Université Sidi Mohamed Ben Abdellah
Faculté des Sciences Dhar El-Mehraz Fès
Département d'Informatique

Développement d'applications graphiques Java avec Swing

Par Noureddine Chenfour

2002-2015

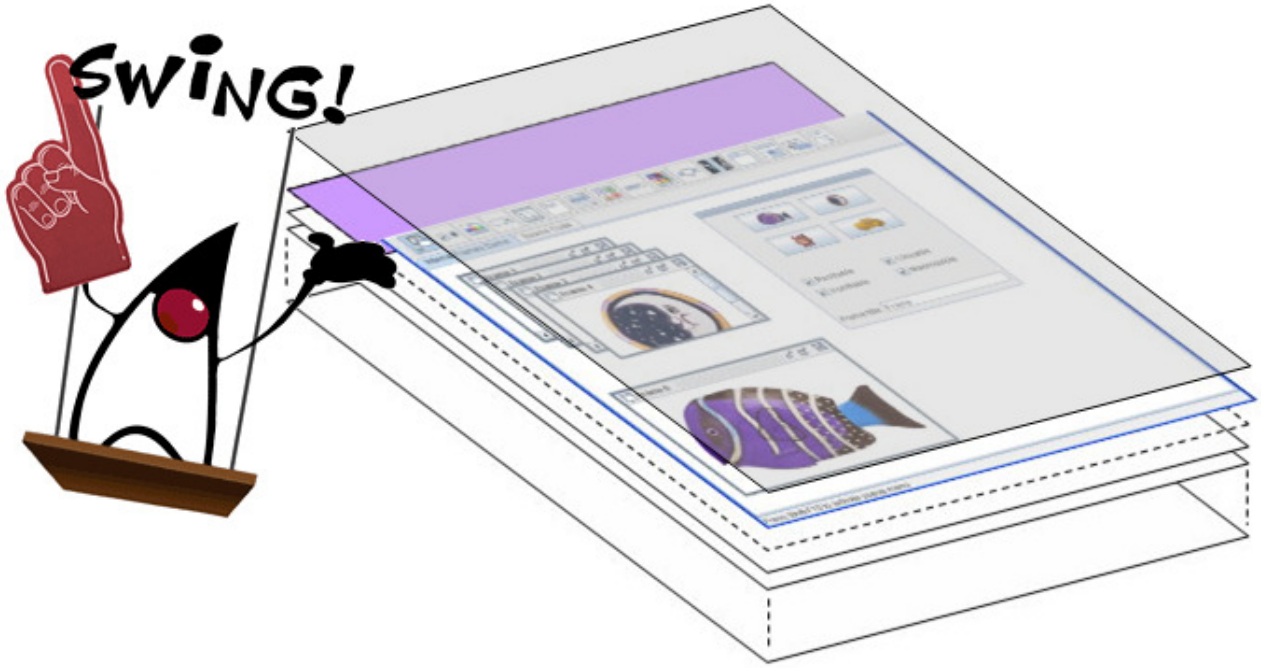
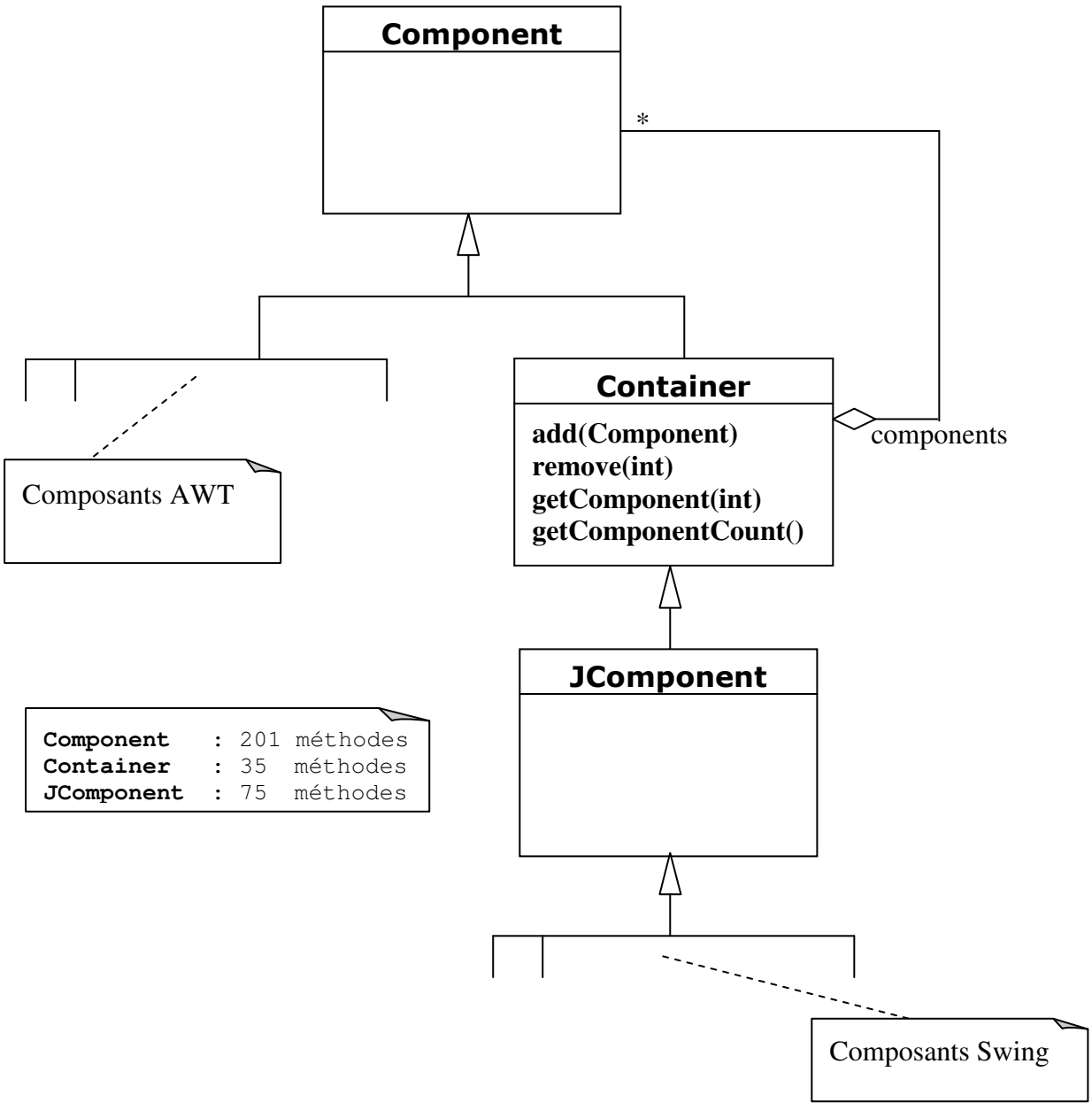


Table des Matières

Chapitre 1. Architecture de la bibliothèque Swing

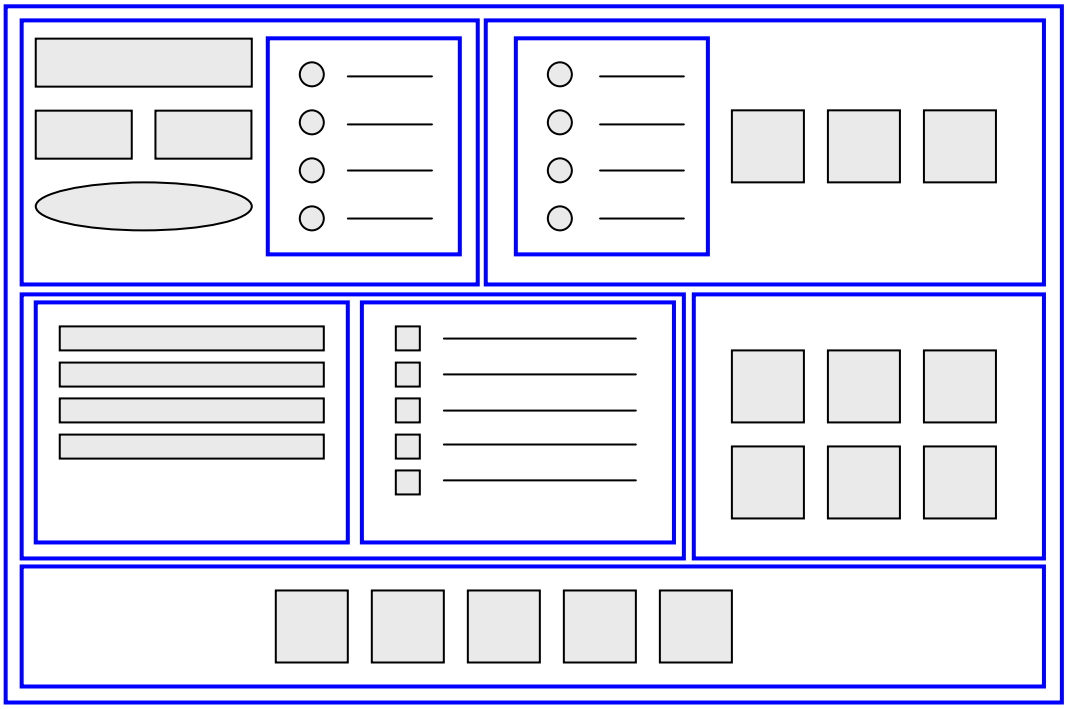
1.1 Structure de l'arbre des composants AWT/Swing : Le Design Pattern « Composite »



Swing est un kit de développement d'interfaces graphiques entièrement conforme à la norme Javabeans de SUN. Une interface utilisateur graphique peut se concevoir comme étant un assemblage de composants à l'intérieur d'un conteneur.

L'ensemble conteneurs + composants forment l'interface graphique utilisateur.

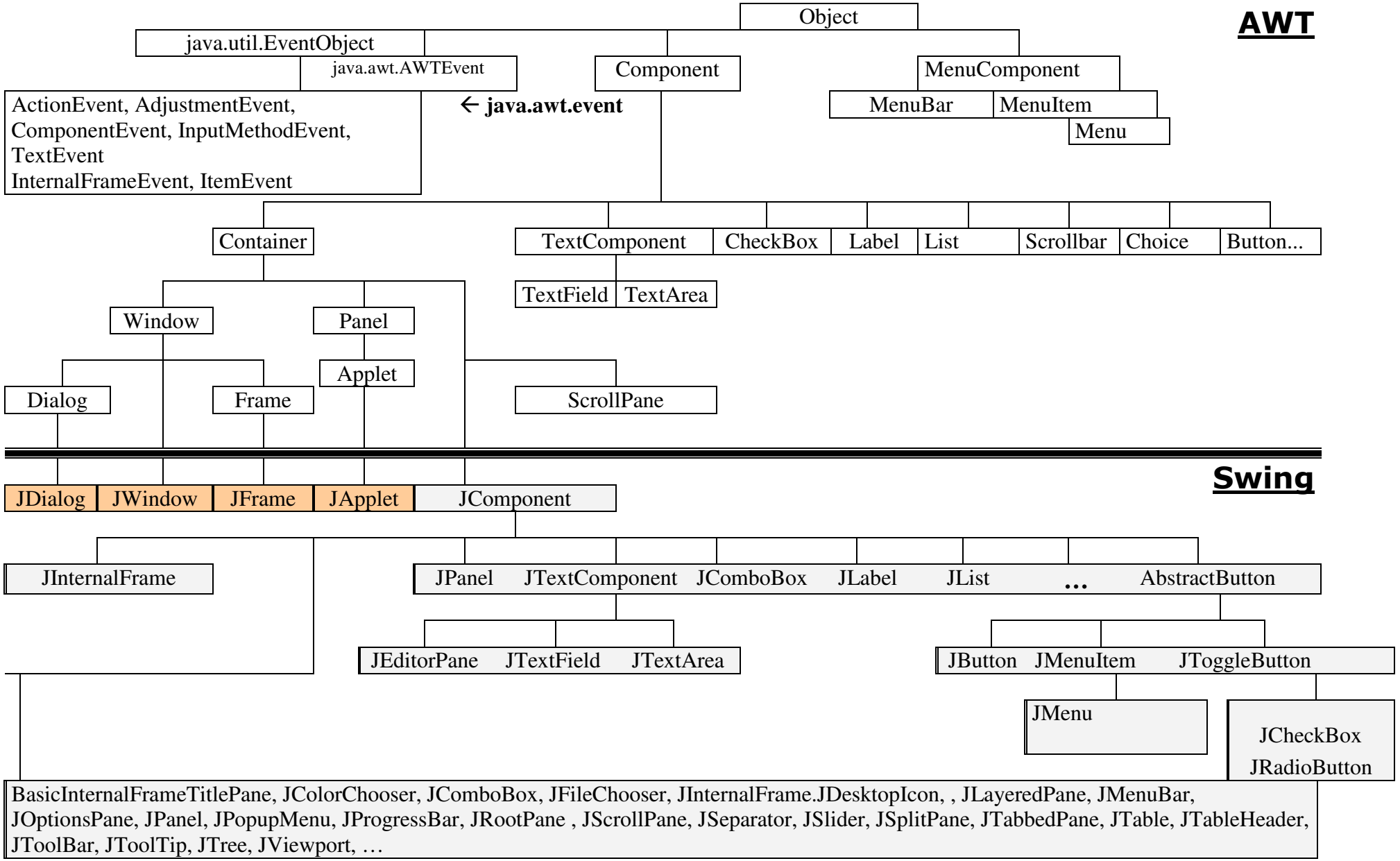
On appelle composant un objet pouvant être affiché dans une interface graphique (Bouton, zone de texte, étiquette, …), et conteneur un objet pouvant contenir des composants ou d'autres conteneurs.



La grande puissance de l'architecture mise en œuvre est qu'un conteneur est aussi un composant (Design Pattern Composite). On aura ainsi la possibilité d'imbriquer les conteneurs pour obtenir l'aspect graphique désiré.

Les conteneurs contiennent et gèrent les autres composants graphiques. Ils dérivent java.awt.Container. A l'exécution, ils apparaissent généralement sous forme de panneaux, de fenêtres ou de boites de dialogues.

1.2. Hierarchie des composants graphiques AWT et Swing



1.3 Fenêtres Swing : les conteneurs de haut niveau : JFrame, JDialog, JWindow et JApplet.

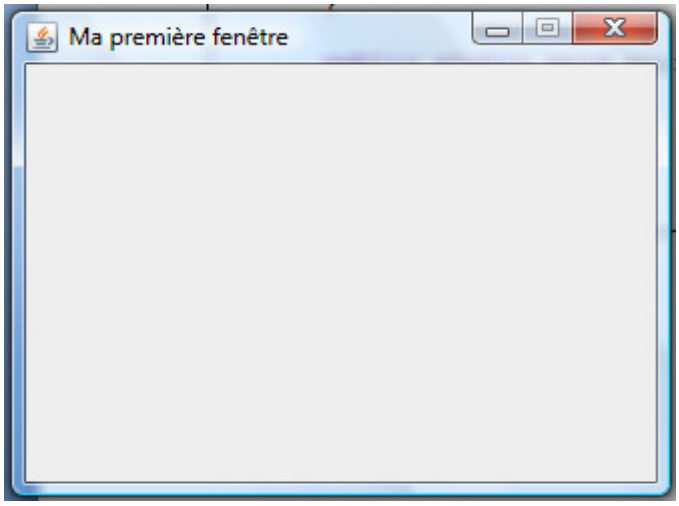
1.3.1 JFrame

C'est un conteneur de haut niveau, avec une bordure et une barre de titre. Un JFrame (cadre) possède les contrôles de fenêtre standard, tels un menu système, des boutons pour réduire ou agrandir la fenêtre et des contrôles et la redimensionner. Il peut aussi contenir une barre de menus.

Exemple :

```
import javax.swing.JFrame;

public class ExempleDeFrame extends JFrame {

	public ExempleDeFrame () {
		build();
		setVisible(true);
	}

	private void build() {
		setTitle("Ma première fenêtre");
		setSize(300, 240);
		setResizable(false);
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	}

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

Exécution :



1.3.2 JDialog

La classe JDialog offre pratiquement les mêmes services que la classe **JFrame.** Cependant, la classe **JFrame** est utilisée pour créer une fenêtre principale alors qu'une **JDialog** est une fenêtre secondaire qui peut être modale par rapport à une **JFrame**.

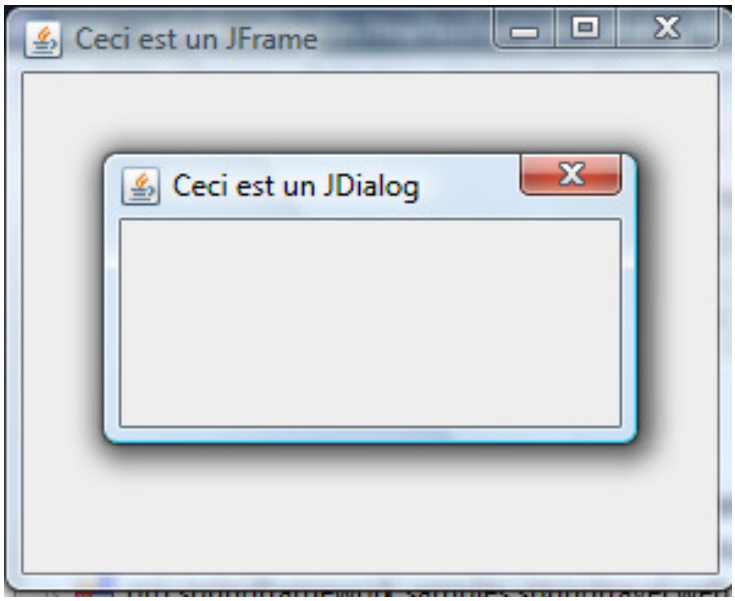
Constructeurs de la classe JDialog :

JDialog()	Crée une JDialog non modale, sans titre et sans parent
JDialog(Dialog, boolean)	Crée une JDialog modale ou non, sans titre, avec parent
JDialog(Frame,String,boolean)	Crée une JDialog modale ou non, avec un titre et un parent

Exemple :

```
import javax.swing.JDialog;
import javax.swing.JFrame;

public class ExempleDeFrame extends JFrame {

	public ExempleDeFrame () {
		super("Ceci est un JFrame");
		build();
		setVisible(true);
		JDialog dialog = new JDialog(this, true);
		dialog.setTitle("Ceci est un JDialog");
		dialog.setSize(220, 120);
		dialog.setLocationRelativeTo(this);
		dialog.setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
		dialog.setVisible(true);
	}

	private void build() {
		setSize(300, 240);
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	}

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

Exécution :

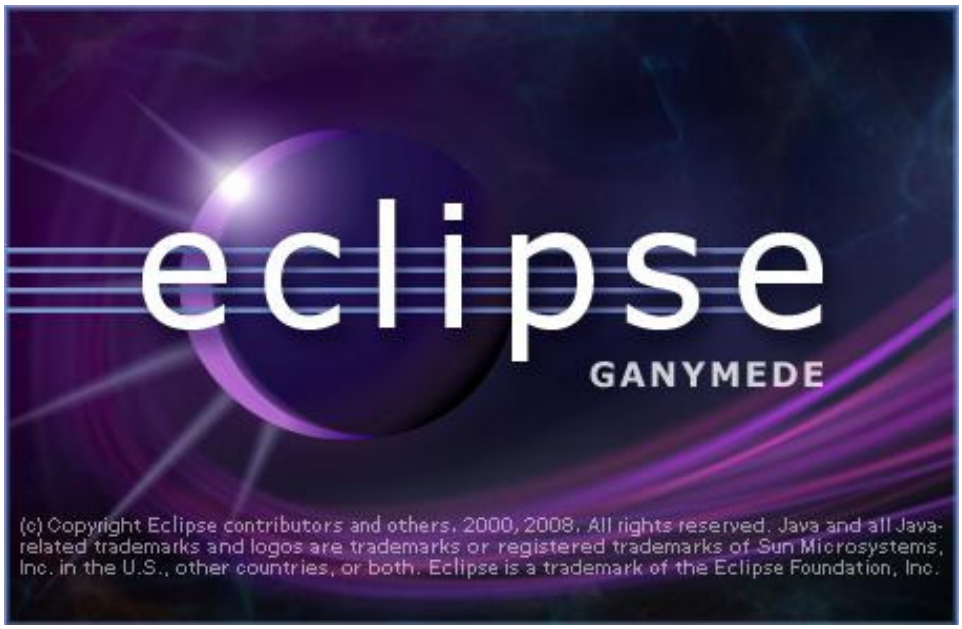


1.3.3 JWindow

La **JWindow** est un cas particulier de fenêtre. Elle ressemble à une **JFrame** mais sans bordure ni barre de titre. Cette fenêtre n'est ni redimensionnable ni déplaçable par l'utilisateur. Il n'existe pas non plus de bouton système de fermeture de l'application, c'est au développeur de programmer la fermeture de ce type de fenêtre.

JWindow est très utilisée généralement pour coder un « splash screen », cette image qui s'affiche au lancement d'une application pendant la durée de l'initialisation de cette dernière.

Exemple :



```
import java.awt.Cursor;
import javax.swing.JWindow;

public class SplashScreen extends JWindow {

	public SplashScreen() {
		build();
		setVisible(true);

		try {
			Thread.sleep(5000);
		}
		catch (Exception e) {}
	}

	private void build() {
		setSize(320, 240);
		setLocation(200, 150);
		setCursor(new Cursor(Cursor.WAIT_CURSOR));
	}

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

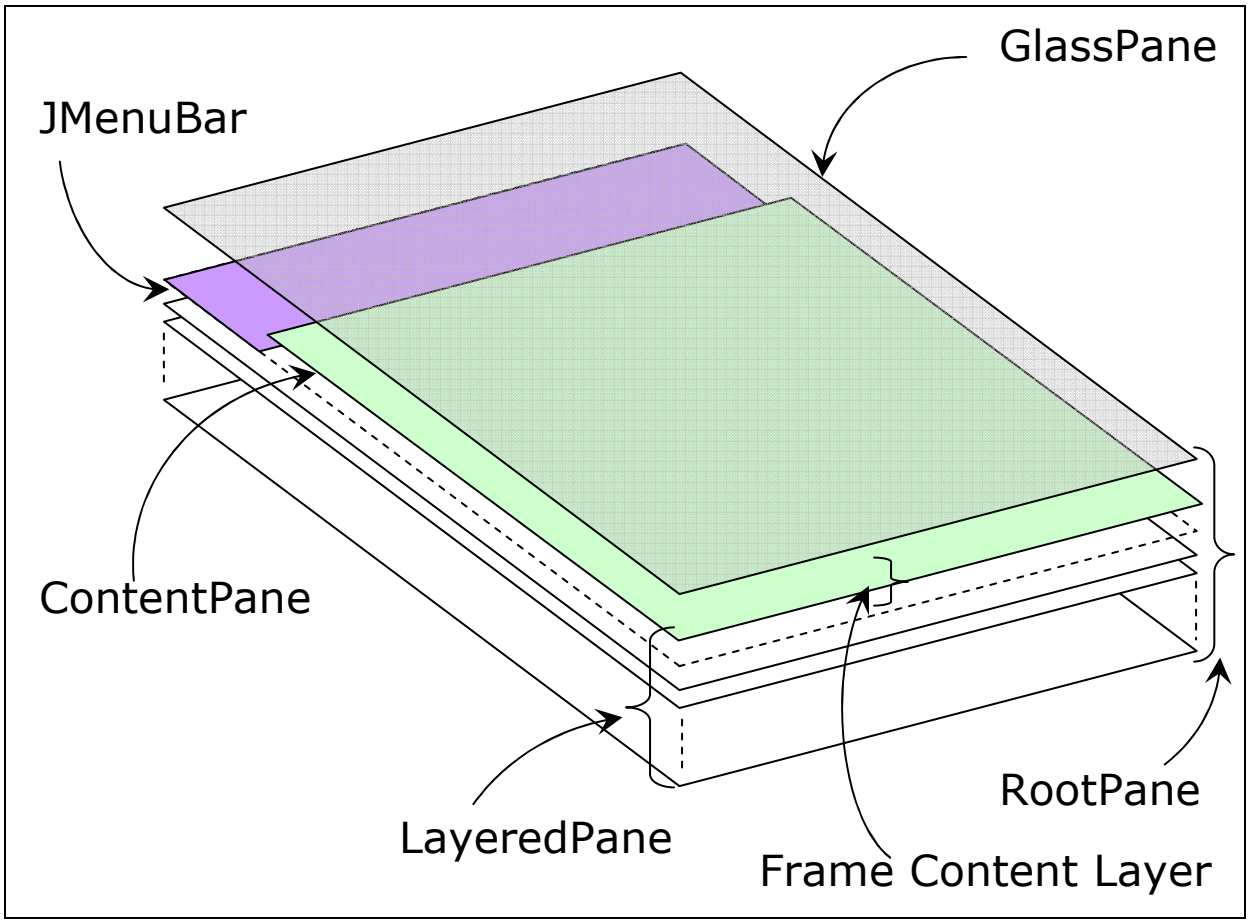
1.3.4 JApplet

La JApplet est aussi un composant de haut niveau qui ressemble dans sa structure à une JFrame et une JDialog, mais elle est conçu pour être affichée sur la fenêtre du navigateur, invoquée par l'intermédiaire de la balise <applet> du langage HTML (voir plus de détail sur les applets au chapitre 6).

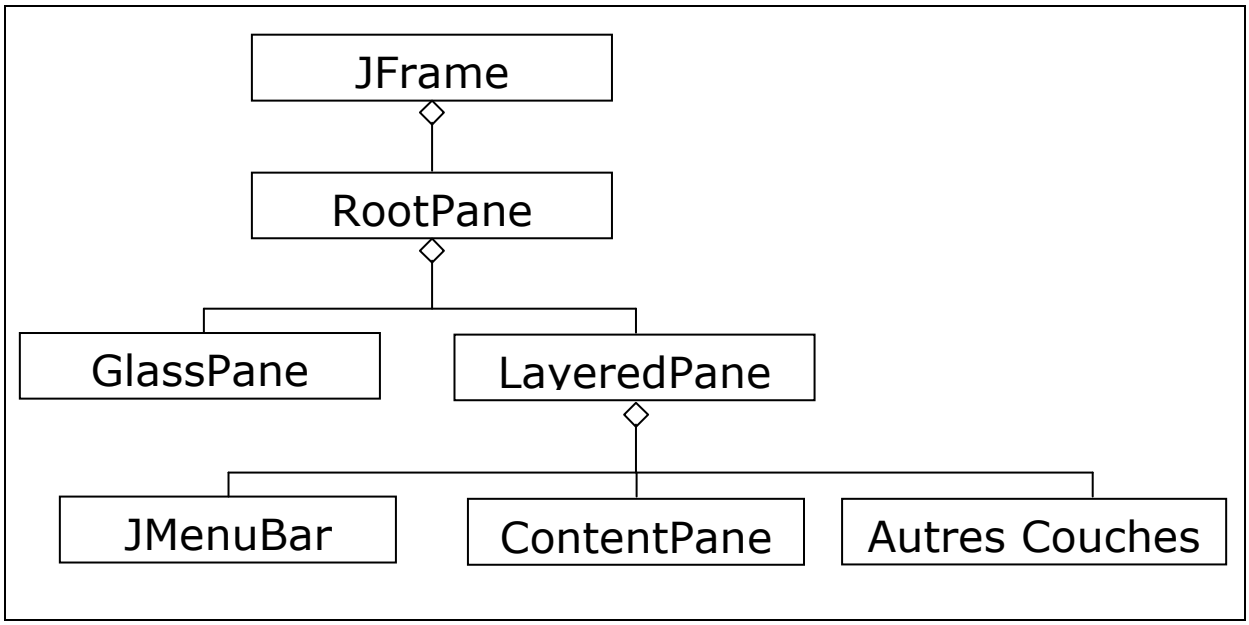
```
<html>
<body>
	...
	<applet  code    ="com.acs.applets .NotreApplet.class"
		 height  ="150"
		 width   ="600">
	</applet>
	...
</body>
</html>
```

1.4 Structure d'une fenêtre Swing

Toutes les fenêtres Swing : **JFrame, JDialog, JWindow** et **JApplet** et même la fenêtre interne **JInternalFrame** (voir plus loin), sont conçues conformément à l'achitecture suivante (à l'exeption de la JWindow qui ne dispose pas du composant JMenuBar).



L'architecture de la JFrame peut aussi être vue de la manière suivante :



1.4.1 GlassPane

C'est un panneau transparent positionné au dessus du JRootPane auquel il est associé. Il permet de mettre des composants graphiques en dessus de tous ceux de l'interface. Il est aussi utilisé pour capturer les événements de la souris.

Il est accessible via les méthodes :

- getGlassPane() et
- setGlassPane(Component c)

de la fenêtre.

1.4.2 JLayeredPane

JLayeredPane est un conteneur qui peut posséder un nombre de couches quasiment illimité. Les composants contenus dans un Layer, ou une couche, sont organisés selon leur position.

On peut avoir plusieurs composants dans une même couche, dans ce cas là, ils sont aussi organisés en couches (ou sous couche). Il seront ainsi caractérisé par leur position dans la couche ; les positions sont numérotées à partir de 0 (0, 1, 2, …) du composant le plus haut vers le plus bas dans la même couche.

En ce qui concerne les couches (ou layers), il sont ordonnées (ou numérotées) du plus bas vers le plus haut, en commençant par des valeurs négatives pour les couches les plus basses.

1.4.3 Couches Standards

Couche	Index
DRAG LAYER	**400**
POPUP LAYER	**300**
MODAL LAYER	**200**
PALETTE LAYER	**100**
DEFAULT LAYER	**0**
FRAME CONTENT LAYER	**-30000**

1.4.4 ContentPane

C'est au niveau de la **ConentPane**, qui est un simple panneau (un JPanel), que tous les autres composants graphiques de l'interface utilisateur doivent être déposés.

1.4.5 JMenuBar

Le composant **JMenuBar** est un composant Swing situé avec la « **ContentPane** » dans la même couche : la **LayerdPane** qui correspond à la **FRAME CONTENT LAYER** (-30000).

Il permet de définir un menu déroulant dont les sous-menus sont des composants de type **JMenu**. Ceux-ci sont des composants flotants et seront alors visualisés dans la couche **POPUP LAYER** (300) soit alors en dessus de tous les autres composants graphiques.

1.5 Etude des classes de base : Component, Container et JComponent

1.5.1 Componant

void **repaint**()

void **requestFocus**() Demande à avoir le focus

void **setBackground**(Color c)

void **setBounds**(int x, int y, int width, int height)

void **setBounds**(Rectangle r)

void **setComponentOrientation**(ComponentOrientation o)

void **setCursor**(Cursor cursor)

void **setEnabled**(boolean b)

void **setFocusable**(boolean focusable)

void **setFont**(Font f)

void **setForeground**(Color c)

void **setLocation**(int x, int y)

void **setLocation**(Point p)

void **setName**(String name)

String **getName**()

void **setPreferredSize**(Dimension d)

void **setSize**(Dimension d)

void **setSize**(int width, int height)

void **setVisible**(boolean b)

1.5.2 Container

Component **add**(Component comp)
Component **add**(Component comp, int index)
Component **getComponent**(int n)
Component **getComponentAt**(int x, int y)
Component **getComponentAt**(Point p)
int **getComponentCount**()
Component[] **getComponents**()
void **remove**(Component comp)
void **remove**(int index)
void **removeAll**()
void **setLayout**(LayoutManager mgr)

1.5.3 JComponant

La classe **JComponent** est la classe mère de tous les composants Swing hormis les conteneurs de haut niveau. Elle étend la classe **java.awt.Container**. Elle apporte de nombreuses fonctionnalités aux composants Swing en dérivant, telles que le support de bordures, la possibilité d'associer des icônes aux composants et de formater du texte en html, le support des « ToolTips » pour commenter les composants survolés par le curseur de la souris, le support du double-buffering pour accélérer le rendu d'images complexes, le support du « plugable look and feel » et bien d'autres fonctionnalités.

Les principales méthodes de ce composant sont :

void **setBorder**(Border border)

void **setOpaque**(boolean isOpaque)

void **setToolTipText**(String text)

1.6 Principe de développement d'une application Swing

1.6.1 Architecture

Une application Swing est réalisée à base de plusieurs composants Swing. On peut énumérer généralement les composants suivants :

1. La fenêtre principale qui étend la classe **JFrame**
2. Des boîtes de dialogues qui étendent la classe **JDialog**
3. Des panneaux imbriqués déposés sur la ContentPane de chaque fenêtre. Ces panneaux étendent généralement la classe **JPanel** (ou tout autre conteneur intermédiaire).
4. Des composants simples ou atomiques instances de classes filles de la classe JComponent (JButton, JTextField, …)

La fenêtre principale peut ainsi définie selon l'architecture suivante :

```
public class MainFrame extends JFrame {

        public MainFrame () {
                super("titre de la fenêtre");
                build();
                setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                setVisible(true);
        }

        private void build() {
                JPanel content = new JPanel();
                content.add(…);
                content.add(…);
                ...
                setContentPane(content);
                setSize(…, …);
        }

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

Remarques :

1. Il est aussi possible d'utiliser la ContentPane par défaut au lieu d'un nouveau panneau conteneur :

```
JPanel content = (JPanel)getContentPane() ;
```

2. Les composants les plus importants de l'interface graphique sont généralement définis comme propriétés de la classe.

1.6.2 Constructeurs et méthodes de la classe JFrame

Constructeurs utiles de la classe JFrame :

- **JFrame()** crée une nouvelle fenêtre initialement invisible
- **JFrame(String)** crée une nouvelle fenêtre initialement invisible avec un titre spécifié

Principales Méthodes de la classe JFrame :

- Container **getContentPane**()
- Component **getGlassPane**()
- JMenuBar **getJMenuBar**()
- JLayeredPane **getLayeredPane**()
- JRootPane **getRootPane**()

- void **setContentPane**(Container contentPane)
- void **setGlassPane**(Component glassPane)
- void **setJMenuBar**(JMenuBar menubar)
- void **setLayeredPane**(JLayeredPane layeredPane)
- void **setRootPane**(JRootPane root)

- void **setDefaultCloseOperation**(int operation)
- void **setIconImage**(Image image)

Méthodes héritées de la classe Frame :

- static Frame[] **getFrames**()
- void **setCursor**(int cursorType)
- void **setExtendedState**(int state) :
  ICONIFIED, MAXIMIZED_HORIZ, MAXIMIZED_VERT, MAXIMIZED_BOTH
- void **setResizable**(boolean resizable)
- void **setState**(int state)
- void **setTitle**(String title)
- void **setUndecorated**(boolean undecorated)

Méthodes héritées de la classe Window :

- Window[] **getOwnedWindows**()
- Window **getOwner**()
- void **pack**()
- void **setLocationRelativeTo**(Component c)
- void **toBack**()
- void **toFront**()

Remarque :

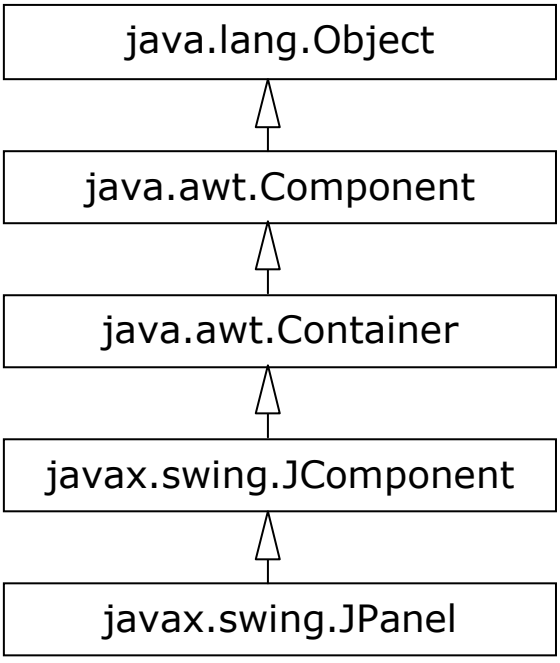
La classe JFrame dérive aussi de la classe « **Component** » et la classe « **Container** », elle hérite alors de toutes les méthodes de celles-ci.

1.7 Développement basé sur les panneaux

La classe JPanel est la principale classe utilisée pour la réalisation des interfaces Swing. Il s'agit à la fois d'un conteneur et d'un composant Swing. Tous les composants Swing de l'interface utilisateur sont réalisés principalement sur un JPanel et jamais directement sur les fenêtres, ce qui constitue une grande amélioration par rapport aux principes des interfaces AWT.

La classe JPanel hérite de toutes les méthodes de ces 3 classes mères : **Component**, **Container** et **JComponent**. Elle ne dispose que de quelques méthodes (6 méthodes) propres à elle et qui que ne sont pas de grande utilité pour le développeur.

Utilisée comme conteneur, on utilisera constamment sa principale méthode héritée de la classe Container : **add(...)**



1.8 Composants lourds et composants légers

On parle de composant lourd lorsque l'API du système est utilisée pour créer le composant en question. C'est le cas des composants AWT qui sont tous créés à l'aide d'une multitude d'appels système.
Les composants Swing quant à eux (à l'exception des 4 fenêtres qui constituent les conteneurs haut niveau), sont tous créés à l'aide de l'API Java sans passer par les librairies graphiques du système. Ils sont ainsi portables, mais en plus, ils disposent d'un aspect visuel (look and feel) portable, qui ne change donc pas d'un système à un autre.

1.9 Gestion des "Look And Feel": Le Design Pattern « Abstract Factory »

1.9.1 Déterminer les différents Look & Feel possibles :

```
UIManager.LookAndFeelInfo LF[];

LF=UIManager.getInstalledLookAndFeels();
```

Exemple :

```
UIManager.LookAndFeelInfo LF[];
LF=UIManager.getInstalledLookAndFeels();
for (int i=0;i<LF.length;i++) {
        System.out.println(
                LF[i].getName()+ " : " +
                LF[i].getClassName()
        );
}
```

➔

```
Metal : javax.swing.plaf.metal.MetalLookAndFeel

CDE/Motif : com.sun.java.swing.plaf.motif.MotifLookAndFeel

Windows : com.sun.java.swing.plaf.windows.WindowsLookAndFeel

Windows Classic : com.sun.java.swing.plaf.windows.WindowsClassicLookAndFeel
```

1.9.2 Changer de Look & Feel

Changement du look & Feel

```
UIManager.setLookAndFeel("classe du Look And Feel");
```

Application du look & feel à un arbre de composants :

```
SwingUtilities.updateComponentTreeUI(Racine de l'arbre);
```

Exemple :

```
try {
        UIManager.setLookAndFeel (
                "com.sun.java.swing.plaf.windows.WindowsLookAndFeel "
        );
}
catch(Exception e) {}
JFrame f = new JFrame();
SwingUtilities.updateComponentTreeUI(f);
```

Chapitre 2. Composition d'interfaces Swing et « Layout Managers »

2.1 Positionnement absolu et disposition par Layout

Positionnement absolu

Pour tous composant Swing « composant » à ajouter sur un panneau « P ». On pourra faire :

- **composant.setLocation(x, y);**
- **composant.setSize(width, height);**
- **composant.setBounds(x, y, width, height);**

A Condition :

- **P.setLayout(null);**

Disposition par Layout

Il s'agit d'associer un Layout Manager (ou gestionnaire d'emplacement) au panneau conteneur.

P.setLayout(new UneClasseLayoutManagerConnue()**) ;**

Celui-ci va calculer automatiquement les positions et les tailles des composants qu'il contient. Le calcul est réalisé suivant une politique d'emplacement propre au « layout manager ».

Différents « layout managers » existent alors, chacun avec sa propre politique d'emplacement.

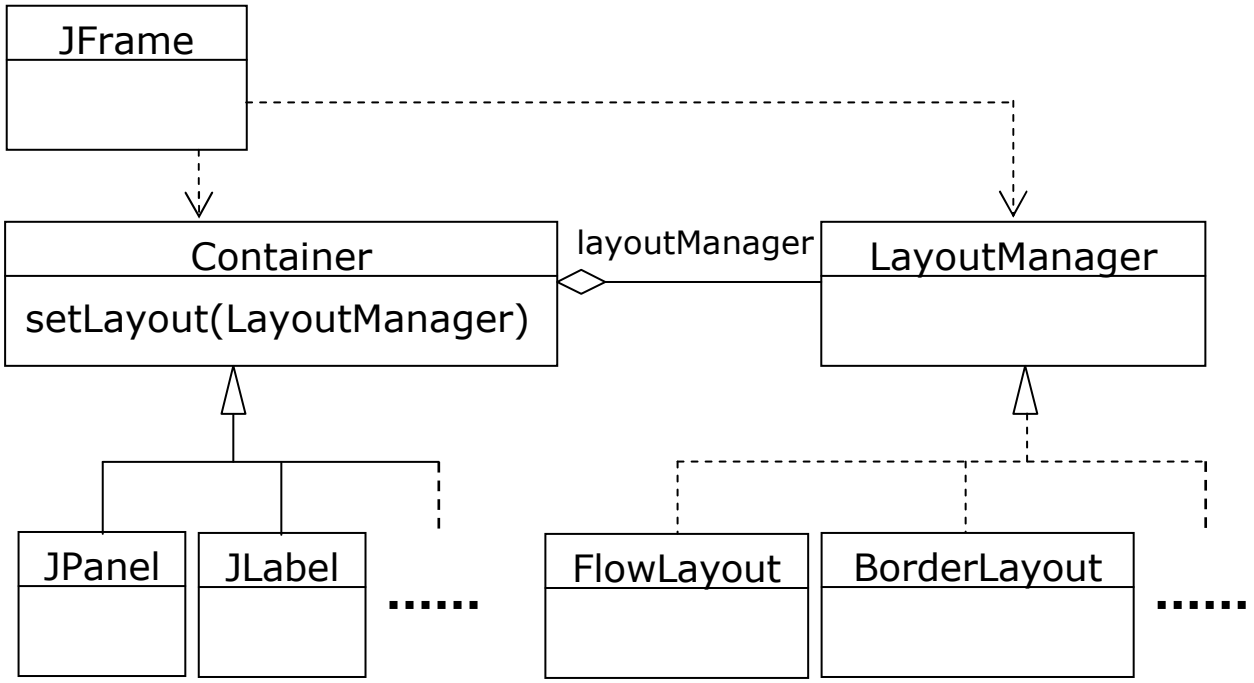
Le concepteur de l'interface aura alors les tâches et possibilités suivantes :

- Choisir des Layout Managers adéquats à associer aux différents panneaux de son interface.
- Utiliser une composition de Layout Managers à l'aide d'une superposition de panneaux
- Créer un Layout Manager approprié.

Remarques :

- Un composant **JPanel** est géré par défaut à l'aide du layout manager « **FlowLayout** »
- La « **ContentPane** » est gérée par défaut à l'aide du « Layout Manager » « **BorderLayout** »

2.2 Principe des Gestionnaires d'emplacement : « Layout Managers » et Design Pattern « Bridge »



2.3 Etude des Layouts Manager de base : FlowLayout, BorderLayout, GridLayout, BoxLayout, GridBagLayout

2.3.1 FlowLayout

- Constructeurs :

- **public FlowLayout()**
- **public FlowLayout(int alignment)**

  FlowLayout.LEFT

  FlowLayout.RIGHT

  FlowLayout.CENTER
- **public FlowLayout(int alignment, int hgap, int vgap)**

  hgap : espacement horizontal (en pixels) entre les composants

  vgap : espacement vertical (en pixels) entre les composants

Exemple :

```
JPanel P = new JPanel() ;
P.setLayout(new FlowLayout(FlowLayout.LEFT)) ;
```

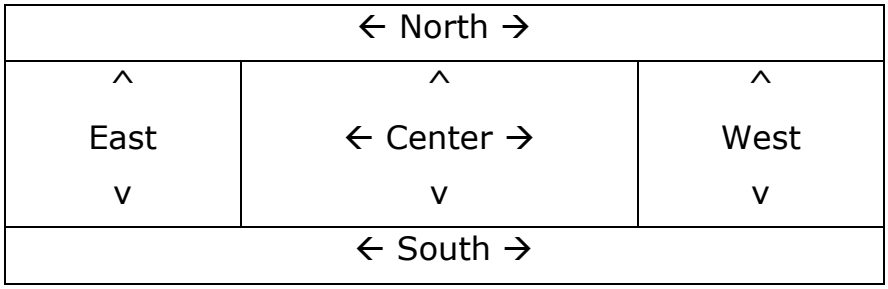
- Méthodes utilisées pour ajouter des composants sur un JPanel :

- **public void add(Componant [, int]) ;**

Exemple :

```
JPanel P = new JPanel() ;
P.setLayout(new FlowLayout(FlowLayout.LEFT)) ;
P.add(new JButton("Ok")) ;
P.add(new JButton("Annuler")) ;
```

2.3.2 BorderLayout



- Constructeurs :

- **public BorderLayout()**
- **public BorderLayout(int hgap, int vgap)**

- Méthodes utilisées pour ajouter des composants sur un JPanel :

- **public void add("région", Composant) ;**
  région prend les valeurs : **North, South, East, West, Center**

Exemple :

```
JPanel P = new JPanel() ;
P.setLayout(new BorderLayout()) ;

JPanel Ps = new JPanel() ;
Ps.setLayout(new FlowLayout()) ;
Ps.add(new JButton("Ok")) ;
Ps.add(new JButton("Annuler")) ;

P.add("Center", new JTextArea(20,10)) ;
P.add("South", Ps) ;
```

2.3.3 BoxLayout

Composant N° 1
Composant N° 2
…
Composant N° N

- Constructeurs :

- **public BoxLayout(Container target, int axis)**

  « target » est le conteneur lui-même, celui auquel on applique le layout.

  « axis » est une constante qui précise si l'alignement est vertical ou horizontal :

  BoxLayout.**X_AXIS**

  BoxLayout.**Y_AXIS**

- Méthodes utilisées pour ajouter des composants sur un JPanel :

- **public void add(Composant [, indice]) ;**

Exemple :

```
JPanel P = new JPanel() ;
P.setLayout(new BoxLayout(P, BoxLayout.Y_AXIS)) ;
P.add(new JButton("Ok")) ;
P.add(new JButton("Annuler")) ;

JPanel p = new JPanel(new BoxLayout(p, BoxLayout.Y_AXIS) );
```

2.3.4 GridLayout

1	2	3
4	5	6

- Constructeurs :

- **public GridLayout()**

➔ Une seule ligne avec un nombre de composants quelconque, un composant par cellule

- **public GridLayout(int rows, int cols)**
- **public GridLayout(int rows, int cols, int hgap, int vgap)**

- Méthodes utilisées pour ajouter des composants sur un JPanel :

- **public void add(Composant [, indice]) ;**

Exemple :

```
JPanel P = new JPanel() ;
P.setLayout(new GridLayout(2,3)) ;
for (int i=1 ;i<=6 ;i++) P.add(new JButton(""+i)) ;
```

Remarques :

```
P.setLayout(new GridLayout(0,3)) ;
```

➔ Crée un Grid avec 3 colonnes et un nombre de lignes quelconque dépendant du nombre de composants ajoutés au panneau.

```
P.setLayout(new GridLayout(3, 0)) ;
```

➔ Crée un Grid avec 3 lignes et un nombre de colonnes quelconque dépendant du nombre de composants ajoutés au panneau.

2.4 Principe de développement de composants complexes : spécialisation des Panneaux

L'interface graphique devrait être conçu comme une composition de composants complexes. Chaque composant complexe est lui même composé d'autres composants complexes ou de composants simples ou atomiques Swing (JButton, JTextField, etc…).

Chaque composant complexe doit être réalisé séparément sous forme d'un panneau **autonome réutilisable** par spécialisation de la classe **JPanel** :

```
public class ComposantComplexe1 extends JPanel {
        …
        add(new ComposantSimple1(…)) ;
        add(new ComposantSimple2(…)) ;
        …
}
```

```
public class ComposantComplexe2 extends JPanel {
        …
        add(new ComposantSimple1(…)) ;
        add(new ComposantSimple2(…)) ;
        …
}
```

```
public class ComposantComplexe3 extends JPanel {
        …
        add(new ComposantComplexe1(…)) ;
        add(new ComposantComplexe2(…)) ;
        …
}
```

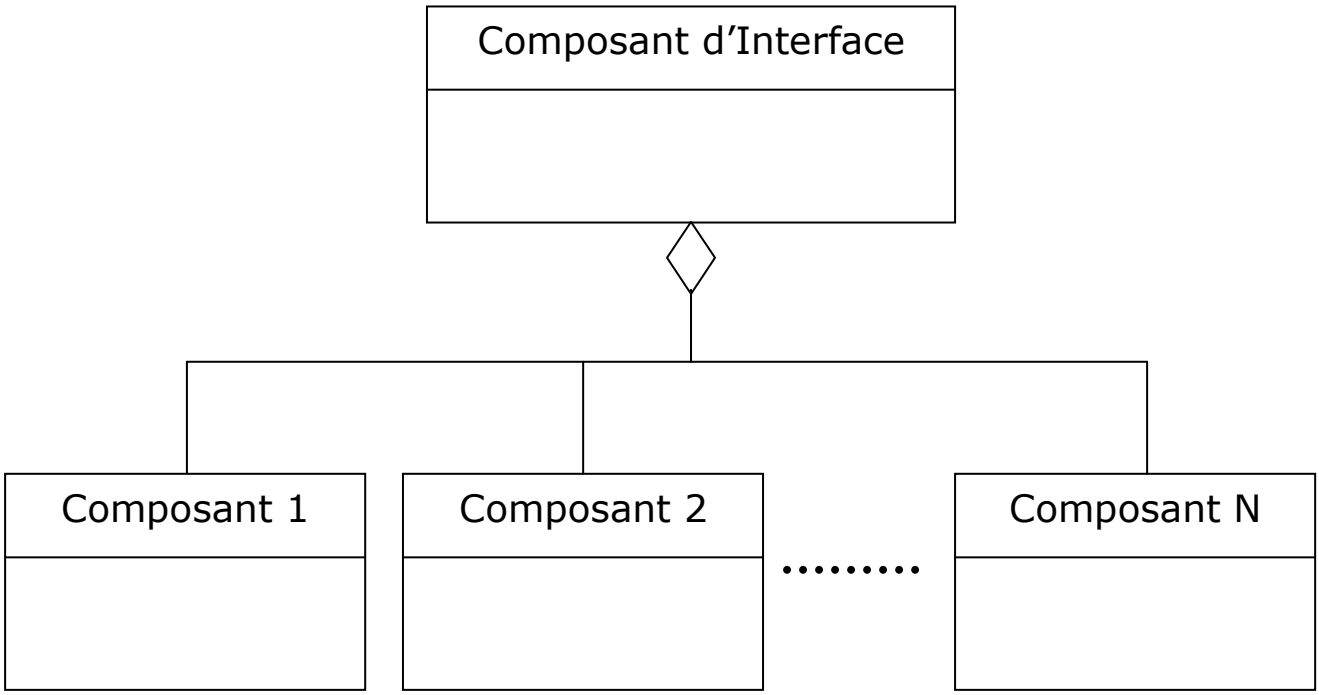
Les fenêtres seront ensuite conçues à base des panneaux déjà réalisés :

```
public class FentrePrincipale extends JFrame {
        …

        setContentPane(new ComposantComplexe3(…)) ;
        ou
        getContentPane().add("…", new ComposantComplexe3(…)) ;
        …
}
```

2.5 Développement de composants Swing réutilisables (Etude de cas) : Les Design Patterns « Façade », « Decorator » et « Dependancy Injection »

2.5.1 Développement de composants d'interface suivant le design pattern « **Façade** »

Ce Design Pattern permet de cacher la complexité issue de l'utilisation d'autres composant qu'il encapsule au sein de lui-même tout en offrant les fonctionnalités requises avec un mécanisme d'accès plus simple.



Implémentation :

```
public class ComposantFacade extends JPanel {
        public ComposantFacade(...) {
                ...
                add(new Composant1(...)) ;
                add(new Composant2(...)) ;
                ...
        }
        ...
}
```

Exemple : Un panneau de boutons

```
public class ButtonPanel extends JPanel {
        public ButtonPanel(String labels[]) {
                for (int i=0; i<labels.length; i++) {
                        add(new JButton(labels[i]));
                }
        }
}
```

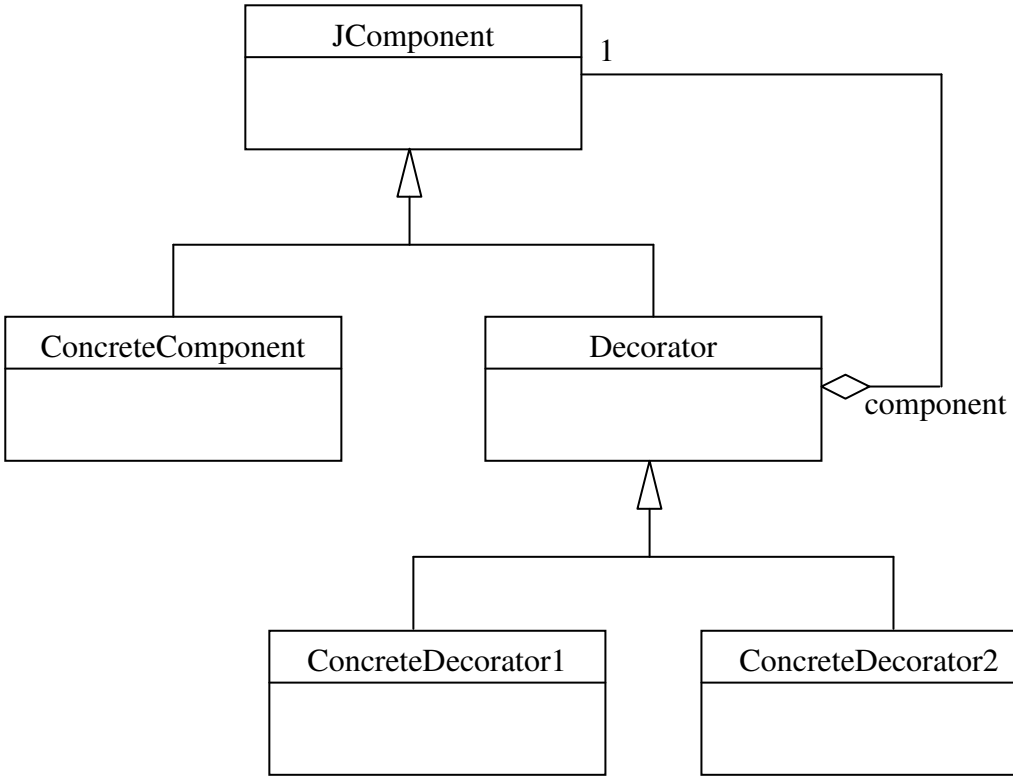
Utilisation :

```
public class MainFrame extends JFrame {

        public MainFrame () {
                build() ;
                pack();
                setVisible(true);
        }

        private void build() {
                JPanel content = (JPanel)getContentPane();
                String B []={"OK","Cancel","Help"};
                content.add("South", new ButtonPanel(B));
        }

        public static void main(String ars[]) {
                new MainFrame();
        }
}
```

2.5.2 Le design pattern « Decorator »

Il ressemble dans sa structure au Design Pattern Composite mais avec un objectif différent : la possibilité de superposer les Décors appliqués à un Component.

La règle : **Le Composant Décoré est lui-même un composant**. Càd raisonner par « héritage + Agrégation » : ainsi, si on prend un Composant C1 et un Décor D1, D1 appliqué à C1 donne D1(C1) qui est un nouveau composant acceptant d'être décoré. Si on prend un nouveau Décor D2, on peut alors faire : D2(C1), mais plus important : D2 (D1 (C1)).

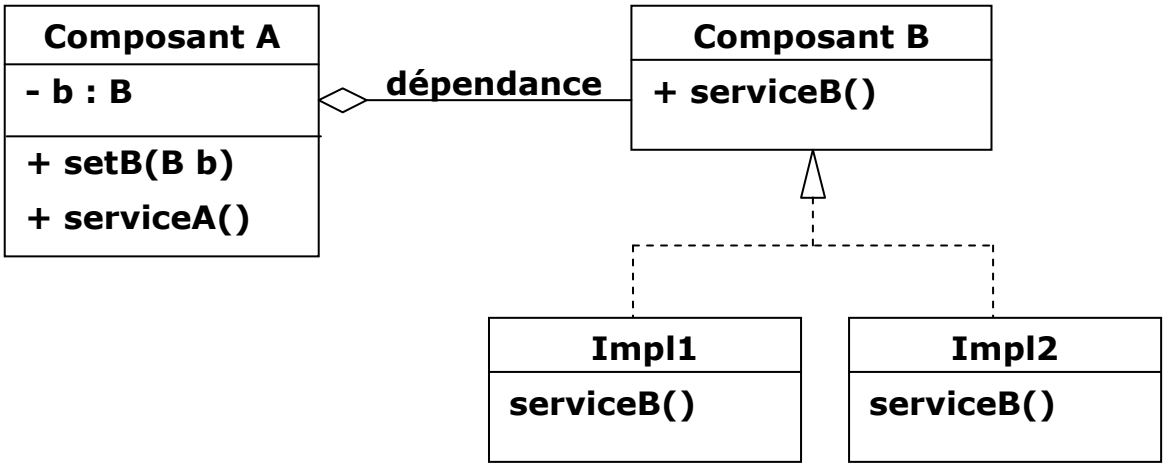


Implémentation :

```
public class Decorator extends JPanel {
        public Decorator(JComponent cmp) {
                ...
        }
        ...
}
```

2.5.3 Le design pattern « Dependancy Injection »

C'est un cas de bridge visant à injecter un composant au choix dans un composant complexe dépendant. Celui-ci pourra changer de comportement, ou d'aspect visuel, en fonction de la nature du composant injecté.



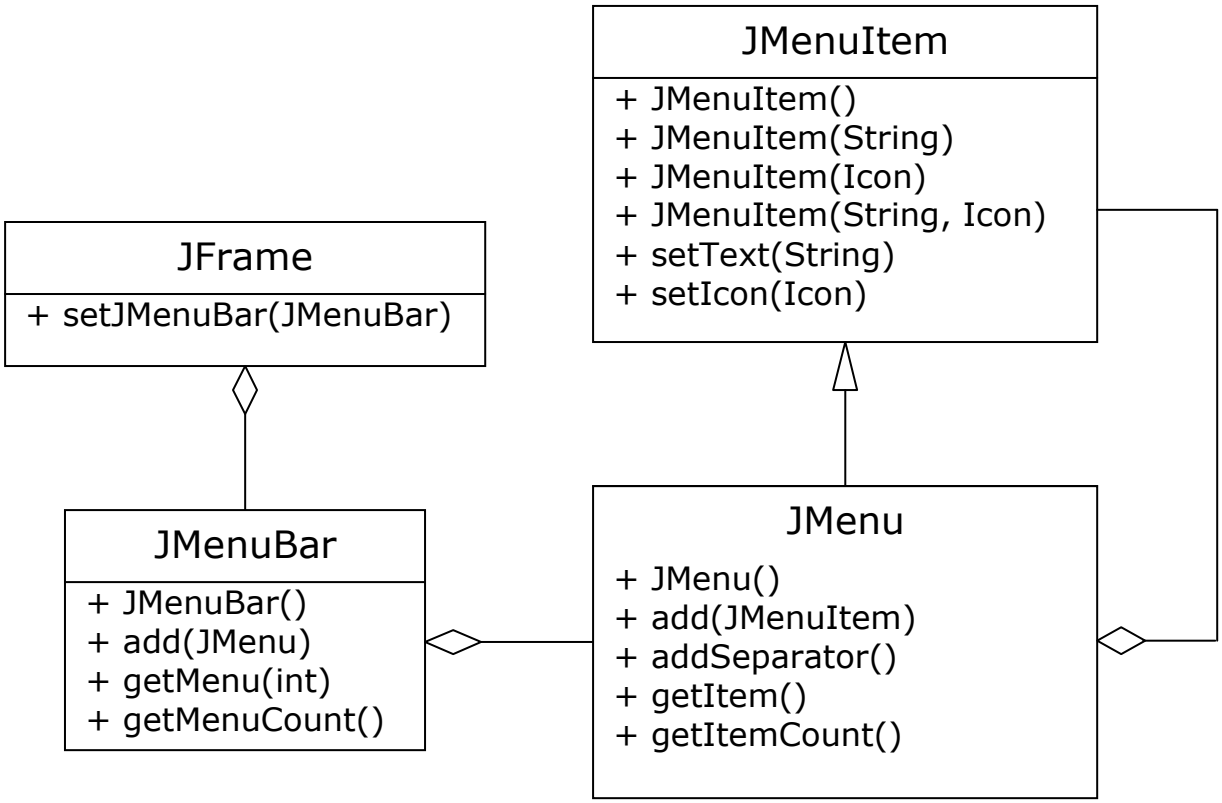
Exemple :

CommandList (Composant A) est une classe qui crée un panneau de composants. Un tableau d'étiquettes est communiqué à cette classe par l'intermédiaire d'une classe issue de l'interface LabelList (Composant B). Celle-ci permet aussi de communiquer à CommandList la nature des composants à créer : JButton, JToggleButton, JCheckBox, ou JRadioButton. L'interface LabelList peut alors avoir une implémentation différente pour chaque type de composant différent.

2.6 Gestion des menus

Trois classes sont à utiliser : **JMenuBar**, **JMenu** et **JMenuItem**
JMenuBar correspond à la barre de menu. Il est composé de plusieurs instances de JMenu, qui sont les éléments visibles directement dans la barre de menu.

Chaque instance de JMenu peut contenir plusieurs instances de JMenuItem, qui sont les éléments visibles quand l'utilisateur clique sur un menu. Un JMenu peut aussi contenir d'autres JMenu.



Création du MenuBar :

```
JMenuBar mb = new JMenuBar() ;

this.setJMenuBar(mb) ;        // je suppose qu'on se trouve dans la fenêtre

                              //principale qui étend la JFrame
```

Création des Menus :

```
JMenu menus = {
        new JMenu("Fichier"),
        new JMenu("Edition"),
        …
}
for (int i=0; i<menus.length; i++) mb.add(menus[i]);
```

Création des Items :

```
JMenuItem items[] = {
        new JMenuItem("Nouveau", new ImageIcon("chemin")) ,
        new JMenuItem("Ouvrir", new ImageIcon("chemin")) ,
        new JMenuItem("Enregistrer", new ImageIcon("chemin")) ,
        …
}

for (int i=0 ; i<N0 ; i++)  menus[0].add(items[i]);
for (int i=N0 ; i<N1 ; i++)  menus[0].add(items[i]);
…
```

Ajouter un Séparateur à un menu :

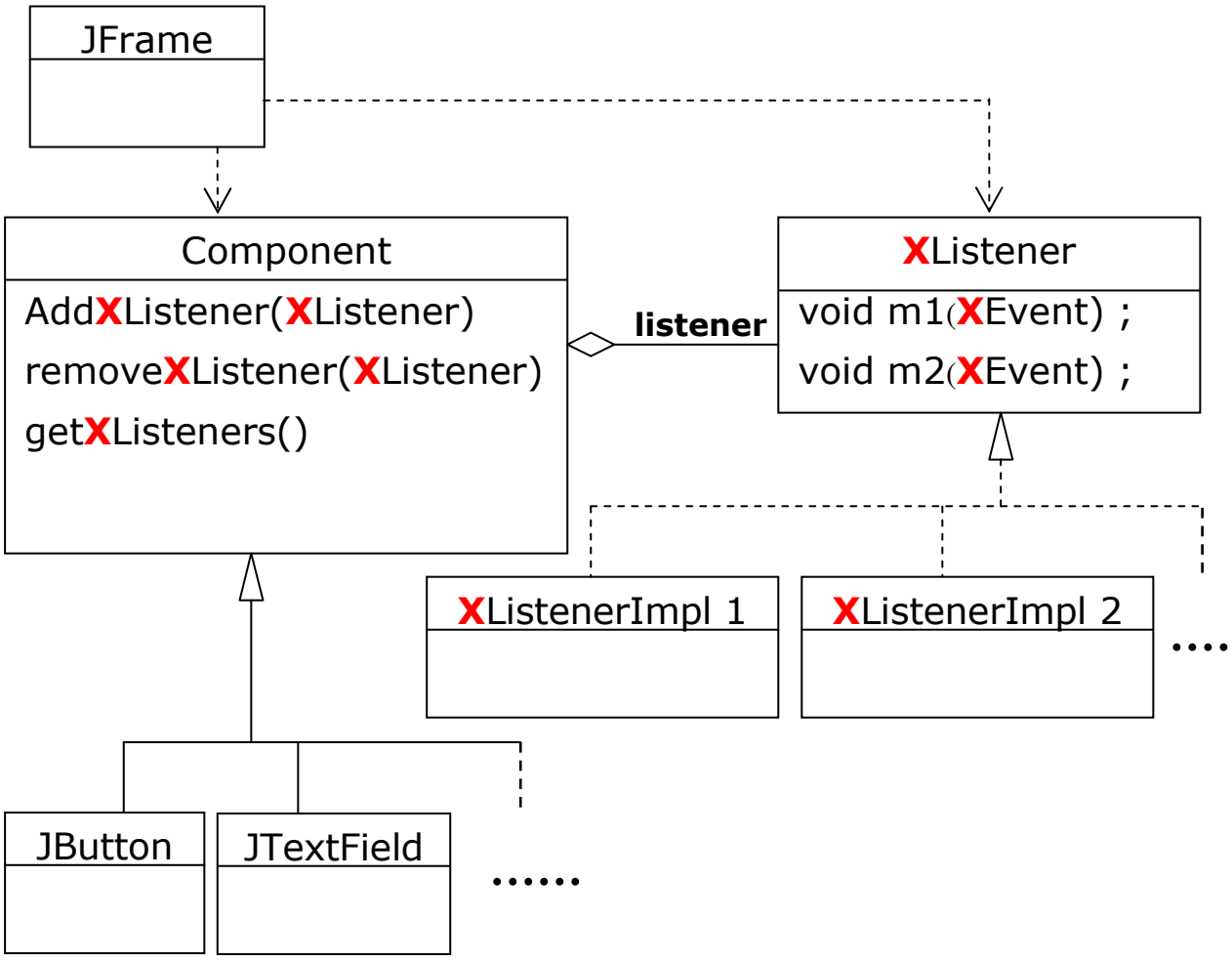
```
menus[0].addSeparator() ;
```

Chapitre 3. Gestion des événements

3.1 Principe des Ecouteurs « Listeners » et design Patterns « Bridge » et « Observer ».

Un écouteur d'événement est un objet de classe qui écoute des actions particulières sur un composant donné. C'est un « observateur » réalisé conformément au design pattern « observer ». On parlera aussi de contrôleur suivant le design pattern MVC (Model View Controler).

L'implémentation d'un écouteur est réalisée sous forme d'un bridge.



Exemple :

```
public class ExitAction implements ActionListener {
        public void actionPerformed(ActionEvent e) {
                System.exit(0);
        }
}
```

Usage :

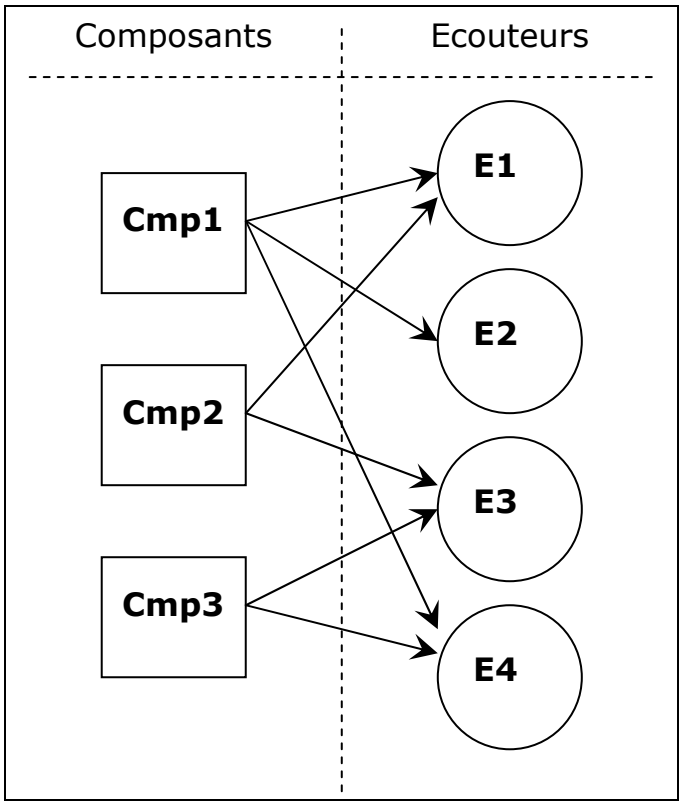
Soit un bouton de l'interface :

```
JButton quitter = new JButton("Quitter");
```

L'instruction suivante permet d'associer au bouton « quitter » l'écouteur d'événement qui permet de quitter l'application suite à une action click sur le bouton:

```
quitter.addActionListener(new ExitAction()) ;
```

3.2 Composition et réutilisabilité des écouteurs



3.3 Différents Modèles d'implémentation des écouteurs d'événements

Il existe 2 modèles d'implémentations possibles :

1. à l'aide d'une classe (ou contrôleur) Externe ➔ Avantage : Réutilisabilité de l'écouteur d'événements
2. Une implémentation interne ➔ Avantage : Accessibilité des composants membres. Dans ce cas, il y a encore 3 types d'implémentation :

- La classe d'interface implémente elle-même l'écouteur
- Le contrôleur est une classe Interne
- Le contrôleur est une classe Locale anonyme

3.3.1 Contrôleur externe

```
public class Controleur implements XListener {
        private propriétés éventuelles ;
        public Controleur(paramètres éventuels) {

        }

        public methodeD'ecoute(XEvent e) {
                traitement de l'événement
        }
        …
}
```

```
Public class ComposantGraphique extends JFrame|JPanel|… {
        private JComponent cmp ;
        …
        private void build() {
                …
                cmp.addXListener(new Controleur(…)) ;
                …
        }
        …
}
```

3.3.2 Contrôleur entrelacé

```
public class ComposantGraphique
        extends JFrame|JPanel|...
        implements XListener
{
        private JComponent cmp ;
        …
        private void build() {
                JComponent cmp = …
                cmp.addXListener(new Controleur(…)) ;
                …
        }
        …
        public methodeD'ecoute(XEvent e) {
                traitement de l'événement
        }
        …
}
```

Remarques :

1. Dans ce cas, le contrôleur a accès à toutes les propriétés de la classe. Il peut donc agir facilement sur les différents composants de l'interface.
2. Dans le cas d'un contrôleur externe, si celui-ci a besoin d'accéder aux composants de l'interface, il faut les lui passer, par exemple en paramètre de son constructeur, ou bien passer la référence de toute la fenêtre ou le panneau en question :

```
cmp.addXListener(new Controleur(this)) ;
```

3.3.3 Contrôleur interne

```
Public class ComposantGraphique extends JFrame|JPanel|… {
    private JComponent cmp ;
    …
    private void build() {
        …
        cmp.addXListener(new Controleur()) ;
        …
    }
    …
    public class Controleur implements XListener {

        public methodeD'ecoute(XEvent e) {
            traitement de l'événement
        }
        …
    }
}
```

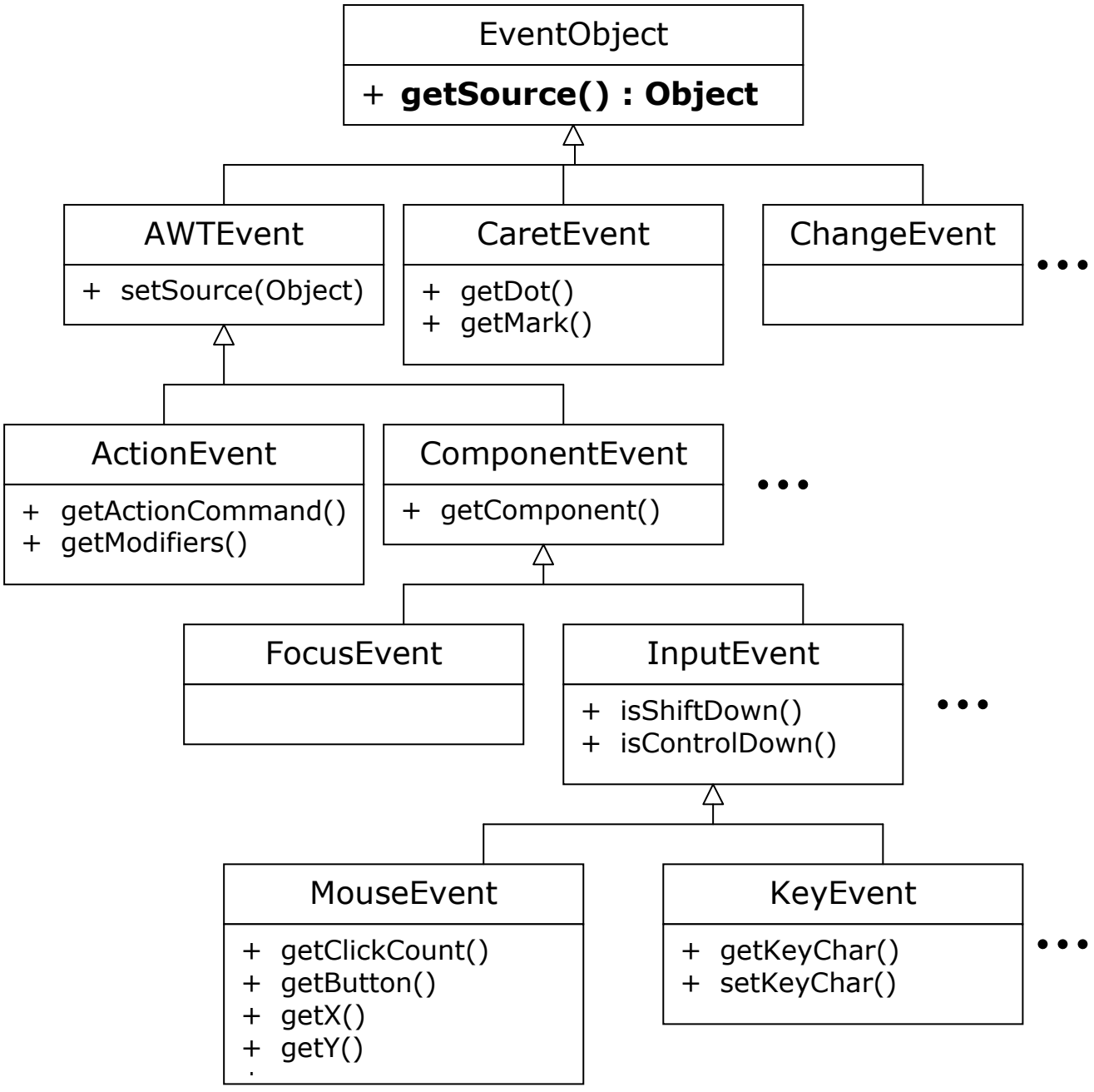
Remarque :

Dans ce cas aussi, le contrôleur a accès à toutes les propriétés de la classe.

3.3.4 Contrôleur anonyme

```
Public class ComposantGraphique extends JFrame|JPanel|… {
    private JComponent cmp ;
    …
    private void build() {
        …
        cmp.addXListener(new Controleur() {
            public methodeD'ecoute(XEvent e) {
                traitement de l'événement
            }
            …
        }) ;
        …
    }
    …
}
```

3.4 Hiérarchie des événements



3.5 Les écouteurs de base

Tous les composants graphiques disposent d'un ensemble d'écouteurs communs (associés à la classe Component). Les plus utilisés sont :

- FocusListener
- KeyListener
- MouseListener
- MouseMotionListener

3.5.1 FocusListener

☒ void **focusGained**(FocusEvent e)
☒ void **focusLost**(FocusEvent e)

3.5.2 KeyListener

☒ void **keyPressed**(KeyEvent e)
☒ void **keyReleased**(KeyEvent e)
☒ void **keyTyped**(KeyEvent e)

KeyEvent

- char **getKeyChar**()
- void **setKeyChar**(char keyChar)
- int **getKeyCode**()
- void **setKeyCode**(int keyCode)
- int **getKeyLocation**()
- static String **getKeyModifiersText**(int modifiers)
- static String **getKeyText**(int keyCode)
- Boolean **isActionKey**()
- void **setModifiers**(int modifiers)

3.5.3 MouseListener

☒ void **mouseClicked**(MouseEvent e)
☒ void **mouseEntered**(MouseEvent e)
☒ void **mouseExited**(MouseEvent e)
☒ void **mousePressed**(MouseEvent e)
☒ void **mouseReleased**(MouseEvent e)

MouseEvent

- int **getButton**()
- **BUTTON1, BUTTON2, BUTTON3**
- int **getX**()
- int **getY**()
- int **getClickCount**()
- Point **getPoint**()
- Boolean **isAltDown**()
- Boolean **isControlDown**()
- void **translatePoint**(x, y)
- boolean **isAltGraphDown**()
- boolean **isShiftDown**()

3.5.4 MouseMotionListener

☒ void **mouseDragged**(MouseEvent e)
☒ void **mouseMoved**(MouseEvent e)

3.5.5 MouseWheelListener

☒ void **mouseWheelMoved**(MouseWheelEvent e)

MouseWheelEvent

- int **getScrollAmount**()
- int **getScrollType**()
- int **getUnitsToScroll**()
- int **getWheelRotation**()

3.6 Ecouteurs complémentaires

3.6.1 ActionListener

☒ void **actionPerformed**(ActionEvent e)

Associé aux composants boutons **AbstractButton**, **JTextField**, **JComboBox** et **Timer**.

3.6.2 TextListener

☒ void **textValueChanged**(TextEvent e)

Associé aux composants de texte : **JTextComponent**

3.6.3 CaretListener

☒ void **caretUpdate**(CaretEvent e)

Associé aux composants de texte : **JTextComponent**

3.6.4 ComponentListener

☒ void **componentHidden**(ComponentEvent e)
☒ void **componentMoved**(ComponentEvent e)
☒ void **componentResized**(ComponentEvent e)
☒ void **componentShown**(ComponentEvent e)

3.6.5 ContainerListener

☒ void **componentAdded**(ContainerEvent e)
☒ void **componentRemoved**(ContainerEvent e)

3.6.6 AncestorListener

☒ void **ancestorAdded**(AncestorEvent event)
☒ void **ancestorMoved**(AncestorEvent event)
☒ void **ancestorRemoved**(AncestorEvent event)

3.6.7 ChangeListener

☒ void **stateChanged**(ChangeEvent e)

3.6 Ecouteurs des fenêtres

WindowListener

- void **windowActivated**(WindowEvent e)
- void **windowClosed**(WindowEvent e)
- void **windowClosing**(WindowEvent e)
- void **windowDeactivated**(WindowEvent e)
- void **windowDeiconified**(WindowEvent e)
- void **windowIconified**(WindowEvent e)
- void **windowOpened**(WindowEvent e)

WindowFocusListener

- void **windowGainedFocus**(WindowEvent e)
- void **windowLostFocus**(WindowEvent e)

WindowStateListener

- void **windowStateChanged**(WindowEvent e)

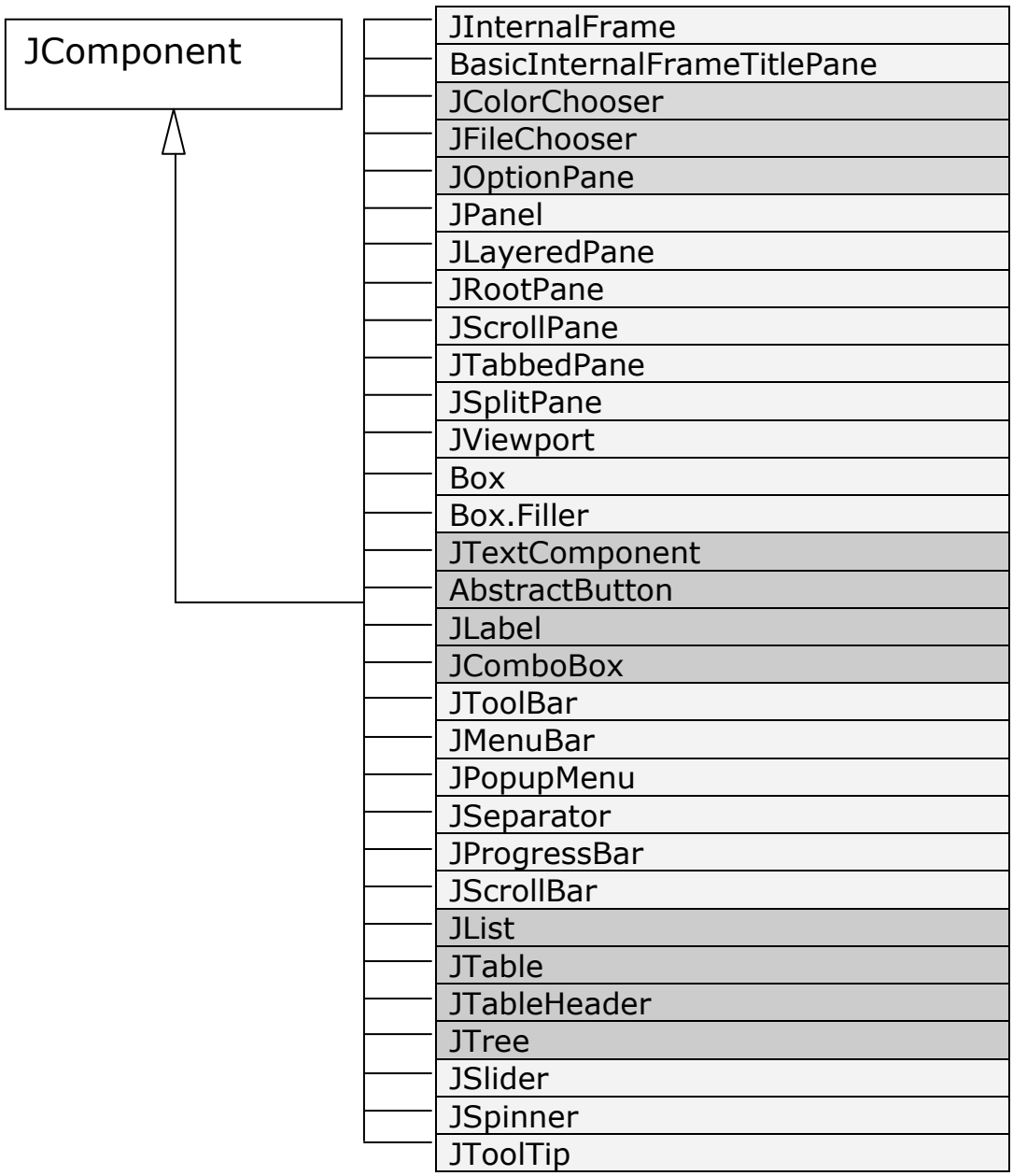
WindowEvent

- int **getNewState**()
- int **getOldState**()
- Window **getOppositeWindow**()
- Window **getWindow**()

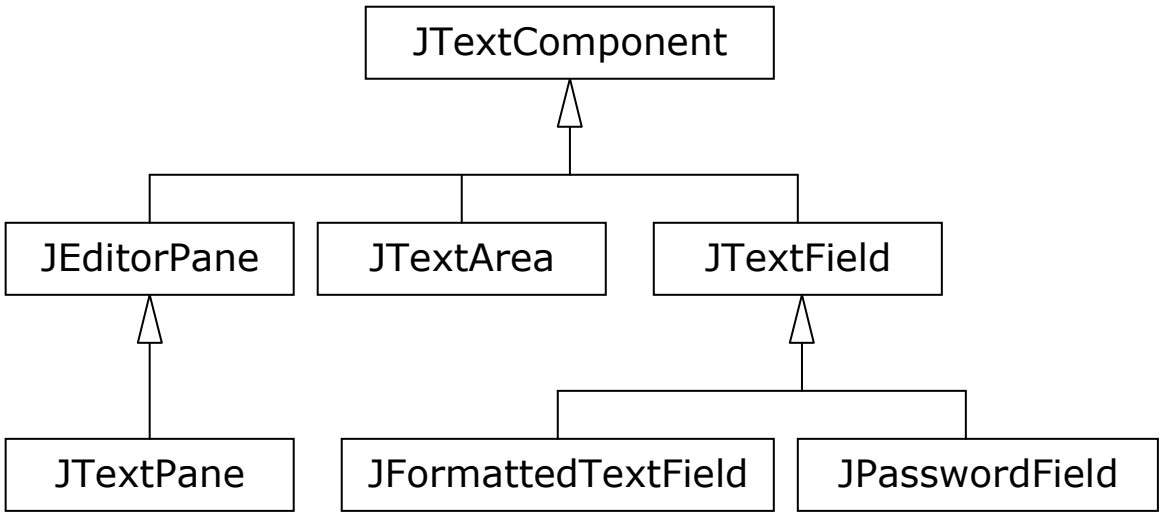
Chapitre 4. Principaux composants Swing

4.1 Hiérarchie des composants Swing

Les composants Swing héritent tous de la classe JComponent (sauf les conteneurs de haut niveau : les 4 fenêtres).



4.2 Composants de texte



4.2.1 Méthodes de la classe JTextComponent

Méthodes d'édition

+ **void setText(String t)**
+ **String getText()**
+ **String getText(int offs, int len)**
+ **void copy()**
+ **void cut()**
+ **void paste()**

Gestion de caret (curseur)

+ **void addCaretListener(CaretListener listener)**
+ **void setCaretPosition(int position)**
+ **int getCaretPosition()**
+ **void moveCaretPosition(int pos)**

Modèles de données

+ **void read(Reader in, Object desc)**
+ **void write(Writer out)**
+ **void setDocument(Document doc)**
+ **Document getDocument()**

Sélection

- **String getSelectedText()**
- **int getSelectionStart()**
- **int getSelectionEnd()**
- **void replaceSelection(String content)**
- **void select(int selectionStart, int selectionEnd)**
- **void setSelectionStart(int selectionStart)**
- **void setSelectionEnd(int selectionEnd)**
- **void selectAll()**

Configuration

- **void setMargin(Insets m)**
- **Insets getMargin()**
- **void setEditable(boolean b)**
- **boolean isEditable()**
- **void setComponentOrientation(ComponentOrientation o)**

4.2.2 Composant JTextField

Un JTextField est une zone d'édition de texte comportant une seule ligne. Ce composant est l'un des plus simple à utiliser pour la saisie de données, il ne permet de saisir qu'une seule ligne de texte et ne prend pas en compte de styles (gras, italique …).

Les méthodes getText() et setText(String s) permettent de récupérer le texte tapé par l'utilisateur, ou d'affecter un texte au JTextField.

Constructeurs de la classe JTextField :

- **JTextField()**
- **JTextField(int columns)**
- **JTextField(String text)**
- **JTextField(String text, int columns)**

Principales Méthodes :

+ **void setHorizontalAlignment(int a)**

Préciser l'alignement horizontal : **JTextField.LEFT, JTextField.RIGHT ou JTextField.CENTER**

+ **void setColumns(int nc)**

Affecte le nombre de colonnes, qui est utilisé pour calculer la taille préférée.

Ecouteur :

+ **void addActionListener(ActionListener l)**

4.2.3 Composant JFormattedTextField

Un JFormattedTextField permet de faire des saisies de texte formatées. Il permet aussi de configurer l'action à entreprendre quand le JFormattedTextField perd le focus, en utilisant la méthode setFocusLostBehavior :

- FormattedTextField.**REVERT** :
  La valeur éditée n'est pas prise en compte

- JFormattedTextField.**COMMIT** :
  La valeur éditée est prise en compte, si elle n'est pas légale une exception ParseException est levée et la valeur n'est pas changée.

- JFormattedTextField.**COMMIT_OR_REVERT** :
  Comme COMMIT et si la valeur n'est pas légale comme REVERT. C'est la valeur par défaut.

- JFormattedTextField.**PERSIST** :
  Ne rien faire, la valeur éditée est gardée dans le JFormattedTextField .

Constructeur de la classe JFormatedTextField :

+ **JFormatedTextField()**

Crée un JFormatTextField

+ **JFormatedTextField(Format f)**

Crée un JFormatTextField avec un format.

+ **JFormatedTextField(JFormattedTextField.AbstractFormatter**

Crée un **JFormattedTextField** avec un « formateur » issu de la classe **AbstractFormatter**, par exemple :

DefaultFormatter ou **MaskFormatter**.

Exemple 1 :

```
NumberFormat nf = NumberFormat.getInstance();
nf.setMinimumIntegerDigits(5);
JFormattedTextField t1;
t1 = new JFormattedTextField(nf);
t1.setColumns(5);
```

Exemple 2 :

```
try {
        MaskFormatter mf = new MaskFormatter("UUUUUUUUUU");
        JFormattedTextField t2 = new JFormattedTextField(mf);
        t2.setColumns(10);
}
catch (Exception e) {}
```

Exemple 3 :

```
DateFormat dateFormat;
dateFormat = DateFormat.getDateInstance(DateFormat.MEDIUM);
JFormattedTextField t3 = new JFormattedTextField(dateFormat);
t3.setValue(new Date());
```

Exemple 4 :

```
try {
        MaskFormatter mf = new MaskFormatter("##.##%");
        JFormattedTextField t4 = new JFormattedTextField(mf);
        t4.setColumns(4);
} catch (Exception e) {}
```

Principales Méthodes de la classe JFormatedTextField :

+ **Object getValue()**

Retourne la dernière valeur valide éditée dans le JFormattedTextField. Cette valeur dépend de la stratégie mise en place par la méthode setFocusLostBehavior.

+ **void setValue(Object v)**

Affecte une valeur au JFormattedTextField.

+ **boolean isEditValid()**

Retourne « true » si la valeur éditée est valide.

+ **int getFocusLostBehavior()**

Retourne le comportement du JFormattedTextField quand il perd le focus.

+ **void setFocusLostBehavior(int s)**

Affecte le comportement du JFormattedTextField quand il perd le focus.

4.2.4 JPasswordField

JPasswordField est un composant qui permet l'édition d'une ligne simple où les caractères tapés sont cachés.

Ce composant permet de créer un champ pour la saisie des mots de passe. Il s'utilise de la même manière que le composant JTextField sauf au cas de saisie de caractère, celui-ci est remplacé par le caractère par défaut '*'. Ce dernier peut être remplacé à l'aide de la méthode setEchoChar(char).

Constructeurs de la classe JPasswordField :

+ **JPasswordField()**
+ **JPasswordField(int c)**
+ **JPasswordField(String text)**
+ **JPasswordField(String text , int c)**

Principales Méthodes de la classe JPasswordField :

+ **char[] getPassword()**

  Retourne un tableau de caractères contenant le mot de passe saisi. (la méthode getText() est dépréciée)

+ **void setEchoChar(char c)**

  Change le caractère écho des caractères tapés.

  setEchoChar((char) 0); ➔ devient comme un JTextField

+ **char getEchoChar()**

  Retourne le caractère qui remplace les caractères tapés.

4.2.5 JLabel

Constructeur de la classe JLabel :

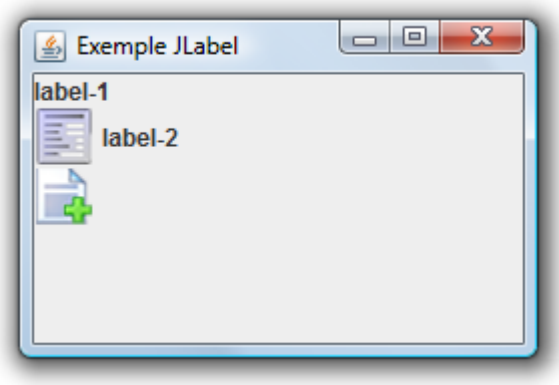
+ **JLabel ()**
  crée une étiquette sans image et avec une chaine vide pour titre.
+ **JLabel(Icon)**
  crée une étiquette avec une image .
+ **JLabel(Icon, int)**
  crée une étiquette avec une image allignée horizontalement.
+ **JLabel(String)**
  crée une étiquette avec du texte.
+ **JLabel (String,int)**
  crée une étiquette avec du texte alligné horizontalement.
+ **JLabel(String, Icon, int)**
  texte + image et alignement horizontal

Principales Méthodes de la classe JLabel :

void setText(String t)	Affecte un texte au label.
String getText()	Retourne le texte du label.
void setFont(Font f)	Affecte une police au label.
Font getFont()	Retourne la police du label.
void setOpaque(boolean b)	Rend le label opaque si b vaut true,par défaut un label est transparent.
boolean isOpaque()	Retourne la propriété opaque du label.
void setIcon(Icon i)	Affecte une icône au label.
Icon getIcon()	Retourne l'icône du label.
void setHorizontalAlignment(int a)	Affecte l'alignement horizontal : LEFT, RIGHT, CENTER, LEADING, TRAILING
void setVerticalAlignment(int a)	Affecte l'alignement vertical : TOP, BOTTOM, CENTER

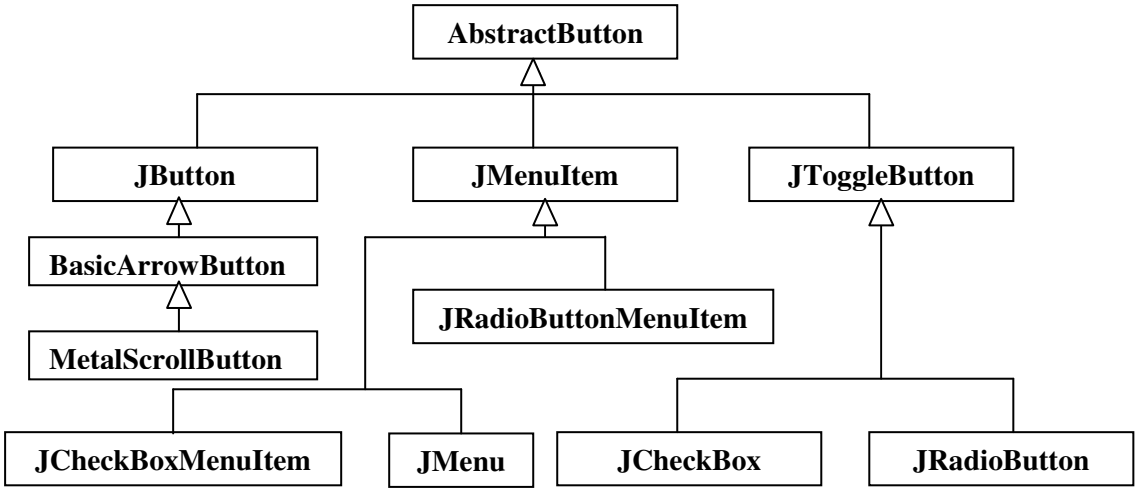
Exemple :

```
public class MainFrame extends JFrame {
	public MainFrame() {
		super("Exemple JLabel");
		build();
		setVisible(true);
	}

	private void build() {
		setSize(260, 170);
		setLocation(100, 80);
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

		JPanel p1 = new JPanel();
		p1.setLayout(new BoxLayout(p1, BoxLayout.Y_AXIS));
		JLabel label1 = new JLabel("label-1");
		p1.add(label1);

		JLabel label2 = new JLabel("label-2");
		label2.setIcon(new ImageIcon("images/viewBill.png"));
		p1.add(label2);

		JLabel label3 = new JLabel(
			new ImageIcon("images/addBill.png"));
		p1.add(label3);

		getContentPane().add(p1);
	}

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



4.3 Composants d'action

AbstcatButton et JComboBox



4.3.1 Méthodes de la classe AbstractButton

- + void **doClick([ms])**
- + void **setIcon**(Icon)
- + void **setPressedIcon**(Icon)
- + void **setDisabledIcon**(Icon)
- + void **setSelectedIcon**(Icon)
- + void **setDisabledSelectedIcon**(Icon)
- + void **setRolloverIcon**(Icon)
- + void **setRolloverSelectedIcon**(Icon)
- + void **setMnemonic**(int / char)
- + void **setBorderPainted**(true / false)
- + void **setText**(String)
- + String **getText**()
- + void **setSelected**(true / false)
- + boolean **isSelected**()

4.3.2 Evénements de la classe AbstractButton

- + void **addActionListener**(ActionListener l)
- + void **addChangeListener**(ChangeListener l)
- + void **addItemListener**(ItemListener l)

4.3.3 Les boutons

JButton, JToggleButton, BasicArrowButton, MetalScrollButton

Constructeurs :

+ JButton(String text)
+ JButton(Icon icon)
+ JButton(String text, Icon icon)

Evénements :

+ addActionListener(ActionListener)

4.3.4 Les composants de choix

JCheckBox, JRadioButton

Constructeurs du JCheckBox:

+ JCheckBox()
+ JCheckBox(Icon icon)
+ JCheckBox(Icon icon, boolean selected)
+ JCheckBox(String text)
+ JCheckBox(String text, boolean selected)
+ JCheckBox(String text, Icon icon)
+ JCheckBox(String text, Icon icon, boolean selected)

Constructeurs du JRadioButton:

+ JRadioButton()
+ JRadioButton(Icon icon)
+ JRadioButton(Icon icon, boolean selected)
+ JRadioButton(String text)
+ JRadioButton(String text, boolean selected)
+ JRadioButton(String text, Icon icon)
+ JRadioButton(String text, Icon icon, boolean selected)

Exemple :

```
JRadioButton R1 = new JRadioButton("Oui");
JRadioButton R2 = new JRadioButton("Non");
ButtonGroup G1 = new ButtonGroup();
G1.add(R1);
G1.add(R2);
JRadioButton R3 = new JRadioButton("Masculin");
JRadioButton R4 = new JRadioButton("Feminin");
ButtonGroup G2 = new ButtonGroup();
G2.add(R3);
G2.add(R4);
```

Evénements :

```
C1.addActionListener(new ActionListener () {
    public void actionPerformed(ActionEvent e)
    {
        …
    }
});
```

```
C2.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent e)
    {
        …
    }
});
```

```
C3.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent e)
    {
        …
    }
});
```

4.3.5 JComboBox

Constructeurs :

+ JComboBox()
+ JComboBox(ComboBoxModel model)
+ JComboBox(Object[] items)
+ JComboBox(Vector items)

Principales Méthodes :

1. Gestion des Items :

+ void **addItem**(Object anObject)
+ void **insertItemAt**(Object anObject, int index)
+ Object **getItemAt**(int index)
+ int **getItemCount**()
+ void **removeItem**(Object anObject)
+ void **removeItemAt**(int anIndex)
+ void **removeAllItems**()

2. Sélection :

+ void **setSelectedIndex**(int anIndex)
+ int **getSelectedIndex**()
+ void **setSelectedItem**(Object anObject)
+ Object **getSelectedItem**()
+ Object[] **getSelectedObjects**()

3. Configuration :

+ void **setEditable**(boolean aFlag)
+ boolean **isEditable**()
+ void **setMaximumRowCount**(int count)
+ int **getMaximumRowCount**()

4. Modèle de données :

+ void **setModel**(ComboBoxModel aModel)
+ ComboBoxModel **getModel**()

Evénements :

+ void **addActionListener**(ActionListener l)
+ void **addItemListener**(ItemListener aListener)

4.4 Composants de structure

4.4.1 JScrollPane

JScrollPane est un conteneur permettant d'associer des barres de défilement à un composant Swing. Ceci permet de visualiser des composants plus grands que l'espace dans lequel ils sont visualisés. Le composant scrollé doit implémenter l'interface « **Scrollable** ».

Contrairement à JPanel, JScrollPane n'accepte qu'un seul composant. Il forme une vue sur le composant qui devrait être placé dedans à l'aide de la méthode **setViewportView**(Composant), ou passé en paramètre du constructeur de la classe JScrollPane.

Quelques composants Swing doivent nécessairement être inscrits dans un JScrollPane pour avoir un aspect visuel adéquat :

- JTextArea
- JList
- JTable

Les JScrollBars sont caractérisées par une stratégie d'affichage qui peut être :

+ VERTICAL_SCROLLBAR_AS_NEEDED
+ VERTICAL_SCROLLBAR_NEVER
+ VERTICAL_SCROLLBAR_ALWAYS
+ HORIZONTAL_SCROLLBAR_AS_NEEDED
+ HORIZONTAL_SCROLLBAR_NEVER
+ HORIZONTAL_SCROLLBAR_ALWAYS

Constructeurs utiles de la classe JScrollPane :

+ JScrollPane() crée un JScrollPane sans composant vue avec des politiques horizontale et verticale AS_NEEDED
+ JScrollPane(Component vue) Crée un JScrollPane avec composant vue, avec des politiques horizontale et verticale AS_NEEDED
+ JScrollPane(Component vue, int vPolitique, int hPolitique) Crée un JScrollPane avec composant vue, une politique horizontale et une politique verticale
+ JScrollPane(int vPolitique, int hPolitique) Crée un JScrollPane sans composant vue, mais avec une politique horizontale et une politique verticale

Principales Méthodes de la classe JScrollPane :

+ void **setViewPortView**(Component c)
+ void **setVerticalScrollBarPolicy**(int p)
+ void **setHorizontalScrollBarPolicy**(int p)
+ void **setWheelScrollingEnabled**(boolean b)

4.4.2 JTabbedPane

Ce conteneur permet d'afficher plusieurs panneaux qui partagent tous le même espace.

Le conteneur JTabbedPane permet de réaliser un panneau contenant d'autres conteneurs. La navigation entre ces différents conteneurs se fait par l'intermédiaire des onglets de navigation qui peuvent être positionnés à gauche, en haut, à droite ou en bas.

Constructeurs utiles de la classe JTabbedPane :

+ JTabbedPane()

  Crée un panneau à onglets placés en haut.

+ JTabbedPane (int tabPlacement)

  Crée un panneau à onglets placés suivant les constantes suivantes :

- JTabbedPane.**TOP**
- JTabbedPane.**BOTTOM**
- JTabbedPane.**LEFT**
- JTabbedPane.**RIGHT**

+ JToolBar((int tabPlacement, int tabLayoutPolicy):

Le paramètre « tabLayoutPolicy » permet de préciser la stratégie de placement. Les valeurs suivantes sont possibles :

JTabbedPane.**WRAP_TAB_LAYOUT** : Les onglets passent à la ligne.

JTabbedPane.**SCROLL_TAB_LAYOUT** : une barre de défilement apparaît.

On peut ensuite créer les onglets un à un à l'aide de la méthode **addTab**().

Principales Méthodes pour la gestion des onglets :

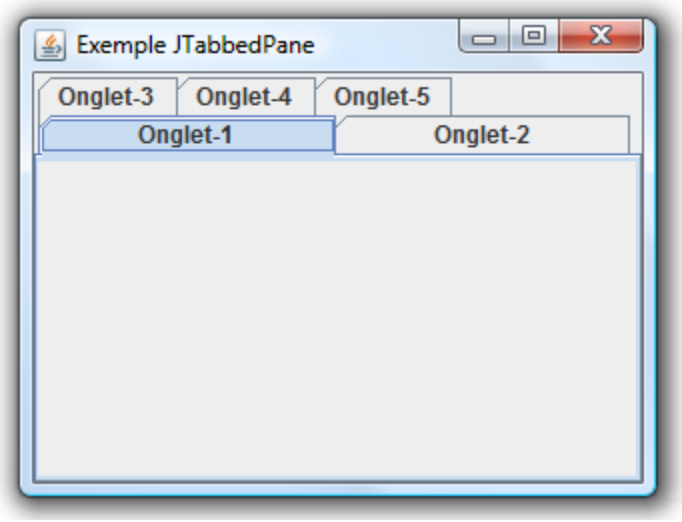
+ void **addTab**(String title, Component component)
+ void **addTab**(String title, Icon icon, Component component)
+ void **addTab**(String title, Icon icon, Component component, String tip)
+ void **insertTab**(String title, Icon icon, Component component, String tip, int index)
+ void **remove**(Component component)
+ void **removeTabAt**(int index)
+ void **setBackgroundAt**(int index, Color background)
+ void **setForegroundAt**(int index, Color foreground)
+ Color **getBackgroundAt**(int index)
+ Color **getForegroundAt**(int index)
+ void **setIconAt**(int index, Icon icon)
+ Icon **getIconAt**(int index)
+ void **setEnabledAt**(int index, boolean enabled)
+ boolean **isEnabledAt**(int index)
+ void **setSelectedIndex**(int index)
+ Component **getSelectedComponent**()
+ int **getSelectedIndex**()
+ int **getTabCount**()
+ void **setTitleAt**(int index, String title)
+ String **getTitleAt**(int index)
+ void **setToolTipTextAt**(int index, String toolTipText)
+ String **getToolTipTextAt**(int index)

Evénement :

+ void **addChangeListener**(ChangeListener l)

Exemple :

```
JTabbedPane tabbedPane = new JTabbedPane(JTabbedPane.TOP);
for (int i = 1; i <= 5; i++) {
        JPanel panel = new JPanel();
        tabbedPane.add("Onglet-" + i, panel);
}
```



4.4.3 JSplitPane

JSplitPane est un conteneur qui permet de diviser une fenêtre en deux parties séparées verticalement ou horizontalement dont les surfaces peuvent changer dynamiquement. La barre de division qui apparaît entre les deux composants peut être déplacée.

L'orientation du JSplitPane se configure en fonction des méthodes de placement utilisées :

- **setLeftComponent**(composant)
- **setRightComponent**(composant)

ou

- **setTopComponent**(composant)
- **setBottomComponent**(composant)

Constructeurs :

+ JSplitPane()

Crée un JSplitPane horizontal et à affichage non continu.

+ JSplitPane(int orientation)

Crée un JSplitPane orienté suivant le paramètre « orientation » et à affichage non continu.

+ JSplitPane(int orientation , boolean continuous)

Crée un JSplitPane orienté suivant orientation et continu si continuous vaut true.

+ JSplitPane(int orientation , boolean continuous, Component cmp1, Component cmp2)

Crée un JSplitPane orienté suivant orientation et continu si continuous vaut true. Les deux composants sont cmp1 et cmp2.

+ **JSplitPane**(int orientation , Component cmp1, Component cmp2) Crée un JSplitPane orienté suivant orientation et à affichage non continu. Les deux composants sont cGauche et cDroit.

Principales Méthodes de la classe JSplitPane :

+ **setContinuousLayout**(true / false)

permet/empêche de redimensionner la taille des deux parties dynamiquement

+ **setOneTouchExpandable**(true)

Permet le double click sur la barre pour la faire apparaître ou disparaître brusquement les composants

+ void **setDividerSize**(int s)

Affecte une taille à la barre de division.

+ int **getDividerSize**()

Retourne la taille de la barre de division.

+ void **setDividerLocation**(int l)

Affecte la position (en valeur absolue) de la barre de division.

+ int **getDividerLocation**()

Retourne la position de la barre de division.

+ void **setDividerLocation**(double d)

Affecte la position (en valeur relative) de la barre de division.

+ void **setOrientation**(int o)

Affecte l'orientation du JSplitPane : JSplitPane.VERTICAL_SPLIT ou JSplitPane.HORIZONTAL_SPLIT

+ int **getOrientation**()

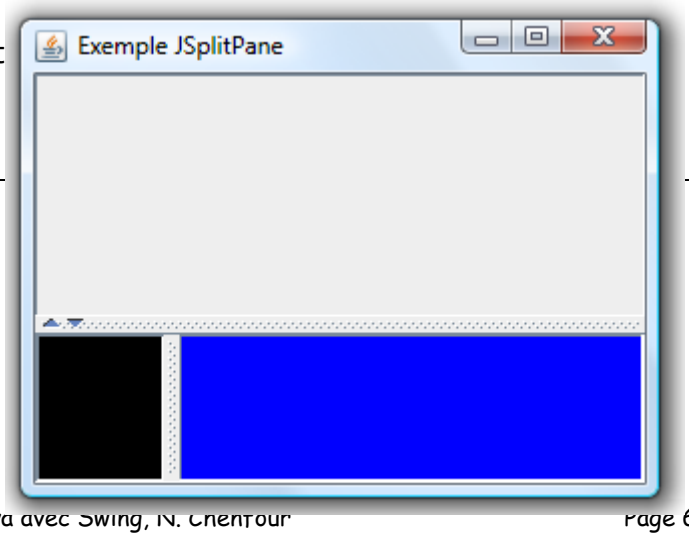
Retourne l'orientation du JSplitPane.

+ void **setResizeweight**(double taux)

Affecte un taux de distribution (valeur entre 0 et 1) au JSplitPane. Les composants 1 et 2 auront respectivement les tailles (taux * taille) et (1-taux) * taille.

Exemple :

```
public class MainFrame extends JFrame {
        ...
        private void build() {
                setTitle("Exemple JSplitPane");
                setSize(320, 240);
                setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                JPanel panel1 = new JPanel();
                panel1.setBackground(Color.BLACK);
                JPanel panel2 = new JPanel();
                panel2.setBackground(Color.BLUE);

                JPanel panel3 = new JPanel();

                JSplitPane splitPane1 = new
                                        JSplitPane(JSplitPane.HORIZONTAL_SPLIT);
                splitPane1.setDividerLocation(50);
                splitPane1.setResizeWeight(0.25);
                splitPane1.setLeftComponent(panel1);
                splitPane1.setRightComponent(panel2);

                JSplitPane splitPane2 = new JSplitPane();
                splitPane2.setDividerLocation(120);
                splitPane2.setOneTouchExpandable(true);
                splitPane2.setOrientation(JSplitPane.VERTICAL_SPLIT);
                splitPane2.setTopComponent(panel3);
                splitPane2.setBottomComponent(splitPane1);

                getContentPane().add(splitPane2);
        }

        public static void main(St
                new MainFrame();
        }
}
```



4.4.4 JSeparator

Un JSeparator permet de séparer différentes zones d'un conteneur (Un panneau par exemple) par l'intermédiaire d'une ligne horizontale ou bien verticale dépendant du choix du développeur.

Constructeurs

+ JSeparator()
+ JSeparator(int orientation)

Principales Méthodes

+ int getOrientation()
+ void setOrientation(int orientation)

4.4.5 JLayeredPane

Constructeur :

+ JLayeredPane();

Principales Méthodes :

+ void **add**(Component[, Object layer]) ;
+ void **add**(Component, int position [, Object layer]) ;

  Avec layer est l'indice (Integer) du Layer

  - Si non précisé ➔ les composants sont insérés (en couches) sur
    Le même Layer (N° 0 par défaut)

  Mais l'apparence des composants est en layer (en couches).

  Ses composants vont occuper les positions 0, 1, ... N-1, -1
    Avec position croissante (0, 1, ...) et indice croissant (0, 1, ...) du haut vers le bas

  - Si précisé ➔ les composants sont insérés (en couches) sur
    des Layers différents (0, -1, -2 , ... –N) de haut vers le bas

Méthode	Position	Index	Layer
LP.add(front) ; LP.add(back) ;	0 1	0 1	0 0
LP.add(front,new Integer(-8)) ; LP.add(back, new Integer(-9)) ;	0 0	0 1	-8 -9
LP.add(front,1) ; LP.add(back,-1) ;	0 1	0 1	0 0
LP.add(front,new Integer(-8),1); LP.add(back,new Integer(-8),-1);	0 1	0 1	-8 -8

On utilise

- + int **getPosition**(Component c);
- + int **getIndexOf**(Component c);
- + int **getLayer**(Component c);

- + void **moveToBack**(Component c) ;
- + void **moveToFront**(Component c) ;

➔ déplace le composant vers front ou back par rapport au même layer.

- + void **setPosition**(Component c, int position) ;

avec position =0 (top), 1, … N-1, -1(bottom)

- + void **setLayer**(Component c, int layer [, int position]) ;

➔ dépose le composant c dans le layer N° layer à la position spécifiée. Si pas position alors le bas du layer.

- + void **setVisible(true/false)**
- + void **remove(int index)** ➔ nécessite LP.repaint

Exemple :

```
JPanel back = new JPanel();
JPanel front = new JPanel();
...
front.setOpaque(false) ;
LP.add(front);
LP.add(back);
```

4.4.6 JInternalFrame et JDesktopPane

Les deux classe JInternalFrame et JDesktopPane sont toujours utilisées conjointement pour réaliser des applications MDI (applications mutli-fenêtres).

+ Une JInternalFrame est une fenêtre interne qui peut géré à l'intérieur d'une JFrame.
+ Le conteneur habituel permettant de gérer les fenêtres internes est un JDesktopPane.

La JInternalFrame contient un JRootPane comme JFrame. On peut donc la gérer comme pour le cas d'une JFrame et avoir accès à son glassPane, contentPane, LayeredPane, et menuBar.

Quelques Constructeurs de la classe JInternalFrame :

+ **JInternalFrame**()

  Crée une JInternalFrame non redimensionnable, n'acceptant pas d'être fermée, non maximisable, non iconifiable et sans titre.

+ **JInternalFrame**(String title)

  Crée une JInternalFrame non redimensionnable, n'acceptant pas d'être fermée, non maximisable et non iconifiable

+ **JInternalFrame**(String title, boolean resizable, boolean closable)

Principales Méthodes de la classe JDeskTopPane :

+ JDesktopPane **getDesktopPane**()
+ void **setFrameIcon**(Icon i)
+ Icon **getFrameIcon**()
+ void **setTitle**(String t)
+ String **getTitle**()
+ void **setClosable**(boolean b)
+ void **setClosed**(boolean b)
+ boolean **isClosed**()
+ void **setIconifiable**(boolean b)
+ boolean **isIconifiable**()
+ void **setIcon**(boolean b)
+ boolean **isIcon**()
+ void **setMaximizable**(boolean b)
+ boolean **isMaximizable**()
+ void **setMaximum**(boolean b)
+ boolean **isMaximum**()
+ void **setResizable**(boolean b)
+ boolean **isResizable**()
+ void **setSelected**(boolean selected)
+ boolean **isSelected**()
+ void **toBack**()
+ void **toFront**()

Principales Méthodes de la classe JDesktopPane

+ JInternalFrame [] getAllFrames()
+ JInternalFrame [] getAllFramesInLayer(int layer)
+ void setSelectedFrame(JInternalFrame f)
+ JInternalFrame getSelectedFrame()
+ DesktopManager getDesktopManager()
+ void setDesktopManager(DesktopManager dm)

Le DesktopManager est responsable de la gestion des fenêtres internes d'un JDesktopPane.

Exemple :

```
public class MDIChild extends JInternalFrame {
	public MDIChild(String Title) {
		super(Title,true,true,true,true);
		setSize(300,200);
		setResizable(true);
		setLocation(20,20);
		...
		setVisible(true);
	}
}

public class MDI extends JFrame implements ActionListener {
	JDesktopPane desktop = new JDesktopPane();

	public MDI() {
		desktop.setBackground(Color.gray);
		getContentPane().add("Center",desktop);
		...
	}

	public void newMDIChild(String title) {
		MDIChild f = new MDIChild(title);
		desktop.add(f,0);
		f.setLocation(0,0);
		desktop.getDesktopManager().maximizeFrame(f);
		try {
			f.setSelected(true);
		}catch (Exception e) {}

	}


	public void activateForm(String title) {
		JInternalFrame T[]=desktop.getAllFrames();
		for(int i=0;i<T.length;i++) {
			if (T[i].getTitle().equals(title)) {
				try {
					T[i].setSelected(true);
				}
				catch(Exception e){}
				return;
			}
		}
	}

	...
}
```

Chapitre 5. Présentation du processus Event Dispatch Thread

5.1 Le Multithreading avec Swing

Malgrais le fait que java supporte le multithreading, il reserve à un seul processus unique la tâche de dessin et d'affichage de tous les composants de l'interface utilisateur. Toutes les méthodes d'affichage place alors leur requete dans la queue des événements ayant à être traité par ce processus. Il s'agit du processus de distribution d'événements, nommé encore : **Event Dispatch Thread ou EDT**

Les opérations de dessin de l'interface ne sont alors pas « thread-safe ». Il est donc nécessaire de comprendre cela pour bien écrire une application Swing multithreads.

Deux principaux problèmes sont alors à gérer :

1. Si on est en train de s'exécuter dans le processus EDT un blocage ou un sommeil dedans (un appel à la méthode sleep de la classe Thread par exemple) bloque le processus d'affichage. Le resultat, pas de rafraîchissement de l'affichage.

2. Si un autre processus essai de mettre à jour l'interface par lui-même sans passer par l'EDT. Un problème de raffraîchissement peut aussi avoir lieu.

5.2 Règles de fonctionnement du processus EDT

Les règles sont les suivantes :

1. Un seul Thread appelé EDT s'occupe du dessin des interfaces graphiques
2. Tout code s'exécutant en réponse aux événements associés aux interfaces est pris en charge automatiquement par le processus EDT
3. Dès qu'un composant a été **réalisé** (càd affiché ou prêt à l'affichage), tout code devant l'affecter ou dépendant de son état **devrait** s'exécuter dans le processus de distribution d'événements l'EDT.

A noter qu'un composant est dit **réalisé** s'il est affiché à l'écran, ou lorsqu'il est prêt à l'être. Ce qui est en rapport avec l'appel de quelques méthodes telques : **setVisible(true)**, **pack()**, **repaint()**, **etc**.

Dès qu'une fenêtre est réalisée, tous les composants qu'elle contient sont alors réalisés. Une autre manière de réaliser un composant est de l'ajouter à un container qui est déjà réalisé.

Dans ce cas, l'ensemble de l'interface graphique peut être construit et affiché dans le thread principal (le « main ») de l'application, tant qu'aucun composant n'a été réalisé.

5.3 Exécution du code d'affichage dans le processus EDT

La plupart des opérations affectant l'interface graphique après son initialisation ont lieu dans l'*event-dispatching thread*. Dès que l'interface est visible, la plupart des programmes sont dirigés par les événements tels que les clicks de souris ou le clavier, qui sont toujours traités dans l'*event-dispatching thread*. Cependant, certains programmes ont besoin d'effectuer des modifications sur l'interface graphique mais ne dépendant pas d'événements usuels.

Dans ce cas, il serait nécessaire d'aiguillonner l'exécution du code vers le processus EDT.

On utilisera alors la classe **EventQueue** (ou bien, on passera indirectement par l'intermédiaire de la classe **SwingUtilities**).

Les méthodes à utiliser sont :

+ **static void invokeAndWait(Runnable runnable)**

Permet d'exécuter la méthode « **run()** » de la runnable par l'intermédiaire du processus EDT tout en bloquant le processus courant jusqu'à ce que le code de la méthode « run() » soit entièrement exécuté.

+ **static void invokeLater(Runnable runnable)**

Permet d'exécuter la méthode « **run()** » de la runnable par l'intermédiaire du processus EDT. A l'inverse de la première méthode, celle-ci retourne immédiatement, sans attendre que le code soit exécuté. C'est généralement celle-ci qui est utilisée

Exemple 1 :

Soit une interface graphique à exécuter à l'aide du processus EDT :

```
public class InterfaceGraphique extends JPanel/JFrame {


}
```

L'exécution sera réalisée come suit :

```
EventQueue.invokeLater(new Runnable() {

        public void run() {

                new InterfaceGraphique();

        }

});
```

Dans ce cas, nous avons instancier une classe d'interface, mais le raisonnement reste le même si on veut juste appeler une méthode ou réliser le traitement directement dans la méthode run().

Un autre Exemple :

```
public class Example2 extends JFrame implements ActionListener {
        private JLabel label;
        private JButton button;
        public Example2() {
                build();
                setDefaultCloseOperation(EXIT_ON_CLOSE);
                setVisible(true);
        }
        private void build() {
                label = new JLabel("Message Initial");
                getContentPane().add("North", label);
                button = new JButton("Modifier");
                button.addActionListener(this);
                getContentPane().add("South", button);
                setSize(200, 100);
        }
```

```
        public void actionPerformed(ActionEvent e) {
                action1();
        }
```

```
        private void action1() {
                label.setText("Nouveau Message!!!");
                try {
                        Thread.sleep(5000);
                } catch (Exception e) {}
                label.setText("Message de Fin!!!");
        }
```

```
        private void action2() {
                label.setText("Nouveau Message!!!");
                Thread p = new Thread() {
                        public void run( ) {
                                try {
                                        Thread.sleep(5000);
                                } catch (Exception e) {}
                                EventQueue.invokeLater(new Runnable( ) {
                                        public void run( ) {
                                                label.setText("Message de Fin!!!");
                                        }
                                });
                        }
                };
                p.start();
        }
```

```
        public static void main(String[] args) {
                new Example2();
        }
}
```

5.4 Rappel sur l'interface Runnable

Un thread peut être créé par implementation de l'interface **Runnable** du package java.lang. Ceci est réalisé par l'intermédiaire de l'un des deux constructeurs suivants qui reçoivent en paramètre un objet d'une classe qui implémente l'interface Runnable :

+ public Thread(**Runnable** target)
+ public Thread(**Runnable** target, String name)

L'interface Runnable déclare une seule méthode « run() » que la classe de l'objet 'target' doit implémenter. La méthode run() sera alors exécutée par le thread créé.

Création d'un processus :

La création d'un processus passe alors par deux étapes :

La première étape consiste à implémenter l'interface Runnable :

```
class Traitement implements Runnable {
        public void run()
        {
                //comportement du processus
        }
}
```

La 2ème étape consiste à créer un thread à base d'un objet de la classe précédente :

```
Tread P = new Thread(new Traitement(), "Nom");
P.start();
```

Autre méthode :

```
class Processus implements Runnable {
        private Thread P;
        Processus(String name)
        {
                P = new Thread(this, name);
                P.start();
        }
        public void run()
        {
                //comportement du processus
        }
}
```

La création est réalisée par l'instruction suivante :

```
Processus P = new Processus("Nom");
```

Chapitre 6. Composants Swing basés sur le design pattern « MVC »

6.1 Etude du composant JList

Constructeurs de la classe JList

- JList()
- JList(ListModel dataModel)
- JList(Object[] listData)
- JList(Vector listData)

Exemple :

```
JList L1 = new JList();
Vector V = new Vector();
void test02()
{
            V.add("PC");
            V.add("Clavier");
            V.add("Ecran");
            V.add("Souris");
            L1.setListData(V);

             p.setLayout(new BoxLayout(p,BoxLayout.Y_AXIS));
            p.add(T1);
            p.add(new JScrollPane(L1));
}
```

```
T1.addActionListener(new ActionListener () {
     public void actionPerformed(ActionEvent e)
     {
       V.add(T1.getText());
       T1.setText("");
        L1.setListData(V);
     }
   });
```

```
   L1.addListSelectionListener(new ListSelectionListener() {
     public void valueChanged(ListSelectionEvent e)
     {
       T1.setText(""+
                    L1.getSelectedValue()+" : "
                    +e.getFirstIndex()+" -> "
                    +e.getLastIndex());
     }
   });
```

```
   L1.addMouseListener( new MouseListener() {
     public void mouseEntered(MouseEvent e) { }
     public void mouseExited(MouseEvent e)  { }
     public void mousePressed(MouseEvent e)  { }
     public void mouseReleased(MouseEvent e)  { }

     public void mouseClicked(MouseEvent e)
     {
       if (e.getClickCount()==2) {
         T1.setText(":: "+L1.getSelectedValue());
       }
     }

   });
```

Principales Méthodes de la classe JList

+ int getFirstVisibleIndex()
+ int getFixedCellHeight()
+ int getFixedCellWidth()
+ int getLastVisibleIndex()
+ int getMaxSelectionIndex()
+ int getMinSelectionIndex()
+ ListModel getModel()
+ int getSelectedIndex()
+ int[] getSelectedIndices()
+ Object getSelectedValue()
+ Object[] getSelectedValues()
+ int getVisibleRowCount()
+ boolean isSelectedIndex(int index)
+ boolean isSelectionEmpty()
+ void setFixedCellHeight(int height)
+ void setFixedCellWidth(int width)
+ void setListData(Object[] listData)
+ void setListData(Vector<?> listData)
+ void setModel(ListModel model)
+ void setSelectedIndex(int index)
+ void setSelectedIndices(int[] indices)
+ void setSelectionInterval(int anchor, int lead)
+ void setVisibleRowCount(int visibleRowCount)

Evénement la classe JList

+ void addListSelectionListener(ListSelectionListener listener)

Utilisation de la classe DefaultListModel

```
JList l1;
DefaultListModel dlm = new DefaultListModel();
l1 = new JList(dlm);
…
dlm.add(t1.getText()) ;
```

Principales méthodes de la classe DefaultListModel

- void **add**(int index, Object element)
- void **addElement**(Object obj)
- void **clear**()
- boolean **contains**(Object elem)
- Object **elementAt**(int index)
- Enumeration<?> **elements**()
- Object **firstElement**()
- Object **get**(int index)
- Object **getElementAt**(int index)
- int **getSize**()
- int **indexOf**(Object elem)
- int **indexOf**(Object elem, int index)
- void **insertElementAt**(Object obj, int index)
- boolean **isEmpty**()
- Object **lastElement**()
- int **lastIndexOf**(Object elem)
- int **lastIndexOf**(Object elem, int index)
- Object **remove**(int index)
- void **removeAllElements**()
- boolean **removeElement**(Object obj)
- void **removeElementAt**(int index)
- void **removeRange**(int fromIndex, int toIndex)
- Object **set**(int index, Object element)
- void **setElementAt**(Object obj, int index)
- int **size**()
- Object[] **toArray**()

Création d'un modèle

```
import java.util.LinkedList;
import javax.swing.AbstractListModel;

public class SimpleListModel extends AbstractListModel {
        private LinkedList list;

        public SimpleListModel() {
                list = new LinkedList();
        }

        public SimpleListModel(String items[]) {
                list = new LinkedList();
                for (int i=0; i<items.length; i++) list.add(items[i]);
        }

        public void addItem(String item) {
                list.add(item);
                this.fireIntervalAdded(this, list.size()-1, list.size()-1);
        }

        public void removeItem(int index) {
                list.remove(index);
                this.fireIntervalRemoved(this, index, index);
        }

        public Object getElementAt(int index) {
                return list.get(index);
        }

        public int getSize() {
                return list.size();
        }

}
```

5.2 Etude du composant JTable

1.1.1 Constructeurs de JTable

+ JTable**()**
+ JTable**(int numRows, int numColumns)**
+ JTable**(Object[][] rowData, Object[] columnNames)**
+ JTable**(TableModel dm)**
+ JTable**(TableModel dm, TableColumnModel cm)**
+ JTable**(TableModel dm, TableColumnModel cm, ListSelectionModel sm)**
+ JTable**(Vector rowData, Vector columnNames)**

Exemple 1 :

```
TableModel dataModel = new AbstractTableModel() {
    public int getColumnCount() {
            return 10;
    }
    public int getRowCount() {
            return 10;
    }
    public Object getValueAt(int row, int col) {
            return new Integer(row*col);
    }
};
JTable table = new JTable(dataModel);
JScrollPane scrollpane = new JScrollPane(table);
```

Exemple 2 :

```
DefaultTableModel  dtm = new DefaultTableModel(10,6);
JTable table = new JTable(dtm);

public String getTextMatrix(int row, int col) {
        return (String)dtm.getValueAt(row, col);
}

public int getColumnCount() {
        return dtm.getColumnCount();
}

public void setTextMatrix(int row, int col, String value) {
        dtm.setValueAt(value, row, col);
}

public void addRow() {
        dtm.addRow((Vector)null);
}

public void addRow(Vector v) {
        dtm.addRow(v);
}
```

1.1.2 Méthodes de JTable

+ Class **getColumnClass**(int column)
+ TableCellRenderer **getDefaultRenderer**(Class columnClass)

+ TableColumnModel **getColumnModel**()

Exemples :

```
public void setColumnWidth(int num, int size) {
        table.getColumnModel().getColumn(num).setPreferredWidth(size);
}

public void setColumnTitle(int num, String title) {
        table.getColumnModel().getColumn(num).setHeaderValue(title);
}
public int getColumnWidth(int num) {
        return table.getColumnModel().getColumn(num).getWidth();
}

public String getColumnTitle(int num) {
        return
    (String)table.getColumnModel().getColumn(num).getHeaderValue();
}
```

+ Void setPreferredScrollableViewportSize(Dimension size)

Exemple :

```
public void setTableSize(Dimension d){
   table.setPreferredScrollableViewportSize(d);
}
```

+ JTableHeader getTableHeader()

Exemples :

```
public void setReorderingAllowed(boolean state) {
     table.getTableHeader().setReorderingAllowed(state);
}

public void setBackgroundFixed(Color c) {
     table.getTableHeader().setBackground(c);
}

public void setForegroundFixed(Color c) {
   table.getTableHeader().setForeground(c);
}

public void setResizable(boolean state) {
     table.getTableHeader().setResizingAllowed(state);
}
```

1.1.3 Autres Méthodes

+ int getColumnCount()
+ String getColumnName(int column)
+ TableModel getModel()
+ int getRowCount()
+ int getRowHeight(int row)
+ int getSelectedColumn()
+ int getSelectedColumnCount()
+ int[] getSelectedColumns()
+ int getSelectedRow()
+ int getSelectedRowCount()
+ int[] getSelectedRows()
+ Object getValueAt(int row, int column)
+ void setDefaultRenderer(Class columnClass, TableCellRenderer renderer)
+ void setIntercellSpacing(Dimension intercellSpacing)
+ void setModel(TableModel dataModel)
+ void setRowHeight(int row, int rowHeight)
+ void setShowGrid(boolean showGrid)
+ void setShowHorizontalLines(boolean showHorizontalLines)
+ void setShowVerticalLines(boolean showVerticalLines)
+ void setValueAt(Object aValue, int row, int column)

DefaultTableModel

1.1.4 Constructeurs de la classe DefaultTableModel

+ DefaultTableModel()
+ DefaultTableModel(int rowCount, int columnCount)
+ DefaultTableModel(Object[][] data, Object[] columnNames)
+ DefaultTableModel(Object[] columnNames, int rowCount)
+ DefaultTableModel(Vector columnNames, int rowCount)
+ DefaultTableModel(Vector data, Vector columnNames)

1.1.5 Méthodes de la classe DefaultTableModel

+ void addColumn(Object columnName)
+ void addColumn(Object columnName, Object[] columnData)
+ void addColumn(Object columnName, Vector columnData)
+ void addRow(Object[] rowData)
+ void addRow(Vector rowData)
+ int getColumnCount()
+ String getColumnName(int column)
+ Vector getDataVector()
+ int getRowCount()
+ Object getValueAt(int row, int column)
+ void insertRow(int row, Object[] rowData)
+ void insertRow(int row, Vector rowData)
+ boolean isCellEditable(int row, int column)
+ void moveRow(int start, int end, int to)
+ void removeRow(int row)
+ void setColumnCount(int columnCount)
+ void setNumRows(int rowCount)
+ void setRowCount(int rowCount)
+ void setValueAt(Object aValue, int row, int column)

5.3 Etude du composant JTree

Le composant JTree permet de présenter des données sous une forme hiérarchique arborescente.

Liste de Constructeurs

+ public JTree();
+ public JTree(Hashtable value);
+ public JTree(Vector value);
+ public JTree(Object[] value);
+ public JTree(TreeModel model);
+ public JTree(TreeNode rootNode);
+ public JTree(TreeNode rootNode, boolean askAllowsChildren);

Principales Méthodes :

+ **setRootVisible(boolean)**

Afficher la racine avec la possibilité de la refermer ou de l'étendre.

+ **setShowsRootHandles(boolean)**

Ajouter ou supprimer le commutateur.

+ **setModel()**

permet d'associer un modèle de données à l'arbre

+ **setEditable(boolean)**

autoriser la modification des étiquetes des nœuds

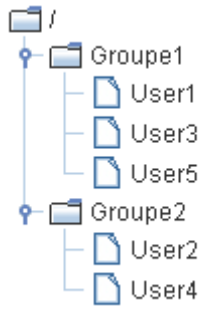
+ **isPathEditable()**

permet de définir quels sont les noeuds de l'arbre qui sont éditables en créant une classe fille de la classe JTree et en redéfinissant la méthode isPathEditable().

Exemples :

```
JTree t1 = new JTree(new String[]{"A", "B", "C"});
```


```
DefaultMutableTreeNode root
        = new DefaultMutableTreeNode("/");

DefaultMutableTreeNode groupes[] = {
        new DefaultMutableTreeNode("Groupe1"),
        new DefaultMutableTreeNode("Groupe2")
};

for (int i=0; i<groupes.length; i++) root.add(groupes[i]);

DefaultMutableTreeNode users[] = {
        new DefaultMutableTreeNode("User1"),
        new DefaultMutableTreeNode("User2"),
        new DefaultMutableTreeNode("User3"),
        new DefaultMutableTreeNode("User4"),
        new DefaultMutableTreeNode("User5")
};

for (int i=0; i<users.length; i++) groupes[i%2].add(users[i]);

JTree t1 = new JTree(root);
```



Principaux Evénements :

+ **void addTreeExpansionListener(TreeExpansionListener tel)**
+ **void addTreeSelectionListener(TreeSelectionListener tsl)**

avec :

TreeExpansionListener

+ **void treeCollapsed(TreeExpansionEvent event)**
+ **void treeExpanded(TreeExpansionEvent event)**

TreeExpansionEvent

+ **TreePath getPath()**

TreeSelectionListener

+ **void valueChanged(TreeSelectionEvent e)**

TreeSelectionEvent

+ **TreePath getPath()**

TreePath

+ **Object[] getPath()**
+ **Object getPathComponent(int element)**
+ **int getPathCount()**
+ **Object getLastPathComponent()**
+ **TreePath getParentPath()**

Exemple :

```
JTree t1 = new JTree(root);
t1.addTreeSelectionListener( new TreeSelectionListener() {
        public void valueChanged(TreeSelectionEvent e) {
                System.out.println(
                        e.getPath().getLastPathComponent()
                        + " (" + e.getPath().getPathCount() +")"
                );
        }
});
```