IHM - avec Swing

Composants lourds, composants 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.

Les composants lourds

En java, comme nous l'avons vu au chapitre AWT, les composants dérivent tous de la classe java.awt.Component. Les composants awt 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 (***d'où le gestionnaire LayOutManager pour positionner les composants !***).

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**.

Tous les composants du package AWT sont des composants lourds.

Les composants 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 par 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.

Les composants **Swing** (nom du package : **javax.swing**) sont pour la majorité d'entre eux des composants **légers**.

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.

Swing contient un minimum de composants lourds

Dans le package Swing le nombre de composants lourds est réduit au strict minimum soient 4 genres de fenêtres.

Les fenêtres Swing sont des composants lourds

Les fenêtres en Java Swing :

- **JFrame** à rapprocher de la classe **Frame** dans AWT
- **JDialog** à rapprocher de la classe **Dialog** dans AWT
- **JWindow** à rapprocher de la classe **Window** dans AWT
- **JApplet** à rapprocher de la classe **Applet** dans AWT

Hiérarchie de classe de ces composants de fenêtres :

```
java.lang.Object
 |
 +--java.awt.Component
      |
      +--java.awt.Container
           |
           +--java.awt.Window
           |     |
           |     +--javax.swing.JWindow
           |     |
           |     +--java.awt.Frame
           |     |     |
           |     |     +--javax.swing.JFrame
           |     +--java.awt.Dialog
           |           |
           |           +--javax.swing.JDialog
           +--java.awt.Panel
                 |
                 +--java.applet.Applet
                       |
                       +--javax.swing.JApplet
```

Le principe appliqué étant que si la fenêtre a besoin de communiquer avec le système, les composants déposés sur la fenêtre eux n'en ont pas la nécessité. C'est pourquoi tous les autres composants de **javax.swing** sont des composants légers. Pour utiliser les Swing, il suffit d'importer le package :

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

Il est bien sûr possible d'utiliser des composants AWT et Swing dans la même application. Les événements sont gérés pour les deux packages par les méthodes de l'interface Listener du package **java.awt.event**.

Ce qui signifie que tout ce qui a été dit au chapitre sur les événements pour les composants **AWT** (***modèle de délégation du traitement de l'événement à un écouteur***) est intégralement reportable aux **Swing** sans aucune modification.

En général, les classes des composants **swing** étendent les fonctionnalités des classes des composants **AWT** dont elles héritent (plus de propriétés, plus d'événements,...).

Les autres composants Swing sont légers

Les composants légers héritent tous directement ou indirectement de la classe **javax.swing.JComponen**t :

```
java.lang.Object
 |
 +--java.awt.Component
      |
      +--java.awt.Container
           |
           +--javax.swing.JComponent
```

Dans un programme Java chaque composant graphique Swing (léger) doit donc disposer d'un conteneur de plus haut niveau sur lequel il doit être placé.

Afin d'assurer la communication entre les composants placés dans une fenêtre et le système, le package Swing organise d'une manière un peu plus complexe les relations entre la fenêtre propriétaires et ses composants.

Le JDK1.4.2 donne la liste suivante des composants légers héritant de JComponent : AbstractButton, BasicInternalFrameTitlePane, JColorChooser, JComboBox, JFileChooser, JInternalFrame, JInternalFrame.JDesktopIcon, JLabel, JLayeredPane, JList, JMenuBar, JOptionPane, JPanel, JPopupMenu, JProgressBar, JRootPane, JScrollBar, JScrollPane, JSeparator, JSlider, JSplitPane, JTabbedPane, JTable, JTableHeader, JTextComponent, JToolBar, JToolTip, JTree, JViewport.

Architecture Modèle-Vue-Contrôleur

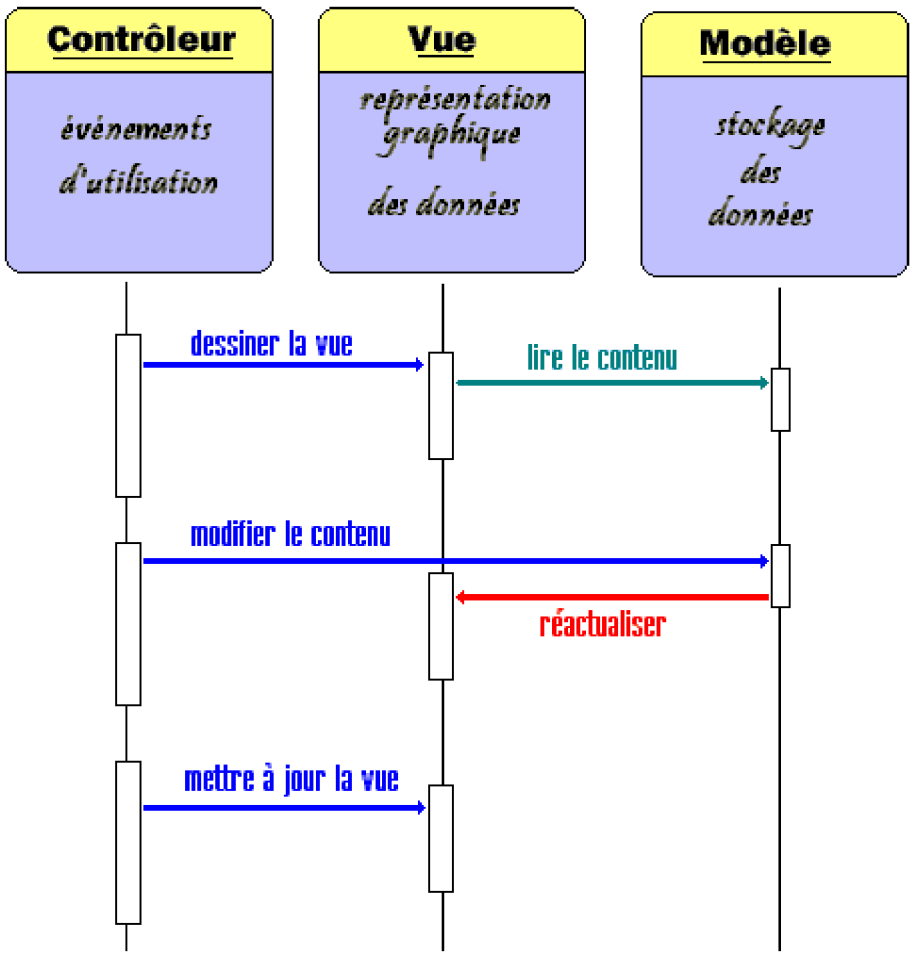
L'architecture Modèle-Vue-Contrôleur en général (MVC)

En technologie de conception orientée objet il est conseillé de ne pas confier trop d'actions à un seul objet, mais plutôt de répartir les différentes responsabilités d'actions entre plusieurs objets. Par exemple pour un composant visuel (bouton, liste etc...) vous déléguez la **gestion du style du composant à une classe** (ce qui permettra de changer facilement le style du composant sans intervenir sur le composant lui-même), vous stockez les données contenues dans le composant dans **une autre classe chargée de la gestion des données de contenu** (ce qui permet d'avoir une gestion décentralisée des données) .

Si l'on recense les caractéristiques communes aux composants visuels servant aux IHM (interfaces utilisateurs), on retrouve 3 constantes générales pour un composant :

- son contenu (les données internes, les données stockées, etc...)
- son apparence (style, couleur, taille, etc...)
- son comportement (essentiellement en réaction à des événements)

Diagramme de séquence UML des interactions MVC



Le schéma précédent représente l'architecture **Modèle-Vue-Contrôleur** (ou design pattern observateur-observé) qui réalise cette conception décentralisée à l'aide de 3 classes associées à chaque composant :

- Le **modèle** qui stocke le contenu, qui contient des méthodes permettant de modifier le contenu et qui n'est pas visuel.
- La **vue** qui affiche le contenu, est chargée de dessiner sur l'écran la forme que prendront les données stockées dans le modèle.
- Le **contrôleur** qui gére les interactions avec l'utilisateur .

Le pluggable look and feel

C'est grâce à cette architecture MVC, que l'on peut implémenter la notion de "**pluggable look and feel (ou plaf)**" qui entend séparer le modèle sous-jacent de la représentation visuelle de l'interface utilisateur. Le code Swing peut donc être réutilisé avec le même modèle mais changer de style d'interface dynamiquement pendant l'exécution.

Voici à titre d'exemple la même interface Java écrite avec des Swing et trois aspects différents (motif, métal, windows) obtenus pendant l'exécution en changeant son look and feel par utilisation de la classe UIManager servant à gérer le look and feel.

avec le système Windows sont livrés 3 look and feel standard : windows (apparence habituelle de windows), motif (apparence graphique Unix) et metal (apparence genre métallique).

Trois exemples de look and feel

Voici ci-après trois aspects de **la même interface utilisateur**, chaque aspect est changé durant l'exécution uniquement par l'appel des lignes de code associées (this représente la fenêtre JFrame de l'IHM) :

Lignes de code pour passer en IHM motif :

```
String UnLook = "com.sun.java.swing.plaf.motif.MotifLookAndFeel" ;
try  {
     UIManager.setLookAndFeel(UnLook); // assigne le look and feel choisi ici motif
     SwingUtilities.updateComponentTreeUI(this.getContentPane( ));  // réactualise le graphisme de l'IHM
     }
catch (Exception exc) {
      exc.printStackTrace( );
}
```

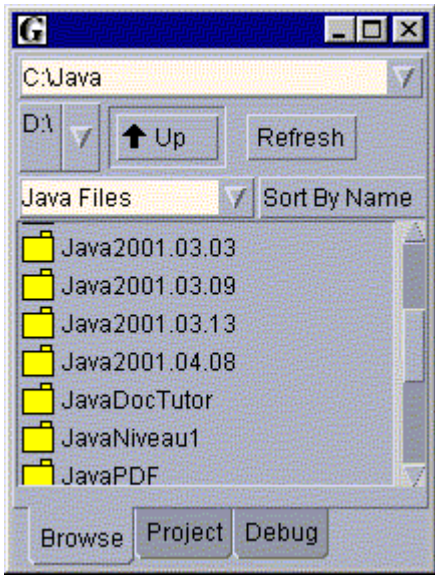
Lignes de code pour passer en IHM métal :

```
String UnLook = "javax.swing.plaf.metal.MetalLookAndFeel" ;
try  {
     UIManager.setLookAndFeel(UnLook); // assigne le look and feel choisi ici metal
     SwingUtilities.updateComponentTreeUI(this.getContentPane( ));  // réactualise le graphisme de l'IHM
     }
catch (Exception exc) {
      exc.printStackTrace( );
}
```

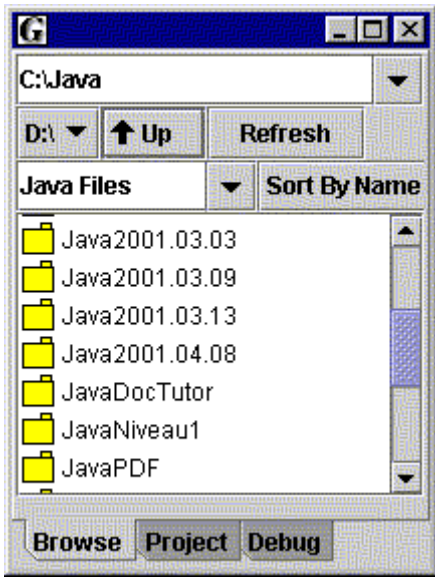
Lignes de code pour passer en IHM Windows :

```
String UnLook = "com.sun.java.swing.plaf.windows.WindowsLookAndFeel" ;
try  {
     UIManager.setLookAndFeel(UnLook); // assigne le look and feel choisi ici windows
     SwingUtilities.updateComponentTreeUI(this.getContentPane( ));  // réactualise le graphisme de l'IHM
     }
catch (Exception exc) {
      exc.printStackTrace( );
}
```

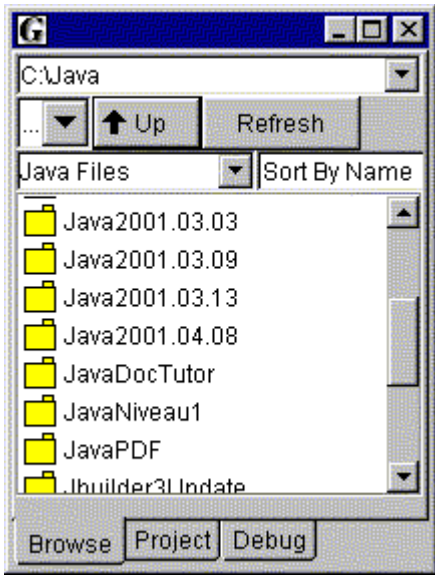
Aspect motif de l'IHM



Aspect métal de l'IHM



Aspect Windows de l'IHM :



Les swing reposent sur MVC

Les composants de la bibliothèque **Swing** adoptent tous cette architecture de type **MVC (Modèle-Vue-Contrôleur)** qui sépare le stockage des données, leur représentation et les interactions possibles avec les données, les composants sont associés à différentes interfaces de modèles de base. Toutefois les swing pour des raisons de souplesse **ne respectent pas strictement l'architecture MVC** que nous venons de citer :

- Le **Modèle**, chargé de stocker les données, qui permet à la vue de lire son contenu et informe la vue d'éventuelles modifications est bien ***représenté par une classe***.
- La **Vue**, permettant une représentation des données (nous pouvons l'assimiler ici à la représentation graphique du composant) peut être ***répartie sur plusieurs classes***.
- Le **Contrôleur**, chargé de gérer les interactions de l'utilisateur et de propager des modifications vers la vue et le modèle peut aussi être réparti sur plusieurs classes, voir même dans des ***classes communes à la vue et au contrôleur***.

Exemple de quelques interfaces de modèles rencontrées dans la bibliothèque Swing

Identificateur de la classe de modèle	*Utilisation*
ListModel	Modèle pour les listes (JList ...)
ButtonModel	Modèle d'état pour les boutons (JButton...)
Document	Modèle de document (JTextField...

La mise en oeuvre des composants Swing ne requiert pas systématiquement l'utilisation des modèles. Il est ainsi généralement possible d'initialiser un composant Swing à l'aide des données qu'il doit représenter. Dans ce cas , le composant exploite un modèle interne par défaut pour stocker les données.

Le composant javax.swing.Jlist

Dans le cas du JList le recours au modèle est impératif, en particulier une vue utilisant le modèle ListModel pour un jList enregistrera un écouteur sur l'implémentation du modèle et effectuera des appels à **getSize**( ) et **getElementAt**( ) pour obtenir le nombre d'éléments à représenter et les valeurs de ces éléments.

Dans l'exemple suivant, l'un des constructeurs de JList est employé afin de définir l'ensemble des données à afficher dans la liste, on suppose qu'il est associé à un modèle dérivant de **ListModel** déjà défini auparavant. L'appel à **getModel**( ) permet d'obtenir une référence sur l'interface ListModel du modèle interne du composant :

```
JList  Jlist1 = new JList(new Objet[] {"un","deux","trois"});
ListModel  modeldeliste = jList1.getModel( );
System.out.println ("Élément       0  :   "  +  modeldeliste.getElementAt(0));
System.out.println ("Nb éléments : "+ modeldeliste.getSize( ))
```

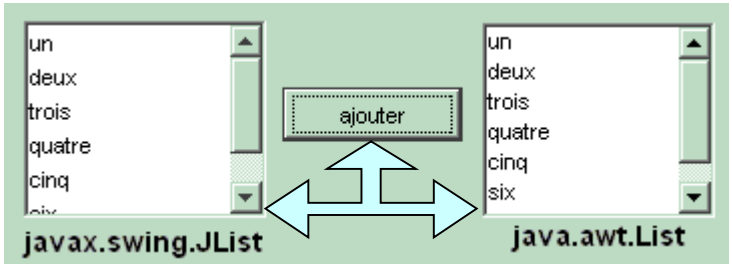
Pour mettre en oeuvre les modèles et les fournir aux composants on utilise la méthode

setModel( ) (***public void*** *setModel (ListModel model) { }*) du composant. Comme *ListModel* est une interface, il nous faut donc implémenter cette interface afin de passer un paramètre effectif (*ListModel model*), nous choisissons la classe *DefaultListModel* qui est une implémentation de l'interface *ListModel* par le biais d'un vecteur. Il est ainsi possible d'instancier le *ListModel* d'agir sur le modèle (ajout, suppression d'éléments) et de l'enregistrer auprès du composant adéquat grâce à **setModel**( ).

Le listing suivant illustre la mise en oeuvre de *DefaultListModel( )* pour un JList :

JList jList1 = **new** JList( ); DefaultListModel dlm = **new** DefaultListModel ( );	***// instanciations d'un JList et d'un modèle.***
dlm.addElement ("un"); dlm.addElement ("deux"); dlm.addElement ("trois"); dlm.addElement ("quatre");	***// actions d'ajout d'éléments dans le modèle.***
jList1.setModel(dlm);	***// enregistrement du modèle pour le JList.***
dlm.removeElementAt(1); dlm.removeRange(0,2); dlm.add(0,"Toto");	***// actions de suppression et d'ajout d'éléments dans le modèle.***

Comparaison awt.List et swing.JList

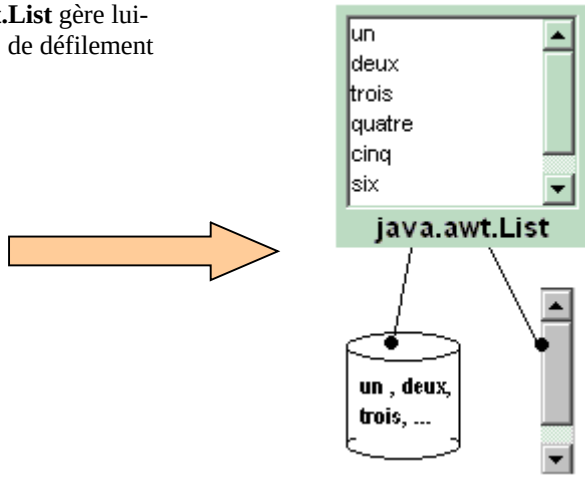


Une IHM dans laquelle, le bouton Ajouter, insère 7 chaînes dans chacun des deux composants.

L'apparence est la même lors de l'affichage des données dans chacun des composants, dans le code il y a une totale différence de gestion entre le composant List et le composant Jlist.

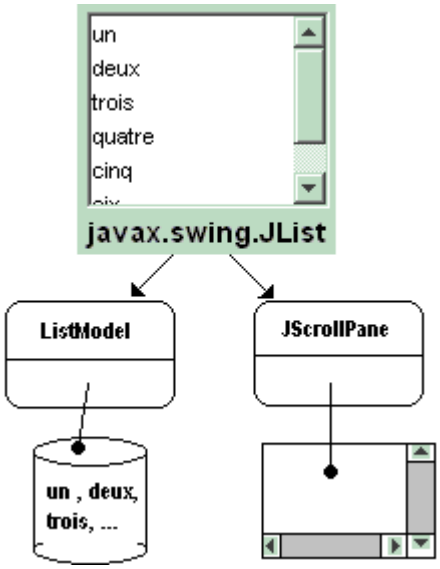
Comme en Delphi le composant **java.awt.List** gère lui-même le stockage des données, et la barre de défilement verticale.

```
List list1 = new List();
list1.add("un");
list1.add("deux");
list1.add("trois");
list1.add("quatre");
list1.add("cinq");
list1.add("six");
list1.add("sept");
```



Le composant **javax.swing.Jlist** délègue le stockage des données à un **modèle** et la gestion de la barre de défilement verticale à un autre composant dédié : un **javax.swing.JscrollPane**.

```
JList jList1 = new JList();
DefaultListModel dlm = new DefaultListModel();
JScrollPane jScrollPane1 = new JScrollPane();
jList1.setModel(dlm);
jScrollPane1.getViewport().add(jList1);
dlm.addElement("un");
dlm.addElement("deux");
dlm.addElement("trois");
dlm.addElement("quatre");
dlm.addElement("cinq");
dlm.addElement("six");
dlm.addElement("sept");
```



Le composant javax.swing. JTextPane

Soit le code suivant :

```
Style styleTemporaire;
StyleContext leStyle = new StyleContext();
Style parDefaut = leStyle.getStyle(StyleContext.DEFAULT_STYLE);
Style styleDuTexte = leStyle.addStyle("DuTexte1", parDefaut);
StyleConstants.setFontFamily(styleDuTexte, "Courier New");
StyleConstants.setFontSize(styleDuTexte, 18);
StyleConstants.setForeground(styleDuTexte, Color.red);
styleTemporaire = leStyle.addStyle("DuTexte2", styleDuTexte);
StyleConstants.setFontFamily(styleTemporaire, "Times New Roman");
StyleConstants.setFontSize(styleTemporaire, 10);
StyleConstants.setForeground(styleTemporaire, Color.blue);
styleTemporaire = leStyle.addStyle("DuTexte3", styleDuTexte);
StyleConstants.setFontFamily(styleTemporaire, "Arial Narrow");
StyleConstants.setFontSize(styleTemporaire, 14);
StyleConstants.setBold(styleTemporaire, true);
StyleConstants.setForeground(styleTemporaire, Color.magenta);
DefaultStyledDocument  format = new DefaultStyledDocument(leStyle);
```

Caractérisation du style n°1 du document

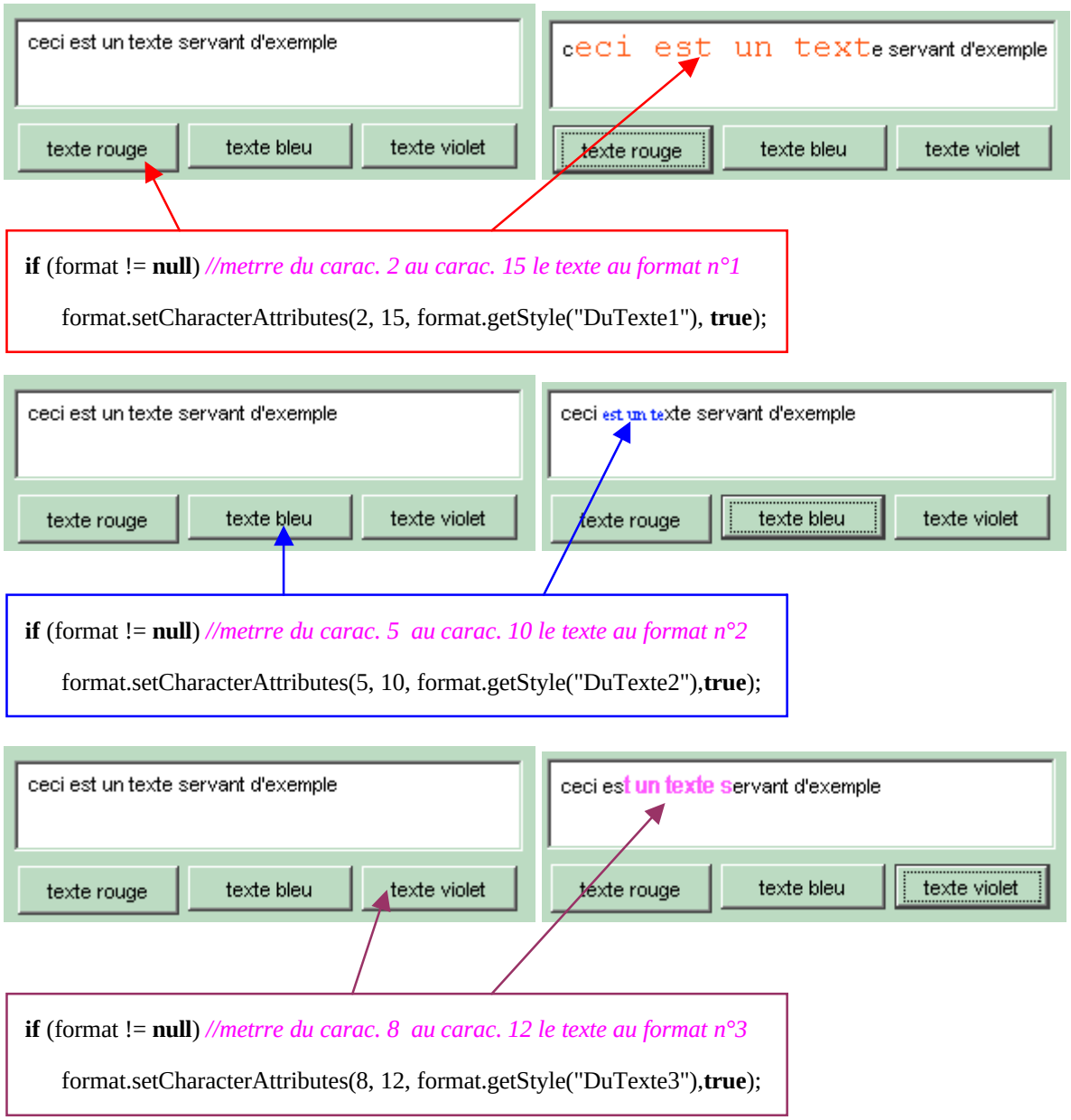
Caractérisation du style n°2 du document

Caractérisation du style n°3 du document

Le document gère les styles du JtextPane.

```
jTextPane1.setDocument(format);
```

Un composant de classe **JTextPane** est chargé d'afficher du texte avec plusieurs styles d'attributs (police, taille, couleur,…), la gestion proprement dite des styles est déléguée à un objet de classe DefaultStyledDocument que nous avons appelé **format** (gestion MVC) :

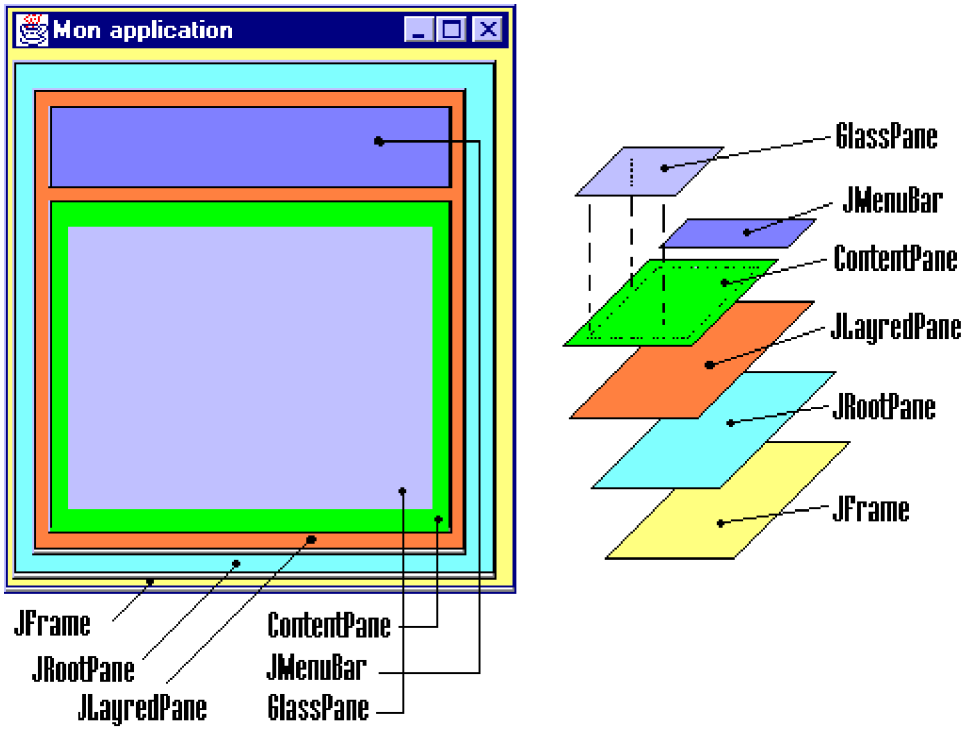


Chaque bouton lance l'application d'un des trois styles d'attributs à une partie du texte selon le modèle de code suivant :

- SetCharacterAttributes (<n° cardébut>,<n° carfin>,< le style>,**true**);
- ensuite automatiquement, le **JTextPane** informé par l'objet format qui gère son modèle de style de document, affiche dans l'image de droite le changement du style .

En Java le JFrame est un conteneur de composants (barre de menus, boutons etc...) qui dispose de 4 niveaux de superposition d'objets à qui est déléguée la gestion du contenu du JFrame.

Système de conteneur pour afficher dans une Fenêtre ou une Applet



Notons que le **JRootPane**, le **JLayredPane** et le **GlassPane** sont utilisés par Swing pour implémenter le look and feel, ils n'ont donc pas à être considérés dans un premier temps par le développeur, la couche qui nous intéresse afin de déposer un composant sur une fenêtre JFrame est la couche **ContentPane** instanciation de la classe Container. Les rectangles colorés imbriqués ci-haut, sont dessinés uniquement à titre pédagogique afin d'illustrer l'architecture en couche, ils ne représentent pas des objets visuels dont les tailles seraient imbriquées. En effet le GlassPane bien que dessiné plus petit (pour mieux le situer) prend par exemple toute la taille du Contentpane.

Swing instancie **automatiquement** tous ces éléments dès que vous instanciez un JFrame (à part JMenuBar qui est facultatif et qu'il faut instancier manuellement).

Pour ajouter des composants à un JFrame, il faut les ajouter à son objet ContentPane (la référence de l'objet est obtenu par la méthode getContentPane( ) du JFrame).

Exemple d'ajout d'un bouton à une fenêtre

Soit à ajouter un bouton de la classe des JButton sur une fenêtre de la classe des JFrame :

JFrame LaFenetre = **new** JFrame( ) ; *// instanciation d'un JFrame*
JButton UnBouton = **new** JButton( ) ; *// instanciation d'un JButton*
Container ContentPane = LaFenetre.**getContentPane**( ) ; *// obtention de la référence du contentPane du JFrame*
....
ContentPane.setLayout(**new** XYLayout( )); *// on choisi le layout manager du ContentPane*
ContentPane.add(UnBouton) ; *// on dépose le JButton sur le JFrame à travers son ContentPane*
....

Attention : avec AWT le dépôt du composant s'effectue directement sur le conteneur. Il faut en outre éviter de mélanger des AWT et des Swing sur le même conteneur.

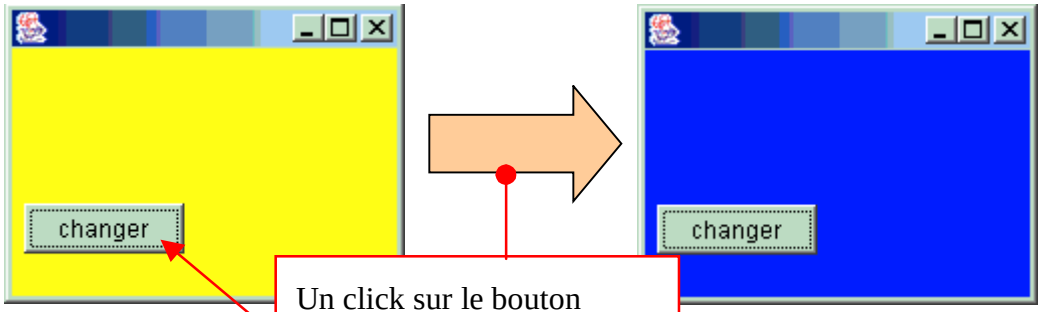
AWT	**Swing**
Frame LaFenetre = **new** Frame( ) ; **Button** UnBouton = **new** Button( ) ; LaFenetre.add(UnBouton) ;	**JFrame** LaFenetre = **new** JFrame( ) ; **JButton** UnBouton = **new** JButton( ) ; **Container** ContentPane = LaFenetre.**getContentPane**( ) ; ContentPane.add(UnBouton) ;

Exemple IHM - JFrame de Swing

Soit l'IHM suivante composée d'une fiche **Fenetre** de classe **JFrame**, d'un bouton **jButton1** de classe **JButton**.

L'IHM réagit uniquement au click de souris :

- Le **jButton1** de classe **JButton** réagit au simple click et fait passer le fond de la fiche à la couleur bleu.
- La fiche de classe **JFrame** est sensible au click de souris pour sa fermeture et arrête l'application.



Un click sur le bouton "**changer**", fait changer la couleur du fond de la fiche.

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

public class Fenetre extends JFrame {
 Container contentPane;
 JButton jButton1 = new JButton( );

 public Fenetre() {
  enableEvents (AWTEvent.WINDOW_EVENT_MASK);
  contentPane =  this.getContentPane( );
  jButton1.setBounds(new Rectangle(10, 80, 80, 25));
  jButton1.setText("changer");
  jButton1.addMouseListener(   new java.awt.event.MouseAdapter( )
                               {
                                  public void mouseClicked(MouseEvent e)  {
                                     GestionnaireClick(e);
                                  }
                               }
  );
  contentPane.setLayout(null);
  this.setSize(new Dimension(200, 150));
  this.setTitle("");
  contentPane.setBackground(Color.yellow);
  contentPane.add(jButton1, null);
 }
```

On récupère dans la variable contentPane, la référence sur le Container renvoyée par la méthode getContentPane.

Version avec une classe anonyme d'écouteur dérivant des MouseAdapter. (identique Awt)

```
protected void processWindowEvent(WindowEvent e) {
  super.processWindowEvent(e);
  if (e.getID( ) == WindowEvent.WINDOW_CLOSING)
   System.exit(100);
 }

 void GestionnaireClick(MouseEvent e) {
   this.contentPane.setBackground(Color.blue);
 }
}

public class  ExoSwing   {

    public static void  main ( String [ ] x )   {
      Fenetre fen  =  new  Fenetre ( );
    }
}
```

Fermeture de la JFrame des Swing identique à la Frame des Awt.

La bibliothèque Swing apporte une amélioration de confort dans l'écriture du code fermeture d'une fenêtre de classe JFrame en ajoutant dans la classe JFrame une nouvelle méthode :

public void setDefaultCloseOperation(**int** operation)

Cette méthode indique selon la valeur de son paramètre de type int, quelle opération doit être effectuée lors que l'on ferme la fenêtre.

Les paramètres possibles sont au nombre quatre et sont des champs static de classe :

WindowConstants.DO_NOTHING_ON_CLOSE
WindowConstants.HIDE_ON_CLOSE
WindowConstants.DISPOSE_ON_CLOSE

JFrame.EXIT_ON_CLOSE

Dans la classe : **WindowConstants**

Dans la classe : **JFrame**

C'est ce dernier que nous retenons pour faire arrêter l'exécution del'application lors de la fermeturede la fenêtre. Il suffit d'insérer dans le constructeur de fenêtre la ligne qui suit :

setDefaultCloseOperation(**JFrame**.EXIT_ON_CLOSE);

Reprise du code de Fenetre avec cette modification spécifique aux JFrame

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

public class Fenetre extends JFrame {
 Container contentPane;
 JButton jButton1 = new JButton( );

 public Fenetre( ) {
  contentPane = this.getContentPane( );
```

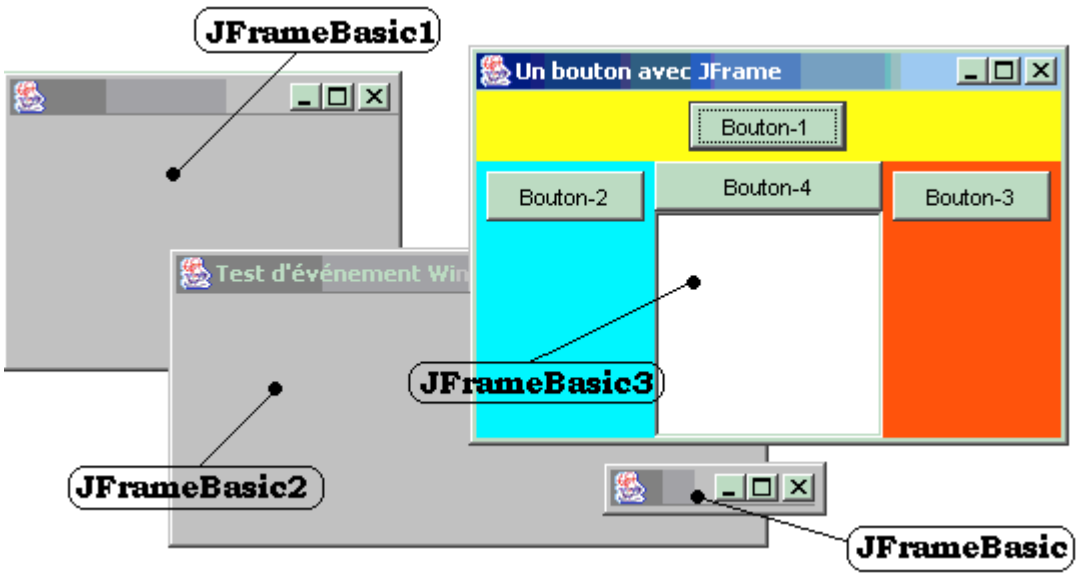
```
jButton1.setBounds(new Rectangle(10, 80, 80, 25));
jButton1.setText("changer");
jButton1.addMouseListener (   new java.awt.event.MouseAdapter( )
                              {
                                 public void mouseClicked(MouseEvent e)  {
                                    GestionnaireClick(e);
                                 }
                              }
);
contentPane.setLayout(null);
this.setSize(new Dimension(200, 150));
this.setTitle("");
contentPane.setBackground(Color.yellow);
contentPane.add(jButton1, null);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}

void GestionnaireClick(MouseEvent e)  {
  this.setBackground(Color.blue);
 }
}
```

Fermeture de la fenêtre spécifique à la classe JFrame des Swing.

Soit l'IHM suivante composée de quatre fiches de classe **JFrame** nommées **JFrameBasic, JFrameBasic1, JFrameBasic2, JFrameBasic3**. Elle permet d'explorer le comportement d'événements de la classe WindowEvent sur une JFrame ainsi que la façon dont une JFrame utilise un layout

- Le **JFrameBasic2** de classe **JFrame** réagit à la fermeture, à l'activation et à la désactivation.
- Le **JframeBasic3** de classe **JFrame** ne fait que présenter visuellement le résultat d'un BorderLayout.



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

  public class JFrameBasic extends JFrame   {
    JFrameBasic( )     {
      this.setVisible(true);
    }
 }

public class JFrameBasic1 extends JFrame {
  JFrameBasic1( )    {
    this.setVisible(true);
    this.setBounds(100,100,200,150);
  }
}


public class JFrameBasic2 extends JFrame {
  JFrameBasic2( )    {
    this.setVisible(true);
    this.setBounds(200,200,300,150);
    this.setTitle("Test d'événement Window");
    setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
  }
  protected void processWindowEvent(WindowEvent e)    {
    super.processWindowEvent(e);
    if (e.getID( ) == WindowEvent.WINDOW_CLOSING)     {
      System.out.println("JFrameBasic2 / WINDOW_CLOSING = "+WindowEvent.WINDOW_CLOSING);
      dispose ( );
    }
    if (e.getID( ) == WindowEvent.WINDOW_CLOSED)     {
      System.out.print("JFrameBasic2 / WINDOW_CLOSED = "+WindowEvent.WINDOW_CLOSED);
      System.out.println(" => JFrameBasic2 a été détruite !");
    }
    if (e.getID( ) == WindowEvent.WINDOW_ACTIVATED)
     System.out.println("JFrameBasic2 / WINDOW_ACTIVATED = " + WindowEvent.WINDOW_ACTIVATED );
    if (e.getID( ) == WindowEvent.WINDOW_DEACTIVATED)
     System.out.println("JFrameBasic2 / WINDOW_DEACTIVATED = "+WindowEvent.WINDOW_DEACTIVATED);
  }

  public class JFrameBasic3 extends JFrame     {
    JButton Bout1 = new JButton("Bouton-1");
    JButton Bout2 = new JButton("Bouton-2");
    JButton Bout3 = new JButton("Bouton-3");
    JButton Bout4 = new JButton("Bouton-4");
    JTextArea jTextArea1 = new JTextArea();

    JFrameBasic3( )    {
      JPanel Panel1 = new JPanel( );
```

```
        JPanel Panel2 = new JPanel( );
        JPanel Panel3 = new JPanel( );
        JPanel Panel4 = new JPanel( );
        JScrollPane jScrollPane1 = new JScrollPane( );
        jScrollPane1.setBounds(20,50,50,40);
        this.setBounds(450,200,300,200);
        jScrollPane1.getViewport( ).add(jTextArea1);
        this.setTitle("Un bouton avec JFrame");
        Bout1.setBounds(5, 5, 60, 30);
        Panel1.add(Bout1);
        Bout2.setBounds(10, 10, 60, 30);
        Panel2.add(Bout2);
        Bout3.setBounds(10, 10, 60, 30);
        Panel3.add(Bout3);
        Bout4.setBounds(10, 10, 60, 30);
        Panel4.setLayout(new BorderLayout( ));
        Panel4.add(Bout4, BorderLayout.NORTH);
        Panel4.add(jScrollPane1, BorderLayout.CENTER);
        Panel1.setBackground(Color.yellow);
        Panel2.setBackground(Color.cyan);
        Panel3.setBackground(Color.red);
        Panel4.setBackground(Color.orange);
        this.getContentPane().setLayout(new BorderLayout( )); //specifique JFrame
        this.getContentPane().add(Panel1, BorderLayout.NORTH); //specifique JFrame
        this.getContentPane().add(Panel2, BorderLayout.WEST); //specifique JFrame
        this.getContentPane().add(Panel3, BorderLayout.EAST); //specifique JFrame
        this.getContentPane().add(Panel4, BorderLayout.CENTER); //specifique JFrame
        this.setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

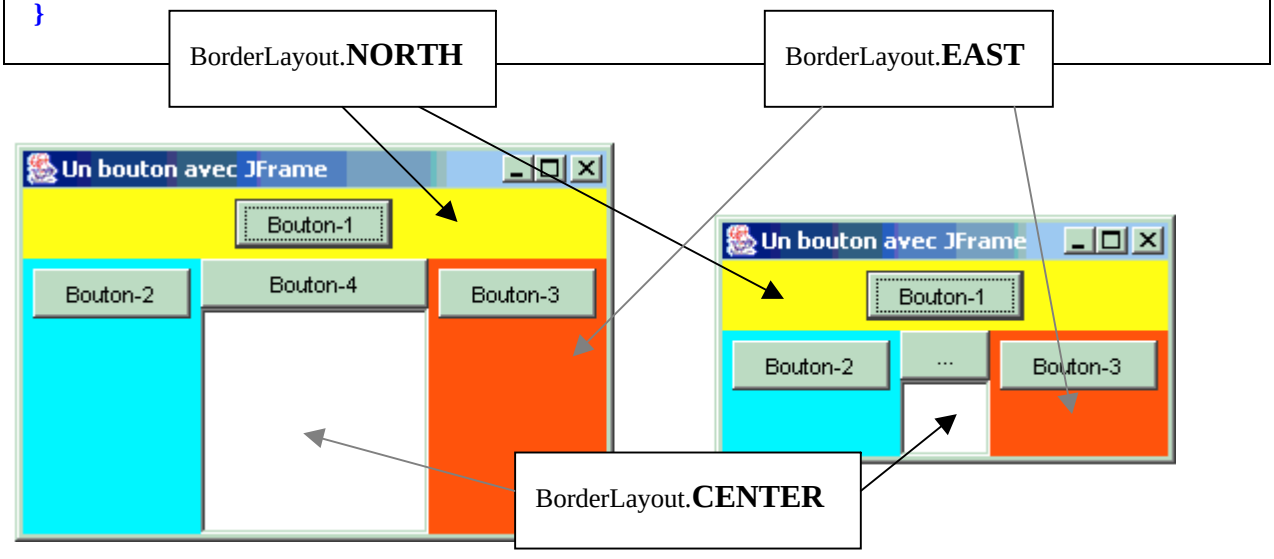
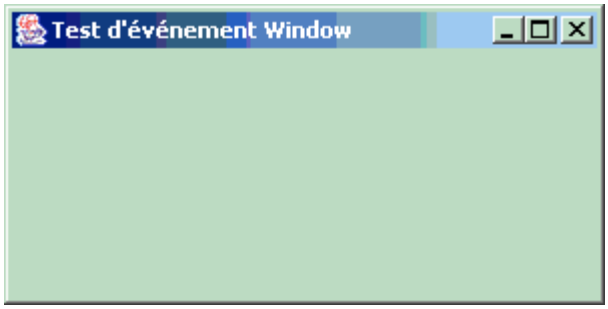


fig-repositionnement automatique des quatre Jpanel grâce au BorderLayout

Au démarrage voici les affichages consoles (la JFrameBasic2 est en arrière plan)

> *JFrameBasic2 / WINDOW_ACTIVATED = 205*
> *JFrameBasic2 / WINDOW_DEACTIVATED = 206*

En cliquant sur JFrameBasic2 elle passe au premier plan



voici l'affichages console obtenu :

> *JFrameBasic2 / WINDOW_ACTIVATED = 205*

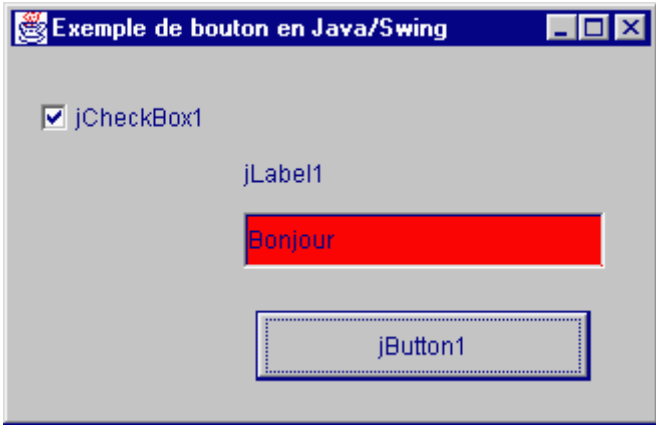
En cliquant sur le bouton de fermeture de JFrameBasic2 elle se ferme mais les autres fenêtres restent, voici l'affichages console obtenu :

JFrameBasic2 / WINDOW_CLOSING = 201
JFrameBasic2 / WINDOW_DEACTIVATED = 206
JFrameBasic2 / WINDOW_CLOSED = 202 => JFrameBasic2 a été détruite !

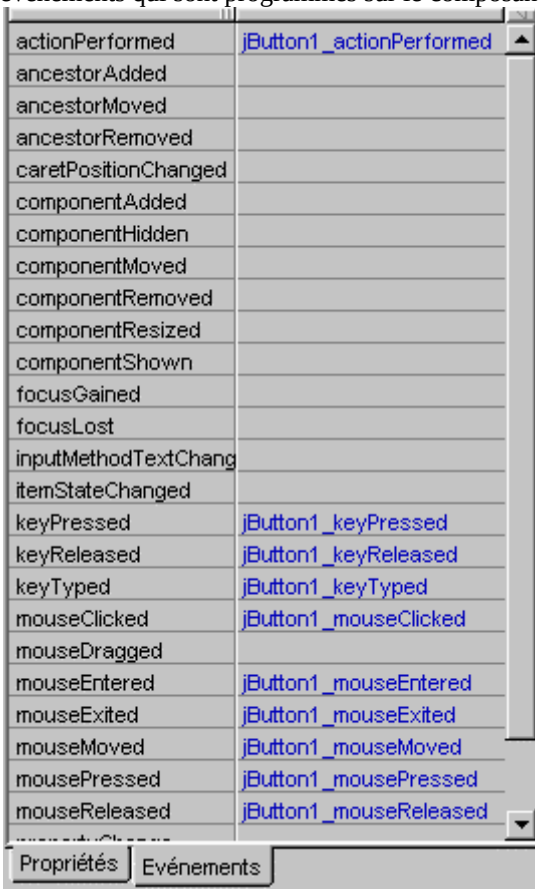
Exemple - JButton de Swing

Objectif : Application simple Java utilisant les événements de souris et de clavier sur un objet de classe JButton.

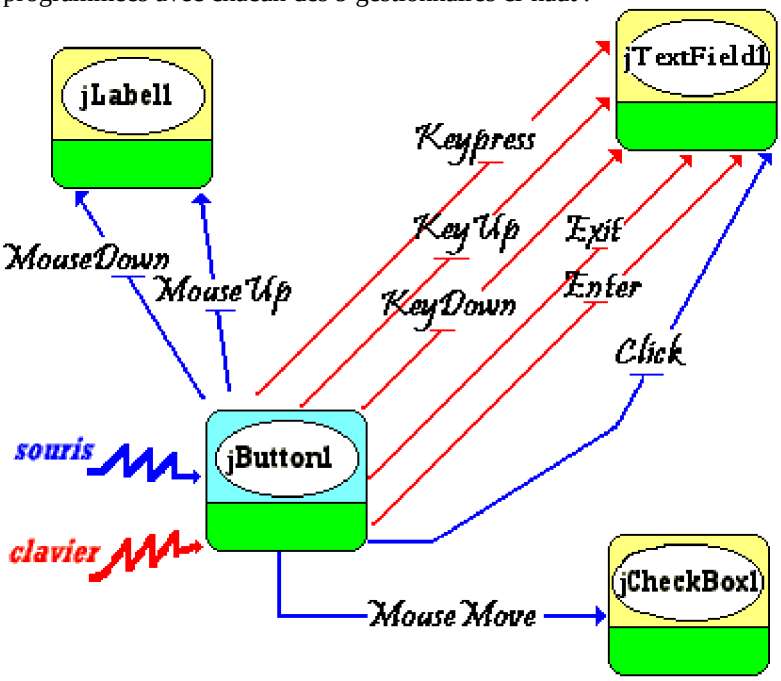
La fenêtre comporte un bouton (JButton jButton1), une étiquette (**JLabel** jLabel1), une case à cocher (**JCheckBox** jCheckBox1) et un éditeur de texte mono-ligne (**JTextField** jTextField1) :



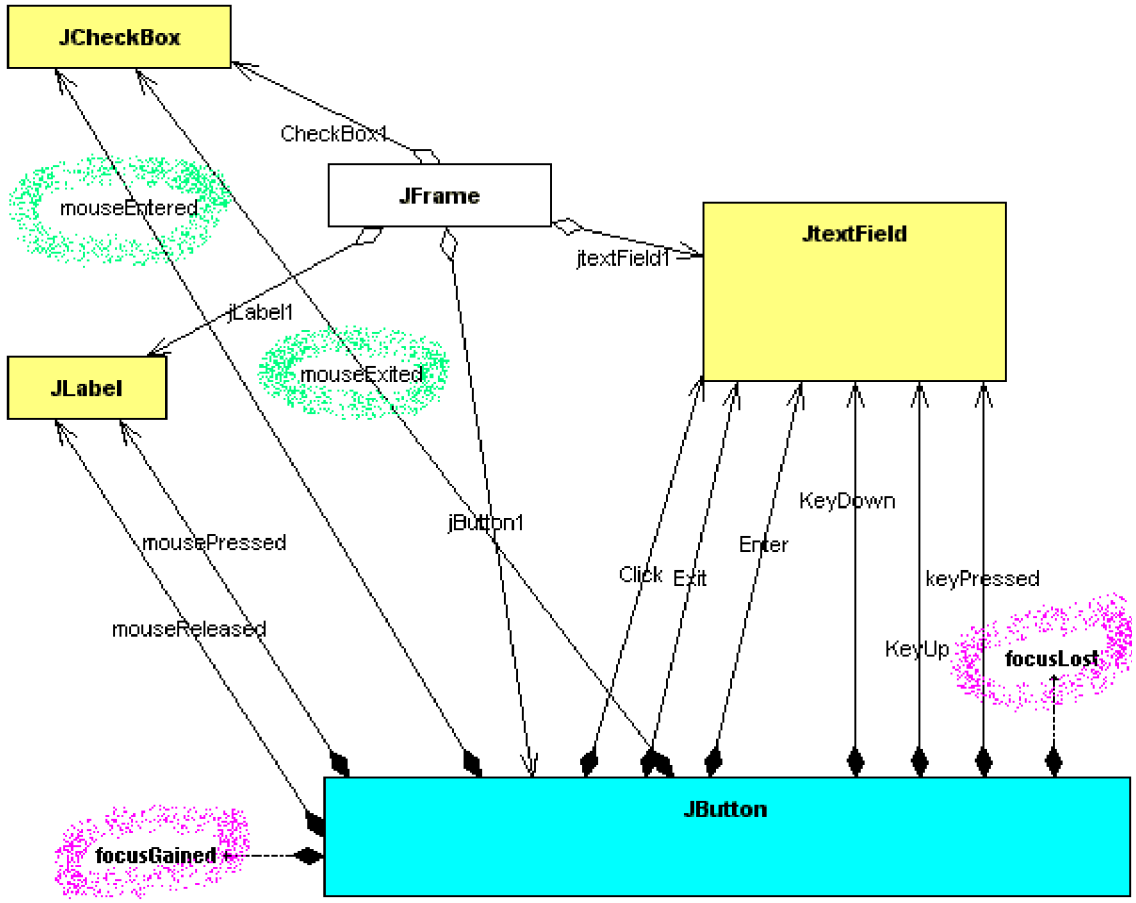
Voici les 10 gestionnaires d'événements qui sont programmés sur le composant jButton de classe **JButton**:



Voici le diagramme événementiel des actions de souris et de clavier sur le bouton jButton1. Ces 9 actions sont programmées avec chacun des 9 gestionnaires ci-haut :

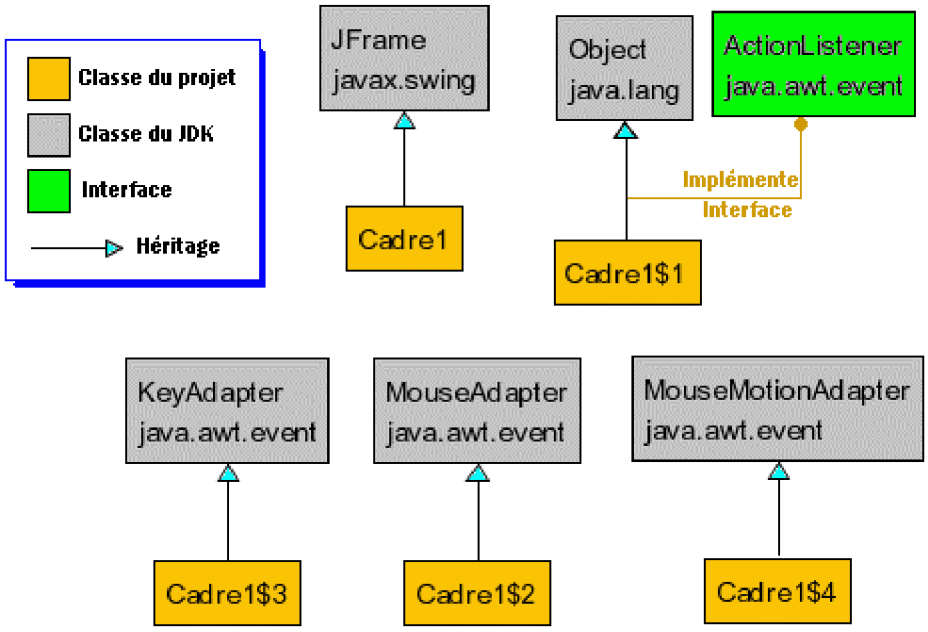


Les actions exit et enter sont représentées en Java par les événements focuGained et focusLost pour le clavier et par les événements mouseEntered et mouseExited pour la souris,. Il a été choisi de programmer les deux événements de souris dans le code ci-dessous.



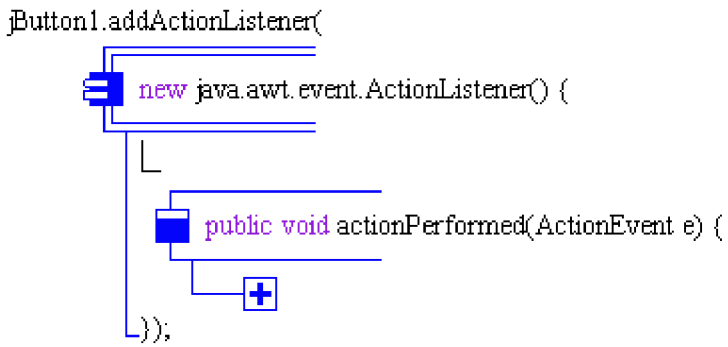
En Java

Comme en java tous les événements sont interceptés par des objets écouteurs, ci-dessous nous donnons les diagrammes UML des classes utilisées par le programme qui est proposé :

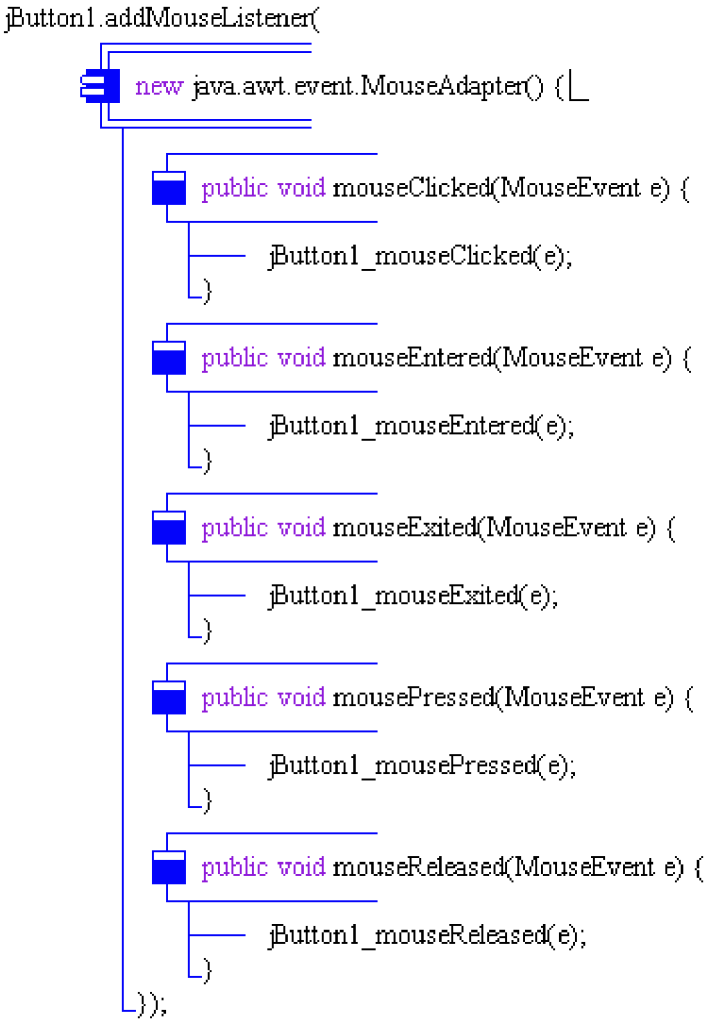


Rappelons que les classes Cadre1$1, Cadre1$2, ... sont la notation des classes anonymes créées lors de la déclaration de l'écouteur correspondant, Java 2 crée donc dynamiquement un objet écouteur interne (dont la référence n'est pas disponible). Ci-dessous les diagrammes jGrasp des quatre classes anonymes cadre1$1, Cadre1$2, Cadre1$3 et Cadre1$4 :

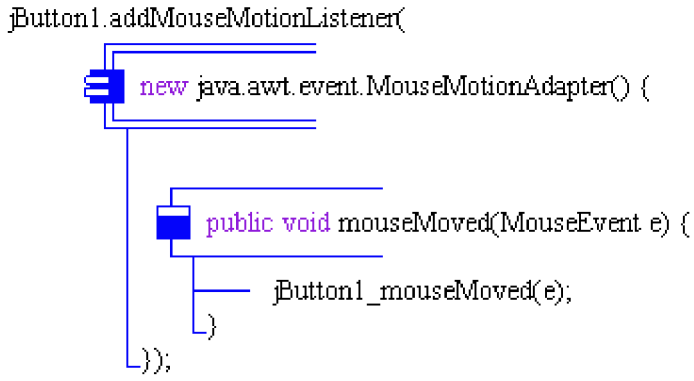
Cadre1$1:



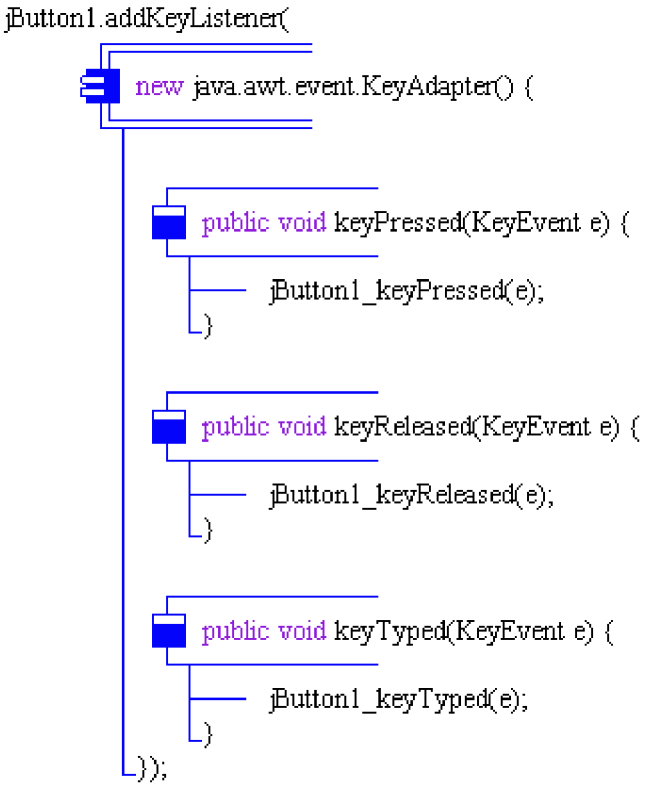
Cadre1$2:



Cadre1$4:



Cadre1$3:



Enfin pour terminer, voici le listing Java/Swing complet de la classe représentant la fenêtre :

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

public class Cadre1 extends JFrame {
  JButton jButton1 = new JButton();
  JTextField jTextField1 = new JTextField();
  JLabel jLabel1 = new JLabel();
  JCheckBox jCheckBox1 = new JCheckBox();

  //Construire le cadre
  public Cadre1() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try  {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  //Initialiser le composant
  private void jbInit() throws Exception  {
    jButton1.setText("jButton1");
    jButton1.addActionListener(new java.awt.event.ActionListener() {

      public void actionPerformed(ActionEvent e) {
```

```
      jButton1_actionPerformed(e);
    }
  });
  jButton1.addMouseListener(new java.awt.event.MouseAdapter() {

    public void mouseClicked(MouseEvent e) {
      jButton1_mouseClicked(e);
    }

    public void mouseEntered(MouseEvent e) {
      jButton1_mouseEntered(e);
    }

    public void mouseExited(MouseEvent e) {
      jButton1_mouseExited(e);
    }

    public void mousePressed(MouseEvent e) {
      jButton1_mousePressed(e);
    }

    public void mouseReleased(MouseEvent e) {
      jButton1_mouseReleased(e);
    }
  });

  jButton1.addKeyListener(new java.awt.event.KeyAdapter() {

    public void keyPressed(KeyEvent e) {
      jButton1_keyPressed(e);
    }

    public void keyReleased(KeyEvent e) {
      jButton1_keyReleased(e);
    }

    public void keyTyped(KeyEvent e) {
      jButton1_keyTyped(e);
    }
  });

  jButton1.addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {

    public void mouseMoved(MouseEvent e) {
      jButton1_mouseMoved(e);
    }
  });
  this.getContentPane().setLayout(null);
  this.setSize(new Dimension(327, 211));
  this.setTitle("Exemple de bouton en Java/Swing");
  jTextField1.setText("jTextField1");
  jTextField1.setBounds(new Rectangle(116, 82, 180, 28));
  jLabel1.setText("jLabel1");
  jLabel1.setBounds(new Rectangle(116, 49, 196, 26));
  jCheckBox1.setText("jCheckBox1");
  jCheckBox1.setBounds(new Rectangle(15, 22, 90, 25));
  this.getContentPane().add(jTextField1, null);
  this.getContentPane().add(jButton1, null);
  this.getContentPane().add(jCheckBox1, null);
  this.getContentPane().add(jLabel1, null);
}
```

```
//Remplacé (surchargé) pour pouvoir quitter lors de System Close
protected void processWindowEvent(WindowEvent e) {
  super.processWindowEvent(e);
  if(e.getID() == WindowEvent.WINDOW_CLOSING) {
    System.exit(0);
  }
}

void jButton1_mouseMoved(MouseEvent e) {
  jCheckBox1.setSelected(true);
}

void jButton1_keyPressed(KeyEvent e) {
 jTextField1.setText("Bonjour");
}

void jButton1_keyReleased(KeyEvent e) {
 jTextField1.setText("salut");
}

void jButton1_keyTyped(KeyEvent e) {
 jTextField1.setForeground(Color.blue);
}

void jButton1_mouseClicked(MouseEvent e) {
 jLabel1.setText("Editeur de texte");
}

void jButton1_mouseEntered(MouseEvent e) {
   jTextField1.setBackground(Color.red);
}

void jButton1_mouseExited(MouseEvent e) {
  jTextField1.setBackground(Color.green);
}

void jButton1_mousePressed(MouseEvent e) {
 jLabel1.setText("La souris est enfoncée");
}

void jButton1_mouseReleased(MouseEvent e) {
 jLabel1.setText("La souris est relachée");
}

void jButton1_actionPerformed(ActionEvent e) {
  jTextField1.setText("Toto");
}
}
```

Attention sur un click de souris l'événement :

mouseClicked est **toujours** généré que le bouton soit **activé** :

ou bien **désactivé** :

Attention sur un click de souris l'événement :

actionPerformed est généré lorsque le bouton est **activé** :

actionPerformed **n**'est **pas** généré lorsque le bouton est **désactivé** :