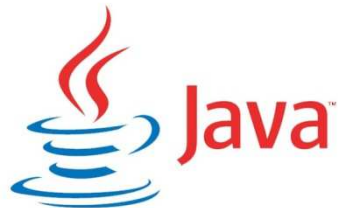




Java 2

Zied ELLEUCH



- Introduction
- Les composants SWING
- Gestionnaire de disposition
- Gestion des événements
- Accès à la base de données
- Le graphisme en JAVA.
- Les applets
- Thread
- Les collections et type générique

Introduction

- Il existe trois grandes familles de composant graphique en JAVA:
 - **AWT** (*Abstract Window Toolkit*, JDK 1.1)
 - **SWING** (JDK/SDK 1.2)
 - **JavaFX** (JDK 8)
- Swing et AWT font partie de JFC (*Java Foundation Classes*) qui offre des facilités pour construire des interfaces graphiques
- Swing est construit au-dessus de AWT
 - **même gestion des événements**
 - **les classes de *Swing* héritent des classes de AWT**

Introduction : AWT

- Les classes du toolkit AWT (Abstract Windows Toolkit) permettent d'écrire des interfaces graphiques indépendantes du système d'exploitation sur lesquelles elles vont fonctionner.
- Cette librairie utilise le système graphique de la plateforme d'exécution (Windows, MacOS, X-Window) pour afficher les objets graphiques.
- Le toolkit contient des classes décrivant les composants graphiques, les polices, les couleurs et les images.

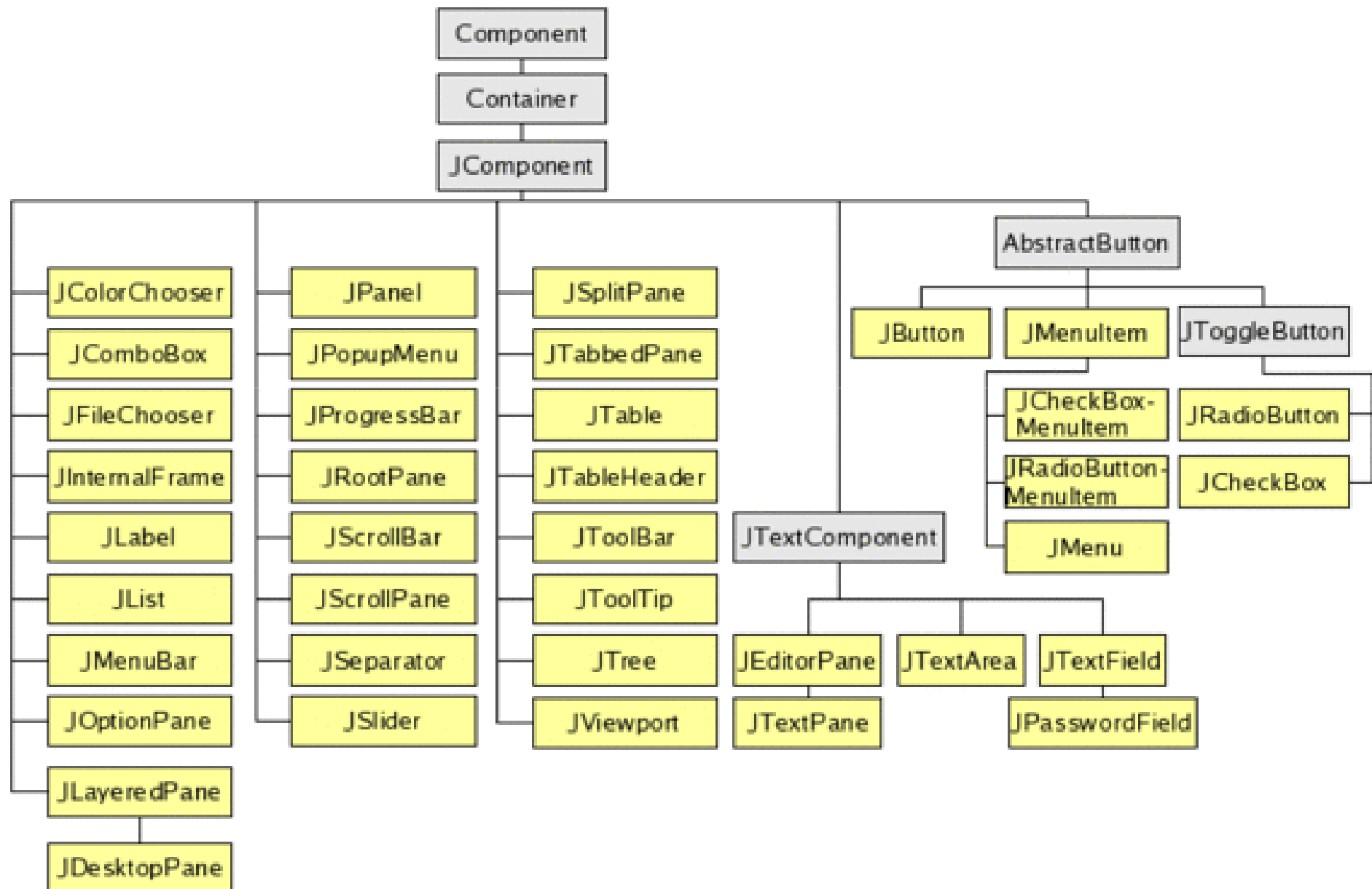
Introduction : SWING

- Swing fait partie de la bibliothèque Java Foundation Classes (JFC). C'est une API (Applications Programming Interface) dont le but est similaire à celle de l'AWT mais dont le mode de fonctionnement et d'utilisation est complètement différent.
- Swing a été intégrée au JDK depuis sa version 1.2. Cette bibliothèque existe séparément pour le JDK 1.1.
- La bibliothèque JFC contient :
 - **l'API Swing : de nouvelles classes et interfaces pour construire des interfaces graphiques.**
 - **L'API AWT.**
 - **Accessibility API**
 - **2D API: support du graphisme en 2D •**
 - **API pour l'impression et le cliquer/glisser •**

Introduction : SWING

- Les composants Swing forment une nouvelle hiérarchie parallèle à celle de l'AWT.
- L'ancêtre de cette hiérarchie est le composant JComponent.
- Presque tous ces composants sont écrits en pure Java : ils ne possèdent aucune partie native sauf ceux qui assurent l'interface avec le système d'exploitation : JApplet, JDialog, JFrame, et JWindow.
- Cela permet aux composants de toujours avoir la même apparence quelque soit le système sur lequel l'application s'exécute.

Hiérarchie des Composants SWING



Introduction : SWING ou AWT?

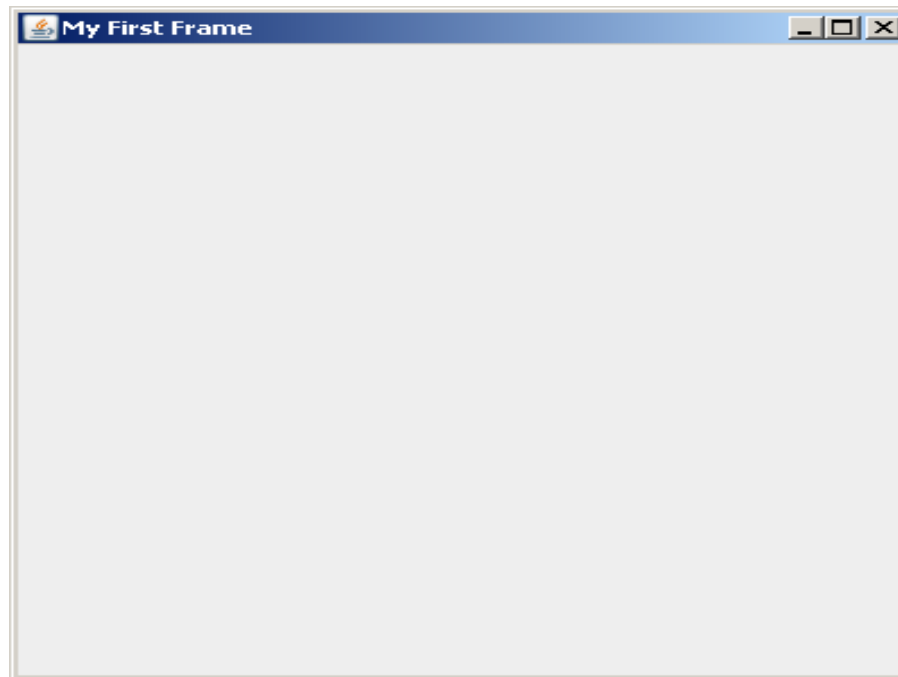
- Tous les composants de AWT ont leur équivalent dans Swing
 - **en plus joli**
 - **avec plus de fonctionnalités**
- Swing offre de nombreux composants qui n'existent pas dans AWT
 - ⇒ Il est fortement conseillé d'utiliser les composants Swing et ce cours sera donc centré sur Swing

Introduction : java.awt.Toolkit

- Les sous-classes de la classe abstraite **Toolkit** implantent la partie de AWT qui est en contact avec le système d'exploitation hôte.
- Quelques méthodes publiques :
 - **getScreenSize,**
 - **getScreenResolution,**
 - **beep,**
 - **getImage,**
 - **createImage,**
 - **getSystemEventQueue**
 - **getDefaultToolkit** fournit une instance de la classe qui implante **Toolkit** (classe donnée par la propriété **awt.toolkit**)

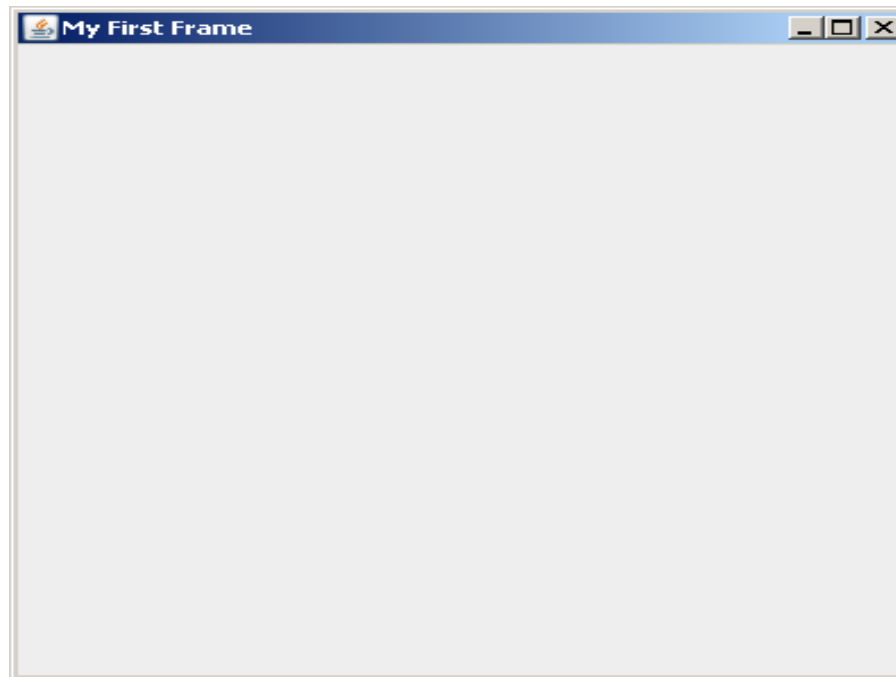
Introduction : Exemple 1

```
package graphique;
public class TestJFrame {
    public static void main(String[] args) {
        javax.swing.JFrame jFrame=new javax.swing.JFrame();
        jFrame.setTitle("My First Frame");
        jFrame.setSize(400,400);
        jFrame.setLocation(200, 200);
        jFrame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
        jFrame.setVisible(true);
    }
}
```



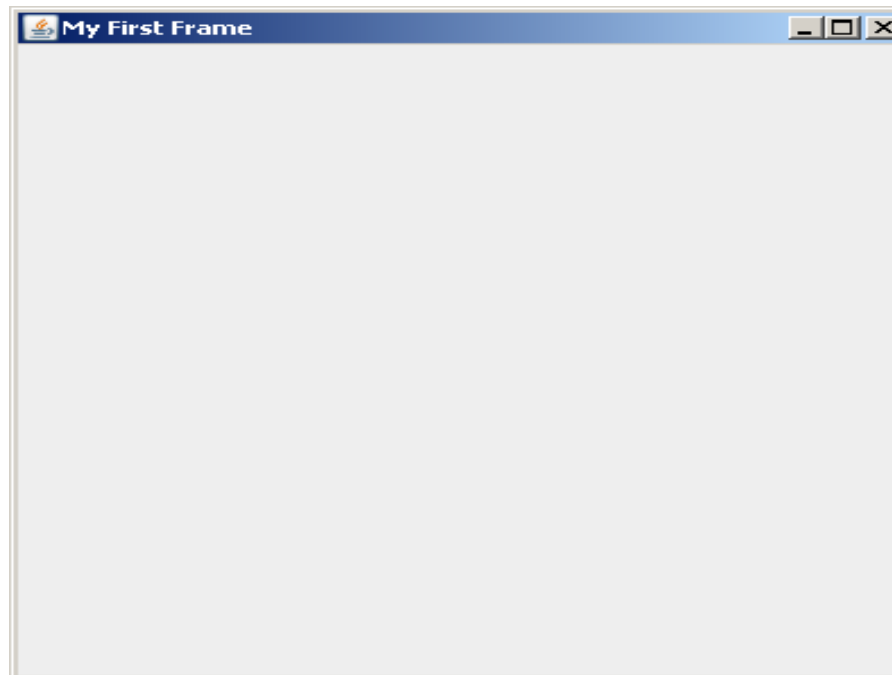
Introduction : Exemple 2

```
package graphique;  
public class TestJFrame {  
    public static void main(String[] args) {  
        javax.swing.JFrame jFrame=new javax.swing.JFrame("My First Frame");  
        jFrame.setBounds(200, 200, 400, 400);  
        jFrame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);  
        jFrame.setVisible(true);  
    }  
}
```



Introduction : Exemple 3

```
package graphique;
public class TestJFrame {
    public static void main(String[] args) {
        javax.swing.JFrame jFrame=new javax.swing.JFrame("My First Frame");
        jFrame.setSize(400,400);
        // Permet de centrer la fenetre par rapport à l'écran
        jFrame.setLocationRelativeTo(null);
        jFrame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
        jFrame.setVisible(true);
    }
}
```



Composants lourds & légers

Selon les bibliothèques de composants visuels utilisées, AWT ou Swing, Java n'adopte pas la même démarche d'implantation. Ceci est dû à l'évidence une évolution rapide du langage qui contient des couches successives de concepts.

En java, les composants awt, qui dérivent tous de la classe `java.awt.Component`, sont liés à la plate-forme locale d'exécution, car ils sont implémentés en code natif du système d'exploitation hôte et la Java Machine y fait appel lors de l'interprétation du programme Java. Ceci signifie que dès lors que vous développez une interface AWT sous windows, lorsque par exemple cette interface s'exécute sous MacOS, l'**apparence visuelle** et le **positionnement** des différents composants (boutons,...) **changent**. En effet la fonction système qui dessine un bouton sous Windows ne dessine pas le même bouton sous MacOS et des chevauchements de composants peuvent apparaître si vous les placez au pixel près

→ De tels composants dépendant du système hôte sont appelés en Java des **composants lourds**. En Java le composant lourd est identique en tant qu'objet Java et il est associé localement lors de l'exécution sur la plateforme hôte à un élément local dépendant du système hôte dénommé **peer**.

Composants lourds & légers

Par opposition aux composants lourds utilisant des **peer** de la machine hôte, les **composants légers** sont entièrement écrits en Java. En outre un tel composant léger n'est pas dessiné visuellement par le système, mais en Java. Ceci apporte une amélioration de portabilité et permet même de changer l'apparence de l'interface sur la même machine grâce au "look and feel". La classe `LookAndFeel` permet de déterminer le style d'aspect employé par l'interface utilisateur.

En Java on ne peut pas se passer de composants lourds (communiquant avec le système) car la Java Machine doit communiquer avec son système hôte. Par exemple la fenêtre étant l'objet visuel de base dans les systèmes modernes elle est donc essentiellement liée au système d'exploitation et donc ce sera en Java un composant lourd. Dans le package `swing` le nombre de composants lourds est réduit au strict minimum soient 4 genres de fenêtres.

Composants lourds & légers

- Pour afficher des fenêtres, Java s'appuie sur les fenêtres fournies par le système d'exploitation hôte dans lequel tourne la JVM
- AWT utilise les widgets du système d'exploitation pour tous les composants graphiques (fenêtres, boutons, listes, menus,...)
- Les composants Java JFrame, JWindow, JDialog et JApplet (les fenêtres de base « top-level ») s'appuient sur des composants du système hôte sont dit « lourds ». Les autres composants, dits légers, sont dessinés par Swing dans ces containers lourds
- L'utilisation de composants lourds améliore la rapidité d'exécution mais nuit à la portabilité, nécessite de grandes ressources matérielles et impose les fonctionnalités des composants
 - ➔ les composants lourds s'affichent toujours au-dessus des composants légers

Conteneurs (Container)

- Pour construire une interface graphique avec Swing, il faut créer un (ou plusieurs) container lourd et placer à l'intérieur les composants légers qui forment l'interface graphique
- Les conteneurs sont des objets graphiques qui sont destinés spécifiquement à recevoir d'autres éléments graphiques.
- Les conteneurs peuvent éventuellement contenir d'autres conteneurs.
- Ils héritent de la classe Container.
- Un composant graphique doit toujours être incorporer dans un conteneur.

Conteneurs (Container)

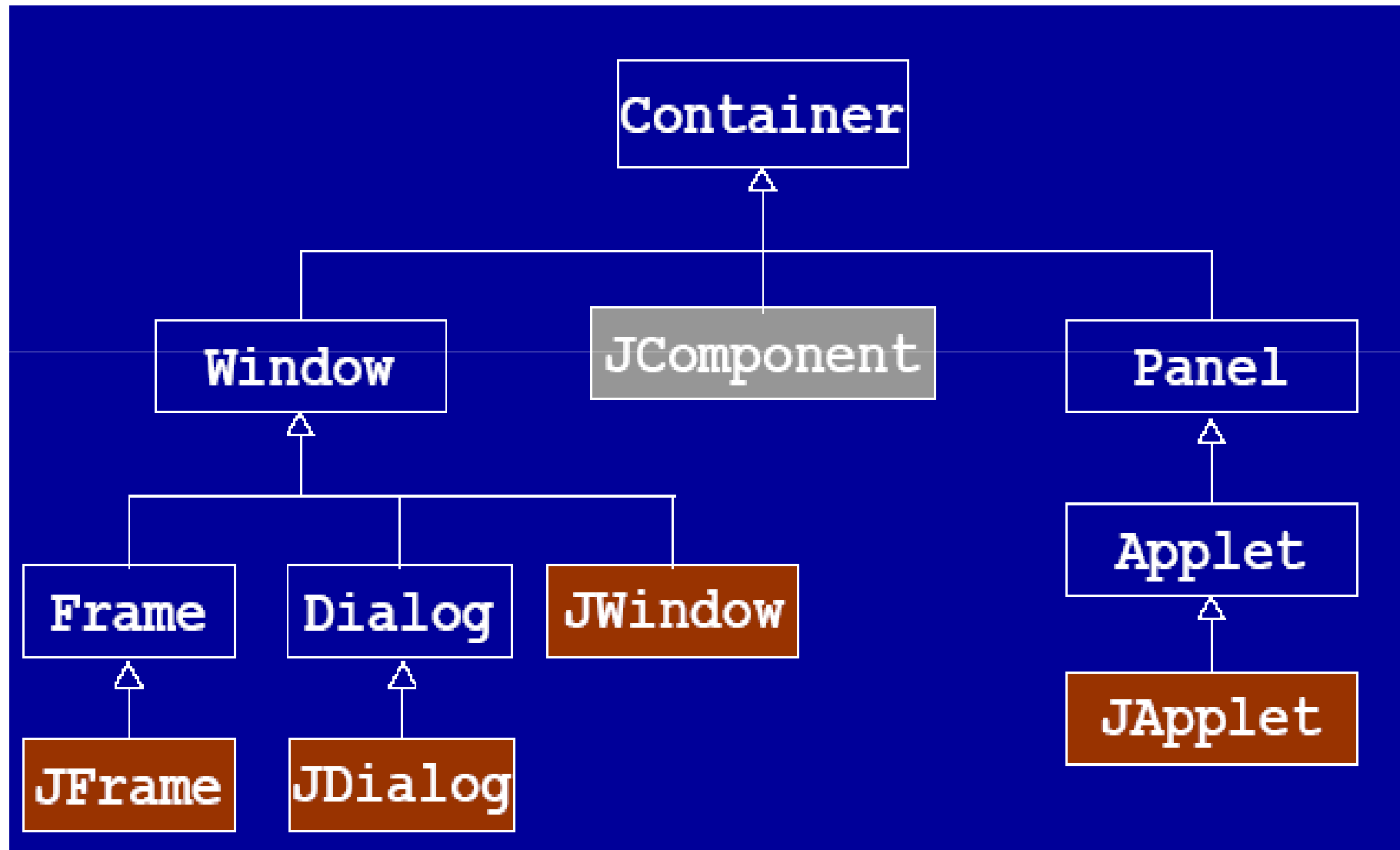
Il y a 4 sortes de containers lourds :

- JFrame fenêtre pour les applications
- JApplet pour les applets
- JDialog pour les fenêtres de dialogue
- JWindow, est plus rarement utilisé

Les containers « intermédiaires » légers :

- JPanel,
- JScrollPane,
- JSplitPane,
- JTabbedPane,
- Box (ce dernier est léger mais n'hérite pas de JComponent)

Hiérarchie d'héritage des containers lourds



Exemple de Container lourds: JFrame

- La Classe `javax.swing.JFrame` :
 - Elle représente une fenêtre principale qui possède un titre, une taille modifiable, éventuellement un menu, etc.
 - Elle utilise un conteneur ou panneau de contenu (content pane) pour insérer des composants (ils ne sont plus insérer directement au `JFrame` mais à l'objet `ContentPane` qui lui est associé).

Exemple de Container léger : JPanel

- **JPanel** est la classe mère des containers intermédiaires les plus simples.
- Un **JPanel** sert à regrouper des composants dans une zone d'écran.
- Il n'a pas d'aspect visuel déterminé ; son aspect visuel est donné par les composants qu'il contient.
- Il peut aussi servir de composant dans lequel on peut dessiner ce que l'on veut, ou faire afficher une image (par la méthode **paintComponent**).

Composant swing : JFrame

javax.swing

Class JFrame

```
java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Window
        java.awt.Frame
          javax.swing.JFrame
```

All Implemented Interfaces:

ImageObserver, MenuContainer, Serializable, Accessible, RootPaneContainer, WindowConstants

Constructor Summary

Constructors

Constructor and Description

JFrame()

Constructs a new frame that is initially invisible.

JFrame(GraphicsConfiguration gc)

Creates a **Frame** in the specified **GraphicsConfiguration** of a screen device and a blank title.

JFrame(String title)

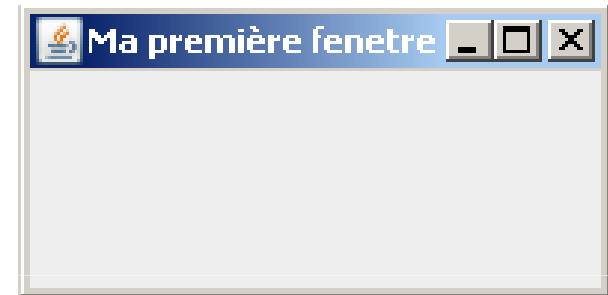
Creates a new, initially invisible **Frame** with the specified title.

JFrame(String title, GraphicsConfiguration gc)

Creates a **JFrame** with the specified title and the specified **GraphicsConfiguration** of a screen device.

Composant swing : JFrame

```
package graphique;
public class TestJFrame extends javax.swing.JFrame{
    public TestJFrame() {
        this("My First Frame");
    }
    public TestJFrame(String title) {
        super(title);
        this.setSize(200,100);
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
    }
    public static void main(String[] args) {
        TestJFrame testJFrame=new TestJFrame("Ma première fenetre");
        testJFrame.setVisible(true);
    }
}
```



Composant swing : JFrame

- Il est possible de préciser comment un objet JFrame, JInternalFrame, ou JDialog réagit à sa fermeture grâce à la méthode setDefaultCloseOperation().
- Cette méthode attend en paramètre une valeur qui peut être :

Constante	Rôle
WindowConstants.DISPOSE_ON_CLOSE	détruit la fenêtre
WindowConstants.DO_NOTHING_ON_CLOSE	rend le bouton de fermeture inactif
WindowConstants.HIDE_ON_CLOSE	cache la fenêtre
WindowConstants.EXIT_ON_CLOSE	Quitte le programme

Composant swing : JButton

javax.swing

Class JButton

[java.lang.Object](#)

└ [java.awt.Component](#)

└ [java.awt.Container](#)

└ [javax.swing.JComponent](#)

└ [javax.swing.AbstractButton](#)

└ **javax.swing.JButton**

All Implemented Interfaces:

[ImageObserver](#), [ItemSelectable](#), [MenuContainer](#), [Serializable](#), [Accessible](#), [SwingConstants](#)



Choisir votre photo

Composant swing : JButton

Constructor Summary

JButton()

Creates a button with no set text or icon.

JButton(Action a)

Creates a button where properties are taken from the Action supplied.

JButton(Icon icon)

Creates a button with an icon.

JButton(String text)

Creates a button with text.

JButton(String text, Icon icon)

Creates a button with initial text and an icon.

Composant swing : JLabel

javax.swing

Class JLabel

[java.lang.Object](#)

└ [java.awt.Component](#)

└ [java.awt.Container](#)

└ [javax.swing.JComponent](#)

└ **javax.swing.JLabel**

All Implemented Interfaces:

[ImageObserver](#), [MenuContainer](#), [Serializable](#), [Accessible](#), [SwingConstants](#)

	Nom الاسم
Password	

Date de naissance

Composant swing : JLabel

Constructor Summary

JLabel()

Creates a JLabel instance with no image and with an empty string for the title.

JLabel(Icon image)

Creates a JLabel instance with the specified image.

JLabel(Icon image, int horizontalAlignment)

Creates a JLabel instance with the specified image and horizontal alignment.

JLabel(String text)

Creates a JLabel instance with the specified text.

JLabel(String text, Icon icon, int horizontalAlignment)

Creates a JLabel instance with the specified text, image, and horizontal alignment.

JLabel(String text, int horizontalAlignment)

Creates a JLabel instance with the specified text and horizontal alignment.

Composant swing : JLabel

Parmi les valeurs que peut prendre `horizontalAlignment`, on peut citer :

- `SwingConstants.CENTER`
- `SwingConstants.LEFT`
- `SwingConstants.RIGHT`

String	<u>getText()</u> Returns the text string that the label displays.
void	<u>setHorizontalAlignment(int alignment)</u> Sets the alignment of the label's contents along the X axis.
void	<u>setText(String text)</u> Defines the single line of text this component will display.

Composant swing : JLabel

Remarque 1 : par défaut un JLabel n'est pas opaque. Donc, il faut le rendre opaque avant si on veut modifier la couleur de fond

```
javax.swing.JLabel jLabelNom = new javax.swing.JLabel("Nom");  
jLabelNom.setOpaque(true) ;  
jLabelNom.setBackground(java.awt.Color.BLUE) ;
```

Remarque 2 : pour avoir une étiquette dont le libellé est sur 2 lignes, ou une étiquette dont le libellé est à moitié en rouge et l'autre moitié en vert, on utilise du HTML.

```
new javax.swing.JLabel("<html>nom <br>de la personne</html>");
```

Remarque 3 : JLabel implémente l'interface SwingConstants. Donc on aurait pu écrire JLabel.CENTER

Composant swing : JTextField

javax.swing

Class JTextField

[java.lang.Object](#)

└ [java.awt.Component](#)

└ [java.awt.Container](#)

└ [javax.swing.JComponent](#)

└ [javax.swing.text.JTextComponent](#)

└ **javax.swing.JTextField**

All Implemented Interfaces:

[ImageObserver](#), [MenuContainer](#), [Serializable](#), [Accessible](#), [Scrollable](#), [SwingConstants](#)

Direct Known Subclasses:

[DefaultTreeCellEditor.DefaultTextField](#), [JFormattedTextField](#), [JPasswordField](#)



Tapper votre nom

Composant swing : JTextField

Constructor Summary

JTextField()

Constructs a new TextField.

JTextField(Document doc, String text, int columns)

Constructs a new JTextField that uses the given text storage model and the given number of columns.

JTextField(int columns)

Constructs a new empty TextField with the specified number of columns.

JTextField(String text)

Constructs a new TextField initialized with the specified text.

JTextField(String text, int columns)

Constructs a new TextField initialized with the specified text and columns.

Composant swing : JTextField

Method Summary

void [setColumns](#)(int columns)

Sets the number of columns in this TextField, and then invalidate the layout.

void [setFont](#)([Font](#) f)

Sets the current font.

[String](#) [getSelectedText](#)()

Returns the selected text contained in this TextComponent.

[Color](#) [getSelectedTextColor](#)()

Fetches the current color used to render the selected text.

[Color](#) [getSelectionColor](#)()

Fetches the current color used to render the selection.

[String](#) [getText](#)()

Returns the text contained in this TextComponent.

void [setEditable](#)(boolean b)

Sets the specified boolean to indicate whether or not this TextComponent should be editable.

void [setSelectedTextColor](#)([Color](#) c)

Sets the current color used to render the selected text.

void [setText](#)([String](#) t)

Sets the text of this TextComponent to the specified text.

Composant swing : JTextArea

La classe *JTextArea* est un composant qui permet la saisie de texte simple en mode multiligne.

`javax.swing`

Class JTextArea

```
java.lang.Object
└─ java.awt.Component
    └─ java.awt.Container
        └─ javax.swing.JComponent
            └─ javax.swing.text.JTextComponent
                └─ javax.swing.JTextArea
```

All Implemented Interfaces:

[ImageObserver](#), [MenuContainer](#), [Serializable](#), [Accessible](#), [Scrollable](#)

Composant swing : JTextArea

Constructor Summary

JTextArea()

Constructs a new TextArea.

JTextArea(Document doc)

Constructs a new JTextArea with the given document model, and defaults for all of the other arguments (null, 0, 0).

JTextArea(Document doc, String text, int rows, int columns)

Constructs a new JTextArea with the specified number of rows and columns, and the given model.

JTextArea(int rows, int columns)

Constructs a new empty TextArea with the specified number of rows and columns.

JTextArea(String text)

Constructs a new TextArea with the specified text displayed.

JTextArea(String text, int rows, int columns)

Constructs a new TextArea with the specified text and number of rows and columns.

Composant swing : JTextArea

Method Summary

void [append](#)([String](#) str)

Appends the given text to the end of the document.

int [getLineCount](#)()

Determines the number of lines contained in the area.

void [setColumns](#)(int columns)

Sets the number of columns for this TextArea.

void [setFont](#)([Font](#) f)

Sets the current font.

void [setRows](#)(int rows)

Sets the number of rows for this TextArea.

Composant swing : JCheckBox.

javax.swing

Class JCheckBox

[java.lang.Object](#)

└ [java.awt.Component](#)

└ [java.awt.Container](#)

└ [javax.swing.JComponent](#)

└ [javax.swing.AbstractButton](#)

└ [javax.swing.JToggleButton](#)

└ **javax.swing.JCheckBox**

All Implemented Interfaces:

[ImageObserver](#), [ItemSelectable](#), [MenuContainer](#), [Serializable](#), [Accessible](#), [SwingConstants](#)

☒ Java ☐ Foot ☐ Music ☐ Cuisine ☐ Voyage

Composant swing : JCheckBox.

Constructor Summary

JCheckBox()

Creates an initially unselected check box button with no text, no icon.

JCheckBox(Action a)

Creates a check box where properties are taken from the Action supplied.

JCheckBox(Icon icon)

Creates an initially unselected check box with an icon.

JCheckBox(Icon icon, boolean selected)

Creates a check box with an icon and specifies whether or not it is initially selected.

JCheckBox(String text)

Creates an initially unselected check box with text.

JCheckBox(String text, boolean selected)

Creates a check box with text and specifies whether or not it is initially selected.

JCheckBox(String text, Icon icon)

Creates an initially unselected check box with the specified text and icon.

JCheckBox(String text, Icon icon, boolean selected)

Creates a check box with text and icon, and specifies whether or not it is initially selected.

Composant swing : JCheckBox.

Method Summary

boolean isSelected()

Returns the state of the button.

void setSelected(boolean b)

Sets the state of the button.

Composant swing : JRadioButton

javax.swing

Class JRadioButton

[java.lang.Object](#)

└ [java.awt.Component](#)

└ [java.awt.Container](#)

└ [javax.swing.JComponent](#)

└ [javax.swing.AbstractButton](#)

└ [javax.swing.JToggleButton](#)

└ **javax.swing.JRadioButton**

All Implemented Interfaces:

[ImageObserver](#), [ItemSelectable](#), [MenuContainer](#), [Serializable](#), [Accessible](#), [SwingConstants](#)



Composant swing : JRadioButton

Constructor Summary

JRadioButton()

Creates an initially unselected radio button with no set text.

JRadioButton(Action a)

Creates a radiobutton where properties are taken from the Action supplied.

JRadioButton(Icon icon)

Creates an initially unselected radio button with the specified image but no text.

JRadioButton(Icon icon, boolean selected)

Creates a radio button with the specified image and selection state, but no text.

JRadioButton(String text)

Creates an unselected radio button with the specified text.

JRadioButton(String text, boolean selected)

Creates a radio button with the specified text and selection state.

JRadioButton(String text, Icon icon)

Creates a radio button that has the specified text and image, and that is initially unselected.

JRadioButton(String text, Icon icon, boolean selected)

Creates a radio button that has the specified text, image, and selection state.

Composant swing : JRadioButton

Remarques :

Si on ajoute plusieurs boutons radios à une fenêtre, ils se comporteront comme s'ils étaient des cases à cocher, c.-à-d. qu'une sélection d'un bouton radio n'entraînera pas la désélection d'un autre bouton radio mais les deux restent sélectionnés. **Or, ce n'est pas le comportement voulu. Pour réaliser le comportement voulu, on doit grouper tous les boutons radios concernés en utilisant la classe *javax.swing.ButtonGroup*.**

Composant swing : JComboBox

javax.swing

Class JComboBox

[java.lang.Object](#)

└ [java.awt.Component](#)

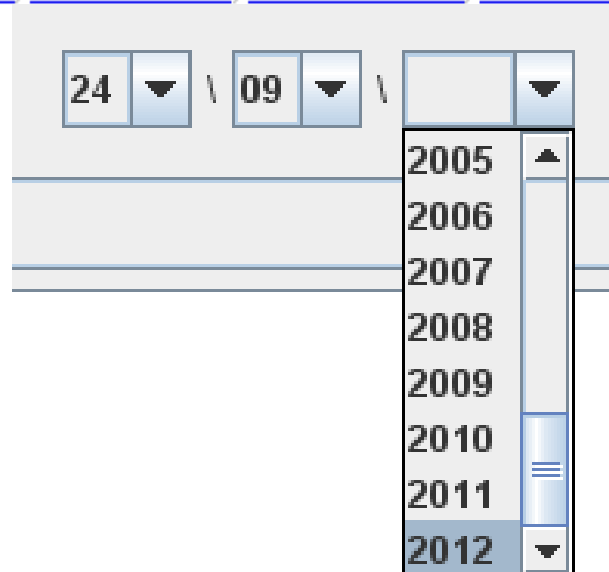
└ [java.awt.Container](#)

└ [javax.swing.JComponent](#)

└ [javax.swing.JComboBox](#)

All Implemented Interfaces:

[ActionListener](#), [ImageObserver](#), [ItemSelectable](#), [MenuContainer](#), [Serializable](#), [EventListener](#), [Accessible](#), [ListDataListener](#)



Composant swing : JComboBox

Constructor Summary

JComboBox()

Creates a JComboBox with a default data model.

JComboBox(ComboBoxModel aModel)

Creates a JComboBox that takes its items from an existing ComboBoxModel.

JComboBox(Object[] items)

Creates a JComboBox that contains the elements in the specified array.

JComboBox(Vector<?> items)

Creates a JComboBox that contains the elements in the specified Vector.

Composant swing : JComboBox

Method Summary

void [addItem](#)([Object](#) anObject)

Adds an item to the item list.

[Object](#) [getItemAt](#)(int index)

Returns the list item at the specified index.

int [getItemCount](#)()

Returns the number of items in the list.

int [getSelectedIndex](#)()

Returns the first item in the list that matches the given item.

[Object](#) [getSelectedItem](#)()

Returns the current selected item.

void [removeItem](#)([Object](#) anObject)

Removes an item from the item list.

void [removeItemAt](#)(int anIndex)

Removes the item at an Index This method works only if the JComboBox uses a mutable data model.

Composant swing : JList

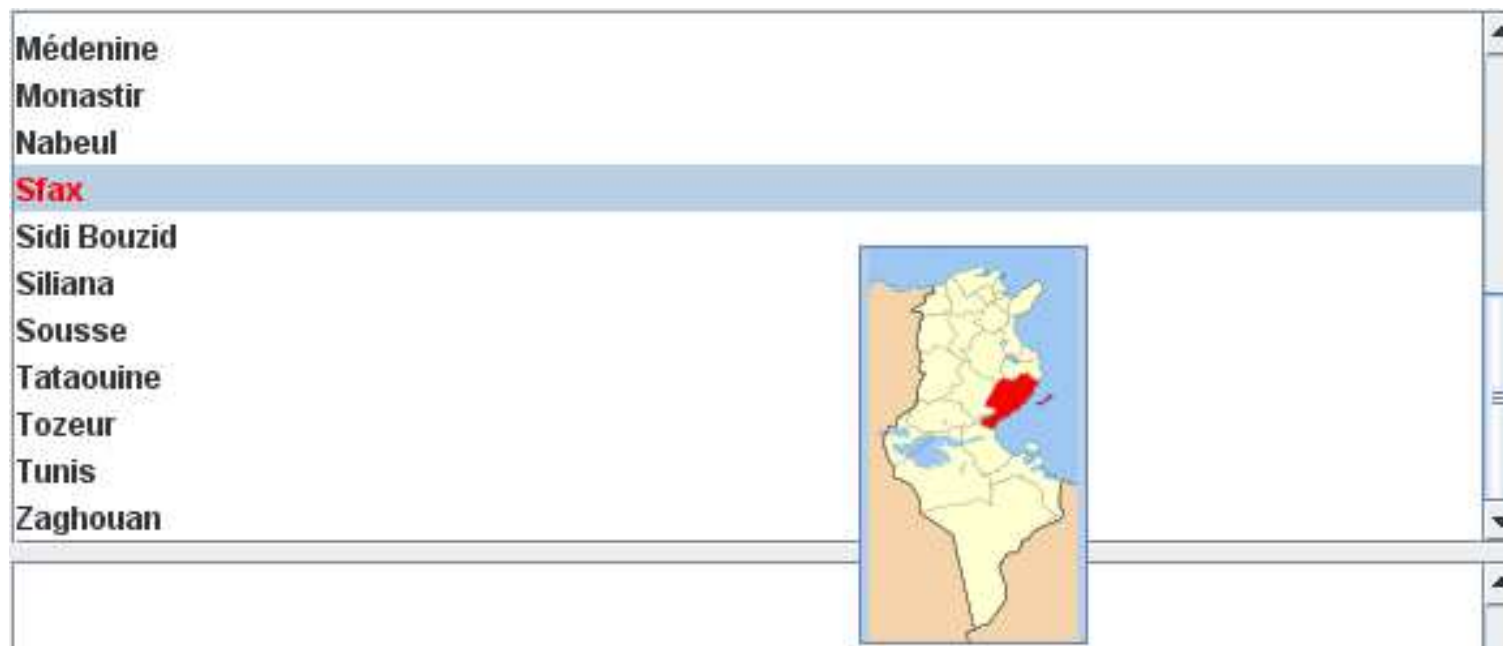
javax.swing

Class JList

```
java.lang.Object
├── java.awt.Component
│   ├── java.awt.Container
│   │   ├── javax.swing.JComponent
│   │   └── javax.swing.JList
```

All Implemented Interfaces:

[ImageObserver](#), [MenuContainer](#), [Serializable](#), [Accessible](#), [Scrollable](#)



Composant swing : JList

Constructor Summary

JList()

Constructs a JList with an empty, read-only, model.

JList(ListModel dataModel)

Constructs a JList that displays elements from the specified, non-null, model.

JList(Object[] listData)

Constructs a JList that displays the elements in the specified array.

JList(Vector<?> listData)

Constructs a JList that displays the elements in the specified Vector.

Composant swing : JList

Method Summary

int [getSelectedIndex\(\)](#)

Returns the smallest selected cell index; *the selection* when only a single item is selected in the list.

int[] [getSelectedIndices\(\)](#)

Returns an array of all of the selected indices, in increasing order.

[Object](#) [getSelectedValue\(\)](#)

Returns the value for the smallest selected cell index; *the selected value* when only a single item is selected in the list.

[Object](#)[] [getSelectedValues\(\)](#)

Returns an array of all the selected values, in increasing order based on their indices in the list.

boolean [isSelectedIndex\(int index\)](#)

Returns true if the specified index is selected, else false.

void [setModel\(ListModel model\)](#)

Sets the model that represents the contents or "value" of the list, notifies property change listeners, and then clears the list's selection.

void [setSelectedValue\(Object anObject, boolean shouldScroll\)](#)

Selects the specified object from the list.

void [setVisibleRowCount\(int visibleRowCount\)](#)

Sets the visibleRowCount property, which has different meanings depending on the layout orientation: For a VERTICAL layout orientation, this sets the preferred number of rows to display without requiring scrolling; for other orientations, it affects the wrapping of cells.

Composant swing : JList

package graphique;

public class TestJList **extends** javax.swing.JFrame{

public TestJList() {

super("Test liste");

this.setSize(200,200);

this.setLocationRelativeTo(**null**);

this.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);

this.getContentPane().setLayout(**new** java.awt.FlowLayout());

 String[]lettre={"A","B","C","D","E","F","G","H","I","J","K"};

 javax.swing.JList jList=**new** javax.swing.JList(lettre);

this.getContentPane().add(jList);

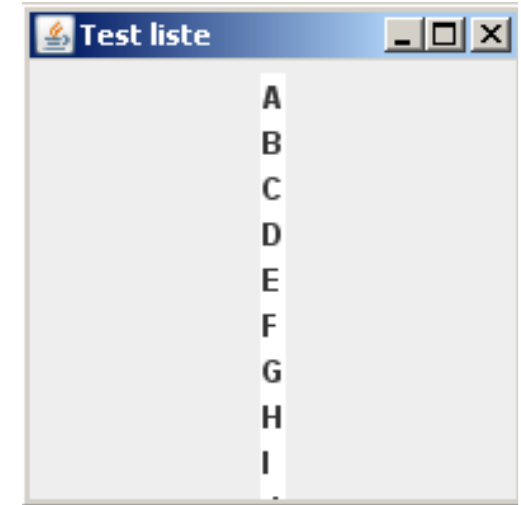
 }

public static void main(String[] args) {

new TestJList().setVisible(**true**);

 }

}

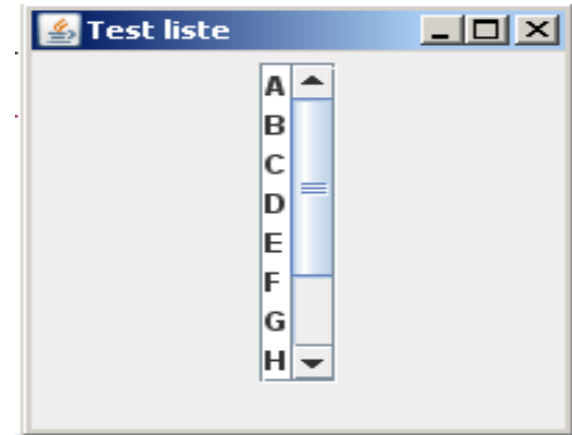


Composant swing : JList & JScrollPane

```
package graphique;
public class TestJList extends javax.swing.JFrame{
    public TestJList() {
        super("Test liste");
        this.setSize(200,200);
        this.setLocationRelativeTo(null);

        this.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
        this.getContentPane().setLayout(new java.awt.FlowLayout());
        String[] lettre={"A","B","C","D","E","F","G","H","I","J","K"};
        javax.swing.JList jList=new javax.swing.JList(lettre);
        javax.swing.JScrollPane jScrollPane=new javax.swing.JScrollPane(jList);
        this.getContentPane().add(jScrollPane);

    }
    public static void main(String[] args) {
        new TestJList().setVisible(true);
    }
}
```



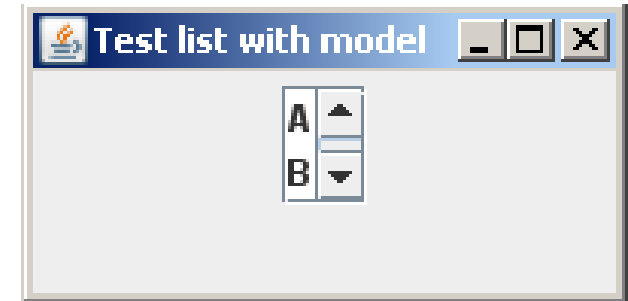
Composant swing : JList with DefaultListModel

```
package graphique;
public class TestJListWithModel extends javax.swing.JFrame{
    public TestJListWithModel() {
        super("Test list with model");
        this.setSize(200,100);
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
        this.getContentPane().setLayout(new java.awt.FlowLayout());

        javax.swing.DefaultListModel defaultListModel=new javax.swing.DefaultListModel();
        defaultListModel.addElement("A");
        defaultListModel.addElement("B");
        defaultListModel.addElement("C");

        javax.swing.JList jList=new javax.swing.JList(defaultListModel);
        jList.setVisibleRowCount(2);

        javax.swing.JScrollPane jScrollPane=new javax.swing.JScrollPane(jList);
        this.getContentPane().add(jScrollPane);
    }
}
```



Composant swing : JList

void setSelectionMode(int selectionMode)
Sets the selection mode for the list.

Si selectionMode est égal à :

- **ListSelectionModel.SINGLE_SELECTION**, alors on ne peut sélectionner qu'une seule valeur à la fois
- **ListSelectionModel.SINGLE_INTERVAL_SELECTION**, alors on peut sélectionner une suite de valeurs successives
- **ListSelectionModel.MULTIPLE_INTERVAL_SELECTION**, alors on peut sélectionner plusieurs valeurs à la fois n'importe où dans la liste (c'est celle par défaut)

Composant swing : JDialog

javax.swing

Class JDialog

[java.lang.Object](#)

└ [java.awt.Component](#)

└ [java.awt.Container](#)

└ [java.awt.Window](#)

└ [java.awt.Dialog](#)

└ **javax.swing.JDialog**

All Implemented Interfaces:

[ImageObserver](#), [MenuContainer](#), [Serializable](#), [Accessible](#), [RootPaneContainer](#), [WindowConstants](#)

Composant swing : JDialog

Constructor Summary

JDialog()

Creates a modeless dialog without a title and without a specified Frame owner.

JDialog(Dialog owner)

Creates a modeless dialog without a title with the specified Dialog as its owner.

JDialog(Dialog owner, boolean modal)

Creates a dialog with the specified owner Dialog and modality.

JDialog(Dialog owner, String title)

Creates a modeless dialog with the specified title and with the specified owner dialog.

JDialog(Dialog owner, String title, boolean modal)

Creates a dialog with the specified title, modality and the specified owner Dialog.

JDialog(Dialog owner, String title, boolean modal, GraphicsConfiguration gc)

Creates a dialog with the specified title, owner Dialog, modality and GraphicsConfiguration.

JDialog(Frame owner)

Creates a modeless dialog without a title with the specified Frame as its owner.

JDialog(Frame owner, boolean modal)

Creates a dialog with the specified owner Frame, modality and an empty title.

JDialog(Frame owner, String title)

Creates a modeless dialog with the specified title and with the specified owner frame.

Composant swing : JOptionPane

javax.swing

Class JOptionPane

[java.lang.Object](#)

└ [java.awt.Component](#)

└ [java.awt.Container](#)

└ [javax.swing.JComponent](#)

└ **javax.swing.JOptionPane**

All Implemented Interfaces:

[ImageObserver](#), [MenuContainer](#), [Serializable](#), [Accessible](#)

Composant swing : JOptionPane





C'est une classe de Java qui permet de créer facilement des boîtes de dialogue. L'utilisation typique de cette classe consiste à des appels à une des méthodes statiques de la forme `showXxxDialog` qui suivent :

Méthodes	Description
<code>showConfirmDialog</code>	Demande une question qui se répond par oui/non/annulé
<code>showInputDialog</code>	Demande de taper une réponse
<code>showMessageDialog</code>	Transcrit un message à l'usage
<code>showOptionDialog</code>	Une méthode générale réunissant les trois précédente

Composant swing : JOptionPane

```
javax.swing.JOptionPane.showMessageDialog(null, "Message", "Titre", messageType);
```

Le type de message est l'une des valeurs suivantes :

- *JOptionPane.PLAIN_MESSAGE*;
- *JOptionPane.ERROR_MESSAGE*; 
- *JOptionPane.INFORMATION_MESSAGE*; 
- *JOptionPane.WARNING_MESSAGE*; 
- *JOptionPane.QUESTION_MESSAGE*; 

```
javax.swing.JOptionPane.showConfirmDialog(null, "Message", "Title", messageType);
```

L'option de message est l'une des valeurs suivantes :

- *JOptionPane.DEFAULT_OPTION*
- *JOptionPane.YES_NO_OPTION*
- *JOptionPane.YES_NO_CANCEL_OPTION*
- *JOptionPane.OK_CANCEL_OPTION*

Composant swing : JTabbedPane

javax.swing

Class JTabbedPane

```
java.lang.Object
├── java.awt.Component
│   ├── java.awt.Container
│       ├── javax.swing.JComponent
│           └── javax.swing.JTabbedPane
```

All Implemented Interfaces:

[ImageObserver](#), [MenuContainer](#), [Serializable](#), [Accessible](#), [SwingConstants](#)

Composant swing : JTabbedPane

Constructor Summary

JTabbedPane()

Creates an empty TabbedPane with a default tab placement of JTabbedPane.TOP.

JTabbedPane(int tabPlacement)

Creates an empty TabbedPane with the specified tab placement of either: JTabbedPane.TOP, JTabbedPane.BOTTOM, JTabbedPane.LEFT, or JTabbedPane.RIGHT.

JTabbedPane(int tabPlacement, int tabLayoutPolicy)

Creates an empty TabbedPane with the specified tab placement and tab layout policy.

Composant swing : JTabbedPane

Method Summary

void [add](#)([Component](#) component, [Object](#) constraints)

Adds a component to the tabbed pane.

void [add](#)([Component](#) component, [Object](#) constraints, int index)

Adds a component at the specified tab index.

[Component](#) [add](#)([String](#) title, [Component](#) component)

Adds a component with the specified tab title.

void [addTab](#)([String](#) title, [Component](#) component)

Adds a component represented by a title and no icon.

void [addTab](#)([String](#) title, [Icon](#) icon, [Component](#) component)

Adds a component represented by a title and/or icon, either of which can be null.

void [addTab](#)([String](#) title, [Icon](#) icon, [Component](#) component, [String](#) tip)

Adds a component and tip represented by a title and/or icon, either of which can be null.

[Component](#) [getComponentAt](#)(int index)

Returns the component at index.

int [getTabCount](#)()

Returns the number of tabs in this tabbedpane.

[String](#) [getTitleAt](#)(int index)

Returns the tab title at index.

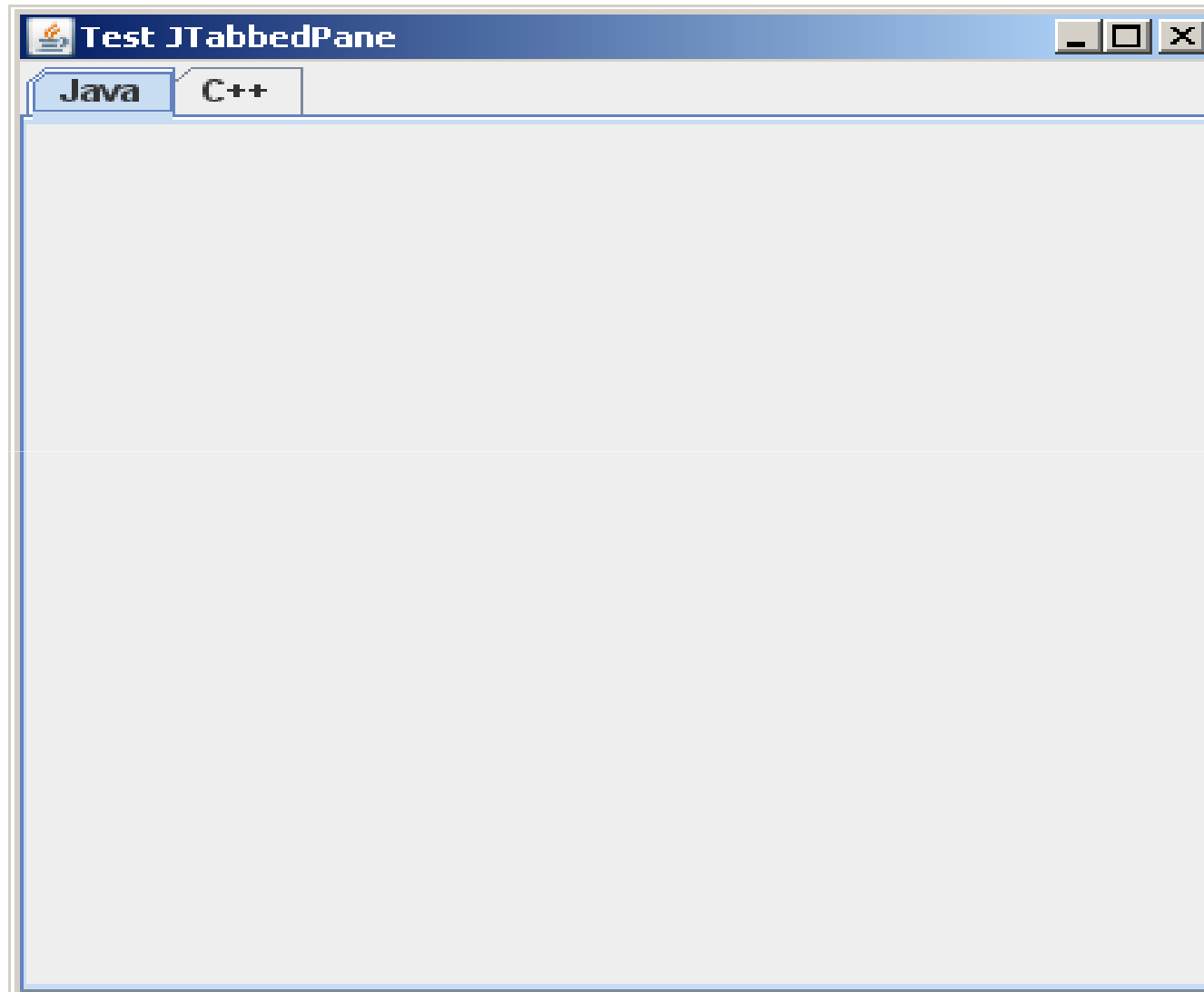
int [indexOfTab](#)([String](#) title)

Returns the first tab index with a given title, or -1 if no tab has this title.

Composant swing : JTabbedPane

```
package graphique;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTabbedPane;
import javax.swing.WindowConstants;
public class TestJTabbedPane extends JFrame{
    private JTabbedPane jTabbedPane;
    public TestJTabbedPane() {
        super("Test JTabbedPane");
        jTabbedPane = new JTabbedPane();
        jTabbedPane.addTab("Java", new JPanel());
        jTabbedPane.addTab("C++", new JPanel());
        this.getContentPane().add(jTabbedPane);
        this.setSize(400,400);
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    }
}
```

Composant swing : JTabbedPane



Composant swing : JSplitPane

javax.swing

Class JSplitPane

[java.lang.Object](#)

└ [java.awt.Component](#)

└ [java.awt.Container](#)

└ [javax.swing.JComponent](#)

└ **javax.swing.JSplitPane**

All Implemented Interfaces:

[ImageObserver](#), [MenuContainer](#), [Serializable](#), [Accessible](#)

Composant swing : JSplitPane

Constructor Summary

JSplitPane()

Creates a new JSplitPane configured to arrange the child components side-by-side horizontally with no continuous layout, using two buttons for the components.

JSplitPane(int newOrientation)

Creates a new JSplitPane configured with the specified orientation and no continuous layout.

JSplitPane(int newOrientation, boolean newContinuousLayout)

Creates a new JSplitPane with the specified orientation and redrawing style.

JSplitPane(int newOrientation, boolean newContinuousLayout, Component newLeftComponent, Component newRightComponent)

Creates a new JSplitPane with the specified orientation and redrawing style, and with the specified components.

JSplitPane(int newOrientation, Component newLeftComponent, Component newRightComponent)

Creates a new JSplitPane with the specified orientation and with the specified components that do not do continuous redrawing.

Composant swing : JSplitPane

Method Summary

int [getOrientation\(\)](#)

Returns the orientation.

void [setLeftComponent\(Component comp\)](#)

Sets the component to the left (or above) the divider.

void [setOneTouchExpandable\(boolean newValue\)](#)

Sets the value of the oneTouchExpandable property, which must be true for the JSplitPane to provide a UI widget on the divider to quickly expand/collapse the divider.

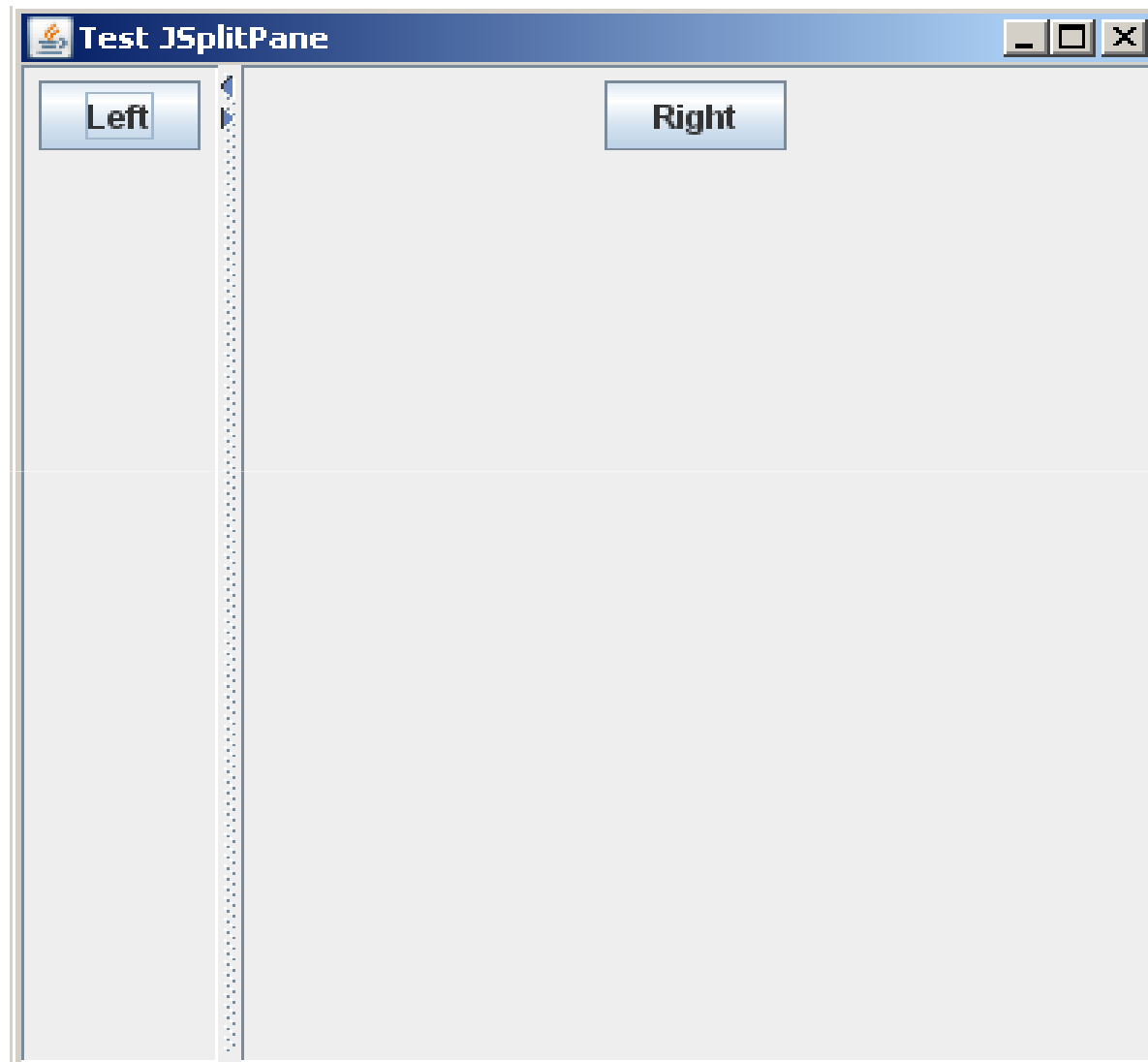
void [setRightComponent\(Component comp\)](#)

Sets the component to the right (or below) the divider.

Composant swing : JSplitPane

```
package graphique;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSplitPane;
public class TestJSplitPane extends JFrame{
    private JSplitPane jSplitPane;
    public TestJSplitPane() {
        super("Test JSplitPane");
        this.setSize(400,400);
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel left=new JPanel();
        left.add(new JButton("Left"));
        JPanel right=new JPanel();
        right.add(new JButton("Right"));
        jSplitPane = new JSplitPane();
        jSplitPane.setOneTouchExpandable(true);
        jSplitPane.setLeftComponent(left);
        jSplitPane.setRightComponent(right);
        this.add(jSplitPane);
    }
}
```

Composant swing : JSplitPane



Composant swing : JMenuBar

javax.swing

Class JMenuBar

[java.lang.Object](#)

└ [java.awt.Component](#)

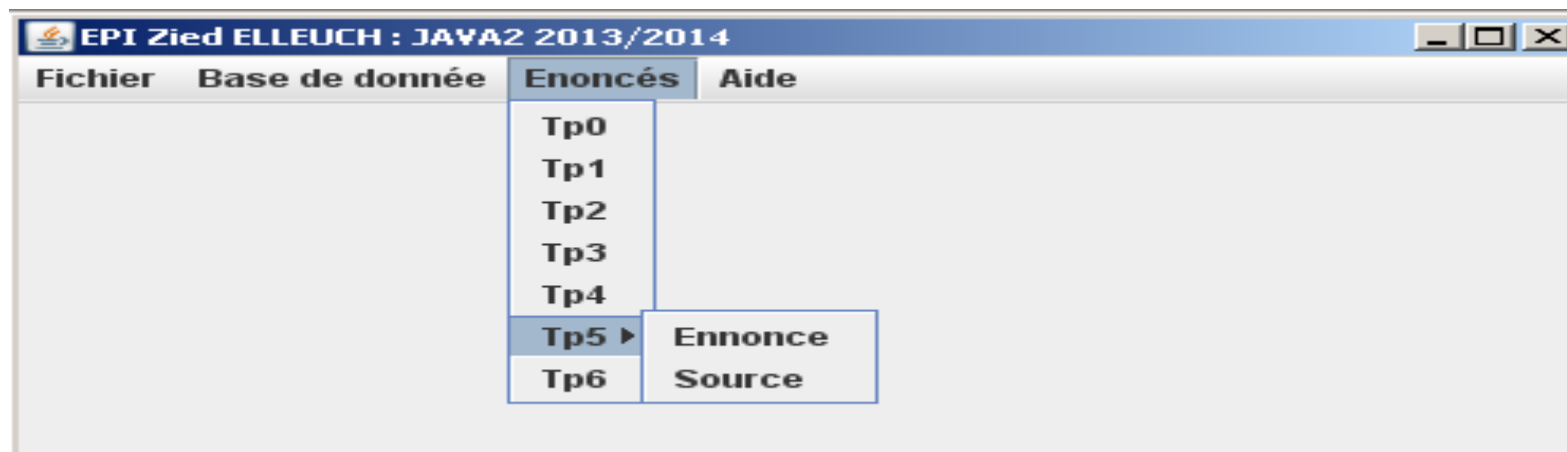
└ [java.awt.Container](#)

└ [javax.swing.JComponent](#)

└ [javax.swing.JMenuBar](#)

All Implemented Interfaces:

[ImageObserver](#), [MenuContainer](#), [Serializable](#), [Accessible](#), [MenuItem](#)



Composant swing : JMenuBar

Constructor Summary

JMenuBar()

Creates a new menu bar.

Method Summary

JMenu**add**(JMenu c)

Appends the specified menu to the end of the menu bar.

Component**getComponentAtIndex**(int i)

Deprecated. *replaced by `getComponent(int i)`*

boolean isBorderPainted()

Returns true if the menu bars border should be painted.

boolean isSelected()

Returns true if the menu bar currently has a component selected.

void setBorderPainted(boolean b)

Sets whether the border should be painted.

Composant swing : JMenu

javax.swing

Class JMenu

[java.lang.Object](#)

└ [java.awt.Component](#)

└ [java.awt.Container](#)

└ [javax.swing.JComponent](#)

└ [javax.swing.AbstractButton](#)

└ [javax.swing.JMenuItem](#)

└ **javax.swing.JMenu**

All Implemented Interfaces:

[ImageObserver](#), [ItemSelectable](#), [MenuContainer](#), [Serializable](#), [Accessible](#), [MenuElement](#), [SwingConstants](#)

Composant swing : JMenu

Constructor Summary

JMenu()

Constructs a new JMenu with no text.

JMenu(Action a)

Constructs a menu whose properties are taken from the Action supplied.

JMenu(String s)

Constructs a new JMenu with the supplied string as its text.

JMenu(String s, boolean b)

Constructs a new JMenu with the supplied string as its text and specified as a tear-off menu or not.

Composant swing : JMenu

Method Summary

JMenuItem add(JMenuItem menuItem)

Appends a menu item to the end of this menu.

void addSeparator()

Appends a new separator to the end of the menu.

int getItemCount()

Returns the number of items on the menu, including separators.

void setAccelerator(KeyStroke keyStroke)

setAccelerator is not defined for JMenu.

Composant swing : JMenuItem

javax.swing

Class JMenuItem

[java.lang.Object](#)

└ [java.awt.Component](#)

└ [java.awt.Container](#)

└ [javax.swing.JComponent](#)

└ [javax.swing.AbstractButton](#)

└ **`javax.swing.JMenuItem`**

All Implemented Interfaces:

[ImageObserver](#), [ItemSelectable](#), [MenuContainer](#), [Serializable](#), [Accessible](#), [MenuElement](#), [SwingConstants](#)

Direct Known Subclasses:

[JCheckBoxMenuItem](#), [JMenu](#), [JRadioButtonMenuItem](#)

Composant swing : JMenuItem

Constructor Summary

JMenuItem()

Creates a JMenuItem with no set text or icon.

JMenuItem(Action a)

Creates a menu item whose properties are taken from the specified Action.

JMenuItem(Icon icon)

Creates a JMenuItem with the specified icon.

JMenuItem(String text)

Creates a JMenuItem with the specified text.

JMenuItem(String text, Icon icon)

Creates a JMenuItem with the specified text and icon.

JMenuItem(String text, int mnemonic)

Creates a JMenuItem with the specified text and keyboard mnemonic.

Composant swing : JMenuItem

Method Summary

KeyStroke **getAccelerator**()

Returns the KeyStroke which serves as an accelerator for the menu item.

void **setAccelerator**(KeyStroke keyStroke)

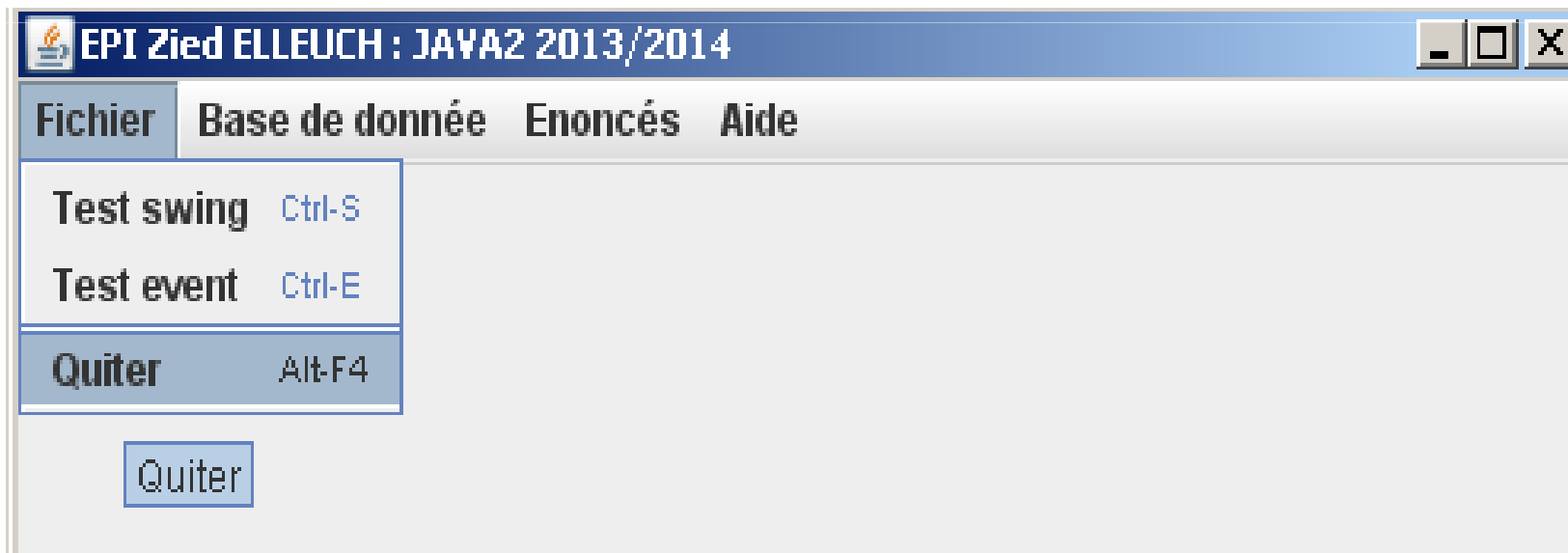
Sets the key combination which invokes the menu item's action listeners without navigating the menu hierarchy.

void **setEnabled**(boolean b)

Enables or disables the menu item.

Composant swing : JMenuItem

```
menuFileQuiter=new JMenuItem(quiter);  
menuFileQuiter.setToolTipText("Quiter");  
menuFileQuiter.setHorizontalTextPosition(JButton.RIGHT);  
menuFileQuiter.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_F4,KeyEvent.ALT_MASK));
```



Composant swing : JSlider

javax.swing

Class JSlider

[java.lang.Object](#)

└ [java.awt.Component](#)

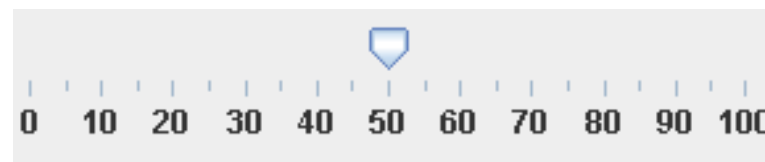
└ [java.awt.Container](#)

└ [javax.swing.JComponent](#)

└ **javax.swing.JSlider**

All Implemented Interfaces:

[ImageObserver](#), [MenuContainer](#), [Serializable](#), [Accessible](#), [SwingConstants](#)



Composant swing : JSlider

Constructor Summary

JSlider()

Creates a horizontal slider with the range 0 to 100 and an initial value of 50.

JSlider(int orientation)

Creates a slider using the specified orientation with the range 0 to 100 and an initial value of 50.

JSlider(int min, int max)

Creates a horizontal slider using the specified min and max with an initial value equal to the average of the min plus max.

JSlider(int min, int max, int value)

Creates a horizontal slider using the specified min, max and value.

JSlider(int orientation, int min, int max, int value)

Creates a slider with the specified orientation and the specified minimum, maximum, and initial values.

Composant swing : JSlider

Method Summary

int [getMaximum\(\)](#)

Returns the maximum value supported by the slider from the BoundedRangeModel.

int [getMinimum\(\)](#)

Returns the minimum value supported by the slider from the BoundedRangeModel.

int [getValue\(\)](#)

Returns the slider's current value from the BoundedRangeModel.

void [setOrientation\(\)](#)(int orientation)

Set the slider's orientation to either SwingConstants.VERTICAL or SwingConstants.HORIZONTAL.

void [setValue\(\)](#)(int n)

Sets the slider's current value to n.

Composant swing : Taille

Tous les composants graphiques (classe **Component**) peuvent indiquer leurs tailles pour l'affichage

taille maximum

taille préférée

taille minimum

Accesseurs et modificateurs associés :

{get|set} {Maximum|Preferred|Minimum}Size

Composant swing : Taille

Taille préférée

La taille préférée est la plus utilisée par les *layout managers* ; un composant peut l'indiquer en redéfinissant la méthode « **Dimension getPreferredSize()** »

On peut aussi l'imposer « de l'extérieur » avec la méthode « **void setPreferredSize(Dimension)** »

Mais le plus souvent, les gestionnaires de mise en place ne tiendront pas compte des tailles imposées de l'extérieur

Taille minimum

Quand un composant n'a plus la place pour être affiché à sa taille préférée, il est affiché à sa taille minimum sans passer par des tailles intermédiaires.

Si la taille minimum est très petite, ça n'est pas du plus bel effet ; il est alors conseillé de fixer une taille minimum, par exemple par **c.setMinimumSize(c.getPreferredSize());**

- Introduction
- Les composants SWING
- **Gestionnaire de disposition**
- Gestion des événements
- Accès à la base de données
- Le graphisme en JAVA.
- Les applets
- Thread
- Les collections et type générique

Gestionnaire de disposition

La disposition des composants dans les conteneurs est gérée par les *LayoutManager*. Il existe plusieurs types de *LayoutManager* avec des algorithmes de placement différents.

L'utilisation des gestionnaires de disposition a les principaux avantages suivants :

- En cas de redimensionnement de la fenêtre, l'aménagement des composants graphiques est délégué aux layout manager (il est inutile d'utiliser les coordonnées absolues). Ainsi, les composants sont automatiquement agrandis ou réduits.
- Il est inutile de préciser les coordonnées de chaque emplacement. Le gestionnaire s'en charge selon la taille de la fenêtre
- ils permettent une indépendance vis à vis des plateformes

Gestionnaire de disposition

Quand on ajoute un composant dans un container on ne donne pas la position exacte du composant. On donne plutôt des indications de positionnement au gestionnaire de mise en place.

- ✓ explicites (**BorderLayout.NORTH**)
- ✓ ou implicites (ordre d'ajout dans le container)

Un *layout manager* place les composants « au mieux » suivant :

- l'algorithme de placement qui lui est propre
- les indications de positionnement des composants
- la taille du container
- les tailles préférées des composants

Il existe plusieurs gestionnaires dans java à savoir **FlowLayout**, **BorderLayout**, **GridLayout**, **BoxLayout**, **GridBagLayout**, **GroupLayout**, **CardLayout**.

Gestionnaire de disposition : java.awt.FlowLayout

Le gestionnaire *FlowLayout* (mise en page flot) place les composants les uns à la suite des autres, sur la même ligne. Chaque ligne est complétée progressivement jusqu'à être remplie, puis passe à la suivante. Toutefois on peut changer le type d'alignement en utilisant les constantes suivantes :

- FlowLayout.LEFT : alignement à gauche
- FlowLayout.RIGHT : alignement à droite
- FlowLayout.CENTER : alignement au centre

JPanel admet par défaut comme gestionnaire de disposition FlowLayout avec un alignement aux centre.

Gestionnaire de disposition : java.awt.FlowLayout

Constructor Summary

FlowLayout()

Constructs a new FlowLayout with a centered alignment and a default 5-unit horizontal and vertical gap.

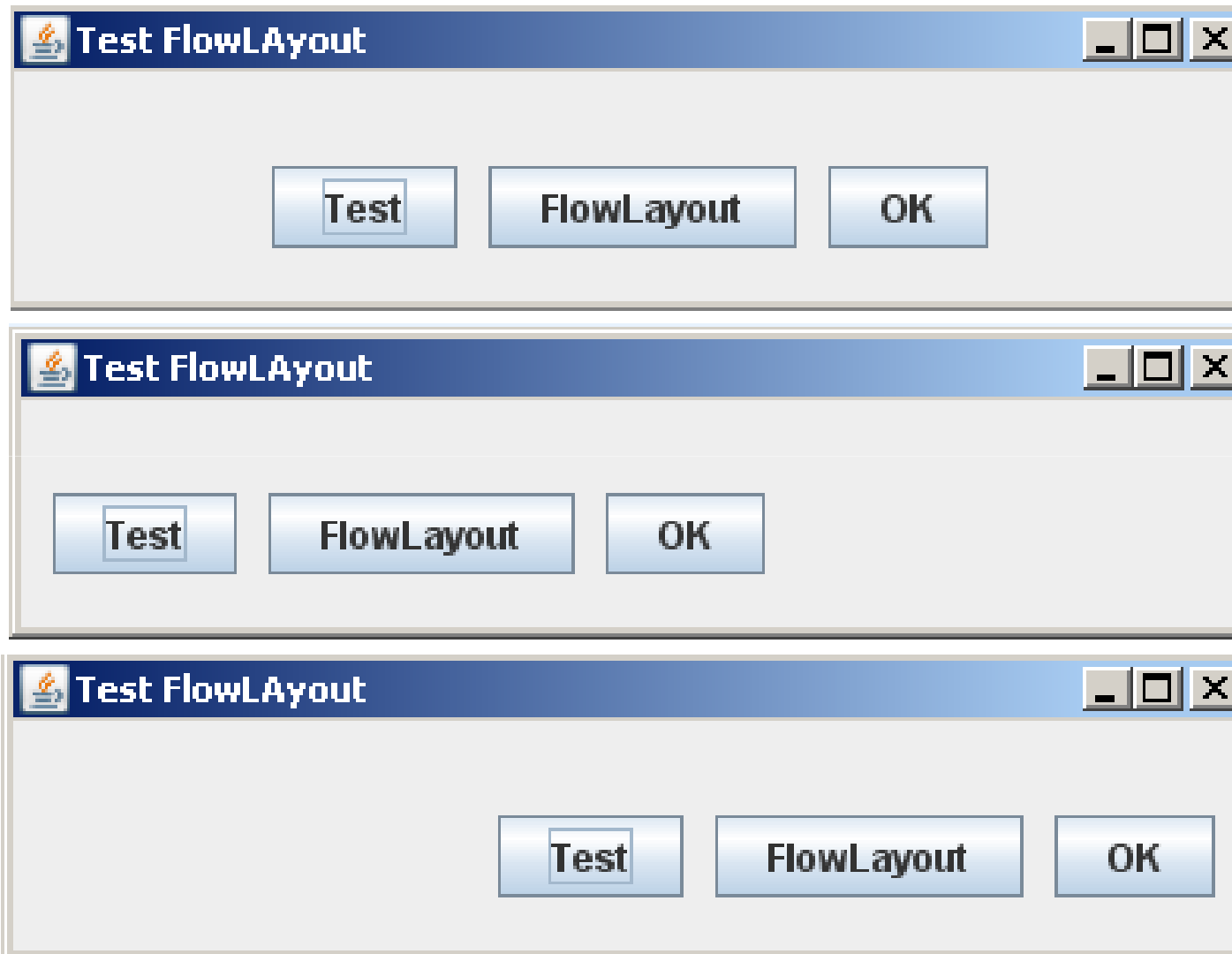
FlowLayout(int align)

Constructs a new FlowLayout with the specified alignment and a default 5-unit horizontal and vertical gap.

FlowLayout(int align, int hgap, int vgap)

Creates a new flow layout manager with the indicated alignment and the indicated horizontal and vertical gaps.

Gestionnaire de disposition : java.awt.FlowLayout



Par défaut, les composants sont espacés de 5 pixels

Gestionnaire de disposition : `java.awt.BorderLayout`

Le gestionnaire `BorderLayout` place les composants :

- soit sur un des bords:
 - Nord (`BorderLayout.NORTH`)
 - Sud (`BorderLayout.SOUTH`)
 - Est (`BorderLayout.EAST`)
 - Ouest (`BorderLayout.WEST`)
- soit au centre :
 - Centre (`BorderLayout.CENTER`)

On peut librement utiliser une ou plusieurs zones

Lorsque le composant est placé sur les bords, il a une épaisseur fixe. Lorsqu'il est placé au centre, il occupe toute la place qui reste

Gestionnaire de disposition : BorderLayout

Constructor Summary

BorderLayout()

Constructs a new border layout with no gaps between components.

BorderLayout(int hgap, int vgap)

Constructs a border layout with the specified gaps between components.

Gestionnaire de disposition : BorderLayout

JFrame admet par défaut comme gestionnaire de disposition BorderLayout.

On peut modifier le layout de n'importe quel container (JPanel, JFrame, JApplet, Container, etc. en invoquant la méthode `setLayout`.

```
jPanel.setLayout(new FlowLayout(FlowLayout.CENTER,10,30));
```

```
jFrame.setLayout(new FlowLayout(FlowLayout.LEFT,10,30));
```

```
jPanel.setLayout(new BorderLayout());
```

Pour choisir l'emplacement d'un composant, il faut fournir un 2^{ème} argument à la méthode `add` qui représente une des constantes suivantes :

BorderLayout.NORTH

BorderLayout.SOUTH,

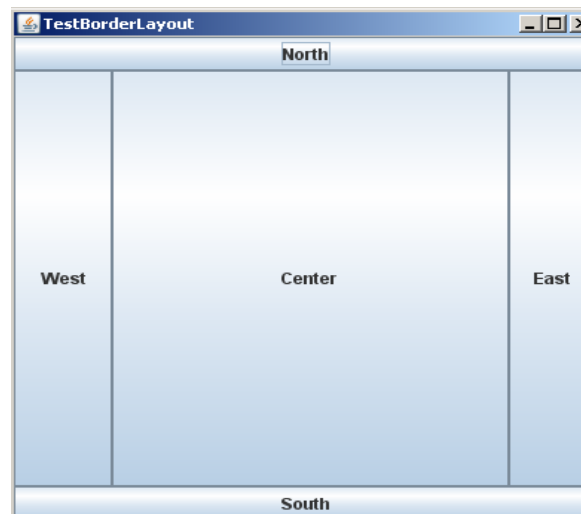
BorderLayout.EAST

BorderLayout.WEST

BorderLayout.CENTER.

Gestionnaire de disposition : BorderLayout

```
public class TestBorderLayout extends JFrame{  
    public TestBorderLayout() {  
        this.setTitle("TestBorderLayout");  
        this.setSize(400,400);  
        this.setLocationRelativeTo(null);  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        Container container = this.getContentPane();  
        container.add(new JButton("Center"));  
        container.add(new JButton("North"), "North");  
        container.add(new JButton("South"),BorderLayout.SOUTH);  
        container.add(new JButton("East"),BorderLayout.EAST);  
        container.add(new JButton("West"),BorderLayout.WEST);  
    }  
}
```



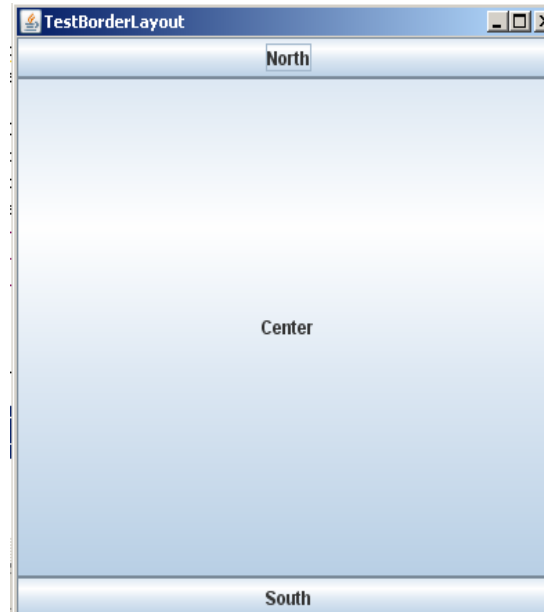
Gestionnaire de disposition : BorderLayout

```
public class TestBorderLayout extends JFrame{  
    public TestBorderLayout() {  
        this.setTitle("TestBorderLayout");  
        this.setSize(400,400);  
        this.setLocationRelativeTo(null);  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        Container container = this.getContentPane();  
        container.setLayout(new BorderLayout(20,50));  
        container.add(new JButton("Center"));  
        container.add(new JButton("North"), "North");  
        container.add(new JButton("South"), BorderLayout.SOUTH);  
        container.add(new JButton("East"), BorderLayout.EAST);  
        container.add(new JButton("West"), BorderLayout.WEST);  
    }  
}
```



Gestionnaire de disposition : BorderLayout

```
public class TestBorderLayout extends JFrame{  
    public TestBorderLayout() {  
        this.setTitle("TestBorderLayout");  
        this.setSize(400,400);  
        this.setLocationRelativeTo(null);  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        Container container = this.getContentPane();  
        container.add(new JButton("Center"));  
        container.add(new JButton("North"), "North");  
        container.add(new JButton("South"), BorderLayout.SOUTH);  
    }  
}
```



Gestionnaire de disposition : java.awt.GridLayout

Ce Layout Manager établit un réseau de cellule identiques qui forment une sorte de quadrillage invisible : les composants sont organisés en lignes et en colonnes. Les éléments insérés dans la grille ont tous la même taille. Les cellules du quadrillage se remplissent de droite à gauche ou de haut en bas.

Par défaut, les composants sont espacés de 5 pixels.

Constructor Summary

GridLayout()

Creates a grid layout with a default of one column per component, in a single row.

GridLayout(int rows, int cols)

Creates a grid layout with the specified number of rows and columns.

GridLayout(int rows, int cols, int hgap, int vgap)

Creates a grid layout with the specified number of rows and columns.

Gestionnaire de disposition : GridLayout

```
public class TestGridLayout extends JFrame {  
    private JPanel jPanel;  
    public TestGridLayout() {  
        this.setTitle("TestBorderLayout");  
        this.setSize(400,400);  
        this.setLocationRelativeTo(null);  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        jPanel = new JPanel();  
        jPanel.setLayout(new GridLayout(5, 3));  
        for(int i=0;i<15;i++)  
        {  
            jPanel.add(new JButton(""+(i+1)));  
        }  
        this.add(jPanel);  
    }  
}
```



Gestionnaire de disposition : GridLayout

```
public class TestGridLayout extends JFrame {  
    private JPanel jPanel;  
    public TestGridLayout() {  
        this.setTitle("TestBorderLayout");  
        this.setSize(400,400);  
        this.setLocationRelativeTo(null);  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        jPanel = new JPanel();  
        jPanel.setLayout(new GridLayout(5, 3,20,10));  
        for(int i=0;i<15;i++)  
        {  
            jPanel.add(new JButton(""+(i+1)));  
        }  
        this.add(jPanel);  
    }  
}
```



Gestionnaire de disposition : GridLayout

```
public class TestGridLayout extends JFrame {  
    private JPanel jPanel;  
    public TestGridLayout() {  
        this.setTitle("TestBorderLayout");  
        this.setSize(400,400);  
        this.setLocationRelativeTo(null);  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        jPanel = new JPanel();  
        jPanel.setLayout(new GridLayout(5, 3,20,10));  
        for(int i=0;i<13;i++)  
        {  
            jPanel.add(new JButton(""+(i+1)));  
        }  
        this.add(jPanel);  
    }  
}
```



Gestionnaire de disposition : javax.swing.BoxLayout

Le gestionnaire BoxLayout dispose des composants suivant une seule ligne ou une seule colonne. Il doit être associé à un conteneur de type Box. Un box a par défaut un gestionnaire par défaut qui est BoxLayout.

On peut créer un Box horizontal ou un Box vertical.

```
Box ligne = Box.createHorizontalBox () ; // box horizontal
```

```
Box col = Box.createVerticalBox () ; // box vertical
```

Pour un box horizontal : Les composants sont mis de gauche à droite sur une même ligne. Si on rétrécit la largeur de la fenêtre, certains composants qui sont à gauche peuvent ne plus apparaître.

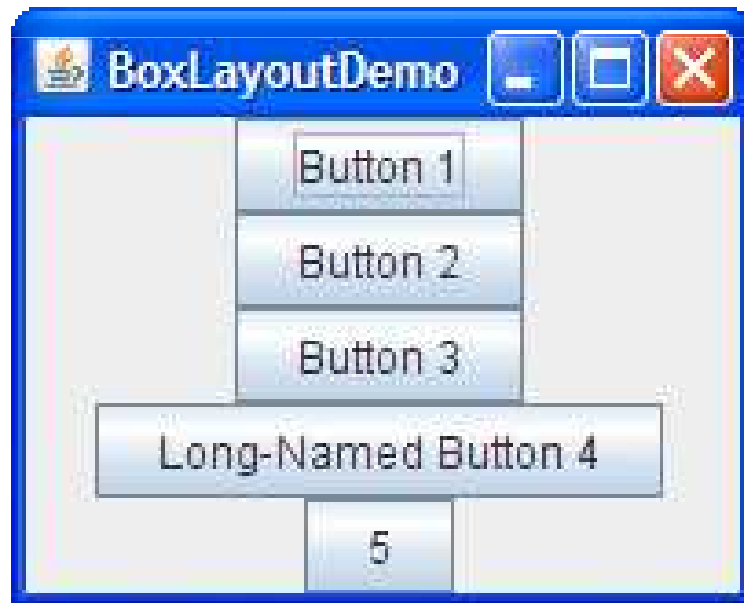
Pour un box vertical : Les composants sont mis de haut en bas sur une même colonne. Si on rétrécit la longueur de la fenêtre, certains composants qui sont en bas peuvent ne plus apparaître⁹⁷

Gestionnaire de disposition : BoxLayout

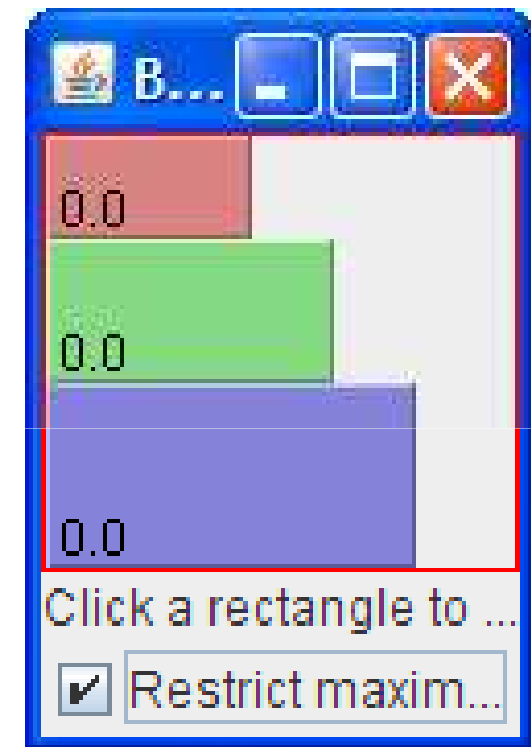
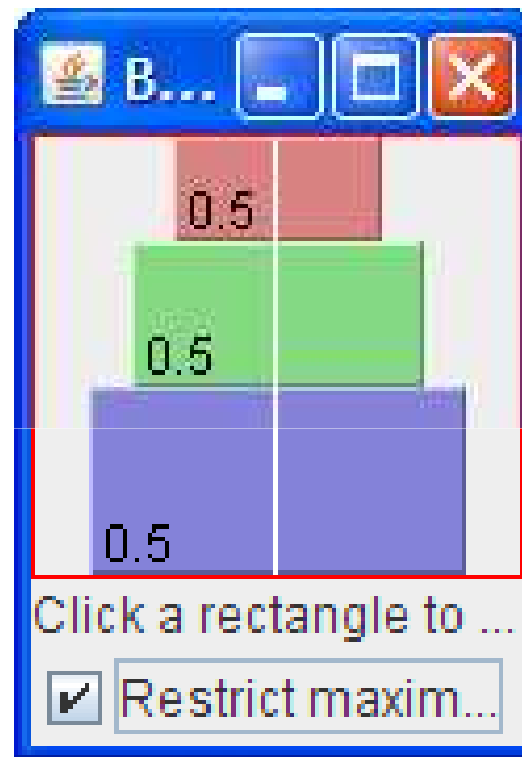
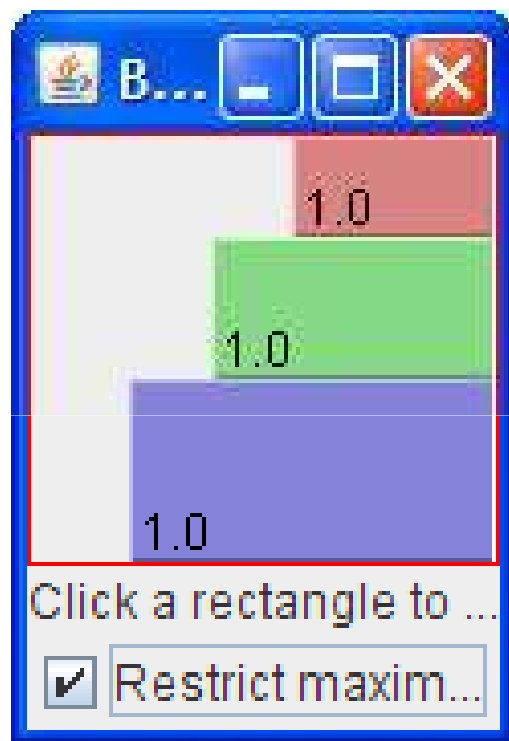
Constructor Summary

BoxLayout(Container target, int axis)

Creates a layout manager that will lay out components along the given axis.

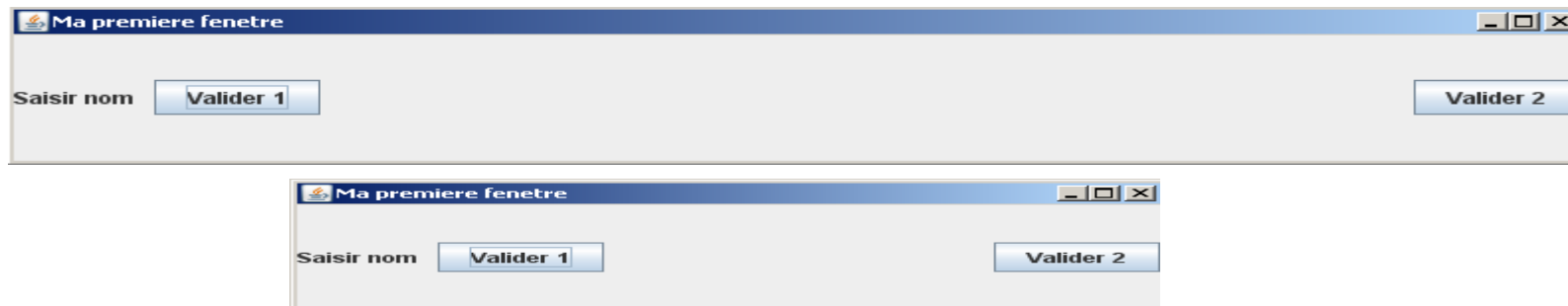


Gestionnaire de disposition : BoxLayout



Gestionnaire de disposition : BorderLayout

```
Box box = Box.createHorizontalBox() ;
container.add(box) ;
jLabel = new JLabel ("Saisir nom") ;
box.add(jLabel);
box.add(Box.createRigidArea(new Dimension(10,0)));
//crée un composant virtuel représentant un espace vide et fixe de 10 px
jButtonValider = new JButton ("Valider 1") ;
box.add(jButtonValider);
box.add(Box.createGlue());
/* crée un emplacement virtuel de taille initialisée de façon à espacer au maximum les
composants situés de part et d'autre du glue. Cette taille augmente (ou diminue) lorsqu'on on
agrandit la feuille (ou on la rétrécit).*/
jButtonValider2 = new JButton ("Valider 2") ;
box.add(jButtonValider2);
```



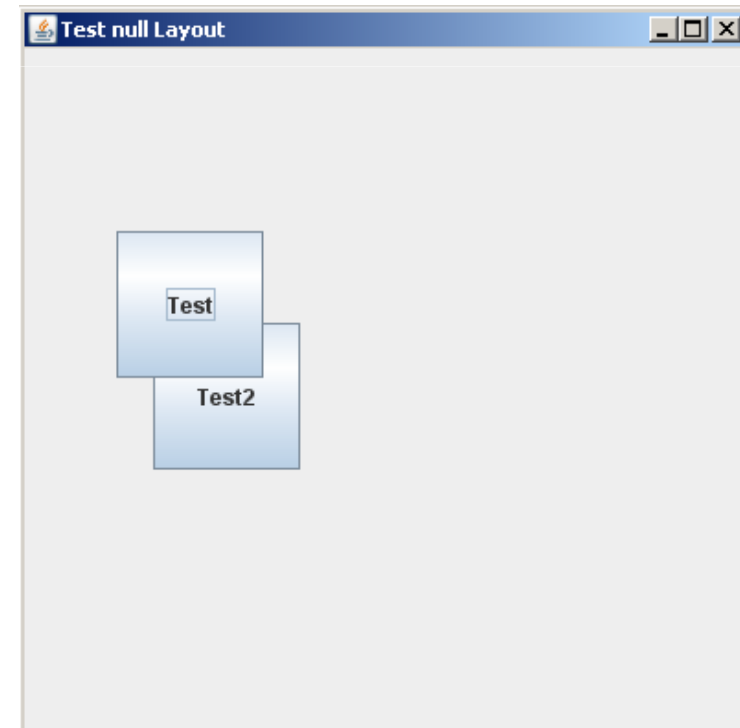
Gestionnaire de disposition : null

On peut fixer les coordonnées dans l'écran de chaque composant lorsque on l'ajoute à la fenêtre. Dans ce cas, notre classe doit annoncer explicitement qu'elle n'utilisera pas de gestionnaire de disposition par l'instruction : `setLayout(null);`

A chaque ajout d'un composant il faut préciser sa taille et son emplacement en utilisant *setSize()*, *et setLocation()* ou tout simplement *setBounds()*

Gestionnaire de disposition : null

```
public class TestLayout extends JFrame{  
    public TestLayout() {  
        this.setTitle("Test null Layout");  
        this.setSize(400,400);  
        this.setLocationRelativeTo(null);  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        this.setLayout(null);  
        JButton jButton=new JButton("Test");  
        jButton.setBounds(50,100, 80, 80);  
        JButton jButton2=new JButton("Test2");  
        jButton2.setBounds(70,150, 80, 80);  
        this.add(jButton);  
        this.add(jButton2);  
    }  
}
```



- Introduction
- Les composants SWING
- Gestionnaire de disposition
- **Gestion des événements**
- Accès à la base de données
- Le graphisme en JAVA.
- Les applets
- Thread
- Les collections et type générique

Gestion des événements

La programmation d'interfaces graphiques recourt à la notion d'événement pour représenter les interactions entre l'interface et l'utilisateur.

La gestion d'un événement met en jeu un objet **source** et un objet **écouteur**. L'objet qui a déclenché un événement est appelé **source**. Pour l'événement clic sur une fenêtre par exemple, la source est la fenêtre.

Les événements sont des objets d'une classe dérivée de **java.util.EventObject**. La sous-classe **java.awt.AWTEvent** est celle des événements créés par des interactions avec les composantes d'interface : bouton pressé, clic souris, changement de taille des fenêtres, touches du clavier pressées, etc.

Gestion des événements

Quelques unes de ses sous-classes sont : `ActionEvent` , `ItemEvent` , `MouseEvent` , `MouseMotionEvent` , `TextEvent` , `KeyEvent` , `FocusEvent` , `WindowEvent`

Pour traiter un événement, on associe à la source un **écouteur**. Cet écouteur doit être un objet dont la classe **implémente** une **interface** particulière correspondant à une catégorie d'événements. Il existe ainsi une interface correspondant à la catégorie d'événement mouse (événements souris), une interface correspondant à la catégorie d'événement key (événements clavier), une interface correspondant à la catégorie d'événement action (événements lié aux boutons et à d'autres composants), etc.

Gestion des événements

Par exemple, l'interface correspondant la catégorie d'événement souris s'appelle **MouseListener** et elle déclare 5 méthodes correspondant chacun à un événement de cette catégorie :

- `public void mouseClicked(MouseEvent e);`
- `public void mouseEntered(MouseEvent e);`
- `public void mouseExited(MouseEvent e);`
- `public void mousePressed(MouseEvent e);`
- `public void mouseReleased(MouseEvent e);`

Pour gérer les événements, on peut utiliser 5 démarches différentes selon nos besoins.

Gestion des événements : 1^{ère} démarche

Supposons qu'on veut gérer l'événement clic de souris sur la fenêtre.

1^{ère} démarche: créer une classe qui implémente l'interface `MouseListener` et associer un objet de cette classe à la fenêtre. Cet objet représente l'écouteur de la fenêtre.

Pour cela, on suivra les étapes suivantes :

Étape 1 : créer une classe qui implémente l'interface **`MouseListener`**.

Les objets instanciés à partir de cette classe joueront le rôle d'écouteur. On appellera cette classe par exemple

`MyJFrameMouseListener`

Étape 2 : associer un écouteur de type ***`MyJFrameMouseListener`*** à la fenêtre sur laquelle on va cliquer. Ceci se fait grâce à la méthode `addMouseListener`.

Gestion des événements : 1^{ère} démarche

```
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import javax.swing.JOptionPane;
public class MyJFrameMouseListener implements MouseListener {
    public void mouseClicked(MouseEvent e) {
        int x = e.getX();
        int y = e.getY();
        int xOnScreen = e.getXOnScreen();
        int yOnScreen = e.getYOnScreen();
        JOptionPane.showMessageDialog(null, "(x,y)=( " + x + ", " + y
            + ")\n(xOnScreen,yOnScreen)=( " + xOnScreen + ", " +
yOnScreen
            + ")", "Location", JOptionPane.PLAIN_MESSAGE);
    }
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
}
```

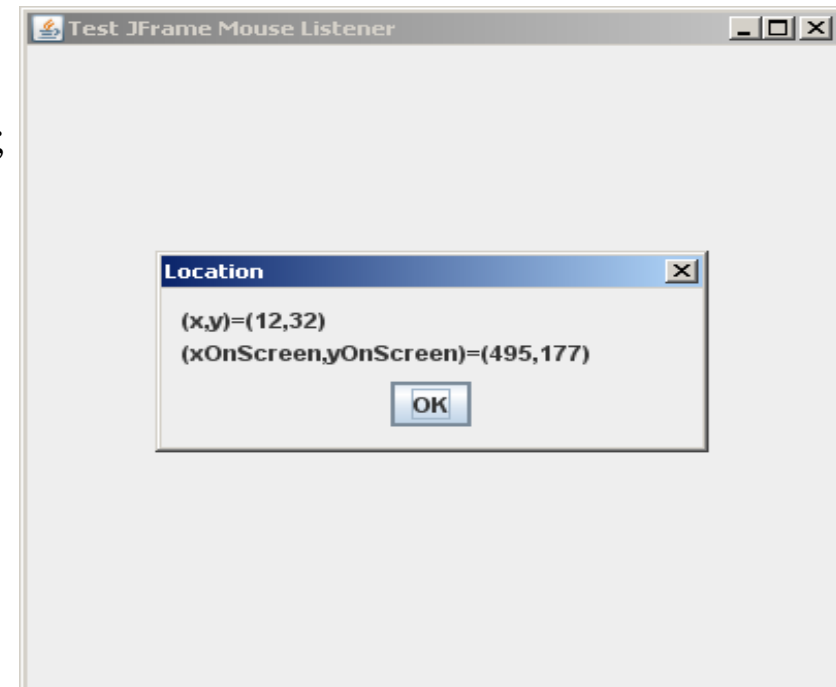
Étape 1

Même si on a besoin que de la méthode `mouseClicked`, on doit implémenter toutes les méthodes de l'interface `MouseListener`.

Gestion des événements : 1^{ère} démarche

```
package evenement;
import javax.swing.JFrame;
public class TestMouseListener extends JFrame {
    public TestMouseListener() {
        super("Test JFrame Mouse Listener ");
        this.setSize(400, 400);
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.addMouseListener(new MyJFrameMouseListener());
    }
    public static void main(String[] args) {
        new TestMouseListener().setVisible(true);
    }
}
```

Étape 2



Gestion des événements : 2^{ème} démarche

2^{ème} démarche : la fenêtre peut être elle-même son propre écouteur. Dans ce cas la fenêtre implémentera l'interface `MouseListener`, puis on associe l'objet courant (la fenêtre courante) à la fenêtre pour la désigner comme son propre écouteur.

Pour cela, on suivra les étapes suivantes :

Étape 1 : changer la classe qui représente la fenêtre pour qu'elle implémente l'interface `MouseListener`

Étape 2 : considérer la fenêtre comme son propre écouteur

Gestion des événements : 2^{ème} démarche

```
public class TestMouseListener2 extends JFrame implements MouseListener {  
    public TestMouseListener2() {  
        super("Test JFrame Mouse Listener");  
        this.setSize(400, 400);  
        this.setLocationRelativeTo(null);  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    }  
    public void mouseClicked(MouseEvent e) {  
        int x = e.getX(); int xOnScreen = e.getXOnScreen();  
        int y = e.getY(); int yOnScreen = e.getYOnScreen();  
        JOptionPane.showMessageDialog(null, "(x,y)=( " + x + ", " + y  
            + ")\n(xOnScreen,yOnScreen)=( " + xOnScreen + ", " +  
yOnScreen  
            + ")", "Location", JOptionPane.PLAIN_MESSAGE);  
    }  
    public void mouseEntered(MouseEvent e) {}  
    public void mouseExited(MouseEvent e) {}  
    public void mousePressed(MouseEvent e) {}  
    public void mouseReleased(MouseEvent e) {}  
}
```

Étape 1

Gestion des événements : 2^{ème} démarche

```
public class TestMouseListener2 extends JFrame implements MouseListener {  
    public TestMouseListener2() {  
        super("Test JFrame Mouse Listener");  
        this.setSize(400, 400);  
        this.setLocationRelativeTo(null);  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        this.addMouseListener(this);  
    }  
    public void mouseClicked(MouseEvent e) {  
        int x = e.getX(); int xOnScreen = e.getXOnScreen();  
        int y = e.getY(); int yOnScreen = e.getYOnScreen();  
        JOptionPane.showMessageDialog(null, "(x,y)=( " + x + ", " + y  
            + ")\n(xOnScreen,yOnScreen)=( " + xOnScreen + ", " +  
yOnScreen  
            + ")", "Location", JOptionPane.PLAIN_MESSAGE);  
    }  
    public void mouseEntered(MouseEvent e) {}  
    public void mouseExited(MouseEvent e) {}  
    public void mousePressed(MouseEvent e) {}  
    public void mouseReleased(MouseEvent e) {}  
}
```

Étape 2

Gestion des événements : 3^{ème} démarche

3^{ème} démarche : utiliser une classe prédéfinie appelée **MouseAdapter** qui implémente déjà l'**interface MouseListener**. Cette classe contient toutes les méthodes de l'interface avec une implémentation vide ({}). Il suffit donc de créer une classe écouteur qui **hérite** de **MouseAdapter** et qui permet de redéfinir seulement la méthode qu'on va utiliser : **MouseClicked**

Pour cela, on suivra les étapes suivantes :

Étape 1 : créer une classe qui hérite de **MouseAdapter**

Étape 2 : associer un écouteur de type **MyJFrameMouseListener2** à la fenêtre sur laquelle on va cliquer. Ceci se fait grâce à la méthode **addMouseListener**

Gestion des événements : 3^{ème} démarche

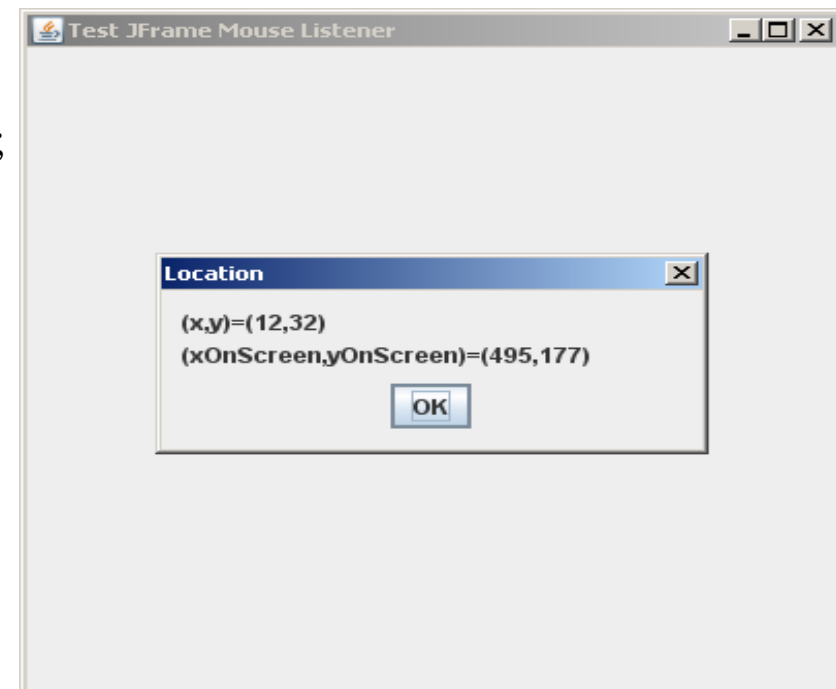
Étape 1

```
package evenement;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.JOptionPane;
public class MyJFrameMouseListener2 extends MouseAdapter {
    public void mouseClicked(MouseEvent e) {
        int x = e.getX();
        int y = e.getY();
        int xOnScreen = e.getXOnScreen();
        int yOnScreen = e.getYOnScreen();
        JOptionPane.showMessageDialog(null, "(x,y)=( " + x + ", " + y
            + ")\n(xOnScreen,yOnScreen)=( " + xOnScreen +
            ", " + yOnScreen
            + ")", "Location",
            JOptionPane.PLAIN_MESSAGE);
    }
}
```

Gestion des événements : 3^{ème} démarche

```
package evenement;
import javax.swing.JFrame;
public class TestMouseListener extends JFrame {
    public TestMouseListener() {
        super("Test JFrame Mouse Listener ");
        this.setSize(400, 400);
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.addMouseListener(new MyJFrameMouseListener2());
    }
    public static void main(String[] args) {
        new TestMouseListener().setVisible(true);
    }
}
```

Étape 2



Gestion des événements : 4^{ème} démarche

4^{ème} démarche : créer une classe anonyme qui hérite de `MouseAdapter`

Les classes internes anonymes (anonymous inner-classes) sont des classes internes qui ne possèdent pas de nom. Elles ne peuvent donc être instanciées qu'à l'endroit où elles sont définies.

Ce type de classe est très pratique lorsqu'une classe doit être utilisée une seule fois : c'est par exemple le cas d'une classe qui doit être utilisée comme un callback.

Une syntaxe particulière de l'opérateur `new` permet de déclarer et instancier une classe interne :

```
new ClassOrInterface (){  
    // définition des attributs et des méthodes de la classe interne  
}
```

Gestion des événements : 4^{ème} démarche

```
public class TestMouseListener4 extends JFrame {
    public TestMouseListener4() {
        super("Test JFrame Mouse Listener ");
        this.setSize(400, 400);

        this.addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                int x = e.getX(); int xOnScreen = e.getXOnScreen();
                int y = e.getY(); int yOnScreen = e.getYOnScreen();

                JOptionPane.showMessageDialog(null, "(x,y)=(" + x + "," + y + ")\n"
                    + "(xOnScreen,yOnScreen)=(" + xOnScreen + "," + yOnScreen
                    + ")", "Location", JOptionPane.PLAIN_MESSAGE);
            }
        });
    }
}
```

Gestion des événements : 5^{ème} démarche

5^{ème} démarche : elle utilise les classes internes mais qu'on ne va pas voir dans le cadre de ce cours.

Gestion des événements : ActionListener

java.awt.event

Interface ActionListener

All Superinterfaces:

[EventListener](#)

All Known Subinterfaces:

[Action](#)

Method Summary

void [actionPerformed](#)([ActionEvent](#) e)

Invoked when an action occurs.

Gestion des événements : MouseListener

java.awt.event

Interface **MouseListener**

All Superinterfaces:

[EventListener](#)

All Known Subinterfaces:

[MouseListener](#)

Method Summary

void [mouseClicked](#)([MouseEvent](#) e)

Invoked when the mouse button has been clicked (pressed and released) on a component.

void [mouseEntered](#)([MouseEvent](#) e)

Invoked when the mouse enters a component.

void [mouseExited](#)([MouseEvent](#) e)

Invoked when the mouse exits a component.

void [mousePressed](#)([MouseEvent](#) e)

Invoked when a mouse button has been pressed on a component.

void [mouseReleased](#)([MouseEvent](#) e)

Invoked when a mouse button has been released on a component.

Gestion des événements : MouseMotionListener

java.awt.event

Interface MouseMotionListener

All Superinterfaces:

[EventListener](#)

All Known Subinterfaces:

[MouseListener](#)

Method Summary

void [mouseDragged](#)([MouseEvent](#) e)

Invoked when a mouse button is pressed on a component and then dragged.

void [mouseMoved](#)([MouseEvent](#) e)

Invoked when the mouse cursor has been moved onto a component but no buttons have been pushed.

Gestion des événements : MouseWheelListener

java.awt.event

Interface MouseWheelListener

All Superinterfaces:

[EventListener](#)

Method Summary

void [mouseWheelMoved](#)([MouseWheelEvent](#) e)

Invoked when the mouse wheel is rotated.

Gestion des événements : KeyListener

java.awt.event

Interface KeyListener

All Superinterfaces:

[EventListener](#)

Method Summary

void **keyPressed**([KeyEvent](#) e)

Invoked when a key has been pressed.

void **keyReleased**([KeyEvent](#) e)

Invoked when a key has been released.

void **keyTyped**([KeyEvent](#) e)

Invoked when a key has been typed.

Gestion des événements : ItemListener

java.awt.event

Interface ItemListener

All Superinterfaces:

[EventListener](#)

Method Summary

void [itemStateChanged](#)([ItemEvent](#) e)

Invoked when an item has been selected or deselected by the user.

Gestion des événements : FocusListener

java.awt.event

Interface FocusListener

All Superinterfaces:

[EventListener](#)

Method Summary

void [focusGained](#)([FocusEvent](#) e)

Invoked when a component gains the keyboard focus.

void [focusLost](#)([FocusEvent](#) e)

Invoked when a component loses the keyboard focus.

Gestion des événements : WindowListener

java.awt.event

Interface WindowListener

All Superinterfaces:

[EventListener](#)

Method Summary

void [windowActivated](#)([WindowEvent](#) e)

Invoked when the Window is set to be the active Window.

void [windowClosed](#)([WindowEvent](#) e)

Invoked when a window has been closed as the result of calling dispose on the window.

void [windowClosing](#)([WindowEvent](#) e)

Invoked when the user attempts to close the window from the window's system menu.

void [windowDeactivated](#)([WindowEvent](#) e)

Invoked when a Window is no longer the active Window.

void [windowDeiconified](#)([WindowEvent](#) e)

Invoked when a window is changed from a minimized to a normal state.

void [windowIconified](#)([WindowEvent](#) e)

Invoked when a window is changed from a normal to a minimized state.

void [windowOpened](#)([WindowEvent](#) e)

Invoked the first time a window is made visible.

Gestion des événements : ListSelectionListener

javax.swing.event

Interface ListSelectionListener

All Superinterfaces:

[EventListener](#)

Method Summary

void **valueChanged**([ListSelectionEvent](#) e)

Called whenever the value of the selection changes.

Gestion des événements : Exemple 1

```
public class TestActionListener extends JFrame implements ActionListener{
    private JButton jButtonValider;
    private JLabel jLabelTest;
    public TestActionListener() {
        super("Test action sur un bouton");
        this.setSize(400,100);
        JPanel jPanelMain=new JPanel();
        jLabelTest = new JLabel();
        jButtonValider = new JButton("Valider");
        jButtonValider.addActionListener(this);

        jPanelMain.add(jLabelTest);
        jPanelMain.add(jButtonValider);
        this.getContentPane().add(jPanelMain);
    }
    public void actionPerformed(ActionEvent e) {
        jLabelTest.setText(jButtonValider.getText());
    }
}
```



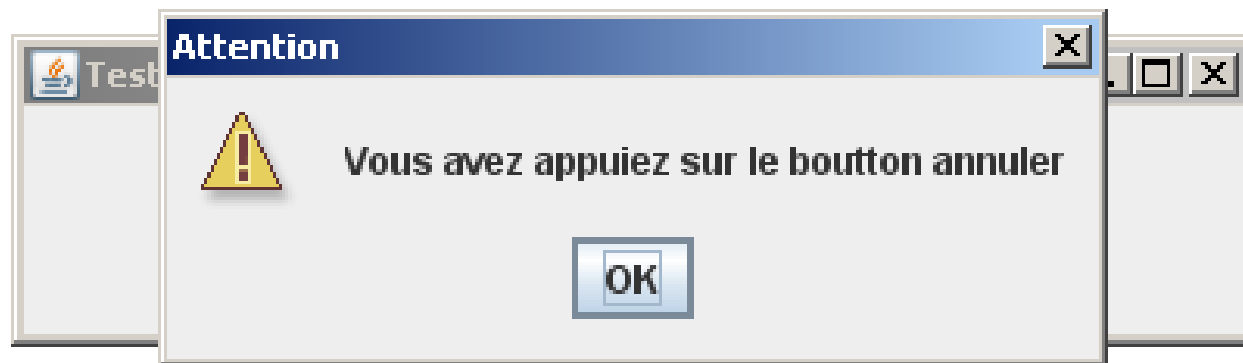
Gestion des événements : Exemple 2

```
public class TestActionListener extends JFrame implements ActionListener{
    private JButton jButtonValider;
    private JLabel jLabelTest;
    public TestActionListener() {
        super("Test action sur un bouton");
        this.setSize(400,100);
        JPanel jPanelMain=new JPanel();
        jLabelTest = new JLabel();
        jButtonValider = new JButton("Valider");
        jButtonValider.addActionListener(this);
        jPanelMain.add(jLabelTest);
        jPanelMain.add(jButtonValider);
        this.getContentPane().add(jPanelMain);
    }
    public void actionPerformed(ActionEvent e) {
        jLabelTest.setText(e.getActionCommand());
    }
}
```



La méthode **getActionCommand()** : cette méthode permet de retourner la chaîne de commande (sous forme d'une chaîne de caractères) associée au bouton qui a déclenché l'événement. Par défaut cette chaîne est l'étiquette du bouton.

Gestion des événements : Exemple 3



Gestion des événements : Exemple 3

```
public class TestActionListener extends JFrame implements ActionListener{
```

```
    private JButton jButtonValider;
```

```
    private JLabel jLabelTest;
```

```
    private JButton jButtonAnnuler;
```

```
    public TestActionListener() {
```

```
        jButtonAnnuler = new JButton("Annuler");
```

```
        jButtonAnnuler.addActionListener(this);
```

```
        jPanelMain.add(jButtonAnnuler);
```

```
    }
```

```
    public void actionPerformed(ActionEvent e) {
```

```
        Object source=e.getSource();
```

```
        if(source instanceof JButton)
```

```
        {
```

```
            JButton temp=(JButton) source;
```

```
            if(temp.equals(jButtonValider))
```

```
                jLabelTest.setText(jButtonValider.getText());
```

```
            else if(temp.equals(jButtonAnnuler))
```

```
                JOptionPane.showMessageDialog(TestActionListener.this,
```

```
                "Vous avez appuyé sur le bouton annuler", "Attention", JOptionPane.WARNING_MESSAGE);
```

```
        }
```

```
    }
```

```
}
```

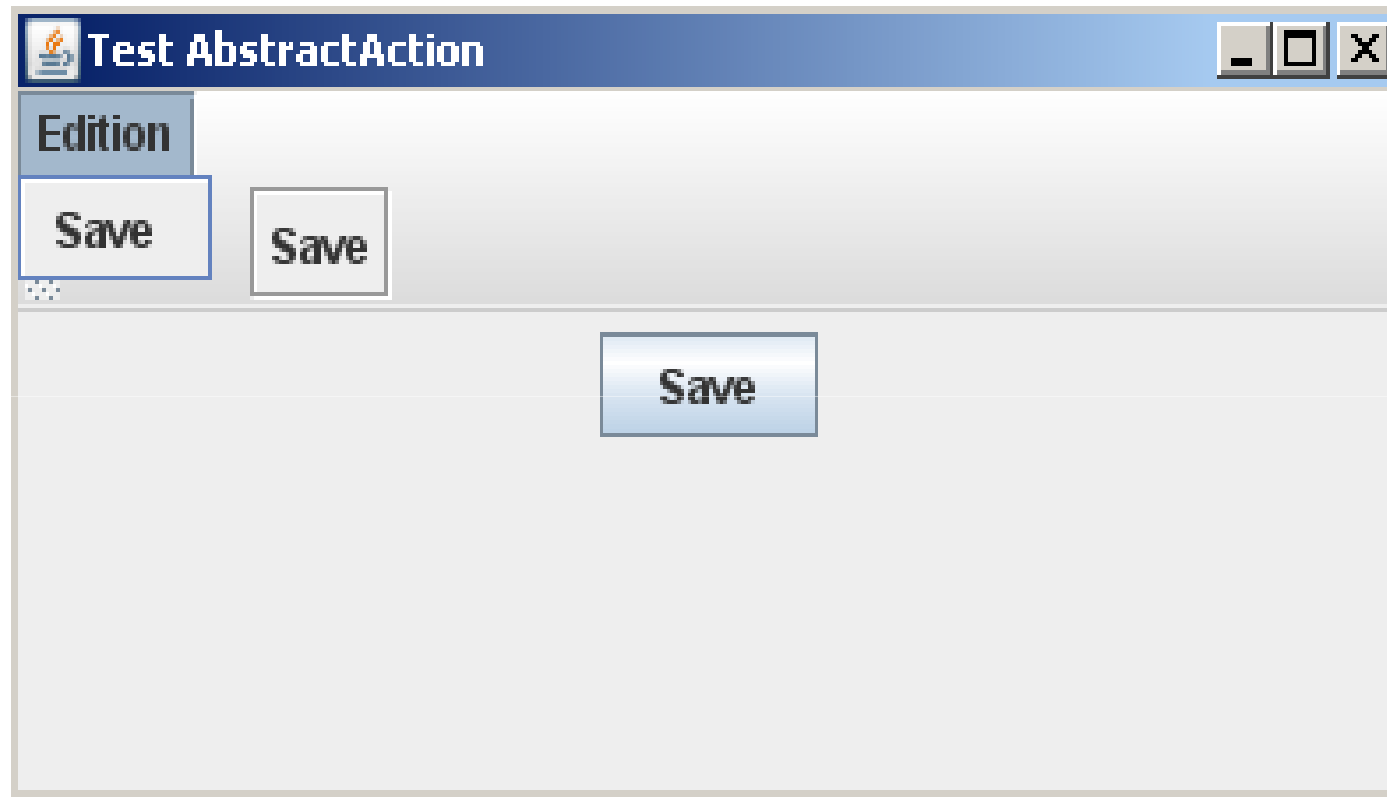
La méthode **getSource()** permet de retourner une référence sur l'objet qui a déclenché l'événement *Action*.

Gestion des événements : `AbstractAction`

`AbstractAction` : Cette classe représente une action de Swing. On peut lui donner un texte et une icône et on peut évidemment lui dire que faire en cas de clic dans la méthode `actionPerformed`.

Dans une application, une même action peut être déclencher à la fois par plusieurs manières (Ex. la fonctionnalité copier dans Word). Créer une seule fois le comportement voulu puis l'associer par exemple à une option d'une menu déroulant ou à une option d'un menu surgissant ou un bouton.

Gestion des événements : AbstractAction



Gestion des événements : AbstractAction

```
public class TestAbstractAction extends JFrame{
    public TestAbstractAction() {
        jMenuBar = new JMenuBar();
        jToolBar = new JToolBar();
        jMenuEdition = new JMenu("Edition");
        jMenuItemSave = new JMenuItem();
        jMenuBar.add(jMenuEdition);
        jMenuEdition.add(jMenuItemSave);
        this.setJMenuBar(jMenuBar);
        jPanel = new JPanel();
        abstractActionSave = new AbstractAction("Save") {
            public void actionPerformed(ActionEvent arg0) {
                System.out.println("Save");
            }
        };
        jButtonSave = new JButton(abstractActionSave);
        jMenuItemSave.setAction(abstractActionSave);
        jToolBar.addSeparator(new Dimension(50,10));
        jToolBar.add(abstractActionSave);
        jPanel.add(jButtonSave);
        this.add(jToolBar,"North");
        this.add(jPanel);
    }
}
```

- Introduction
- Les composants SWING
- Gestionnaire de disposition
- Gestion des événements
- Accès à la base de données
- Le graphisme en JAVA.
- Les applets
- Thread
- Les collections et type générique

Accès à la base de données

Dans de nombreuses applications, il est nécessaire de se connecter à une base de données.

En JAVA, pour se connecter à une base de données et effectuer des manipulations sur cette base, il existe l'API JDBC.

JDBC est l'acronyme de « **J**ava **D**ata**B**ase **C**onnectivity » et désigne une API (en français interface de programmation, en anglais Application Programming Interface) créée par Sun (achetée depuis par Oracle) qui permet d'exécuter des instructions SQL afin de permettre un accès aux bases de données avec Java. L'api JDBC est disponible dans le package **java.sql** qui est nativement présent au sein de l'api JAVA.

Accès à la base de données

Pour manipuler une base de données, il faut importer des classes du package `java.sql`. Les principales classes de ce package sont : **DriverManager**, **Connection**, **Statement**, **PreparedStatement** et **ResultSet**. Ces classes seront utilisées pour accéder à une base de données depuis un programme java.

➤ La classe **DriverManager** charge et configure le driver de la base de données.

➤ La classe **Connection** réalise la connexion et l'authentification à la base de données.

➤ La classe **Statement** (et **PreparedStatement**) contient la requête SQL et la transmet à la base de données.

➤ La classe **ResultSet** permet de parcourir les informations retournées par la base de données dans le cas d'une sélection de données. Il contient les données de la ligne courante et un pointeur sur la ligne suivante.

Accès à la base de données

Concrètement, pour manipuler une base de données, il faut suivre les étapes suivantes :

- 1) Chargement du pilote
- 2) Connexion à la base de données
- 3) Exécution de la requête
- 4) Exploitation des résultats de la requête s'il s'agit d'une requête de sélection
- 5) Fermeture de la connexion

Accès à la base de données : chargement du pilote

Cette étape consiste à choisir le **pilote (driver)** du SGBD à utiliser ultérieurement pour la connexion et le charger.

Pilote: contient toutes les classes nécessaires pour communiquer avec une base de données.

Syntaxe :

Class.forName("<nom du pilote>")

Exemple 1

Class.forName("[sun.jdbc.odbc.JdbcOdbcDriver](#)") ; permet de se connecter à toute base de données via **ODBC** à travers une source de données ODBC.

- très utilisée avec base de données sous windows
- ne nécessite pas un téléchargement de pilote mais une création d'une source de données ODBC.

Accès à la base de données : chargement du pilote

Exemple 2

`Class.forName("oracle.jdbc.driver.OracleDriver");` permet de se connecter à une base de données créée sous **Oracle**. Il faut avant tout télécharger le pilote (un fichier .jar).

Remarque : l'utilisation de la méthode `Class.forName()` peut lever une exception de type **java.lang.ClassNotFoundException** (exception explicite) si le pilote recherché n'a pas pu être trouvé. Donc cette exception doit être traitée.

```
try {  
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
} catch (ClassNotFoundException e) {  
    System.out.println("Impossible de charger le driver"); e.printStackTrace();return;  
}  
try {  
    Class.forName("oracle.jdbc.driver.OracleDriver");  
} catch (ClassNotFoundException e) {  
    System.out.println("Impossible de charger le driver"); e.printStackTrace(); System.exit(-1);  
}
```

Accès à la base de données : connexion à la base

Pour établir une connexion, on utilisera la méthode statique **getConnection** de la classe **DriverManager** qui retourne un objet de type **Connection**

Syntaxe :

```
Connection connection=DriverManager.getConnection("<url de la base de données>",[nomUtilisateur],[motDePasse]);
```

Method Summary

static [Connection](#)**getConnection**([String](#) url)

Attempts to establish a connection to the given database URL.

static [Connection](#)**getConnection**([String](#) url, [Properties](#) info)

Attempts to establish a connection to the given database URL.

static [Connection](#)**getConnection**([String](#) url, [String](#) user, [String](#) password)

Attempts to establish a connection to the given database URL.

Accès à la base de données : connexion à la base

Afin de localiser la base de données, il est indispensable de spécifier une adresse sous forme d'URL qui est dérivée des url d'internet dont le schéma général est :

String url = "jdbc:<subprotocol>:<subname>";

➤<jdbc>: Le protocole dans une URL JDBC est toujours jdbc

➤<subprotocol> Cela correspond au nom du driver ou au mécanisme de connexion à la base de données.

➤<subname> Une manière d'identifier la source de données. Ce dernier élément dépend complètement du sous-protocole et du driver.

Exemple :

✓jdbc:odbc:maBase;CacheSize=30;

✓jdbc:mysql://localhost/maBase

✓jdbc:oracle:oci8@:maBase

✓jdbc:oracle:thin@://localhost:8000:maBase

✓jdbc:sybase:Tds:localhost:5020/maBase

Le nom de la base de données doit être celui saisi dans le nom de la source de données sous ODBC.

Accès à la base de données : connexion à la base

1^{er} exemple d'une connexion avec un pilote `sun.jdbc.odbc.JdbcOdbcDriver` (en supposant que « `EPI_INFO_04` » est le nom d'une source de données ODBC)

```
try {  
    Connection connection=  
        DriverManager.getConnection("jdbc:odbc:EPI_INFO_04", "",  
    "");  
} catch (SQLException e) {  
    e.printStackTrace();  
}
```

Remarque : l'utilisation de la méthode **getConnection** peut lever une exception de type **java.sql.SQLException** (exception explicite) si la connexion échoue. Donc cette exception doit être traitée.

Accès à la base de données : connexion à la base

2^{ème} exemple d'une connexion avec un pilote
sun.jdbc.odbc.JdbcOdbcDriver (sans créer une source de données)

```
try {  
    Connection connection = DriverManager.getConnection("jdbc:odbc:  
driver={Microsoft Access Driver (*.mdb, *.accdb)};  
dbq=C:/ze/etudiant.accdb");  
  
} catch (SQLException e) {  
    e.printStackTrace();  
}
```


Accès à la base de données : connexion à la base

3^{ème} exemple d'une connexion avec un pilote `sun.jdbc.odbc.JdbcOdbcDriver` (sans créer une source de données) et un objet instance de la classe **`java.util.Properties`**

```
Properties props = new Properties();
props.setProperty("user", "userName");
props.setProperty("password", "motDePasse");
props.setProperty("autoReconnect", "true");
try {
    Connection connection=
        DriverManager.getConnection("jdbc:odbc:EPI_INFO_04",props)
;
} catch (SQLException e) {
    e.printStackTrace();
}
```

Accès à la base de données : exécuter la requête

Une fois la connexion établie, il est possible d'exécuter des requêtes SQL.

Les objets qui peuvent être utilisés pour obtenir des informations sur la base de données sont :

Classe	Rôle
DatabaseMetaData	Informations à propos de la base de données : nom des tables, index, version ...
ResultSet	Résultat d'une requête et information sur une table. L'accès se fait enregistrement par enregistrement.
ResultSetMetaData	Informations sur les colonnes (nom et type) d'un ResultSet

A partir de la connexion, créer un « statement » (état) correspondant à une requête particulière

Accès à la base de données : exécuter la requête

Les requêtes SQL sont exécutées avec les méthodes d'un objet **Statement** (état) que l'on obtient à partir d'un objet **Connection**

```
try {  
    Statement statement=connection.createStatement();  
} catch (SQLException e) {  
    e.printStackTrace();  
}
```

Pour une requête de type interrogation (SELECT), la méthode à utiliser de la classe Statement est **executeQuery()**. Pour des traitements de mise à jour, il faut utiliser la méthode **executeUpdate()**. Lors de l'appel à la méthode d'exécution, il est nécessaire de lui fournir en paramètre la requête SQL sous forme de chaîne.

Accès à la base de données : exécuter la requête

Le résultat d'une requête d'interrogation est renvoyé dans un objet de la classe **ResultSet** par la méthode **executeQuery()**.

```
try {  
    ResultSet resultSet = statement.executeQuery("SELECT * FROM etudiant");  
} catch (SQLException e) {  
    e.printStackTrace();  
}
```

La méthode **executeQuery()** retourne un objet instance de la classe **ResultSet** qui contient des méthodes permettant de parcourir tous les enregistrements sélectionnés.

Remarque : pour utiliser **executeQuery**, il faut traiter l'exception **java.sql.SQLException** qui est lancée au cas où il y a une erreur dans la requête.

Accès à la base de données : exécuter la requête

La méthode **executeUpdate()** permet d'exécuter une requête de mise à jours et retourne le nombre d'enregistrements qui ont été mis à jour.

```
try {  
    String requete = " insert into etudiant values('Med',20)";  
    Statement statement=connection.createStatement();  
    int nbUpdate = statement.executeUpdate(requete);  
    System.out.println("Le nombre de ligne mis à jours : "+nbUpdate);  
} catch (SQLException e) {  
    e.printStackTrace();  
}
```

Lorsque la méthode `executeUpdate()` est utilisée pour exécuter un traitement de type DDL (Data Definition Langage : définition de données) comme la création d'une table, elle retourne 0. Si la méthode retourne 0, cela peut signifier deux choses : le traitement de mise à jour n'a affecté aucun enregistrement ou le traitement concernait un traitement de type DDL.

Si l'on utilise `executeQuery()` pour exécuter une requête SQL ne contenant pas d'ordre `SELECT`, alors une exception de type `SQLException` est levée.

Accès à la base de données : ResultSet

La classe **ResultSet** contient des méthodes qui permettent de parcourir tous les enregistrements sélectionnés. Notons, qu'au contraire de la plupart des structures de données en java (tableaux, vector, List, etc.) les index des lignes représentant les colonnes commencent à 1.

C'est une classe qui représente une abstraction d'une table qui se compose de plusieurs enregistrements constitués de colonnes qui contiennent les données. Les principales méthodes pour obtenir des données sont :

Méthode	Rôle
getInt(int)	retourne sous forme d'entier le contenu de la colonne dont le numéro est passé en paramètre.
getInt(String)	retourne sous forme d'entier le contenu de la colonne dont le nom est passé en paramètre.
getDate(int)	retourne sous forme de date le contenu de la colonne dont le numéro est passé en paramètre.
getDate(String)	retourne sous forme de date le contenu de la colonne dont le nom est passé en paramètre.
next()	se déplace sur le prochain enregistrement : retourne false si la fin est atteinte
close()	ferme le ResultSet
getMetaData()	retourne un objet de type ResultSetMetaData associé au ResultSet.

Accès à la base de données : ResultSet

```
try {  
    String requete = "select nom,age from etudiant";  
    ResultSet resultSet = statement.executeQuery(requete);  
    while(resultSet.next())  
    {  
        System.out.println(resultSet.getString("nom"));  
        System.out.println(resultSet.getInt("age"));  
    }  
} catch (SQLException e) {  
    e.printStackTrace();  
}
```

Remarque 1 : la méthode next() fait avancer le curseur d'un enregistrement à un autre. Quand il n'y plus aucun enregistrement, la méthode retourne la valeur false.

Remarque 2 : juste après l'exécution de executeQuery, le curseur est placé juste avant le premier enregistrement. Il faut donc exécuter la méthode next pour que le curseur soit sur le premier enregistrement

Remarque 3 : à part getString et getInt, il existe d'autres méthodes de type getXxx telles que getFloat getDouble, getDate, getBoolean. Ces méthodes permettent de convertir le champ SQL dans le type Xxx

Accès à la base de données : déconnexion

Fermeture de la connexion

Un objet de type **Connection** est automatiquement détruit par le ramasse-miettes dès qu'il ne sera plus utilisé. Cependant, dans certaines applications (cas où plusieurs utilisateurs veulent accéder à la base de données), il est préférable de détruire cet objet dès qu'on a terminé de travailler avec.

Accès à la base de données : déconnexion

```
String driver = "nom.du.driver";String url = "url";String login = "login";String password = "password";
Connection connection = null;
try{
    Class.forName(driver);
    connection = DriverManager.getConnection(url,login,password);
    //travail avec les données
}
catch(ClassNotFoundException cnfe){
    System.out.println("Driver introuvable : ");
    cnfe.printStackTrace();
}
catch(SQLException sqle){
    System.out.println("Erreur SQL : ");
    //Cf. Comment gérer les erreurs ?
}
catch(Exception e){
    System.out.println("Autre erreur : ");
    e.printStackTrace();
}
finally
{
    if(connection!=null){try{connection.close();}catch(Exception e){e.printStackTrace();}}
    //etc.
}
```

Accès à la base de données : déconnexion

Tout comme pour une **Connection**, même si le Garbage Collector, libérera les ressources allouées au **Statement**, il est conseillé de le fermer explicitement.

```
Connection connection = null; Statement statement = null;
try{
    //initialisation de la connexion et du statement
}
catch(SQLException sqle){ }
catch(AutreException ae){ }
finally{
    if(statement !=null)
    {
        try{ statement.close(); } catch(Exception e){ e.printStackTrace(); }
    }
    if(connection !=null)
    {
        try{ connection.close(); } catch(Exception e){ e.printStackTrace(); }
    }
}
```

Accès à la base de données : Exemple

```
private Connection con;
private Statement stmt;
public TestConnexionMySQL() {
    // Chargement du pilote
    try {
        Class.forName("com.mysql.jdbc.Driver");
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(99);
    }
    // Connexion à la base de données MySQL "EPI" avec
    // le login "" et le mot de passe ""
    try {
        String dBurl = "jdbc:mysql://localhost:3306/EPI";
        con = DriverManager.getConnection(dBurl,"root","");
        stmt = con.createStatement();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Accès à la base de données : Exemple

```
public int insertPersonne(String nom, String prenom, int age) {  
    ResultSet resultats = null;  
    int idGenere = -1;  
    try {  
        stmt.executeUpdate("INSERT INTO Personne (nom, prenom,  
age)values ('" + nom+ "', '" + prenom + "'," + age + ")",  
Statement.RETURN_GENERATED_KEYS);  
        resultats = stmt.getGeneratedKeys();  
        if (resultats.next()) {  
            idGenere = resultats.getInt(1);  
        }  
        resultats.close();  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
    return idGenere;  
}
```

Accès à la base de données : Exemple

```
public void affichePersonne(int key) {  
    ResultSet resultats = null;  
    try {  
        boolean encore = false;  
        resultats = stmt.executeQuery("Select Nom, Prenom, Age From  
Personne Where Numero="+ key);  
        if (resultats != null)  
            encore = resultats.next();  
        if (encore)  
            System.out.println(resultats.getString(1) + " "+  
resultats.getString(2) + " (" + resultats.getInt(3) + " ans)");  
        else  
            System.out.println("Il n'y a personne avec ce numero !");  
        resultats.close();  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```

Accès à la base de données : Exemple

```
public void afficheToutesLesPersonnes() {  
    ResultSet resultats = null;  
    try {  
        resultats = stmt.executeQuery("Select nom, prenom, age, numero  
from Personne");  
        boolean encore = resultats.next();  
        while (encore) {  
            System.out.println(resultats.getInt(4) + " : "+  
resultats.getString(1) + " " + resultats.getString(2) + " (" + resultats.getInt(3) + " ans)");  
            encore = resultats.next();  
        }  
        resultats.close();  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```

Accès à la base de données : Exemple

```
public void supprimeToutesLesPersonnes() {  
    try {  
        stmt.executeUpdate("Delete From Personne");  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}  
  
public void supprimePersonne(String nom) {  
    try {  
        stmt.executeUpdate("Delete From Personne Where nom=" + nom + "");  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}  
  
public void creerTable(String nomTable, String champs) {  
    ResultSet resultats = null;  
    try {  
        stmt.executeUpdate("CREATE TABLE " + nomTable + " (" + champs + ")");  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```

Accès à la base de données : Exemple

```
public static void main(java.lang.String[] args) {  
    TestConnexionMySQL test = new TestConnexionMySQL ();  
    // Création de la table Personne  
    test.creerTable("Personne", "numero integer primary key AUTO_INCREMENT,nom  
VARCHAR(25),prenom VARCHAR(25), age integer");  
    test.supprimeToutesLesPersonnes ();  
    System.out.println ("Insertion de trois personnes");  
    int id1 = test.insertPersonne ("Mohamed", "Ben Foulen", 36);  
    int id2 = test.insertPersonne ("Isam", "Ben Foulen", 40);  
    int id3 = test.insertPersonne ("Imen", "Ben Foulen", 48);  
    System.out.println ("Affichage de la personne de clef primaire = "+id1);  
    test.affichePersonne (id1);  
    System.out.println ("Affichage de toutes les personnes");  
    test.afficheToutesLesPersonnes ();  
    System.out.println ("Suppression de Mohamed et affichage de toutes les personnes");  
    test.supprimePersonne ("Mohamed");  
    test.afficheToutesLesPersonnes ();  
}
```


Accès à la base de données : Exemple

Insertion de trois personnes

Affichage de la personne de clef primaire = 1

Mohamed Ben Foulen (36 ans)

Affichage de toutes les personnes

1 : Mohamed Ben Foulen (36 ans)

2 : Isam Ben Foulen (40 ans)

3 : Imen Ben Foulen (48 ans)

Suppression de Mohamed et affichage de toutes les personnes

2 : Isam Ben Foulen (40 ans)

3 : Imen Ben Foulen (48 ans)

Accès à la base de données : PreparedStatement

Ce type de requête est particulièrement adapté pour une exécution répétée d'une même requête avec des paramètres différents.

L'interface **PreparedStatement** hérite de l'interface **Statement**.

```
try {  
    String req="insert into etudiant values (?,?,?,?,?)";  
    PreparedStatement preparedStatement=connection.prepareStatement(req);  
    preparedStatement.setInt(1, 1);  
    preparedStatement.setString(2, "Foulen");  
    preparedStatement.setString(3, "Ben Foulen");  
    preparedStatement.setInt(4, 35);  
    preparedStatement.setFloat(5, 35.5f);  
    preparedStatement.executeUpdate();  
} catch (SQLException e) {  
    e.printStackTrace();  
}
```

Accès à la base de données : Parcours

La version JDBC 2 prévoit :

- D'autres méthodes de déplacement du curseur (autre que next) :
avant, arrière, saut.
- Des méthodes de modification du résultat
- Exploitation d'une nouvelle interface : RowSet

Accès à la base de données : Parcours

```
Statement createStatement(int resultSetType,  
                           int resultSetConcurrency,  
                           int resultSetHoldability)
```

Pour les possibilités de déplacement (int resultSetType) :

- **ResultSet.TYPE_FORWARD_ONLY** : valeur par défaut. Elle indique que les déplacements du curseur ne peuvent se faire « qu'en avant ».
- **ResultSet.TYPE_SCROLL_INSENSITIVE** : le curseur peut être déplacé dans les deux sens. Le terme insensitive indique que le ResultSet est insensible aux modifications des valeurs dans la base de données. => vue statique des données
- **ResultSet.TYPE_SCROLL_SENSITIVE** : le curseur peut être déplacé dans les deux sens. Le terme sensitive indique que le ResultSet est sensible aux modifications des valeurs dans la base de données. => vue dynamique des données

Accès à la base de données : Parcours

```
Statement createStatement(int resultSetType,  
                           int resultSetConcurrency,  
                           int resultSetHoldability)
```

Pour les possibilités d'actualisation (int resultSetConcurrency) :

- **ResultSet.CONCUR_READ_ONLY** : c'est la valeur par défaut. Elle indique que les données contenues dans le ResultSet ne peuvent qu'être lues.
- **ResultSet.CONCUR_UPDATABLE** : cette valeur indique que l'on peut modifier les données de la base via le ResultSet.

Accès à la base de données : Parcours

```
Statement createStatement(int resultSetType,  
                           int resultSetConcurrency,  
                           int resultSetHoldability)
```

Pour le maintien des curseurs (int resultSetHoldability) :

- **ResultSet.HOLD_CURSORS_OVER_COMMIT** : les objets ResultSet ne sont pas fermés. Ils restent ouverts lorsqu'une validation est effectuée implicitement ou explicitement ;
- **ResultSet.CLOSE_CURSORS_AT_COMMIT** : les objets ResultSet sont fermés lorsqu'une validation est effectuée implicitement ou explicitement.

Accès à la base de données : Parcours

- **next()** : déplace le curseur sur la ligne suivante. Elle retourne true si le curseur est positionné sur une ligne, false si le curseur a dépassé la fin du tableau.
- **previous()** : déplace le curseur sur la ligne précédente.
- **first(), last()** : positionne le curseur sur la première ligne ou la dernière ligne du tableau. Retourne true ou false suivant que cette ligne existe ou pas.
- **beforeFirst(), afterLast()** : positionne le curseur sur l'une des lignes virtuelles se trouvant avant la première ligne, ou après la dernière. Ce mouvement est toujours possible, même sur un tableau vide.
- **relative(int n)** : déplace le curseur de n lignes. Le déplacement a lieu vers le bas du tableau si ce nombre est positif, vers le haut s'il est négatif.
- **absolute(int n)** : positionne le curseur en absolu dans le tableau. La première ligne du tableau est numérotée 1, donc **absolute(1)** positionne le curseur sur cette première ligne.

Accès à la base de données : Parcours

```
Statement statement = connection.createStatement(  
    ResultSet.TYPE_SCROLL_SENSITIVE,  
    ResultSet.CONCUR_READ_ONLY);  
    String sql = "SELECT * FROM MaTable";  
    ResultSet resultat = statement.executeQuery(sql);  
    resultat.first();  
    // on récupère le "prix" de la première ligne  
    double d1 = resultat.getDouble("prix");  
    // un traitement quelconque  
    resultat.fisrt();  
    // on vérifie que le prix n'a pas changé durant le traitement  
    resultat.refreshRow();  
    double d2 = resultat.getDouble("prix");  
    if (d1 != d2) {  
        // le prix a changé  
    }  
}
```


Accès à la base de données : RowSet

```
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.sql.rowset.JdbcRowSet;
import javax.sql.rowset.RowSetProvider;
import com.sun.rowset.JdbcRowSetImpl;
public class TestRowSet {
    public static void main(String[] args) throws ClassNotFoundException, SQLException {
        // CHARGEMENT DU DRIVER
        Class.forName("com.mysql.jdbc.Driver");
        // CREATION D'UN ROWSET
        JdbcRowSet rowset = new JdbcRowSetImpl(); // JdbcRowSet
rowset=RowSetProvider.newFactory().createJdbcRowSet();
        // propriétés du RowSet
        rowset.setUrl("jdbc:mysql://localhost:3306/EPI");
        rowset.setUsername("root");
        rowset.setPassword("");
        rowset.setType(ResultSet.TYPE_SCROLL_INSENSITIVE);
        rowset.setConcurrency(ResultSet.CONCUR_UPDATABLE);
        rowset.setCommand("SELECT * FROM Personne");
        // exécution et peuplement du RowSet
        rowset.execute();
        // TRAITEMENT DES DONNEES
        rowset.last();
        // parcours des résultats du dernier au premier tuple
        while (rowset.previous()) {
            System.out.println("Nom = " + rowset.getObject("nom") + " Prenom = " + rowset.getString(3));
        }
    }
}
```

Accès à la base de données : RowSet

```
rowset.first();
rowset.updateString("nom", "modification");
rowset.updateRow();
// positionnement sur la seconde ligne et suppression de celle-ci
rowset.relative(1);
rowset.deleteRow();
// insertion d'une nouvelle ligne
rowset.moveToInsertRow();
rowset.updateString("prenom", "un nouveau prénom");
rowset.updateString("nom", "un nouveau nom");
rowset.updateInt("age", 29);
rowset.insertRow();
rowset.moveToCurrentRow();
```

```
}
```

```
}
```

Accès à la base de données : Parcours

```
Connection connection = ...;
//création d'une instruction renvoyant des résultats

//pouvant être mis à jour
Statement statement =
connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
//on sélectionne tous les tuples de la table Annuaire
String sql = "SELECT * FROM Annuaire";
ResultSet resultat = statement.executeQuery(sql);
// on se place sur le premier tuple récupéré
resultat.first();
//on récupère la valeur de la colonne "nom"
String nom1 = resultat.getString("nom");
//on met à jour la valeur de la colonne "nom"
resultat.updateString("nom", "nouveauNom");
//on met à jour la valeur dans la table
resultat.updateRow();
String nom2 = resultat.getString("nom");
System.out.println("Ancien nom = "+nom1+ "Nouveau nom = "+nom2);
```

Accès à la base de données : métadonnées

Les métadonnées sont des connaissances qu'on peut avoir sur la structure de tables (noms des champs et leurs types, ...) mais aussi des informations sur le SGBD lui-même. Ainsi deux types de métadonnées peuvent être utilisés :

- Des métadonnées associés à l'objet ResultSet qui permettent d'avoir des informations sur la structure des tables (nom des tables, noms des champs et leurs types, etc.)
- Des métadonnées associées à une base et qui permettent d'avoir des informations relatives au SGBD et au pilote, etc.

Métadonnées associées au ResultSet

La méthode `getMetaData()` de **ResultSet** retourne un objet de type **ResultSetMetaData** qui contient les méthodes permettant d'accéder aux métadonnées :

```
try {  
    ResultSet rs = statement.executeQuery("SELECT a, b, c FROM  
TABLE2");  
    ResultSetMetaData rsmd = rs.getMetaData();  
    int numberOfColumns = rsmd.getColumnCount();  
    boolean b = rsmd.isSearchable(1);  
} catch (SQLException e) {  
    e.printStackTrace();  
}
```

Accès à la base de données : métadonnées

Métadonnées associées au ResultSet

String**getColumnClassName**(int column)

Returns the fully-qualified name of the Java class whose instances are manufactured if the method `ResultSet.getObject` is called to retrieve a value from the column.

int**getColumnCount**()

Returns the number of columns in this `ResultSet` object.

String**getColumnName**(int column)

Get the designated column's name.

int**getColumnType**(int column)

Retrieves the designated column's SQL type.

String**getTableName**(int column)

Gets the designated column's table name.

boolean**isAutoIncrement**(int column)

Indicates whether the designated column is automatically numbered.

int**isNullable**(int column)

Indicates the nullability of values in the designated column.

boolean**isSearchable**(int column)

Indicates whether the designated column can be used in a where clause.

Accès à la base de données : métadonnées

Métadonnées associées à la base

On commence par récupérer un objet de type **DatabaseMetadata** à partir de la méthode **getMetaData()** de la classe **Connection**

```
try {  
    DatabaseMetaData metaData = connection.getMetaData();  
} catch (SQLException e) {  
    e.printStackTrace();  
}
```

<http://docs.oracle.com/javase/6/docs/api/java/sql/DatabaseMetaData.html>

Accès à la base de données : batch

```
public class TestBatch {  
    private Connection connection;  
    private Statement statement;  
    static{  
        try {  
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
        } catch (ClassNotFoundException e) {  
            System.out.println("Impossible de charger le driver");  
        }  
    }  
    public TestBatch() {  
        Savepoint savepoint=null;  
        try {  
connection = DriverManager.getConnection("jdbc:odbc:driver={Microsoft Access Driver  
(*.*.mdb, *.accdb)};dbq=c:/ze/etudiant.accdb");  
        statement = connection.createStatement();  
        DatabaseMetaData metaData = connection.getMetaData();  
        System.out.println(metaData.supportsBatchUpdates());  
        System.out.println(metaData.supportsSavepoints());  
        savepoint = connection.setSavepoint("first");  
        System.out.println("Start insert");  
        }  
    }  
}
```


Accès à la base de données : batch

```
        for (int i = 0; i < 50; i++) {
            System.out.println(i);
            String request="insert into etudiant (nom,
prenom,age)values ('test"+i+"','test"+i+"',i)";
            statement.addBatch(request);
        }
        int[] executeBatch = statement.executeBatch();
        System.out.println(Arrays.toString(executeBatch));
        connection.commit();
        connection.rollback(savepoint);
        connection.close();

    } catch (SQLException e) {
        e.printStackTrace();
    }

}

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

}
```

Composant swing : JTable

Constructor Summary

JTable()

Constructs a default JTable that is initialized with a default data model, a default column model, and a default selection model.

JTable(int numRows, int numColumns)

Constructs a JTable with numRows and numColumns of empty cells using DefaultTableModel.

JTable(Object[][] rowData, Object[] columnNames)

Constructs a JTable to display the values in the two dimensional array, rowData, with column names, columnNames.

JTable(TableModel dm)

Constructs a JTable that is initialized with dm as the data model, a default column model, and a default selection model.

JTable(TableModel dm, TableColumnModel cm)

Constructs a JTable that is initialized with dm as the data model, cm as the column model, and a default selection model.

JTable(TableModel dm, TableColumnModel cm, ListSelectionModel sm)

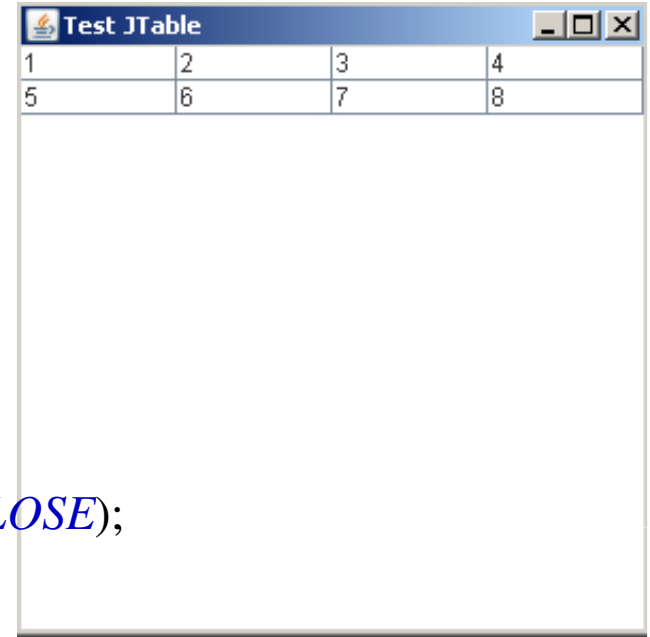
Constructs a JTable that is initialized with dm as the data model, cm as the column model, and sm as the selection model.

JTable(Vector rowData, Vector columnNames)

Constructs a JTable to display the values in the Vector of Vectors, rowData, with column names, columnNames.

Composant swing : JTable

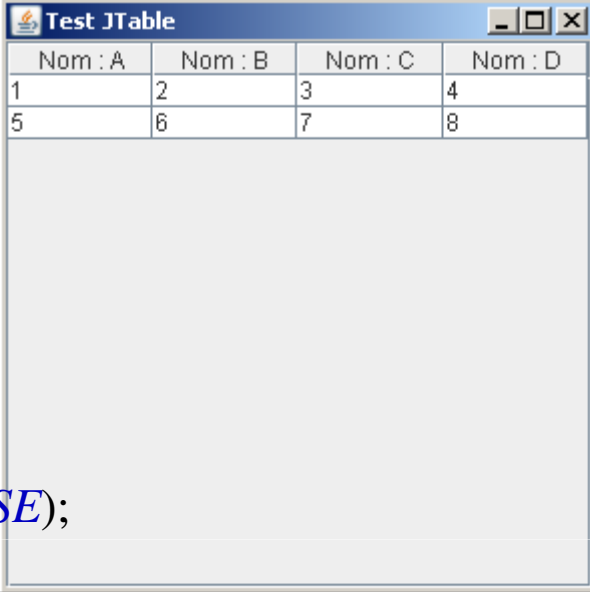
```
package graphique;
import javax.swing.JFrame;
import javax.swing.JTable;
public class TestJTable extends JFrame{
    public TestJTable() {
        super("Test JTable");
        this.setSize(300,300);
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        Object [][]data={
                                {1,2,3,4},
                                {5,6,7,8}
        };
        String[] tableHeader={"Nom : A","Nom : B","Nom : C","Nom : D"};
        JTable jTable=new JTable(data,tableHeader);
        this.getContentPane().add(jTable);
    }
    public static void main(String[] args) {
        new TestJTable().setVisible(true);
    }
}
```



1	2	3	4
5	6	7	8

Composant swing : JTable

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
public class TestJTable extends JFrame{
    public TestJTable() {
        super("Test JTable");
        this.setSize(300,300);
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        Object [][]data={
                                {1,2,3,4},
                                {5,6,7,8}
        };
        String[] tableHeader={"Nom : A","Nom : B","Nom : C","Nom : D"};
        JTable jTable=new JTable(data,tableHeader);
        this.getContentPane().add(new JScrollPane(jTable));
    }
    public static void main(String[] args) {
        new TestJTable().setVisible(true);
    }
}
```



Nom : A	Nom : B	Nom : C	Nom : D
1	2	3	4
5	6	7	8

Composant swing : JTable

Inconvénients de cette méthode :

- Toutes les cellules sont éditables
- Elle exige que toutes les données soient mises dans un tableau ou un vecteur ce qui n'est pas très appropriée pour des données extraites d'une base de données

Solution :

JTable(TableModel dm)

Constructs a JTable that is initialized with dm as the data model, a default column model, and a default selection model.

Composant swing : JTable

Pour créer un JTable à partir d'un modèle, il faut que l'objet modèle soit un objet d'une classe **C** qui hérite de la classe abstraite **AbstractTableModel** (cette classe implémente l'interface **TableModel**). Pour cela, il faudra que la classe **C** implémente au moins les 3 méthodes suivantes :

- `public int getRowCount();` // retourne le nombre de lignes du tableau
- `public int getColumnCount();` //retourne le nombre de colonnes du tableau
- `public Object getValueAt(int numeroLigne, int numColonne);`
//retourne la valeur de la cellule numeroLigne et numColonne.

Composant swing : JTable

```
package graphique;
import javax.swing.table.AbstractTableModel;
class MonModele extends AbstractTableModel {
    String[] nomColonnes = { "Nom", "Age", "Etudiant" };
    Object[][] donnees = { { "Berriri", 15, false }, { "Sassi", 21, true },
                           { "Jabeur", 18, false }, { "Slim", 23, true } };
    // Les 3 méthodes suivantes sont obligatoires
    public int getRowCount() {
        return donnees.length;
    }
    public int getColumnCount() {
        return nomColonnes.length;
    }
    public Object getValueAt(int ligne, int col) {
        return donnees[ligne][col];
    }
}
```

Composant swing : JTable

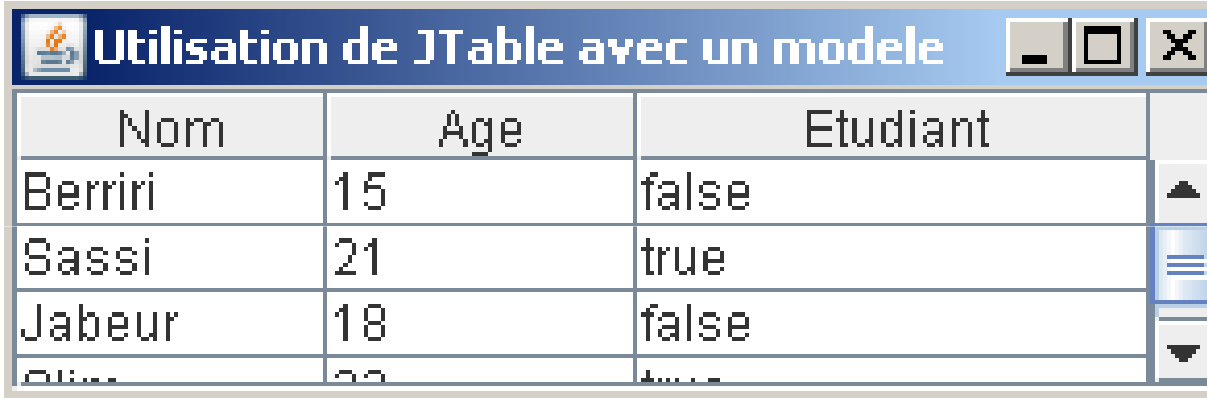
// Les méthodes suivantes ne sont obligatoires

```
public String getColumnName(int col) {  
    return nomColonnes[col];  
}  
public boolean isCellEditable(int ligne, int col) {  
    if (col <= 1) {  
        return false;  
    } else {  
        return true;  
    }  
}  
public void setValueAt(Object value, int ligne, int col) {  
    donnees[ligne][col] = value;  
    fireTableCellUpdated(ligne, col); // pour mettre à jour la table  
}  
}
```


Composant swing : JTable

```
import javax.swing.JTable;
import javax.swing.ListSelectionModel;
import javax.swing.table.TableColumn;
public class MaTableSansBD extends JFrame {
    private JTable table; private MonModele modele ;
public MaTableSansBD() {
    modele = new MonModele();
    table = new JTable(modele); JScrollPane scrollPane = new JScrollPane(table);
    table.setFillViewportHeight(true); // permet de prendre tout l'espace du conteneur
    table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    // permet à l'utilisateur de trier le tableau en cliquant sur une des colonnes
    table.setAutoCreateRowSorter(true);
    // cette partie permet de changer la taille de certaines colonnes
    TableColumn column = null;
    for (int i = 0; i < 3; i++) {
        column = table.getColumnModel().getColumn(i);
        if (i == 2) {column.setPreferredWidth(100); // la colonne est plus grande}
        else {column.setPreferredWidth(50);}
    }
    container.add(scrollPane);
}
}
```

Composant swing : JTable



Nom	Age	Etudiant	
Berriri	15	false	▲
Sassi	21	true	☰
Jabeur	18	false	▼
...

Accès à la base de données : JTable & ResultSet

Remarques :

- On n'est pas obligé de stocker les données dans un tableau multidimensionnel. Le plus important est que les trois méthodes `getRowCount()`; `getColumnCount()`; `getValueAt(int numeroLigne, int numColonne)` soient implémentés et remplissent leur rôles.
- Si on veut ajouter ou supprimer une ligne dans un JTable, il faudra appeler, juste après la modification, la méthode **`fireTableDataChanged()`** pour que les changements s'appliquent sur le JTable

Question : comment faire pour qu'un JTable soit rempli des informations d'une table d'une base de données?

- Introduction
- Les composants SWING
- Gestionnaire de disposition
- Gestion des événements
- Accès à la base de données
- **Le graphisme en JAVA.**
- Les applets
- Thread
- Les collections et type générique