

# **Licence Professionnelle**

## **Ingénierie des Systèmes Informatiques et Logiciels (ISIL)**

# **PROGRAMMATION ORIENTÉE OBJET (JAVA)**

# **Les interfaces Graphiques**

***Pr. Saïd BENKIRANE***

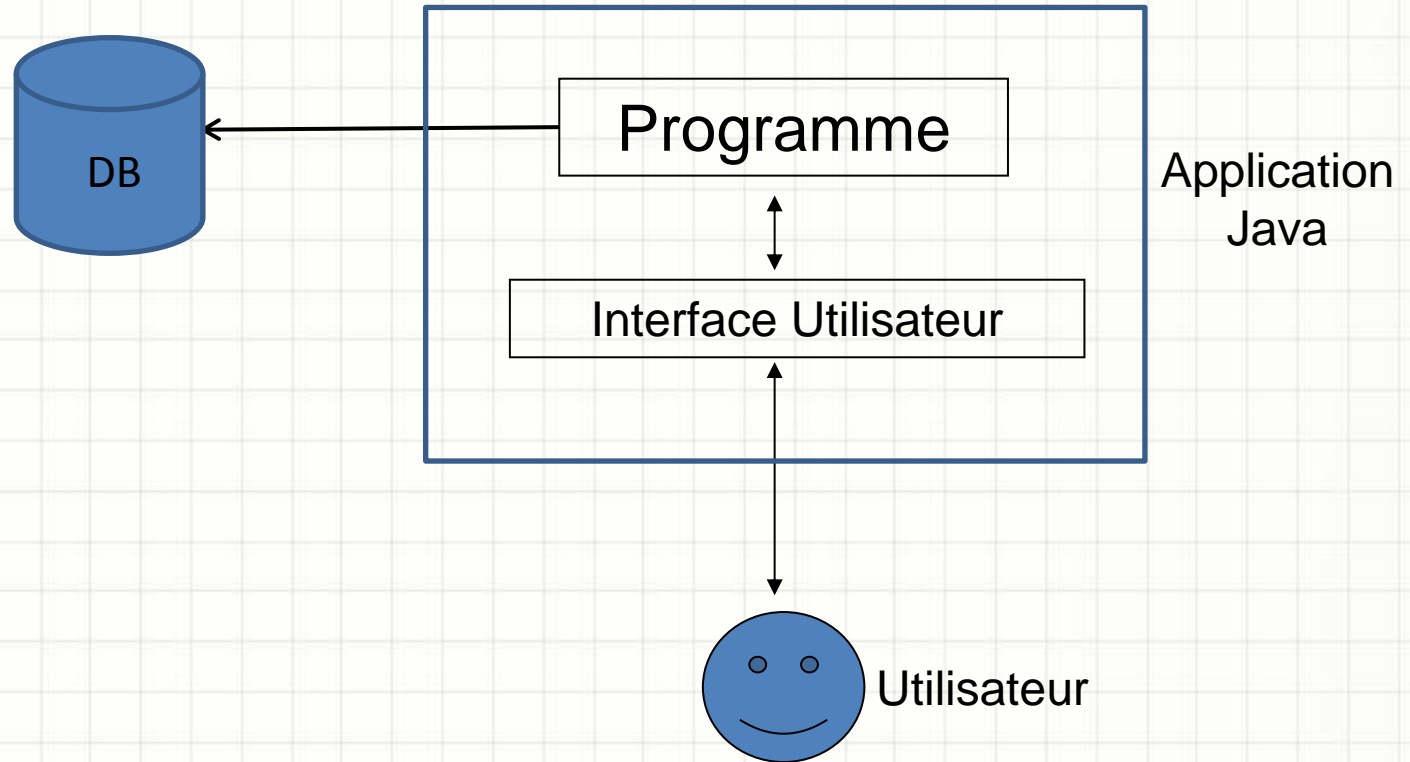
***2016/2017***

# Partie 5: Les interfaces Graphiques

- ☐ Les Composants graphiques
- ☐ Les Gestionnaires de mise en forme
- ☐ Les Contrôleurs d'évènements



# **Awt & swing**



Rôles d'une interface utilisateur:

- Montrer le résultat de l'exécution
- Permettre à l'utilisateur d'interagir
- (1) Montrer – (2) Réagir

# AWT:

## Composants graphiques lourds (1)

- Un composant graphique lourd (*heavyweight GUI component*) s'appuie sur le gestionnaire de fenêtres local, celui de la machine sur laquelle le programme s'exécute.
  - awt ne comporte que des composants lourds.
  - Ce choix technique a été initialement fait pour assurer la portabilité.

# Composants graphiques lourds (2)

- Exemple :
  - Un bouton de type `java.awt.Button` intégré dans une application Java sur la plate-forme Unix est représenté grâce à un vrai bouton Motif (appelé son pair - *peer* en anglais).
  - Java communique avec ce bouton Motif en utilisant la Java Native Interface. Cette communication induit un coût.
  - C'est pourquoi ce bouton est appelé composant lourd.



# Swing:

## Composants légers

- Un composant graphique léger (en anglais, *lightweight GUI component*) est un composant graphique indépendant du gestionnaire de fenêtre local.
  - Un composant léger ressemble à un composant du gestionnaire de fenêtre local mais n'en est pas un : un composant léger émule les composants de gestionnaire de fenêtre local.
  - Un bouton léger est un rectangle dessiné sur une zone de dessin qui contient une étiquette et réagit aux événements souris.
  - Tous les composants de Swing, exceptés **JApplet**, **JDialog**, **JFrame** et **JWindow** sont des composants légers.

# Atouts de Swing

- Plus de composants, offrant plus de possibilités.
- Les composants Swing dépendent moins de la plateforme :
  - Il est plus facile d'écrire une application qui satisfasse au slogan *"Write once, run everywhere"*
  - Swing peut pallier aux faiblesses (bogues ?) de chaque gestionnaire de fenêtre.



# Conventions de nommage







- Les composants Swing sont situés dans le paquetage `javax.swing` et ses sous paquetages.
- Ils portent des noms similaires à leurs correspondants de AWT précédés d'un J.
  - JFrame, JPanel, JTextField, JButton, JCheckBox, JLabel, etc.

# Outils de construction d'interfaces graphiques:

**I: Méthode native (code pur, tout à la main)**

**II: Méthode avec des outils graphiques (plugins)**

## II: Méthode avec des outils graphiques

Eclipse WindowBuilder		37	46,84%
Swing GUI Builder Netbeans		35	44,30%
Intelij IDEA		1	1,27%
JFormDesigner		3	3,80%
Jigloo		1	1,27%
Autre		11	13,92%


Matisse

MigLayout

Etc..



# **Le schéma Modèle-Vue-Contrôleur (MVC)**

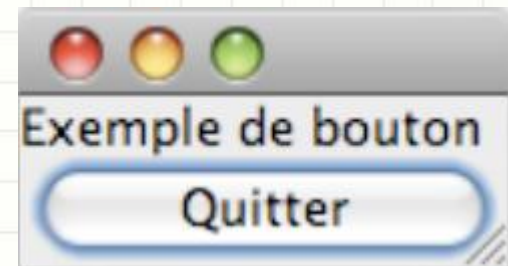


Le schéma Modèle-Vue-Contrôleur (MVC) est un mode de conception d'applications graphiques préconisant la distinction des données, de leur apparence graphique et des traitements qui peuvent être effectués. Un composant graphique est décomposé en trois parties :


- ✓ Le **modèle** contenant ses données et son état courant ;
- ✓ La **vue** qui correspond à son apparence graphique ;
- ✓ Le **contrôleur** associant des traitements à des événements pouvant se produire au sein du composant.

la décomposition MVC du bouton ci-dessous (instance de la classe JButton) est constituée :

- ❑ Du modèle contenant la chaîne de caractères “Quit” ;
- ❑ De la vue constituée d’un rectangle gris à bord bleu, d’une taille donnée, à une position donnée et au sein duquel est écrite la chaîne de caractères du modèle ;
- ❑ Du contrôleur implémentant le traitement à effectuer lors d’un click sur le bouton, c’est-à-dire l’appel à l’instruction de fermeture de la fenêtre.





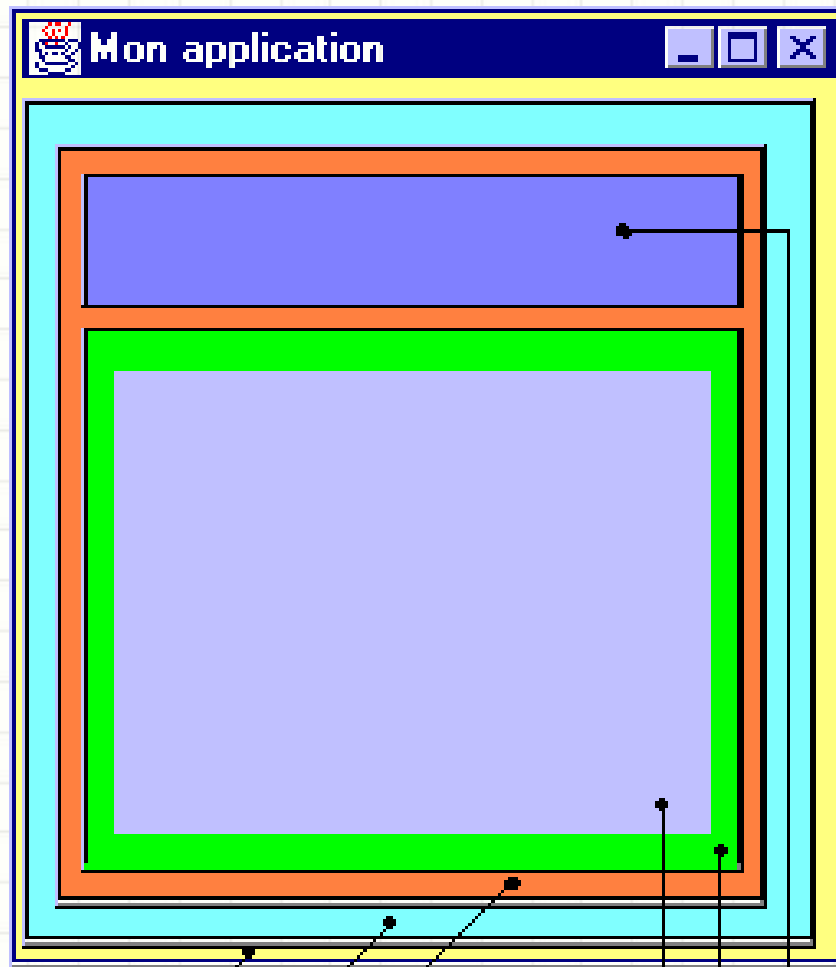


Tous les composants graphiques du package swing ont un modèle, une vue et peuvent avoir plusieurs contrôleurs.

Pour faciliter la programmation, certaines classes (tel que JButton) encapsulent le modèle et la vue dans une seule classe disposant de méthodes d'accès et de modifications des données ou de l'apparence. Dans ces cas-là, un modèle par défaut est utilisé.



# **Composition d'une interface graphique**



JFrame

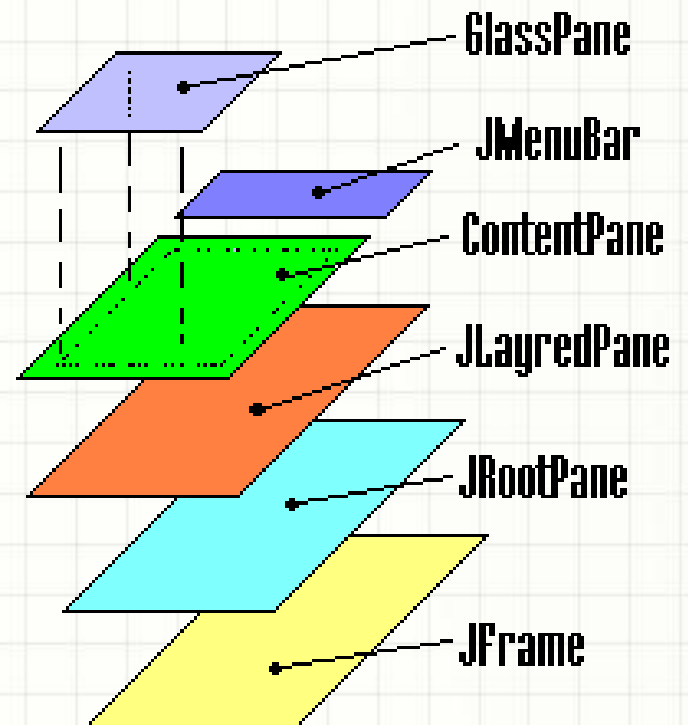
JRootPane

JLayeredPane

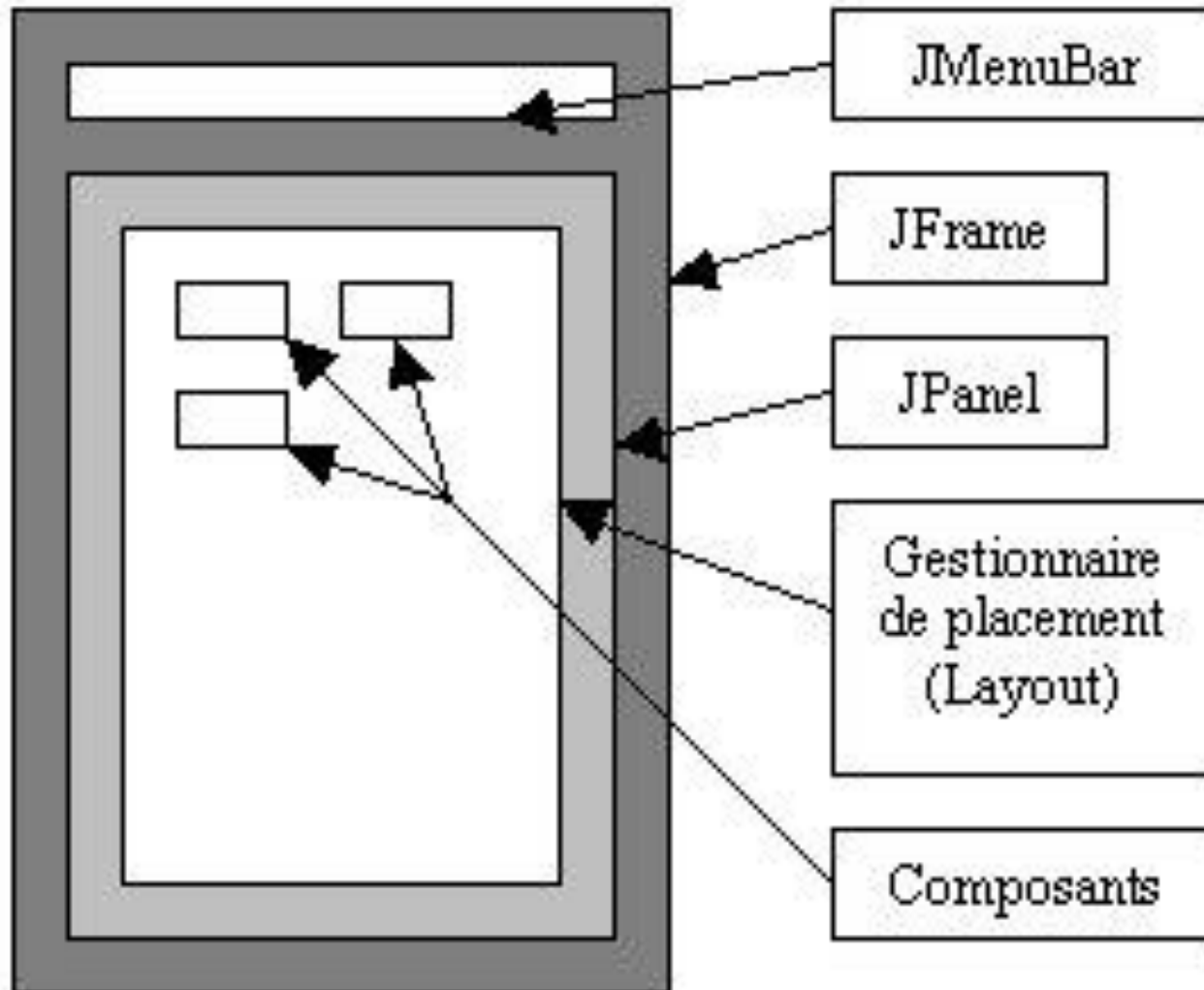
ContentPane

JMenuBar

GlassPane



Une fenêtre peut donc être construite selon le principe suivant :



## Le conteneur(Container):

En plus des caractéristiques d'un composant simple, un container contient un ensemble de composants ainsi que des contraintes sur leur disposition. L'ajout d'un composant à un container se fait par une méthode `add()`. Il existe plusieurs méthodes `add()` acceptant des paramètres d'entrée différents. La méthode à utiliser dépend du type de disposition choisi.


# Liste des principaux conteneurs SWING:

- Conteneurs généraux
  - JPanel
  - Box
  - JDesktopPane
- Conteneurs spécialisés
  - JRootPane
  - JLayeredPane
  - JScrollPane
  - JTabbedPane
  - JSplitPane
- Menus et barre d'outils
  - JMenu
  - JPopupMenu
  - JMenuBar
  - JToolBar





# **Gestionnaires de placement (Layout manager)**



Les gestionnaires de placement (layout manager) permettent de disposer des composants dans un panneau en fonction des caractéristiques qui leurs sont propres, notamment leur *preferredSize*.

De nombreux gestionnaires de placement sont disponibles en Swing, et on peut évidemment programmer un gestionnaire de placement original, si le besoin s'en fait sentir.

Le *LayoutManager* doit être ajouté au container, soit lors de l'appel à son constructeur, soit par la méthode *setLayout()*.

**Remarque:** Un seul *LayoutManager* est autorisé par container.

## 1- Pas de gestionnaire:

Le conteneur n'a pas de gestionnaire de placement : *setLayout(**null**)*. Les composants ont un emplacement et une taille fixe.

## 2- BorderLayout:

Le conteneur est divisé en 5 zones : nord, sud, est ouest, et le centre. Il ne peut donc pas contenir plus de 5 composants. Dans les zones nord, sud est et ouest, les composants sont placés en fonction de leur taille préférée, le composant du centre occupe alors toute la place restante.

Les composant sont séparés horizontalement et verticalement par des espaces, qui peuvent être modifiés par les méthodes *setHgap(int g)* et *setVgap(int g)*

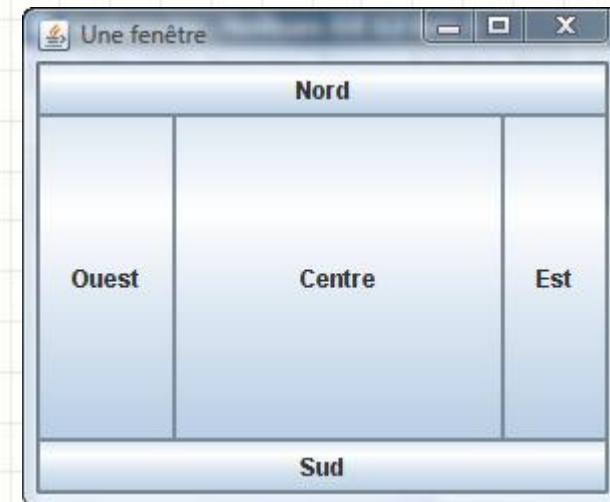
## 2- BorderLayout:

```
package gestionnaire;
import java.awt.BorderLayout;
import javax.swing.*.*;

public class Fenêtre extends JFrame {
    private JButton nord = new JButton("Nord");
    private JButton ouest = new JButton("Ouest");
    private JButton sud = new JButton("Sud");
    private JButton centre = new JButton("Centre");
    private JButton est = new JButton("Est");

    public Fenêtre() {
        setTitle("Une fenêtre");
        setSize(300, 250);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        add(nord, BorderLayout.NORTH);
        add(ouest, BorderLayout.WEST);
        add(sud, BorderLayout.SOUTH);
        add(centre, BorderLayout.CENTER);
        add(est, BorderLayout.EAST);
        setVisible(true);
    }

    public static void main(String[] args) {
        new Fenêtre();
    }
}
```



### 3- FlowLayout

Les composants sont placés les uns à la suite des autres horizontalement, en passant à la ligne suivante quand il n'y a plus de place sur la ligne. Chaque composant a sa taille préférée. Le gestionnaire de placement peut:

- centrer les composants : *FlowLayout.CENTER* (par défaut)
- aligner à gauche : *FlowLayout.LEFT*
- aligner à droite : *FlowLayout.RIGHT*
- leading : *FlowLayout.LEADING* les composants sont alignés à partir du début du conteneur ( à gauche si le composant est orienté *ComponentOrientation.LEFT\_TO\_RIGHT* et à droite si le composant est orienté *ComponentOrientation.RIGHT\_TO\_LEFT*
- trailing: *FlowLayout.TRAILING*

Les composant sont séparés horizontalement et verticalement par des espaces, qui peuvent être modifiés par les méthodes *setHgap(int g)* et *setVgap(int g)*



### 3- FlowLayout

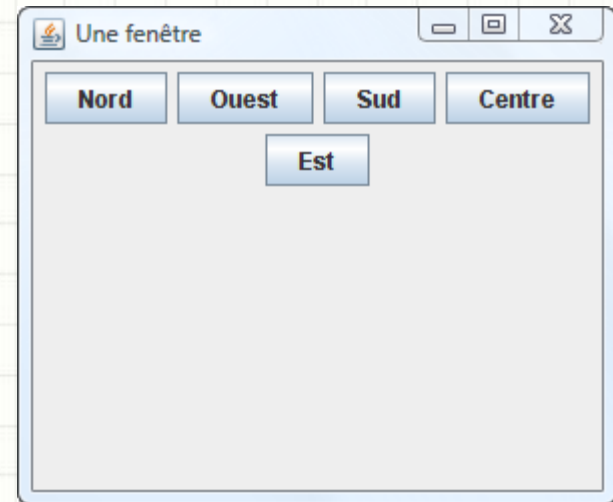
```
package gestionnaire;

import javax.swing.*.*;

public class Fenêtre extends JFrame {
    private JButton nord = new JButton("Nord");
    private JButton ouest = new JButton("Ouest");
    private JButton sud = new JButton("Sud");
    private JButton centre = new JButton("Centre");
    private JButton est = new JButton("Est");
    private JPanel panneau = new JPanel();

    public Fenêtre() {
        setTitle("Une fenêtre");
        setSize(300, 250);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        panneau.add(nord);
        panneau.add(ouest);
        panneau.add(sud);
        panneau.add(centre);
        panneau.add(est);
        add(panneau);
        setVisible(true);
    }

    public static void main(String[] args) {
        new Fenêtre();
    }
}
```



## 4- GridLayout:

Les composants sont placés dans un tableau à deux dimensions de *getColumns()* colonnes et *getRows()* lignes. S'il y a plus de *getColumns()\*getRows()* éléments on ajoute autant de colonnes qu'il faut pour que *getColumns()\*getRows()* soit supérieur ou égal au nombres de composants du conteneur.

Les composant sont séparés horizontalement et verticalement par des espaces, qui peuvent être modifiés par les méthodes *setHgap(int g)* et *setVgap(int g)*.

Le placement dépend de la propriété *componentOrientation* du composant

## 4- GridLayout

```
package gestionnaire;
```

```
import java.awt.*;
```

```
import javax.swing.*;
```

```
public class Fenêtre extends JFrame {  
    private JButton nord = new JButton("Nord");  
    private JButton ouest = new JButton("Ouest");  
    private JButton sud = new JButton("Sud");  
    private JButton centre = new JButton("Centre");  
    private JButton est = new JButton("Est");
```

```
    public Fenêtre() {  
        setTitle("Une fenêtre");  
        setSize(300, 250);  
        setDefaultCloseOperation(EXIT_ON_CLOSE);  
        setLayout(new GridLayout(3, 2));  
        add(nord);  
        add(ouest);  
        add(sud);  
        add(centre);  
        add(est);  
        setVisible(true);  
    }  
    public static void main(String[] args) {  
        new Fenêtre();  
    }  
}
```



## 5- BoxLayout

Permet de placer des composants soit horizontalement, soit verticalement. Ce gestionnaire de placement respecte la taille maximum et l'alignement de chaque composant.

Les quatre *BoxLayout* sont :

1. *X\_AXIS* : les composants sont placés horizontalement de gauche à droite, en respectant leur alignement par rapport à une ligne horizontale passant au milieu du conteneur.
2. *Y\_AXIS* : les composants sont placés verticalement de haut en bas, en respectant leur alignement par rapport à une ligne verticale passant au milieu du conteneur.
3. *LINE\_AXIS* : comme *X\_AXIS*, mais en respectant l'orientation du conteneur, ce sera éventuellement de droite à gauche.
4. *PAGE\_AXIS* : comme *Y\_AXIS*, mais en respectant l'orientation du conteneur, ce sera éventuellement de bas en haut.

## 5- BorderLayout

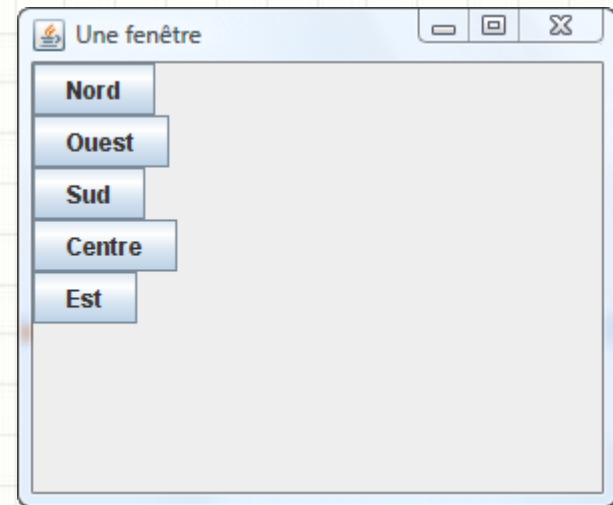
```
package gestionnaire;

import java.awt.*;
import javax.swing.*;

public class Fenêtre extends JFrame {
    private JButton nord = new JButton("Nord");
    private JButton ouest = new JButton("Ouest");
    private JButton sud = new JButton("Sud");
    private JButton centre = new JButton("Centre");
    private JButton est = new JButton("Est");

    public Fenêtre() {
        setTitle("Une fenêtre");
        setSize(300, 250);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new BorderLayout(getContentPane(), BorderLayout.Y_AXIS));
        add(nord);
        add(ouest);
        add(sud);
        add(centre);
        add(est);
        setVisible(true);
    }

    public static void main(String[] args) {
        new Fenêtre();
    }
}
```



## 6- CardLayout

Permet de placer des composants les uns au dessus des autres de façon qu'un seul composant, celui qui est au dessus, soit visible. Les ajouts dans un conteneur géré par un *CardLayout* se font en spécifiant pour chaque composant un nom.

```
add(Component c, String s);
```

Le composant affiché est par défaut le premier composant ajouté. On peut changer le composant affiché en utilisant une des méthodes de *CardLayout*.



## 6- CardLayout

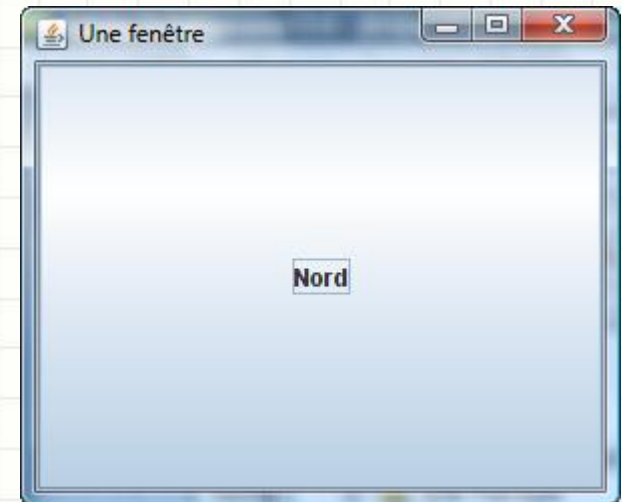
```
package gestionnaire;

import java.awt.*;
import javax.swing.*;

public class Fenêtre extends JFrame {
    private JButton nord = new JButton("Nord");
    private JButton ouest = new JButton("Ouest");
    private JButton sud = new JButton("Sud");
    private JButton centre = new JButton("Centre");
    private JButton est = new JButton("Est");

    public Fenêtre() {
        setTitle("Une fenêtre");
        setSize(300, 250);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new CardLayout());
        add(nord, "Nord");
        add(ouest, "Ouest");
        add(sud, "Sud");
        add(centre, "Centre");
        add(est, "Est");
        setVisible(true);
    }

    public static void main(String[] args) {
        new Fenêtre();
    }
}
```



## 7- GridBagLayout

Pour plus de détails voir : [How to Use GridBagLayout](#) de Sun.

Pour pouvoir utiliser un *GridBagLayout* il faut ajouter les composants dans le conteneur en leur associant une contrainte de type *GridBagConstraints* : Une contrainte de type *GridBagConstraints* contient les informations suivantes :

### 1. informations de position

- *gridx* : position en x dans la grille.
- *gridy* : position en y dans la grille.
- *gridwidth* : nombre de colonnes occupées par le composant.
- *gridheight* : nombre de lignes occupées par le composant.

## stratégie de mise en forme du composant

- *weightx* :
- *weighty* :
- *anchor* : ancrage du composant dans la cellule :
- *fill* : indique comment remplir la cellule si le composant est plus petit que la cellule. NONE  
HORIZONTAL VERTICAL BOTH
- *insets* : espace autour du composant.
- *ipadx* : espace à gauche et à droite du composant.
- *ipady* : espace au dessus et en dessous du composant.





## 8- Fabriquer un LayoutManager

```
public class MonLayout implements LayoutManager {
    public MonLayout() {
        super();
    }
    public void addLayoutComponent
        (String name, Component comp) {
    }
    public void removeLayoutComponent
        (Component comp) {
    }
    public Dimension preferredLayoutSize
        (Container parent) {
        return null;
    }
    public Dimension minimumLayoutSize(Container parent) {
        return null;
    }
    }
    public void layoutContainer(Container parent) {
        Component[] comps = parent.getComponents();
        int l = parent.getWidth();
        int h = parent.getHeight();
        int nl = comps.length/2+1;
        int nc = (comps.length+1)/2;
        int th = h/nl;
        int tl = l/nc;
        for(int i = 0; i<comps.length;++i ){
            comps[i].setBounds(0,    i/2*th,
                                l-(i/2)*tl, th);

            ++i;
            if(i==comps.length) break;
            comps[i].setBounds(l-((i+1)/2)*tl,
                                (i+1)/2*th, tl, h-((i+1)/2)*th);
        }
    }
}
```



# Calculs nécessaires pour placer les composants

 **Mon dialogue** 

**Nom**

**Prenom**

**Adresse**

☐ Tennis

☐ Squash

☐ Natation

☐ Athlétisme

☐ Randonnée

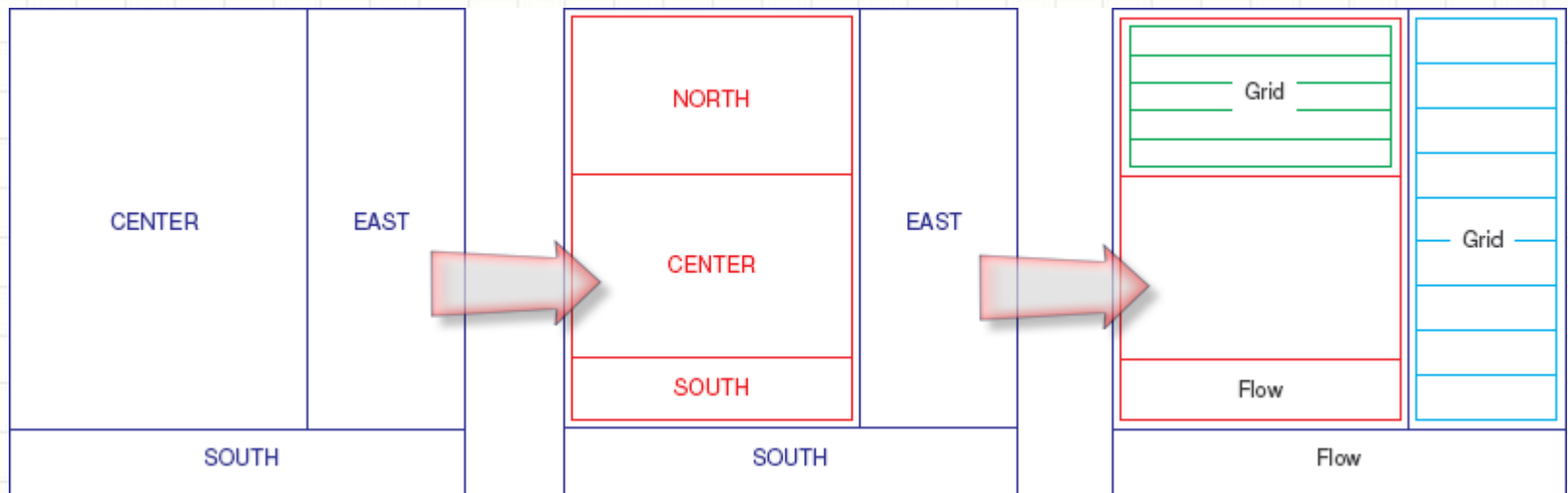
☐ Foot

☐ Basket

☐ Volley

☐ Petanque

**Sexe** ☒ Homme ☐ Femme



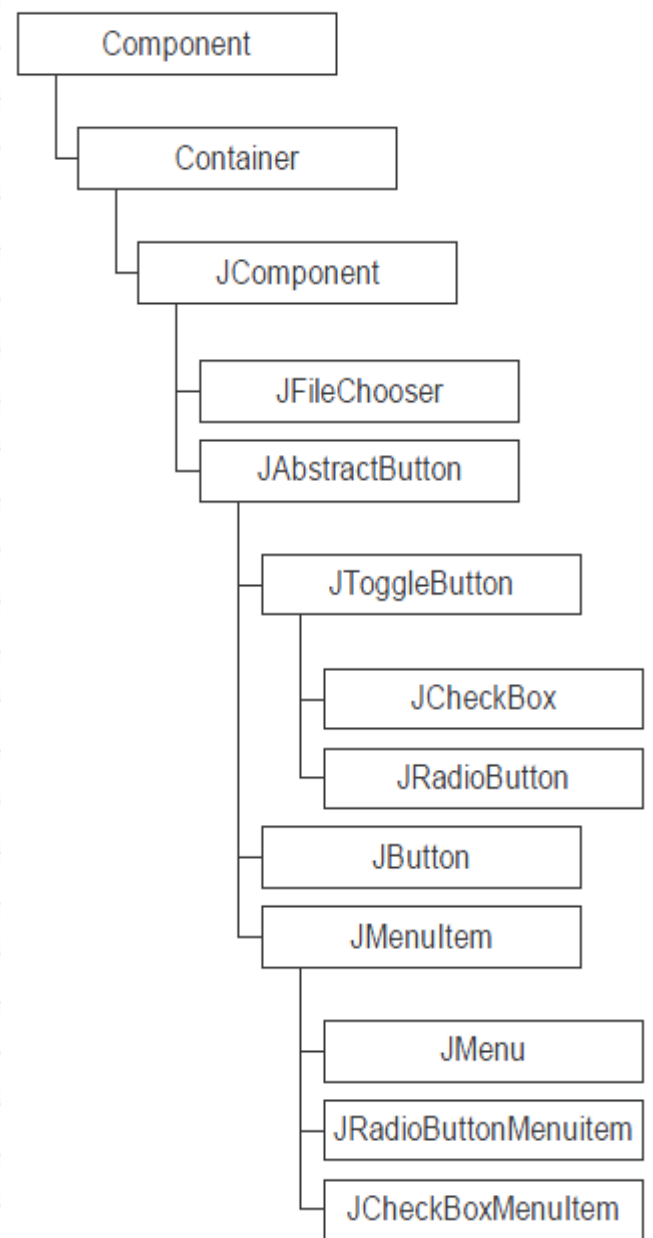
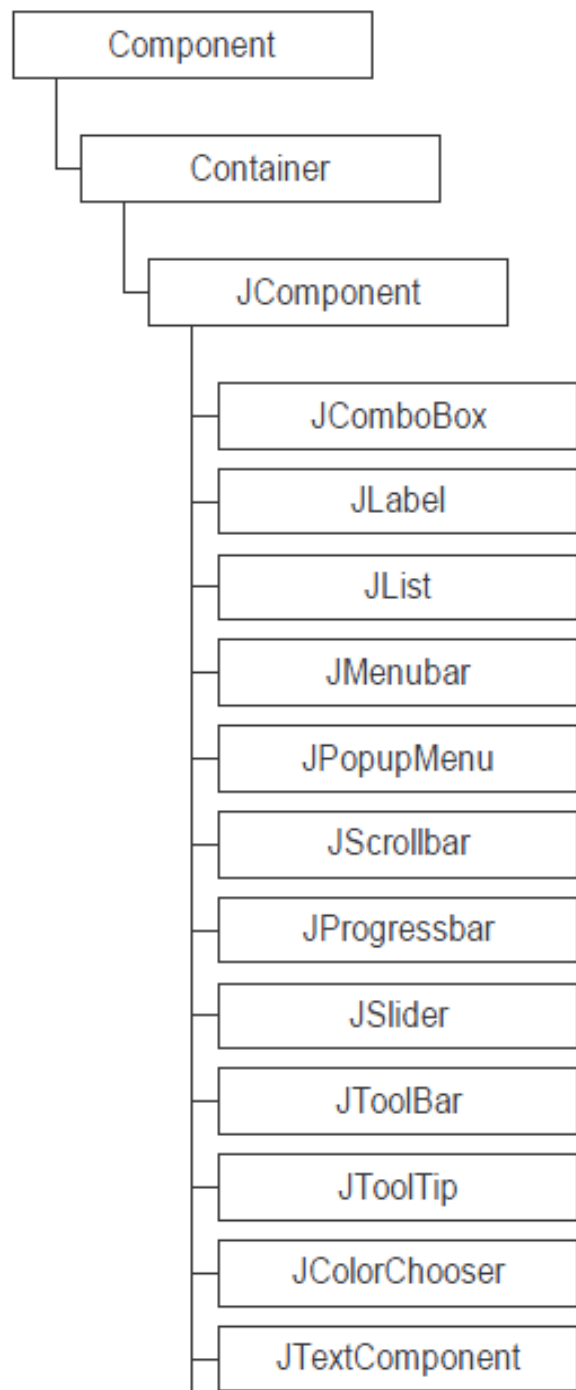


# **Les Composants graphiques (Component)**



# Composer une fenêtre

- Créer une fenêtre (1)
- Créer un ou des composants intermédiaires (2)
  - Pour *JFrame*, un *conteneur* est associé implicitement (ContentPane)
- Créer des composants de base (3)
- Insérer (3) dans (2)
- Insérer (2) dans (1) (s'ils ne sont pas déjà associés)
- Afficher
- Gérer les évènements



# Composants de base (pour obtenir des données)

- *JButton*
- *JCheckBox* a toggled on/off button displaying state to user.
- *JRadioButton* a toggled on/off button displaying its state to user.
- *JComboBox* a drop-down list with optional editable text field. The user can key in a value or select a value from drop-down list.
- *Jlist* allows a user to select one or more items from a list.
- *Jmenu* popup list of items from which the user can select.
- *Jslider* lets user select a value by sliding a knob.
- *JTextField* area for entering a single line of input.

# Composants de base pour afficher l'information

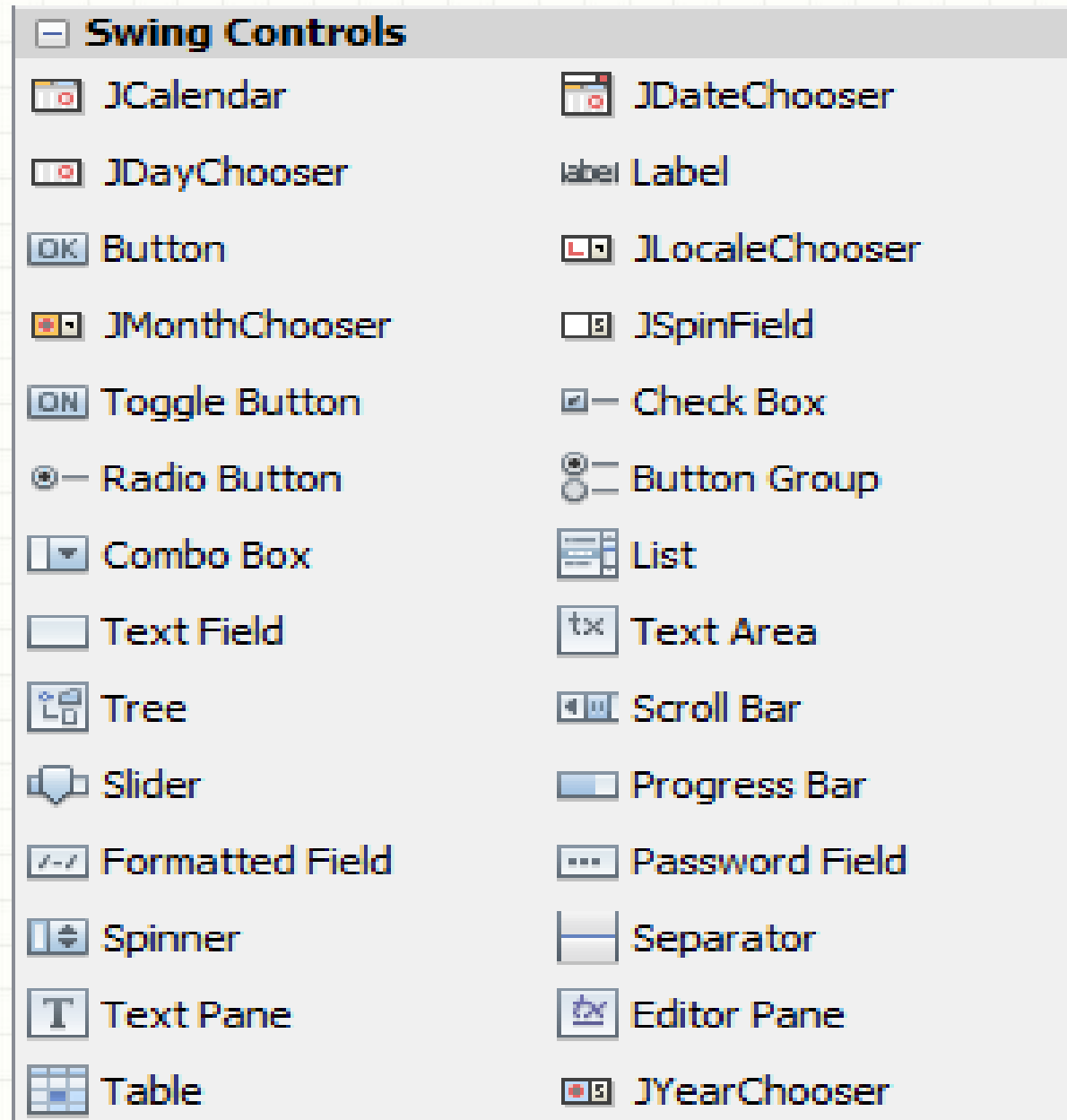
- *Jlabel* contains text string, an image, or both.
- *JProgressBar* communicates progress of some work.
- *JToolTip* describes purpose of another component.
- *Jtree* a component that displays hierarchical data in outline form.
- *Jtable* a component user to edit and display data in a two-dimensional grid.
- *JTextArea*, *JTextPane*, *JEditorPane*
  - define multi-line areas for displaying, entering, and editing text.

# Composants intermédiaires

- Utilisés pour organiser ou positionner d'autres composants (de base)
  - *JPanel* utilisé pour regrouper d'autres composants
  - *JScrollPane* fournir une vue avec scroll bars
  - *JSplitPane* divise en 2 composants
  - ...

```
JPanel p = new JPanel();  
p.add(new JButton("on"));  
p.add(new JButton("off"));
```

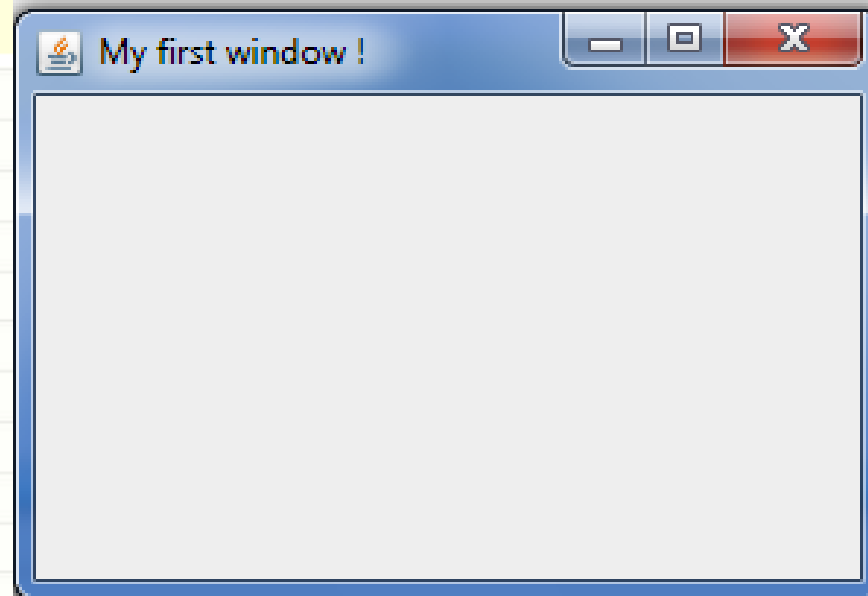
# Les composants Swing:





# JFrame:

```
import javax.swing.JFrame;  
public class PremiereFenetre {  
    public static void main (String[ ] args){  
        JFrame f = new JFrame( );  
        f.setTitle("My first window ! ") ;  
        f.setSize (600,600) ;  
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        f.setVisible (true) ;  
    }  
}
```

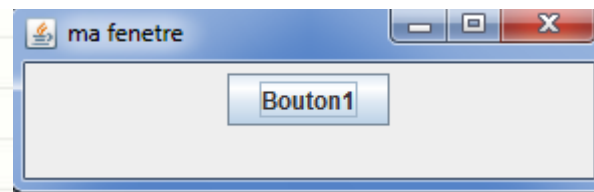


# Bouton:

```
import javax.swing.*;

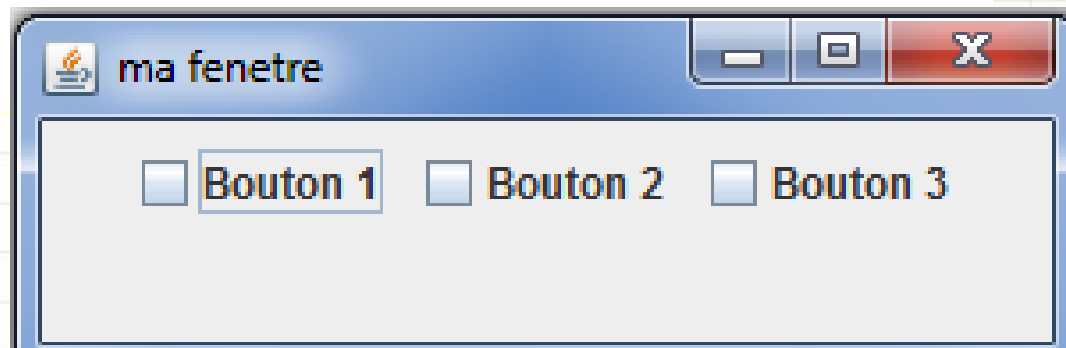
public class TestBouton {

    public static void main(String argv[]) {
        JFrame f = new JFrame("ma fenetre");
        f.setSize(300,100);
        JPanel pannel = new JPanel();
        JButton bouton1 = new JButton("Bouton1");
        f.getContentPane().add(pannel);
        f.setVisible(true);
    }
}
```



# CheckBox:

```
import javax.swing.*;  
public class TestJCheckBox1 {  
    public static void main(String argv[]) {  
        JFrame f = new JFrame("ma fenetre");  
        f.setSize(300,100);  
        JPanel pannel = new JPanel();  
        JCheckBox bouton1 = new JCheckBox("Bouton 1");  
        pannel.add(bouton1);  
        JCheckBox bouton2 = new JCheckBox("Bouton 2");  
        pannel.add(bouton2);  
        JCheckBox bouton3 = new JCheckBox("Bouton 3");  
        pannel.add(bouton3);  
        f.getContentPane().add(pannel);  
        f.setVisible(true);  
    }  
}
```

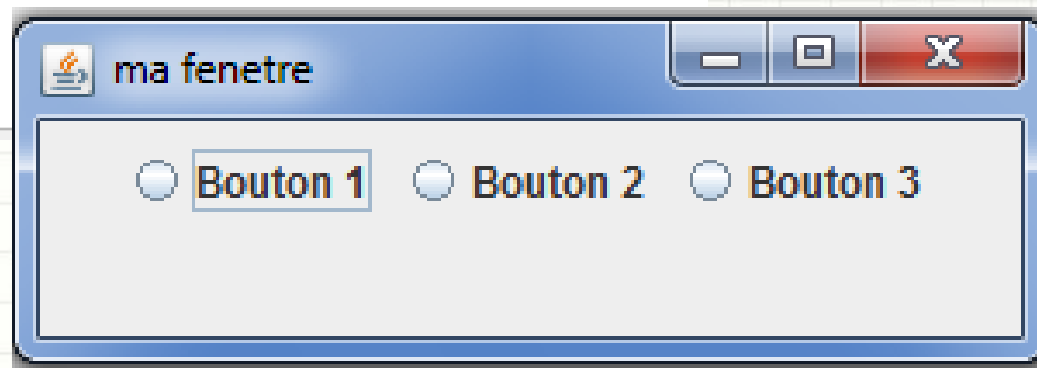


# BoutonRadio:

```
import javax.swing.*;

public class TestGroupButton1 {

    public static void main(String argv[]) {
        JFrame f = new JFrame("ma fenetre");
        f.setSize(300,100);
        JPanel pannel = new JPanel();
        ButtonGroup groupe = new ButtonGroup();
        JRadioButton bouton1 = new JRadioButton("Bouton 1");
        groupe.add(bouton1);
        pannel.add(bouton1);
        JRadioButton bouton2 = new JRadioButton("Bouton 2");
        groupe.add(bouton2);
        pannel.add(bouton2);
        JRadioButton bouton3 = new JRadioButton("Bouton 3");
        groupe.add(bouton3);
        pannel.add(bouton3);
        f.getContentPane().add(pannel);
        f.setVisible(true);
    }
}
```



# Les Menus:

## **Swing Menus**



Menu Bar



Menu



Menu Item



Menu Item / CheckBox



Menu Item / RadioButton



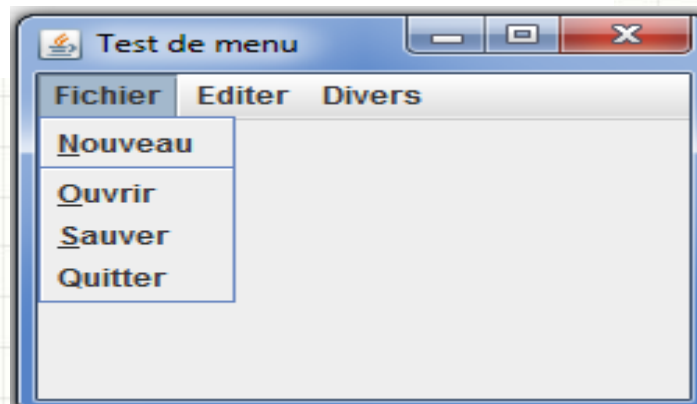
Popup Menu



Separator

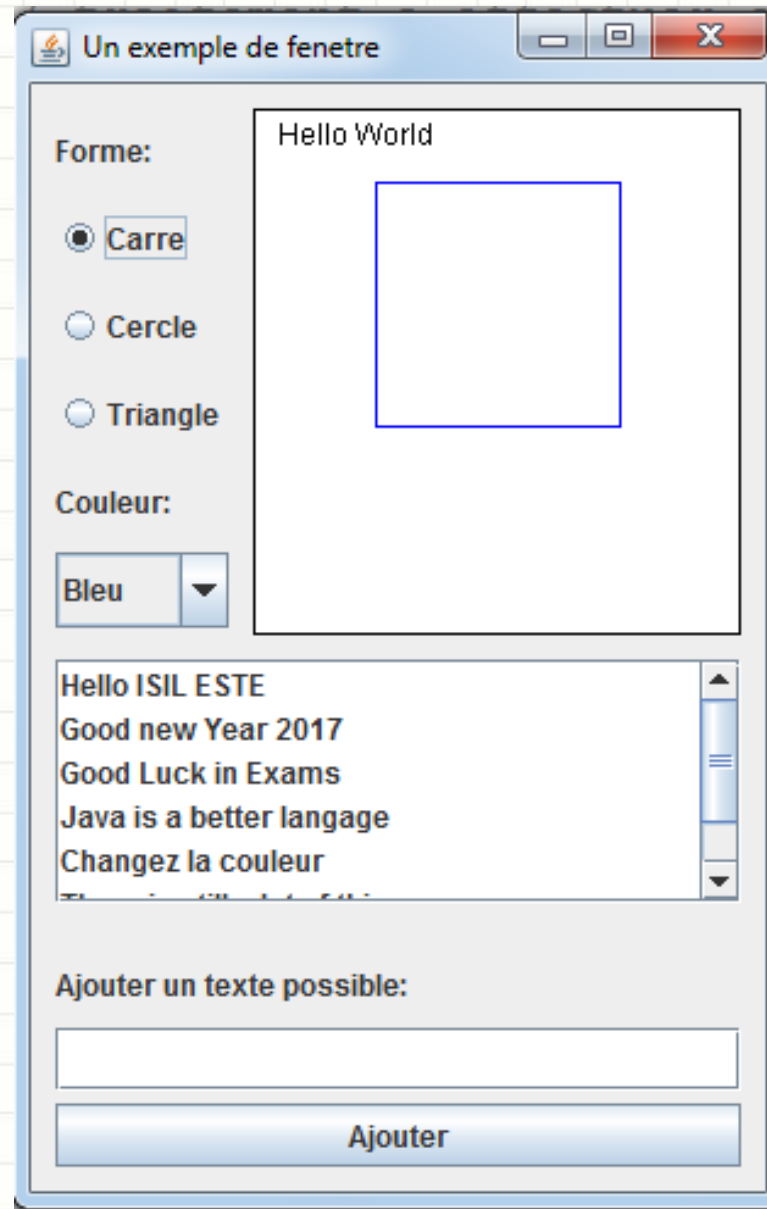
# Les menus:

```
// Création du menu Fichier
JMenu fichierMenu = new JMenu("Fichier");
JMenuItem item = new JMenuItem("Nouveau", 'N');
item.addActionListener(afficherMenuListener);
fichierMenu.add(item);
item = new JMenuItem("Ouvrir", 'O');
item.addActionListener(afficherMenuListener);
fichierMenu.add(item);
item = new JMenuItem("Sauver", 'S');
item.addActionListener(afficherMenuListener);
fichierMenu.insertSeparator(1);
fichierMenu.add(item);
item = new JMenuItem("Quitter");
item.addActionListener(afficherMenuListener);
fichierMenu.add(item);
```





# Travail Pratique global



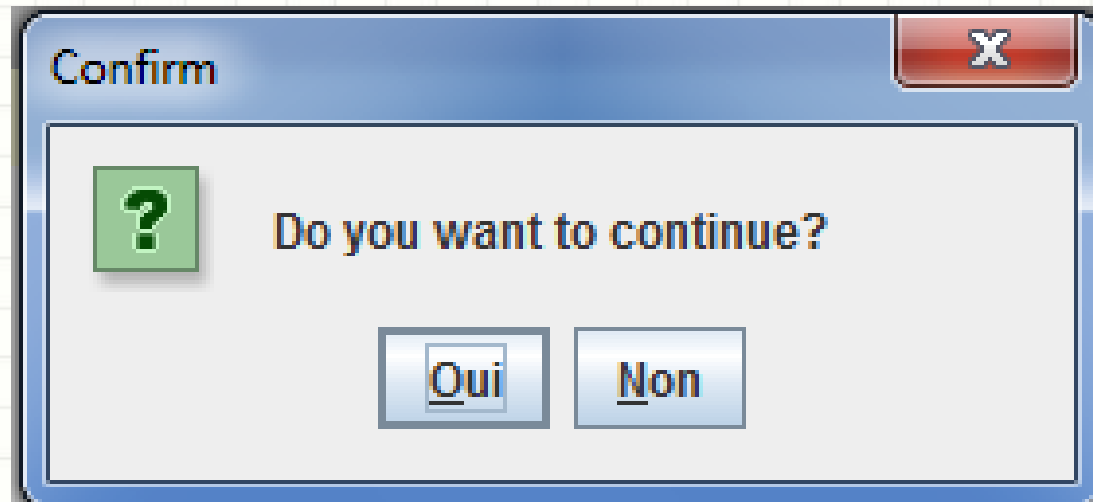
Voir TP



# **Les Boites de dialogue**

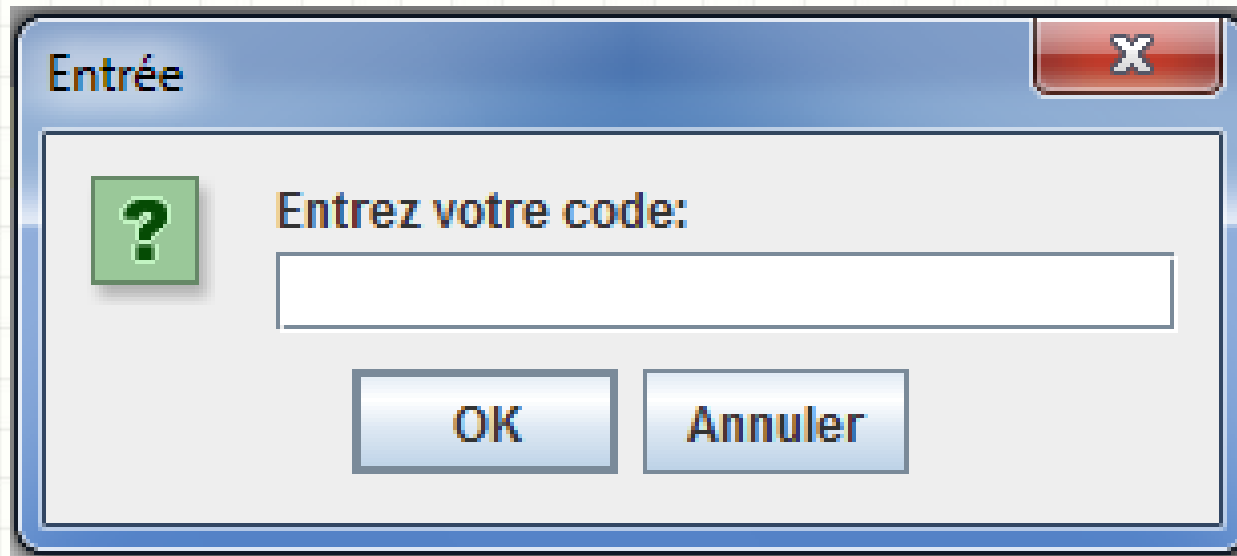
# OutputBox

```
package DialogBox;
import javax.swing.JOptionPane;
public class OutDialog {
    public static void main(String[] args) {
        //JDialog.setDefaultLookAndFeelDecorated(true);
        int response = JOptionPane.showConfirmDialog(null, "Do you want to continue?",
"Confirm",JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE);
        if (response == JOptionPane.NO_OPTION) {
            System.out.println("No button clicked"+response);
        } else if (response == JOptionPane.YES_OPTION) {
            System.out.println("Yes button clicked"+response);
        } else if (response == JOptionPane.CLOSED_OPTION) {
            System.out.println("JOptionPane closed"+response);
        }
    }
}
```




# InputDialog

```
package DialogBox;  
import javax.swing.JOptionPane;  
public class InpDialog {  
    public static void main(String[] a) {  
        String input = JOptionPane.showInputDialog("Entrez votre  
code:");  
        System.out.println(input);  
    }  
}
```






# **Les Contrôleurs d'évènements (listeners)**



Dans le contexte d'une interface graphique (Swing, AWT, etc), les listeners permettent au programmeur de réagir suite aux actions de l'utilisateur (clic de souris, touche du clavier enfoncée, etc).

Les « listeners » sont des interfaces. Ces interfaces fournissent une ou plusieurs méthodes qui peuvent donc être implémentées différemment selon les cas et les besoins, pour répondre aux événements.





Les interfaces « listener » sont présentes principalement dans le package `java.awt.event`, mais également dans le package `javax.swing.event`.

Chaque listener dispose d'une classe `Event` associée. Cette classe étend `java.awt.event.EventObject` et fournit une description de l'évènement capturé. Par exemple, la classe `ActionEvent` décrit les évènements capturés par un `ActionListener`.

# Comment utiliser les listeners

1ère possibilité : implémentation de l'interface dans la classe principale.

```
import java.awt.event.ActionListener;

public class MaClasse implements ActionListener {

    JButton monBouton = new JButton("Mon Bouton");
    JButton monBouton2 = new JButton("Mon Bouton2");

    public MaClasse() {
        monBouton.addActionListener(this);
        monBouton2.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        // Étape 2bis
        if(e.getSource() == monBouton) {
            // Bouton 1 a été cliqué
        }else {
            // Bouton 2 a été cliqué
        }
    }
}
```

## 2ème méthode : Utilisation des classes anonymes

```
public class MaClasse {  
    JButton monBouton = new JButton("Mon Bouton");  
    JButton monBouton2 = new JButton("Mon Bouton 2");  
    monBouton.addActionListener(new ActionListener() {  
        public void actionPerformed(ActionEvent e) {  
            // Cette méthode ne sera appelée que pour les événements sur le bouton  
        }  
    });  
    monBouton2.addActionListener(new ActionListener() {  
        public void actionPerformed(ActionEvent e) {  
            // Cette méthode ne sera appelée que pour les événements sur le bouton  
        }  
    });  
}
```

## Une autre possibilité qui est simple:

```
public class MaClasse {  
    JButton monBouton = new JButton("Mon Bouton...");  
    JButton monBouton2 = new JButton("Mon Bouton 2...");  
    ActionListener listener = new ActionListener() {  
        public void actionPerformed(ActionEvent e) {  
            if(e.getSource() == monBouton) {  
                // Bouton 1 a été cliqué  
            }else {  
                // Bouton 2 a été cliqué  
            }  
        }  
    };  
    monBouton.addActionListener(listener);  
    monBouton2.addActionListener(listener);  
}
```

### 3<sup>ème</sup> méthode : création d'une classe dédiée.

```
// Fichier : MonListener.java  
public class MonListener implements ActionListener {  
    public void actionPerformed(ActionEvent e) {  
    }  
}
```

Et dans votre classe principale :

```
monButton.addActionListener(new MonListener());
```

Cette possibilité est très simple, il vous suffit d'implémenter le comportement voulu dans la classe qui implémente le listener, puis, vous créez une instance de cette classe que vous passez en paramètre de la méthode addXXXListener située dans votre classe où sont déclarés les composants (boutons, listes, etc).

**ActionListener**: est, comme son nom l'indique, un listener utilisé pour réagir aux actions utilisateurs. Celles-ci sont multiples, la principale étant l'activation d'un bouton (par un clic ou par appui de la touche Entrée lorsque le bouton est sélectionné). Une seule méthode est déclarée dans cette interface : `public void actionPerformed(java.awt.event.ActionEvent e)`.

La classe `ActionEvent` étend la classe `java.util.EventObject`, et hérite donc de ses méthodes. Parmi elles, `getSource()` est particulièrement intéressante. Elle renvoie l'objet concerné par l'événement (par exemple le bouton qui a été cliqué).

Cela nous permettra de différencier les composants sources dans l'implémentation de la méthode `actionPerformed`.

La classe `ActionEvent` fournit quant à elle quelques méthodes spécifiques aux événements d'action. Les plus utilisées sont `getWhen()` et `getActionCommand()`.



**KeyListener** : est utilisé pour réagir aux événements du clavier, et est donc utilisable sur des composants permettant la saisie de texte (JTextField, JTextArea, etc).


Trois méthodes sont déclarées dans l'interface du KeyListener : `keyTyped(KeyEvent e)`, `keyPressed(KeyEvent e)` et `keyReleased(KeyEvent e)`. Elles permettent respectivement de réagir lorsqu'une touche a été : tapée (pressé puis relâché), pressée, relâchée.

La classe `KeyEvent` étend `java.util.EventObject` et dispose donc des méthodes déclarées dans cette classe (notamment `getSource()`), mais fournit également une dizaine d'autres méthodes spécifiques aux événements relatifs au clavier. Parmi elles : `getKeyChar()` retourne le caractère associé à la touche appuyée, `getKeyCode()` récupère le code de la touche pressée, `isActionKey()` retourne `true` si la touche appuyée est une touche d'action (CAPS LOCK, Ver Num, etc), et `getKeyText(int keyCode)` retourne le texte associée à la touche (par ex. F1, A, etc).

## MouseListener :

Est utilisé pour les événements relatifs à la souris (clics, déplacements).

5 méthodes sont déclarées dans l'interface `MouseListener` :  
`mouseClicked(MouseEvent e)` prévient des clics (la souris a été pressée puis relâchée), `mousePressed(MouseEvent e)` pour les pressions sur la souris (donc on enfonce le bouton sans le relâcher), `mouseReleased(MouseEvent e)` prévient du relâchement d'un bouton de la souris, `mouseEntered(MouseEvent e)` indique que la souris est entrée dans l'espace d'un composant, `mouseExited(MouseEvent e)` indique que la souris est sortie de l'espace d'un composant.



La classe `MouseEvent` étend `java.util.EventObject` et dispose donc des méthodes déclarées dans cette classe (notamment `getSource()` ), mais fournit également 12 autres méthodes spécifiques aux événements relatifs à la souris, notamment `getButton()` retourne le bouton qui a été cliqué, `getClickCount()` retourne le nombre de clics (utile pour gérer le double clic), `getLocationOnScreen()` retourne un objet `Point` représentant la position de la souris à l'écran, et enfin `isPopupTrigger()` précise si le bouton cliqué est celui habituellement utilisé pour afficher la liste déroulante (bouton droit sur le bureau Windows par exemple).

**WindowListener:** est utilisé pour les événements relatifs aux fenêtres (activation, fermeture, ouverture, etc).

Cette interface déclare 7 méthodes :

`windowActivated(WindowEvent e)` indique que la fenêtre a été activé, `windowDeactivated(WindowEvent e)`

indique que la fenêtre n'est plus la fenêtre

active, `windowClosed(WindowEvent e)` indique que la

fenêtre a été fermé, `windowClosing(WindowEvent e)`

indique que l'utilisateur a demandé la fermeture de la

fenêtre, `windowOpened(WindowEvent e)` est appelé la


première fois que la fenêtre est rendue visible,

`windowIconified(WindowEvent e)` indique que la fenêtre

a été réduite dans la barre de tâche,

`windowDeiconified(WindowEvent e)` indique que la

fenêtre a été restauré depuis la barre de tâche.



La classe `WindowEvent` étend `java.util.EventObject` et dispose donc des méthodes déclarées dans cette classe (notamment `getSource()` ), mais fournit également 5 autres méthodes spécifiques aux événements relatifs aux fenêtres, notamment `getNewState()` et `getOldState()` qui fournissent respectivement le nouvel état et l'ancien état de la fenêtre, mais aussi `getWindow()`, qui retourne la fenêtre source de l'événement.

Afin d'ajouter un `WindowListener` sur une fenêtre, vous disposez de la méthode `addWindowListener(WindowListener)` de la classe `Window` (étendue par la classe `JFrame` notamment).



**FocusListener:** est utilisé pour les événements relatifs au focus clavier.

Cette interface déclare 2 méthodes : `focusGained(FocusEvent e)` indique que le composant a gagné le focus clavier tandis que `focusLost(FocusEvent e)` indique que le composant a perdu le focus clavier.

La classe `FocusEvent` étend `java.util.EventObject` et dispose donc des méthodes déclarées dans cette classe (notamment `getSource()`), mais fournit également 3 autres méthodes spécifiques aux événements relatifs aux fenêtres, notamment `isTemporary()` qui indique si le composant n'a le focus que temporairement, et `getOppositeComponent()` qui retourne l'autre composant impliqué dans le changement de focus.

Afin d'ajouter un `FocusListener`, vous disposez de la méthode `addFocusListener(FocusListener)` de la classe `Component`.

**ItemListener:** est utilisé pour les événements relatifs aux éléments (liste, checkbox, etc).

Cette interface déclare une seule méthode :  
`itemStateChanged(ItemEvent e)` qui indique que l'élément a changé d'état.

La classe `ItemEvent` étend `java.util.EventObject` et dispose donc des méthodes déclarées dans cette classe (notamment `getSource()` ), mais fournit également 4 autres méthodes spécifiques aux événements relatifs aux fenêtres, notamment `getItem()` qui retourne l'élément affecté par l'évènement, `getStateChange()` retourne le nouvel état de l'élément (sélectionné ou désélectionné), et `getItemSelectable()` qui retourne le composant originaire de l'évènement.

Afin d'ajouter un `ItemListener` sur les composants qui le permettent (les boutons, `JcomboBox`, les listes, etc), vous disposez de la méthode `addItemListener(ItemListener)`.





**FIN DE LA PARTIE V**