si nous reprenons notre code de début de chapitre qui copie notre fichier *test.txt* vers *test2.txt*, nous aurons ceci :

```
try ( FileInputStream fis = new FileInputStream("test.txt");
      FileOutputStream fos = new FileOutputStream("test2.txt") )
{
    byte[] buf = new byte[8];
    int n = 0;

    while ( (n = fis.read(buf)) >= 0)
    {
        fos.write(buf);
        for ( byte bit : buf )
            System.out.print("\t" + bit + "(" + (char)bit + ")");
        System.out.println("");
    }

    System.out.println("Copie terminée !");
}
catch (IOException e)
{
    e.printStackTrace();
}
```

C'est tout de même beaucoup plus clair et plus lisible qu'avant, surtout que vous n'avez plus à vous soucier de la fermeture dans le bloc finally.

Il faut cependant prendre quelques précautions notamment pour ce genre de déclaration :

```
try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("test.txt"))) { ... }
```

Le fait d'avoir des ressources encapsulées dans d'autres ne rend pas « visible » les ressources encapsulées. Dans le cas précédent, si une exception est levée, le flux correspondant à l'objet *FileInputStream* ne sera pas fermé. Pour pallier ce problème il suffit de bien découper toutes les ressources à utiliser, comme ceci :

```
try (FileInputStream fis = new FileInputStream("test.txt");
     ObjectInputStream ois = new ObjectInputStream(fis)) { ... }
```

Pour rendre la fermeture automatique possible, les développeurs de la plateforme Java 7 ont créé l'interface : *java.lang.AutoCloseable*. Seuls les objets implémentant cette interface peuvent être utilisés de la sorte ! (la classe File n'en fait pas parti).

- Depuis Java 7 : **nio II**

L'une des grandes nouveautés de Java 7 réside dans NIO.2 avec un nouveau package **java.nio.file** en remplacement de la classe **java.io.File**.

Voici un bref listing de quelques nouveautés :

- une **meilleure gestion des exceptions** : la plupart des méthodes de la classe File se contentent de renvoyer une valeur nulle en cas de problème; des exceptions seront levées permettant de mieux cibler la cause du (ou des) problème(s) ;
- un **accès complet au système de fichiers** (support des liens/liens symboliques, etc.) ;
- l'ajout de **méthodes utilitaires** tels que le déplacement / la copie de fichier, la lecture / écriture binaire ou texte...
- récupérer la liste des fichiers d'un répertoire via un flux ;
- remplacement de la classe **java.io.File** par l'interface **java.nio.file.Path**.

Afin d'être le plus souple et complet possible, les développeurs de la plateforme ont créé une **interface java.nio.file.Path** dont le rôle est de récupérer et manipuler des chemins de fichiers de dossier et une une **classe java.nio.file.Files** qui contient tout un tas de méthodes qui simplifient certaines actions (copie, déplacement, etc.) et permet aussi de récupérer tout un tas d'informations sur un chemin.

```
Path path = Paths.get("test.txt");
System.out.println("Chemin absolu du fichier : " + path.toAbsolutePath());
System.out.println("Nom du fichier : " + path.getFileName());
System.out.println("Est-ce qu'il existe ? " + Files.exists(path));
System.out.println("Est-ce un répertoire ? " + Files.isDirectory(path));
```

La classe **Files** vous permet aussi de lister le contenu d'un répertoire mais via un objet **DirectoryStream** qui est un *itérateur*. Ceci évite de charger tous les fichiers en mémoire pour récupérer leurs informations.

```
// on récupère maintenant la liste des répertoires dans une collection typée via l'objet FileSystem
// qui représente le système de fichier de l'OS hébergeant la JVM
Iterable<Path> roots = FileSystems.getDefault().getRootDirectories();

for(Path chemin : roots) // parcourir
{
    System.out.println(chemin);
    try ( DirectoryStream<Path> listing = Files.newDirectoryStream(chemin))
    {
        int i = 0;
        for(Path nom : listing)
        {
            System.out.print("\t\t" + ((Files.isDirectory(nom)) ? nom + "/" : nom));
            i++;
            if ( i % 4 == 0 ) System.out.println("\n");
        }
    }
    catch (IOException e) { e.printStackTrace(); }
}
```

Vous avez également la possibilité d'ajouter un filtre à votre listing de répertoire afin qu'il ne liste que certains fichiers :

```
try(DirectoryStream<Path> listing = Files.newDirectoryStream(chemin, "*.txt"))
{ ... }
```

C'est vrai que cela change grandement la façon de faire et elle peut paraître plus complexe. Mais l'objet Files simplifie aussi beaucoup de choses.

Voici quelques exemple de méthodes utilitaires :

- la **copie** de fichier

```
Path source = Paths.get("test.txt");
Path cible = Paths.get("test2.txt");
try
{
    Files.copy(source, cible, StandardCopyOption.REPLACE_EXISTING);
}
catch (IOException e) { e.printStackTrace(); }
```

- le **déplacement** de fichier

```
Path source = Paths.get("test2.txt");
Path cible = Paths.get("test3.txt");
try
{
    Files.move(source, cible, StandardCopyOption.REPLACE_EXISTING);
}
catch (IOException e) { e.printStackTrace(); }
```

- **supprimer** un fichier : **Files.delete(path)**

- **créer** un fichier vide : **Files.createFile(path)**

- ouvrir des flux : pratique pour lire ou écrire dans un fichier

```
Path source = Paths.get("test.txt");

// ouverture en lecture :
try ( InputStream input = Files.newInputStream(source) ) { ... }

// ouverture en écriture :
try ( OutputStream output = Files.newOutputStream(source) ) { ... }

// ouverture d'un Reader en lecture :
try ( BufferedReader reader = Files.newBufferedReader(source,
StandardCharsets.UTF_8) ) { ... }

// ouverture d'un Writer en écriture :
try ( BufferedWriter writer = Files.newBufferedWriter(source,
StandardCharsets.UTF_8) ) { ... }
```

- gérer les fichiers ZIP grâce à l'objet FileSystem

```
// création d'un système de fichiers en fonction d'un fichier ZIP
try (FileSystem zipFS =
      FileSystems.newFileSystem(Paths.get("monFichier.zip"), null))
{
    // suppression d'un fichier à l'intérieur du ZIP :
    Files.deleteIfExists( zipFS.getPath("test.txt") );

    // création d'un fichier à l'intérieur du ZIP :
    Path path = zipFS.getPath("nouveau.txt");
    String message = "Hello World !!!!";
    Files.write(path, message.getBytes());

    // Parcours des éléments à l'intérieur du ZIP :
    try (DirectoryStream<Path> stream =
          Files.newDirectoryStream(zipFS.getPath("/")))
    {
        for (Path entry : stream)
            { System.out.println(entry); }
    }

    // copie d'un fichier du disque vers l'archive ZIP :
    Files.copy(Paths.get("fichierSurDisque.txt"),
               zipFS.getPath("fichierDansZIP.txt"));
}
```

- Afin de pouvoir ajouter des fonctionnalités aux objets gérant les flux, Java utilise le **pattern « decorator »**

Vous avez pu remarquer que les objets IO utilisent des instances d'objets de même supertype dans leur constructeur :

```
DataInputStream dis = new DataInputStream(
    new BufferedInputStream(
        new FileInputStream(
            new File("sdz.txt"))));
```

La raison d'agir de la sorte est simple : c'est pour **ajouter de façon dynamique des fonctionnalités à un objet**. En fait, au moment de récupérer les données de notre objet *DataInputStream*, celles-ci vont d'abord transiter par les objets passés en paramètre. Ce mode de fonctionnement suit une certaine structure et une certaine hiérarchie de classes : c'est le pattern decorator. Ce pattern de conception permet d'ajouter des fonctionnalités à un objet sans avoir à modifier son code source.

**Exemple** : travailler avec un objet Gateau qui héritera d'une classe abstraite Patisserie. Le but du jeu est de pouvoir ajouter des couches à notre gâteau sans avoir à modifier son code source.

Vous avez vu avec le **pattern strategy** que la composition (« A un ») est souvent préférable à l'héritage (« Est un ») : vous aviez défini de nouveaux comportements pour vos objets en créant un supertype d'objet par comportement. Ce pattern aussi utilise la composition comme principe de base : vous allez voir que nos objets seront composés d'autres objets. La différence réside dans le fait que nos nouvelles fonctionnalités ne seront pas obtenues uniquement en créant de nouveaux objets, mais en associant ceux-ci à des objets existants. Ce sera cette association qui créera de nouvelles fonctionnalités !

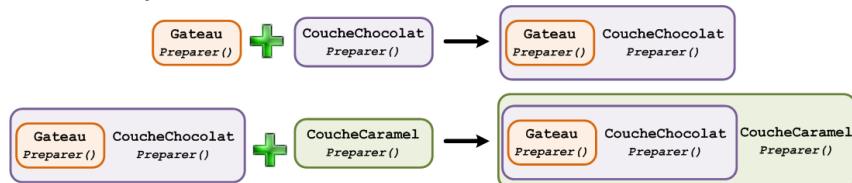
Nous allons procéder de la façon suivante :

- nous allons créer un objet Gateau;
- nous allons lui ajouter une CoucheChocolat;
- nous allons aussi lui ajouter une CoucheCaramel;
- nous appellerons la méthode qui confectionnera notre gâteau.

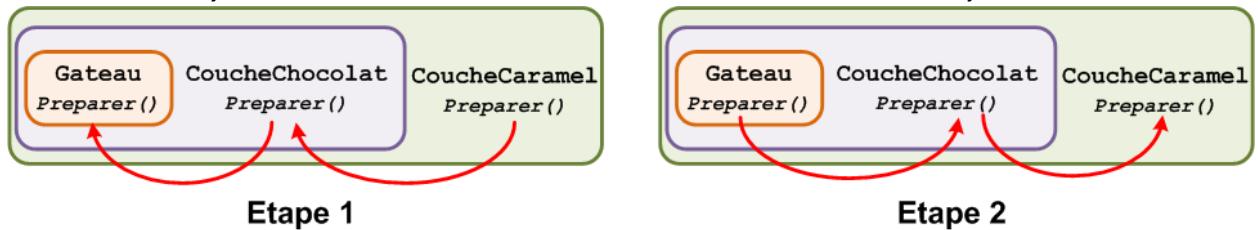
Tout cela démarre avec un concept fondamental : l'objet de base et les objets qui le décorent doivent être du même type, et ce, toujours pour la même raison : le polymorphisme, le polymorphisme, et le polymorphisme !

En fait, les objets qui vont décorer notre gâteau posséderont la même méthode *Preparer()* que notre objet principal, et nous allons faire fondre cet objet dans les autres. Cela signifie que nos objets qui vont servir de décorateurs comporteront une instance de type Patisserie; ils vont englober les instances les unes après les autres et du coup, nous pourrons appeler la méthode *Preparer()* de manière récursive !

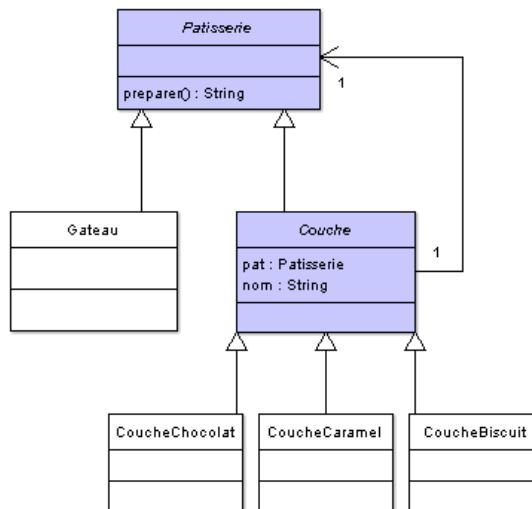
Vous pouvez voir les décorateurs comme des poupées russes : il est possible de mettre une poupée dans une autre. Cela signifie que si nous décorons notre gateau avec un objet CoucheChocolat un objet CoucheCaramel.



L'objet CoucheCaramel contient l'instance de la classe CoucheChocolat qui, elle, contient l'instance de Gateau : en fait, on va passer notre instance d'objet en objet ! Nous allons ajouter les fonctionnalités des objets « **décorants** » en appelant la méthode *préparer()* de l'instance se trouvant dans l'objet avant d'effectuer les traitements de la même méthode de l'objet courant.



Ce système ressemble fortement à la pile d'invocations de méthodes :



Vous remarquez sur ce diagramme que notre classe mère Patisserie est en fait la *strategy* (une classe encapsulant un comportement fait référence au pattern strategy : on peut dire qu'elle est la *strategy* de notre hiérarchie) de notre structure, c'est pour cela que nous pourrons appeler la méthode *préparer()* de façon récursive afin d'ajouter des fonctionnalités à nos objets.

Voici les différentes classes utilisées :

```
Patisserie.java
public abstract class Patisserie
{
    public abstract String preparer();
}
```

```
Gateau.java
public class Gateau extends Patisserie
{
    public String preparer()
    {
        return "Je suis un gâteau et je suis constitué des éléments suivants. \n";
    }
}
```

```
Couche.java
public abstract class Couche extends Patisserie
{
    protected Patisserie pat;
    protected String nom;

    public Couche(Patisserie p)
    { pat = p; }

    public String preparer()
    {
        String str = pat.preparer();
        return str + nom;
    }
}
```

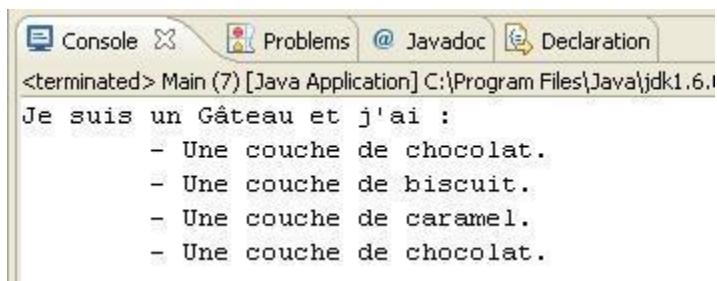
```
CoucheChocolat.java
public class CoucheChocolat extends Couche
{
    public CoucheChocolat(Patisserie p)
    {
        super(p);
        this.nom = "\t- Une couche de chocolat.\n";
    }
}
```

```
CoucheCaramel.java
public class CoucheCaramel extends Couche
{
    public CoucheCaramel(Patisserie p)
    {
        super(p);
        this.nom = "\t- Une couche de caramel.\n";
    }
}
```

```
CoucheBiscuit.java
public class CoucheBiscuit extends Couche
{
    public CoucheBiscuit(Patisserie p)
    {
        super(p);
        this.nom = "\t- Une couche de biscuit.\n";
    }
}
```

Et voici un code de test ainsi que son résultat :

```
public class Main
{
    public static void main(String[] args)
    {
        Patisserie pat = new CoucheChocolat(
            new CoucheCaramel(
                new CoucheBiscuit(
                    new CoucheChocolat(
                        new Gateau()))));
        System.out.println(pat.preparer());
    }
}
```



The screenshot shows a Java application running in an IDE. The title bar says "Main (7) [Java Application] C:\Program Files\Java\jdk1.6.0\_20\bin". The console tab is active, displaying the following text:

```
<terminated> Main (7) [Java Application] C:\Program Files\Java\jdk1.6.0_20\bin
Je suis un Gâteau et j'ai :
    - Une couche de chocolat.
    - Une couche de biscuit.
    - Une couche de caramel.
    - Une couche de chocolat.
```

Ce pattern permet d'encapsuler une fonctionnalité et de l'invoquer de façon récursive sur les objets étant composés de décorateurs.

Vous avez pu voir que l'invocation des méthodes se faisait en allant jusqu'au dernier élément pour remonter ensuite la pile d'invocations. Pour inverser ce fonctionnement, il vous suffit d'inverser les appels dans la méthode *préparer()* : affecter d'abord le nom de la couche et ensuite le nom du décorateur.

- !!!! Un exemple complet lecture et écriture **fichier text**

Les objets de type **DataInputStream** ne sont pas adapté à la lecture des fichiers text.

Pour lire dans les fichiers texte vaut mieux utiliser les objets de type :

**FileReader, BufferedReader, Scanner, Formatter**

Pour écrire dans les fichiers texte, vaut mieux utiliser les objets de type : **FileWriter**,

Ecrire dans un fichier :

```
File fileW = new File("textFileWords.txt");
File fileN = new File("textFileNumbers.txt");

// écrire des mots dans un fichier
try ( FileWriter fw = new FileWriter(fileW) )
{
    fw.write("Bonjour à tous, amis Zéros !\n");
    fw.write("\tComment allez-vous ? \n");
    fw.write("\n\n");
    fw.write(" Ca \t bim _ + ?");
}
catch (FileNotFoundException e)
{
    System.out.println("Catch error : " + e.getMessage());
}
catch (IOException e)
{
    System.out.println("Catch error : " + e.getMessage());
}

// écrire des numéros dans un fichier
try ( FileWriter fw = new FileWriter(fileN) )
{
    fw.write("5 4\n");
    fw.write("9\n");
    fw.write("1 2\n");
    fw.write("1 4 6\n");
    fw.write("7 4 5 2\n");
    fw.write("6 7 3\n");
}
catch (FileNotFoundException e)
{
    System.out.println("Catch error : " + e.getMessage());
}
catch (IOException e)
{
    System.out.println("Catch error : " + e.getMessage());
}
```

Lire dans un fichier avec un objet BufferedReader :

```
File fileW = new File("textFileWords.txt");
File fileN = new File("textFileNumbers.txt");

// lire des mots dans un fichier -- v1 avec BufferedReader
System.out.println("\n\nLire des mots dans un fichier avec BufferedReader");

try (
    InputStream fis = new FileInputStream(fileW);
    InputStreamReader isr = new InputStreamReader(fis, Charset.forName("UTF-8"));
    BufferedReader br = new BufferedReader(isr); )
{
    while ( (line = br.readLine()) != null )
    {
        String[] words = line.split("\\s");
        for ( String word : words)
            System.out.println(word); // pour int : System.out.println(Integer.parseInt(word))
    }
}
catch (FileNotFoundException e)
{
    System.out.println("Catch error : " + e.getMessage());
}
catch (IOException e)
{
    System.out.println("Catch error : " + e.getMessage());
}
```

Résultats :

|            |   |
|------------|---|
| Bonjour    | 5 |
| à          | 4 |
| tous,      | 9 |
| amis       | 1 |
| Zéros      | 2 |
| !          | 1 |
| Comment    | 1 |
| allez-vous | 4 |
| ?          | 6 |
|            | 7 |
|            | 4 |
| Ca         | 5 |
|            | 2 |
|            | 6 |
|            | 7 |
|            | 3 |
| bim        |   |

-  
+  
?

Lire dans un fichier avec un objet Scanner :

```
File fileW = new File("textFileWords.txt");
File fileN = new File("textFileNumbers.txt");

// lire des mots dans un fichier -- v2 avec Scanner
System.out.println("\n\nLire des mots dans un fichier avec Scanner");

try ( Scanner sc = new Scanner(fileW) ) // pour int : try ( Scanner sc = new Scanner(fileN) )
{
    while (sc.hasNext() )
    {
        String word = sc.next();           // pour int : int nb = Integer.parseInt(sc.next());
        System.out.println(word);         // pour int : System.out.println(nb);
    }
}
catch (FileNotFoundException e)
{
    System.out.println("Catch error : " + e.getMessage());
}
catch (IOException e)
{
    System.out.println("Catch error : " + e.getMessage());
}
```

Résultats :

|            |    |
|------------|----|
| Bonjour    | 5  |
| à          | 4  |
| tous,      | 9  |
| amis       | 1  |
| Zéros      | 2  |
| !          | 1  |
| Comment    | 4  |
| allez-vous | 6  |
| ?          | 7  |
| Ca         | 4  |
| bim        | 5  |
| -          | 26 |
| +          | 7  |
| ?          | 3  |

## Java et la réflexivité

La **réflexivité**, aussi appelée **introspection**, consiste à découvrir de façon dynamique des informations relatives à une classe ou à un objet. C'est notamment utilisé au niveau de la machine virtuelle Java lors de l'exécution du programme. En gros, **la machine virtuelle stocke les informations relatives à une classe dans un objet**.

La réflexivité n'est que le moyen de connaître toutes les informations concernant une classe donnée. Vous pourrez même créer des instances de classe de façon dynamique grâce à cette notion.

- L'objet **Class**

Au chargement d'une classe Java, votre **JVM** crée automatiquement un objet. Celui-ci récupère toutes les caractéristiques de votre classe ! Il s'agit d'un objet **Class**.

Exemple : objets String

```
public static void main(String[] args)
{
    Class c = String.class;
    Class c2 = new String().getClass();
}
```

Fonctionnalités de l'objet Class

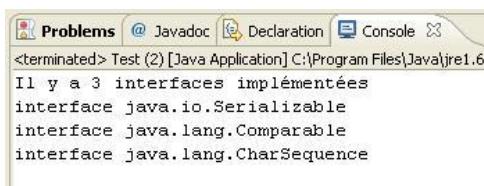
- connaître la superclasse d'une classe

```
System.out.println("La superclasse de " + String.class.getName());
System.out.println(" est : " + String.class.getSuperclass());
// => La superclasse de la classe java.lang.String est : class java.lang.Object
// => La superclasse de la classe java.lang.Object est : null
```

- connaître la liste des interfaces d'une classe

```
Class c = new String().getClass(); // équivalent avec Class c = String.class;
Class[] faces = c.getInterfaces(); // getInterfaces() retourne un tableau de Class

System.out.println("Il y a " + faces.length + " interfaces implémentées");
for(int i = 0; i < faces.length; i++)
    System.out.println(faces[i]);
```



- connaître la liste des méthodes d'une classe

```
Class c = new String().getClass();
Method[] m = c.getMethods();

System.out.println("Il y a " + m.length + " méthodes implémentées ");
for(int i = 0; i < m.length; i++)
    System.out.println(m[i]);
```

- connaître la liste des champs (variable de classe ou d'instance)

```
Class c = new String().getClass();
Field[] m = c.getDeclaredFields();

System.out.println("Il y a " + m.length + " champs dans cette classe");
for(int i = 0; i < m.length; i++)
    System.out.println(m[i].getName());
```

- connaître la liste des constructeurs

```
Class c = new String().getClass();
Constructor[] construc = c.getConstructors();
System.out.println("Il y a " + construc.length + " constructeurs ");

for(int i = 0; i < construc.length; i++)
{
    System.out.println(construc[i].getName());

    Class[] param = construc[i].getParameterTypes();
    for(int j = 0; j < param.length; j++)
        System.out.println(param[j]);
}
```

### • Instanciation dynamique

Exercice : créer un objet *Paire* sans utiliser l'opérateur **new**

| Paire                                                                                                                                                                               |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| valeur1 : String<br>valeur2 : String                                                                                                                                                |
| toString() : String<br>getValeur1() : String<br>getValeur2() : String<br>setValeur1() : void<br>setValeur2() : void<br>Paire() : void<br>Paire(val1 : String, val2 : String) : void |

```
public static void main(String[] args)
{
    String nom = Paire.class.getName();
    try
    {
        Class cl = Class.forName(nom);           // on crée un objet Class
        Object o = cl.newInstance();             // nouvelle instance de la classe Paire

        Class[] types = new Class[]{String.class, String.class}; //params du construct
        Constructor ct = cl.getConstructor(types);          // construct avec les 2 params
        Object o2 = ct.newInstance(new String[]{"val1", "val2"} ); // instancie l'objet
```

```

// chercher la méthode toString, sans aucun paramètre
Method m = cl.getMethod("toString", null);

// la méthode invoke exécute la méthode sur l'objet passé en paramètre
// pas de paramètre, donc null en deuxième paramètre de la méthode invoke !

System.out.println("-----");
System.out.println("Méthode " + m.getName() + " sur o2: " + m.invoke(o2, null));
System.out.println("Méthode " + m.getName() + " sur o: " + m.invoke(o, null));
}

catch (SecurityException e) { e.printStackTrace(); }
catch (IllegalArgumentException e) { e.printStackTrace(); }
catch (ClassNotFoundException e) { e.printStackTrace(); }
catch (InstantiationException e) { e.printStackTrace(); }
catch (IllegalAccessException e) { e.printStackTrace(); }
catch (NoSuchMethodException e) { e.printStackTrace(); }
catch (InvocationTargetException e) { e.printStackTrace(); }
}

```

<terminated> Test (2) [Java Application] C:\Program Files\Java\jre1.6.0\_03\bin\javaw.exe (26 févr. 08 17:19:25)

Instanciation ! !

Instanciation avec des paramètres ! !

-----

Méthode toString sur o2: Je suis un objet qui a pour valeur: valeur 1 - valeur 2

Méthode toString sur o: Je suis un objet qui a pour valeur: null - null

Nous avons créé deux instances d'une classe sans passer par l'opérateur new. Mieux encore, nous avons pu appeler une méthode de nos instances !

## Les interfaces graphiques

Interfaces graphiques : IHM (interfaces homme-machine), GUI (Graphical User Interfaces)

Programmation événementielle : ces programmes ne répondent plus à des saisies au clavier, mais à des événements provenant d'un composant graphique : un bouton, une liste, un menu, ...

Bibliothèques utiles pour programmer des IHM : **javax.swing** (composants) et **java.awt** (interactions)

Composants **awt**, lourds (*HeavyWeight*), propres à java 1.0

Composants **swing**, légers (*LightWeight*), propres à java 1.2 (java 2)

Conseil : ne pas mélanger des objets *awt* et *swing* dans une même fenêtre



Nous avons dans l'ordre :

- **fenêtre**
- **RootPane** (vert) : conteneur principal, contient les autres composants
- **LayeredPane** (violet) : forme juste un panneau composé du conteneur global et de la barre de menu (*MenuBar*)
- **MenuBar** (orange) : la barre de menu, quand il y en a une
- **ContentPane** (rose) : dans celui-ci nous placerons nos composants

utilisé pour intercepter les actions de l'utilisateur avant qu'elles ne

- L'objet **JFrame**

Création d'une **fenêtre** : définir la taille, le titre, la position, la fermeture du programme

```
import javax.swing.JFrame;

public class Fenetre extends JFrame
{
    public Fenetre()
    {
        this.setTitle("Ma première fenêtre Java");
        this.setSize(400, 500);
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
    }
}

public class Test
{
    public static void main(String[] args)
    {
        Fenetre fenetre = new Fenetre ();
    }
}
```

Par défaut, une fenêtre à une taille minimale et n'est pas visible.

Autres méthodes utiles :

- **setLocation(int x, int y)** : coordonnées en pixels; origine = coin supérieur gauche
- **setResizable(boolean b)** : empêche le redimensionnement de la fenêtre
- **setAlwaysOnTop(boolean b)** : garder la fenêtre au premier plan
- **setUndecorated(boolean b)** : retire les contours et les boutons de contrôle

- L'objet **JPanel**

Composant de type conteneur, dont la vocation est d'accueillir d'autres conteneurs ou des objets de type composant (boutons, cases à cocher, ...).

Pour utiliser un JPanel, il faut l'instancier, lui spécifier une couleur de fond pour mieux le distinguer et avertit notre JFrame que ce sera notre JPanel qui constituera son content pane.

```
public class Fenetre extends JFrame
{
    public Fenetre()
    {
        ...
        JPanel pan = new JPanel();
        pan.setBackground(Color.ORANGE);

        this.setContentPane(pan);
        this.setVisible(true);
    }
}
```

- L'objet **Graphics**

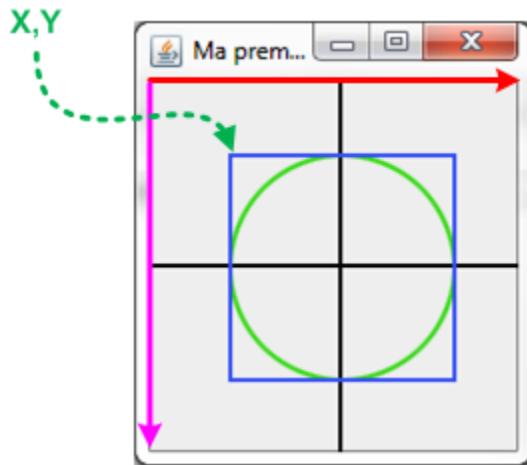
Pour l'utiliser : le récupérer du système via la méthode **getGraphics()** d'un composant swing.

```
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel
{
    public void paintComponent(Graphics g)
    {
        System.out.println("Je suis exécutée !");
        g.fillOval(20, 20, 75, 75); // trace un rond plein à la position x=20, y=20
                                    // de taille 75px de large et 75 px de haut
    }
}
```

**paintComponent(Graphics g)**: méthode appelée automatiquement pour dessiner sur la fenêtre

Pour placer le rond au milieu de la fenêtre, faut faire les calculs suivants :



```
public void paintComponent(Graphics g)
{
    int x1 = this.getWidth() / 4;
    int y1 = this.getHeight() / 4;
    g.fillOval(x1, y1, this.getWidth() / 2, this.getHeight() / 2);
}
```

Autres méthodes utiles :

- **drawOval(x1, y1, w, h)** : dessine un rond vide
- **drawRect(x1, y1, w, h)** : dessine des rectangles vides
- **drawRoundRect(x1, y1, w, h)** : dessine des rectangles vides et arrondis
- **fillRoundRect(x1, y1, w, h)** : dessine des rectangles remplis et arrondis
- **drawLine(x1, y1, x2, y2)** : dessine une ligne droite entre les coordonnées de départ et les coordonnées d'arrivée
- **drawPolygon(x[], y[], nbPoints)** : dessine des polygones vides entre les coordonnées de tous les points qui les forment
- **fillPolygon(x[], y[], nbPoints)** : des polygones remplis
- **drawString(s, x, y)** : permet d'écrire du texte à des coordonnées précises
- **setFont(font)** : permet de changer de police
- **setColor(color)** : permet de changer de couleur
- **drawImage(image, x, y, obs)** : dessine une image

Il faut charger l'image grâce à trois objets : un *Image*, un *ImageIO*, un *File*

```
public void paintComponent(Graphics g)
{
    int x1 = this.getWidth() / 4;
    int y1 = this.getHeight() / 4;

    g.drawOval(x1, y1, this.getWidth() / 2, this.getHeight() / 2);
    g.fillOval(x1, y1, this.getWidth() / 2, this.getHeight() / 2);

    g.drawRect(10, 10, 50, 60);
    g.fillRect(65, 65, 30, 40);

    g.drawRoundRect(10, 10, 30, 50, 10, 10);
    g.fillRoundRect(55, 65, 55, 30, 5, 5);

    g.drawLine(0, 0, this.getWidth(), this.getHeight());
    g.drawLine(0, this.getHeight(), this.getWidth(), 0);

    int x[] = {20, 30, 50, 60, 60, 50, 30, 20};
    int y[] = {30, 20, 20, 30, 50, 60, 60, 50};
    g.drawPolygon(x, y, 8);

    int x2[] = {50, 60, 80, 90, 90, 80, 60, 50};
    int y2[] = {60, 50, 50, 60, 80, 90, 90, 80};
    g.fillPolygon(x2, y2, 8);

    g.drawString("Tiens ! Le Site du Zéro !", 10, 20);

    Font font = new Font("Courier", Font.BOLD, 20);
    g.setFont(font);
    g.setColor(Color.red);
    g.drawString("Tiens ! Le Site du Zéro !", 30, 40);

    try
    {
        Image img = ImageIO.read(new File("images.jpg"));
        g.drawImage(img, 0, 0, this);

        // pour une image de fond
        //g.drawImage(img, 0, 0, this.getWidth(), this.getHeight(), this);
    }
    catch (IOException e)
    { e.printStackTrace(); }
}
```

- L'objet **Graphics2D** : pour des dessins plus évolués

Pour utiliser cet objet il suffit de caster l'objet *Graphics* en *Graphics2D*

```
Graphics2D g2d = (Graphics2D) g
```

Méthodes utiles :

- **GradientPaint(xc1, yc1, color1, xc2, yc2, color2, true)** :

permet de peindre des objets avec des dégradés de couleurs

xc1, yc1 : coordonnées où doit commencer la première couleur

xc2, yc2 : coordonnées où doit commencer la seconde couleur

true : si le dégradé doit se répéter

```
public void paintComponent(Graphics g)
{
    Graphics2D g2d = (Graphics2D)g;

    // dégradé oblique
    GradientPaint gp = new GradientPaint(0, 0, Color.RED, 30, 30, Color.cyan, true);
    g2d.setPaint(gp);
    g2d.fillRect(0, 0, this.getWidth(), this.getHeight());

    // dégradé verticale
    GradientPaint gp = new GradientPaint(0, 0, Color.RED, 0, 30, Color.cyan, true);
    g2d.setPaint(gp);
    g2d.fillRect(0, 0, this.getWidth(), this.getHeight());
}
```

## Le fil rouge : une animation

En utilisant les classes JFrame et JPanel nous allons pouvoir créer un **effet de déplacement** : il va falloir modifier les coordonnées d'un rond et de forcer le panneau à se redessiner (dans une boucle).

```
// le panneau
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;

public class Panneau extends JPanel
{
    private int posX = -50;
    private int posY = -50;

    public void paintComponent(Graphics g)
    {
        // nettoie l'écran à chaque appel
        g.setColor(Color.white);
        g.fillRect(0, 0, this.getWidth(), this.getHeight());

        // dessine le nouveau rond
        g.setColor(Color.red);
        g.fillOval(posX, posY, 50, 50);
    }

    public int getPosX()
    {
        return posX;
    }

    public int getPosY()
    {
        return posY;
    }

    public void setPosX(int posX)
    {
        this.posX = posX;
    }

    public void setPosY(int posY)
    {
        this.posY = posY;
    }
}
```

```
// la fenêtre
import java.awt.Dimension;
import javax.swing.JFrame;

public class Fenetre extends JFrame
{
    private Panneau pan = new Panneau();

    public Fenetre()
    {
        this.setTitle("Animation");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);
        this.setContentPane(pan);
        this.setVisible(true);
        go();
    }

    private void go()
    {
        for ( int i = -50; i < pan.getWidth(); i++ )
        {
            int x = panPosX(), y = panPosY();
            x++;
            y++;
            panPosX(x);
            panPosY(y);
            pan.repaint();

            try
            {
                Thread.sleep(10); // pause de 10ms
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

Pour une boucle infinie :

```
private void go()
{
    while (true)
    {
        int x = pan.getPosX(), y = pan.getPosY();

        if ( x == pan.getWidth() || y == pan.getHeight() )
        {
            x = -50; y = -50;
        }

        x++; y++;
        pan.setPosX(x);
        pan.setPosY(y);
        pan.repaint();
    }

    try
    {
        Thread.sleep(10);
    }
    catch (InterruptedException e)
    {
        e.printStackTrace();
    }
}
```

Pour le rebondi :

```
private void goRebondi()
{
    int x = pan.getPosX(), y = pan.getPosY();
    boolean backX = false;
    boolean backY = false;

    while(true)
    {
        if ( x < 1 )                      backX = false;
        if ( x > pan.getWidth() - 50 )     backX = true;

        if ( y < 1 )                      backY = false;
        if ( y > pan.getHeight() - 50 )    backY = true;

        if ( ! backX )      pan.setPosX(++x);
        else                  pan.setPosX(--x);

        if ( ! backY )      pan.setPosY(++y);
        else                  pan.setPosY(--y);

        pan.repaint();
    }
}
```

## Positionner des boutons

- Un bouton s'utilise avec la classe **JButton** présente dans le package **javax.swing**. Pour ajouter un bouton dans une fenêtre, vous devez utiliser la méthode **add()** du content pane. La classe **Fenetre** héritant de **JFrame** va contenir un objet **JPanel** et un objet **JButton**.

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Fenetre extends JFrame
{
    private JPanel pan = new JPanel();
    private JButton bouton = new JButton("Mon bouton");

    public Fenetre()
    {
        this.setTitle("Animation");
        this.setSize(300, 150);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        pan.add(bouton);                                // ajout du bouton à notre content pane
        this.setContentPane(pan);

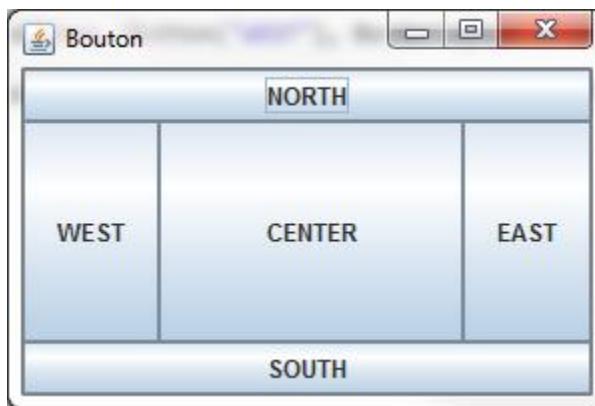
        this.setVisible(true);
    }
}
```

- Il existe des objets permettant de **positionner les composants sur un content pane** ou un conteneur : les **layout managers** (se trouvent dans le package **java.awt**).

On définit un layout sur un conteneur grâce à la méthode **setLayout()**.

Il existe plusieurs types de layout managers :

- L'objet **BorderLayout** : le layout manager par défaut du content pane d'un objet **JFrame**



Très pratique pour positionner les composants de façon simple, par rapport à une position cardinale du conteneur.

La fenêtre dans l'exemple est composée de cinq JButton positionnés aux cinq endroits différents que propose un BorderLayout. Voici le code de cette fenêtre :

```
import java.awt.BorderLayout;
import javax.swing.JButton;
import javax.swing.JFrame;

public class Fenetre extends JFrame
{
    public Fenetre()
    {
        this.setTitle("Bouton");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        // on définit le layout à utiliser sur le content pane
        this.setLayout(new BorderLayout());

        // on ajoute des bouton au content pane de la JFrame : centre, nord, sud, ouest, est
        this.getContentPane().add(new JButton("CENTER"), BorderLayout.CENTER);
        this.getContentPane().add(new JButton("NORTH"), BorderLayout.NORTH);
        this.getContentPane().add(new JButton("SOUTH"), BorderLayout.SOUTH);
        this.getContentPane().add(new JButton("WEST"), BorderLayout.WEST);
        this.getContentPane().add(new JButton("EAST"), BorderLayout.EAST);

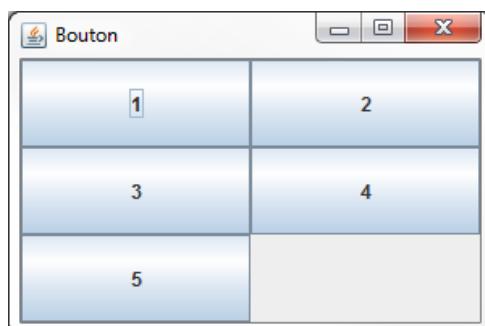
        this.setVisible(true);
    }
}
```

On utilise un 2eme paramètre avec la méthode **add()** : attribut statique BorderLayout.

- L'objet **GridLayout** : permet d'ajouter des composants suivant une grille définie par un nombre de lignes et de colonnes.

Les éléments sont disposés à partir de la case située en haut à gauche.

Pour une grille de 3 lignes et 2 colonnes :



```

import java.awt.GridLayout;
import javax.swing.JButton;
import javax.swing.JFrame;

public class Fenetre extends JFrame
{
    public Fenetre()
    {
        this.setTitle("Bouton");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        // on définit le layout à utiliser sur le content pane : 3 lignes sur 2 colonnes
this.setLayout(new GridLayout(3, 2));

        // on ajoute le bouton au content pane de la JFrame
        this.getContentPane().add(new JButton("1"));
        this.getContentPane().add(new JButton("2"));
        this.getContentPane().add(new JButton("3"));
        this.getContentPane().add(new JButton("4"));
        this.getContentPane().add(new JButton("5"));

        this.setVisible(true);
    }
}

```

Ici nous utilisons un autre layout manager et nous n'avons pas besoin de définir le positionnement lors de l'ajout du composant avec la méthode add.

Il est également possible d'ajouter de l'espace entre les colonnes et les lignes :

```

GridLayout gl = new GridLayout(3, 2);
gl.setHgap(5); // cinq pixels d'espace entre les colonnes (H comme Horizontal)
gl.setVgap(5); // cinq pixels d'espace entre les lignes (V comme Vertical)

```

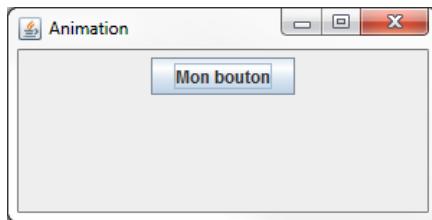
// ou en abrégé :

```
GridLayout gl = new GridLayout(3, 2, 5, 5);
```

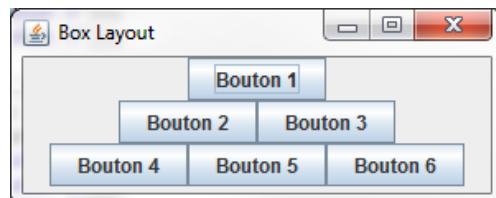


- L'objet **FlowLayout** : le layout manager par défaut d'un objet **JPanel**

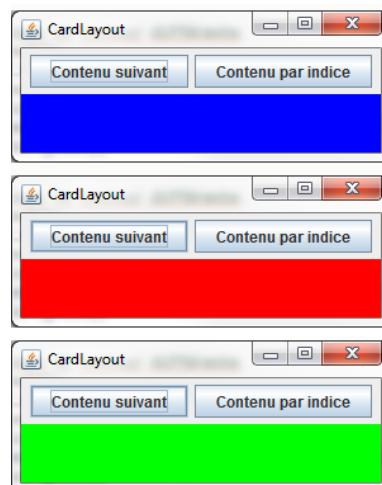
Le plus simple à utiliser. Il se contente de centrer les composants dans le conteneur.



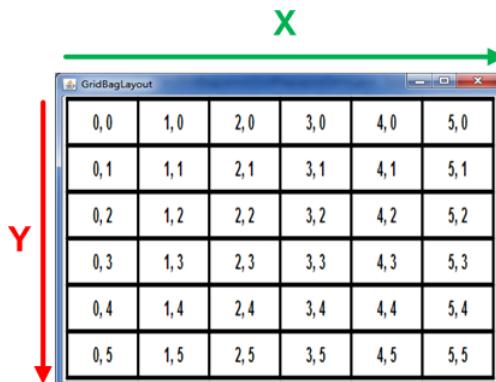
- L'objet **BoxLayout** : pour ranger ces composants à la suite, sur une ligne ou sur une colonne



- L'objet **CardLayout** : pour gérer les conteneurs comme un tas de cartes (les unes sur les autres), et basculer d'un contenu à l'autre

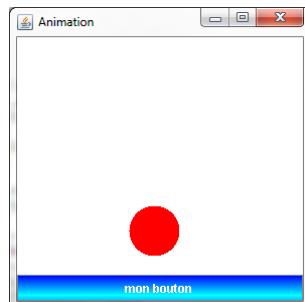


- L'objet **GridBagLayout** : se présente sous la forme d'une grille à la façon d'un tableau Excel. Vos devez positionner vos composants en vous servant des coordonnées des cellules, définir les marges et la façon dont vos composant se répliquent dans les cellules



## Interagir avec des boutons

- Une classe Bouton personnalisée



Classe Bouton :

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.GradientPaint;
import javax.swing.JButton;

public class Bouton extends JButton
{
    private String name;

    public Bouton(String name)
    {
        super(name);
        this.name = name;
    }

    public void paintComponent(Graphics g)
    {
        Graphics2D g2d = (Graphics2D)g;
        GradientPaint gp = new GradientPaint(0, 0, Color.blue, 0, 20, Color.cyan, true);

        g2d.setPaint(gp);
        g2d.fillRect(0, 0, this.getWidth(), this.getHeight());

        g2d.setColor(Color.white);
        g2d.drawString(this.name, 3 * this.getWidth() / 8, this.getHeight() / 2 + 5);
    }
}
```

Classe Panneau :

```
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;
```

```

public class Panneau extends JPanel
{
    protected int posX = -50;
    protected int posY = -50;

    public int getPosX() { return posX; }
    public int getPosY() { return posY; }
    public void setPosX(int posX) { this.posX = posX; }
    public void setPosY(int posY) { this.posY = posY; }

    public void paintComponent(Graphics g)
    {
        g.setColor(Color.white); // nettoie l'écran
        g.fillRect(0, 0, this.getWidth(), this.getHeight());

        g.setColor(Color.red); // dessine le rond rouge
        g.fillOval(posX, posY, 50, 50);
    }
}

```

Classe Fenetre :

```

import java.awt.Color; import javax.swing.JPanel;
import java.awt.BorderLayout; import javax.swing.JButton;
import javax.swing.JFrame; import javax.swing.JLabel;

public class Fenetre extends JFrame
{
    private Bouton bouton = new Bouton("Mon bouton");
    private Panneau panneau = new Panneau();
    private JPanel container = new JPanel();
    private JLabel label = new JLabel("Mon label");

    public Fenetre()
    {
        this.setTitle("Animation");
        this.setSize(300, 400);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setLayout(new BorderLayout());
        container.setBackground(Color.white);
        container.add(label, BorderLayout.NORTH);
        container.add(bouton, BorderLayout.SOUTH);
        container.add(panneau, BorderLayout.CENTER);

        this.setContentPane(container);
        this.setVisible(true);
        goRebondi();
    }
    private void goRebondi() { ... panneau.repaint(); ... }
}

```

- Pour interagir avec un composant avec la **souris** il faut implémenter l'**interface MouseListener**.

Pour détecter les actions de la souris, il faut :

- implémenter l'interface **MouseListener** dans la classe Bouton
- préciser à la classe qu'elle devra **se** tenir au courant de ses changements d'état par rapport à la souris

**Notre classe va donc s'écouter** : dès que le bouton obtiendra des informations concernant les actions effectuées par la souris, il indiquera à lui-même ce qu'il doit effectuer.

Cela est réalisable grâce à la méthode **addMouseListener(MouseListener obj)** (son paramètre ici, **this**). Dans ce cas, la méthode **repaint()** est appelée de façon implicite : lorsqu'un événement est déclenché, notre objet se redessine automatiquement !

Changements dans la **classe Bouton**

```
public class Bouton extends JButton implements MouseListener
{
    private String name;
    private Color couleur = Color.blue;

    public Bouton(String str)
    {
        super(str);
        this.name = str;

        // notre objet va s'écouter : dès qu'un événement de la souris sera intercepté, il en sera averti
        this.addMouseListener(this);
    }

    public void paintComponent(Graphics g)
    {
        ...
        g2d.setColor(couleur);
        g2d.fillRect(0, 0, this.getWidth(), this.getHeight());
        ...
    }

    // méthode appelée lors du clic de souris
    public void mouseClicked(MouseEvent event) { }

    // méthode appelée lors du survol de la souris
    public void mouseEntered(MouseEvent event) { couleur = Color.yellow; }

    // méthode appelée lorsque la souris sort de la zone du bouton
    public void mouseExited(MouseEvent event) { couleur = Color.green; }

    // méthode appelée lorsque l'on presse le bouton gauche de la souris
    public void mousePressed(MouseEvent event) { couleur = Color.yellow; }

    // méthode appelée lorsque l'on relâche le clic de souris
    public void mouseReleased(MouseEvent event) { couleur = Color.red; }
}
```

Amélioration possible : vérifier la position de la souris lorsqu'on la relâche

```
public void mouseReleased(MouseEvent event)
{
    // nous changeons la couleur de fond de notre bouton pour orange lorsque nous relâchons le clic
    // seulement si la souris est toujours sur le bouton
    if ( ( event.getY() > 0 && event.getY() < this.getHeight() ) &&
        ( event.getX() > 0 && event.getX() < this.getWidth() ) )
        { couleur = Color.orange; }
    else
        { couleur = Color.blue; } // si on se trouve à l'extérieur, on dessine le fond par défaut
}
```

- Afin de gérer les différentes actions à effectuer selon le bouton sur lequel on clique, nous allons utiliser l'**interface ActionListener** dans la classe **Fenetre** ( but : faire en sorte que lorsque l'on clique sur le bouton, il se passe quelque chose dans notre application)

Exemple : modifier le comportement d'une étiquette JLabel (texte, alignement, police, couleur)

```
public class Fenetre extends JFrame implements ActionListener
{
    ...
    private int compteur = 0;

    public Fenetre()
    {
        ...
        label.setText("Mon premier JLabel");
        label.setFont(new Font("Tahoma", Font.BOLD, 16));
        label.setForeground(Color.blue);
        label.setHorizontalAlignment(JLabel.CENTER);
        container.add(label, BorderLayout.NORTH);
        ...
        // nous ajoutons notre fenêtre à la liste des auditeurs de notre bouton
        bouton.addActionListener(this);
        container.add(label, BorderLayout.SOUTH);
        ...
    }

    public void actionPerformed(ActionEvent arg0)
    {
        this.compteur++;
        label.setText("Vous avez cliqué " + this.compteur + " fois");
    }
}
```

- Lorsque vous implémentez une **interface** <...>**Listener**, vous indiquez à votre classe qu'elle doit se préparer à observer des événements du type de l'interface. Vous devez donc spécifier **qui doit observer et ce que la classe doit observer** grâce aux méthodes de type **add<...>Listener(<...>Listener)**.

L'interface utilisée dans ce chapitre est **ActionListener** issue du package **java.awt**.

La méthode à implémenter de cette interface est **actionPerformed()**.

- Ecouter deux boutons :

```
public class Fenetre extends JFrame implements ActionListener
{
    private JButton bouton1 = new JButton("bouton 1");
    private JButton bouton2 = new JButton("bouton 2");
    ...

    public Fenetre()
    {
        ...
        JPanel south = new JPanel();
        bouton1.addActionListener(this);
        bouton2.addActionListener(this);
        south.add(bouton1);
        south.add(bouton2);
        container.add(south, BorderLayout.SOUTH);
        ...
    }

    public void actionPerformed(ActionEvent arg0)
    {
        if( arg0.getSource() == bouton1 )
            label.setText("Vous avez cliqué sur le bouton 1");
        if( arg0.getSource() == bouton2 )
            label.setText("Vous avez cliqué sur le bouton 2");
    }
}
```

- Une **classe interne** est une classe se trouvant à l'intérieur d'une classe.

En Java, l'on peut créer des classes internes. Cela consiste à déclarer une classe à l'intérieur d'une autre classe.

En effet, les classes internes possèdent tous les avantages des classes normales, de l'héritage d'une superclasse à l'implémentation d'une interface. Elles bénéficient donc du polymorphisme et de la covariance des variables. En outre, elles ont l'avantage d'avoir accès aux attributs et méthodes de la classe dans laquelle elles sont déclarées (de sa classe externe).

Dans notre cas : cela permet de créer une implémentation de l'interface **ActionListener** détachée de notre classe **Fenetre**, mais pouvant utiliser ses attributs.

Grâce à cela, nous pourrons concevoir une classe spécialisée dans l'écoute des composants et qui effectuera un travail bien déterminé. Dans notre exemple, nous créerons deux classes internes implémentant chacune l'interface **ActionListener** et redéfinissant la méthode **actionPerformed()** :

- Bouton1Listener : écoutera le premier bouton
- Bouton2Listener : écoutera le deuxième bouton

Ensute il ne nous reste plus qu'à indiquer à chaque bouton "qui l'écoute" grâce à la méthode **addActionListener()**.

Classe Fenetre mise-à-jour

```
import java.awt.Font;
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class Fenetre extends JFrame
{
    private Panneau pan = new Panneau();
    private JButton bouton1 = new JButton("bouton 1");
    private JButton bouton2 = new JButton("bouton 2");
    private JPanel container = new JPanel();
    private JLabel label = new JLabel("Le JLabel");
    private int compteur = 0;

    public Fenetre()
    {
        this.setTitle("Animation");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);

        container.setLayout(new BorderLayout());
        container.setBackground(Color.white);

        JPanel south = new JPanel();
        bouton1.addActionListener(new Bouton1Listener());
        bouton2.addActionListener(new Bouton2Listener());
        south.add(bouton1);
        south.add(bouton2);
        container.add(south, BorderLayout.SOUTH);
    }
}
```

```

        Font police = new Font("Tahoma", Font.BOLD, 16);
        label.setFont(police);
        label.setForeground(Color.blue);
        label.setHorizontalAlignment(JLabel.CENTER);
        container.add(label, BorderLayout.NORTH);

        container.add(pan, BorderLayout.CENTER);

        this.setContentPane(container);
        this.setVisible(true);

        goRebondi();
    }

    private void goRebondi() { ... }

    // classe écoutant notre premier bouton
    class Bouton1Listener implements ActionListener
    {
        public void actionPerformed(ActionEvent arg0)
        {
            label.setText("Vous avez cliqué sur le bouton 1");
        }
    }

    // classe écoutant notre second bouton
    class Bouton2Listener implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            label.setText("Vous avez cliqué sur le bouton 2");
        }
    }
}

```

On peut demander même à plusieurs classes d'écouter un seul bouton; il faut seulement appeler la méthode `addActionListener()` pour chacune de ces classes.

- Les **classes anonymes** sont le plus souvent utilisées pour la gestion d'événements ponctuels, lorsque créer une classe pour un seul traitement est trop lourd.

Exemple :

```

 JButton bouton = new JButton("Contenu suivant");
 bouton.addActionListener(new ActionListener()
 {
    public void actionPerformed(ActionEvent event)
    { ... }
 });

```

Particularité : l'écouteur **n'écoutera que ce composant**.

Il existe aucune déclaration de classe et nous instancions une interface par l'instruction **new ActionListener()**.

```
JButton bouton = new JButton("Contenu suivant");
bouton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event) {
        cl.next(content);
    }
});
```

Procéder de cette manière revient à créer une classe fille (interne ... ?) **sans être obligé de créer cette classe de façon explicite**. L'héritage se produit automatiquement.

Sachez aussi que les classes anonymes peuvent être utilisées pour implémenter des classes abstraites.

Les classes anonymes sont **soumises aux mêmes règles** que les classes "normales" :

- utilisation des méthodes non redéfinies de la classe mère ;
- obligation de redéfinir **toutes les méthodes** d'une interface ;
- obligation de redéfinir les méthodes abstraites d'une classe abstraite.

Cependant, ces classes possèdent des **restrictions** à cause de leur rôle et de leur raison d'être :

- elles ne peuvent pas être déclarées abstract ;
- elles ne peuvent pas non plus être déclarées static ;
- elles ne peuvent pas définir de constructeur ;
- elles sont automatiquement déclarées **final** : on ne peut dériver de cette classe, l'héritage est donc impossible !

- **Contrôler son animation** : lancement et arrêt

Il va falloir modifier un peu le code de la classe **Fenetre**.

Les boutons auront associé le texte "Go" et "Stop".

Pour éviter d'interrompre l'animation alors qu'elle n'est pas lancée ou de l'animer quand elle l'est déjà, nous allons **activer et désactiver les boutons**, chacun à son tour :

- au lancement, le bouton Go ne sera pas cliquable, alors que le bouton Stop oui
- si l'animation est interrompue, le bouton Stop n'est plus cliquable, mais le bouton Go oui

La méthode gérant le changement d'état d'un bouton est **setEnabled(boolean)** :

```
JButton bouton = new JButton("bouton");
bouton.setEnabled(false); // le bouton n'est plus cliquable
bouton.setEnabled(true); // le bouton est de nouveau cliquable
```

À l'exécution, vous remarquez que :

- le bouton Go n'est pas cliquable et l'autre l'est ;
- l'animation se lance ;
- l'animation s'arrête lorsque l'on clique sur le bouton Stop ;
- le bouton Go devient alors cliquable ;
- lorsque vous cliquez dessus, l'animation ne se relance pas !

#### Explication :

Un **thread** est lancé au démarrage de l'application : c'est le processus principal du programme.

Au démarrage, l'animation est donc lancée dans le même **thread** que notre objet **Fenetre**. Lorsque nous lui demandons de s'arrêter, aucun problème : les ressources mémoire sont libérées, on sort de la boucle infinie et l'application continue à fonctionner.

Mais lorsque nous redemandons à l'animation de se lancer, l'instruction se trouvant dans la méthode **actionPerformed()** appelle la méthode **go()** et, étant donné que nous nous trouvons à l'intérieur d'une boucle infinie, nous restons dans la méthode **go()** et ne sortons plus de la méthode **actionPerformed()**.

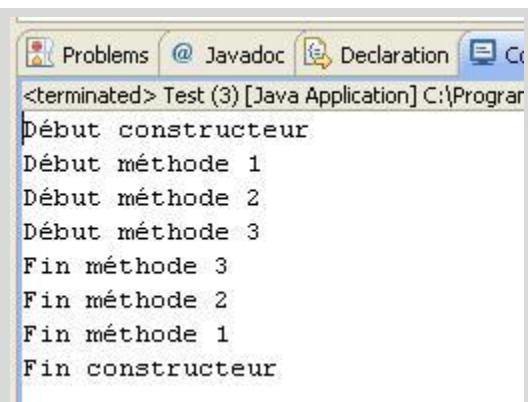
La JVM traite les méthodes appelées en utilisant une **pile qui définit leur ordre d'exécution**. Une méthode est empilée à son invocation, mais n'est dépilerée que lorsque toutes ses instructions ont fini de s'exécuter.

```
public class TestPile
{
    public TestPile()
    {
        System.out.println("Début constructeur");
        methode1();
        System.out.println("Fin constructeur");
    }

    public void methode1()
    {
        System.out.println("Début méthode 1");
        methode2();
        System.out.println("Fin méthode 1");
    }

    public void methode2()
    {
        System.out.println("Début méthode 2");
        methode3();
        System.out.println("Fin méthode 2");
    }

    public void methode3()
    {
        System.out.println("Début méthode 3");
        System.out.println("Fin méthode 3");
    }
}
```



```
Problems @ Javadoc Declaration C
<terminated> Test (3) [Java Application] C:\Programme\NetBeans\Projects\JavaTest\src\javatest\

Début constructeur
Début méthode 1
Début méthode 2
Début méthode 3
Fin méthode 3
Fin méthode 2
Fin méthode 1
Fin constructeur
```

Afin de pallier ce problème, nous allons utiliser **un nouveau thread**. Grâce à cela, la méthode **go()** se trouvera dans une pile à part.

- Pour détecter les événements qui surviennent sur votre composant, Java utilise ce qu'on appelle le **design pattern observer** (technique de programmation). Le pattern observer permet d'utiliser des objets faiblement couplés (qui dépendent l'un de l'autre). Grâce à ce pattern, les objets restent indépendants.

Le couplage entre objets est l'un des problèmes principaux relatifs à la réutilisation des objets.

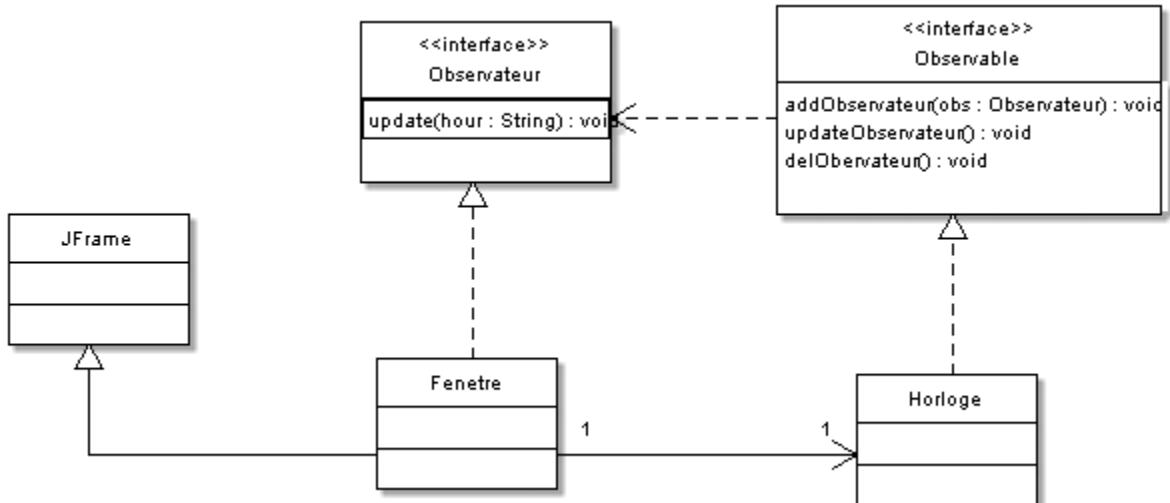
C'est là que le pattern observer entre en jeu :

- il fait communiquer des objets entre eux sans qu'ils se connaissent réellement
- c'est un bon moyen d'éviter le couplage d'objets
- tout se fait automatiquement !

**Exemple** : afficher l'heure d'un horloge dans un label

L'horloge digitale doit pouvoir avertir l'objet servant à afficher l'heure lorsqu'il doit rafraîchir son affichage toutes les secondes. Elle ne se contentera pas d'avertir un seul objet que sa valeur a changé : elle pourra en effet mettre plusieurs observateurs au courant !

Grâce à ce schéma, vous pouvez comprendre que notre **objet** défini comme **observable** pourra être surveillé par plusieurs objets : il s'agit d'une relation dite de un à plusieurs vers l'objet **Observateur**.



La deuxième interface - celle dédiée à l'objet Horloge - possède trois méthodes :

- une permettant d'ajouter des observateurs (nous allons donc gérer une collection d'observateurs) ;
- une permettant de retirer les observateurs ;
- enfin, une mettant à jour les observateurs.

Grâce à cela, nos objets ne sont plus liés par leurs types, mais par leurs interfaces ! L'interface qui apportera les méthodes de mise à jour, d'ajout d'observateurs, etc. travaillera donc avec des objets de type Observateur.

Le couplage ne s'effectue plus directement, il s'opère par le biais de ces interfaces. Ici, il faut que nos deux interfaces soient couplées pour que le système fonctionne.

Voici comment fonctionnera l'application :

- nous instancierons la classe **Horloge** dans notre classe **Fenetre** ;
- cette dernière implémentera l'**interface Observateur** ;
- notre objet Horloge, implémentant l'**interface Observable**, préviendra les objets spécifiés de ses changements ;
- nous informerons l'horloge que notre fenêtre l'observe ;
- à partir de là, notre objet Horloge fera le reste : à chaque changement, nous appellerons la méthode mettant tous les observateurs à jour.

Code source :

```
Observateur.java
```

```
package com.sdz.observer;  
public interface Observateur  
{  
    public void update(String hour);  
}
```

```
Observer.java
```

```
package com.sdz.observer;  
public interface Observable  
{  
    public void addObservateur(Observateur obs);  
    public void updateObservateur();  
    public void delObservateur();  
}
```

```
Horloge.java
```

```
package com.sdz.model;  
import java.util.ArrayList;  
import java.util.Calendar;  
import com.sdz.observer.Observable;  
import com.sdz.observer.Observateur;  
  
public class Horloge implements Observable  
{  
    private Calendar cal; // on récupère l'instance d'un calendrier pour connaître l'heure actuelle  
    private String hour = "";  
    private ArrayList<Observateur> listObservateur = new ArrayList<Observateur>();  
  
    public void run()  
    {  
        while(true)  
        {  
            this.cal = Calendar.getInstance();  
            this.hour = this.cal.get(Calendar.HOUR_OF_DAY) + " : "  
                + " : " + ( this.cal.get(Calendar.MINUTE) < 10  
                    ? "0" + this.cal.get(Calendar.MINUTE)  
                    : this.cal.get(Calendar.MINUTE) )  
                + " : " + ( this.cal.get(Calendar.SECOND) < 10  
                    ? "0" + this.cal.get(Calendar.SECOND)  
                    : this.cal.get(Calendar.SECOND) );  
        }  
    }  
}
```

```

// on avertit les observateurs que l'heure a été mise à jour
this.updateObservateur();

try
{
    Thread.sleep(1000);
}
catch (InterruptedException e)
{
    e.printStackTrace();
}
}

public void addObservateur(Observateur obs)           // ajoute un observateur à la liste
{
    this.listObservateur.add(obs);
}

public void delObservateur()                          // retire tous les observateurs de la liste
{
    this.listObservateur = new ArrayList<Observateur>();
}

public void updateObservateur()          // avertit les observateurs que l'objet observable a changé
{
    for(Observateur obs : this.listObservateur )
        obs.update(this.hour);
}
}

```

```

Fenetre.java
package com.sdz.vue;
import java.awt.BorderLayout;
import java.awt.Font;
import javax.swing.JFrame;
import javax.swing.JLabel;
import com.sdz.model.Horloge;
import com.sdz.observer.Observateur;

public class Fenetre extends JFrame
{
    private JLabel label = new JLabel();
    private Horloge horloge;

    public Fenetre()
    {
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLocationRelativeTo(null);
        this.setResizable(false);
        this.setSize(200, 80);
    }
}

```

```

        this.horloge = new Horloge(); // on initialise l'horloge
        this.horloge.addObserver(new Observateur() // on place un écouteur sur l'horloge
        {
            public void update(String hour)
            {
                label.setText(hour);
            }
        });
    }

    // on initialise le JLabel
    Font police = new Font("DS-digital", Font.TYPE1_FONT, 30);
    this.label.setFont(police);
    this.label.setHorizontalAlignment(JLabel.CENTER);

    this.getContentPane().add(this.label, BorderLayout.CENTER);
    this.setVisible(true);

    this.horloge.run();
}

// méthode main() lançant le programme
public static void main(String[] args)
{
    Fenetre fen = new Fenetre();
}
}

```

Lorsque l'heure change, la méthode **updateObserver()** est invoquée. Celle-ci parcourt la collection d'objets Observateur et appelle sa méthode **update(String hour)**. La méthode étant redéfinie dans notre classe Fenetre afin de mettre JLabel à jour, l'heure s'affiche !

## Exécuter des tâches simultanément

- Un nouveau thread permet de créer une nouvelle pile d'exécution.

Les threads sont des fils d'exécution de notre programme. Lorsque nous en créons plusieurs, nous pouvons exécuter des tâches simultanément.

Il existe deux façons de créer un nouveau thread :

- créer une classe **héritant de la classe Thread**
- créer une implémentation de l'**interface Runnable** et **instancier un objet Thread** avec l'implémentation de cette interface

La classe **Thread** et l'interface **Runnable** se trouvent dans le package **java.lang**, aucun import spécifique n'est donc nécessaire pour leur utilisation.

La méthode **main est le thread principal** de notre application :

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.println("Thread principal"+ Thread.currentThread().getName());
    }
}
```

Une fois instancié, un thread attend son lancement. Dès que c'est fait, il invoque sa méthode **run()** qui va lui permettre de connaître les tâches qu'il a à effectuer.

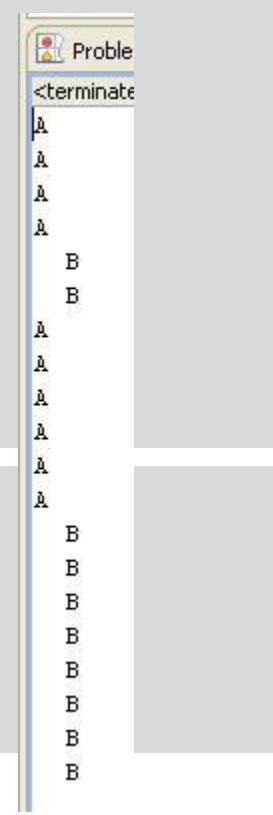
Les opérations à effectuer dans une autre pile d'exécution sont à placer dans la méthode **run()**, qu'il s'agisse d'une classe héritant de Thread ou d'une implémentation Runnable.

- **Exemple** des threads créés avec l'héritage

```
public class TestThread extends Thread
{
    public TestThread(String name)
    {
        super(name);
    }

    public void run()
    {
        for(int i = 0; i < 10; i++)
            System.out.println(this.getName());
    }
}
```

```
public class Test{
    public static void main(String[] args){
        TestThread t1 = new TestThread("A");
        TestThread t2 = new TestThread("  B");
        t1.start();
        t2.start();
    }
}
```



- Un thread se lance lorsqu'on invoque la méthode **start()**. Cette dernière invoque automatiquement la méthode **run()**.
- L'**ordre d'exécution est souvent aléatoire**, car Java utilise un **ordonnanceur**. Si vous utilisez plusieurs threads dans une application, ceux-ci ne s'exécutent pas toujours en même temps ! En fait, l'ordonnanceur gère les threads de façon aléatoire : il va en faire tourner un pendant un certain temps, puis un autre, puis revenir au premier, etc., jusqu'à ce qu'ils soient terminés. Lorsque l'ordonnanceur passe d'un thread à un autre, le thread interrompu est mis en sommeil tandis que l'autre est en éveil.

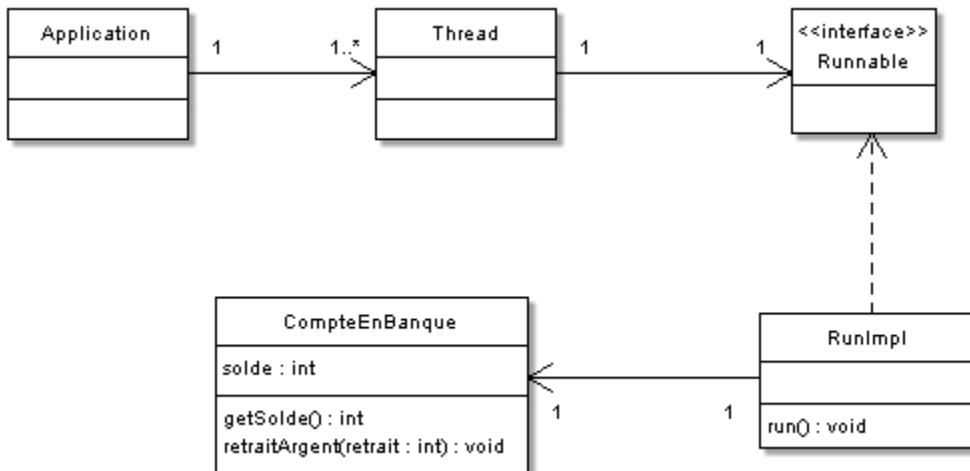
**Notez** qu'avec les **processeurs multi-coeurs** aujourd'hui, il est désormais possible d'exécuter deux tâches exactement en même temps. Tout dépend donc de votre ordinateur.

- Les threads peuvent présenter **plusieurs états**:
  - **NEW** : lors de sa création
  - **RUNNABLE** : lorsqu'on invoque la méthode **start()**; le thread est prêt à travailler
  - **BLOCKED** : lorsque l'ordonnanceur place un thread en sommeil pour en utiliser un autre
  - **WAITING** : lorsque le thread est en attente indéfinie
  - **TIMED\_WAITING** : lorsque le thread est en pause (ex : à cause de la méthode **sleep()**)
  - **TERMINATED** : lorsque le thread a effectué toutes ses tâches; on dit aussi qu'il est "mort". Il ne peut plus être relancé avec la méthode **start()**

Le thread principal crée un second thread qui se lance et construit une pile dont la base est sa méthode **run()**; celle-ci appelle une méthode, l'empile, effectue toutes les opérations demandées, et une fois qu'elle a terminé, elle dépile cette dernière. La méthode **run()** prend fin, **la pile est alors détruite**.

Pour récupérer l'état d'un thread faut utiliser la méthode **getState()**.

- **Exemple** de threads utilisant l'interface Runnable



Créer un objet **CompteEnBanque** contenant une somme d'argent par défaut, une méthode pour retirer de l'argent et une méthode retournant le solde. Avant de retirer de l'argent il faut vérifier que nous ne sommes pas à découvert.

Codes source:

```
RunImpl.java
public class RunImpl implements Runnable
{
    private CompteEnBanque cb;

    public RunImpl(CompteEnBanque cb)
    { this.cb = cb; }

    public void run()
    {
        for ( int i = 0; i < 25; i++ )
        {
            if ( cb.getSolde() > 0 )
            {
                cb.retraitArgent(2);
                System.out.println("Retrait effectué");
            }
        }
    }
}
```

```
CompteEnBanque.java
public class CompteEnBanque
{
    private int solde = 100;

    public int getSolde()
    {
        if(this.solde < 0) System.out.println("Vous êtes à découvert !");
        return this.solde;
    }

    public void retraitArgent(int retrait)
    {
        solde = solde - retrait;
        System.out.println("Solde = " + solde);
    }
}
```

```
Test.java
public class Test
{
    public static void main(String[] args)
    {
        CompteEnBanque cb = new CompteEnBanque();
        Thread t = new Thread(new RunImpl(cb));
        t.start();
    }
}
```

```
<terminated> Test (4) [Java Application]
Retrait effectué
Solde = 98
Retrait effectué
Solde = 96
Retrait effectué
Solde = 94
Retrait effectué
Solde = 92
Retrait effectué
Solde = 90
Retrait effectué
Solde = 88
Retrait effectué
Solde = 86
Retrait effectué
Solde = 84
```

- Pour **synchroniser des threads** il faut indiquer à la JVM qu'un thread est en train d'utiliser des données qu'un autre thread est susceptible d'altérer. Ainsi, lorsque l'ordonnanceur met en sommeil un thread qui traitait des données utilisables par un autre thread, **ce premier thread garde la priorité sur les données** et tant qu'il n'a pas terminé son travail, les autres threads n'ont pas la possibilité d'y toucher.

Pour protéger l'intégrité des données accessibles à plusieurs threads, utilisez le mot clé **synchronized** dans la **déclaration de vos méthodes**.

Exemple : méthode retraitArgent() synchronisée

```
public class CompteEnBanque
{
    ...
    public synchronized void retraitArgent(int retrait)
    {
        solde = solde - retrait;
        System.out.println("Solde = " + solde);
    }
}
```

- Contrôler l'animation graphique rond qui bouge + bouton avec des threads

**Solution** : créer un nouveau thread lorsqu'on clique sur le bouton **Go** en lui passant en paramètre une implémentation de Runnable, qui, elle, va appeler la méthode **go()**

```
import java.awt.Font;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class Fenetre extends JFrame
{
    ...
    private JButton bouton1 = new JButton("Go");
    private JButton bouton2 = new JButton("Stop");
    ...
    private Thread t;

    public Fenetre()
    { ... }

    private void go()
    { ... }
```

```

class Bouton1Listener implements ActionListener
{
    public void actionPerformed(ActionEvent arg0)
    {
        animated = true;
        t = new Thread(new PlayAnimation());
        t.start();
        bouton1.setEnabled(false);
        bouton2.setEnabled(true);
    }
}

class Bouton2Listener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        animated = false;
        bouton1.setEnabled(true);
        bouton2.setEnabled(false);
    }
}

class PlayAnimation implements Runnable
{
    public void run()
    {
        goRebondi();
    }
}

```

- Depuis Java 7 : le **pattern Fork / Join**

La version 7 de Java met à disposition des développeurs plusieurs classes qui permettent de mettre en application ce qu'on appelle « le pattern Fork/Join ».

Ce dernier n'est rien de plus que la mise en application de la technique "divide-et-impera" !

Dans certains cas, il serait bon de pouvoir **découper une tâche en plusieurs sous-tâches**, faire en sorte que ces sous-tâches **s'exécutent en parallèle** et pouvoir récupérer le résultat de tout ceci une fois que tout est terminé. C'est exactement ce qu'il est possible de faire avec ces nouvelles classes.

Avant de commencer il faut préciser qu'il y a un certain nombre de **prérequis** à cela :

- la machine qui exécutera la tâche devra posséder un processeur à **plusieurs coeurs** ;
- la tâche doit pouvoir être **découpée en plusieurs sous-tâches** ;
- s'assurer qu'il y a un réel **gain de performance** ! Dans certains cas, découper une tâche rend le traitement plus long.

**Exemple** : recherche de fichiers (simplifiée au maximum pour ne pas surcharger le code).

```
ScanException.java
public class ScanException extends Exception
{
    public ScanException(String message)
        { super(message); }
}

FolderScanner.java
import java.io.IOException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;

public class FolderScanner
{
    private Path path = null;
    private String filter = "*"; // extension
    private long result = 0;

    public FolderScanner() {}

    public FolderScanner(Path p, String f)
    {
        path = p;
        filter = f;
    }

    public long sequentialScan() throws ScanException
    {
        if ( path == null || path.equals("") ) // si le chemin n'est pas valide => exception
            throw new ScanException("Chemin à scanner non valide (vide ou null) !");

        System.out.print("Scan du dossier : " + path + " ");
        System.out.println("à la recherche des fichiers avec ext " + this.filter);

        // on liste le contenu du répertoire pour traiter les sous-dossiers
        try ( DirectoryStream<Path> listing = Files.newDirectoryStream(path) )
        {
            for ( Path nom : listing )
            {
                if ( Files.isDirectory(nom.toAbsolutePath()) ) // dossier
                {
                    FolderScanner f = new FolderScanner(nom.toAbsolutePath(), this.filter);
                    result += f.sequentialScan();
                }
            }
        } catch (IOException e) { e.printStackTrace(); }
```

```

// on filtre le contenu de ce même dossier sur le filtre défini
try ( DirectoryStream<Path> listing = Files.newDirectoryStream(path, filter) )
{
    for ( Path nom : listing )
        result++;
}
catch (IOException e) { e.printStackTrace(); }

return result;
}
}

```

```

Main.java
import java.nio.file.Path;
import java.nio.file.Paths;

public class Main
{
    public static void main(String[] args)
    {
        Path chemin = Paths.get("E:\\Mes Documents");
        String filtre = "*.psd";

        FolderScanner fs = new FolderScanner(chemin, filtre);

        try
        {
            Long start = System.currentTimeMillis();
            Long resultat = fs.sequentialScan();
            Long end = System.currentTimeMillis();
            System.out.println("Il y a " + resultat + " fichier(s) ext " + filtre);
            System.out.println("Temps de traitement : " + (end - start));
        }
        catch (ScanException e) { e.printStackTrace(); }
    }
}

```

Lorsque je lance ce code le temps de traitement est vraiment long

Il est possible de **découper le scan de chaque dossier dans une sous-tâche**. Pour ce faire, nous devons faire hériter notre classe **FolderScanner** d'une des classes permettant ce découpage.

La plateforme Java 7 nous met à disposition deux classes qui héritent de la classe abstraite **ForkJoinTask<V>** :

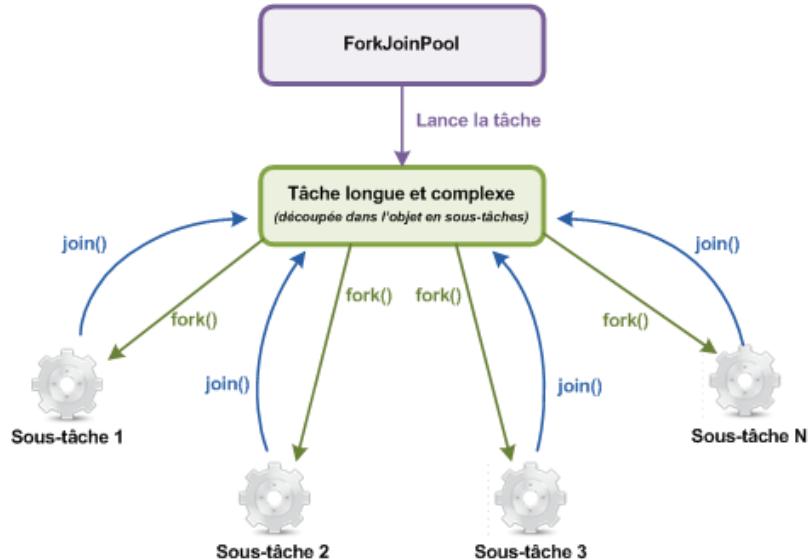
- **RecursiveAction** : classe permettant de découper une tâche ne renvoyant aucune valeur particulière. Elle hérite de `ForkJoinTask<Void>` ;
- **RecursiveTask<V>** : identique à la classe précédente mais retourne une valeur, de type `<V>`, en fin de traitement. C'est cette classe que nous allons utiliser pour pouvoir nous retourner le nombre de fichiers trouvés.

Nous allons devoir utiliser, en plus de l'objet de découpage, un objet qui aura pour rôle de superviser l'exécution des tâches et sous-tâches afin de pouvoir fusionner les threads en fin de traitement : **ForkJoinPool**.

Avant de vous présenter le code complet, voici comment ça fonctionne. Les objets qui permettent le découpage en sous-tâches fournissent trois méthodes qui permettent cette gestion :

- **compute()** : méthode abstraite à redéfinir dans l'objet héritant afin de définir le traitement à effectuer ;
- **fork()** : méthode qui crée un nouveau thread dans le pool de thread (`ForkJoinPool`) ;
- **join()** : méthode qui permet de récupérer le résultat de la méthode `compute()`.

Ces classes nécessitent que vous redéfinissiez la méthode `compute()` afin de définir ce qu'il y a à faire. La figure suivante est un schéma représentant la façon dont les choses se passent.



Concrètement, avec notre exemple, voici ce qu'il va se passer :

- nous allons lancer le scan de notre dossier ;
- notre objet qui sert à scanner le contenu va vérifier le contenu pour voir s'il n'y a pas de sous-dossiers ;
- pour chaque sous-dossier, nous allons créer une nouvelle tâche et la lancer ;
- nous allons compter le nombre de fichiers qui correspondent à nos critères dans le dossier en cours de scan ;
- nous allons récupérer le nombre de fichiers trouvés par les exécutions en tâche de fond ;
- nous allons retourner le résultat final.

## Code source

```
FolderScanner.java
public class FolderScanner extends RecursiveTask<Long>
{
    private Path path = null;
    private String filter = "*";
    private long result = 0;

    public FolderScanner() { }

    public FolderScanner(Path p, String f)
    {
        path = p;
        filter = f;
    }

    public long sequentialScan() throws ScanException { ... }

    public long parallelScan() throws ScanException
    {
        // List d'objet qui contiendra les sous-tâches créées et lancées
        List<FolderScanner> list = new ArrayList<>();

        if (path == null || path.equals("") )
            throw new ScanException("Chemin à scanner non valide (vide ou null) !");

        System.out.print("Scan du dossier : " + path + " ");
        System.out.println("à la recherche des fichiers avec ext " + this.filter);

        try ( DirectoryStream<Path> listing = Files.newDirectoryStream(path) )
        {
            for ( Path nom : listing )
            {
                if ( Files.isDirectory(nom.toAbsolutePath()) )
                {
                    // créer un nouvel objet FolderScanner qui se chargera de scanner ce dossier
                    FolderScanner f = new FolderScanner(nom.toAbsolutePath(), filter);

                    // nous l'ajoutons à la liste des tâches en cours pour récupérer le résultat plus tard
                    list.add(f);

                    // c'est cette instruction qui lance l'action en tâche de fond
                    f.fork();
                }
            }
        } catch (IOException e) { e.printStackTrace(); }
    }
}
```

```

// on compte maintenant les fichiers, correspondant au filtre, présents dans ce dossier
try ( DirectoryStream<Path> listing = Files.newDirectoryStream(path, this.filter) )
{
    for ( Path nom : listing )
        result++;
} catch (IOException e) { e.printStackTrace(); }

// enfin, nous récupérons le résultat de toutes les tâches de fond
for ( FolderScanner f : list )
    result += f.join();

// nous renvoyons le résultat final
return result;
}

protected Long compute()
{
    long resultat = 0;
    try
    {
        resultat = this.parallelScan();
    }
    catch (ScanException e) { e.printStackTrace(); }
    return resultat;
}

public long getResultat() { return this.result; }

}

```

```

classe de test
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.concurrent.ForkJoinPool;

public class Main
{
    public static void main(String[] args)
    {
        Path chemin = Paths.get("E:\\Mes Documents");
        String filtre = "*.psd";

        FolderScanner fs = new FolderScanner(chemin, filtre);

        int processeurs = Runtime.getRuntime().availableProcessors();
        ForkJoinPool pool = new ForkJoinPool(processeurs);
        Long start = System.currentTimeMillis();
        pool.invoke(fs);
    }
}

```

```
    Long end = System.currentTimeMillis();
    System.out.println("Il y a " + fs.getResultat() +" fichier(s) " + filtre);
    System.out.println("Temps de traitement : " + (end - start));
}
}
```

Pour vous donner un ordre d'idée, le scan en mode **mono thread** de mon dossier Mes Documents prend en moyenne 2 minutes alors que le temps moyen en **mode Fork/Join** est d'environ 10 secondes !

Dans cet exemple nous avons créé dynamiquement autant de threads que nécessaires pour traiter nos tâches. Vous n'aurez peut-être pas besoin de faire ceci pour des problèmes où seulement 2 ou 3 sous-tâches suffisent, surtout si vous le savez à l'avance. L'idée maîtresse revient à **définir un seuil** au delà duquel **le traitement se fera en mode Fork/Join**, sinon, il se fera dans un seul thread (il se peut que ce mode de fonctionnement soit plus lent et consommateur qu'en mode normal).

## Les champs de formulaire

- L'objet **JComboBox** : une liste de type drop-down qui permet la sélection d'un seul élément

```
public class Fenetre extends JFrame
{
    private JPanel container = new JPanel();
    private JLabel label = new JLabel("Une ComboBox");
    private JComboBox combo = new JComboBox();

    public Fenetre()
    {
        ...
        combo.setPreferredSize(new Dimension(100, 20));
        combo.setForeground(Color.blue);
        combo.addItem("Option 1");
        combo.addItem("Option 2");
        combo.addItem("Option 3");
        combo.addItem("Option 4");
        combo.addItemListener(new ItemState());
        combo.addActionListener(new ItemAction());

        JPanel top = new JPanel();
        top.add(label);
        top.add(combo);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());
        container.add(top, BorderLayout.NORTH);

        this.setContentPane(container);
        this.setVisible(true);
    }

    // classe interne implémentant l'interface ItemListener
    class ItemState implements ItemListener
    {
        public void itemStateChanged(ItemEvent e)
        { System.out.println("événement déclenché sur : " + e.getItem()); }
    }

    class ItemAction implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        { System.out.println("ActionListener : " + combo.getSelectedItem()); }
    }
}
```

Vous pouvez ajouter des éléments dans une liste avec la méthode **addItem(Object obj)**.

Vous pouvez aussi instancier une liste avec un tableau de données.

```
String[] tab = {"Option 1", "Option 2", "Option 3", "Option 4"};
combo = new JComboBox(tab);
```

Vous pouvez assigner un choix par défaut avec la méthode `setSelectedIndex(int index)`.

Vous avez aussi la possibilité de changer la couleur du texte, la couleur de fond ou la police.

Depuis Java 7, l'objet `JComboBox` peut être paramétré avec un type générique, comme ceci :

```
JComboBox<String> combo = new JComboBox<String>();
```

ce qui permet de mieux gérer le contenu des listes et ainsi mieux récupérer leurs valeurs.

Pour communiquer avec un `JComboBox` il faudra implémenter l'**interface ItemListener**. Elle est appelée automatiquement lorsqu'un élément a changé d'état. Lorsque nous cliquons sur une autre option, notre objet commence par modifier l'état de l'option précédente (l'état passe en `DESELECTED`) avant de changer celui de l'option choisie (celle-ci passe à l'état `SELECTED`).

Nous pouvons aussi utiliser l'**interface ActionListener** pour récupérer l'option sélectionnée. La méthode `getSelectedItem()` retourne une variable de type `Object` : pensez donc à effectuer un `cast`, ou à utiliser la méthode `toString()` si les éléments ne sont pas des `Strings`.

- Les objets `JCheckBox` : cases à cocher

```
import javax.swing.JCheckBox;
public class Fenetre extends JFrame
{
    private JPanel container = new JPanel();
    private JCheckBox check1 = new JCheckBox("Case 1");
    private JCheckBox check2 = new JCheckBox("Case 2");

    public Fenetre()
    {
        ...
        JPanel top = new JPanel();
        check1.addActionListener(new StateListener());
        check2.addActionListener(new StateListener());
        top.add(check1);
        top.add(check2);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());
        container.add(top, BorderLayout.NORTH);

        this.setContentPane(container);
        this.setVisible(true);
    }

    class StateListener implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            System.out.println("source : " + ((JCheckBox)e.getSource()).getText());
            System.out.println("état : " + ((JCheckBox)e.getSource()).isSelected());
        }
    }
}
```

Vous pouvez déterminer si l'un de ces composants est sélectionné grâce à la méthode **isSelected()**. Cette méthode retourne true si l'objet est sélectionné, false dans le cas contraire.

- Les objets **JRadioButton** : proposer au moins deux choix, mais ne permettant d'en sélectionner qu'un à la fois

```
import javax.swing.JRadioButton;
public class Fenetre extends JFrame
{
    private JPanel container = new JPanel();
    private JRadioButton jr1 = new JRadioButton("Radio 1");
    private JRadioButton jr2 = new JRadioButton("Radio 2");

    public Fenetre()
    {
        ...
        JPanel top = new JPanel();
        jr1.addActionListener(new StateListener());
        jr2.addActionListener(new StateListener());
        top.add(jr1);
        top.add(jr2);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());
        container.add(top, BorderLayout.NORTH);

        this.setContentPane(container);
        this.setVisible(true);
    }

    class StateListener implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            System.out.println("Source : " + ((JRadioButton)e.getSource()).getText());
            System.out.println("Etat : " + ((JRadioButton)e.getSource()).isSelected());
        }
    }
}
```



|                                 |
|---------------------------------|
| source : Radio 1 - état : true  |
| source : Radio 2 - état : false |
| source : Radio 1 - état : false |
| source : Radio 2 - état : true  |
| source : Radio 1 - état : true  |
| source : Radio 2 - état : false |

Le problème, ici, c'est que nous pouvons sélectionner les deux options (alors que ce n'est normalement pas possible). Pour qu'un seul bouton radio soit sélectionné à la fois, nous devons définir un **groupe de boutons** à l'aide de **ButtonGroup**. Nous y ajouterons nos boutons radio, et seule une option pourra alors être sélectionnée.

```
ButtonGroup bg = new ButtonGroup();
bg.add(jr1);
bg.add(jr2);
```

- Les objets **JTextField** : les champs de texte

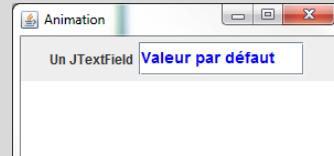
Par default, ils acceptent tous les types de caractères.

```
import javax.swing.JTextField;
public class Fenetre extends JFrame
{
    private JPanel container = new JPanel();
    private JLabel label = new JLabel("Un JTextField");
    private JTextField jtf = new JTextField("Valeur par défaut");

    public Fenetre()
    {
        JPanel top = new JPanel();
        Font police = new Font("Arial", Font.BOLD, 14);
        jtf.setFont(police);
        jtf.setPreferredSize(new Dimension(150, 30));
        jtf.setForeground(Color.BLUE);
        top.add(label);
        top.add(jtf);

        container.setBackground(Color.white);
        container.setLayout(new BorderLayout());
        container.add(top, BorderLayout.NORTH);

        this.setContentPane(container);
        this.setVisible(true);
    }
}
```



- Un **JFormattedTextField** représente un JTextField plus restrictif.

Cet objet permet de créer un JTextField formaté pour recevoir un certain type de données saisies (date, pourcentage etc.). Ainsi, nous pourrons éviter beaucoup de contrôles et de casts sur le contenu de nos zones de texte.

Cet objet retourne une valeur uniquement si celle-ci correspond à ce que vous avez autorisé.

```
public class Fenetre extends JFrame
{
    private JPanel container = new JPanel();
    private JFormattedTextField jtf1 = new JFormattedTextField
        (NumberFormat.getIntegerInstance());
    private JFormattedTextField jtf2 = new JFormattedTextField
        (NumberFormat.getPercentInstance());
    private JLabel label = new JLabel("Un JTextField");
    private JButton b = new JButton ("OK");

    public Fenetre()
    {
        JPanel top = new JPanel();
```

```

Font police = new Font("Arial", Font.BOLD, 14);
jtf1.setFont(police);
jtf1.setPreferredSize(new Dimension(150, 30));
jtf1.setForeground(Color.BLUE);
jtf2.setPreferredSize(new Dimension(150, 30));

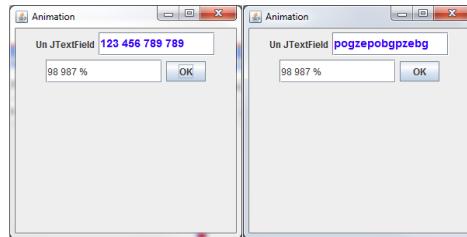
b.addActionListener(new BoutonListener());
top.add(label);
top.add(jtf1);
top.add(jtf2);
top.add(b);

container.setBackground(Color.white);
container.setLayout(new BorderLayout());

this.setContentPane(top);
this.setVisible(true);
}

class BoutonListener implements ActionListener{
    public void actionPerformed(ActionEvent e) {
        System.out.println("TEXT : " + jtf1.getText());
        System.out.println("TEXT : " + jtf2.getText());
    }
}
}

```



Cet objet met automatiquement la saisie en forme lorsqu'elle est valide : il espace les nombres tous les trois chiffres afin d'en faciliter la lecture.

Voici ce que vous pouvez utiliser dans ce champ :

- **NumberFormat** : getIntegerInstance(), getPercentInstance(), getNumberInstance()
- **DateFormat** : getTimeInstance(), getDateInstance()
- **MessageFormat**
- **MaskFormatter**

```

try{
    MaskFormatter tel = new MaskFormatter("## ## ## ## ##");
    MaskFormatter tel2 = new MaskFormatter("##-##-##-##-##");
    JFormattedTextField jtf = new JFormattedTextField(tel2);
}
catch(ParseException e) { e.printStackTrace(); }

```

- Pour contrôler les **événements clavier** utiliser une implémentation de l'**interface KeyListener**

Elle nous permet d'intercepter les événements clavier lorsque l'on :

- presse une touche : **keyPressed(KeyEvent event)**
- relâche une touche : **keyReleased(KeyEvent event)**
- tape sur une touche : **keyTyped(KeyEvent event)**

L'objet **KeyEvent** nous permettra d'obtenir des informations sur les touches qui ont été utilisées.

Parmi celles-ci, nous utiliserons :

- **getKeyCode()** : retourne le code de la touche ;
- **getKeyChar()** : retourne le caractère correspondant à la touche.

Exemple code source :

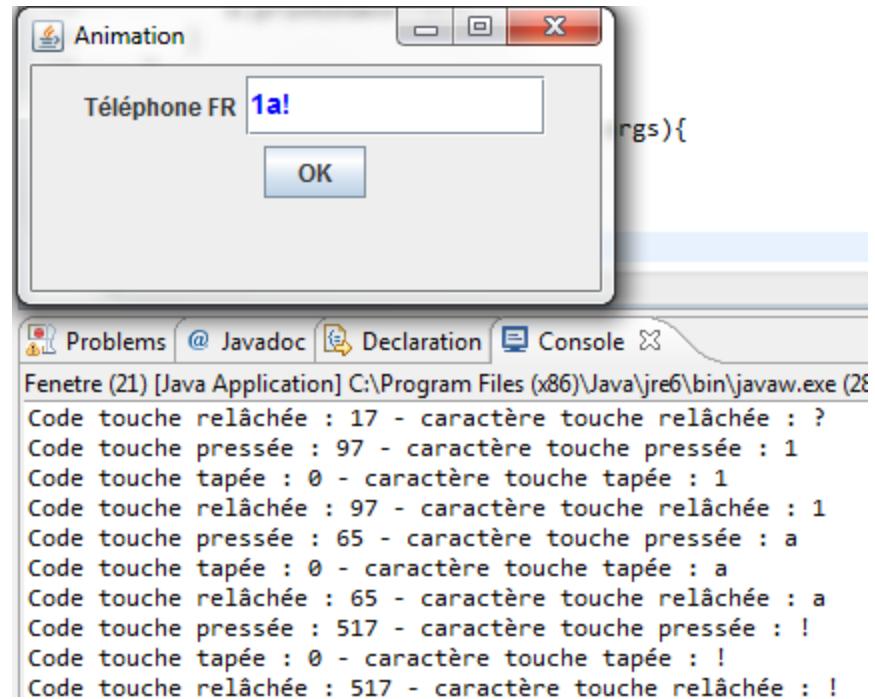
```
public Fenetre()
{
    ...
    jtf.addKeyListener(new ClavierListener());
}

private void pause()
{
    try { Thread.sleep(1000); }
    catch (InterruptedException e) { e.printStackTrace(); }
}

class ClavierListener implements KeyListener
{
    public void keyPressed(KeyEvent event)
    {
        System.out.println("Code touche pressée : " + event.getKeyCode());
        System.out.println("Caractère pressée : " + event.getKeyChar());
        pause();
    }

    public void keyReleased(KeyEvent event)
    {
        System.out.println("Code touche relâchée : " + event.getKeyCode());
        System.out.println("Caractère relâchée : " + event.getKeyChar());
        pause();
    }

    public void keyTyped(KeyEvent event)
    {
        System.out.println("Code touche tapé : " + event.getKeyCode());
        System.out.println("Caractère tapé : " + event.getKeyChar());
        pause();
    }
}
```



Vous pouvez maintenant vous rendre compte de l'**ordre dans lequel les événements du clavier sont gérés** : en premier, lorsqu'on **presse** la touche, en deuxième, lorsqu'elle est **tapée**, et enfin, lorsqu'elle est **relâchée**.

## Les menus et boîtes de dialogue

### • Les boîtes de dialogue

Une petite fenêtre pouvant servir à plusieurs choses :

- afficher une information (message d'erreur, d'avertissement...) ;
- demander une validation, une réfutation ou une annulation ;
- demander à l'utilisateur de saisir une information dont le système a besoin ;
- etc.

Elles peuvent servir à beaucoup de choses. Il faut toutefois les utiliser avec parcimonie : il est assez pénible pour l'utilisateur qu'une application ouvre une boîte de dialogue à chaque notification, car toute boîte ouverte doit être fermée !

Les boîtes de dialogue sont dites **modales**. Cela signifie que lorsqu'une boîte fait son apparition, celle-ci  bloque toute interaction avec un autre composant, et ceci tant que l'utilisateur n'a pas mis fin au dialogue !

Exemple de boîtes informatives :

```
JOptionPane jop1, jop2, jop3;

// boîte du message d'information
jop1 = new JOptionPane();
jop1.showMessageDialog(null, "Message informatif", "Information",
                      JOptionPane.INFORMATION_MESSAGE);

// boîte du message préventif
jop2 = new JOptionPane();
jop2.showMessageDialog(null, "Message préventif", "Attention",
                      JOptionPane.WARNING_MESSAGE);

// boîte du message d'erreur
jop3 = new JOptionPane();
jop3.showMessageDialog(null, "Message d'erreur", "Erreur",
                      JOptionPane.ERROR_MESSAGE);
```



Types de boîtes possibles : `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `ERROR_MESSAGE`, `PLAIN_MESSAGE`, `QUESTION_MESSAGE`

### • Les boîtes de confirmation

Ceux-ci permettent de **valider**, d'**invalider** ou d'**annuler** une décision.

Nous utiliserons toujours l'objet **JOptionPane**, mais ce sera cette fois avec la méthode **showConfirmDialog()**, une méthode qui retourne un entier correspondant à l'option que vous aurez choisie dans cette boîte : Yes, No ou Cancel.

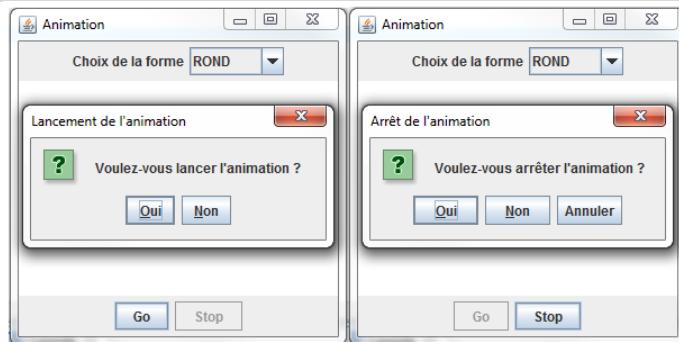
**Exemple** : utiliser une boîte de confirmation lorsque nous cliquons sur l'un des boutons contrôlant l'animation (Go ou Stop)

```
public class Bouton1Listener implements ActionListener // Go button
{
    public void actionPerformed(ActionEvent arg0)
    {
        JOptionPane jop = new JOptionPane();
        int option = jop.showConfirmDialog(null,
   "Voulez-vous lancer l'animation ?",
   "Lancement de l'animation",
   JOptionPane.YES_NO_OPTION,
   JOptionPane.QUESTION_MESSAGE);

        if (option == JOptionPane.OK_OPTION)
        {
            ...
        }
    }

    class Bouton2Listener implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            JOptionPane jop = new JOptionPane();
            int option = jop.showConfirmDialog(null,
   "Voulez-vous arrêter l'animation ?",
   "Arrêt de l'animation",
   JOptionPane.YES_NO_CANCEL_OPTION,
   JOptionPane.QUESTION_MESSAGE);

            if (option != JOptionPane.NO_OPTION &&
                option != JOptionPane.CANCEL_OPTION &&
                option != JOptionPane.CLOSED_OPTION)
            {
                ...
            }
        }
    }
}
```



Nous affectons le résultat que renvoie la méthode `showConfirmDialog()` à une variable de type `int`. Nous nous servons de cette variable afin de savoir quel bouton a été cliqué (oui ou non).

- Les boîtes de saisie

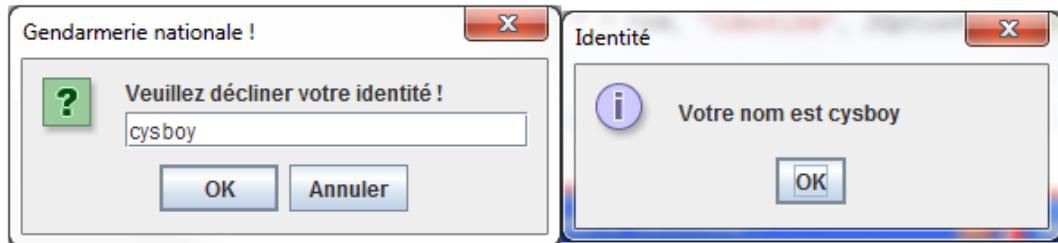
Ceux-ci permettent de saisir du texte ou de faire un choix dans une liste déroulante

```
import javax.swing.JOptionPane;

public class Test
{
    public static void main(String[] args)
    {
        JOptionPane jop1 = new JOptionPane(), jop2 = new JOptionPane();

        String nom = jop1.showInputDialog(null,
   "Veuillez décliner votre identité !",
   "Gendarmerie nationale !",
   JOptionPane.QUESTION_MESSAGE);

        jop2.showMessageDialog(null, "Votre nom est " + nom,
                             "Identité",
                             JOptionPane.INFORMATION_MESSAGE);
    }
}
```



Avec une liste

```
String[] sexe = {"masculin", "féminin", "indéterminé"};
String nom = (String) jop.showInputDialog(null,
   "Veuillez indiquer votre sexe !",
   "Gendarmerie nationale !",
   JOptionPane.QUESTION_MESSAGE,
   null,
   sexe,
   sexe[2]);
```



- Les boîtes de dialogue personnalisées

```

import javax.swing.JDialog;
import javax.swing.JFrame;

public class ZDialog extends JDialog
{
    private ZDialogInfo zInfo = new ZDialogInfo();
    private boolean sendData;
    private JLabel nomLabel, sexeLabel, cheveuxLabel, ageLabel,
                  tailleLabel,taille2Label, icon;
    private JRadioButton tranche1, tranche2, tranche3, tranche4;
    private JComboBox sexe, cheveux;
    private JTextField nom, taille;

    public ZDialog(JFrame parent, String title, boolean modal)
    {
        super(parent, title, modal);           // on appelle le constructeur de JDialog correspondant
        this.setSize(550, 270);                // on spécifie une taille
        this.setLocationRelativeTo(null);       // la position
        this.setResizable(false);             // pas être redimensionnable
        this.setVisible(true);                // enfin on l'affiche
        initComponents();
    }
    ...
}

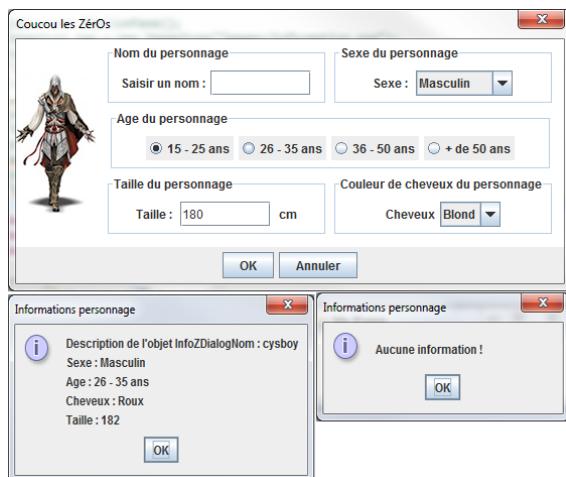
```

Test :

```

bouton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        ZDialog zd = new ZDialog(null, "Coucou les ZérOs", true);
        ZDialogInfo zInfo = zd.showZDialog();
        JOptionPane jop = new JOptionPane ();
        jop.showMessageDialog(null, zInfo.toString(),
                            "Informations personnage", JOptionPane.INFORMATION_MESSAGE);
    }
});

```



- Les menus

L'objet servant à insérer une barre de menus sur vos IHM swing est un **JMenuBar**.

Dans cet objet, vous pouvez mettre des objets **JMenu** afin de créer un menu déroulant. Celui-ci accepte des objets **JMenu**, **JMenuItem**, **JCheckBoxMenuItem** et **JRadioButtonMenuItem**.

Afin de permettre des interactions avec nos futurs menus, nous allons devoir implémenter l'interface **ActionListener**. Ces implémentations serviront à écouter les objets **JMenuItem** : ce sont ces objets qui déclencheront l'une ou l'autre opération. Les **JMenu**, eux, se comportent automatiquement : si on clique sur un titre de menu, celui-ci se déroule tout seul et, dans le cas où nous avons un tel objet présent dans un autre **JMenu**, une autre liste se déroulera toute seule !

Code source :

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.ButtonGroup;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JRadioButtonMenuItem;

public class ZFenetreb extends JFrame
{
    private JMenuBar menuBar = new JMenuBar();
    private JMenu test1 = new JMenu("Fichier");
    private JMenu test1_2 = new JMenu("Sous ficher");
    private JMenu test2 = new JMenu("Edition");

    private JMenuItem item1 = new JMenuItem("Ouvrir");
    private JMenuItem item2 = new JMenuItem("Fermer");
    private JMenuItem item3 = new JMenuItem("Lancer");
    private JMenuItem item4 = new JMenuItem("Arrêter");

    private JCheckBoxMenuItem jcmi1 = new JCheckBoxMenuItem("Choix 1");
    private JCheckBoxMenuItem jcmi2 = new JCheckBoxMenuItem("Choix 2");
    private JRadioButtonMenuItem jrm1 = new JRadioButtonMenuItem("Radio 1");
    private JRadioButtonMenuItem jrm2 = new JRadioButtonMenuItem("Radio 2");

    public static void main(String[] args)
    {
        ZFenetreb zFen = new ZFenetreb();
    }
}
```

```

public ZFenetre()
{
    this.setSize(400, 200);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);

    // on initialise nos menus
    this.test1.add(item1);

    // on ajoute les éléments dans notre sous-menu
    this.test1_2.add(jcmi1);
    this.test1_2.add(jcmi2);
    this.test1_2.addSeparator();                                // ajout d'un séparateur

    ButtonGroup bg = new ButtonGroup();                      // on met nos radios dans un ButtonGroup
    bg.add(jrmi1);
    bg.add(jrmi1);
    jrmi1.setSelected(true);                                // on présélectionne la première radio

    this.test1_2.add(jrmi1);
    this.test1_2.add(jrmi2);

    //Ajout du sous-menu dans notre menu
    this.test1.add(this.test1_2);
    this.test1.addSeparator();                                // ajout d'un séparateur

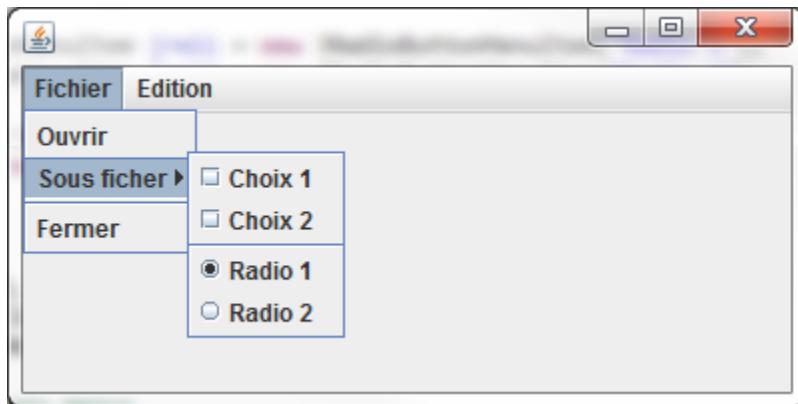
    item2.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent arg0)
        {
            System.exit(0);
        }
    });

    this.test1.add(item2);
    this.test2.add(item3);
    this.test2.add(item4);

    // l'ordre d'ajout va déterminer l'ordre d'apparition dans le menu de gauche à droite
    // le premier ajouté sera tout à gauche de la barre de menu et inversement pour le dernier
    this.menuBar.add(test1);
    this.menuBar.add(test2);
    this.setJMenuBar(menuBar);
    this.setVisible(true);
}
}

```

L'action attachée au JMenuItemFermer permet de quitter l'application.



- Afin d'interagir avec vos points de menu, vous pouvez utiliser une implémentation de l'interface **ActionListener**.
- Pour faciliter l'accès aux menus de la barre de menus, vous pouvez ajouter des **mnémoniques** à ceux-ci.
- L'ajout d'accélérateurs permet de déclencher des actions, le plus souvent par des combinaisons de touches.
- Afin de récupérer les codes des touches du clavier, vous devrez utiliser un objet **KeyStroke** ainsi qu'un objet **KeyEvent**.
- Un menu contextuel fonctionne comme un menu normal, à la différence qu'il s'agit d'un objet **JPopupMenu**. Vous devez toutefois spécifier le composant sur lequel doit s'afficher le menu contextuel.
- La détection du clic droit se fait grâce à la méthode **isPopupTrigger()** de l'objet **MouseEvent**.
- L'ajout d'une barre d'outils nécessite l'utilisation de l'objet **JToolBar**.

## Conteneurs, sliders et barres de progression

- Autres conteneurs

- L'objet **JSplitPane**

```
public class Fenetre extends JFrame
{
    private JSplitPane split;

    public Fenetre()
    {
        ...

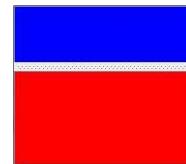
        // on crée deux conteneurs de couleurs différentes
        JPanel pan1 = new JPanel();
        pan1.setBackground(Color.blue);

        JPanel pan2 = new JPanel();
        pan2.setBackground(Color.red);

        // on construit enfin notre séparateur
        split = new JSplitPane(JSplitPane.VERTICAL_SPLIT, pan1, pan2);

        // on le passe ensuite au content pane de notre objet Fenetre, placé au centre
        this.getContentPane().add(split, BorderLayout.CENTER);
        this.setVisible(true);
    }

    public static void main(String[] args)
    {
        Fenetre fen = new Fenetre();
    }
}
```



Pour une separation horizontale : **JSplitPane.HORIZONTAL\_SPLIT**

Les deux autres paramètres ne sont pas nécessairement des JPanel. Vous pouvez utiliser n'importe quelle classe dérivant de JComponent (conteneur, bouton, case à cocher...)

- L'objet **JScrollPane**

```
import javax.swing.JScrollPane;

public class Fenetre extends JFrame
{
    private JTextArea textPane = new JTextArea();
    private JScrollPane scroll = new JScrollPane(textPane);
```

```

public Fenetre()
{
    ...
    JButton bouton = new JButton("Bouton");
    bouton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            System.out.println("Texte écrit dans le JTextArea : ");
            System.out.println(textPane.getText());
        }
    });
}

//On ajoute l'objet au content pane de notre fenêtre
this.getContentPane().add(scroll, BorderLayout.CENTER);
this.getContentPane().add(bouton, BorderLayout.SOUTH);
this.setVisible(true);
}

public static void main(String[] args)
{
    Fenetre fen = new Fenetre();
}
}

```



- L'objet **JTabbedPane**

Sert à créer des onglets (des "pages").

```

public class Fenetre extends JFrame
{
    private JTabbedPane onglet;

    public Fenetre()
    {
        ...
        // création de plusieurs Panneau
        Panneau[] tPan = { new Panneau(Color.RED), new Panneau(Color.GREEN),
                           new Panneau(Color.BLUE)};

        // Crédation de notre conteneur d'onglets
        onglet = new JTabbedPane();
    }
}

```

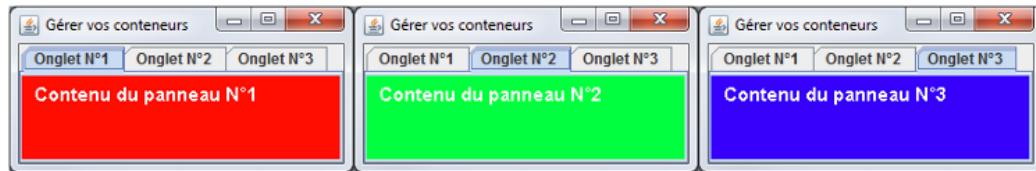
```

int i = 0;
for ( Panneau pan : tPan )
    onglet.addTab("Onglet n° " + (++i), pan);

// on passe ensuite les onglets au content pane
this.getContentPane().add(onglet);
this.setVisible(true);
}

public static void main(String[] args)
{
    Fenetre fen = new Fenetre();
}
}

```



- L'objet **JWindow** : une JFrame sans les contours permettant de réduire, fermer ou agrandir la fenêtre

```

public class Window extends JWindow
{
    public Window(){
        setSize(220, 165);
        setLocationRelativeTo(null);
        JPanel pan = new JPanel();
        JLabel img = new JLabel(new ImageIcon("planète.jpeg"));
        img.setVerticalAlignment(JLabel.CENTER);
        img.setHorizontalAlignment(JLabel.CENTER);
        pan.setBorder(BorderFactory.createLineBorder(Color.blue));
        pan.add(img);
        getContentPane().add(pan);
    }

    public static void main(String[] args)
    {
        Window wind = new Window();
        wind.setVisible(true);
    }
}

```

- L'objet **JSlider**

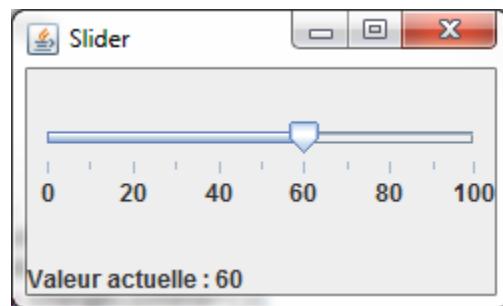
Ce composant permet d'**utiliser un système de mesure** pour une application : redimensionner une image, choisir le tempo d'un morceau de musique, l'opacité d'une couleur, etc.

```
public class Fenetre extends JFrame
{
    private JLabel label = new JLabel("Valeur actuelle : 30");

    public Fenetre ()
    {
        ...
        JSlider slide = new JSlider();
        slide.setMaximum(100);
        slide.setMinimum(0);
        slide.setValue(30);
        slide.setPaintTicks(true);
        slide.setPaintLabels(true);
        slide.setMinorTickSpacing(10);
        slide.setMajorTickSpacing(20);
        slide.addChangeListener(new ChangeListener()
        {
            public void stateChanged(ChangeEvent event)
            {
                label.setText("Valeur : " + ((JSlider)event.getSource()).getValue());
            }
        });
    }

    this.getContentPane().add(slide, BorderLayout.CENTER);
    this.getContentPane().add(label, BorderLayout.SOUTH);
}

public static void main(String[] args)
{
    Fenetre f = new Fenetre ();
    f.setVisible(true);
}
```



- L'objet **JProgressBar**

```
public class Fenetre extends JFrame
{
    private Thread t;
    private JProgressBar bar;
    private JButton launch ;
    ...
    launch = new JButton("Lancer");
    launch.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            t = new Thread(new Traitement());
            t.start();
        }
    });
    bar = new JProgressBar();
    bar.setMaximum(500);
    bar.setMinimum(0);
    bar.setStringPainted(true);

    this.getContentPane().add(bar, BorderLayout.CENTER);
    this.getContentPane().add(launch, BorderLayout.SOUTH);
    t = new Thread(new Traitement());
    t.start();
    this.setVisible(true);
}

class Traitement implements Runnable
{
    public void run()
    {
        launch.setEnabled(false);
        for ( int val = 0; val <= 500; val++ )
        {
            bar.setValue(val);
            try
            {
                t.sleep(10);
            }
            catch (InterruptedException e) { e.printStackTrace(); }
        }
        launch.setEnabled(true);
    }
}
public static void main(String[] args)
{ Progress p = new Progress(); }
}
```



## Les interfaces des tableaux

- Les **tableaux** sont des **composants** qui permettent d'afficher des données de façon **structurée**

Pour utiliser le composant « tableau », vous devrez utiliser l'objet **JTable**.

```
public class Fenetre extends JFrame
{
    public Fenetre()
    {
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setTitle("JTable");
        this.setSize(300, 120);

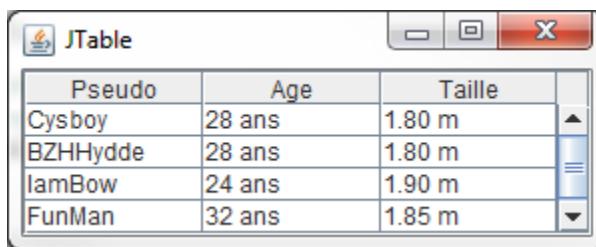
        // les données du tableau
        Object[][] data = { {"Cysboy", "28 ans", "1.80 m"}, 
                            {"BZHHydde", "28 ans", "1.80 m"}, 
                            {"IamBow", "24 ans", "1.90 m"}, 
                            {"FunMan", "32 ans", "1.85 m"} };

        // les titres des colonnes
        String title[] = {"Pseudo", "Age", "Taille"};

        JTable tableau = new JTable(data, title);

        // ajoute le tableau au contentPane dans un scroll; sinon les titres des colonnes ne s'afficheront pas
        this.getContentPane().add(new JScrollPane(tableau));
    }

    public static void main(String[] args)
    {
        Fenetre fen = new Fenetre();
        fen.setVisible(true);
    }
}
```



Celui-ci prend en **paramètres** un tableau d'objets à deux dimensions (un tableau de données) correspondant aux données à afficher, et un tableau de chaînes de caractères qui, lui, affichera les titres des colonnes.

Les titres des colonnes du tableau peuvent être de type String ou de type Object, tandis que les données sont obligatoirement de type Object.

- **Les cellules**

Les tableaux sont composés de celles. Elles sont encadrées de bordures noires et contiennent les données que vous avez mises dans le tableau d'Object et de String. Celles-ci peuvent être retrouvées par leurs coordonnées ( $x$ ,  $y$ ) où  $x$  correspond au numéro de la ligne et  $y$  au numéro de la colonne ! Une cellule est donc l'intersection d'une ligne et d'une colonne.

Afin de **modifier une cellule**, il faut récupérer la ligne et la colonne auxquelles elle appartient.

Changer la taille d'une colonne et d'une ligne :

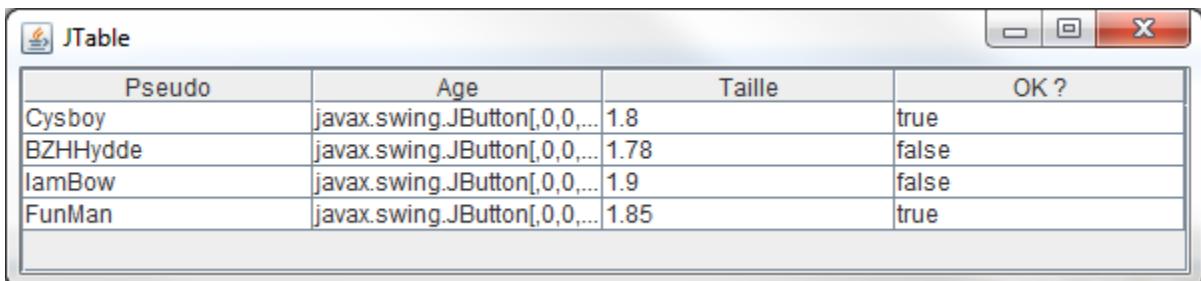
```
public void changeSize(int width, int height)
{
    // créer un objet TableColumn afin de travailler sur la colonne
    TableColumn col;
    for ( int i = 0; i < tableau.getColumnCount(); i++ )
    {
        if ( i == 1 )                                // affecte la taille de la colonne 1
        {
            col = tableau.getColumnModel().getColumn(i); // récupère le modèle de la colonne
            col.setPreferredWidth(width);                // affecte la nouvelle valeur
        }
    }

    for ( int i = 0; i < tableau.getRowCount(); i++ )
    {
        if (i == 1)                                  // affecte la taille de la ligne 1
            tableau.setRowHeight(i, height);
    }
}
```



- Afin de gérer vous-mêmes le contenu du tableau, vous pouvez utiliser un modèle de données (**JTableModel**). Cet objet fait le lien entre votre tableau et vos données. C'est également lui qui stocke vos données.

Pour être plus flexible, on peut créer son propre modèle qui va stocker les données du tableau, il suffit de créer une classe **héritant de AbstractTableModel**.



```

public class Fenetre extends JFrame
{
    private JTable tableau;

    public Fenetre()
    {
        ...
    }

    Object[][] data = {
        {"Cysboy", new JButton("6boy"), new Double(1.80), new Boolean(true)},
        {"BZHHydde", new JButton("BZH"), new Double(1.78), new Boolean(false)},
        {"IamBow", new JButton("Bow"), new Double(1.90), new Boolean(false)},
        {"FunMan", new JButton("Year"), new Double(1.85), new Boolean(true)}};

    String title[] = {"Pseudo", "Age", "Taille", "OK ?"};

    ZModel model = new ZModel(data, title);
    System.out.println("Nombre de colonne : " + model.getColumnCount());
    System.out.println("Nombre de ligne : " + model.getRowCount());

    this.tableau = new JTable(model);
    this.getContentPane().add(new JScrollPane(tableau), BorderLayout.CENTER);
}

// classe modèle personnalisée
class ZModel extends AbstractTableModel
{
    private Object[][] data;
    private String[] title;

    public ZModel(Object[][] data, String[] title)
    {
        this.data = data;
        this.title = title;
    }

    public int getColumnCount() { return this.title.length; }
    public int getRowCount() { return this.data.length; }
    public String getColumnName(int col) { return this.title[col]; }
    public Object getValueAt(int row, int col) { return this.data[row][col]; }
}

public static void main(String[] args)
{
    Fenetre fen = new Fenetre();
    fen.setVisible(true);
}
}

```

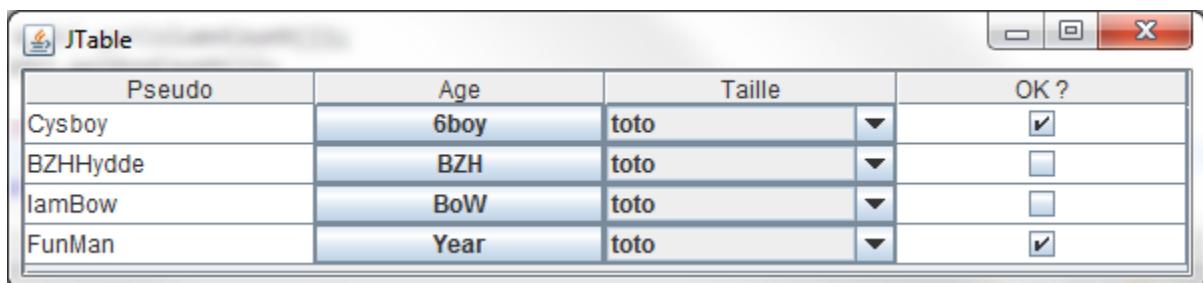
- Pour ajouter ou retirer des **lignes** à un tableau, il faut passer par un **modèle de données**. Ainsi, l'affichage est mis à jour automatiquement. Il en va de même pour l'ajout et la suppression de **colonnes**.
- La gestion de l'affichage brut (hors édition) des cellules peut se gérer colonne par colonne à l'aide d'une classe dérivant de **DefaultTableCellRenderer**.

Le but est de définir une nouvelle façon de dessiner les composants dans les cellules.

Nous allons donc dire à notre tableau que la valeur contenu dans une cellule donnée est un composant (bouton ou autre). Il suffit de créer une classe héritant de **DefaultTableCellRenderer** et de redefinir la méthode **getTableCellRendererComponent**

```
public class TableComponent extends DefaultTableCellRenderer
{
    public Component getTableCellRendererComponent(JTable table, Object value,
   boolean isSelected, boolean hasFocus, int row, int column)
    {
        if ( value instanceof JButton )
            return (JButton) value;
        else if ( value instanceof JComboBox )
            return (JComboBox) value;
        else
            return this;
    }
}

// dire à notre tableau qu'il doit utiliser ce rendu de cellules
this.tableau.setDefaultRenderer(JComponent.class, new TableComponent());
```



- La gestion de l'affichage brut lors de l'édition d'une cellule se gère colonne par colonne avec une classe dérivant de **DefaultTableCellEditor**.

```
public class ButtonEditor extends DefaultCellEditor
{
    protected JButton button;
    private boolean isPushed;
    private ButtonListener bListener = new ButtonListener();

    public ButtonEditor(JCheckBox checkBox)
    {
        super(checkBox);
        button = new JButton();
```

```

        button.setOpaque(true);
        button.addActionListener(bListener);
    }

    public Component getTableCellEditorComponent(JTable table, Object value,
  boolean isSelected, int row, int column)
    {
        bListener.setRow(row);                                // on affecte le numéro de ligne au listener
        bListener.setColumn(column);                          // idem pour le numéro de colonne
        bListener.setTable(table);                           // passe le tableau en param pour des actions potentielles
        button.setText( (value == null) ? "" : value.toString() ); // réaffecte le libellé
        return button;                                     // renvoie le bouton
    }

    // listener pour le bouton
    class ButtonListener implements ActionListener
    {
        private int column, row;
        private JTable table;
        private int nbre = 0;
        private JButton button;

        public void setColumn(int col){this.column = col;}
        public void setRow(int row){this.row = row;}
        public void setTable(JTable table){this.table = table;}

        public JButton getButton(){return this.button;}

        public void actionPerformed(ActionEvent event)
        {
            System.out.println("Bouton :" + ((JButton)event.getSource()).getText());
            // on affecte un nouveau libellé à une cellule de la ligne
            ((AbstractTableModel)table.getModel()).setValueAt("New Value " +
                (++nbre), this.row, (this.column -1));
            // permet de dire à notre tableau qu'une valeur a changé à l'emplacement précisé
            ((AbstractTableModel)table.getModel()).fireTableCellUpdated(this.row,
                this.column - 1);
            this.button = ((JButton)event.getSource());
        }
    }

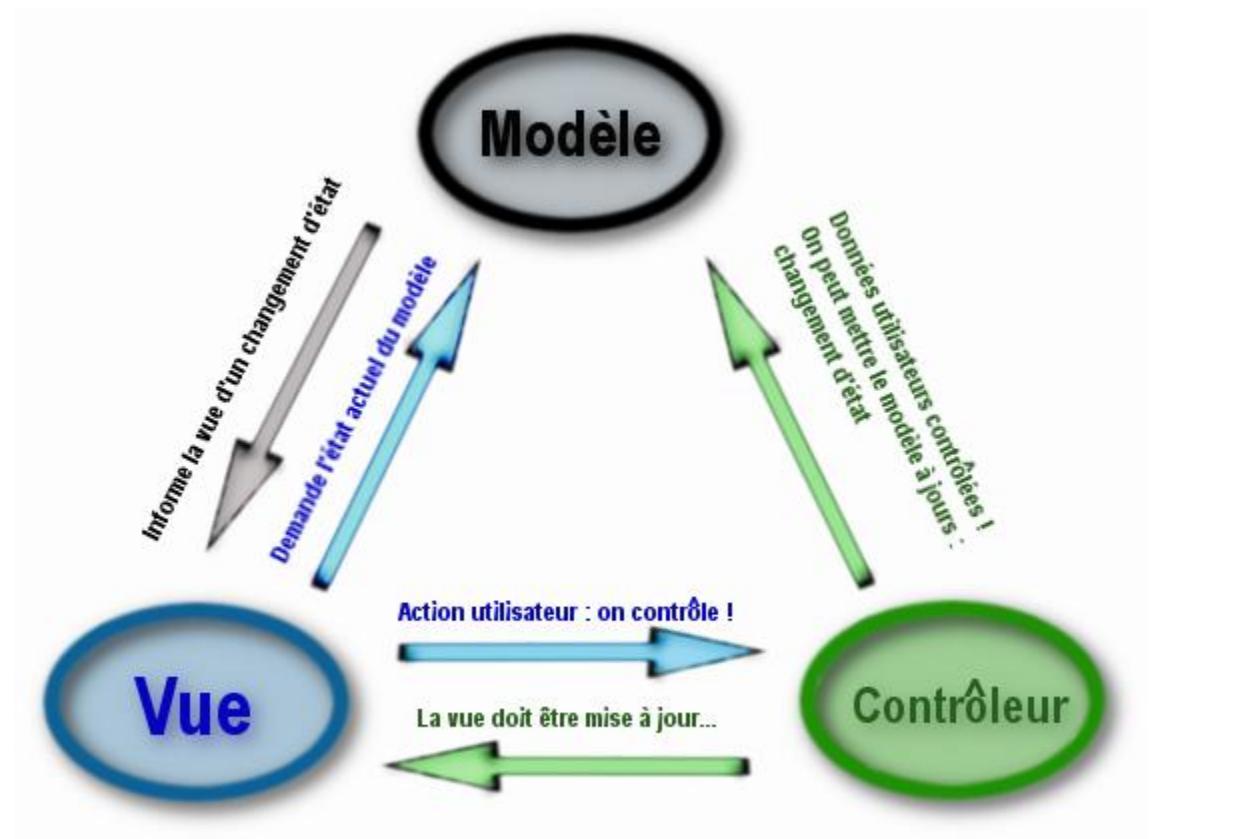
    // classe Fenetre
    public Fenetre()
    {
        ...
        this.tableau.getColumn("Age").setCellRenderer(new ButtonRenderer());
        this.tableau.getColumn("Age").setCellEditor(new ButtonEditor(new
            JCheckBox()));
    }
}

```

## Mieux structurer son code : Le pattern MVC

Un design pattern qui impose de **découper son code en trois parties** : **modèle**, **vue** et **contrôleur**.

C'est un pattern composé, ce qui signifie qu'il est constitué d'au moins deux patterns.



### La vue

Ce que l'on nomme « la vue » est en fait **une IHM**. Elle représente ce que l'utilisateur a sous les yeux.

La vue peut donc être :

- une application graphique Swing, AWT, SWT pour Java ;
- une page web ;
- un terminal Linux ou une console Windows ;
- etc.

### Le modèle

Le modèle peut être divers et varié. C'est là que se trouvent **les données**. Il s'agit en général d'**un ou plusieurs objets** Java. Ces objets s'apparentent généralement à ce qu'on appelle souvent « la couche métier » de l'application et effectuent des traitements absolument transparents pour l'utilisateur.

Par exemple, on peut citer des objets dont le rôle est de gérer une ou plusieurs tables d'une base de données. En trois mots, il s'agit du cœur du programme !

## Le contrôleur

Cet objet permet de **faire le lien entre la vue et le modèle** lorsqu'une action utilisateur est intervenue sur la vue. C'est cet objet qui aura pour rôle de **contrôler les données**.

Afin de travailler sur un exemple concret, nous allons reprendre notre calculatrice issue d'un TP précédent. Dans une application structurée en MVC, voici ce qu'il peut se passer :

- l'utilisateur effectue une action sur votre calculatrice (un clic sur un bouton) ;
- l'action est captée par le contrôleur, qui va vérifier la cohérence des données et éventuellement les transformer afin que le modèle les comprenne. Le contrôleur peut aussi demander à la vue de changer ;
- le modèle reçoit les données et change d'état (une variable qui change, par exemple) ;
- le modèle notifie la vue (ou les vues) qu'il faut se mettre à jour ;
- l'affichage dans la vue (ou les vues) est modifié en conséquence en allant chercher l'état du modèle.

Vous pouvez isoler deux patterns dans cette architecture.

- Le **pattern observer** se trouve au niveau du **modèle**. Lorsque celui-ci va changer d'état, tous les objets qui l'observeront seront mis au courant automatiquement (avec un couplage faible)
- Le deuxième est le **pattern strategy** ! Ce pattern est situé au niveau du **contrôleur**. On dit aussi que le contrôleur est la stratégie de la vue. En fait, le contrôleur va transférer les données de l'utilisateur au modèle et il a tout à fait le droit de modifier le contenu.

## Exemple concret : calculatrice & MVC

Le **Modèle** est l'objet qui sera chargé de stocker les données nécessaires à un calcul (nombre et opérateur) et d'avoir le résultat. Afin de prévoir un changement éventuel de modèle, nous créerons le notre à partir d'un supertype de modèle : de cette manière, si un changement s'opère, nous pourrons utiliser les différentes classes filles de façon polymorphe.

Pour réaliser des calculs simples, notre modèle devra :

- récupérer et stocker au moins un nombre ;
- stocker l'opérateur de calcul ;
- calculer le résultat ;
- renvoyer le résultat ;
- tout remettre à zéro.

Nous allons utiliser le **pattern observer** afin de faire communiquer notre modèle avec d'autres objets. Il nous faudra donc une implémentation de ce pattern ; la voici, dans un package **com.sdz.observer**.

### **Observable.java**

```
package com.sdz.observer;  
  
public interface Observable  
{  
    public void addObserver(Observer obs);  
    public void removeObserver();  
    public void notifyObserver(String str);  
}
```

### Observer.java

```
package com.sdz.observer;

public interface Observer
{
    public void update(String str);
}
```

Notre classe abstraite devra donc implémenter ce pattern afin de centraliser les implémentations. Puisque notre supertype implémente le pattern observer, les classes héritant de cette dernière hériteront aussi des méthodes de ce pattern !

Voici donc le code de notre **classe abstraite** que nous placerons dans le package **com.sdz.model**

```
package com.sdz.model;

import java.util.ArrayList;
import com.sdz.observer.Observable;
import com.sdz.observer.Observer;

public abstract class AbstractModel implements Observable
{
    protected double result = 0;
    protected String operateur = "", operande = "";
    private ArrayList<Observer> listObserver = new ArrayList<Observer>();

    public abstract void reset();                                // efface
    public abstract void calcul();                               // effectue le calcul
    public abstract void getResultat();                         // affichage forcé du résultat
    public abstract void setOperateur(String operateur);       // définit l'opérateur de l'opération
    public abstract void setNombre(String nbre) ;             // définit le nombre à utiliser pour l'opération

    public void addObserver(Observer obs)
    {
        this.listObserver.add(obs);
    }

    public void removeObserver()
    {
        listObserver = new ArrayList<Observer>();
    }

    public void notifyObserver(String str)
    {
        if ( str.matches("^0[0-9]+") )
            str = str.substring(1, str.length());

        for ( Observer obs : listObserver )
            obs.update(str);
    }
}
```

Maintenant, nous allons créer une classe concrète héritant de AbstractModel.

```
package com.sdz.model;
import com.sdz.observer.Observable;
public class Calculator extends AbstractModel
{
    public void setOperateur(String ope) // définit l'opérateur
    {
        calcul(); // on lance le calcul
        this.operateur = ope; // on stocke l'opérateur
        if ( ! ope.equals("=") ) // si l'opérateur n'est pas =
            this.operande = ""; // on réinitialise l'opérande
    }

    public void setNombre(String result) // définit le nombre
    {
        this.operande += result; // on concatène le nombre
        notifyObserver(this.operande); // on met à jour
    }

    public void getResultat() // force le calcul
    { calcul(); }

    public void reset() // réinitialise tout
    {
        this.result = 0;
        this.operande = "0";
        this.operateur = "";
        notifyObserver(String.valueOf(this.result)); // mise à jour
    }

    public void calcul() // calcul
    {
        if ( this.operateur.equals("") ) // pas d'opérateur ? le résultat est le nombre saisi
            this.result = Double.parseDouble(this.operande);
        else
        {
            if ( ! this.operande.equals("") ) // opérande pas vide ? calcule avec l'opérateur de calcul
            {
                if ( this.operateur.equals("+") )
                    this.result += Double.parseDouble(this.operande);
                if ( this.operateur.equals("-") )
                    this.result -= Double.parseDouble(this.operande);
                if ( this.operateur.equals("*") )
                    this.result *= Double.parseDouble(this.operande);
                if ( this.operateur.equals("/") )
                {
                    try { this.result /= Double.parseDouble(this.operande); }
                    catch(ArithmetricException e) { this.result = 0; }
                }
            }
            this.operande = "";
            notifyObserver(String.valueOf(this.result)); // lance aussi la mise à jour !
        }
    }
}
```

## Le contrôleur

Celui-ci sera chargé de faire le lien entre notre vue et notre modèle. Nous créerons aussi une classe abstraite afin de définir un supertype de variable pour utiliser, le cas échéant, des contrôleurs de façon polymorphe.

Que doit faire notre contrôleur? C'est lui qui va intercepter les actions de l'utilisateur, qui va modeler les données et les envoyer au modèle.

Il devra donc :

- agir lors d'un clic sur un chiffre ;
- agir lors d'un clic sur un opérateur ;
- avertir le modèle pour qu'il se réinitialise dans le cas d'un clic sur le bouton reset ;
- contrôler les données.

Puisque le contrôleur doit interagir avec le modèle, il faudra qu'il possède une instance de notre modèle.

```
package com.sdz.controller;

import java.util.ArrayList;
import com.sdz.model.AbstractModel;

public abstract class AbstractController
{
    protected AbstractModel calc;
    protected String operateur = "", nbre = "";
    protected ArrayList<String> listOperateur = new ArrayList<String>();

    public AbstractController(AbstractModel cal)
    {
        this.calc = cal;
        this.listOperateur.add("+"); //définit la liste des opérateurs afin de s'assurer qu'ils sont corrects
        this.listOperateur.add("-");
        this.listOperateur.add("*");
        this.listOperateur.add("/");
        this.listOperateur.add("=");
    }

    public void setOperateur(String ope) // définit l'opérateur
    {
        this.operateur = ope;
        control();
    }

    public void setNombre(String nombre) // définit le nombre
    {
        this.nbre = nombre;
        control();
    }

    public void reset() // efface
    { this.calc.reset(); }

    abstract void control(); // méthode de contrôle
}
```

Nous avons défini les actions globales de notre objet de contrôle et vous constatez aussi qu'à chaque action dans notre contrôleur, celui-ci invoque la méthode **control()**. Celle-ci va vérifier les données et informer le modèle en conséquence.

Nous allons voir maintenant ce que doit effectuer notre instance concrète.

```
package com.sdz.controller;
import com.sdz.model.AbstractModel;
public class CalculetteController extends AbstractController
{
    public CalculetteController(AbstractModel cal)
    {
        super(cal);
    }
    public void control() // on notifie le modèle d'une action si le contrôle est bon
    {
        if ( this.listOperateur.contains(this.operateur) ) // si l'opérateur est dans la liste
        {
            if ( this.operateur.equals("=") ) // si l'opérateur est =
                this.calc.getResultat(); // on ordonne au modèle d'afficher le résultat
            else
                this.calc.setOperateur(this.operateur); // sinon, on passe l'opérateur au modèle
        }

        if ( this.nbre.matches("[0-9.]+") ) // si le nombre est conforme
            this.calc.setNombre(this.nbre);

        this.operateur = "";
        this.nbre = "";
    }
}
```

Cette classe redéfinit la méthode **control()** et permet d'indiquer les informations à envoyer à notre modèle. Celui-ci mis à jour, les données à afficher dans la vue seront envoyées via l'implémentation du pattern observer entre notre modèle et notre vue.

## La vue

Elle sera créée avec le package javax.swing. Elle sera stocké dans le package com.sdz.vue

```
package com.sdz.vue;
public class Calculette extends JFrame implements Observer
{
    private JPanel container = new JPanel();
    String[] tabString = {"1","2","3","4","5","6","7","8","9","0",".",",","C","+","-","*","/"};
    JButton[] tabButton = new JButton[tabString.length];
    private JLabel ecran = new JLabel();
    private String operateur = "";
    private Dimension dim = new Dimension(50, 40);
    private Dimension dim2 = new Dimension(50, 31);
    private double chiffre1;
    private boolean clicOperateur = false, update = false;
    private AbstractController controller; // l'instance de notre objet contrôleur
```

```

public Calculette(AbstractController controller)
{
    this.setSize(240, 260);
    this.setTitle("Calculette");
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);
    this.setResizable(false);
    initComposant();
    this.controller = controller;
    this.setContentPane(container);
    this.setVisible(true);
}

private void initComposant()
{
    JPanel panOperateur = new JPanel();
    panOperateur.setPreferredSize(new Dimension(55, 225));
    JPanel panChiffre = new JPanel();
    panChiffre.setPreferredSize(new Dimension(165, 225));
    JPanel panEcran = new JPanel();
    panEcran.setPreferredSize(new Dimension(220, 30));

    OperateurListener opeListener = new OperateurListener(); // même listener pour les opérateurs

    for ( int i = 0; i < tabString.length; i++ )
    {
        tabButton[i] = new JButton(tabString[i]);
        tabButton[i].setPreferredSize(dim);

        switch(i)
        {
            case 11 :
                tabButton[i].addActionListener(opeListener);
                panChiffre.add(tabButton[i]);
                break;
            case 12 :
                tabButton[i].setForeground(Color.red);
                tabButton[i].addActionListener(new ResetListener());
                tabButton[i].setPreferredSize(dim2);
                panOperateur.add(tabButton[i]);
                break;
            case 13 :
            case 14 :
            case 15 :
            case 16 :
                tabButton[i].setForeground(Color.red);
                tabButton[i].addActionListener(opeListener);
                tabButton[i].setPreferredSize(dim2);
                panOperateur.add(tabButton[i]);
                break;
            default :
                tabButton[i].addActionListener(new ChiffreListener());
                panChiffre.add(tabButton[i]);
                break;
        }
    }
}

```

```

Font police = new Font("Arial", Font.BOLD, 20);
ecran = new JLabel("0");
ecran.setFont(police);
ecran.setHorizontalAlignment(JLabel.RIGHT);
ecran.setPreferredSize(new Dimension(220, 20));

panEcran.add(ecran);
panEcran.setBorder(BorderFactory.createLineBorder(Color.black));

container.add(panEcran, BorderLayout.NORTH);
container.add(panChiffre, BorderLayout.CENTER);
container.add(panOperateur, BorderLayout.EAST);
}

// les listeners pour nos boutons
class ChiffreListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        String str = ((JButton)e.getSource()).getText(); //affiche le chiffre en plus dans le label

        if ( ! ecran.getText().equals("0") )
            str = ecran.getText() + str;

        controler.setNombre(((JButton)e.getSource()).getText());
    }
}

class OperateurListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        controler.setOperateur(((JButton)e.getSource()).getText());
    }
}

class ResetListener implements ActionListener
{
    public void actionPerformed(ActionEvent arg0)
    {
        controler.reset();
    }
}

// implémentation du pattern observer
public void update(String str)
{
    ecran.setText(str);
}
}

```

Vous constaterez que **la vue contient le contrôleur** (juste avant le constructeur de la classe).

Il ne nous manque plus qu'une classe de test afin d'observer le résultat. Elle crée les trois composants qui vont dialoguer entre eux : le modèle (données), la vue (fenêtre) et le contrôleur qui lie les deux.

```
import com.sdz.controler.*;
import com.sdz.model.*;
import com.sdz.vue.Calcullette;

public class Main
{
    public static void main(String[] args)
    {
        AbstractModel calc = new Calculator();                                //modèle
        AbstractController controller = new CalculletteController(calc);      // contrôleur
        Calcullette calcullette = new Calcullette(controller); // fenêtre avec le contrôleur en param
        calc.addObserver(calcullette);   // ajout de la fenêtre comme observer du modèle
    }
}
```

Lorsque nous cliquons sur un chiffre :

- l'action est envoyée au contrôleur.
- celui-ci vérifie si le chiffre est conforme.
- il informe le modèle.
- ce dernier est mis à jour et informe la vue de ses changements.
- la vue rafraîchit son affichage.

Lorsque nous cliquons sur un opérateur :

- l'action est toujours envoyée au contrôleur.
- celui-ci vérifie si l'opérateur envoyé est dans sa liste.
- le cas échéant, il informe le modèle.
- ce dernier agit en conséquence et informe la vue de son changement.
- la vue est mise à jour.

Il se passera la même chose lorsque nous cliquerons sur le bouton reset.

## Le drag'n drop

Le Drag'n Drop - traduit par « Glisser-Déposer » - revient à sélectionner un élément graphique d'un clic gauche, à le déplacer grâce à la souris tout en maintenant le bouton enfoncé et à le déposer à l'endroit voulu en relâchant le bouton.

En Java, cette notion est arrivée avec JDK 1.2, dans le système graphique `awt`. Ce système est fondu et simplifié avec `swing`

### Le Drag'n Drop avec swing

Pour activer le drag'n drop il faut activer cette fonctionnalité dans les composants concernés.

```
public class TestSwing extends JFrame
{
    public TestSwing()
    {
        super("Test de Drag'n Drop");
        this.setSize(300, 200);

        // une textarea avec son contenu déplaçable
        JTextArea label = new JTextArea("Texte déplaçable !");
        label.setPreferredSize(new Dimension(300, 130));
        label.setDragEnabled(true);

        // un textfield avec son contenu déplaçable
        JTextField text = new JTextField();
        text.setDragEnabled(true);

        // un textfield sans son contenu déplaçable
        JTextField text2 = new JTextField();

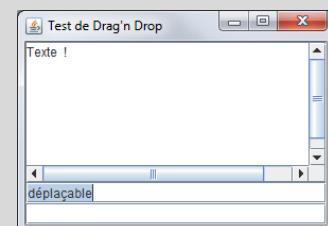
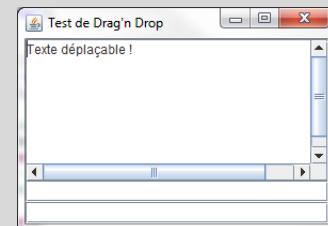
        JPanel pan1 = new JPanel();
        pan1.setBackground(Color.white);
        pan1.setLayout(new BorderLayout());
        pan1.add(new JScrollPane(label), BorderLayout.NORTH);

        JPanel pan2= new JPanel();
        pan2.setBackground(Color.white);
        pan2.setLayout(new BorderLayout());
        pan2.add(text2, BorderLayout.SOUTH);
        pan2.add(text, BorderLayout.NORTH);

        pan1.add(pan2, BorderLayout.SOUTH);

        this.add(pan1, BorderLayout.CENTER);
        this.setVisible(true);
    }

    public static void main(String[] args)
    { new Test1(); }
}
```



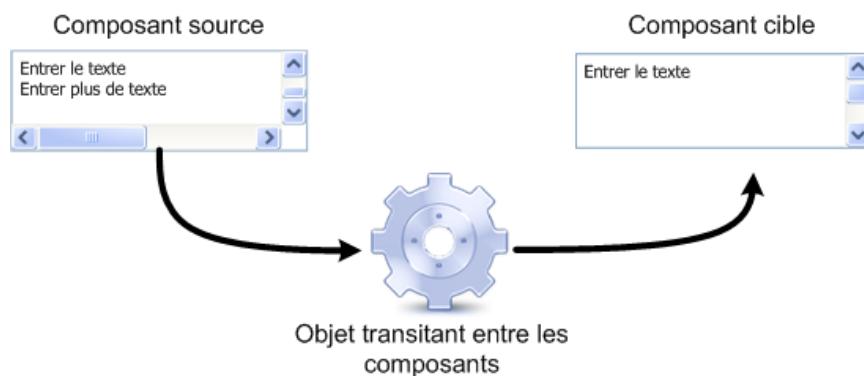
Par défaut, le drag'n drop n'est disponible que pour certains composants.

Il ne faut pas confondre l'action « drag » et l'option « drop ». Certains composants autorisent les deux alors que d'autres n'autorisent que le drag.

Voici un tableau récapitulatif des actions autorisées par composant :

| Composant           | Drag | Drop |
|---------------------|------|------|
| JEditorPane         | X    | X    |
| JColorChooser       | X    | X    |
| JFileChooser        | X    | .    |
| JTextPane           | X    | X    |
| JTextField          | X    | X    |
| JTextArea           | X    | X    |
| JFormattedTextField | X    | X    |
| JPasswordField      | .    | X    |
| JLabel              | .    | .    |
| JTable              | X    | .    |
| JTree               | X    | .    |
| JList               | X    | .    |

Il faut garder en mémoire que lorsqu'on parle de « drag », il y a deux notions implicites à prendre en compte : le « drag déplacement » et le « drag copie ».. L'action « drag déplacement » indique les composants autorisant, par défaut, l'action de type couper/coller, l'action « drag copie » indique que les composants autorisent les actions de type copier/coller.



Pendant l'opération de drag'n drop, les données transitent d'un composant à l'autre via un objet.

Dans l'API Swing, le mécanisme de drag'n drop est encapsulé dans l'objet `JComponent` dont tous les objets graphiques héritent, ce qui signifie que tous les objets graphiques peuvent implémenter cette fonctionnalité.

Exemple : ajouter le comportement drag'n drop à un label

```
JLabel srcLib = new JLabel("Source de drag : ", JLabel.RIGHT);

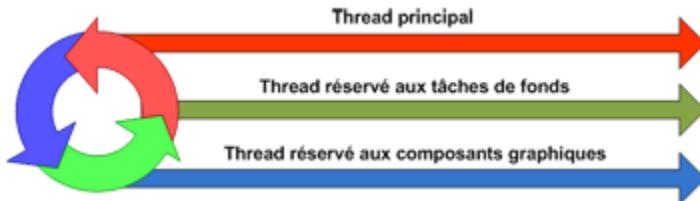
JLabel src = new JLabel("Texte à déplacer !");
src.setTransferHandler(new TransferHandler("text"));

// on spécifie au composant qu'il doit envoyer ses données via son objet TransferHandler
src.addMouseListener(new MouseAdapter()
{
    // on utilise cet événement pour que les actions soient visibles dès le clic de souris...
    public void mousePressed(MouseEvent e)
    {
        JComponent lab = (JComponent)e.getSource();           // récupère le JComponent
        TransferHandler handle = lab.getTransferHandler(); // récupère l'objet de transfert
        handle.exportAsDrag(lab, e, TransferHandler.COPY); // lui ordonne d'amorcer le drag'n'drop
    }
});

JLabel destLib = new JLabel("Destination de drag : ", JLabel.RIGHT);
JTextField dest = new JTextField();
dest.setDragEnabled(true);                                // on active le comportement par défaut de ce composant
```

## Mieux gérer les interactions avec les composants

- Au lancement d'un programme Java, trois threads se lancent : le thread **principal**, celui gérant les **tâches de fond** et l'**EDT** (Event Dispatch Thread)



Il s'agit d'un thread, d'une pile d'appel. Cependant celui-ci a une particularité, il s'occupe de gérer toutes les modifications portant sur un composant graphique :

- le redimensionnement ;
- le changement de couleur ;
- le changement de valeur ;
- ...

Vos applications graphiques seront plus performantes et plus sûres lorsque vous utiliserez ce thread pour effectuer tous les changements qui pourraient intervenir sur votre IHM.

La philosophie de Java est que toute modification apportée à un composant se fait obligatoirement dans l'EDT : lorsque vous utilisez une méthode *actionPerformed*, celle-ci, son contenu compris, est exécutée dans l'EDT (c'est aussi le cas pour les autres intercepteurs d'événements).

La politique de Java est simple : toute **action modifiant l'état d'un composant graphique** doit se faire **dans un seul et unique thread**, l'EDT.

Pourquoi ? C'est simple, les composants graphiques ne sont pas « thread-safe » : ils ne peuvent pas être utilisés par plusieurs threads simultanément et assurer un fonctionnement sans erreurs ! Alors, pour s'assurer que les composants sont utilisés au bon endroit, on doit placer toutes les interactions dans l'EDT.

Par contre, cela signifie que si dans une méthode *actionPerformed* nous avons un traitement assez long, c'est toute notre interface graphique qui sera figée !

- Java préconise que toute modification des composants graphiques se fasse dans l'EDT.
- Si vos IHM se figent, c'est peut-être parce que vous avez lancé un traitement long dans l'EDT.
- Afin d'améliorer la réactivité de vos applications, vous devez choisir au mieux dans quel thread vous allez traiter vos données.
- Java offre la classe **SwingUtilities**, qui permet de lancer des actions dans l'EDT depuis n'importe quel thread.

Cette classe offre plusieurs **méthodes statiques** permettant d'insérer du code dans l'EDT :

- **invokeLater(Runnable doRun)** : exécute le thread en paramètre dans l'EDT et rend immédiatement la main au thread principal ;
- **invokeAndWait(Runnable doRun)** : exécute le thread en paramètre dans l'EDT et attend la fin de celui-ci pour rendre la main au thread principal ;
- **isEventDispatchThread()** : retourne vrai si le thread dans lequel se trouve l'instruction est dans l'EDT.

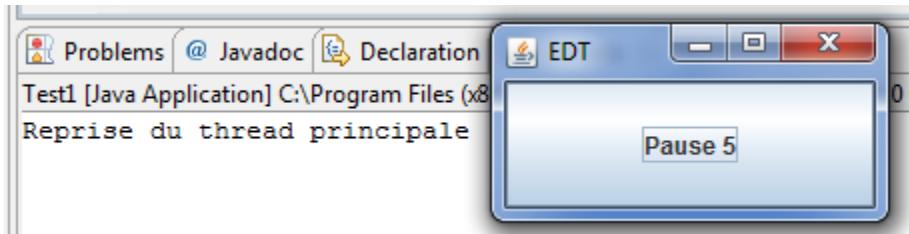
Maintenant que vous savez comment exécuter des instructions dans l'EDT :

```
public class Test1
{
    static int count = 0, count2 = 0;
    static JButton bouton = new JButton("Pause");

    public static void updateBouton()
    {
        for(int i = 0; i < 5; i++)
        {
            try { Thread.sleep(1000); }
            catch (InterruptedException e) { e.printStackTrace(); }
            bouton.setText("Pause " + ++count);
        }
    }

    public static void main(String[] args)
    {
        JFrame fen = new JFrame("EDT");
        fen.getContentPane().add(bouton);
        fen.setSize(200, 100);
        fen.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        fen.setLocationRelativeTo(null);
        fen.setVisible(true);
        updateBouton();
        System.out.println("Reprise du thread principal");
    }
}
```

Au lancement de ce test, vous constatez que le thread principal ne reprend la main qu'après la fin de la méthode *updateBouton()* :



La solution pour rendre la main au thread principal avant la fin de la méthode : créez un nouveau thread, mais cette fois vous allez également exécuter la mise à jour du bouton dans l'EDT.

Voilà donc ce que nous obtenons :

```
public class Test1
{
    static int count = 0;
    static JButton bouton = new JButton("Pause");

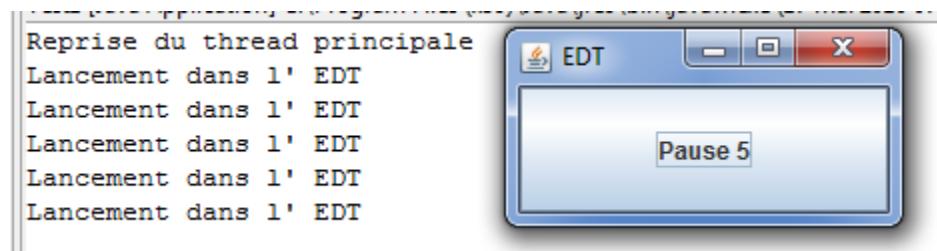
    public static void updateBouton()
    {
        new Thread(new Runnable()
        {
            public void run()
            {
                for(int i = 0; i < 5; i++)
                {
                    try { Thread.sleep(1000); }
                    catch (InterruptedException e) { e.printStackTrace(); }

                    // modification de notre composant dans l'EDT
                    Thread t = new Thread(new Runnable()
                    {
                        public void run()
                        {
                            bouton.setText("Pause " + ++count);
                        }
                    });
                }
            }

            if ( SwingUtilities.isEventDispatchThread() )
                t.start();
            else
            {
                System.out.println("Lancement dans l' EDT");
                SwingUtilities.invokeLater(t);
            }
        });
    }).start();
}

public static void main(String[] args)
{
    JFrame fen = new JFrame("EDT");
    fen.getContentPane().add(bouton);
    fen.setSize(200, 100);
    fen.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    fen.setLocationRelativeTo(null);
    fen.setVisible(true);
    updateBouton();
    System.out.println("Reprise du thread principal");
}
```

Le rendu correspond à la figure suivante.



- Depuis Java 6, la classe **SwingWorker(<T, V>)** vous offre la possibilité de lancer des traitements dans un thread en vous assurant que les mises à jour des composants se feront dans l'EDT.

Vu que cette classe est abstraite, vous allez devoir redéfinir une méthode : **doInBackground()**. Elle permet de redéfinir ce que doit faire l'objet en tâche de fond. Une fois cette tâche effectuée, la méthode **doInBackground()** prend fin.

Vous avez la possibilité de redéfinir la méthode **done()**, qui a pour rôle d'interagir avec votre IHM tout en s'assurant que ce sera fait dans l'EDT. Implémenter la méthode **done()** est optionnel.

Voici un exemple d'utilisation :

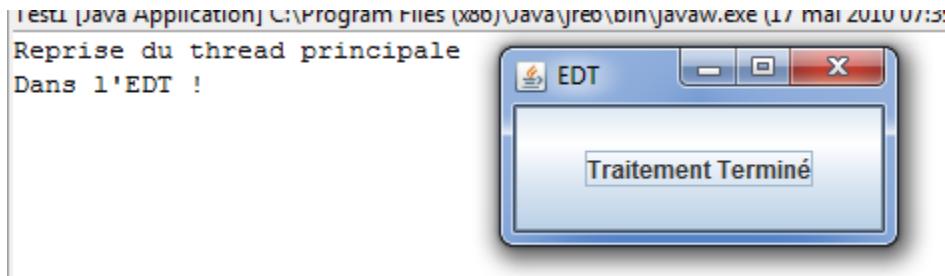
```
public class Test1
{
    static int count = 0;
    static JButton bouton = new JButton("Pause");

    public static void updateBouton()
    {
        SwingWorker sw = new SwingWorker() // crée le SwingWorker
        {
            protected Object doInBackground() throws Exception
            {
                for(int i = 0; i < 5; i++)
                {
                    try { Thread.sleep(1000); }
                    catch (InterruptedException e) { e.printStackTrace(); }
                }
                return null;
            }

            public void done()
            {
                if ( SwingUtilities.isEventDispatchThread() )
                    System.out.println("Dans l'EDT ! ");
                bouton.setText("Traitement terminé");
            }
        };
        sw.execute(); // lance le SwingWorker
    }
}
```

```
public static void main(String[] args)
{
    JFrame fen = new JFrame("EDT");
    fen.getContentPane().add(bouton);
    fen.setSize(200, 100);
    fen.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    fen.setLocationRelativeTo(null);
    fen.setVisible(true);
    updateBouton();
    System.out.println("Reprise du thread principal");
}
```

Le traitement se fait bien en tâche de fond et votre composant est mis à jour dans l'EDT.



## JDBC : la porte d'accès aux bases de données

La **JDBC** (Java DataBase Connectivity) contient de classes Java permettant de se connecter et d'interagir avec des bases de données.

### Rappel sur les bases de données

Les bases de données (BDD) permettent de stocker des données. Il s'agit d'un système de fichiers contenant les données de votre application.

Cependant, ces fichiers sont totalement transparents pour l'utilisateur d'une base de données, donc totalement transparents pour vous ! La différence avec les fichiers classiques se trouve dans le fait que ce n'est pas vous qui les gérez : c'est votre BDD qui les organise, les range et, le cas échéant, vous retourne les informations qui y sont stockées.

De plus, plusieurs utilisateurs peuvent accéder simultanément aux données dont ils ont besoin, le tout en réseau. Imaginez-vous gérer tout cela manuellement alors que les BDD le font automatiquement...

Les données sont ordonnées par « **tables** », c'est-à-dire par regroupements de plusieurs valeurs. C'est vous qui créerez vos propres tables, en spécifiant quelles données vous souhaiterez y intégrer.

Une base de données peut être vue comme une gigantesque armoire à tiroirs dont vous spécifiez les noms et qui contiennent une multitude de fiches dont vous spécifiez aussi le contenu.



Exemple : dans cette base de données, nous trouvons deux tables : une dont le rôle est de stocker des informations relatives à des personnes (noms, prénoms et âges) ainsi qu'une autre qui s'occupe de stocker des pays, avec leur nom et leur capitale.

Vous pouvez même **interroger** vos BDD en leur posant des questions via un langage précis. Le langage permettant d'interroger des BDD est le langage **SQL** (*Structured Query Language*).

Exemple : vous pouvez interroger votre BDD en lui donnant les instructions suivantes

- « Donne-moi la fiche de la table Personne pour le nom HERBY » ;
- « Donne-moi la fiche de la table Pays pour le pays France » ;
- etc.

Pour utiliser une BDD, vous avez besoin de deux éléments :

- la **base de données**
- le **SGBD (Système de Gestion de Base de Données)**.

Il existe plusieurs bases de données : **PostgreSQL** ; **MySQL** ; **SQL Server** ; **Oracle** ; **Access** (liste non exhaustive). Certaines sont payantes (Oracle), d'autres sont plutôt permissives avec les données qu'elles contiennent (MySQL), d'autres encore sont dotées d'un système de gestion très simple à utiliser (MySQL), etc.

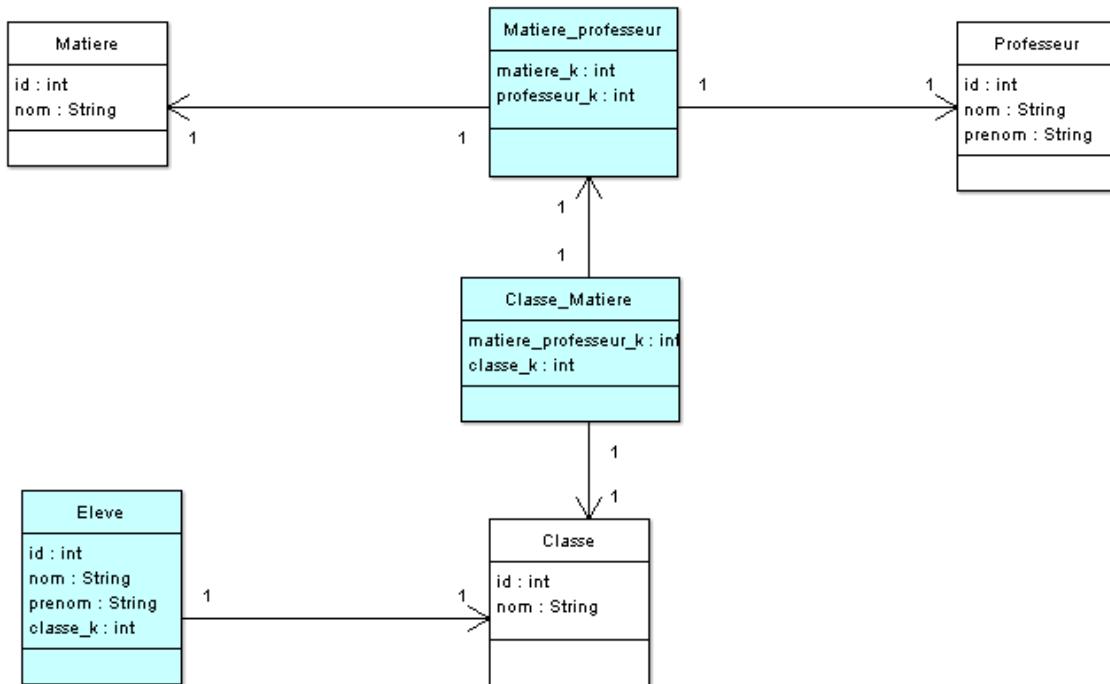
Mon choix s'est porté sur **PostgreSQL** qui est gratuit et complet. Son SGBD associé est **pgAdminIII**.

Les bases de données servent à stocker des informations. Pour ranger correctement nos informations, nous devrons les analyser.

#### Exemple : gérer une école

- cette école est composée de classes ;
- chaque classe est composée d'élèves ;
- à chaque classe est attribué un professeur pour chacune des matières dispensées ;
- un professeur peut enseigner plusieurs matières et exercer ses fonctions dans plusieurs classes.

En théorie, nous devrions établir un **dictionnaire** des données, vérifier à qui appartient quelle donnée, poursuivre avec une modélisation à la façon **MCD** (Modèle Conceptuel de Données) et simplifier le tout selon certaines règles, pour terminer avec un **MPD** (Modèle Physique de Données).



Tous ces éléments correspondent à **nos futures tables** ; les attributs qui s'y trouvent se nomment des « **champs** ». Tous les acteurs mentionnés figurent dans ce schéma (classe, professeur, élève...).

Vous constatez que chaque acteur possède un attribut nommé « id » correspondant à son identifiant : c'est un champ de type *entier* qui s'incrémentera à chaque nouvelle entrée ; c'est également grâce à ce champ que nous pouvons créer des liens entre les acteurs.

Vous devez savoir que les **flèches** du schéma signifient « **a un** » ; de ce fait, un élève « a une » classe.

Certaines tables contiennent un champ se terminant par « \_k ».

Quelques-unes de ces tables possèdent deux champs de cette nature, pour une raison très simple : parce que nous avons décidé qu'un professeur pouvait enseigner plusieurs matières, nous avons alors besoin de ce qu'on appelle une « **table de jointures** ». Ainsi, nous pouvons spécifier que tel professeur enseigne telle ou telle matière et qu'une association professeur/matière est assignée à une classe. Ces liens se feront par les identifiants (*id*).

De plus, chaque champ possède un type (int, double, date, boolean...).

Pour créer une base de données on va utiliser l'outil **pgAdminIII**.

Code SQL pour créer une nouvelle base de données

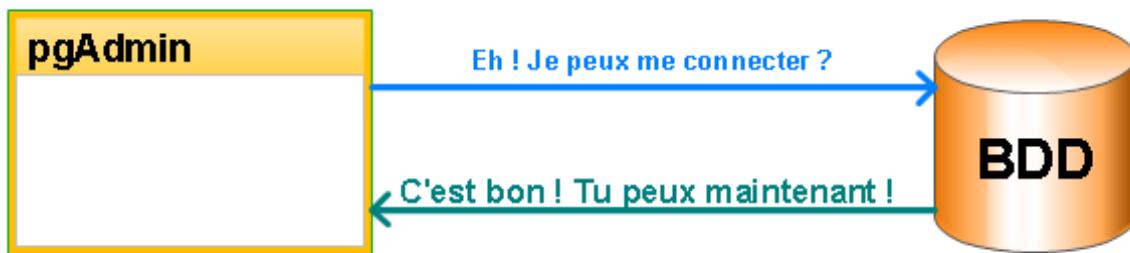
```
CREATE DATABASE "Ecole"  
    WITH OWNER = postgres  
        ENCODING =      'UTF8';
```

Code SQL pour créer un tableau

```
CREATE TABLE classe (  
    cls_id integer NOT NULL,  
    cls_nom character varying(64) NOT NULL  
);
```

### Se connecter à la base de données

Beaucoup de choses se passent entre **pgAdmin** et **PostgreSQL** (les termes « PostgreSQL » et « Postgres » sont souvent indifféremment utilisés) ! En effet, le premier est un programme qui établit une connexion avec la BDD afin qu'ils puissent communiquer. **pgAdmin** utilise donc un driver pour se connecter à la base de données.



Pour vous connecter à une base de données, il vous faut un **fichier .jar** qui correspond au fameux pilote et qui contient tout ce dont vous aurez besoin pour vous connecter à une base PostgreSQL. Il existe un fichier .jar pour se connecter à chacune des bases de données : MySQL ; SQL Server ; Oracle ; d'autres bases.

Pour trouver le **driver JDBC** : une recherche sur votre moteur de recherche répondra à vos attentes.

Une fois l'archive téléchargé **il faut la placer** :

- soit dans votre projet et l'ajouter au CLASSPATH ;
- soit dans le dossier lib/ext présent dans le dossier d'installation du JRE.

Le tout est de savoir si votre application est vouée à être exportée sur différents postes; dans ce cas, l'approche CLASSPATH est la plus judicieuse (sinon, il faudra ajouter l'archive dans tous les JRE...).

En ce qui nous concerne, nous utiliserons la deuxième méthode afin de ne pas surcharger nos projets.

La base de données est prête, les tables sont créées, remplies et nous possédons le driver nécessaire !

Il ne nous reste plus qu'à nous connecter. Voici le code source permettant la connexion :

```
public class Connect
{
    public static void main(String[] args)
    {
        try
        {
            Class.forName("org.postgresql.Driver");
            System.out.println("Driver O.K.");

            String url = "jdbc:postgresql://localhost:5432/Ecole";
            String user = "postgres";
            String passwd = "postgres";

            Connection conn = DriverManager.getConnection(url, user, passwd);
            System.out.println("Connexion effective !");
        }
        catch (Exception e) { e.printStackTrace(); }
    }
}
```

Cette procédure lève une exception en cas de problème (mot de passe invalide, ...).

Dans un premier temps, nous avons créé une instance de l'objet **Driver** présent dans le fichier **.jar** que nous avons téléchargé.

À ce stade, il existe comme un pont entre votre programme Java et votre BDD, mais le trafic routier n'y est pas encore autorisé : il faut qu'une connexion soit effective afin que le programme et la base de données puissent communiquer. Cela se réalise grâce à cette ligne de code :

```
Connection conn = DriverManager.getConnection(url, user, passwd);
```

Nous avons défini au préalable trois String contenant respectivement :

- l'URL de connexion ;
- le nom de l'utilisateur ;
- le mot de passe utilisateur.

L'URL de connexion est indispensable à Java pour se connecter à n'importe quelle BDD. L'URL se décompose en 3 blocs :

**jdbc:postgresql://localhost:5432/Ecole**

- le premier bloc contient des informations concernant le pilote JDBC. Cela permet à Java de savoir quel pilote utiliser. Cet bloc commence toujours par **jdbc:** ; dans notre cas, il est suivi par la dénomination **postgresql:** (car nous utilisons PostgreSQL)
- le deuxième bloc contient la localisation de la machine physique sur le réseau, suivi du numéro de port utilisé ; ici, nous travaillons en local, nous utilisons donc **//localhost:5432**.
- le dernier bloc contient le nom de notre base de données.

## Fouiller dans sa base de données

- Les recherches en BDD se font via les objets **Statement** et **ResultSet**.

Exemple d'exécution d'une requête SQL et récupération des lignes retournées

```
public static void main(String[] args)
{
    try
    {
        Class.forName("org.postgresql.Driver");

        String url = "jdbc:postgresql://localhost:5432/Ecole";
        String user = "postgres";
        String passwd = "postgres";

        Connection conn = DriverManager.getConnection(url, user, passwd);

        Statement state = conn.createStatement();           // création d'un objet Statement
        ResultSet result = state.executeQuery("SELECT * FROM classe"); // le résultat
        ResultSetMetaData resMeta = result.getMetaData(); // les MetaData

        System.out.println("\n*****\n*****\n*****\n*****\n*****");

        // affiche le nom des colonnes
        for ( int i = 1; i <= resMeta.getColumnCount(); i++ )
            System.out.print("\t" + resMeta.getColumnName(i).toUpperCase() + "\t ");

        System.out.println("\n*****\n*****\n*****\n*****\n*****");

        while ( result.next() )
        {
            for ( int i = 1; i <= resMeta.getColumnCount(); i++ )
                System.out.print("\t" + result.getObject(i).toString() + "\t |");
            System.out.println("\n-----");
        }

        result.close();
        state.close();
    }
    catch (Exception e) { e.printStackTrace(); }
}
```

| CLS_ID | CLS_NOM | CLS_NOM_2 |
|--------|---------|-----------|
| 1      | 6° A    |           |
| 2      | 6° B    |           |
| 3      | 6° C    |           |
| 4      | 5° A    |           |
| 5      | 5° B    |           |
| 6      | 5° C    |           |
| 7      | 4° A    |           |
| 8      | 4° B    |           |
| 9      | 4° C    |           |
| 10     | 3° A    |           |
| 11     | 3° B    |           |
| 12     | 3° C    |           |

Les choses se sont déroulées en quatre étapes distinctes :

- création de l'objet **Statement** ;
- exécution** de la requête SQL ;
- récupération** et **affichage** des données via l'objet **ResultSet** ;
- fermeture des objets** utilisés (bien que non obligatoire, c'est recommandé).

- L'objet **Statement** permet d'exécuter des instructions SQL, il interroge la base de données et retourne les résultats. Ensuite, ces résultats sont stockés dans l'objet **ResultSet**, grâce auquel on peut parcourir les lignes de résultats et les afficher.

Les requêtes SQL exécutées par l'objet **Statement** peuvent être de différents types : CREATE, INSERT, UPDATE, SELECT, DELETE.

La méthode **next()** de l'objet **ResultSet** permet de positionner l'objet sur la ligne suivante de la liste de résultats.

Dans le cas où l'on connaît le contenu des colonnes : (pas besoin de passer par les métadonnées)

```
while ( result.next() )
{
    System.out.print("\t" + result.getInt("cls_id") + "\t |");
    System.out.print("\t" + result.getString("cls_nom") + "\t |");
    System.out.println("\n-----");
```

Il existe une méthode **get\_\_()** par type primitif et quelques types SQL :

`getArray(int); getAscii(int); getBigDecimal(int); getBinary(int); getBlob(int); getBoolean(int);  
getBytes(int); getCharacter(int); getDate(int); getDouble(int); getFloat(int); getInt(int);  
getLong(int); getObject(int); getString(int).`

- L'objet de type **ResultSetMetaData** permet de récupérer les métadonnées de ma requête, c'est à dire ses informations globales. J'ai ensuite utilisé cet objet afin de récupérer le nombre de colonnes renvoyé par la requête SQL ainsi que leur nom.

Cet objet de métadonnées permet de récupérer des informations très utiles, comme :

- le nombre de colonnes d'un résultat ;
- le nom des colonnes d'un résultat ;
- le type de données stocké dans chaque colonne ;
- le nom de la table à laquelle appartient la colonne (dans le cas d'une jointure de tables) ;
- etc.

**! Attention :** les indices de colonnes SQL commencent à 1 !

- Il existe aussi des objets **DataBaseMetaData** donnant accès à des informations globales sur une base de données
- Exemple de requêtes SQL sur les **tables jointes**

```
public static void main(String[] args)
{
    try
    {
        ...
        Statement state = conn.createStatement();

        String query = "SELECT prof_nom, prof_prenom, mat_nom FROM professeur";
        query += " INNER JOIN j_mat_prof ON jmp_prof_k = prof_id";
        query += " INNER JOIN matiere ON jmp_mat_k = mat_id ORDER BY prof_nom";
```

```

ResultSet result = state.executeQuery(query);

String nom = "";
while ( result.next() )
{
    if ( ! nom.equals(result.getString("prof_nom")) )
    {
        nom = result.getString("prof_nom");
        System.out.println(nom + " " + result.getString("prof_prenom") + " enseigne : ");
    }
    System.out.println("\t\t\t - " + result.getString("mat_nom"));
}

result.close();
state.close();
}
catch (Exception e) { e.printStackTrace(); }
}

```

```

Problems @ Javadoc Declaration Console
<terminated> Exo2 [Java Application] C:\Program Files\Java\jdk-11.0.2\bin\javac.exe
BADEN Baden enseigne :
                    - Français
                    - Sport
BORA Kernel enseigne :
                    - Anglais
CAISSE Jean enseigne :
                    - Physique
JADEN Boudy enseigne :
                    - Français
                    - Sport
MAMOU Daniel enseigne :
                    - Biologie
                    - Mathématiques
MIOU Miou enseigne :
                    - Anglais
MOISSAT Marc enseigne :
                    - Physique
SACRE Sophie enseigne :
                    - Biologie
                    - Mathématiques

```

- **Paramètres** pour l'initialisation des objets **Statement**

Vous pouvez spécifier des paramètres pour la création de l'objet Statement. Ces paramètres sont des variables statiques de la classe ResultSet. Ces paramètres permettent différentes actions lors du parcours des résultats via l'objet ResultSet.

Le premier paramètre est utile pour la lecture du jeu d'enregistrements :

- **TYPE\_FORWARD\_ONLY** : le résultat n'est consultable qu'en avançant dans les données renvoyées, il est donc impossible de revenir en arrière lors de la lecture ;
- **TYPE\_SCROLL\_SENSITIVE** : le parcours peut se faire vers l'avant ou vers l'arrière et le curseur peut se positionner n'importe où, et si des changements surviennent dans la base pendant la lecture, ils seront directement visibles lors du parcours des résultats ;
- **TYPE\_SCROLL\_INSENSITIVE** : à la différence du précédent, les changements ne seront pas visibles.

Le second concerne la possibilité de mise à jour du jeu d'enregistrements :

- **CONCUR\_READONLY** : les données sont consultables en lecture seule, c'est-à-dire que l'on ne peut modifier des valeurs pour mettre la base à jour ;
- **CONCUR\_UPDATABLE** : données modifiables ; lors d'une modification, base mise à jour.

Par défaut, les **ResultSet** issus d'un Statement sont de type **TYPE\_FORWARD\_ONLY** pour le parcours et **CONCUR\_READONLY** pour les actions réalisables.

Voici comment créer un Statement permettant à l'objet ResultSet de pouvoir être lu d'avant en arrière avec possibilité de modification :

```
Statement state = conn.createStatement(resultSet.TYPE_SCROLL_INSENSITIVE,  
                                     resultSet.CONCUR_UPDATABLE);
```

- **Les requêtes préparées**

Il existe un autre objet qui fonctionne de la même manière que ResultSet mais qui précompile la requête et permet d'utiliser un système de requête à trous : l'objet **PreparedStatement**

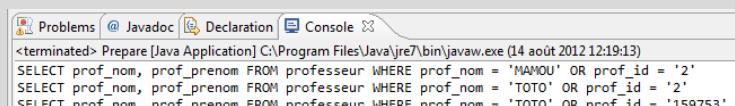
Une requête SQL précompilée a pour effet de réduire le temps d'exécution dans le moteur SQL de la BDD. En règle générale, on utilise ce genre d'objet pour des requêtes contenant beaucoup de paramètres ou pouvant être exécutées plusieurs fois.

Il existe une autre différence de taille entre les objets PreparedStatement et Statement : dans le premier, on peut utiliser **des paramètres à trous** !

En fait, vous pouvez **insérer un caractère spécial** dans vos requêtes et remplacer ce caractère grâce à des méthodes de l'objet PreparedStatement en spécifiant sa place et sa valeur (son type étant défini par la méthode utilisée).

Voici un exemple :

```
public static void main(String[] args)  
{  
    try  
    {  
        ...  
        // on crée la requête  
        String query = "SELECT prof_nom, prof_prenom FROM professeur";  
        query += " WHERE prof_nom = ?";           // premier trou pour le nom du professeur  
        query += " OR prof_id = ?";                // deuxième trou pour l'id du professeur  
  
        // on crée l'objet avec la requête en paramètre  
        PreparedStatement prepare = conn.prepareStatement(query);  
        prepare.setString(1, "MAMOU");             // remplace le 1er trou par le nom du professeur  
        prepare.setInt(2, 2);                      // remplace le 2eme trou par l'id du professeur  
        System.out.println(prepare.toString());     // affiche la requête exécutée  
  
        prepare.setString(1, "TOTO");              // on modifie le premier trou  
        System.out.println(prepare.toString());      // affiche la requête exécutée  
        prepare.setInt(2, 159753);                 // on modifie le deuxième trou  
        System.out.println(prepare.toString());      // affiche la requête exécutée  
  
        prepare.close();  
    } catch (Exception e) { e.printStackTrace(); }  
}
```



- Les méthodes de l'objet ResultSet

L'objet **ResultSet** offre beaucoup de méthodes permettant d'explorer les résultats, à condition que vous ayez bien préparé l'objet Statement.

Vous avez la possibilité de :

- vous placer avant la première ligne de votre résultat : **res.beforeFirst()**
- vous placer sur la première ligne de votre résultat : **res.first()**
- vous placer sur la dernière ligne : **res.last()**
- vous placer après la dernière ligne de résultat : **res.afterLast()**
  
- savoir si vous vous trouvez avant la première ligne : **res.isBeforeFirst()**
- savoir si vous vous trouvez sur la première ligne : **res.isFirst()**
- savoir si vous vous trouvez sur la dernière ligne : **res.isLast()**
- savoir si vous vous trouvez après la dernière ligne : **res.isAfterLast()**
  
- aller de la première ligne à la dernière : **res.next()**
- aller de la dernière ligne à la première : **res.previous()**
  
- vous positionner sur une ligne précise de votre résultat : **res.absolute(5)**
- vous positionner sur une ligne par rapport à l'emplacement actuel : **res.relative(-3)**

Je vous ai concocté un morceau de code que j'ai commenté et qui met tout cela en oeuvre.

```
public static void main(String[] args)
{
    try {
        ...
        Statement state = conn.createStatement	ResultSet.TYPE_SCROLL_SENSITIVE,
   ResultSet.CONCUR_UPDATABLE);

        String query = "SELECT prof_nom, prof_prenom FROM professeur";
        ResultSet res = state.executeQuery(query);
        int i = 1;

        System.out.println("\n\t-----");
        System.out.println("\tLECTURE STANDARD.");
        System.out.println("\t-----");

        while ( res.next() )
        {
            System.out.println("\tNom :" + res.getString("prof_nom") + "\t prénom:"
                               + res.getString("prof_prenom"));
            if ( res.isLast() )
                System.out.println("\t\t* DERNIER RESULTAT !\n");
            i++;
        }
    }
}
```

```

// une fois la lecture terminée, on contrôle si on se trouve bien à l'extérieur des lignes de résultat
if ( res.isAfterLast() )
    System.out.println("\tNous venons de terminer !\n");

System.out.println("\t-----");
System.out.println("\tLecture en sens contraire.");
System.out.println("\t-----");

// on se trouve alors à la fin; on peut parcourir le résultat en sens contraire
while ( res.previous() )
{
    System.out.println("\tNom :" + res.getString("prof_nom") + "\t prénom:"
                      + res.getString("prof_prenom"));
    // on regarde si on se trouve sur la première ligne du résultat
    if ( res.isFirst() )
        System.out.println("\t\t* RETOUR AU DEBUT !\n");
}

// on regarde si on se trouve avant la première ligne du résultat
if ( res.isBeforeFirst() )
    System.out.println("\tNous venons de revenir au début !\n");

System.out.println("\t-----");
System.out.println("\tAprès positionnement absolu du curseur à la place "+ i/2 +" .");
System.out.println("\t-----");

// on positionne le curseur sur la ligne i/2, peu importe où on se trouve
res.absolute(i/2);

while ( res.next() )
    System.out.println("\tNom :" + res.getString("prof_nom") + "\tprénom: "
                      + res.getString("prof_prenom"));

System.out.println("\t-----");
System.out.println("\tAprès pos relatif du curseur à la place"+(i-(i-2)) + ".");
System.out.println("\t-----");

// on place le curseur à la ligne actuelle moins i-2;
// si on n'avait pas mis de signe moins, on aurait avancé de i-2 lignes
res.relative(-(i-2));
while ( res.next() )
    System.out.println("\tNom :" + res.getString("prof_nom") + "\tprénom: "
                      + res.getString("prof_prenom"));
res.close();
state.close();
} catch (Exception e) { e.printStackTrace(); }
}

```

Lorsque vous souhaitez placer le curseur sur la première ligne, vous devez utiliser **absolute(1)** quel que soit l'endroit où vous vous trouvez ! En revanche, cela nécessite que le ResultSet soit de type TYPE\_SCROLL\_SENSITIVE ou TYPE\_SCROLL\_INSENSITIVE, sans quoi vous aurez une exception.

- **Modifier des données**

Avec un ResultSet autorisant l'édition des lignes, vous pouvez invoquer la méthode **update\_\_()** pour changer la valeur d'un champ.

Exemple de méthodes de mise-à-jour :

- **updateFloat(String nomColonne, float value)**
- **updateString(String nomColonne, String value)**

Changer la valeur d'un champ est donc très facile.

Cependant, il faut, en plus de changer les valeurs, valider ces changements pour qu'ils soient effectifs : cela se fait par la méthode **updateRow()**.

De la même manière, vous pouvez annuler des changements grâce à la méthode **cancelRowUpdates()**. Sachez que si vous devez annuler des modifications, vous devez le faire avant la méthode de validation, sinon l'annulation sera ignorée.

**Exemple** de mise-à-jour :

```
public static void main(String[] args)
{
    try {
        ...
        // on autorise la mise à jour des données et la mise à jour de l'affichage
        Statement state = conn.createStatement(resultSet.TYPE_SCROLL_INSENSITIVE,
  resultSet.CONCUR_UPDATABLE);
        // on va chercher une ligne dans la base de données
        String query = "SELECT * FROM professeur WHERE prof_nom = 'MAMOU'";

        ResultSet res = state.executeQuery(query);

        //on affiche ce que l'on trouve
        res.first();
        System.out.println("NOM : " + res.getString("prof_nom") + " - PRENOM : "
                           + res.getString("prof_prenom"));

        // on met à jour les champs
        res.updateString("prof_nom", "COURTEL");
        res.updateString("prof_prenom", "Angelo");
        res.updateRow();                                // on valide

        // on affiche les modifications
        System.out.println("*****");
        System.out.println("APRES MODIFICATION : ");
        System.out.println("\tNOM : " + res.getString("prof_nom") + " - PRENOM : "
                           + res.getString("prof_prenom") + "\n");
        res.close();
        state.close();
    } catch (Exception e) { e.printStackTrace(); }
}
```

- **La création ou la suppression de données**

Vous pouvez utiliser la méthode `executeUpdate(String query)`.

```
Statement state = conn.createStatement	ResultSet.TYPE_SCROLL_SENSITIVE,
	ResultSet.CONCUR_UPDATABLE);

state.executeUpdate("INSERT INTO professeur (prof_nom, prof_prenom)
VALUES ('SALMON', 'Dylan')");

state.executeUpdate("DELETE FROM professeur WHERE prof_nom = 'MAMOU'");
```

On peut également utiliser cette méthode avec des requêtes préparées

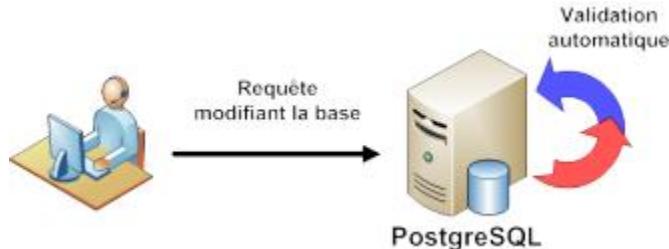
```
PreparedStatement prepare = conn.prepareStatement("UPDATE professeur set
prof_prenom = ? "+"WHERE prof_nom = 'MAMOU'");

prepare.setString(1, "Gérard"); // on paramètre notre requête
prepare.executeUpdate(); // on l'exécute
prepare.close();
```

- **Gérer les transactions manuellement**

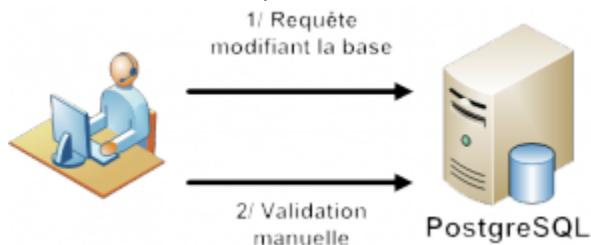
Certains moteurs SQL comme PostgreSQL vous proposent de **gérer vos requêtes SQL** grâce à ce que l'on appelle des **transactions**.

Lorsque vous insérez, modifiez ou supprimez des données dans PostgreSQL, il se produit un événement automatique : la **validation des modifications par le moteur SQL**.



Lorsque vous exécutez une requête de type INSERT, CREATE, UPDATE ou DELETE, le type de cette requête modifie les données présentes dans la base. Une fois qu'elle est exécutée, le moteur SQL valide directement ces modifications !

Cependant, vous pouvez avoir la main sur ce point, afin de maîtriser **l'intégrité de vos données**



Pour gérer manuellement les transactions, on spécifie au moteur SQL de ne pas valider automatiquement les requêtes SQL grâce à la méthode `setAutoCommit(boolean)`

```
Connection conn = DriverManager.getConnection(url, user, passwd);
conn.setAutoCommit(false);
```

En mode `setAutoCommit(false)`, si vous ne validez pas vos requêtes, elles ne seront pas prises en compte. Si vous souhaitez que vos requêtes soient prises en compte, il vous faut les valider en utilisant la méthode `conn.commit()` (avant la fin du script).

Vous pouvez revenir à tout moment au mode de validation automatique grâce à `setAutoCommit(true)`

## Limiter le nombre de connexions

- Pour économiser les ressources, vous ne devriez créer qu'un seul objet de connexion.

Le **pattern singleton** permet de disposer d'une instance unique d'un objet. Ce pattern repose sur un **constructeur privé** associé à une méthode retournant l'instance créée dans la classe elle-même.

Classe qui permet de s'assurer qu'il soit impossible de créer plus d'un objet de connexion

```
public class SdzConnection
{
    private String url = "jdbc:postgresql://localhost:5432/Ecole";
    private String user = "postgres";
    private String passwd = "postgres";

    private static Connection connect; // constructeur privé

    private SdzConnection() {
        try {
            connect = DriverManager.getConnection(url, user, passwd);
        }
        catch (SQLException e) { e.printStackTrace(); }
    }

    // méthode qui va nous retourner notre instance et la créer si elle n'existe pas
    public static Connection getInstance()
    {
        if ( connect == null )
            new SdzConnection(); // appel constructeur
        return connect;
    }
}
```

- Afin de pallier au problème du multithreading, il vous suffit d'utiliser le mot clé synchronized dans la déclaration de votre méthode de récupération de l'instance, mais cette synchronisation n'est utile qu'une fois. À la place, vous pouvez instancier l'objet au chargement de la classe par la JVM, avant tout appel à celle-ci.

## Annexe : La documentation Java

**Javadoc**, c'est avant tout un outil, développé par Sun Microsystems. Mais par analogie, c'est aussi la documentation générée par cet outil. Cet outil permet, en inspectant le code Java des classes, de produire une documentation web très complète de votre code.

L'outil génère ainsi des pages **HTML** contenant au minimum la liste des classes, la liste des méthodes et la liste des variables. Nous verrons ensuite qu'il est possible d'ajouter tout un tas d'informations, de commentaires, afin de générer une véritable documentation, exhaustive et facilement lisible.

L'avantage du **format HTML** utilisé dans la documentation produite tient dans la présence de liens hypertextes. Il devient très facile de naviguer dans la documentation, au fil de la lecture.

En effet, l'outil **Javadoc** utilise des **tags** mis dans le code pour compléter la documentation générée.

Pour rappel, il existe **trois types de commentaires** en Java : commentaires en ligne, commentaires sur plusieurs lignes et les commentaires Javadoc.

Voici un exemple présentant ces trois types de commentaire.

```
/**  
 * Ceci est un commentaire Javadoc.  
 * Il commence par un slash suivis de deux étoiles.  
 * Chaque ligne doit ensuite commencer par une étoile.  
 * Enfin, il fini par une étoile suivie d'un slash.  
 */  
protected Vector<Zero> getVectorAmis()  
{  
    // Ceci est un commentaire sur une ligne  
    Vector<Zero> vector = new Vector<Zero>();  
    /* Ceci est un commentaire sur  
    plusieurs lignes */  
    for (Zero z : listeAmis)  
        vector.add(z);  
    return vector;  
}
```

Donc toutes les informations que nous mettrons se trouveront dans ce commentaire Javadoc. Il doit se **situer sur la ligne immédiatement avant le nom** de la classe, de la méthode, ou de la variable.

Il y a 9 tags JavaDoc :

- **@param**

Sert à renseigner le ou les paramètres de la méthode

Derrière le tag il faut renseigner le nom du paramètre (son type sera inclus automatiquement)

```
/**  
 * Met à jour le niveau du membre.  
 *  
 * @param level  
 *      Le nouveau level du membre.  
 */  
protected void setLevel(SDZLevel level)  
{  
    this.level = level;  
}
```

- **@return**

Sert à renseigner l'objet retourné par la méthode

```
/**  
 * Retourne le level du zéro.  
  
 * @return Une instance de SDZLevel, qui correspond à niveau du membre sur SDZ.  
 */  
public SDZLevel getLevel()  
{  
    return level;  
}
```

- **@throws**

Indique la présence d'une exception qui sera propagée si elle se lève.

Il faut bien indiquer le type de l'exception, et la raison de l'exception.

```
/**  
 * Retourne l'adresse du profil du Zero.  
  
 * @return L'URL du profil du Zero, générée à partir de l'id du Zero.  
 * @throws MalformedURLException Si jamais l'url est mal formée.  
 */  
public URL getURLProfil() throws MalformedURLException  
{  
    URL url = new URL("http://www.siteduzero.com/membres-294-"+id+".html");  
    return url;  
}
```

- **@author , @version**

Le tag **@author** renseigne le nom de l'auteur de la classe.

Le tag **@version** indique le numéro de version de la classe; utilisé ensuite par le tag **@since**

**Important** : ces tags ne peuvent être utilisés que pour une classe ou une interface (pas méthode)

```
/*  
 * Zero est la classe représentant un membre du Site du Zéro.  
  
 * @author dworkin  
 * @version 3.0  
 */  
public class Zero  
{  
    ...  
}
```

- **@see**

Permet de faire une référence à une autre méthode, classe, etc. Concrètement, cela se symbolisera par un lien hypertexte dans la Javadoc. C'est donc un des tags les plus importants.

```
/**  
 * Le "level" du Zéro. Ce "level" peut être modifié.  
 *  
 * @see SDZLevel  
 */  
private SDZLevel level;
```

- **@since**

Permet de dater la présence d'une méthode, d'un paramètre.

Derrière ce tag, il faut noter un numéro de version de la classe.

```
/**  
 * Met à jour le pseudo du membre.  
 *  
 * @since 3.0  
 */  
public void setPseudo(String pseudo)  
{  
    this.pseudo = pseudo;  
}
```

- **@serial**

- **@deprecated**

Doit décrire la version depuis laquelle cette méthode / classe est dépréciée. Mais aussi ce qu'il faut utiliser à la place.

```
/**  
 * Retourne la liste des amis du zéro.  
 *  
 * @deprecated Depuis Java 1.4, remplacé par getListeAmis()  
 */  
protected Vector<Zero> getVectorAmis()  
{  
    Vector<Zero> vector = new Vector<Zero>();  
    for (Zero z : listeAmis)  
        vector.add(z);  
    return vector;  
}
```

Il existe des **conventions pour la mise en forme de la Javadoc**. Elles concernent les tags d'une part, et les commentaires d'autre part.

Voici les principaux :

- La première phrase Javadoc doit être une courte description de la classe / méthode / variable, etc
- Les phrases doivent être relativement courtes.
- Il faut utiliser la troisième personne pour commenter une méthode.
- Il faut détailler le fonctionnement (l'algorithme) des méthodes si besoin.
- Utiliser "ce" plutôt que "le".
- Les tags **@param** et **@return** doivent être systématiquement indiqués (sauf méthodes sans paramètres ou méthodes void).

L'on peut utiliser du code HTML dans les commentaires Javadoc. Mais il faut les utiliser seulement pour des choses simples. Les balises les plus utilisées sont **<b>**, **<i>**, **<ul>** et **<p>**.

Générer la JavaDoc avec **Eclipse** : Project -> Generate JavaDoc -> path javadoc.exe

S'il n'y a pas d'erreurs, votre documentation est prête.

Par défaut, elle se trouve dans le répertoire de votre projet, dans un dossier "doc".

### Un exemple complet :

```
/**  
 * <b>Zero</b> est la classe représentant un membre du Site du Zéro.</b>  
 * <p>  
 * Un membre du SDZ est caractérisé par les informations suivantes :  
 * <ul>  
 * <li>Un identifiant unique attribué définitivement.</li>  
 * <li>Un pseudo, susceptible d'être changé.</li>  
 * <li>Un "level". Il peut être "zéro", newser, validateur, modérateur, etc.</li>  
 * </ul>  
 * </p>  
 * <p>  
 * De plus, un Zéro a une liste d'amis Zéro. Le membre pourra ajouter ou enlever  
 * des amis à cette liste.  
 * </p>  
 *  
 * @see SDZLevel  
 *  
 * @author dworkin  
 * @version 3.0  
 */  
public class Zero  
{
```

```

    /**
     * L'ID du Zéro. Cet ID n'est pas modifiable.
     *
     * @see Zero#Zero(int, String)
     * @see Zero#getId()
     */
    private int id;

    /**
     * Le pseudo du Zéro. Ce pseudo est changeable.
     *
     * @see Zero#getPseudo()
     * @see Zero#setPseudo(String)
     */
    private String pseudo;

    /**
     * Le "level" du Zéro. Ce "level" peut être modifié.
     * <p>
     * Pour de plus amples informations sur les "levels" possibles, regardez la
     * documentation de la classe SDZLevel.
     * </p>
     *
     * @see SDZLevel
     *
     * @see Zero#getLevel()
     * @see Zero#setLevel(SDZLevel)
     */
    private SDZLevel level;

    /**
     * La liste des amis du Zéro.
     * <p>
     * Il est possible d'ajouter ou de retirer des amis dans cette liste.
     * </p>
     *
     * @see Zero#getListeAmis()
     * @see Zero#ajouterAmi(Zero)
     * @see Zero#retirerAmi(Zero)
     */
    private List<Zero> listeAmis;

    /**
     * Constructeur Zero.
     * <p>
     * A la construction d'un objet Zéro, le "level" est fixé à SDZLevel.ZERO,
     * ce qui correspond au niveau d'un membre.
     * De plus la liste des amis est créée vide,
     * </p>
     *
     * @param id
     *          L'identifiant unique du Zéro.
     * @param pseudo
     *          Le pseudo du Zéro.
     *
     * @see Zero#id

```

```

* @see Zero#pseudo
* @see Zero#level
* @see Zero#listeAmis
*/
public Zero(int id, String pseudo) {
    this.id = id;
    this.pseudo = pseudo;
    this.level = SDZLevel.ZERO;
    listeAmis = new ArrayList<Zero>();
}

/**
 * Ajoute un Zero à la liste des amis.
 *
 * @param ami
 *          Le nouvel ami du Zéro.
 *
 * @see Zero#listeAmis
*/
public void ajouterAmi(Zero ami) {
    listeAmis.add(ami);
}

/**
 * Retire un Zero à la liste des amis.
 *
 * @param ancienAmi
 *          Un ancien ami du Zéro.
 *
 * @see Zero#listeAmis
*/
public void retirerAmi(Zero ancienAmi) {
    listeAmis.remove(ancienAmi);
}

/**
 * Retourne l'ID du zéro.
 *
 * @return L'identifiant du membre.
*/
public int getId() {
    return id;
}

/**
 * Retourne le pseudo du zéro.
 *
 * @return Le pseudo du membre, sous forme d'une chaîne de caractères.
*/
public String getPseudo() {
    return pseudo;
}

...

```

```

/**
 * Met à jour le pseudo du membre.
 *
 * @param pseudo
 *          Le nouveau pseudo du membre.
 *
 * @since 3.0
 */
public void setPseudo(String pseudo) {
    this.pseudo = pseudo;
}

/**
 * Retourne la liste des amis du zéro.
 *
 * @return La liste des amis du zéro, sous la forme d'un vecteur.
 *
 * @deprecated Depuis Java 1.4, remplacé par getListeAmis()
 *
 * @see Zero
 * @see Zero#getListeAmis
 */
protected Vector<Zero> getVectorAmis(){
    Vector<Zero> vector = new Vector<Zero>();
    for (Zero z : listeAmis){
        vector.add(z);
    }
    return vector;
}

/**
 * Retourne l'adresse du profil du Zero.
 *
 * @return L'URL du profil du Zero, générée à partir de l'id du Zero.
 *
 * @throws MalformedURLException Si jamais l'url est mal formée.
 *
 * @see Zero#id
 */
public URL getURLProfil() throws MalformedURLException
{
    URL url = new URL("http://www.siteduzero.com/membres-294-"+id+".html");
    return url;
}
}

```

Si l'on veut avoir la JavaDoc dans un autre format, il faut utiliser les **Doclets**.

En effet, il existe un grand nombre de Doclets, qui permettent d'exporter votre JavaDoc en différents formats, tels que **PDF**, **XML**, **DocBook**, **LaTeX**, etc. Pour plus d'informations, jetez un coup d'oeil sur [doclet.com](http://doclet.com)

## Annexe : Les annotations

Les **annotations** sont une nouveauté apportée par la version 5 de Java (*Tiger*) et permettent une standardisation de ce qu'on appelle des **métadonnées**.

Des **métadonnées** sont des informations sémantiques (textuelles) ajoutant un niveau d'information supplémentaire à vos objets, vos interfaces, vos paramètres, vos variables d'instances, vos packages et même à d'autres annotations.

Ces informations ainsi ajoutées pourront être visualisées (dans la JavaDoc par exemple) ou utilisées grâce à l'introspection (pour exécuter du code par exemple).

Les annotations permettent de rajouter des métadonnées dans vos codes sources (on parle aussi d'ajout sémantique). Elles peuvent être utilisées sur plusieurs composantes de vos programmes Java :

- sur des classes ;
- sur des méthodes ;
- sur des interfaces ;
- sur des variables ;
- sur des packages.

Ces fameuses données sémantiques peuvent être lues par des programmes comme des générateurs de JavaDoc mais elles sont aussi lues par le compilateur Java qui peut aussi permettre leur utilisation grâce à l'introspection.

Les annotations servent principalement à **marquer certains éléments** du langage Java afin de leur **attribuer des données particulières** (voir même des actions).

De plus, les annotations peuvent être utilisées à différents moments du cycle de vie d'un programme :

- lors de son écriture : les annotations peuvent être utilisées pour documenter et donner des informations aux différents développeurs ;
- lors de la compilation : cela permet d'interagir avec le compilateur et lui faire exécuter des tâches automatiquement ;
- pendant l'exécution du programme : en utilisant l'introspection, vous pouvez accéder aux annotations et agir sur vos méthodes, vos champs, vos classes...

Il existe plusieurs types d'annotations :

- les annotations **marqueurs** : on parle aussi d'annotations standards. Ce sont des marqueurs placés qui permettent au compilateur de faire ou ne pas faire certaines actions.  
On trouve notamment `@Deprecated`, `@SuppressWarnings`, `@Override`, ...
- les annotations **paramétrées** : ce sont des annotations qui prennent un paramètre pour pouvoir fonctionner. Exemple : `@Retention(RetentionPolicy.RUNTIME)`;
- les annotations **multiparamétrées** : ces dernières prennent plusieurs paramètres.

## Les annotations standards

Les annotations standards sont des annotations marqueurs proposées par le langage Java.

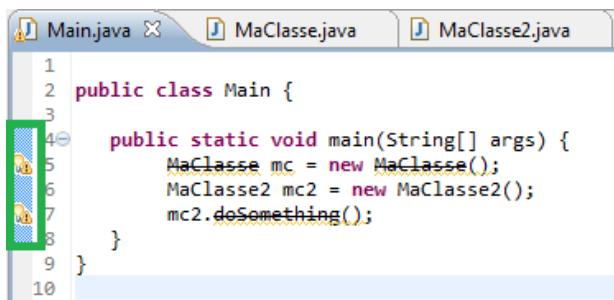
Elles sont au nombre de trois et ont toutes une utilisation bien précise.

- **@Deprecated** : cette annotation indique au compilateur que l'élément marqué ne devrait plus être utilisé. Si un tel élément est tout de même utilisé, Eclipse vous avertira en affichant un warning dans la vue mise à disposition à cet effet.

Voici un exemple illustrant le comportement cette annotation :

```
@Deprecated  
public class MaClasse { }
```

```
public class MaClasse2  
{  
    @Deprecated  
    public void doSomething(){}
}
```



Ce marqueur n'empêchera pas la compilation de votre code ni son fonctionnement mais il vous avertit sur le caractère obsolète d'un élément du langage ou de votre code.

- **@SuppressWarnings** : permet d'indiquer au compilateur que certains éléments du code ne doivent pas générer de warning lors de la compilation

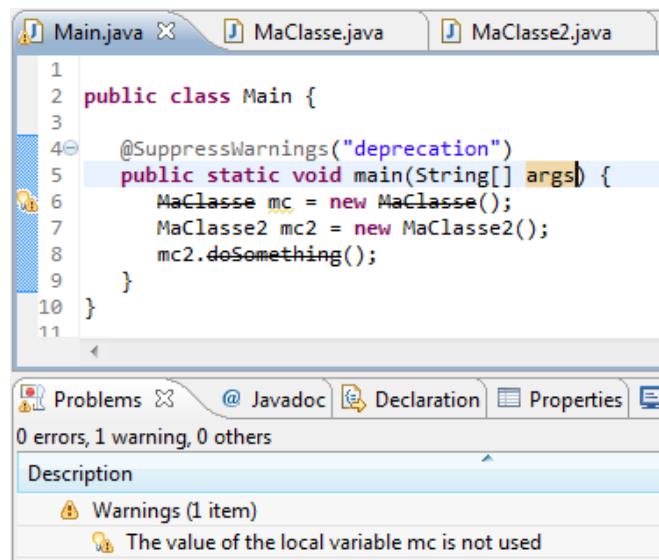
Elle prend en paramètre un tableau de String correspondant aux différents cas pouvant être gérés par cette annotation, mais les valeurs possibles ne sont pas standardisées et dépendent du compilateur utilisé.

Voici donc quelques valeurs tolérées par Eclipse pour cette annotation :

- **all** : aucun warning ne sera remonté.
- **boxing** : contre les warnings relatifs au boxing ou l'unboxing.
- **cast** : pour supprimer les warnings sur des cast.
- **dep-ann** : évite les warning causés par une annotation dépréciée.
- **deprecation** : annule les warnings remontés par l'annotation @Deprecated.
- **fallthrough** : en cas d'oubli d'une instruction break dans un bloc switch.
- **finally** : pour les warnings levés à cause d'un bloc finally douteux.
- **serial** : pour retirer le warning généré lorsque le champ serialVersionUID est manquant pour une classe serializable.
- **super** : supprime les warnings dans le cas d'une redéfinition de méthode sans invocation du mot clé super.
- **unused** : retire le warning sur un morceau de code inutilisé.

Exemples :

```
public class Main
{
    @SuppressWarnings("deprecation")
    public static void main(String[] args) {
        MaClasse mc = new MaClasse();
        MaClasse2 mc2 = new MaClasse2();
        mc2.doSomething();
    }
}
```



```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class Main
{
    // Nous pouvons passer un tableau de valeurs pour gérer plusieurs types de warnings d'un coup :
    // pas de warning sur la dépréciation et l'utilisation de type non générique
    @SuppressWarnings({ "deprecation", "unchecked" })
    public static void main(String[] args)
    {
        // pas de warning sur les éléments non utilisés
        @SuppressWarnings("unused")
        int i = 1, j = 2;
        // pas de warning sur un type générique utilisé sans typage
        @SuppressWarnings("rawtypes")
        List list = new ArrayList();
        list.add("toto");
        List<String> listString = new ArrayList<String>();
        listString.addAll(list);
    }
}
```

```

// pas de warning même si la méthode n'est pas utilisée
@SuppressWarnings("unused")
private static void doIt(int i)
{
    switch(i){
        case 1:
        case 2:
            System.out.println("coucou");
        case 3 :
    }
}

@SuppressWarnings("serial")
public class ExempleSerializable implements Serializable
{
    // plus de Warning en cas d'absence du serialVersionUID
}

```

- **@Override** : informe le compilateur qu'une méthode est redéfinie et donc que celle-ci doit être présente dans la classe mère.

Cette annotation ne doit donc être utilisée que sur des méthodes redéfinies via le principe d'héritage. Si cette annotation est utilisée sur une méthode non redéfinie, votre programme ne compilera pas.

Exemple :

```

public class ExempleOverride
{
    @Override
    public String toString()
    {
        return "ExempleOverride [getClass()=" + getClass() + ", hashCode()="
               + hashCode() + ", toString()=" + super.toString() + "]";
    }
}

```

## Nos propres annotations

Exemple d'annotation maison :

```
public @interface AnnotationZ { }
```

Pour l'utiliser dans vos programmes :

```
public class TestAnnotation
{
    private String nom = "toto";

    @AnnotationZ
    public String faisQuelqueChose(){
        return "Je ne fais rien...";
    }

    @AnnotationZ public String faisQuelqueChoseDAutre(){
        return "Je ne fais rien...";
    }
}
```

Il faut indiquer le nom de l'annotation avant (au dessus de) la définition de la variable / méthode / classe que l'on souhaite annoter. Vous pouvez utiliser plusieurs annotations différentes sur un élément du langage. Depuis Java 8 vous pouvez même utiliser plusieurs fois la même annotation sur un élément du langage.

## Les méta-annotations

Ces annotations servent à marquer d'autres annotations, surtout pour indiquer au compilateur comment et quand il doit interpréter les annotations qui les utilisent.

Les annotations présentes dans le package [java.lang.annotation](#)

- **@Documented** : indique à l'utilitaire JavaDoc que l'annotation doit être présente dans la documentation générée

```
import java.lang.annotation.Documented;

@Documented
public @interface AnnotationZ { }
```

- **@Inherit** : indique que tous les enfants d'une classe marquée avec une annotation, en hériteront.

```
import java.lang.annotation.Documented;
import java.lang.annotation.Inherited;

@Documented
@Inherited
public @interface AnnotationZ { }

@AnnotationZ
@Temoin
public class ClasseMere { }
```

```
public class ClasseFille extends ClasseMere{ }
```

Ici, la classe *ClasseFille* héritera seulement l'annotation 'AnnotationZ'.

- **@Retention** : permet de régler une "durée de vie" pour l'annotation, ce qui indique au compilateur comment il doit la gérer
- **@Target** : permet de cibler sur quels éléments du langage nous pourrons utiliser des annotations

## Annexe : Les tests unitaires

Il existe plusieurs types de tests :

- **tests d'intégration** : le programme créé s'intègre bien dans son environnement d'exécution ?
- **tests d'acceptation** : l'utilisateur final accepte-t-il le logiciel ?
- **tests unitaires** : destinés à tester une unité du logiciel.

Ce sont ces derniers qui nous intéresseront et les unités que nous allons tester seront les méthodes de nos classes.

Voici un exemple simple : soit cette méthode `String concatene(String a, String b) {...}`. Nous voulons tester si elle concatène bien les deux chaînes a et b. Deux méthodes pour tester cette fonction :

- la tester dans notre programme : on appelle cette méthode dans le main avec deux chaînes et on affiche le résultat.
- **créer une classe dédiée à ce test**, c'est précisément le but du test unitaire.

On va créer **une classe de test par classe à tester**. Dans chaque classe de test, il y aura **une méthode par méthode à tester**. Donc en fait, pour **chaque classe du logiciel, on va avoir sa sœur pour le test**.

L'objectif est de trouver un maximum de bug. Pourquoi pas tous ? Parce que ce serait trop long et trop difficile, il faudrait être sûr que dans tous les cas, si un certain nombre de préconditions sont remplies, alors un certain nombre de post-conditions le seront. C'est parce que prouver que son logiciel est exempt de bug est trop difficile que nous allons seulement mettre en place un moyen de trouver quelques bugs.

Pour tester, nous allons nous baser sur **deux assomptions** :

- si ça marche une fois, ça marchera les autres fois;
- si ça marche pour quelques valeurs, ça marchera pour toutes les autres.

Ces deux assomptions réduisent drastiquement le nombre de cas de test à effectuer.

Définition : **un cas de test est un ensemble composé de trois objets**.

- un **état** (ou contexte) **de départ**;
- un **état** (ou contexte) **d'arrivée**;
- un **oracle**, c'est à dire un outil qui va prédire l'état d'arrivée en fonction de l'état de départ et comparer le résultat théorique et le résultat pratique.

Un cas de test peut donc s'appliquer à plusieurs méthodes, par exemple plusieurs classes implémentant la même interface.

Exemple de test :

```
public boolean concateneTest()
{
    MyString classATester = new MyString();
    String a = "salut les ";
    String b = "zeros";
    String resultatAttendu = "salut les zeros";
    String resultatObtenu = classATester.concatene(a, b);
```

```

if ( resultatAttendu.compareTo(resultatObtenu) == 0 )
    return true;
else
    return false;
}

```

Nous pouvons observer plusieurs choses de ce bout de code :

- le test ne dit pas quelle est l'erreur, il dit seulement qu'il y en a une;
- le test ne corrige pas l'erreur;
- ce n'est pas parce que le test passe qu'il n'y a pas d'erreur;
- ce n'est pas parce que vous corriger l'erreur qu'il n'y en a plus.

Nous allons maintenant voir comment les mettre en pratique grâce à **JUnit**, le **framework de test unitaire de Java**. Bien que JUnit soit intégré à la plupart des IDE, il ne fait pas partie de la librairie standard de Java.

En fait, JUnit est le framework de test unitaire qui fait partie d'un plus grand ensemble nommé **XUnit**. XUnit désigne tous les frameworks de test répondant à certains critères pour une multitude de langage. Il y a par exemple CUnit pour le C, CPPUNIT pour le C++, PHPUnit pour PHP, ...

Pour simplifier la maintenance du code et le packaging de notre logiciel, nous allons créer deux packages principaux : main et test. Dans main, nous mettrons toutes nos classes pour le logiciel et dans test, nos classes de test.

Encore une chose à propos des tests : il y a des **tests boîte noire** (*black-box*) et des **tests boîte blanche** (*white-box*). Les tests *boîte noire* se font sans que le testeur ne connaisse le contenu de la méthode qu'il va tester alors que les tests *boîte blanche* donne accès au contenu de la méthode à tester. Les deux ont leurs avantages et inconvénients : lorsque l'on teste en *boîte noire*, on teste réellement ce que devrait faire la méthode. Lorsque l'on teste la méthode en connaissant son fonctionnement. Le risque est alors de tester le fonctionnement et d'oublier le but final de la méthode. En contre-partie, nos tests pourront être plus précis.

Exemple : classe qui permet de calculer le résultat d'opérations mathématiques de base

```

package main;

public interface Calculator
{
    int multiply(int a, int b);
    int divide(int a, int b);
    int add(int a, int b);
    int subtract(int a, int b);

}

```

Certaines techniques de développement préconisent même d'écrire tous les tests avant de commencer le logiciel. Au fur et à mesure du développement, de plus en plus de tests vont réussir et lorsqu'ils réussissent tous, le logiciel est terminé. C'est la méthode de développement dite "test-driven".

## Test de la méthode add(...)

Pour écrire un test correct, nous allons tester quelques valeurs puis nous allons généraliser. Cependant, nous n'allons pas choisir ces valeurs au hasard et c'est là que réside tout l'art d'écrire un test.

Nous avons donc nos arguments a et b. Nous allons les additionner. Et nous allons tester plusieurs cas spéciaux : si a ou b ou les deux est (sont) négatif(s), nul ou positifs.

D'une manière générale, lorsque vous écrivez un test, il faut **tester avec quelques valeurs standards**, qui n'ont pas de signification particulière. Puis il faut **tester avec les cas limites** : nombres négatifs, nuls... Si vous prenez des objets, et s'ils étaient *null*, s'ils étaient une sous-classe du type demandé ? Ou bien si l'objet était mal initialisé ? Tout ceci sont des choses auxquels vous devez penser lorsque vous créez vos tests. Malgré cela, vos **tests doivent restés très simples**, s'ils deviennent compliqués, vous risquez d'introduire des bugs dans les tests.

Voici **un squelette de test** :

- instancier et initialiser la classe à tester T;
- générer les arguments pour la méthode à tester;
- générer le résultat;
- tester la méthode avec les arguments
- vérifier le résultat;
- recommence depuis 2 tant qu'il y a des cas à tester.

Voici mon test :

```
@Test
public final void testAdd()
{
    Calculator calc= new CalculatorImpl();
    int a, b, res;

    a = 5; b = 5; res = a + b;
    if ( calc.add(a, b) != res )
        fail("a et b positif");

    a = 0; b = 5; res = a + b;
    if ( calc.add(a, b) != res )
        fail("a nul");

    a = 5; b = 0; res = a + b;
    if ( calc.add(a, b) != res )
        fail("b nul");

    a = 0; b = 0; res = a + b;
    if ( calc.add(a, b) != res )
        fail("a et b nuls");
}
```

Il y a donc sept cas de tests avec à chaque fois a et b qui varient. On laisse un message pour savoir quel cas échoue lorsqu'il y a un échec et nous avons notre oracle : on calcule le résultat théorique d'une manière aussi sûre que possible puis on le compare grâce à un test d'égalité (ou de différence lorsqu'on veut trouver l'échec).

Pour exécuter le test, il y a deux choix possibles : le test passe au vert => implémentez la méthode suivante et son test; le test reste rouge => trouvez le bug et corrigez le.

Je vais vous montrer maintenant **comment gérer les exceptions**. Pour cela nous allons implémenter la méthode de *division*. Nous allons jeter une exception si b vaut 0.

Voici le cas de test pour les cas où aucune exception ne devrait être jetée :

```
@Test
public final void testDivide()
{
    Calculator calc= new CalculatorImpl();
    int a, b, res;

    a = 5; b = 5; res = a / b;
    if ( calc.divide(a, b) != res )
        fail("a et b positif");

    a = 0; b = 5; res = a / b;
    if ( calc.divide(a, b) != res )
        fail("a nul");

    a = -5; b = 5; res = a / b;
    if ( calc.divide(a, b) != res )
        fail("a negatif");
}
```

```
a = 5; b = -5; res = a / b;
if ( calc.divide(a, b) != res )
    fail("b negatif");

a = -5; b = -5; res = a / b;
if ( calc.divide(a, b) != res )
    fail("a et b negatif");
```

Maintenant la partie difficile : gérer **le cas où une exception devrait être lancée**. Ce que nous voulons c'est dire "**Ce bout de code doit jeter une exception. S'il ne le fait pas c'est que la méthode ne réagit pas comme elle devrait.**" Notre test doit donc échouer si aucune exception n'est levée. Pour l'indiquer à JUnit, nous allons dire que le test attend une exception par le biais de l'annotation **@Test (expected = LaClassDeNotreException)** (*expected* signifie s'attend à ou attend une).

Voici donc le code :

```
@Test (expected = ArithmeticException.class)
public final void testDivideByZero()
{
    Calculator calc = new CalculatorImpl();
    int a, b, res;

    a = 5; b = 0; res = 0;
    if ( calc.divide(a, b) != res )
        fail("b nul");
}
```

```
a = 0; b = 0; res = 0;
if ( calc.divide(a, b) != res )
    fail("a et b nuls");
```

Voici maintenant tout un tas de **méthodes qui vont vous permettre de faire échouer vos tests** à la place de cet affreux **if(...) fail()** :

```
assertTrue(message, condition);
assertFalse(message, condition);
assertEquals(message, expected, actual);           // pour des objets ou des longs
assertNotNull(message, object);
```

Voici ce que ça donne pour l'un de nos tests :

```
@Test
public final void testAdd()
{
    Calculator calc = new CalculatorImpl();
    int a, b, res;

    a = 5; b = 5; res = a + b;
    assertTrue("a et b positif", calc.add(a, b) == res);

    a = 0; b = 5; res = a + b;
    assertTrue("a nul", calc.add(a, b) == res);

    a = 5; b = 0; res = a + b;
    assertTrue("b nul", calc.add(a, b) == res);

    a = 0; b = 0; res = a + b;
    assertTrue("a et b nuls", calc.add(a, b) == res);

    a = -5; b = 5; res = a + b;
    assertTrue("a negatif", calc.add(a, b) == res);

    a = 5; b = -5; res = a + b;
    assertTrue("b negatif", calc.add(a, b) == res);

    a = -5; b = -5; res = a + b;
    assertTrue("a et b negatifs", calc.add(a, b) == res);
}
```

C'est plus court et plus concentré.

### La couverture du code

Pour évaluer votre code avec le plugin **EclEmma**, il faut tout d'abord l'installer. Aller dans Help >> Install New Software >> Add. Donner un nom (EclEmma) et une adresse (<http://update.eclemma.org/>). Puis faites Ok et installer le plugin. Vous devrez accepter la licence puis redémarrer Eclipse.

Enfin, à coté de l'icône debug, un nouvel icône est apparu : celui de EclEmma.



Cliquez dessus et voyez votre **code se colorer** en rouge, jaune et vert. Les parties vertes sont les parties vérifiées par vos tests alors que les rouges ne l'ont pas été. Les lignes jaunes n'ont été que partiellement couvertes (une condition par exemple).

EclEmma génère aussi un rapport vous détaillant quelle fraction de votre code est testée dans chaque package, dans chaque classe et même dans chaque méthode.

Quatre **nouvelles méthodes**, précédées d'une annotation qui a une utilité bien précise :

| Annotation   | Utilité                                               |
|--------------|-------------------------------------------------------|
| @BeforeClass | La méthode annotée sera lancée avant le premier test. |
| @AfterClass  | La méthode annotée sera lancée après le dernier test. |
| @Before      | La méthode annotée sera lancée avant chaque test.     |
| @After       | La méthode annotée sera lancée après chaque test.     |

Les méthodes appelées avant et après tous les test nous servirons à initialiser des variables et ressources communes à tous les tests et à les nettoyer à la fin.

Les méthodes appelées avant et après chaque test nous servirons à initialiser la liste avant et à la remettre à zéro après.

Un test d'une liste enchainée utilisant un fichier de propriétés pour configurer le test :

```
src/config.properties  
taille=6  
nombre=1 2 3 4 5 6
```

```
public class MyListImplTest  
{  
    private static int expectedSize;           // la taille à l'origine  
    private static Properties prop;            // les propriétés  
    private static FileInputStream propFile;   // le fichier de propriétés  
  
    private static List<Integer> testSet;      // les nombres que nous mettrons dans notre class  
    private static MyList<Integer> sut;         //la classe à tester  
  
    @BeforeClass  
    public static void setUpBeforeClass() throws Exception  
    {  
        testSet = new LinkedList<Integer>();  
  
        propFile = new FileInputStream("src/config.properties"); //charge le fichier de pp  
        prop = new Properties();  
        prop.load(propFile);  
        expectedSize = Integer.parseInt(prop.getProperty("taille")); //parse la taille  
  
        String numbers = prop.getProperty("nombre"); //récupère les nombre à mettre dans la liste  
        for ( String i : numbers.split(" ") )          //pour chaque nombre  
            testSet.add(Integer.parseInt(i.trim())); // l'enregistrer en tant que int  
  
        sut = new MyListImpl<Integer>();             // instancier la classe à tester  
    }
```

```

@AfterClass
public static void tearDownAfterClass() throws Exception
{
    propFile.close();                                // on ferme le fichier à la fin du test
}

@Before
public void setUp() throws Exception
{
    for (int i : testSet)
        sut.add(new Integer(i));                  // on ajoute les nombres au début de chaque test
}

@After
public void tearDown() throws Exception
{
    sut.reset();                                    // à la fin de chaque test, on reset notre liste
}

```

Ce que vous venez de voir, c'est comment **automatiser l'instanciation** de vos classes ainsi que leur initialisation, leur remise à zéro. Vous avez aussi vu comment paramétriser vos tests.

Si j'ai utilisé un fichier de configuration ici, ce n'est pas tellement pour paramétriser le test mais plutôt pour vous montrer que l'on peut charger et libérer des ressources externes grâce aux méthodes appelées avant et après tous les tests. C'est d'ailleurs pour cette raison que j'ai tenu à ce que le FileInputStream soit une variable de classe. Ce n'était pas obligatoire.

Maintenant nous voulons tester des **méthodes qui changent le contexte**.

Très bien, prenons la méthode **void add(T e)** de notre classe de liste. Son seul effet est d'ajouter un élément à notre liste, nous allons donc vérifier que l'élément a bien été ajouté.

Il y a deux choses à tester :

- la taille de la liste;
- la présence des éléments.

Voici le test qui nous dit si tous les éléments sont présents et au bon endroit :

```

@Test
public void testAdd()
{
    assertEquals(expectedSize, sut.getSize());
    sut.add(new Integer(8));
    assertEquals(expectedSize+1, sut.getSize());

    for ( int i = 0; i < testSet.size(); i++ )
        assertEquals(testSet.get(i), sut.getAt(i));
}

```

## Les mocks

Un mock est une classe qui va en simuler une autre.

Je m'explique : la classe A a besoin de la classe B pour travailler. Ce n'est pas vrai. La classe A a besoin d'une classe implémentant l'interface **I** (souvenez-vous de l'importance de la programmation par interface) et il se trouve que la classe B implémente l'interface **I**. La classe B fait un travail sérieux qu'il est nécessaire de tester, on ne peut donc pas l'instancier dans le test de la classe A. Cependant, il nous faut bien passer une classe implémentant l'interface **I** pour tester A. C'est pour cela que nous allons **créer la classe BMock** qui implémente l'interface **I** mais qui soit si simple qu'il n'est plus la peine de la tester.

Je vais prendre un exemple aussi très simple mais qui sera réaliste : application en trois couches

- une **couche d'accès** aux données qui gère le pool de connexion et les requêtes;
- une **couche métier** qui prend les données, leurs applique une transformation utile
- une **couche présentation** qui récupère les résultats de la couche métier et les affiche d'une manière lisible par l'homme.

Ceci est une **organisation très célèbre nommée trois tiers**. Mais vous voyez immédiatement qu'il y a une forte dépendance entre une couche et la suivante.

Dans une application réelle il y a des mocks pour chaque couche excepté la couche présentation. Ainsi, la couche présentation peut tourner soit avec le mock de la couche métier soit avec son implémentation et la couche métier peut travailler sur de vraies données ou sur des données contenues dans un mock. C'est d'ailleurs comme ça qu'on cache le retard : on dit qu'une fonctionnalité est implémentée alors qu'elle tourne avec un mock.

On peut aussi avoir des applications 2 tiers (client-serveur) ou n-tiers (en général des applications 2 ou 3 tiers chaînées). Dans une architecture n-tiers, la couche *k* ne peut accéder qu'à la couche *k-1* ! La couche *k-1* ne peut accéder à la couche *k* que par le retour d'une méthode et deux couches séparées par une troisième ne peuvent pas communiquer directement. De plus, seule la couche présentation peut dialoguer avec le monde extérieur.

Ainsi, tout est facilité : imaginez que votre base de données ne vous convienne plus, changez là et changez la couche d'accès aux données et tout marche. Vous ne voulez plus dialoguer avec des humains, changez la couche présentation, formatez les entrées-sorties en suivant le bon protocole et vous dialoguez maintenant avec r2-d2.

Exemple : soit deux classes, une de la couche présentation et une de la couche métier. L'une va afficher une adresse et l'autre va la chercher quelque part. Voici donc les deux interfaces.

### Les interfaces :

```
package main.inter;
import main.impl.Address;
public interface AddressFetcher
{
    Address fetchAddress(String name);
}

package main.inter;
public interface AddressDisplayer
{
    String displayAddress();
    void setAddressFetcher(AddressFetcher af);
}
```

### Les implémentations :

```
package main.impl;

import main.inter.AddressDisplayer;
import main.inter.AddressFetcher;

public class AddressDisplayerImpl implements AddressDisplayer
{
    private AddressFetcher addressFetcher;

    @Override
    public String displayAddress(String name)
    {
        Address a = addressFetcher.fetchAddress(name);
        String address = a.getName() + "\n";
        address += a.getNb() + " " + a.getStreet() + "\n";
        address += a.getZip() + " " + a.getTown();
        return address;
    }

    @Override
    public void setAddressFetcher(AddressFetcher af)
    {
        this.addressFetcher = af;
    }
}
```

### Le chercheur d'adresse :

```
package main.impl;

import main.inter.AddressFetcher;

public class AddressFetcherImpl implements AddressFetcher
{
    @Override
    public Address fetchAddress(String name)
    {
        // ...
        return null;
    }
}
```

La classe `Address` :

```
package main.impl;

public class Address
{
    private String street;
    private String name;
    private int nb;
    private int zip;
    private String town;

    public Address(String street, String name, int nb, int zip, String town)
    {
        super();
        this.street = street;
        this.name = name;
        this.nb = nb;
        this.zip = zip;
        this.town = town;
    }

    public String getStreet()
    {
        return street;
    }

    public void setStreet(String street)
    {
        this.street = street;
    }

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }

    public int getNb()
    {
        return nb;
    }

    public void setNb(int nb)
    {
        this.nb = nb;
    }

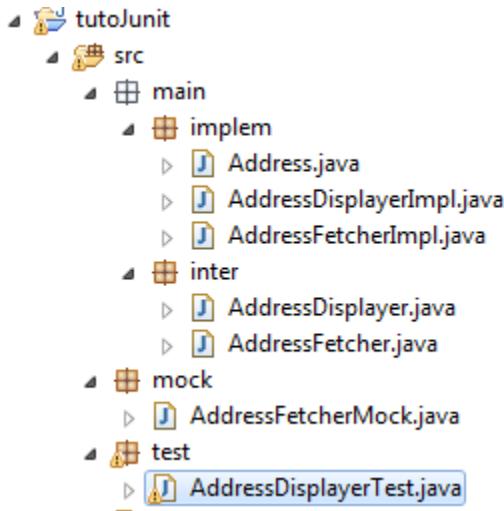
    public int getZip()
    {
        return zip;
    }

    public void setZip(int zip)
    {
        this.zip = zip;
    }

    public String getTown()
    {
        return town;
    }

    public void setTown(String town)
    {
        this.town = town;
    }
}
```

Afin d'avoir un logiciel maintenable, il faut adopter une structure de package correcte :



Ainsi, dans mon **main**, je n'ai que mon logiciel, interface d'un coté, implémentation de l'autre. Dans mon package **test** il n'y a que les tests et dans **mock**, que les mocks. Lorsqu'il faudra livrer le logiciel, je ne donnerai que le package **main**.

Idéalement, dans **inter** et **impl**, je devrais avoir les packages **donnees**, **metier** et **présentation** mais je ne voulais pas surcharger.

Il ne nous reste donc plus que le test et le mock. En réalité, un mock ne va faire que le strict minimum : implémenter l'interface et c'est tout. Le mock va rendre les données en dur.

Voici donc enfin **notre mock** :

```
package mock;

import main.impl.Address;
import main.inter.AddressFetcher;

public class AddressFetcherMock implements AddressFetcher
{
    @Override
    public Address fetchAddress(String name)
    {
        return new Address("Avenue champs-Elysés", "Mathias Dupond", 5, 75005, "Paris");
    }
}
```

Vous pouvez maintenant tester notre classe comme vous le souhaitez. Voici mon test :

```
package test;
import static org.junit.Assert.*;
import main.impl.AddressDisplayerImpl;
import main.inter.AddressDisplayer;
import mock.AddressFetcherMock;
import org.junit.BeforeClass;
import org.junit.Test;
```

```
public class AddressDisplayerTest
{
    private static AddressDisplayer sut;

    @BeforeClass
    public static void setUpBeforeClass() throws Exception
    {
        sut = new AddressDisplayerImpl();
        sut.setAddressFetcher(new AddressFetcherMock());
    }

    @Test
    public void testDisplayAddress()
    {
        String resultatTheorique = "Mathias Dupond\n5 Avenue champs-Elysés\n75005 Paris";
        String ResultatPratique = sut.displayAddress("Dupond");
        assertTrue(ResultatPratique.compareTo(resultatTheorique) == 0);
    }
}
```

Et voilà, tout ça pour ça. On teste seulement si l'affichage est correct, on ne cherche pas à savoir si on affiche la bonne personne (c'est le boulot de *AddressFetcher* de trouver la bonne personne), on cherche à savoir si on affiche correctement la personne X. Pour cela il nous faut une personne et on prend la première venue. Par contre, s'il y avait eu plusieurs modes d'affichage, là oui, il aurait fallu tous les tester.

## Annexe : Le debugging en Java

Tout développeur est confronté un jour à un programme qui plante pour une raison apparemment inexpliquée. Dans ce cas, il y a plusieurs solutions :

- passer en revue le code en espérant trouver l'erreur
- garnir le code d'affichages divers et variés (System.out.print...) afin de tracer son exécution et de déterminer la ligne ou la méthode qui pose problème. Bien que largement utilisée, cette solution est fastidieuse et lorsque le problème est résolu, il faut ensuite supprimer ces affichages ;
- utiliser un débogueur qui est un programme qui exécute le code et permet de suspendre son exécution, la reprendre, afficher le contenu de variables, etc.

### Eclipse

Lancement d'un programme avec des [arguments de ligne de commande](#) :

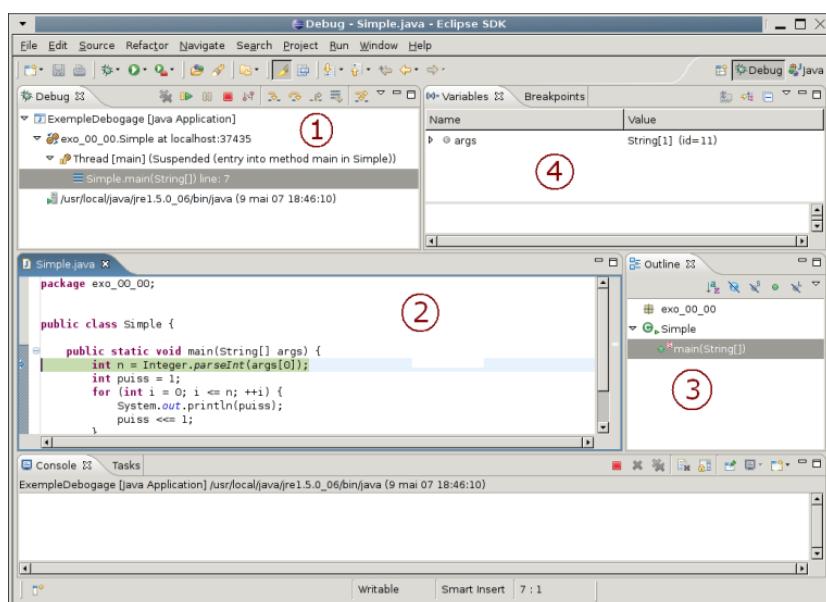
**Run → Run configurations... → Arguments**

Le lancement du [débogage](#) :

- **Run → Debug configurations... → Main** → check "Stop in main"
- Placer au besoin d'autre breakpoints : clic droit sur la marge de gauche → *Toggle Breakpoint*
- **Run → Debug** : Eclipse ouvre ensuite la perspective Debug

Cette perspective est une composition de plusieurs vues :

- la [vue Debug](#) (1) qui indique le(s) thread(s) en cours et les lignes sur lesquelles ils sont suspendus
- la [vue Display](#) (2) qui présente le code en cours de débogage (ici, Simple.java)
- la [vue Outline](#) (3) qui permet d'explorer les packages/classes
- la vue (4) composée de 2 onglets présentant chacun une vue : la [vue Variables](#) qui permet de visualiser les variables, leur valeur et éventuellement leurs données membres; et la [vue Breakpoints](#) qui affiche les différents points d'arrêts qui ont été placés



Plusieurs boutons de la vue Debug permettent de continuer, suspendre ou terminer l'exécution du programme :

- **(Resume)** pour continuer l'exécution du programme depuis l'instruction où il a été suspendu, jusqu'à rencontrer un point d'arrêt, être suspendu manuellement, ou terminer (normalement ou par une exception non capturée) ;
- **(Suspend)** pour suspendre l'exécution du programme là où il en est ;
- **(Terminate)** pour terminer l'exécution du programme (comme si l'on tapait CTRL-C).

D'autres boutons permettent de contrôler finement l'exécution du programme lorsqu'il a été suspendu :

- **(Step Into)** pour exécuter la ligne courante. Si celle-ci contient un appel à une méthode, alors le débogueur se placera sur la première ligne de cette méthode. Notons que le débogueur ne pourra explorer une méthode que si le code source lui est accessible ;
- **(Step Over)** pour exécuter la ligne courante et se placer sur la ligne qui suit (éventuellement en sortant d'une méthode). Ainsi, contrairement au Step Into, s'il y avait un appel de méthode, celle-ci aura été exécutée mais pas explorée ;
- **(Step Return)** pour retourner de l'appel d'une méthode. Ainsi le débogueur va exécuter toutes les instructions restantes afin de terminer l'appel courant de la méthode en cours d'exploration.

Efin, il est possible de demander au débogueur d'exécuter toutes les instructions jusqu'à une ligne donnée (si tant est que cela soit possible) en se plaçant sur la ligne souhaitée et en tapant CTRL-R (qui est un raccourci pour Run to Line du menu Run).

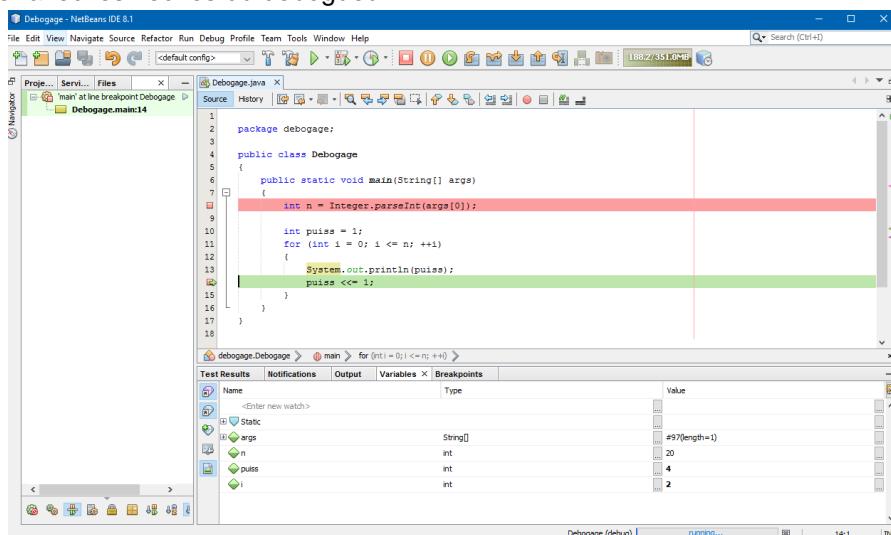
## Netbeans

Lancement d'un programme avec des [arguments de ligne de commande](#) :

Run → Set Project configuration → Customize... → Run → Arguments

Le lancement du [débogage](#) :

- Placer des breakpoints: clic droit sur la marge de gauche → **Breakpoint** → *Toggle Breakpoint*
- **Debug** → **Debug Project**
- Avancer avec les flèches du débogueur



## Annexe : Les mises-à-jour Java au fil des années

| <b>Release</b>           | <b>Année</b> | <b>Mise à jour</b>                                                                                                                                                                                                                                                                                                                                                                                      |
|--------------------------|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| JDK Beta                 | 1995         |                                                                                                                                                                                                                                                                                                                                                                                                         |
| JDK 1.0<br>(Oak)         | 1996         |                                                                                                                                                                                                                                                                                                                                                                                                         |
| JDK 1.1                  | 1997         | <ul style="list-style-type: none"> <li>• mise à jour du model événementiel AWT</li> <li>• classes internes</li> <li>• JDBC</li> <li>• JavaBeans</li> <li>• réflexion et introspection</li> <li>• compilateur JIT (Just In Time)</li> </ul>                                                                                                                                                              |
| J2SE 1.2<br>(Playground) | 1998         | <ul style="list-style-type: none"> <li>• mot clé : <i>strictfp</i></li> <li>• l'API de Swing incluse dans les classes de base</li> <li>• plug-ins java</li> <li>• Java IDL</li> <li>• framework des collections</li> </ul>                                                                                                                                                                              |
| J2SE 1.3<br>(Kestrel)    | 2000         | <ul style="list-style-type: none"> <li>• HotSpot JVM</li> <li>• Java Platform Debugger Architecture(JPDA)</li> <li>• JavaSound</li> <li>• Java Naming and Directory Interface (JNDI) incluse dans les librairies de base</li> </ul>                                                                                                                                                                     |
| J2SE 1.4<br>(Merlin)     | 2002         | <ul style="list-style-type: none"> <li>• mot clé : <i>assert</i></li> <li>• expressions régulières modélisées d'après Perl</li> <li>• chaines d'exceptions</li> <li>• support IPv6</li> <li>• classe NIO : non-blocking IO</li> <li>• parseur XML</li> <li>• extension sécurité et cryptographie (JCE, JSSE, JAAS)</li> <li>• java web start</li> <li>• API de préférences (java.util.prefs)</li> </ul> |
| J2SE 5.0<br>(Tiger)      | 2004         | <ul style="list-style-type: none"> <li>• <b>généricité</b></li> <li>• annotations (meta data)</li> <li>• autoboxing / unboxing : conversions automatiques entre types primitifs et les classes Enveloppes (int &lt;=&gt; Integer)</li> <li>• énumérations (<i>enum</i>)</li> <li>• <b>varargs</b> : le dernier paramètre d'une méthode peut être</li> </ul>                                             |

|                        |      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------------|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                        |      | <p>déclaré en utilisant un type suivi par 3 points<br/>(ex : void drawText(String... lines)</p> <p>=&gt; plusieurs paramètres de ce type peuvent être utilisés et ils seront ensuite placés dans un tableau à passer à la méthode; ou, le code appelant peut passer un tableau de ce type</p> <ul style="list-style-type: none"> <li>• boucle 'for each'</li> <li>• amélioration de la sémantique de l'exécution des programmes multi-thread Java.</li> <li>• classe <b>Scanner</b></li> <li>• services offerts pour la 'concurrence'</li> </ul>                                                                                                                                                                                             |
| Java SE 6<br>(Mustang) | 2006 | <ul style="list-style-type: none"> <li>• API pour l'intégration des langages de script</li> <li>• amélioration de la performance de la plateforme et de Swing</li> <li>• amélioration des services Web</li> <li>• JDBC 4.0</li> <li>• API permettant à un programme Java de sélectionner et invoquer un compilateur Java</li> <li>• supporte les annotations 'pluggable'</li> <li>• améliorations de l'interface graphique : intégration de SwingWorker dans l'API, tables de tri et filtrage</li> <li>• JVM : synchronisation, amélioration de la performance du compilateur, de nouveaux algorithmes et des améliorations à des algorithmes de 'ramasse-miettes' existantes, et les performances des applications de démarrage.</li> </ul> |
| Java SE 7<br>(Dolphin) | 2011 | <ul style="list-style-type: none"> <li>• JVM supporte les langages dynamiques</li> <li>• petites modifications : Strings acceptés en switch, try-with-ressources, underscores acceptés dans la déclaration de variables numériques, multiples catch</li> <li>• nouvelle librairie I/O : java.nio.file</li> <li>• nouvel algorithme de tri dans les collections : Timsort</li> </ul>                                                                                                                                                                                                                                                                                                                                                          |
| Java SE 8              | 2014 | <ul style="list-style-type: none"> <li>• supporte les <b>expressions lambda</b></li> <li>• des <b>méthodes "default"</b></li> <li>• possibilité d'inclure du code JavaScript</li> <li>• annotations possibles sur les types Java</li> <li>• annotations répétitives</li> <li>• API 'DATE and TIME'</li> <li>• lancer des applications <b>JavaFx (GUI)</b></li> </ul>                                                                                                                                                                                                                                                                                                                                                                         |

## Méthodes "default" dans les interfaces

```
interface InterfaceA
{
    void saySomething();

    default void sayHi()
    {
        System.out.println("Hi");
    }
}
```

Le mot clé **default** placé devant le type de retour d'une méthode définie dans une interface permet de ne pas obliger les classes implémentant cette interface à redéfinir cette méthode. Cependant, il faut absolument lui associer un corps.

Conflit avec des multiples interfaces qui comportent la même méthode default : obligation de redéfinir la méthode dans une classe fille implémentant ces deux interfaces.

## Méthodes "statiques" dans les interfaces

```
interface MyData
{
    default void print(String s)
    {
        if ( !isNull(s) )
            System.out.println("MyData Print::" + s);
    }

    static boolean isNull(String s)
    {
        System.out.println("Interface Null Check");
        return s == null ? true : "".equals(s) ? true : false;
    }
}
```

Le mot clé **static** placé devant le type de retour d'une méthode définie dans une interface ne permet pas la redéfinition de cette méthode dans les classes implémentant cette interface. Il faut donc absolument lui associer un corps (qui sera *unique*).

Les méthodes statiques d'une interface peuvent être appelées directement seulement à partir du nom de l'interface.

```
MyDataImpl obj = new MyDataImpl(); // MyDataImpl = classe implementant l'interface MyData
obj.print(""); // Interface Null Check
obj.print("abc"); // Interface Null Check \n MyData Print::abc
obj.isNull("abc"); // ERROR : méthode non définie dans la classe MyDataImpl
MyDataImpl.isNull("abc"); // ERROR : méthode statique non définie dans la classe MyDataImpl
MyData.isNull("abc"); // Interface Null Check
```

## Expressions lambda

Des fonctions anonymes (sans portée, sans nom et sans type de retour) **qui permettent de simplifier l'écriture de classes anonymes**. Elles permettent également de passer des méthodes comme arguments.

Différentes formes de syntaxe :

|                                                                |                                                                         |
|----------------------------------------------------------------|-------------------------------------------------------------------------|
| (int a, int b) -> expression                                   | // 1 seule instruction; sans ";" à la fin; sans accolades               |
| (a, b) -> expression                                           |                                                                         |
| (a) -> expression                                              |                                                                         |
| a -> expression                                                | // parenthèses optionnelles s'il y a un seul paramètre                  |
| () -> expression                                               | // aucun paramètre                                                      |
| (a, b) -> {<br>instr1();<br>instr2();<br>...<br>instrN();<br>} | // instructions déclarées dans un bloc<br>// séparées par des accolades |
| (a, b) -> { return 42; }                                       | // accolades obligatoires si le mot-clé "return" est présent            |
| (a, b) -> 42                                                   | // version simplifiée                                                   |

Les expressions lambda sont possibles grâce aux améliorations apportées dans les versions 7 et 8 Java.

**L'inférence de type** : mécanisme mis en œuvre dans Java 7 qui permet de ne pas préciser à nouveau le type d'objet dans l'instanciation d'un objet de type générique.

Dépuis Java 8, toutes les **variables locales** sont **par défaut constantes** (déclarées *final*). Par contre, dès lors qu'une variable est modifiée quelque part dans le code, celle-ci n'est plus considérée comme constante.

Exemples sans lambda et avec lambda :

```
public static void main(String[] args)
{
    System.out.println("==> RunnableTest ==>");

    // anonymous Runnable
    Runnable r1 = new Runnable()
    {
        @Override
        public void run()
        {
            System.out.println("Hello world one!");
        }
    };
}
```

```

// lambda Runnable
Runnable r2 = () -> System.out.println("Hello world two!");

// run them
r1.run();
r2.run();
}

public static void main(String[] args)
{
    List<Person> personList = Person.createShortList();

    // sort with Inner Class
    Collections.sort(personList, new Comparator<Person>()
    {
        public int compare(Person p1, Person p2)
        { return p1.getSurName().compareTo(p2.getSurName()); }
    });

    // use Lambda instead
    Collections.sort(personList,
                    (Person p1, Person p2) -> p1.getSurName().compareTo(p2.getSurName()));

    // print Descending
    Collections.sort(personList,
                    (Person p1, Person p2) -> p2.getSurName().compareTo(p1.getSurName()));

}

public static void main(String[] args)
{
    JButton testButton = new JButton("Test Button");

    testButton.addActionListener(new ActionListener()
    {
        @Override
        public void actionPerformed(ActionEvent ae)
        {
            System.out.println("Click Detected by Anon Class");
        }
    });

    testButton.addActionListener(
        e -> System.out.println("Click Detected by Lambda Listener"));

    // Swing stuff
    JFrame frame = new JFrame("Listener Test");
    frame.add(testButton, BorderLayout.CENTER);
    frame.pack();
    frame.setVisible(true);
}

```

## Les Stream

Les 3 étapes vers le pattern stream

- **mapper** : permet d'appliquer des opérations sur 1 élément, pour changer sa valeur ou son type.  
Exemple : transformer un objet Person en nombre, en appliquant la méthode getAge()
- **filter** : éliminer les valeurs selon un critère.  
Exemple : éliminer les valeurs inférieurs à 18
- **reduce** : opération terminale permettant de générer le résultat.  
Exemple : calculer le nombre de résultats par incrémentation d'un accumulateur

Java 8 nous propose l'**API Stream** pour simplifier les traitements sur des Collections ou des tableaux en introduisant **un nouvel objet, Stream**.

Un stream se **construit à partir d'une source de données** (une collection, un tableau ou des sources I/O par exemple), et possède un certain nombre de **propriétés** spécifiques :

- Un stream **ne stocke pas de données**, contrairement à une collection. Il se contente de les transférer d'une source vers une suite d'opérations.
- Un stream **ne modifie pas les données de la source** sur laquelle il est construit. S'il doit modifier des données pour les réutiliser, il va construire un nouveau stream à partir du stream initial. Ce point est très important pour garder une cohérence lors de la parallélisation du traitement.
- Le **chargement des données** pour des opérations sur un stream **s'effectue de façon lazy**. Cela permet d'optimiser les performances de nos applications. Par exemple, si l'on recherche dans un stream de chaînes de caractères une chaîne correspondant à un certain pattern, cela nous permettra de ne charger que les éléments nécessaires pour trouver une chaîne qui conviendrait, et le reste des données n'aura alors pas à être chargé.
- Un stream **peut ne pas être borné**, contrairement aux collections. Il faudra cependant veiller à ce que nos opérations se terminent en un temps fini – par exemple avec des méthodes comme **limit(n)** ou **findFirst()**.
- Enfin, **un stream n'est pas réutilisable**. Une fois qu'il a été parcouru, si l'on veut réutiliser les données de la source sur laquelle il avait été construit, nous serons obligés de reconstruire un nouveau stream sur cette même source.

Il existe **deux types d'opérations que l'on peut effectuer sur un stream** :

- les **opérations intermédiaires** : (Stream.map ou Stream.filter par exemple) sont effectuées de façon lazy et renvoient un nouveau stream, ce qui crée une succession de streams que l'on appelle **stream pipelines**. Tant qu'aucune opération terminale n'aura été appelée sur un stream pipelines, les opérations intermédiaires ne seront pas réellement effectuées.
- les **opérations terminales** : quand une opération terminale sera appelée (Stream.reduce ou Stream.collect par exemple), on va alors traverser tous les streams créés par les opérations intermédiaires, appliquer les différentes opérations aux données puis ajouter l'opération terminale. Dès lors, tous les streams seront dit consommés, ils seront détruits et ne pourront plus être utilisés.

### Exemple :

```
List<String> strings = Arrays.asList("girafe", "chameau", "chat", "poisson", "cachalot");

strings.stream()
    .filter(x -> x.contains("cha"))                                // filtrage
    .map(x -> x.substring(0, 1).toUpperCase() + x.substring(1)) // reformatage
    .sorted()   // tri par ordre alphabétique
    .forEach( System.out::println );                                // outputs: Cachalot Chameau Chat
```

Il existe plusieurs types de Stream :

- **Stream<T>** : templatisé pour les types Object
- **IntStream, LongStream, DoubleStream** : spécialisées pour les *types primitifs*

Les classes spécialisées possèdent des méthodes supplémentaires comme par exemple `average()`, `sum()`, et les opérations sont aussi spécialisées dans le même type.

La plus simple façon de créer un stream consiste à appeler la méthode `stream()` ou `parallelStream()` sur une collection,

Les différents types d'**opérations intermédiaires** sur les Stream :

- **skip()** : permet d'ignorer des éléments en début de Stream
- **limit()** : permet de ne traiter qu'un certain nombre d'éléments

```
Stream.of("A", "B", "C", "D", "E", "F", "G")
    .skip(2)
    .limit(3)
    .forEach(System.out::println);                                // C D E
```

- **map()** : changer les données

```
Stream.of("ABC", "DEF", "GHI")
    .map(str -> str.charAt(0))
    .forEach(System.out::print);                                // A D G
```

- **filter()** : exclure des valeurs à l'exécution (comparaison => true / false)

```
IntStream.of(1, 2, 3, 4, 5, 6)
    .filter(i -> i % 2 == 0)
    .forEach(System.out::print);                                // 2 4 6
```

- actions qui agissent sur l'état du Stream :

- parallel() / sequential(),
- distinct()
- sorted()
- unordered()

Les différents types d'**opérations terminales** sur les Stream :

- reductions simples : **sum()**, **avg()**, **count()** : ces réductions produiront le résultat à partir de l'ensemble des éléments
- reductions mutables : **collect()** : ces réductions stockeront l'ensemble des résultats dans un accumulateur, produisant ainsi un nouvel ensemble

```
Map<String, List<String>> groupedByIdentity =  
    Stream.of("A", "A", "B", "C", "C", "C")  
        .collect( groupingBy( identity() ) );      // {A=[A, A], B=[B], C=[C, C, C]}  
  
Map<Integer, List<String>> groupedByLength =  
    Stream.of("AA", "BB", "C", "DDD", "EEE", "FFF")  
        .collect( groupingBy( String::length ) ); // {1=[C], 2=[AA, BB], 3=[DDD, EEE, FFF]}  
  
String csv = Stream.of("A", "B", "C", "D")  
        .collect( joining( ";" ) );      // A;B;C;D
```

## JavaFX

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.PasswordField;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class Login extends Application
{
    @Override
    public void start(Stage primaryStage)
    {
        primaryStage.setTitle("Hello World!");

        GridPane grid = new GridPane();
        grid.setAlignment(Pos.CENTER);
        grid.setHgap(10);
        grid.setVgap(10);
        grid.setPadding(new Insets(25, 25, 25, 25));

        Text sceneTitle = new Text("Welcome");
        sceneTitle.setFont(Font.font("Tahoma", FontWeight.NORMAL, 20));
        grid.add(sceneTitle, 0, 0, 2, 1);

        Label userName = new Label("User name:");
        grid.add(userName, 0, 1);

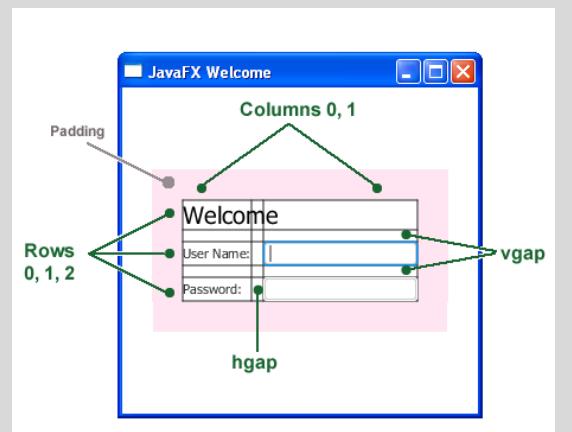
        TextField userTField = new TextField();
        grid.add(userTField, 1, 1);

        Label pw = new Label("Password:");
        grid.add(pw, 0, 2);

        PasswordField userPW = new PasswordField();
        grid.add(userPW, 1, 2);

        Scene scene = new Scene(grid, 300, 275);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args)
    { launch(args); }
}
```



## JavaFX - with CSS style

```
Login.java
public void start(Stage primaryStage)
{
    ...
    Text sceneTitle = new Text("Welcome");
    sceneTitle.setId("welcome-text");

    final Text actionTarget = new Text();
    actionTarget.setId("actionTarget");

    Scene scene = new Scene(grid, 300, 275);
    scene.getStylesheets().add(Login.class.getRessources("Login.css").toExternalForm());

    primaryStage.setScene(scene);
    primaryStage.show();
}
```



```
Login.css
.root {
    -fx-background-image: url("background.jpg");
}

.label {
    -fx-font-size: 12px;
    -fx-font-weight: bold;
    -fx-text-fill: #333333;
    -fx-effect: dropshadow(gaussian, rgba(255, 255, 255, 0.5), 0, 0, 0, 1);
}

#welcome-text {
    -fx-font-size: 32px;
    -fx-font-family: "Arial Black";
    -fx-fill: #818181;
    -fx-effect: innershadow(three-pass-box, rgba(0, 0, 0, 0.7), 6, 0.0, 0, 2);
}

#actionTarget {
    -fx-fill: FIREBRICK;
    -fx-font-weight: bold;
    -fx-effect: dropshadow(gaussian, rgba(255, 255, 255, 0.5), 0, 0, 0, 1);
}

.button {
    -fx-text-fill: white;
    -fx-font-family: "Arial Narrow";
    -fx-font-weight: bold;
    -fx-background-color: linear-gradient(#61a2b1, #2A5058);
    -fx-effect: dropshadow( three-pass-box , rgba(0,0,0,0.6) , 5, 0.0 , 0 , 1 );
}

.button:hover {
    -fx-background-color: linear-gradient(#2A5058, #61a2b1);
}
```

## JavaFX - with fxml file

```
FXMLExample.java
public void start(Stage primaryStage) throws Exception
{
    Parent root = FXMLLoader.load(getClass().getResource("fxml_example.fxml"));
    Scene scene = new Scene(root, 300, 275);

    stage.setTitle("FXML Welcome");
    stage.setScene(scene);
    stage.show();
}
```

```
fxml_example.fxml
<?xml version="1.0" encoding="UTF-8"?>

<?import java.net.*?>
<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.text.*?>

<GridPane xmlns:fx="http://javafx.com/fxml"
           fx:controller="fxmlexample.FXMLExampleController"
           alignment="center" hgap="10" vgap="10">
<padding><Insets top="25" right="25" bottom="10" left="25"/></padding>

    <Text text="Welcome" id="welcome-text"
          GridPane.columnIndex="0" GridPane.rowIndex="0"
          GridPane.columnSpan="2"/>

    <Label text="User Name:" GridPane.columnIndex="0" GridPane.rowIndex="1"/>

    <TextField GridPane.columnIndex="1" GridPane.rowIndex="1"/>

    <Label text="Password:" GridPane.columnIndex="0" GridPane.rowIndex="2"/>

    <PasswordField fx:id="passwordField"
                   GridPane.columnIndex="1" GridPane.rowIndex="2"/>

    <HBox spacing="10" alignment="bottom_right" gp.getColumnIndex="1" gp.getRowIndex="4">
        <Button text="Sign In" onAction="#handleSubmitButtonAction"/>
    </HBox>

    <Text fx:id="actionTarget" GridPane.columnIndex="1" GridPane.rowIndex="6"/>

    <stylesheets> <URL value="@Login.css"/> </stylesheets>
</GridPane>
```



```

FXMLExampleController
public class FXMLExampleController
{
    @FXML private Text actionTarget;

    @FXML protected void handleSubmitButtonAction(ActionEvent event)
    {
        actionTarget.setText("Sign in button pressed");
    }
}

```

## JavaFX - contrôleur

- [Label](#)

```

Label l1 = new Label();
Label l2 = new Label("Search");

Image img = new Image(getClass().getResourceAsStream("label.png"));
Label l3 = new Label("Search", new ImageView(img));

l1.setText("Label label label labeeeel");
l1.setFill(Color.web("#0076a3"));
l1.setFont(new Font("Arial", 30));
l1.setWrapText(true);
l1.setGraphic(new ImageView(img));

l2.setRotate(270);
l2.setTranslateY(50);

l2.setOnMouseEntered( e -> { l2.setScaleX(1.5); l2.setScaleY(1.5); });
l2.setOnMouseExited( e -> { l2.setScaleX(1); l2.setScaleY(1); });

l2.setOnMouseEntered(new EventHandler<MouseEvent>() {
    @Override public void handle(MouseEvent e) {
        l2.setScaleX(1.5);
        l2.setScaleY(1.5);
    }
});

l2.setOnMouseExited(new EventHandler<MouseEvent>() {
    @Override public void handle(MouseEvent e) {
        l2.setScaleX(1);
        l2.setScaleY(1);
    }
});

```

- **Button**

```
Button b1 = new Button();
Button b2 = new Button("Accept");
Button b3 = new Button("Accept", new ImageView(img));

b1.setText("Reject");
b1.setGraphic(new ImageView(img));
b1.setStyle("-fx-font: 22 arial; -fx-base: #b6e7c9;");

b1.setOnAction(e -> l1.setText("Rejected!"));
b2.setOnAction(e -> l1.setText("Accepted!"));

DropShadow shadow = new DropShadow(); //10, Color.BLUE;

b1.setOnMouseEntered(e -> b1.setEffect(shadow));
b1.setOnMouseExited(e -> b1.setEffect(null));
```

- **RadioButton + ToggleGroup**

```
ImageView rbImage = new ImageView();
final ToggleGroup rbGroup = new ToggleGroup();

RadioButton rb1 = new RadioButton("Home");
RadioButton rb2 = new RadioButton("Calendar");
RadioButton rb3 = new RadioButton("Contacts");

rb1.setUserData("Home");
rb2.setUserData("Calendar");
rb3.setUserData("Contacts");
rb1.setToggleGroup(rbGroup);
rb2.setToggleGroup(rbGroup);
rb3.setToggleGroup(rbGroup);

rbGroup.selectedToggleProperty().addListener( new ChangeListener<Toggle>()
{
    @Override
    public void changed(ObservableValue<? extends Toggle> observable, Toggle ov, Toggle nv)
    {
        if ( rbGroup.getSelectedToggle() != null )
        {
            String fileName = rb.getSelectedToggle().getUserData().toString() + ".png";
            final Image image = new Image(getClass().getResourceAsStream(fileName),
   100, 100, false, false);
            rbImage.setImage(image);
        }
    }
});

rb1.setSelected(true);
rb1.requestFocus();
```

- **CheckBox**

```
String[] names = new String[]{"Home", "Calendar", "Contacts"};
Image[] images = new Image[names.length];
ImageView[] icons = new ImageView[names.length];
CheckBox[] cbs = new CheckBox[names.length];

for ( int i = 0; i < names.length; i++ )
{
    Image image = images[i] = new Image(getClass().getResourceAsStream(
        names[i] + ".png"));
    ImageView icon = icons[i] = new ImageView();
    CheckBox cb = cbs[i] = new CheckBox(names[i]);

    cb.selectedProperty().addListener(new ChangeListener<Boolean>()
    {
        public void changed(ObservableValue<? extends Boolean> ov,
            Boolean old_val, Boolean new_val)
        {
            icon.setImage(new_val ? image : null);
            icon.setFitWidth(50);
            icon.setFitHeight(50);
        }
    });
}

cb.selectedProperty().addListener( e ->
{ icon.setImage(cb.isSelected() ? image : null) ;
  icon.setFitWidth(50);
  icon.setFitHeight(50);
});
}
```

- **ChoiceBox**

```
ChoiceBox cb = new ChoiceBox(FXCollections.observableArrayList(
    "First", "Second", "Third"));
cb.setItems(FXCollections.observableArrayList(
    "New Document", "Open ",
    new Separator(), "Save", "Save as"));
```

- **TextField**

```
TextField tf1 = new TextField();
tf1.setPromptText("Enter your first name");
String text = tf1.getText();
tf1.clear();
```

- **ScrollBar**

```
ScrollBar sc = new ScrollBar();
sc.setMin(0);
sc.setMax(100);
sc.setValue(50);
sc.setOrientation(Orientation.VERTICAL);
sc.setPrefHeight(180);
sc.valueProperty().addListener(new ChangeListener<Number>()
{
    public void changed(ObservableValue<? extends Number> ov, Number oldV, Number nv)
    {
        vb.setLayoutY(-nv.doubleValue());
    }
});
```

- **ListView**

```
Label llv = new Label();

ListView<String> list = new ListView<String>();
list.setItems(FXCollections.observableArrayList
    ("Single", "Double", "Suite", "Family App"));

list.setPrefWidth(100);
list.setPrefHeight(70);
list.setOrientation(Orientation.VERTICAL);

list.getSelectionModel().selectedItemProperty().addListener(
    new ChangeListener<String>()
    {
        public void changed(ObservableValue<? extends String> ov, String oldV, String nv)
        {
            llv.setText(nv);
            llv.setTextFill(Color.web(nv));
        }
    });
});
```

- **TableView**

```
ObservableList<Person> data = FXCollections.observableArrayList(
    new Person("Jacob", "Smith", "jacob.smith@example.com"),
    new Person("Isabella", "Johnson", "isabella.johnson@example.com"),
    new Person("Emma", "Jones", "emma.jones@example.com"),
    new Person("Michael", "Brown", "michael.brown@example.com")
);
final Label label = new Label("Address Book");
label.setFont(new Font("Arial", 20));
TableView table = new TableView();
```

```

table.setEditable(true);

TableColumn firstNameCol = new TableColumn("First Name");
firstNameCol.setMinWidth(100);
firstNameCol.setCellValueFactory(new PropertyValueFactory<Person, String>("firstName"));

TableColumn lastNameCol = new TableColumn("Last Name");
lastNameCol.setMinWidth(100);
lastNameCol.setCellValueFactory(new PropertyValueFactory<Person, String>("lastName"));

TableColumn emailCol = new TableColumn("Email");
emailCol.setMinWidth(200);
emailCol.setCellValueFactory(new PropertyValueFactory<Person, String>("email"));

table.getColumns().addAll(firstNameCol, lastNameCol, emailCol);
table.setItems(data);

```

- **Slider**

```

Slider slider = new Slider();
slider.setMin(0);
slider.setMax(100);
slider.setValue(40);
slider.setShowTickLabels(true);
slider.setShowTickMarks(true);
slider.setMajorTickUnit(50);
slider.setMinorTickCount(5);
slider.setBlockIncrement(10);

```

- **ProgressBar**

```

ProgressBar pb = new ProgressBar();
ProgressIndicator pi = new ProgressIndicator();
for ( int i = 0; i < values.length; i++ )
{
    pb.setProgress(values[i]);
    pi.setProgress(values[i]);
}

```

- **Hyperlink**

```

Hyperlink link = new Hyperlink();
link.setText("http://example.com");
link.setOnAction( e -> System.out.println("This link is clicked"));
link.setVisited(false);

```

- **Tooltip**

```
PasswordField pf = new PasswordField();
Tooltip tooltip = new Tooltip();
tooltip.setText("\nYour password must be\n at least 8 characters \n");
tooltip.setGraphic(new ImageView(image));
pf.setTooltip(tooltip);
```

- **TitledPane & Accordion**

```
TitledPane tp = new TitledPane();
tp.setText("My Titled Pane");
tp.setContent(new Button("Button"));
tp.setCollapsible(false);
tp.setAnimated(false);

Accordion accordion = new Accordion ();
for (int i = 0; i < imageNames.length; i++)
{
    images[i] = new Image(getClass().getResourceAsStream(imageNames[i] + ".jpg"));
    pics[i] = new ImageView(images[i]);
    tps[i] = new TitledPane(imageNames[i],pics[i]);
}

accordion.getPanes().addAll(tps);
accordion.expandedPaneProperty().addListener(new ChangeListener<TitledPane>()
{
    public void changed(ObservableValue<? extends TitledPane> tp, TitledPane ov, TitledPane nv)
    {
        if ( nv != null)
            label.setText(accordion.getExpandedPane().getText() + ".jpg");
    }
});
```

- **Menu**

```
MenuBar menuBar = new MenuBar();

// --- Menu File
Menu menuFile = new Menu("File");
MenuItem add = new MenuItem("Shuffle", new ImageView(img));
add.setOnAction( e -> { shuffle(); vbox.setVisible(true); });
MenuItem clear = new MenuItem("Clear");
clear.setAccelerator(KeyCombination.keyCombination("Ctrl+X"));
clear.setOnAction( e -> vbox.setVisible(false));
menuFile.getItems().addAll(add);

// --- Menu Edit
Menu menuEdit = new Menu("Edit");
// --- Menu View
```

```

Menu menuView = new Menu("View");
CheckMenuItem titleView = createMenuItem ("Title", name);
CheckMenuItem binNameView = createMenuItem ("Binomial name", binName);
CheckMenuItem picView = createMenuItem ("Picture", pic);
CheckMenuItem descriptionView = createMenuItem ("Description", description);
menuView.getItems().addAll(titleView, binNameView, picView, descriptionView);

menuBar.getMenus().addAll(menuFile, menuEdit, menuView);
scene.getChildren().addAll(menuBar, vbox);

```

- **ColorPicker**

```

ColorPicker colorPicker1 = new ColorPicker();
ColorPicker colorPicker2 = new ColorPicker(Color.BLUE);
ColorPicker colorPicker3 = new ColorPicker(Color.web("#ffccce6"));
colorPicker1.setValue(Color.CORAL);
colorPicker1.setOnAction( e -> text.setFill(colorPicker1.getValue()));
colorPicker1.getStyleClass().add("split-button");      // sets the split-menu-button
colorPicker1.getStyleClass().add("button");           //sets the button

```

- **Pagination & AnchorPane**

```

Pagination pagination1 = new Pagination();
Pagination pagination2 = new Pagination(5);
Pagination pagination3 = new Pagination(5, 2);

pagination1.setStyle("-fx-border-color:red;");
pagination.setPageFactory(new Callback<Integer, Node>()
{
    @Override
    public Node call(Integer pageIndex)
    {
        return createPage(pageIndex);
    }
});
AnchorPane anchor = new AnchorPane();
AnchorPane.setTopAnchor(pagination1, 10.0);
AnchorPane.setRightAnchor(pagination1, 10.0);
AnchorPane.setBottomAnchor(pagination1, 10.0);
AnchorPane.setLeftAnchor(pagination1, 10.0);
anchor.getChildren().addAll(pagination);
Scene scene = new Scene(anchor);
primaryStage.setScene(scene);

```

- **FileChooser**

```

Desktop desktop = Desktop.getDesktop();

FileChooser fileChooser = new FileChooser();
fileChooser.setTitle("View Pictures");
fileChooser.setInitialDirectory(new File(System.getProperty("user.home")))

```

```

fileChooser.getExtensionFilters().addAll(
    new FileChooser.ExtensionFilter("All Images", "*.*"),
    new FileChooser.ExtensionFilter("JPG", "*.jpg"),
    new FileChooser.ExtensionFilter("PNG", "*.png")
);

File file = fileChooser.showOpenDialog(primaryStage);
if (file != null)
{
    try
    { desktop.open(file); }
    catch (IOException ex)
    { Logger.getLogger(FileChooserSample.class.getName()).log(
        Level.SEVERE, null, ex) }
}

List<File> list = fileChooser.showOpenMultipleDialog(stage);

File file = fileChooser.showSaveDialog(stage);
if (file != null)
{
    try
    {
        ImageIO.write(SwingFXUtils.fromFXImage(pic.getImage(), null), "png", file);
    }
    catch (IOException ex) { System.out.println(ex.getMessage()); }
}

```

- [Canvas](#)

```

public class Dessin extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage)
    {
        Canvas canvas = new Canvas(300, 250);
        GraphicsContext gc = canvas.getGraphicsContext2D();
        drawShapes(gc);

        Group root = new Group();
        root.getChildren().add(canvas);

        primaryStage.setTitle("Drawing Operations Test");
        primaryStage.setScene(new Scene(root));
        primaryStage.show();
    }
}

```

```

private void drawShapes(GraphicsContext gc)
{
    gc.setFill(Color.GREEN);
    gc.setStroke(Color.BLUE);
    gc.setLineWidth(5);

    gc.strokeLine(40, 10, 10, 40);

    gc.fillOval(10, 60, 30, 30);
    gc.strokeOval(60, 60, 30, 30);

    gc.fillRoundRect(110, 60, 30, 30, 10, 10);
    gc.strokeRoundRect(160, 60, 30, 30, 10, 10);

    gc.fillArc(10, 110, 30, 30, 45, 240, ArcType.OPEN);
    gc.fillArc(60, 110, 30, 30, 45, 240, ArcType.CHORD);
    gc.fillArc(110, 110, 30, 30, 45, 240, ArcType.ROUND);

    gc.strokeArc(10, 160, 30, 30, 45, 240, ArcType.OPEN);
    gc.strokeArc(60, 160, 30, 30, 45, 240, ArcType.CHORD);
    gc.strokeArc(110, 160, 30, 30, 45, 240, ArcType.ROUND);

    gc.fillPolygon(new double[]{10, 40, 10, 40},
                  new double[]{210, 210, 240, 240}, 4);

    gc.strokePolygon(new double[]{60, 90, 60, 90},
                    new double[]{210, 210, 240, 240}, 4);

    gc.strokePolyline(new double[]{110, 140, 110, 140},
                      new double[]{210, 210, 240, 240}, 4);
}
}

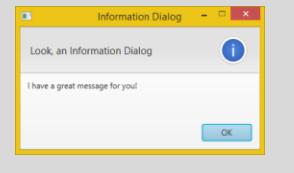
```

- Alert

```

Alert alert = new Alert(AlertType.INFORMATION);
alert.setTitle("Information Dialog");
alert.setHeaderText("Look, an Information Dialog");
alert.setContentText("I have a great message for you!");
alert.showAndWait();

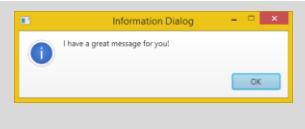
```



```

Alert alert = new Alert(AlertType.INFORMATION);
alert.setTitle("Information Dialog");
alert.setHeaderText(null);
alert.setContentText("I have a great message for you!");
alert.showAndWait();

```



```

Alert alert = new Alert(AlertType.WARNING);
alert.setTitle("Warning Dialog");
alert.setHeaderText("Look, a Warning Dialog");
alert.setContentText("Careful with the next step!");
alert.showAndWait();

```



```

Alert alert = new Alert(AlertType.ERROR);
alert.setTitle("Error Dialog");
alert.setHeaderText("Look, an Error Dialog");
alert.setContentText("Ooops, there was an error!");
alert.showAndWait();

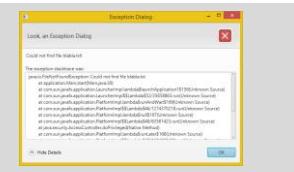
```



```

Alert alert = new Alert(AlertType.ERROR);
alert.setTitle("Exception Dialog");
alert.setHeaderText("Look, an Exception Dialog");
alert.setContentText("Could not find file blabla.txt!");

```



```
Exception ex = new FileNotFoundException("Could not find file blabla.txt");
```

```

StringWriter sw = new StringWriter();
PrintWriter pw = new PrintWriter(sw);
ex.printStackTrace(pw);
String exceptionText = sw.toString();

```

```

Label label = new Label("The exception stacktrace was:");
TextArea textArea = new TextArea(exceptionText);
textArea.setEditable(false);
textArea.setWrapText(true);
textArea.setMaxWidth(Double.MAX_VALUE);
textArea.setHeight(Double.MAX_VALUE);

```

```

GridPane.setVgrow(textArea, Priority.ALWAYS);
GridPane.setHgrow(textArea, Priority.ALWAYS);

```

```

GridPane expContent = new GridPane();
expContent.setMaxWidth(Double.MAX_VALUE);
expContent.add(label, 0, 0);
expContent.add(textArea, 0, 1);

```

```

// Set expandable Exception into the dialog pane.
alert.getDialogPane().setExpandableContent(expContent);
alert.showAndWait();

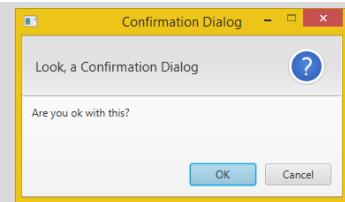
```

```

Alert alert = new Alert(AlertType.CONFIRMATION);
alert.setTitle("Confirmation Dialog");
alert.setHeaderText("Look, a Confirmation Dialog");
alert.setContentText("Are you ok with this?");

Optional<ButtonType> result = alert.showAndWait();
if (result.get() == ButtonType.OK)
{
    // ... user chose OK
}
else
{
    // ... user chose CANCEL or closed the dialog
}

```



```

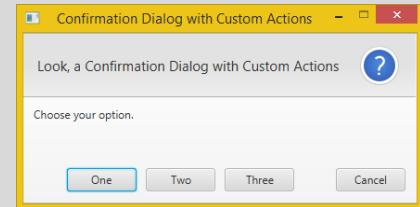
Alert alert = new Alert(AlertType.CONFIRMATION);
alert.setTitle("Confirmation Dialog with Custom Actions");
alert.setHeaderText("Look, a Confirmation Dialog with Custom Actions");
alert.setContentText("Choose your option.");

ButtonType buttonTOne = new ButtonType("One");
ButtonType buttonTTwo = new ButtonType("Two");
ButtonType buttonTThree = new ButtonType("Three");
ButtonType buttonTCancel = new ButtonType("Cancel", ButtonData.CANCEL_CLOSE);

alert.getButtonTypes().addAll(buttonTOne, buttonTTwo, buttonTThree, buttonTCancel);

Optional<ButtonType> result = alert.showAndWait();
if (result.get() == buttonTypeOne)
{
    // ... user chose "One"
}
else if (result.get() == buttonTypeTwo)
{
    // ... user chose "Two"
}
else if (result.get() == buttonTypeThree)
{
    // ... user chose "Three"
}
else
{
    // ... user chose CANCEL or closed the dialog
}

```



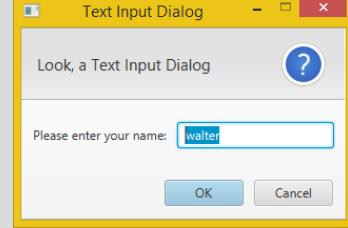
- **TextInputDialog**

```
TextInputDialog dialog = new TextInputDialog("walter");
dialog.setTitle("Text Input Dialog");
dialog.setHeaderText("Look, a Text Input Dialog");
dialog.setContentText("Please enter your name:");

Optional<String> result = dialog.showAndWait();

// traditional way to get the response value.
if ( result.isPresent() )
    System.out.println("Your name: " + result.get());

// the Java 8 way to get the response value (with lambda expression).
result.ifPresent(name -> System.out.println("Your name: " + name));
```



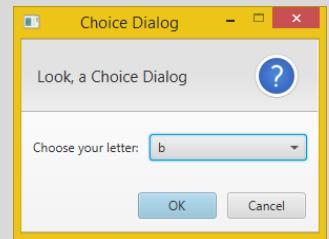
- **ChoiceDialog**

```
ChoiceDialog<String> dialog = new ChoiceDialog("a", Arrays.asList("a", "b", "c"));
dialog.setTitle("Choice Dialog");
dialog.setHeaderText("Look, a Choice Dialog");
dialog.setContentText("Choose your letter:");

Optional<String> result = dialog.showAndWait();

// traditional way to get the response value.
if ( result.isPresent() )
    System.out.println("Your choice : " + result.get());

// the Java 8 way to get the response value (with lambda expression).
result.ifPresent(letter -> System.out.println("Your choice: " + letter));
result.ifPresent(name -> System.out.println("Your name: " + name));
```



- **Dialog<Pair<T, S>>**

```
ButtonType loginButtonType = new ButtonType("Login", ButtonData.OK_DONE);

Dialog<Pair<String, String>> dialog = new Dialog<>();
dialog.setTitle("Login Dialog");
dialog.setHeaderText("Look, a Custom Login Dialog");
dialog.getDialogPane().getButtonTypes().addAll(loginButtonType, ButtonType.CANCEL);

TextField username = new TextField();
username.setPromptText("Username");

PasswordField password = new PasswordField();
password.setPromptText("Password");
```

```

GridPane grid = new GridPane();
grid.setHgap(10);
grid.setVgap(10);
grid.setPadding(new Insets(20, 150, 10, 10));

grid.add(new Label("Username:"), 0, 0);
grid.add(username, 1, 0);
grid.add(new Label("Password:"), 0, 1);
grid.add(password, 1, 1);

// Enable/Disable login button depending on whether a username was entered.
Node loginButton = dialog.getDialogPane().lookupButton(loginButtonType);
loginButton.setDisable(true);

// Do some validation (using the Java 8 lambda syntax).
username.textProperty().addListener((observable, oldValue, newValue) -> {
    loginButton.setDisable(newValue.trim().isEmpty());
});

dialog.getDialogPane().setContent(grid);

// Request focus on the username field by default.
Platform.runLater(() -> username.requestFocus());

// Convert the result to a username-password-pair when the login button is clicked.
dialog.setResultConverter(dialogButton -> {
    if ( dialogButton == loginButtonType )
        return new Pair<>(username.getText(), password.getText());
    return null;
});

Optional<Pair<String, String>> result = dialog.showAndWait();

result.ifPresent(usernamePassword -> {
    System.out.println("Username=" + usernamePassword.getKey() +
                       ", Password=" + usernamePassword.getValue());
});

```

