Interface Graphique en Java
Introduction à Swing

Ilhem BOUSSAID

USTHB

5 octobre 2009

Plan

Les composants atomiques

- Un composant atomique est considéré comme étant une entité unique.
- Java propose beaucoup de composants atomiques :
  - boutons, CheckBox, Radio
  - Combo box
  - List, menu
  - TextField, TextArea, Label
  - FileChooser, ColorChooser,
  - ...

Les boutons

Java propose différent type de boutons :

- Le bouton classique est un **JBouton**.
- **JCheckBox** pour les case à cocher
- **JRadioBouton** pour un ensemble de bouton
- **JMenutItem** pour un bouton dans un menu
- **JCheckBoxMenuItem**
- **JRadioButtonMenuItem** **JToggleButton** Super Classe de **CheckBox** et **Radio**

JButton

La classe **JButton** permet de définir des boutons sur lesquels on peut cliquer.
Ses principales méthodes sont :

- **JButton(String)** construction avec définition du texte contenu dans le bouton
- **JButton(ImageIcon)** construction avec une icone dans le bouton
- **JButton(String, ImageIcon)** construction avec définition du texte et d'une icône dans le bouton
- **String getText()** qui retourne le texte contenu dans le bouton
- **void setText(String)** qui définit le texte contenu dans le bouton
- **void addActionListener(ActionListener)** pour associer l'objet qui traitera les clics sur le bouton

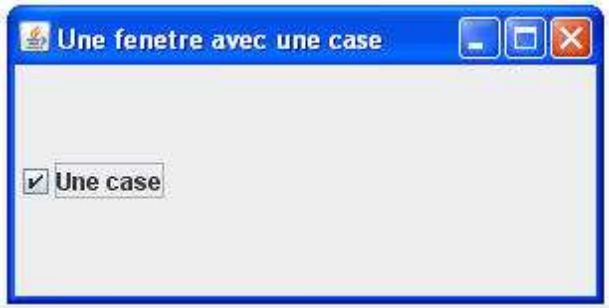
JCheckBox

- **JCheckBox** permet de définir des cases à cocher. Ses principales méthodes sont :
  - **JCheckBox(String)** construction avec définition du texte contenu dans la case à cocher
  - **JCheckBox(String,boolean)** construction avec en plus définition de l'état initial de la case à cocher
  - **boolean isSelected()** qui retourne l'état de la case à cocher (cochée ou non). Par défaut, une case à cocher est construite dans l'état non coché (false).
  - **void setSelected(boolean)** qui définit l'état de la case à cocher (cochée ou non).
  - **String getText()** qui retourne le texte contenu dans la case à cocher
  - **void setText(String)** qui définit le texte contenu dans la case à cocher
  - **void addActionListener(ActionListener)** pour associer l'objet qui traitera les actions sur la case à cocher

Exemple

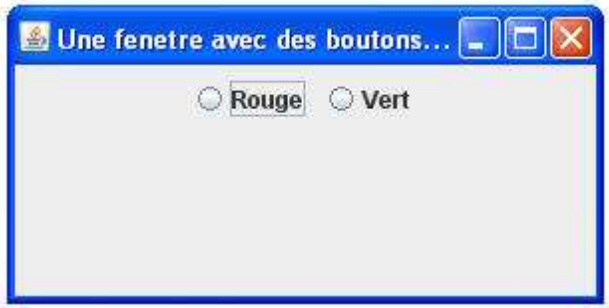
La case à cocher de type **JCheckBox** permet à l'utilisateur d'effectuer un choix de type oui/non.

```
import java.awt.*;
import javax.swing.*;
public class CheckBoxFrame extends JFrame {
private JCheckBox MaCase;
public CheckBoxFrame () {
super("Une fenetre avec une case") ;
setBounds(10,40,300,200) ;
MaCase = new JCheckBox("Une case") ;
MaCase.setSelected(true) ;
getContentPane().add(MaCase) ;
}
/* + méthode main */
}
```



JRadioButton

- Le bouton radio de type **JRadioButton** permet à l'utilisateur d'effectuer un choix de type oui/non.
- Mais sa vocation est de faire partie d'un groupe de boutons dans lequel u**ne seule option peut être sélectionnée à la fois.**
- Un objet de type **ButtonGroup** sert uniquement à assurer la désactivation automatique d'un bouton lorsqu'un bouton du groupe est activé.
- Un bouton radio qui n'est pas associé à un groupe, exception faite de son aspect, se comporte exactement comme une case à cocher.



JRadioButton

```
import java.awt.*;
import javax.swing.*;
public class BoutonRadioFrame extends JFrame {
private JRadioButton bRouge;
private JRadioButton bVert;
public BoutonRadioFrame () {
super("Une fenetre avec des boutons radio") ;
setBounds(10,40,300,200) ;
bRouge = new JRadioButton("Rouge") ;
bVert = new JRadioButton("Vert") ;
ButtonGroup groupe = new ButtonGroup() ;
groupe.add(bRouge) ; groupe.add(bVert) ;
Container contenu = getContentPane() ;
contenu.setLayout(new FlowLayout()) ;
contenu.add(bRouge) ;
contenu.add(bVert) ;
}
public static void main(String args[]) {
BoutonRadioFrame fen = new BoutonRadioFrame();
fen.setVisible(true);
fen.setBounds(10, 200, 300, 150);
} }
```

JComboBox

- Un **JComboBox** est un composant permettant de faire un **seul choix** parmi plusieurs propositions.
- Quelques méthodes de JComboBox :
  - **JComboBox(Object[])** construction avec définition de la liste. On peut utiliser un tableau de chaînes de caractères (**String**) ou de toute autre classe d'objets.
  - **void addItem(Object)** qui ajoute une valeur possible de choix
  - **int getSelectedIndex()** qui retourne le numéro du choix actuel
  - **Object getSelectedItem()** qui retourne l'objet associé au choix actuel. **Attention** l'objet retourné est de classe Object, il faudra le transformer en sa classe d'origine (**String** par exemple).
  - **void setSelectedIndex(int)** qui sélectionne un élément défini par son numéro
  - **void addActionListener(ActionListener)** pour associer l'objet qui traitera les choix faits

Exemple

```
import java.awt.*;
import javax.swing.*;
public class ComboBoxFrame extends JFrame {
  private JComboBox combo = new JComboBox();
  private JLabel label = new JLabel("Une liste déroulante");
  String[] tab = {"Option 1", "Option 2", "Option 3", "Option 4"};
  public ComboBoxFrame(){
  this.setTitle("Fenetre avec une liste déroulante");
  this.setSize(300, 300);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setLocationRelativeTo(null);
  Container pane = this.getContentPane();
  pane.setBackground(Color.white);
  pane.setLayout(new BorderLayout());
  combo.setPreferredSize(new Dimension(100,20));
  combo = new JComboBox(tab);
  JPanel top = new JPanel();
  top.add(label);
  top.add(combo);
  pane.add(top, BorderLayout.NORTH);
  this.setVisible(true);
  }}
```

Illustration

JList

- Une **JList** propose plusieurs éléments rangés en colonne.
- Une **JList** peut proposer une sélection simple ou multiple. Les valeurs possibles sont définies par des constantes de la classe ListSelectionModel :
  - **SINGLE_SELECTION** : **Un seul élément** peut être sélectionné
  - **SINGLE_INTERVAL_SELECTION** : **Une zone d'éléments** contigus peut être sélectionnée
  - **MULTIPLE_INTERVAL_SELECTION** : **Un ensemble d'éléments quelconques** peut être sélectionné (comportement par défaut)
- Les **JList** sont souvent contenues dans un scrolled pane (**JScrollPane**)

Quelques méthodes de JList

Quelques méthodes de JList :

- **JList(Object[])** construction avec définition de la liste. On peut utiliser un tableau de chaînes de caractères (**String**) ou de toute autre classe d'objets.
- **setListData(Object[])** définition de la liste. On peut utiliser un tableau de chaînes de caractères (String) ou de toute autre classe d'objets.
- **void setVisibleRowCount(int)** qui définit le nombre d'éléments visibles sans ascenseur
- **int getSelectedIndex()** qui retourne le numéro du premier élément sélectionné
- **int[] getSelectedIndices()** qui retourne les numéros des éléments sélectionnés
- **Object getSelectedValue()** qui retourne le premier objet sélectionné.

Quelques méthodes de JList

Quelques méthodes de JList :

- **Object[] getSelectedValues()** qui retourne les objets actuellement sélectionnés.
- **void setSelectedIndex(int)** qui sélectionne l'élément désigné par son numéro
- **void setSelectedIndices(int[])** qui sélectionne les éléments désignés par leurs numéros
- **void clearSelection()** qui annulle toutes les sélections
- **int getModel().getSize()** retourne le nombre d'éléments dans la liste
- **Object getModel().getElementAt(int)** retourne l'objet de la liste correspondant au rang donné en paramètre.
- **void addListSelectionListener(ListSelectionListener)** pour associer l'objet qui traitera les sélections dans la liste

JList

- le constructeur de **JList** avec un tableau d'objets permet de créer une liste initialisée mais pas de la modifier dynamiquement.
- Pour créer une liste modifiable dynamiquement (ajout/suppression d'éléments) il faut lui associer un contenu sous forme d'un modèle :

```
DefaultListModel contenu = new DefaultListModel ();
JList listeDynamique = new JList(contenu);
```

- On pourra alors ajouter des éléments à la liste en les ajoutant au modèle par :

```
contenu.addelement(Object) /* ou */
contenu.insertElementAt(Object,int)
```

- Et en enlever par :

```
contenu.removeElementAt(int) /* ou */
contenu.removeElement(Object)
```

JSlider

- Les **JSlider** permettent de définir des curseurs horizontaux ou verticaux gradués
- Les **JSlider** permettent la saisie graphique d'un nombre
- Un JSlider doit contenir les bornes max et min
- Quelques méthodes de JSlider :
  - JSlider(int,int,int,int,int) qui définit l'ascenseur avec dans l'ordre des paramètres son orientation (**JSlider.HORIZONTAL** ou **JSlider.VERTICAL**), ses valeurs minimales et maximales et sa position initiale.
  - **void setMajorTickSpacing(int)** qui détermine le nombre d'unités correspondant à un graduation plus longue.
  - **void setMinorTickSpacing(int)** qui détermine le nombre d'unités correspondant à un graduation courte.
  - **void setPaintTicks(boolean)** qui détermine si les graduations sont ou non dessinées.

JSlider

- Quelques méthodes de JSlider (suite) :
  - **void setPaintTrack(boolean)** qui détermine si la piste du curseur est ou non dessinée.
  - **Hashtable createStandardLabels(int,int)** qui permet de créer une table d'étiquettes constituée de nombres entiers. Le premier paramètres est le pas, le second est la valeur de départ. Une telle table pourra être associée au curseur par sa méthode **setLabelTable** (la classe **Hashtable** est une classe directement héritée de **Dictionary**).
  - **void setLabelTable(Dictionary)** qui associe une table d'étiquettes aux graduations longues du curseur
  - **void setPaintLabels(boolean)** qui détermine si les valeurs correpondant aux graduations longues sont ou non écrites.
  - **int getValue()** qui retourne la positon actuelle de l'ascenseur
  - **void setValue(int)** qui définit la position de l'ascenseur

JSlider

```
import java.awt.*;
import javax.swing.*;
public class SliderFrame extends JFrame {
Container  pane = this.getContentPane();
public SliderFrame() {
setTitle("Fenetre avec JSlider");
pane.setLayout(new BorderLayout());
JPanel panel = new JPanel(new FlowLayout());
pane.add(panel,BorderLayout.CENTER);
panel.add(new JSlider(1,100,50));
}
public static void main(String []arg){
SliderFrame frame = new SliderFrame();
frame.setBounds(100, 150, 300,100 );
frame.setVisible(true);
}
}
```

JSpinner

- **JSpinner** permet de sélectionner une information parmi une séquence ordonnée de valeurs, à l'aide de deux boutons permettant de faire défiler les valeurs possibles.
- Le composant **JSpinner** est un conteneur composé de trois sous-composants :
  - Un champ appelé éditeur dans lequel seront affichées ou éventuellement éditées les valeurs. Par défaut le champ d'édition est un **JFormattedTextField** (mais n'importe quel **JComponent** peut être utilisé).
  - Deux boutons permettant de faire défiler les valeurs possibles
- Le **modèle** du composant **JSpinner** doit implémenter la classe abstraite **AbstractSpinnerModel**.
- La librairie **Swing** offre trois modèles par défaut :
  - SpinnerListModel
  - SpinnerNumberModel
  - SpinnerDateModel

JSpinner

```
import java.awt.*;
import java.util.*;
import javax.swing.*;
public class MonthSpiner extends JFrame {
public MonthSpiner() {
  super("Month␣Spinner");
  setSize(200, 100);
    setDefaultCloseOperation(EXIT_ON_CLOSE);

  Container c = getContentPane();
  c.setLayout(new FlowLayout(FlowLayout.LEFT, 4, 4));

  c.add(new JLabel("Expiration␣Date:"));
  Date today = new Date();
  // Start the spinner today, but don't set a min or max date
  // The increment should be a month
  JSpinner s = new JSpinner(new SpinnerDateModel(today, null,
  null, Calendar.MONTH));
  JSpinner.DateEditor de = new JSpinner.DateEditor(s, "MM/yy");
  s.setEditor(de);
  c.add(s);
  setVisible(true); } /* + Methode main */ }
```

JScrollBar

- Un **JScrollBar** permet de définir des ascenseurs horizontaux ou verticaux
- Quelques méthodes de JScrollBar :
  - **JScrollbar(int,int,int,int,int)** qui définit l'ascenseur avec dans l'ordre des paramètres son orientation (on peut y mettre les constantes **JScrollbar.HORIZONTAL** ou **JScrollbar.VERTICAL**), sa position initiale, le pas avec lequel on le déplace en mode page à page, ses valeurs minimales et maximales.
  - **int getValue()** qui retourne la positon actuelle de l'ascenseur
  - **void setValue(int)** qui définit la position de l'ascenseur
  - **int getBlockIncrement()** qui retourne la valeur utilisée pour le pas en mode page à page
  - **void setBlockIncrement (int)** qui définit la valeur utilisée pour le pas en mode page à page

JScrollBar

- Quelques méthodes de JScrollBar (suite) :
  - **int getUnitIncrement()** qui retourne la valeur utilisée pour le pas unitaire
  - **void setUnitIncrement (int)** qui définit la valeur utilisée pour le pas unitaire
  - **int getMaximum()** qui retourne la valeur maximale actuelle
  - **void setMaximum (int)** qui définit la valeur maximale actuelle
  - **int getMinimum()** qui retourne la valeur minimale actuelle
  - **void setMinimum (int)** qui définit la valeur minimale actuelle
  - **void addAdjustmentListener(AdjustmentListener)** pour associer l'objet qui traitera les déplacements de l'ascenseur

JProgressBar

- Un **JProgressBar** permet de dessiner une barre dont la longueur représente une quantité ou un pourcentage
- Quelques méthodes de JProgressBar :
  - **JProgressBar (int,int,int)** qui définit la barre avec dans l'ordre des paramètres son orientation (**JProgressBar.HORIZONTAL** ou **JProgressBar.VERTICAL**) et ses valeurs minimales et maximales.
  - **int getValue()** qui retourne la positon actuelle de la barre
  - **void setValue(int)** qui définit la position de la barre
  - **int getMaximum()** qui retourne la valeur maximale actuelle
  - **void setMaximum (int)** qui définit la valeur maximale actuelle
  - **int getMinimum()** qui retourne la valeur minimale actuelle
  - **void setMinimum (int)** qui définit la valeur minimale actuelle
  - **void addChangeListener(ChangeListener)** pour associer l'objet qui traitera les déplacements de la barre

JTextArea

- Un **JTextArea** permet de définir des zones de texte sur plusieurs lignes modifiables sans ascenseurs (pour disposer d'ascenseur, faire appel à **JScrollPane**).
- Quelques méthodes de JTextArea :
  - **JTextArea(String)** qui crée une zone de texte avec un contenu initial
  - **JTextArea(String,int,int)** qui crée une zone de texte avec un contenu initial et précise le nombre de lignes et de colonnes de la zone de texte
  - **void append(String)** qui ajoute la chaîne à la fin du texte affiché

JTextArea

- Quelques méthodes de JTextArea (suite) :
  - **void insert(String,int)** qui insère la chaîne au texte affiché à partir du rang donné
  - **void setTabSize(int)** qui définit la distance entre tabulations.
  - **void setLineWrap(boolean)** qui détermine si les lignes longues doivent ou non être repliées.
  - **void setWrapStyleWord(boolean)** qui détermine si les lignes sont repliées en fin de mot (true) ou pas.

JTextField

- Un **JTextField** est un composant qui permet d'écrire du texte.
- Un **JTextField** a une seul ligne contrairement au **JTextArea**
- Le **JPasswordField** permet de cacher ce qui est écrit
- Quelques méthodes de JTextField :
  - **JTextField(String)** qui la crée avec un contenu initial
  - **JTextField(String,int)** qui la crée avec un contenu initial et définit le nombre de colones
  - **void addActionListener(ActionListener)** pour associer l'objet qui traitera les modifications du texte

JTextField

- Lorsque l'on saisit du texte dans un tel composant celui-ci adapte sa taille au texte saisi au fur et à mesure. Ce comportement n'est pas toujours souhaitable, on peut l'éviter en lui définissant une taille préférentielle par sa méthode **setPreferredSize** et une taille minimale par sa méthode **SetMinimumSize**

Exemple

```
import java.awt.*;
import javax.swing.*;
class TextFieldFrame extends JFrame {
private JTextField jtf = new JTextField("Valeur par défaut");
private JLabel label = new JLabel("Un JTextField");
public TextFieldFrame () {
  this.setTitle("Une fenetre avec zone de texte");
  this.setSize(300, 200);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setLocationRelativeTo(null);
  JPanel top = new JPanel();
  Font police = new Font("Arial", Font.BOLD, 14);
  jtf.setFont(police);
  jtf.setPreferredSize(new Dimension(150, 30));
  jtf.setForeground(Color.BLUE);
  top.add(label);
  top.add(jtf);
  Container pane = this.getContentPane();;
  pane.setLayout(new BorderLayout());
  pane.add(top, BorderLayout.NORTH);
  this.setVisible(true);
} }
```

JTextField et JTextArea

- **JTextField** et **JTextArea** ont en commun un certain nombre de méthodes dont voici les principales :
  - **void copy()** qui copie dans le bloc-notes du système la partie de texte sélectionnée
  - **void cut()** qui fait comme copy puis supprime du texte la partie sélectionnée
  - **void paste()** qui fait copie dans le texte le contenu du bloc-notes du système
  - **String getText()** qui retourne le texte contenu dans la zone de texte
  - **void setText(String)** qui définit le texte contenu dans la zone de texte
  - **int getCaretPosition()** qui retourne la position du curseur d'insertion dans le texte (rang du caractère)
  - **int setCaretPosition(int)** qui place le curseur d'insertion dans le texte au rang indiqué (rang du caractère)

Méthodes de JTextField et JTextArea (suite)

- Principales méthodes de **JTextField** et **JTextArea**(suite) :
  - **int moveCaretPosition(int)** qui déplace le curseur d'insertion dans le texte en sélectionnant le texte depuis la position précédente.
  - **setEditable(boolean)** qui rend la zone de texte modifiable ou pas
  - **int getSelectionStart()** qui retourne la position du début du texte sélectionné (rang du caractère)
  - **int getSelectionEnd()** qui retourne la position de la fin du texte sélectionné (rang du caractère)
  - **void setSelectedTextColor(Color c)** qui définit la couleur utilisée pour le texte sélectionné
  - **String getSelectedText()** qui retourne le texte sélectionné

Méthodes de JTextField et JTextArea (suite)

- Principales méthodes de **JTextField** et **JTextArea**(suite) :
  - **void select(int,int)** qui sélectionne le texte compris entre les deux positions données en paramètre
  - **void selectAll()** qui sélectionne tout le texte
  - **Document getDocument()** qui retourne le gestionnaire de document associé à cette zone de texte. C'est ce gestionnaire qui recevra les événements de modification de texte
  - **void addCaretListener(CaretListener)** pour associer l'objet qui traitera les déplacements du curseur de saisie dans le texte

La classe Document

- Prend en charge l'édition de texte.
- Ses principales méthodes sont :
  - **int getLength()** qui retourne le nombre de caractères du document
  - **String getText(int,int)** qui retourne la partie du texte qui commence à la position donnée en premier paramètre et dont la longueur est donnée par le second paramètre
  - **void removeText(int,int)** qui supprime la partie du texte qui commence à la position donnée en premier paramètre et dont la longueur est donnée par le second paramètre
  - **void addDocumentListener(DocumentListener)** pour associer l'objet qui traitera les modifications du texte
- **Note** : Ces deux dernières méthodes peuvent lever une exception de classe **BadLocationException** si les paramètres sont incorrects

JLabel

- Un JLabel permet d'afficher du texte fixe ou une image.
- Un JLabel peut contenir plusieurs lignes et il comprend les tag HTML.
- Le **JPasswordField** permet de cacher ce qui est écrit
- Quelques méthodes de JLabel :
  - **JLabel(String)** qui construit le titre avec son contenu
  - **JLabel(Icon)** qui construit le titre avec une image comme contenu (le paramètre peut être de classe **ImageIcon**)
  - **void setText(String)** qui définit le texte
  - **String getText()** qui retourne le texte
  - **void setIcon(ImageIcon)** qui définit l'image
  - **Icon getIcon()** qui retourne l'image sous forme d'un objet de classe **Icon**. On peut le récupérer dans un objet de classe **ImageIcon** en faisant :

```
ImageIcon img = (ImageIcon) label.getIcon()
```

Exemple 1

```
import java.awt.*;
import javax.swing.*;
public class LabelFrame extends JFrame {
public LabelFrame(){
super("Une fenetre avec un Label") ;
this.setBounds(100, 200, 300, 100);
Container pane = this.getContentPane();
pane.setLayout(new FlowLayout());
JLabel jl = new JLabel("BONJOUR!");
jl.setFont(new Font("Arial", Font.BOLD|Font.ITALIC, 16));
jl.setForeground(Color.red);
pane.add(jl);
}
public static void main(String []arg){
LabelFrame frame = new LabelFrame();
frame.setVisible(true); } }
```

Exemple 2

```
import java.awt.*;
import javax.swing.*;
public class LabelFrame2 extends JFrame {
public LabelFrame2(){
super("Une fenetre avec un Label") ;
this.setBounds(100, 200, 300, 100);
Container pane = this.getContentPane();
pane.setLayout(new FlowLayout());
JLabel jlg = new JLabel(new ImageIcon("images/s5.gif"));
pane.add(jlg);
}
```

Les menu

- Si on a une barre de menu **JMenuBar**, on ajoute des **JMenu** dans la barre.
- Définir des Menus ⇒ utiliser 3 classes **JMenuBar**, **JMenu**, **JMenuItem**
  - **JMenuBar** : barre des menus placée en haut de la fenêtre d'une application
  - La méthode **addSeparator()** peut être utilisée pour insérer un espace entre les composants de la barre d'outils.
  - Une barre de menu est composée de **JMenus**
  - Un objet **JMenu** possède un label, et quand on clique dessus, il peut montrer un menu déroulant
  - Un item d'un objet **JMenu** peuvent être un objet de type **JMenuItem**, **JCheckBoxMenuItem** ou **JRadioButtonMenuItem**
  - Un objet **JMenuItem** est un simple élément de menu avec un label. Il peut avoir une icône en plus de son label

Exemple

```
import java.awt.*;
import javax.swing.*;
public class FrameMenu extends JFrame {
Container  pane = this.getContentPane();
public FrameMenu() {
setTitle("FrameMenu");

JMenuBar jmb = new JMenuBar();
this.setJMenuBar(jmb);

JMenu menuDefinir = new JMenu("Définir");
jmb.add(menuDefinir);
menuDefinir.add(new JMenuItem("le nom"));
menuDefinir.add(new JMenuItem("le prenom"));

JMenu menuAfficher = new JMenu("Afficher");
jmb.add(menuAfficher);
}
/* + méthode main */
}
```

Illustration

JTree

- Un **JTree** permet d'afficher des informations sous forme d'arbre. Les noeuds de l'arbre sont des objets qui doivent implanter l'interface **MutableTreeNode**.
- Une classe de base est proposée pour les noeuds : **DefaultMutableTreeNode.**
- Pour construire un arbre il est conseillé de passer par la classe **TreeModel** qui est la représentation abstraite de l'arbre.