Formation JAVA

Cours N°2
Interfaces graphiques utilisateur (GUI)

Version 1.8 - Juin 2007

Génaël VALET

GITA - 61, rue David d'Angers – 75019 PARIS
Tél. : 01.40.40.36.27/28---- Fax : 01.40.40.36.30
Email : infos@gita.greta.fr , http://www.gita.greta.fr
N° SIRET 197 507 122 00046 – code APE 804

Ce document permet de connaître quelques classes utilisées couramment dans les JFC (Java Fondation Classes) pour créer des interfaces graphiques évoluées. Il est indispensable d'étudier ce document en ayant la documentation des API Java sous la main. Ceci dans le but d'obtenir des compléments nécessaires à l'implémentation de certains objets graphiques. Ce document n'est pas un recueil exhaustif sur les classes mais il vous aidera sans doute à mieux comprendre l'architecture des classes JFC.

Table des matières

GITA - 61, rue David d'Angers – 75019 PARIS
Tél. : 01.40.40.36.27/28---- Fax : 01.40.40.36.30
Email : infos@gita.greta.fr , http://www.gita.greta.fr
N° SIRET 197 507 122 00046 – code APE 804

Chapitre 1 : Introduction

ATTENTION : Tous les programmes d'exemple fournis dans ce document sont implémentés de la même façon, autour d'une classe chargée de tout faire. Dans la réalisation de vos programmes, vous apprendrez à créer plusieurs classes ayant chacune une fonction bien précise. Le dernier chapitre de ce document vous aidera à construire vos propres applications graphiques.

Le monde de l'informatique, quel que soit le système, utilise une interface graphique qui permet à l'utilisateur d'accéder aux fonctions du logiciel. Les développeurs de logiciels consacrent beaucoup de temps à la conception des ces ***GUI***[1] et l'ergonomie en dépend.
Maintenant que vous avez une bonne connaissance des structures fondamentales du langage, il est grand temps de s'en servir au profit de la réalisation des ces fameuses GUI.
Ce document va présenter plusieurs aspects de la programmation graphique :

- Les différentes classes utilisées pour les couleurs, les polices de caractère ainsi que les outils de dessin et les images.
- Les classes ***SWING***[2] et les composants de l'interface utilisateur tels que les boutons, les listes, les menus.
- La gestion des évènements en Java
- La création d'***applets***[3] pour le Web et la compatibilité avec le plug-in Java

A. Classes AWT (Abstract Windowing Toolkit)

Avant l'arrivée de *Java 2*, les programmeurs avaient à leur disposition des classes *AWT* qui leur permettaient déjà de satisfaire leurs exigences. D'ailleurs même sous *Java 2*, ces classes gèrent encore tout le mécanisme sous jacent du fenêtrage ou la gestion des évènements.
Les nouvelles classes *JFC* ou *SWING* proposent de simplifier et de compléter l'étendue de composants. Ces classes, par l'intermédiaire de l'héritage sont en fait, une extension des classes *AWT*.
Tout cela pour dire que vous utiliserez des classes *AWT* indirectement en utilisant les classes *SWING*.

B. Classes SWING ou JFC (Java Fundation Classes)

B.1. Comparaison

Même si les classes *SWING* sont un peu plus lentes à l'exécution et que, de plus, elles ne sont pas encore disponibles sur toutes les plate-formes Java, elles ont le mérite d'avoir les avantages suivants :

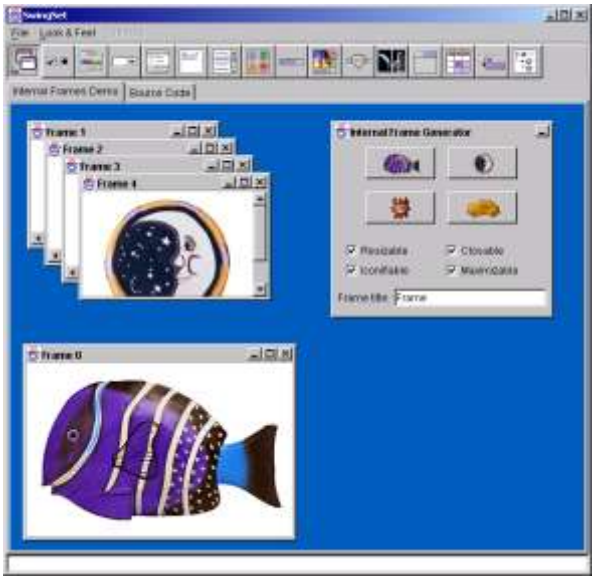
- Elles rendent le développement de GUI plus facile tout en proposant plus de classes
- Elles sont moins dépendantes de la plate-forme d'exécution surtout qu'elles proposent des gestionnaires de contenu pouvant s'adapter automatiquement aux caractéristiques de système d'exécution.

[1] Graphic User Interface
[2] Nom utilisé pour la dernière génération des composants graphiques de JAVA 2. Aussi appelés JFC (Java Fundation Classes)
[3] Petite application JAVA fonctionnant dans une page Web

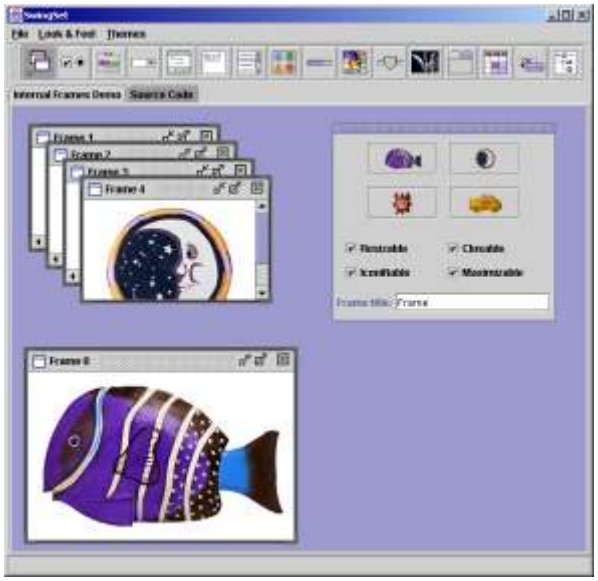
- Elles ont tendance à se généraliser à l'ensemble des plates-formes Java ce qui constitue pour le développeur un net avantage s'il souhaite travailler sur plusieurs environnements différents.

B.2. Look And Feel

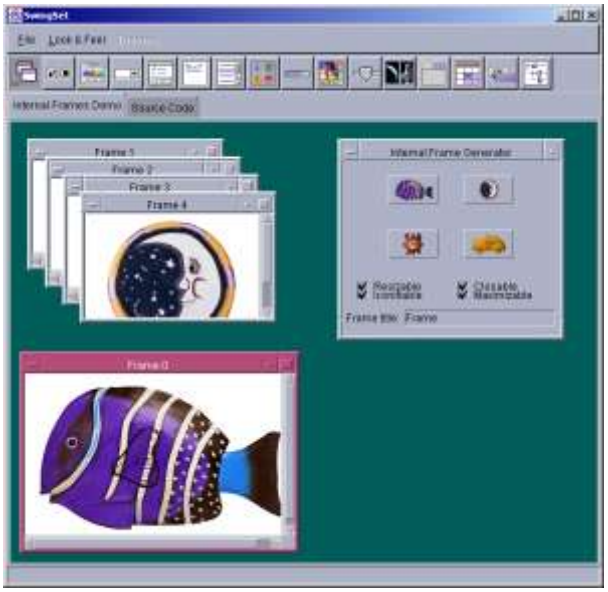
Ce terme désigne l'aspect que vont avoir tous les composants graphiques. Java 2 autorise l'utilisation de *Look & Feel* différents qui ressemblent à ce que l'on peut trouver dans les principaux systèmes d'exploitation tels que *Windows , Unix* ou *Mac OS …*
Les figures suivantes présentent un aperçu des différents *L&F* :



L&F Windows



L&F Java



L&F Motif

➲ Pour tester les L&F sous tous les angles, le kit de développement fourni par Sun (SDK version 1.3)vous propose une application Java située dans le répertoire ***\demo\SwingSet2\ .*** Il vous suffit de taper la commande suivante pour l'exécuter. De plus, l'application vous propose le CODE SOURCE de tous les composants utilisés !!!

```
java SwingSet2
```

Chapitre 2 : Modèle de conception graphique

Les classes graphiques de Java sont très nombreuses mais il existe trois principaux types d'objets que l'on peut identifier facilement :

- ***Component :*** Le composant peut être un bouton, une liste, un champ de texte ou encore une case à cocher.
- ***Container :*** Le conteneur qui est un composant contient d'autres composants. Il peut-être une fenêtre, un panneau ou une boîte de dialogue.
- ***Layout Manager :*** Le gestionnaire de contenu qui va gérer la façon dont les composants sont placés dans les conteneurs. Il existe plusieurs types de gestionnaire de contenu *BorderLayout , GridLayout, ...)*

Voici un diagramme objet qui résume le modèle :

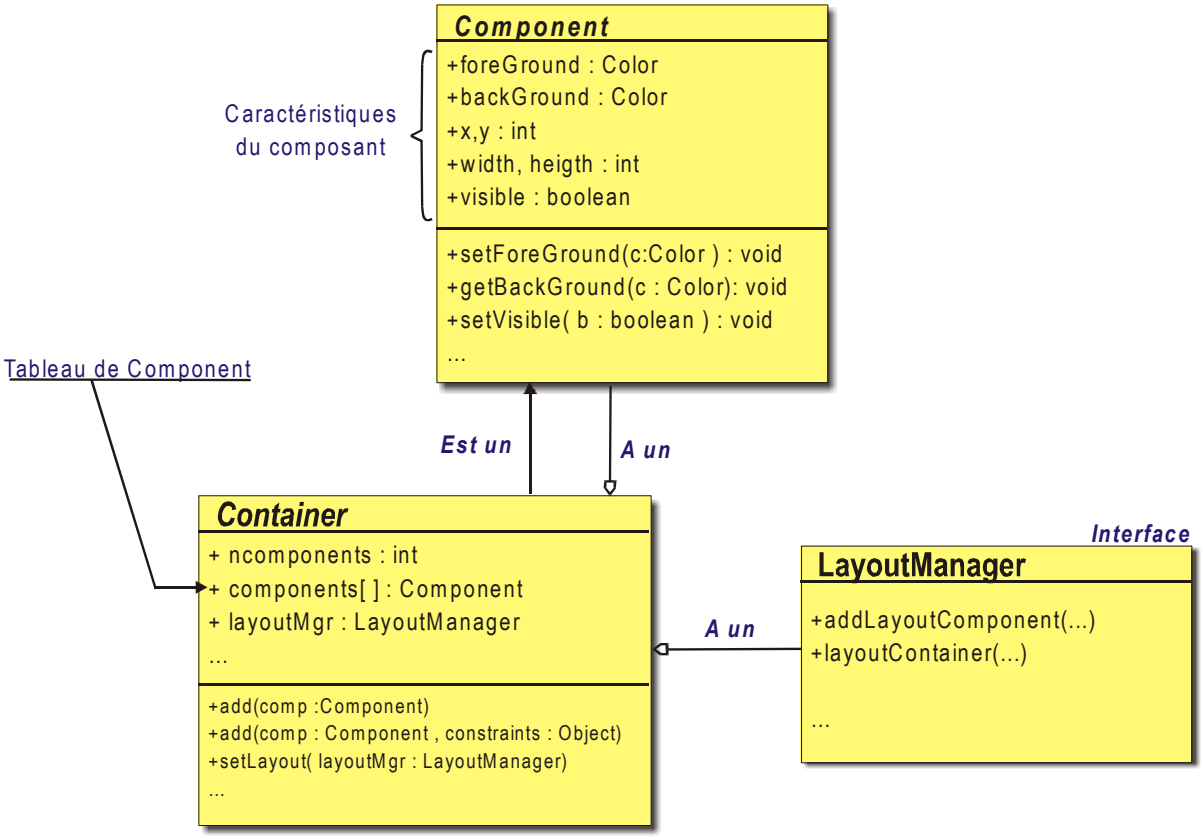


Diagramme objet des composants graphiques sous Java

Vous remarquerez que même si *Container* est un *Component*, cela ne l'empêche pas de posséder des *Component*.

➲ Si vous parcourez l'API Java, vous ne verrez pas tous les champs de ces classes. En fait, leur nombre est si important que la plupart n'ont pas été documentés grâce à l'outil ***javadoc*.** Si vous souhaitez les voir, il faut éditer le code source des classes qui se situent dans le répertoire du JDK dans le fichier compressé ***src.jar***

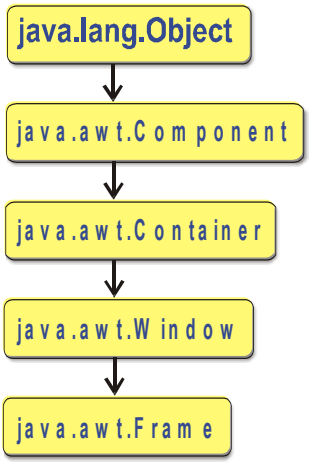
A. Fenêtrage avec SWING

Qu'ils soient fournis par *AWT* ou *JFC*, ces conteneurs héritent de la classe *Container* ou d'une de ses sous classes. Cette partie ne présente pas de manière exhaustive les conteneurs mais plutôt ceux qui sont utilisés le plus fréquemment.

A.1. JFrame

a. Chaîne d'héritage

C'est le premier conteneur que vous utiliserez puisque votre application logera certainement dans une fenêtre. Voici le diagramme d'héritage de *JFrame* :



Issues des classes *AWT*, les classes *Frame* et *Window* constituent l'ossature de la gestion de fenêtrage en Java.

> La plupart des classes issues de *JFC ou SWING* ont la lettre J devant leur nom de manière à distinguer leur provenance.

JFrame ne fait pas exception à la règle et la plupart des méthodes intéressantes sont accessibles à partir de *Frame* ,voir de *Window*.

Afin de vous éviter une recherche fastidieuse dans les API Java, n'oubliez jamais que la méthode que vous recherchez n'est pas forcément documentée dans la classe elle-même s'il s'agit d'une méthode appartenant au parent. Apprenez donc à chercher dans la hiérarchie les informations souhaitées. Voici toutes les méthodes héritées de **Frame** *et* **Window** *et que les objets* **JFrame** *peuvent utiliser :*

Methods inherited from class java.awt.Frame
addNotify, finalize, getCursorType, getFrames, getIconImage, getMenuBar, getState, getTitle, isResizable, remove, removeNotify, setCursor, setIconImage, setMenuBar, setResizable, setState, setTitle

Méthodes héritées de Frame

Methods inherited from class java.awt.Window
addWindowListener, applyResourceBundle, applyResourceBundle, dispose, getFocusOwner, getGraphicsConfiguration, getInputContext, getListeners, getLocale, getOwnedWindows, getOwner, getToolkit, getWarningString, hide, isShowing, pack, postEvent, processEvent, removeWindowListener, setCursor, show, toBack, toFront

Méthodes héritées de Window

Methods inherited from class java.awt.Container
add, add, add, add, add, addContainerListener, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getAlignmentX, getAlignmentY, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getInsets, getLayout, getMaximumSize, getMinimumSize, getPreferredSize, insets, invalidate, isAncestorOf, layout, list, list, locate, minimumSize, paint, paintComponents, preferredSize, print, printComponents, processContainerEvent, remove, removeAll, removeContainerListener, setFont, validate, validateTree

Méthodes héritées de Container

Methods inherited from class java.awt.Component
action, add, addComponentListener, addFocusListener, setBounds, setComponentOrientation, setDropTarget, setEnabled, setForeground, setLocale, setLocation, setLocation, setName, **setSize**, **setSize**, **setVisible**, show, size, toString, transferFocus

Liste PARTIELLE des méthodes héritées de Window

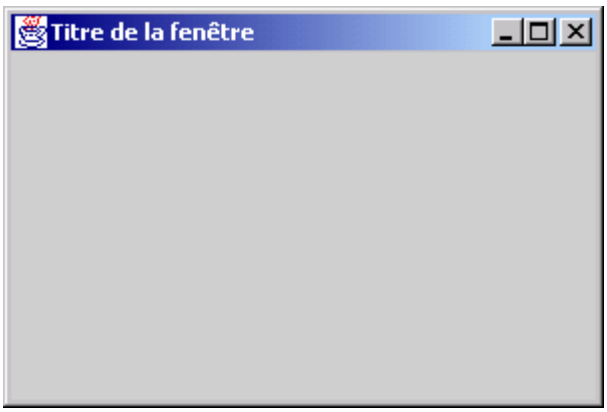
b. Créer des objets JFrame

Rien de plus simple :

```
1  import javax.swing.*;
2
3  public class Ex1 {
4
5          public static void main ( String [] args ) {
6
7          JFrame myFrame = new JFrame("Titre de la fenêtre");
8
9          myFrame.setVisible(true);    // Rend la fenêtre visible
10         myFrame.setSize(300,200);    // Fixe la taille
11         }
12 }
```

Ex1.java

Les méthodes setVisible(**boolean** b) et setSize(**int** width , **int** heigth) sont des méthodes de la classe « Component ».
Le résultat donné par cet exemple est le suivant :



Exemple d'utilisation de JFrame

c. Créer des JFrame en tant qu'application

Dans la plupart des applications, la classe *JFrame* n'est pas utilisée directement. Le développeur préférera dériver la classe *JFrame* pour créer sa propre fenêtre. L'avantage d'un tel système, est de pouvoir gérer la fenêtre comme le centre nerveux de toute l'application. La gestion des évènements est alors centralisée par la fenêtre. Nous reverrons la gestion des évènements plus loin dans ce cours.
Voici ce que donne le même programme précédent :

```
1  import javax.swing.*;

public class Ex2 extends JFrame {

            // Constructeur
            public Ex2(String titre) {

            setTitle(titre);
            setSize(300,200);
            setVisible(true);
            }


            public static void main ( String [] args ) {

            Ex2 myFrame = new Ex2("JFrame en tant qu'application");
            }
}
```

Fichier Ex2.java

- **Ligne 3** : En général, le nom de classe – ici Ex2 – correspond au nom de votre application graphique. Cette classe est considérée comme une fenêtre puisqu'elle hérite de la classe *JFrame*.
- **Ligne 14** : La méthode *main* , toujours statique, est appelée par la machine virtuelle lors de l'exécution du programme.
- **Ligne 16** : La classe *Ex2* s'instancie elle-même.

d. Problème de fermeture

Vous remarquerez que l'exemple précédent pose un problème lors de la fermeture de la fenêtre. Le clic sur l'icône de fermeture ne quitte pas l'application mais se contente de **cacher la**

fenêtre. Le processus continue de réquisitionner de la mémoire, ce qui peut devenir très gênant dans le cas de grosses applications.
La solution consiste à écouter l'événement de clic sur cet icône et d'exécuter une instruction fermant proprement l'application. Java va alors, vider la mémoire occupée par tous les objets de votre programme.
Dans la pratique, nous utiliserons une classe interne , dont le rôle sera de gérer l'événement de fermeture de la fenêtre. Nous allons dériver la classe *WindowAdapter* et référencer l'événement de fermeture auprès de la fenêtre :

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


public class Ex3 extends JFrame {

            public Ex3(String titre) {

            setTitle(titre);
            setSize(300,200);
            setVisible(true);

            WindowCloser wc = new WindowCloser();

            // Ajout du gestionnaire d'évènement auprès de la fenêtre
            addWindowListener(wc);
            }

            static void main ( String [] args ) {

            // instanciation directement au sein de la classe
            Ex3 myFrame = new Ex3("JFrame en tant qu'application");
            }

            // Classe interne
            public class WindowCloser extends WindowAdapter {
                 public void windowClosing(WindowEvent we) {
                           System.exit(0);
                 }
            }
}
```

Fichier Ex3.java

- **Ligne 26 à 30** : Définition de la classe interne *WindowCloser* contenant une seule méthode *windowClosing* qui sera appelée chaque fois que l'on cliquera sur le bouton de fermeture de la fenêtre.
- **Ligne 13** : Création d'un objet de type *WindowCloser* nommé *wc*
- **Ligne 16** : L'objet *wc* devient, par cette intruction, responsable de la fermeture de la fenêtre.

La gestion de la fermeture de l'application implique l'utilisation de classes telles que ***WindowAdapter***. Cette classe implémente l'interface ***WindowListener***. En regardant de plus près cette interface, on s'aperçoit qu'elle contient 7 méthodes concernant tous les évènements pouvant se produire sur une fenêtre. La classe ***WindowAdapter*** nous permet de sélectionner seulement un ou plusieurs des évènements proposés en implémentant la méthode correspondante. Voici la liste des méthodes possibles :

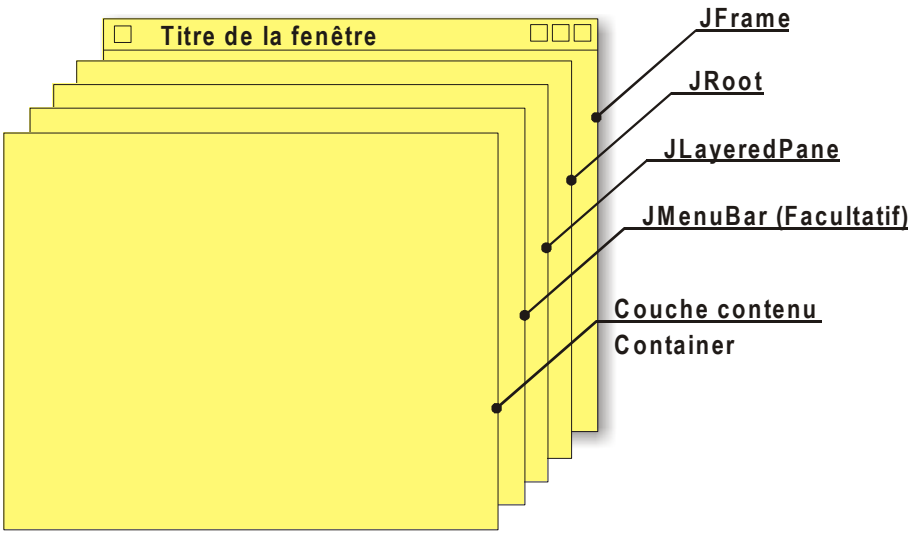
Method Summary	
void	**windowActivated**(WindowEvent e) Appelée lorsque la fenêtre est sur le point de devenir la fenêtre active, ce qui signifie qu'elle ou un de ses composants peut recevoir des évènements provoqués par le clavier.
void	**windowClosed**(WindowEvent e) Appelée lorsqu'une fenêtre a été fermée suite à l'appel a *dispose()*.
void	**windowClosing**(WindowEvent e) Appelée quand l'utilisateur tente de fermer la fenêtre depuis le menu système ou par un clic sur l'icône de fermeture.
void	**windowDeactivated**(WindowEvent e) Appelée lorsqu'une fenêtre n'est plus la fenêtre active, ce qui signifie que les évènements clavier ne seront plus notifiés à la fenêtre ou a ses composants.
void	**windowDeiconified**(WindowEvent e) Appelée lorsque la fenêtre passe de l'état d'icône à l'état normal.
void	**windowIconified**(WindowEvent e) Appelée lorsque la fenêtre passe de l'état normal à l'état d'icône.
void	**windowOpened**(WindowEvent e) Appelée la première fois que la fenêtre est rendue visible

Liste des méthodes appelées lorsqu'un événement se produit sur un objet de type JFrame

Pour plus d'informations sur la gestion des évènements, consultez le chapitre Gestion des évènements plus loin dans ce cours.

e. *Structure multi-couches de* JFrame

Voici un diagramme expliquant la structure de couches de l'objet *JFrame* :

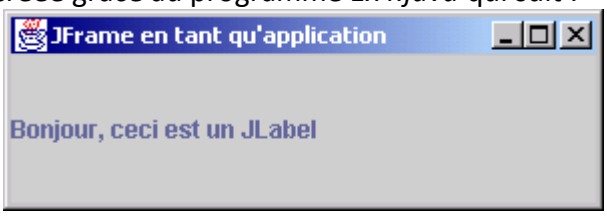


Modèle interne de JFrame

Même si elle s'utilise simplement, la classe *JFrame* a une structure interne complexe. Il est inutile de détailler chaque couche de *JFrame* , puisque la plus intéressante est la couche de contenu. C'est dans cette couche que vont s'insérer tous les composants de la fenêtre. Elle est représentée par un objet de type *Container*
Pour cela, nous aurons recours à la méthode *getContentPane()* :

Container	**getContentPane**() Renvoie le conteneur représentant la couche contenu de l'objet JFrame

Voici la fenêtre qui sera créée grâce au programme *Ex4.java* qui suit :



Fenêtre avec du texte

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


public class Ex4 extends JFrame {

            // Champ de type JLabel
            private JLabel texte;

            public Ex4(String titre) {

            setTitle(titre);
            setSize(300,100);

            // Récupération de la couche contenu
            Container contentPane = getContentPane();

            // Création du texte
            texte = new JLabel("Bonjour, ceci est un JLabel");
```

```
            // Ajout dans le contentPane
            contentPane.add(texte);

            setVisible(true);

            // Ajout du gestionnaire d'évènement auprès de la fenêtre
            addWindowListener(new WindowCloser());
            }

            public static void main ( String [] args ) {

            // instanciation directement au sein de la classe
            Ex4 myFrame = new Ex4("JFrame en tant qu'application");
            }

            // Classe interne
            public class WindowCloser extends WindowAdapter {
                 public void windowClosing(WindowEvent we) {
                           System.exit(0);
                 }
            }
}
```

Ex4.java

- **Ligne 17** : la méthode *getContentPane* récupère l'objet *Container* servant à ajouter des composants.
- **Ligne 23** : Ajout du texte dans la couche contenu récupérée

A.2. JPanel

L'exemple précédent a un inconvénient majeur, lorsqu'il s'agit de placer plusieurs composants. En effet, la grande force des classes *SWING* est de pouvoir adapter la taille et l'agencement des composants en fonction de l'espace disponible. Les panneaux sont très utiles pour gérer le placement de plusieurs composants.
Le conteneur *JPanel* dispose d'une méthode *add* utilisée pour ajouter des composants. Cette méthode lui vient de la classe *Container*. Rappelez-vous que tout conteneur peut contenir des composants :

Component	**add**(Component comp) Ajoute le composant spécifié à la fin

L'exemple *Ex4.java* (*Voir page précédente*) donnera , avec un *JPanel* :

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


public class Ex5 extends JFrame {

            // Champ de type JButton
            private JLabel texte;

            public Ex5(String titre) {

            setTitle(titre);
```

```
            setSize(300,100);

            // Récupération de la couche contenu
            Container contentPane = getContentPane();


            JPanel panel = new JPanel();
            // Ajout dans le contentPane
            contentPane.add(panel);

            // Création du texte
            texte = new JLabel("Bonjour, ceci est un JLabel");

            // Ajout du texte dans le JPanel
            panel.add(texte);

            setVisible(true);

            // Ajout du gestionnaire d'évènement auprès de la fenêtre
            addWindowListener(new WindowCloser());
            }

            static void main ( String [] args ) {

            // instanciation directement au sein de la classe
            Ex5 myFrame = new Ex5("JFrame en tant qu'application");
            }

            // Classe interne
            public class WindowCloser extends WindowAdapter {
                 public void windowClosing(WindowEvent we) {
                           System.exit(0);
                 }
            }
}
```

Ex5.java

➲ Les ***JPanel*** ne se révèlent vraiment intéressants que lorsqu'ils sont utilisés avec des gestionnaires de mise en forme tels que ***BorderLayout*** . Voir plus loin ***Gestion de la mise en forme***.

B. Gestion de la mise en forme

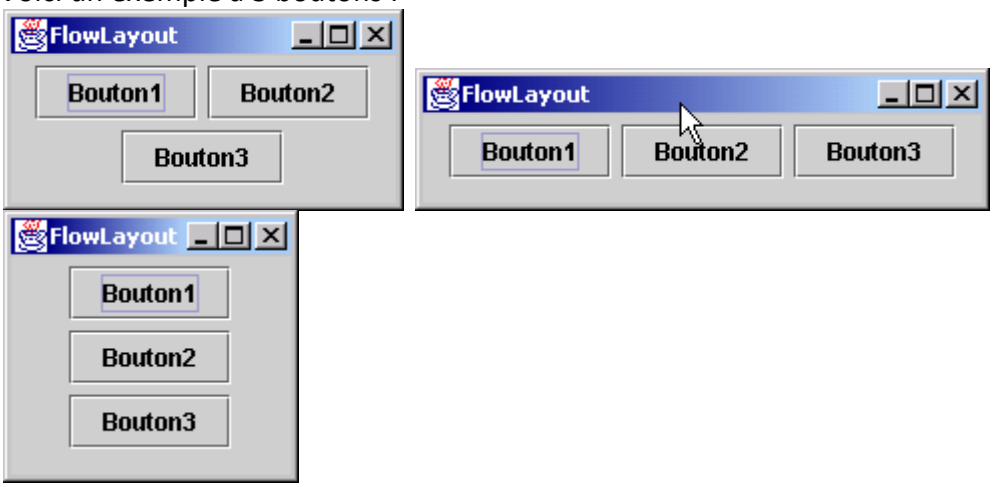
Au début de ce chapitre, nous parlions du modèle de conception graphique et que chaque conteneur est associé avec un gestionnaire de contenu (*LayoutManager*).
Il existe plusieurs types de gestionnaires répondant aux besoins de placement de composants à l'intérieur d'une fenêtre ou d'un panneau.

B.1. Gestionnaire *FlowLayout*

C'est le plus simple et aussi le gestionnaire par défaut de tout conteneur. En effet, si vous ne spécifiez pas de gestionnaire, c'est un *FlowLayout* qui sera implémenté.

a. Principe

Ce gestionnaire place les composants dans l'ordre dans lequel ils sont ajoutés au conteneur. De plus, il organise l'espace en lignes. Dès qu'une ligne est remplie, il en commence une autre.
Voici un exemple a 3 boutons :



Lorsque l'utilisateur redimensionne la fenêtre, les composants se replacent dynamiquement. De même, si vous ajoutez un bouton, le gestionnaire réagit immédiatement :

Le code source de ce programme est dans le fichier ***Ex6.java*.**

Pour mettre en place ce type de gestionnaire, il suffit d'utiliser la méthode *setLayout* du conteneur :

```
1  JPanel panel = new JPanel();     // Le conteneur
panel.setLayout( new FlowLayout() );
```

ou

```
1  FlowLayout lmanager = new FlowLayout();
JPanel panel = new JPanel();   // Le conteneur
panel.setLayout( lmanager );
```

ou plus simplement

```
1  JPanel panel = new JPanel(new FlowLayout() );
```

Dans cet exemple, le conteneur est un *JPanel*.

b. Alignement et intervalle

Il est possible d'aligner les composants à gauche, à droite ou au centre grâce au constructeur ou à la méthode *setAlignment* :

Constructeurs de FlowLayout
FlowLayout()   Initialise un FlowLayout avec un alignement centré et un intervalle de 5 unités vertical et horizontal.
FlowLayout(int align)   Initialise un FlowLayout avec l'alignement spécifié (Voir champs statiques) et un intervalle de 5 unités vertical et horizontal.
FlowLayout(int align, int hgap, int vgap)   Initialise un FlowLayout avec l'alignement spécifié (Voir champs statiques) et les alignements spécifiés. Des valeurs négatives pour les alignements provoquent un chevauchement.

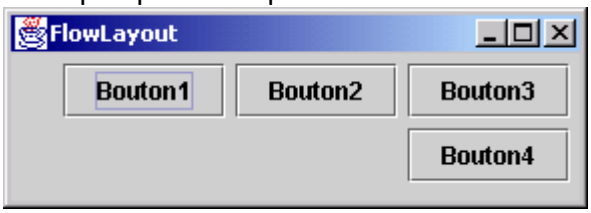
Constructeurs de FlowLayout (API Java2)

Champs statiques de FlowLayout	
static int	**CENTER** Cette valeur indique que chaque ligne de composants doit être centrée.
static int	**LEADING** Cette valeur indique que chaque ligne doit être justifiée à droite en fonction de l'orientation du conteneur dans le cas d'orientations de gauche à droite
static int	**LEFT** Cette valeur indique que chaque ligne de composants doit être justifiée à gauche
static int	**RIGHT** Cette valeur indique que chaque ligne de composants doit être justifiée à droite
static int	**TRAILING** Cette valeur indique que chaque ligne doit être justifiée à gauche

	en fonction de l'orientation du conteneur dans le cas d'orientations de gauche à droite

Champs statiques de FlowLayout (API Java2)

Voici quelques exemples d'utilisation du *FlowLayout :*



Alignement à droite , intervalle horizontal et vertical de 5

```
1  JPanel panel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
```

ou

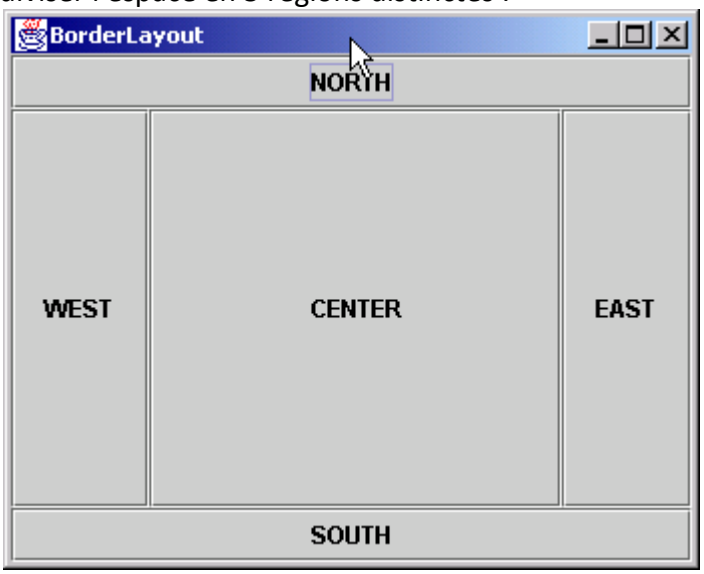

```
1  JPanel panel = new JPanel(new FlowLayout(FlowLayout.CENTER, 1 ,2));
```

Les exemples complets sont stockés dans le fichier **Ex7.java**

B.2. Gestionnaire BorderLayout

a. Principe

Pour un agencement plus complexe, l'utilisation du gestionnaire *BorderLayout* permet de diviser l'espace en 5 régions distinctes :



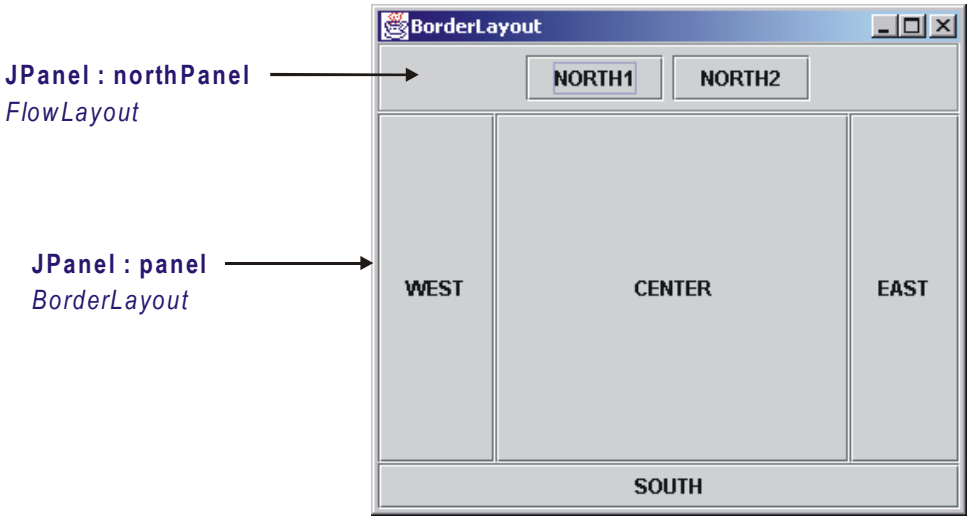
Ex8.java

Les composants des régions en bordure sont placés en premier. L'espace restant est occupé par la région centrale. En cas de redimensionnement, c'est la région centrale qui est changée. Les autres régions restent intactes.

➲ Contrairement au gestionnaire ***FlowLayout*** , le gestionnaire ***BorderLayout*** augmente la taille des composants pour remplir l'espace disponible.

b. En association avec un JPanel

L'inconvénient de chaque région est de ne pouvoir accueillir qu'un seul composant. Pour résoudre ce problème, il vous suffit d'ajouter un *JPanel* à la région concernée et d'y ajouter les composants :



Voici le code de cet exemple :

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


public class Ex9 extends JFrame {

            // Champ de type JButton
            private JButton l1,l1bis , l2 , l3, l4, l5;

            // Champ de type Border
            private Border etchedBorder;

            public Ex9(String titre) {

            setTitle(titre);
            setSize(350,300);

            // Récupération de la couche contenu
            Container contentPane = getContentPane();

            JPanel panel = new JPanel(new BorderLayout());

            // Ajout dans le contentPane
            contentPane.add(panel);

            // Création du texte
            l1 = new JButton ( "NORTH1");
            l1bis = new JButton ( "NORTH2");
            l2 = new JButton ( "SOUTH");
```

```
            l3 = new JButton ( "EAST");
            l4 = new JButton ( "WEST");
            l5 = new JButton ( "CENTER");

            // Création d'un JPanel pour la région NORTH
            JPanel northPanel = new JPanel();

            // Création de la bordure pour le northPanel
            etchedBorder = BorderFactory.createEtchedBorder();
            northPanel.setBorder(etchedBorder);

            // Ajout du northPanel au panneau principal
            panel.add(northPanel, BorderLayout.NORTH);

            // Ajout des 2 boutons au Panneau nord
            northPanel.add(l1);
            northPanel.add(l1bis);

            // Ajout des autres boutons
            panel.add(l2, BorderLayout.SOUTH);
            panel.add(l3, BorderLayout.EAST);
            panel.add(l4, BorderLayout.WEST);
            panel.add(l5, BorderLayout.CENTER);

            setVisible(true);

            // Ajout du gestionnaire d'évènement auprès de la fenêtre
            addWindowListener(new WindowCloser());
            }

            public static void main ( String [] args ) {

            // instanciation directement au sein de la classe
            Ex9 myFrame = new Ex9("BorderLayout");
            }

            // Classe interne
            public class WindowCloser extends WindowAdapter {
                 public void windowClosing(WindowEvent we) {
                           System.exit(0);
                 }
            }
}
```

Ex9.java

- **Ligne 40 et 41** : Initialisation et affectation de la bordure au northPanel
- **Ligne 44** : Le panneau est affecté à la région NORTH du panneau principal
- **Ligne 47 et 48** : Les boutons *l1* et *l2* sont ajoutés à ce nouveau panneau

c. Conclusion

Pour des mises en forme complexe, il ne faut pas hésiter à imbriquer, à l'image de l'exemple précédent, les conteneurs de type *JPanel* en jouant sur les gestionnaires de placement.

B.3. Gestionnaire GridLayout

Ce gestionnaire organise les composants en lignes et colonnes à l'instar d'un tableau. Néanmoins, les cellules font toutes la même taille et si vous souhaitez affecter des tailles différentes, mieux utiliser un gestionnaire de type *BoxLayout*.

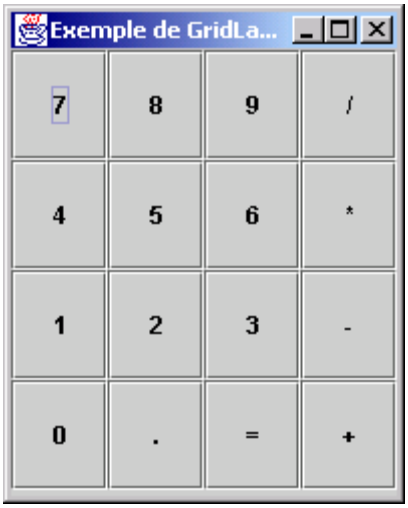
a. Principe

Le constructeur de la classe *GridLayout* offre la possibilité de fixer à l'avance le nombre de lignes et de colonnes :

Constructeurs de GridLayout
GridLayout() Crée un gestionnaire GridLayout avec une colonne par composant sur une seule ligne
GridLayout(int rows, int cols) Crée un gestionnaire GridLayout avec *cols* colonnes et *rows* lignes
GridLayout(int rows, int cols, int hgap, int vgap) Crée un gestionnaire GridLayout avec *cols* colonnes et *rows* lignes, un espacement vertical de v*gap* et un espacement horizontal de *hgap*.

b. Exemple de calculatrice

Le programme suivant illustre l'intérêt de ce gestionnaire pour l'implémentation graphique d'une calculatrice :




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


public class Ex10 extends JFrame {

            // Champ de type JButton
            private JButton[] btab = new JButton[16];
            private JPanel centerPanel;

            public Ex10(String titre) {
```

```
            setTitle(titre);
            setSize(200,250);

            // Récupération de la couche contenu
            Container contentPane = getContentPane();

            // Création d'un JPanel pour la région CENTER
            centerPanel = new JPanel(new GridLayout(4,4) );

            // Ajout du centerPanel au panneau principal
            contentPane.add(centerPanel, BorderLayout.CENTER);

            createButtons();
            setVisible(true);

            // Ajout du gestionnaire d'évènement auprès de la fenêtre
            addWindowListener(new WindowCloser());
            }


            /* Méthode permettant de créer les
               boutons automatiquement à partir
               d'une chaîne de caractères */
            public void createButtons() {

            // Création des boutons
            String cchaine = "789/456*123-0.=+";

            for (int i=0;i<cchaine.length();i++)
                 {
                 btab[i] = new JButton( cchaine.substring(i,i+1) );
                 centerPanel.add(btab[i]);
                 }
            }

            public static void main ( String [] args ) {

            // instanciation directement au sein de la classe
            Ex10 myFrame = new Ex10("Exemple de GridLayout");
            }

            // Classe interne
            public class WindowCloser extends WindowAdapter {
                 public void windowClosing(WindowEvent we) {
                           System.exit(0);
                 }
            }
}
```

Ex10.java

- **Ligne 21** : Création du *JPanel* et du gestionnaire *GridLayout* de 4 colonnes sur 3 lignes.
- **Lignes 37 à 47** : Il s'agit d'une boucle qui parcoure chaque caractère de la chaîne et crée le bouton correspondant en l'ajoutant au tableau de boutons et au panneau central.

B.4. Box et BoxLayout

a. Gestionnaire BoxLayout

Ce gestionnaire permet de placer les composants sur une seule ligne ou colonne. En général, ce gestionnaire s'utilise avec le conteneur *Box* mais il peut également être utilisé avec un *JPanel*. D'ailleurs, le gestionnaire par défaut de l'objet *Box* est un *BoxLayout*. L'intérêt d'utiliser l'objet *Box* est qu'il contient un certain nombre de méthodes statiques très utiles pour la gestion des *BoxLayout*.

Les objets de type *Box* étant des conteneurs, ils s'utilisent de la façon suivante :

```
1  Box b = Box.createHorizontalBox();
```

L'objet *b* va créer un conteneur disposant d'un *LayoutManager* de type *BoxLayout*. Vous pouvez maintenant y ajouter, grâce à la méthode *add* des composants qui se rangeront de gauche à droite.

b. Exemple

Voici un programme utilisant un objet de type *Box* :

Le code de ce programme est le suivant :

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


public class Ex11 extends JFrame {

            // Champ de type JButton
            private JButton l1, l2 , l3;

            public Ex11(String titre) {

            setTitle(titre);

            // Récupération de la couche contenu
            Container contentPane = getContentPane();

            Box panelBox = Box.createVerticalBox();

            // Ajout dans le contentPane
            contentPane.add(panelBox);

            // Création du texte
            l1 = new JButton ( "Bouton 1");
            l2 = new JButton ( "Bouton 2");
            l3 = new JButton ( "Bouton 3");
```

```
            panelBox.add(l1);
            panelBox.add(l2);
            panelBox.add(l3);

            // Permet de redimensionner les composants de
            // la fenêtre à leur taille préférée
            pack();

            setVisible(true);

            // Ajout du gestionnaire d'évènement auprès de la fenêtre
            addWindowListener(new WindowCloser());
            }

            static void main ( String [] args ) {

            // instanciation directement au sein de la classe
            Ex11 myFrame = new Ex11("BoxLayout");
            }

            // Classe interne
            public class WindowCloser extends WindowAdapter {
                 public void windowClosing(WindowEvent we) {
                           System.exit(0);
                 }
            }
}
```

Ex11.java

c. Réserves

Vous avez sans doute remarqué que le gestionnaire *BoxLayout* ne gère pas les espaces entre les composants. Il faut donc ajouter des *fillers* (Réserves) entre chaque composant. Il existe trois types différents de *fillers* :

- **Strut :** Ajoute un peu d'espace entre les composants. Cet espace est une valeur en pixels. Il existe les *Struts* horizontaux et verticaux. S'il s'agit d'un conteneur vertical, ajoutez un *Strut* vertical.
- **RigidArea** : Identique à une paire de *Struts* à la différence qu'il sépare les composants adjacents, mais ajoute aussi une hauteur ou une largeur dans l'autre direction.
- **Glue :** Ajoute autant d'espace que possible entre les composants. Autant que le permet le conteneur.

Voici un échantillon des effets de ces réserves sur le programme précédent :

Strut vertical

```
1  panelBox.add(l1);
panelBox.add(Box.createVerticalStrut(15));
panelBox.add(l2);
```

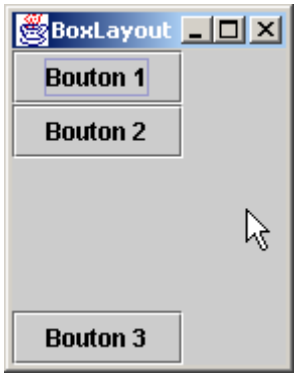
```
panelBox.add(l3);
```

Utilisation de la méthode statique createVerticalStrut

RigidArea

```
1  panelBox.add(l1);
panelBox.add(l2);
panelBox.add(Box.createRigidArea(new Dimension(5,20) ));
panelBox.add(l3);
```

La méthode *createRigidArea* prend un objet de type *Dimension* . Un objet invisible d'une largeur de 5 pixels et de hauteur de 20 pixels



Glue

```
1  panelBox.add(l1);
2  panelBox.add(l2);
3  panelBox.add(Box.createGlue( ));
4  panelBox.add(l3);
```

La méthode *createGlue* insère un espace de hauteur variable. Cet espace est aussi grand que le conteneur le permet. Si vous agrandissez la fenêtre, l'espace sera plus grand. Cette réserve est très utile pour des mises en place de composants complexes.

B.5. Gestionnaire GridBagLayout

Il s'agit d'une amélioration du gestionnaire *GridLayout*. En effet, là où ce dernier ne permettait pas d'obtenir des tailles variables pour les lignes et le colonnes, *GridBagLayout* peut fusionner des cellules comme sait le faire un traitement de texte ou un tableur.
Ce gestionnaire peut s'avérer très complexe à manipuler mais offre de nombreuses possibilités de mise en forme. De plus, il y a de grandes chances pour que vos programmes soient destinés à fonctionner sur diverses plate-formes et il faudra prendre grand soin de réfléchir à une mise en page soignée qui pourra s'adapter à différentes tailles et types de polices de caractères.

a. Contraintes de placement (GridBagConstraints)

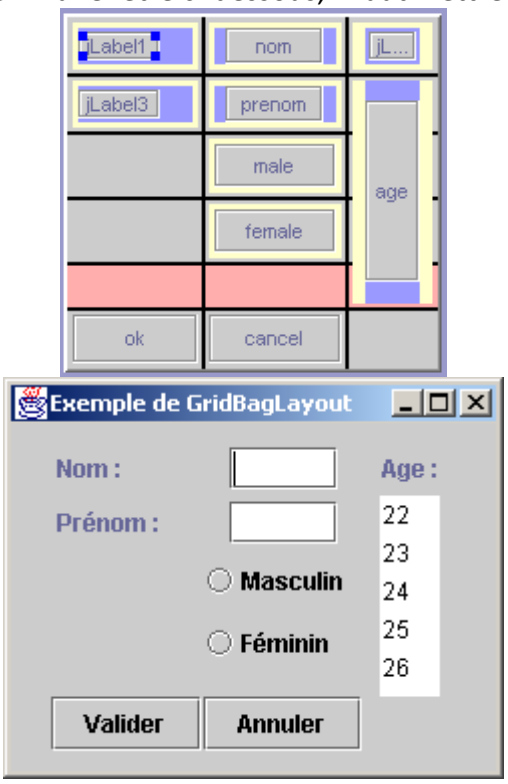
Chaque composant géré par ce gestionnaire peut être placé automatiquement sur la grille avec des contraintes que vous fixez à l'avance. Pour cela, on utilise l'objet *GridBagContraints*.
Cette objet permet d'agir sur la manière dont le composant va être placé sur la grille. Parmi les champs de cet objet, on y trouve :

Field Summary	
int	**anchor** Ce champ est utilise lorsque le composant est plus petit que l'espace don't il dispose. Ce composant peut alors être placé aux 4 points cardinaux (EAST, WEST, NORTH, SOUTH, NORTHEAST, NORTHWEST, SOUTHEAST, SOUTHWEST). Voir la doc complète
int	**gridheight** Spécifie le nombre de cellules en hauteur que va nécessiter le composant pour s'afficher
int	**gridwidth** Spécifie le nombre de cellules en largeur que va nécessiter le composant pour s'afficher
int	**gridx** Numéro de la rangée où doit se situer le composant en partant de la gauche. Le composant le plus à gauche a un *gridx* égal à 0
int	**gridy** Numéro de la colonne où doit se situer le composant en partant du haut. Le composant le plus en haut a un *gridy* égal à 0
Insets	**insets** Ce champ représente la marge interne de la cellule où se situe le composant. Il s'agit de l'espace séparant les extremités du composant avec celles de la zone d'affichage.
int	**ipadx** Ce champ représente la quantité d'espace en largeur à ajouter à la taille minimale du composant.
int	**ipady** Ce champ représente la quantité d'espace en hauteur à ajouter à la taille minimale du composant.
double	**weightx** Ce champ détermine le poids horizontal du composant. Le poids représente la manière dont le composant va être redimensionné lorsque le conteneur change de taille. Si aucun agrandissement n'est souhaité, laissez la valeur à 0.
double	**weighty** Idem pour le poids en hauteur

Liste non exhaustive des champs de la classe GridBagConstraints

b. Exemple

Avant d'utiliser ce type de gestionnaire, il faut réfléchir à la mise en forme des composants dans le conteneur. Pour obtenir la fenêtre ci-dessous, il faut mettre en place la grille qui suit :



Résultat de l'utilisation du GridBagLayout

La première étape pour obtenir ce résultat est de mettre en place une fenêtre avec un *Jpanel* et un gestionnaire *GridBagLayout* :

```
1  JPanel centerPanel = new javax.swing.JPanel();
centerPanel.setLayout(new java.awt.GridBagLayout());
```

Ensuite, nous allons créer un objet de type *GridBagConstraints* que nous utiliserons pour tous les composants :

```
java.awt.GridBagConstraints gridBagConstraints1;
```

Chaque composant va être crée puis ajouté au *Jpanel* avec l'objet *GridBagConstraints* :

```
jLabel1.setText("Nom :");
jLabel1.setMaximumSize(new java.awt.Dimension(45, 16));

gridBagConstraints1 = new java.awt.GridBagConstraints();
gridBagConstraints1.gridx = 0; // Rangée 0
gridBagConstraints1.gridy = 0; // Colonne 0
gridBagConstraints1.insets = new java.awt.Insets(3, 3, 3, 3);
gridBagConstraints1.anchor = java.awt.GridBagConstraints.WEST;
```

```
centerPanel.add(jLabel1, gridBagConstraints1); // Ajout
```

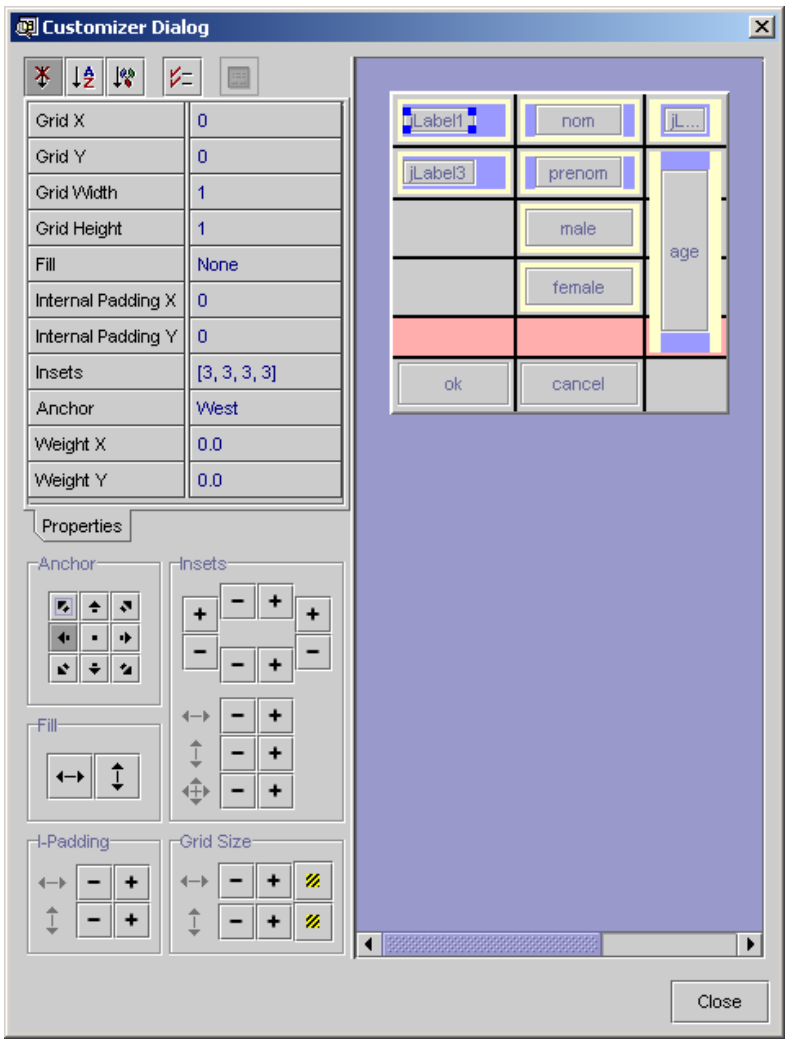
La méthode va être identique pour tous les composants. Une exception cependant pour la liste des ages, puisque ce composant va s'étendre sur 4 rangées (*gridheight = 4)* :

```
private javax.swing.JList age = new javax.swing.JList();

String []  maListe = { "22" , "23", "24","25","26","27"};
age.setListData(maListe);

gridBagConstraints1 = new java.awt.GridBagConstraints();
gridBagConstraints1.gridx = 2;
gridBagConstraints1.gridy = 1;
gridBagConstraints1.gridheight = 4;
gridBagConstraints1.insets = new java.awt.Insets(0, 7, 0, 7);
centerPanel.add(age, gridBagConstraints1);
```

Le listing complet est disponible dans le fichier *Ex13.java*

c. Conclusion

Les environnements de développement Java proposent des interfaces graphiques très poussées permettant de tirer partie de la puissance de ce gestionnaire. Ces interfaces vous permettent, grâce à une représentation graphique simple de paramétrer le gestionnaire. Voici un exemple avec l'environnement *Forte* de *Sun :*

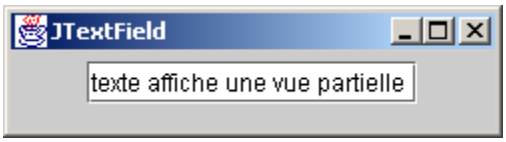

Customizer Dialog
Grid X 0
Grid Y 0
Grid Width 1
Grid Height 1
Fill None
Internal Padding X 0
Internal Padding Y 0
Insets [3, 3, 3, 3]
Anchor West
Weight X 0.0
Weight Y 0.0
Properties
Anchor
Insets
Fill
I-Padding
Grid Size
jLabel1
nom
jL...
jLabel3
prenom
male
age
female
ok
cancel
Close

Chapitre 3 : Composants d'interface graphique

A. Architecture Modèle Vue Contrôleur

Java étant un langage orienté objet, il va s'attacher à ne pas donner à chaque objet trop de responsabilité en partant du principe que chaque objet peut compter sur les autres pour faire le travail qui leur a été confié. C'est pourquoi, les concepteurs de ce langage ont utiliser un type d'architecture basé sur le modèle, la vue et le contrôleur :

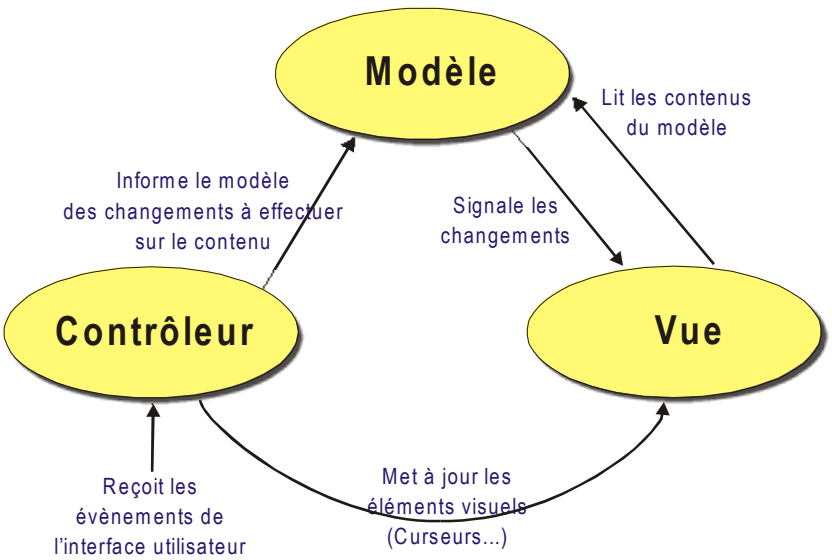
- **Le modèle** : Chargé de maintenir le contenu. Pour un champ de texte, le modèle est un objet *String* qui contient la valeur du champ de texte
- **La vue** : Chargé d'afficher le contenu. Pour ce même champ de texte, la vue s'apparente à un espace rectangulaire où va s'afficher la chaîne
- **Le contrôleur** : Chargé de gérer l'action de l'utilisateur. Pour le champ de texte, le contrôleur va gérer l'action du l'utilisateur sur les touches du clavier



Vue d'un champ de texte

Le modèle de ce champ est un objet chaîne qui contient :
"Le champ de texte affiche une vue partielle"
En résumé, le modèle détient tout le contenu, alors que la vue en donne une représentation visuelle partielle ou complète, pendant que le contrôleur gère l'action de l'utilisateur et les modifications sur le modèle ou la vue.

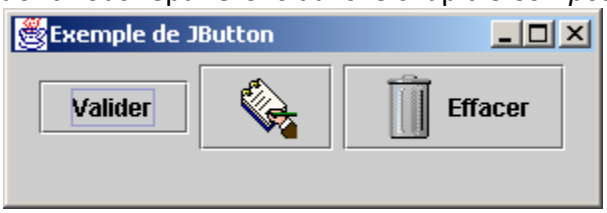
> Le mot modèle est sûrement mal choisi puisqu'on a l'habitude d'employer ce terme pour désigner un concept abstrait. Ici, le modèle est plutôt associé au contenu.



Architecture Modèle Vue Contrôleur

B. Boutons

Les exemples utilisés dans les chapitres précédents décrivaient les méthodes pour utiliser les boutons. Il existe aussi des variétés de boutons telles que les boutons radio, les cases à cocher dont nous reparlerons dans le chapitre *Composants de choix* . Voici des exemples de boutons :



Bouton texte, image , image + texte

B.1. JButton

Ce composant s'utilise très simplement avec ses constructeurs :

Constructeurs de JButton
JButton()   Crée un nouveau bouton sans texte ni icône
JButton(Action a)   Crée un bouton dont les propriétés sont issues de l'objet *Action* fourni en paramètre. Les objets *Action* ou *ActionListener* sont utiles pour la gestion des évènements
JButton(Icon icon)   Crée un bouton avec l'icône spécifié
JButton(String text)   Crée un bouton avec le texte spécifié.
JButton(String text, Icon icon)   Crée un bouton avec le texte et l'icône spécifié

Voici un exemple de création de bouton :

```
1  JButton monBouton = new JButton("Valider");
```

Après la création, le bouton peut être ajouté à un conteneur :

```
conteneur.add(monBouton);
```

B.2. JButton avec une image

Le constructeur attend un objet de type *Icon* tel que *ImageIcon*. L'exemple qui suit montre comment créée un tel bouton :

```
Jbutton bouton = new JButton ( new ImageIcon("writit1.gif"));
```

B.3. Jbutton avec une image et du texte

Même principe avec un paramètre en plus dans le constructeur :

```
1  Jbutton bouton
2  bouton = new JButton ( "Effacer" , new ImageIcon("trash1.gif"));
```

B.4. Jbutton et les évènements

L'intérêt d'un bouton est de pouvoir réaliser une action lorsque l'on clique dessus. Ceci est possible grâce aux évènements qui feront l'objet du chapitre *Gestion des évènements*. Nous allons tout de même aborder le sujet.
De manière générale, les évènements en Java sont gérés par le principe de *l'écouteur* et de l'objet *écouté*. Chaque objet capable de provoquer un événement, doit être recensé auprès de l'objet *écouteur* pour qu'une action soit entreprise lors de l'appui sur le bouton.
En général, *l'écouteur* est un conteneur qui va gérer les évènements de tous les composants qu'il contient.
L'exemple suivant montre comment un conteneur de type *JFrame* peut gérer le bouton qui s'y trouve :

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


public class Ex15 extends JFrame implements ActionListener {

            // Champ de type JButton
            private JButton bouton;
            private JPanel panel;

            public Ex15(String titre) {

            // Récupération de la couche contenu
            Container contentPane = getContentPane();

            // Création du bouton
            bouton = new JButton("Fermer");
            setSize(150,100);
            setTitle(titre);
            contentPane.add(bouton);

            setVisible(true);

            // Ajout du gestionnaire d'évènement auprès de la fenêtre
            addWindowListener(new WindowCloser());

            // Recensement du bouton auprès de l'objet écouteur
            // this représente l'objet issu de cette classe (myFrame)
            bouton.addActionListener(this);

            }
```

>> Suite à la page suivante

```
            // Méthode appelée lorsqu'un évènement provoqué
            // par le bouton se produit

            public void actionPerformed( ActionEvent evt) {
                 // Si la source de l'évènement est le bouton
                 // le programme se termine
                 if (evt.getSource() == bouton)
                      System.exit(0);
                 }

            // Programme principal
            static void main ( String [] args ) {

            // instanciation directement au sein de la classe
            Ex15 myFrame = new Ex15("JButton");
            }

            // Classe interne
            public class WindowCloser extends WindowAdapter {
                 public void windowClosing(WindowEvent we) {
                           System.exit(0);
                 }
            }
}
```

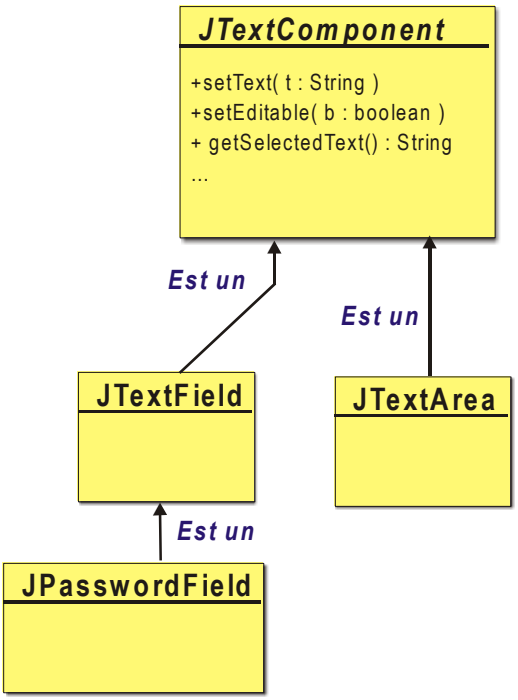
Ex15.java

- **Ligne 8 :** *Ex15* implémente l'interface *ActionListener*. Donc, en plus d'être un objet de type fenêtre, *Ex15* devient un *écouteur d'actions*.
- **Ligne 32 :** L'objet écouté (Le bouton) informe l'objet écouteur (La fenêtre) qui doit exécuter la méthode *actionPerformed* si le bouton venait à être pressé
- **Ligne 42 :** La méthode *getSource* d'un objet *ActionEvent* renvoie le composant qui a provoqué l'événement. Donc s'il s'agit bien du bouton, l'application sera fermée.

➲ Si tout cela vous paraît confus, ne vous alarmez pas. Lorsque vous lirez le chapitre sur les évènements, il sera toujours temps de revenir sur ce chapitre pour mieux comprendre comment gérer les évènements sur un bouton.

C. Entrées de texte

Il existe plusieurs composants possibles pour permettre à l'utilisateur d'afficher ou de saisir du texte. Tous héritent de la classe abstraite *JTextComponent*. Cette dernière étant abstraite, vous ne pourrez l'instancier. Cependant, ses méthodes telles que *setText* ou *setEditable* seront bien utiles pour manipuler les composants :



Modèle objet des composants de texte

C.1. JTextField

Associé à un composant *JLabel* , *JTextField* permet à l'utilisateur de saisir du texte. Voici un exemple d'utilisation :

```
1  JTextField champ = new JTextField("Saisissez votre texte",40);
```

L'exemple précédent construit un champ de texte de 40 colonnes. L'utilisateur peut saisir plus de 40 caractères, mais l'affichage sera limité à cette valeur. Pour changer le texte affiché, la méthode *setText* de la superclasse va s'avérer utile.

> D'une manière générale, lorsque vous ne trouvez pas les méthodes nécessaires au fonctionnement d'un objet dans la liste de ses méthodes, il est très probable que les méthodes recherchées se trouvent dans les classes parents. La documentation des API vous aident à chercher puisque toutes les méthodes des parents sont données sur la même page (Methods inherited from class ObjetParent)

C.2. JTextArea

a. Principe

JTextArea permet la saisie de texte sur plusieurs lignes. Il est plus complexe et offre plus de possibilités que *JTextField*. Il est très utilisé lorsqu'il s'agit de saisir une quantité importante de texte. Il offre de nombreuses fonctionnalités comme le retour à la ligne automatique.
Pour créer un objet de type *JTextArea* , de 40 colonnes et 8 lignes :

```
1  JTextArea zone = new JTextArea(8,40);
```

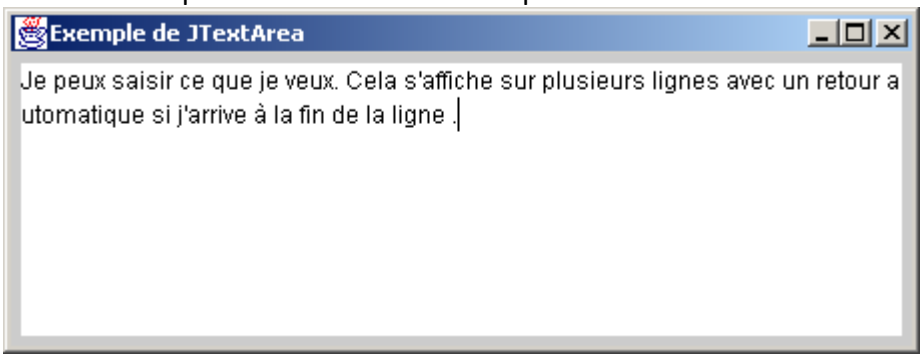
> Le fait de spécifier un nombre de colonnes ou de lignes n'engage à rien puisque le gestionnaire de mise en forme est parfois amené à redimensionner les composants du conteneur qu'il gère. Vous pouvez toujours spécifier ces valeurs mais l'affichage risque d'être différent.

Pour autoriser le retour à la ligne automatique, on appelle la méthode *setLineWrap* :

```
zone.setLineWrap(true);
```

b. Exemple

Voici un exemple d'utilisation de ce composant :



Exemple d'utilisation de JTextArea

```
1  import javax.swing.*;
2  import java.awt.event.*;
3  import java.awt.*;
4
5
6  public class Ex17 extends JFrame {
7
8           public Ex17(String titre) {
9
10          setTitle(titre);
11
12          JPanel panel = new JPanel();
13          // Construction de l'objet
14          JTextArea champ = new JTextArea("Saisissez votre texte",
   8,40);
15
16          champ.setLineWrap(true);    // Retour à la ligne auto
17
18          Container contentPane = getContentPane();
19          contentPane.add(panel);
20          panel.add(champ);
```

```
21          setVisible(true);
22          // Ajuste la taille du conteneur à celle des composants
23          pack();
24
25          // Ajout du gestionnaire d'évènement auprès de la fenêtre
26          addWindowListener(new WindowCloser());
27          }
28
29          static void main ( String [] args ) {
30
31          // instanciation directement au sein de la classe
32          Ex17 myFrame = new Ex17("Exemple de JTextArea");
33          }
34
35          // Classe interne
36          public class WindowCloser extends WindowAdapter {
37               public void windowClosing(WindowEvent we) {
38                         System.exit(0);
39               }
40          }
41  }
```

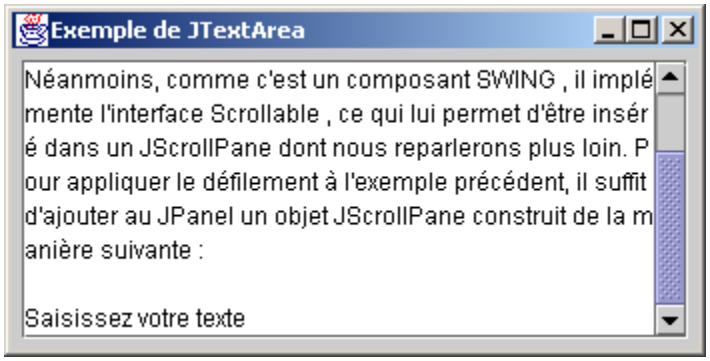
Fichier Ex17.java

c. Barres de défilement

JTextArea, contrairement à *TextArea* ne gère pas le défilement. Néanmoins, comme c'est un composant *SWING* , il implémente l'interface *Scrollable* , ce qui lui permet d'être inséré dans un *JScrollPane* dont nous reparlerons plus loin. Pour appliquer le défilement à l'exemple précédent, il suffit d'ajouter au *JPanel* un objet *JScrollPane* construit de la manière suivante :

```
1  JScrollPane scrollpane = new JScrollPane(champ);
2  panel.add(scrollpane);
```

Le défilement du champ *JTextArea* est assuré par l'élément *JScrollPane* et ce dernier est ajouté au *JPanel*.



Même exemple avec *JScrollPane*

C.3. JPasswordField

Pour des raisons de sécurité, un mot de passe ne doit pas être visible lorsqu'on le tape. Pour une sécurité accrue, ce composant ne stocke pas son contenu dans une chaîne mais dans un tableau de caractères.

Pour récupérer le mot de passe, vous pouvez utiliser la méthode *getPassword* :

```
1  JPasswordField pass = new JPasswordField(10);
2  char [] tableau = pass.getPassword();
```

Le constructeur spécifie que le mot de passe s'affichera sur 10 caractères.

D. Composants de choix

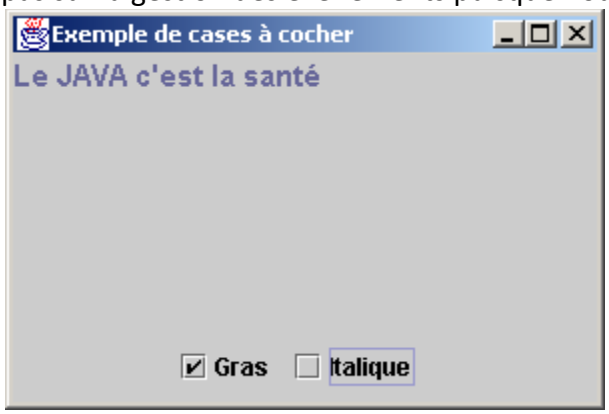
Les composants tels que les boutons radio, cases à cocher , liste modifiables ou non modifiables permettent à l'utilisateur de préciser ses choix dans une application.

D.1. Cases à cocher

La classe *JCheckBox* est un composant réactif représentant une case à cocher. L'état de la case est soit cochée ou non cochée.
Le constructeur suivant permet de créer une case à cocher avec le texte "Gras" associé en spécifiant l'état de départ comme cochée :

```
1  JCheckBox chkbox = new JCheckBox( "Gras" , true );
```

Le programme suivant permet d'illustrer les possibilités des cases à cocher. Ne vous attardez pas sur la gestion des événements puisque nous en reparlerons en détail dans ce document :



Exemple de programme avec case à cocher

> Si vous ne comprenez pas l'intégralité du programme, ne vous affolez pas. Focalisez votre attention sur les cases à cocher et leur possibilité de provoquer l'exécution de la méthode ***actionPerformed*** lorsque l'une d'elle est activée ou désactivée. Cette méthode réinitialise la fonte avant de traiter le gras ou l'italique. Sachez que tout ce qu'il est important de comprendre est décrit après le programme.

```
1  import javax.swing.*;
2  import java.awt.event.*;
3  import java.awt.*;
4
5
6  public class Ex18 extends JFrame implements ActionListener {
7
8           JPanel panel , southPanel;
9           JCheckBox gras,italique;
10          Container contentPane;
11          JLabel label;
12
13
14          public Ex18(String titre) {
15
16          setTitle(titre);
17
18          // Création des JPanel
19          panel = new JPanel(new BorderLayout());
20          southPanel = new JPanel();
21
22          //Création du texte avec la police de caractère
23          label = new JLabel("Le JAVA c'est la santé");
24          label.setFont(new Font("SansSerif", Font.BOLD , 15));
25
26          // Initialisation des cases à cocher
27          gras  = new JCheckBox("Gras",true);
28          italique = new JCheckBox("Italique");
29
30          // La fenêtre doit écouter les 2 cases
31          gras.addActionListener(this);
32          italique.addActionListener(this);
33
34          // Récupération et remplissage du contentPane
35          contentPane = getContentPane();
36          contentPane.add(panel);
37
38          // Ajout des composants dans le panneau principal
39          panel.add(label, BorderLayout.NORTH);
40          panel.add(southPanel, BorderLayout.SOUTH);
41
42          // Ajout des composants dans le panneau du bas
43          southPanel.add(gras);
44          southPanel.add(italique);
45
46          setSize(300,200);
47          setVisible(true);
48
49          // Ajout du gestionnaire d'évènement auprès de la fenêtre
50          addWindowListener(new WindowCloser());
51
52          }
53
54          // Méthode exécutée lorsqu'un événement recensé se produit
55          public void actionPerformed( ActionEvent evt) {
56
57          Font ft = label.getFont();  // Récupération de la fonte
58          ft = ft.deriveFont(Font.PLAIN); // Remet en PLAIN
59
```

```
60          int style = 0;
61
62               // Evénement provient-il de gras ou italique ?
63               if (evt.getSource() == gras | evt.getSource()== italique)
64                    {
65                    if (gras.isSelected()) // gras sélectionnée ?
66                         style = style | Font.BOLD;
67                    if (italique.isSelected())// italique sélectionnée?
68                         style = style | Font.ITALIC;
69
70                    ft = ft.deriveFont(style); // Application du style
71                    label.setFont(ft);    // Application sur le texte
72                    }
73          }
74
75
76          static void main ( String [] args ) {
77
78          // instanciation directement au sein de la classe
79          Ex18 myFrame = new Ex18("Exemple de cases à cocher");
80          }
81
82          // Classe interne
83          public class WindowCloser extends WindowAdapter {
84               public void windowClosing(WindowEvent we) {
85                         System.exit(0);
86               }
87          }
88 }
```

Ex18.java

- **Lignes 27 et 28** : Création des cases à cocher. La case gras état cochée dès le départ alors que la case italique est non cochée.
- **Lignes 31 et 32** : La fenêtre va gérer les évènements qui vont se produire sur les 2 cases à cocher. L'objet *this* correspond à l'objet issu de la classe elle-même.
- **Ligne 65** : La méthode *isSelected* de l'objet *gras* permet de connaître l'état de la case à cocher.

D.2. Boutons radio

La classe *JRadioButton* permet à l'utilisateur de faire un choix unique parmi plusieurs. Elle est utilisée conjointement avec la classe *ButtonGroup*. La méthode *add* de cette dernière permet d'ajouter des boutons radio à un groupe. Tous les boutons radio appartenant au même groupe fonctionnent de manière exclusive : lorsque l'utilisateur clique sur un des boutons, tous les autres sont décochés.
La première étape consiste à créer un groupe de bouton :

```
1  ButtonGroup bg = new ButtonGroup();
```

Ensuite, vous associez les boutons radio avec le groupe en utilisant la méthode *add* :

```
bg.add(gras);
bg.add(italique);
```

Et Java s'occupe de tout !!!

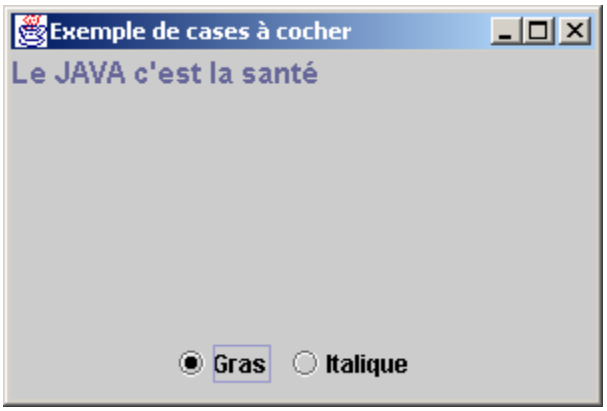
➲ Le fichier ***Ex19.java*** contient les modifications nécessaires pour transformer le programme précédent (***Ex18.java***) en boutons radio.

Voici le nouveau code source :

```
2  import javax.swing.*;
3  import java.awt.event.*;
4  import java.awt.*;
5
6
7  public class Ex19 extends JFrame implements ActionListener {
8
9           JPanel panel , southPanel;
10          JRadioButton gras,italique;
11          ButtonGroup bg;
12          Container contentPane;
13          JLabel label;
14
15
16          public Ex19(String titre) {
17
18          setTitle(titre);
19
20          // Création des JPanel
21          panel = new JPanel(new BorderLayout());
22          southPanel = new JPanel();
23
24          //Création du texte avec la police de caractère
25          label = new JLabel("Le JAVA c'est la santé");
26          label.setFont(new Font("SansSerif", Font.BOLD , 15));
27
28          // Initialisation des cases à cocher
29          gras  = new JCheckBox("Gras",true);
30          italique = new JCheckBox("Italique");
31
32          // Initialisation du groupe de boutons
33          bg = new ButtonGroup();
34          bg.add(gras);
35          bg.add(italique);
36
37          // La fenêtre doit écouter les 2 cases
38          gras.addActionListener(this);
39          italique.addActionListener(this);
40
41          // Récupération et remplissage du contentPane
42          contentPane = getContentPane();
43          contentPane.add(panel);
44
45          // Ajout des composants dans le panneau principal
46          panel.add(label, BorderLayout.NORTH);
47          panel.add(southPanel, BorderLayout.SOUTH);
48
49          // Ajout des composants dans le panneau du bas
50          southPanel.add(gras);
51          southPanel.add(italique);
52
53          setSize(300,200);
54          setVisible(true);
55
56          // Ajout du gestionnaire d'évènement auprès de la fenêtre
57          addWindowListener(new WindowCloser());
58
```

```
59          }
60
61          // Méthode exécutée lorsqu'un événement recensé se produit
62          public void actionPerformed( ActionEvent evt) {
63
64          Font ft = label.getFont();  // Récupération de la fonte
65          ft = ft.deriveFont(Font.PLAIN); // Remet en PLAIN
66
67          int style = 0;
68
69               // Evénement provient-il de gras ou italique ?
70               if (evt.getSource() == gras | evt.getSource()== italique)
71                    {
72                    if (gras.isSelected()) // gras sélectionnée ?
73                         style = style | Font.BOLD;
74                    if (italique.isSelected())// italique sélectionnée?
75                         style = style | Font.ITALIC;
76
77                    ft = ft.deriveFont(style); // Application du style
78                    label.setFont(ft);    // Application sur le texte
79                    }
80          }
81
82
83          static void main ( String [] args ) {
84
85          // instanciation directement au sein de la classe
86          Ex19 myFrame = new Ex19("Exemple de cases à cocher");
87          }
88
89          // Classe interne
90          public class WindowCloser extends WindowAdapter {
91               public void windowClosing(WindowEvent we) {
92                         System.exit(0);
93               }
94          }
95 }
```

Ex19.java



Résultat du programme Ex19.java

➲ Les boutons radio d'un même groupe sont souvent entourés par une bordure qui permet à l'utilisateur de comprendre que les boutons sont associés. Il existe plusieurs sortes de bordures que nous verrons à la fin de ce chapitre. En général les bordures sont associées à un panneau (***JPanel*** par exemple)

D.3. Listes

La classe *JList* propose d'agrémenter vos applications de listes qui prennent moins d'espace que les cases à cocher ou les boutons radio, surtout lorsque le nombre d'options est élevé.
Tout comme les cases à cocher, il est possible de faire plusieurs choix dans la liste. L'utilisateur maintient la touche *Ctrl* en cliquant sur les choix suivants. La liste réagit et sélectionne plusieurs entrées en même temps.

a. Constructeurs

Selon l'utilisation que l'on souhaite faire de la liste, il existe plusieurs constructeurs à votre disposition :

Constructeurs de JList
JList() Crée un JList avec un modèle vide.
JList(ListModel dataModel) Crée une JList affichant les éléments du modèle spécifié.
JList(Object[] listData) Crée une JList affichant une représentation texte des objets présents dans le tableau d'objets donné en paramètres.
JList(Vector listData) Même chose avec un objet Vector

Constructeurs de JList

Voici un exemple d'utilisation du constructeur :

```
1  String [ ] wordList = {"gras", "italique", "gras italique" };
2  JList list = new JList( worlist );
```

b. Barres de défilement

Par défaut, Java n'intègre pas de barres de défilement. Il faut donc utiliser la classe *JScrollPane* et y insérer votre liste. Vous ajouterez ensuite ce panneau, au conteneur de votre application :

```
JScrollPane sp = new JScrollPane(list);
panel.add(sp);
```

Les barres défilement apparaîtront si nécessaire.

c. Sélection multiple

La méthode *setSelectionMode* permet de définir le mode de sélection. Voici toutes les valeurs possibles que l'on peut obtenir par les champs statiques suivants de l'interface *ListSelectionModel* :

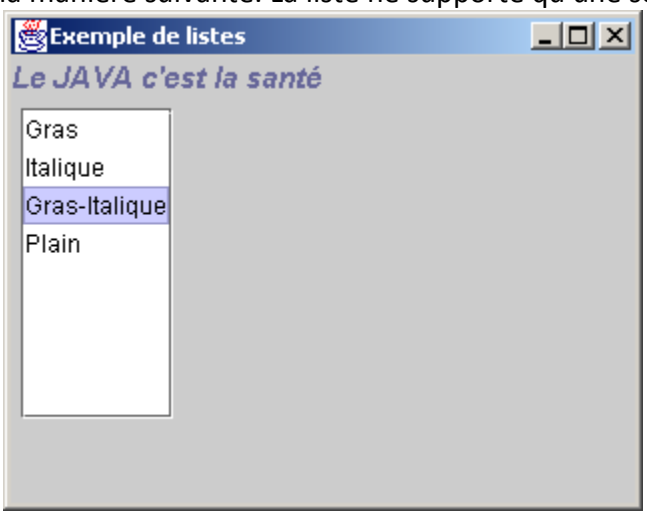
- **SINGLE_SELECTION** : Une seule entrée peut être sélectionnée.
- **SINGLE_INTERVAL_SELECTION** : Un seul intervalle contigu d'entrées peut être sélectionné.
- **MULTIPLE_INTERVAL_SELECTION** : Dans ce mode, il n'y a pas de restriction particulière. C'est le mode par défaut.

Pour autoriser la sélection d'une seule entrée à la fois :

```
list.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
```

d. Gérer les événements

Le principal intérêt d'une liste est de récupérer le ou les éléments sélectionnés par l'utilisateur. Dans le cas d'une sélection multiple, l'opération est plus compliquée.
En tout cas, la liste sera référencée auprès d'un "*écouteur de liste*" , c'est à dire, une classe qui implémente l'interface *ListSelectionListener* qui se doit d'implémenter la méthode *valueChanged* qui sera appelée si un changement est effectué dans la sélection.
Nous reprenons l'exemple des polices de caractère mais cette fois-ci, les choix apparaissent de la manière suivante. La liste ne supporte qu'une seule sélection à la fois:



Exemple d'utilisation des listes

Voici le code source de ce cette application :

```
1  import javax.swing.*;
2  import javax.swing.event.*;
3  import java.awt.event.*;
4  import java.awt.*;
5
6  public class Ex20 extends JFrame implements ListSelectionListener {
7
8           JPanel panel , westPanel;
9           JScrollPane sp;
10          Container contentPane;
11          JLabel label;
12          String[] wordList = { "Gras", "Italique",
13                                "Gras-Italique", "Plain" };
14          JList myList;
15
16          public Ex20(String titre) {
17
18          setTitle(titre);
19
20          // Création des JPanel
21          panel = new JPanel(new BorderLayout());
22          westPanel = new JPanel();
23
24          //Création du texte avec la police de caractère
25          label = new JLabel("Le JAVA c'est la santé");
26          label.setFont(new Font("SansSerif", Font.BOLD , 15));
27          // Initialisation de la liste et des barres de défilement
28          myList = new JList(wordList);
29          sp = new JScrollPane(myList);
30
31          // Selection d'une seule entrée à la fois
```

```
32          myList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
33
34          // La fenêtre doit écouter la liste
35          myList.addListSelectionListener(this);
36
37          // Récupération et remplissage du contentPane
38          contentPane = getContentPane();
39          contentPane.add(panel);
40
41          // Ajout des composants dans le panneau principal
42          panel.add(label, BorderLayout.NORTH);
43          panel.add(westPanel, BorderLayout.WEST);
44
45          // Ajout de la liste au panneau
46          westPanel.add(sp);
47
48          setSize(320,250);
49          setVisible(true);
50          // Ajout du gestionnaire d'évènement auprès de la fenêtre
51          addWindowListener(new WindowCloser());
52          }
53
54          // Méthode exécutée lorsqu'un événement recensé se produit
55          public void valueChanged( ListSelectionEvent evt) {
56
57          Font ft = label.getFont();  // Récupération de la fonte
58
59          int style = 0;
60          int index = myList.getSelectedIndex();
61
62          // Test de tous les cas possibles
63          switch (index) {
64                    case 0 : style = Font.BOLD;
65                              break;
66                    case 1 : style = Font.ITALIC;
67                              break;
68                    case 2 : style = Font.ITALIC | Font.BOLD;
69                              break;
70                    case 3 : style = Font.PLAIN;
71          }
72          ft = ft.deriveFont(style); // Application du style
73          label.setFont(ft);    // Application sur le texte
74          }
75
76          static void main ( String [] args ) {
77
78          // instanciation directement au sein de la classe
79          Ex20 myFrame = new Ex20("Exemple de listes");
80          }
81          // Classe interne
82          public class WindowCloser extends WindowAdapter {
83               public void windowClosing(WindowEvent we) {
84                         System.exit(0);
85               }
86          }
87 }
```

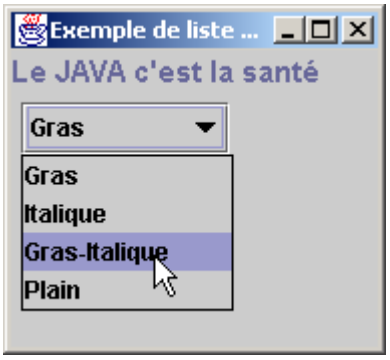
Ex20.java

e. Conclusion

Les listes sont des éléments complexes lorsque vous souhaitez obtenir des effets moins standards. En effet, avec un modèle de listes, vous pouvez gérer de longues listes de manière dynamique, ce qui n'est pas le cas des exemples que nous avons vu.
N'hésitez pas à consulter les tutoriels de Sun ainsi que la documentation des classes *ListModel , DefaultListModel, ListCellRenderer* et *JList*.

D.4. Listes combinées (Combo box)

Il s'agit de listes déroulantes économes en espace. Lorsque l'utilisateur clique sur le champ, une liste de choix s'affiche :



Exemple de liste combinée

a. Méthodes

Voici les principales méthodes utiles pour manipuler ce genre de composant :

Méthodes de JComboBox (Non exhaustive)	
void	**addItem**(Object anObject) Ajoute un element à la liste
Object	**getItemAt**(int index) Renvoie l'élément situé à la position donnée par index.
int	**getItemCount**() Renvoie le nombre d'éléments de la liste
int	**getMaximumRowCount**() Renvoie le nombre maximum d'éléments affichables sans la nécessité d'une barre de défilement
int	**getSelectedIndex**() Renvoie un entire correspondant à l'élément actuellement sélectionné
Object	**getSelectedItem**() Renvoie l'élément actuellement sélectionné
void	**insertItemAt**(Object anObject, int index) Insère l'élément fourni, à la place indiquée dans index
boolean	**isEditable**() Renvoie *true* si l'élément est modifiable
boolean	**isPopupVisible**()

	Renvoie *true* si la liste est déroulée
void	**removeItem**(Object anObject) Enlève l'élément donné en argument
void	**setEditable**(boolean aFlag) Détermine si la liste est modifiable ou pas.
void	**setEnabled**(boolean b) Active la liste
void	**setMaximumRowCount**(int count) Fixe le nombre de lignes que la liste peut afficher.
void	**setSelectedIndex**(int anIndex) Sélectionne l'élément correspondant à l'index indiqué
void	**setSelectedItem**(Object anObject) Séléctionne l'élément passé en argument.

Principales méthodes de JComboBox

b. Constructeurs

JComboBox dispose des mêmes constructeurs que *JList*. Consultez la doc API Java ou le paragraphe précédent (3.a)

c. Gestion des évènements

L'objet qui sera chargé (La fenêtre en général) d'être à l'écoute des listes combinées doit implémenter l'interface *ActionListener* :

```
1   public class NomApplication extends JFrame implements ActionListener
    {…}
```

La liste aura en charge de se référencer auprès de l'objet *"écouteur"* :

```
myList.addActionListener(this);
```

Lorsqu'un changement de selection interviendra sur la liste, la méthode *actionPerformed* sera appelée.
Le code source complet de l'exemple ci-dessus se trouve dans le fichier *Ex21.java*

E. Menus

La plupart des applications disposent de menus pour accéder aux fonctions. La définition des menus est simple en Java. Vous pouvez imbriquer les menus pour créer des sous-menus, insérer des éléments de menus, ajouter des icônes à l'intérieur des menus.

E.1. JMenuBar

Le point de départ de la construction d'un menu est la classe *JMenuBar*. Ce composant peut être ajouté n'importe où. La plupart du temps, la barre de menu est associée à la fenêtre qui contient le menu :

```
1  JMenuBar menuBar = new JMenuBar();
2
3  frame.setJMenuBar(menuBar); // Association du menu avec la fenêtre
```

E.2. JMenu

Pour créer un menu dans notre barre de menus, vous devrez utiliser la classe *JMenu*. Vous aurez autant d'objets *JMenu* que de menus :

```
1  JMenu fileMenu = new JMenu("Fichier");
2  JMenu editMenu = new JMenu("Edition");
3  JMenu helpMenu = new JMenu("Aide");
```

Méthodes de JMenu	
Component	**add**(Component c) Ajoute un composant à la fin du menu.
Component	**add**(Component c, int index) Ajoute un composant à la place indiquée
void	**addSeparator**() Ajoute un élément séparateur à la fin du menu

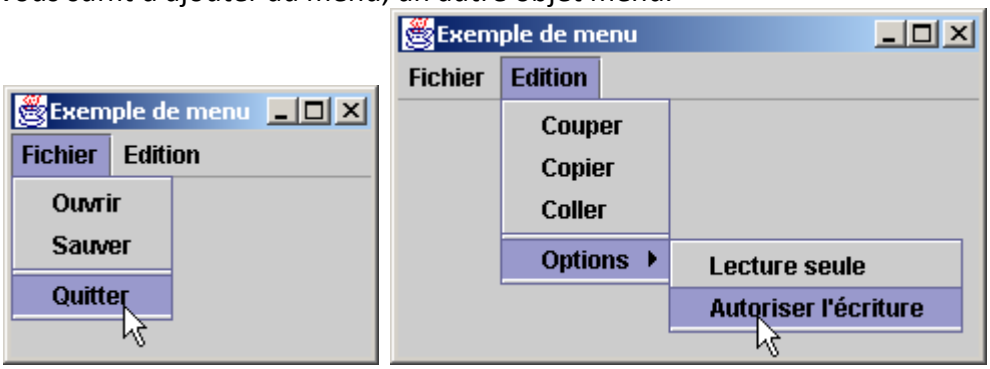
E.3. JMenuItem

Evidemment, chaque menu pourra contenir plusieurs éléments de menus. Pour cela, vous utiliserez la classe *JMenuItem* en ajoutant chaque élément de menu au menu correspondant. Par exemple, pour ajouter un élément *Ouvrir* à notre menu *Fichier* :

```
1  JMenuItem openItem = new JMenuItem("Ouvrir…");
2  fileMenu.add(openItem);
```

E.4. Imbriquer les menus

Le menu *Edition* contient un autre menu présentant des options. Pour arriver à un tel résultat, il vous suffit d'ajouter au menu, un autre objet menu.



Voici le code source :

```
1  import javax.swing.*;
2  import javax.swing.event.*;
3  import java.awt.event.*;
4  import java.awt.*;
5
6  public class Ex22 extends JFrame {
7
8            JMenuBar menuBar;
9            JMenu fileMenu,editMenu,helpMenu;
10           JMenu subEdit;
```

```
11
12          JMenuItem openItem, saveItem, quitItem;
13          JMenuItem cutItem, copyItem, pasteItem;
14          JMenuItem readOnlyItem,writeItem;
15          JMenuItem aboutItem;
16
17          public Ex22(String titre) {
18
19          // Création du menu Fichier
20          fileMenu = new JMenu("Fichier");
21
22          openItem = new JMenuItem("Ouvrir");
23          saveItem = new JMenuItem("Sauver");
24          quitItem = new JMenuItem("Quitter");
25          fileMenu.add(openItem);
26          fileMenu.add(saveItem);
27          fileMenu.addSeparator();
28          fileMenu.add(quitItem);
29
30          // Création du menu Edition
31          editMenu = new JMenu("Edition");
32          cutItem = new JMenuItem("Couper");
33          copyItem = new JMenuItem("Copier");
34          pasteItem = new JMenuItem("Coller");
35          editMenu.add(cutItem);
36          editMenu.add(copyItem);
37          editMenu.add(pasteItem);
38          editMenu.addSeparator();
39
40          // Création du sous-menu de Edition
41          subEdit = new JMenu("Options");
42          readOnlyItem = new JMenuItem("Lecture seule");
43          writeItem = new JMenuItem("Autoriser l'écriture");
44          subEdit.add(readOnlyItem);
45          subEdit.add(writeItem);
46
47          editMenu.add(subEdit);
48
49          // Création et initialisation de la barre de menus
50          menuBar = new JMenuBar();
51          menuBar.add(fileMenu);
52          menuBar.add(editMenu);
53          setJMenuBar(menuBar);
54
55          setTitle(titre);
56
57          setSize(320,250);
58          setVisible(true);
59
60          // Ajout du gestionnaire d'évènement auprès de la fenêtre
61          addWindowListener(new WindowCloser());
62          }
63
64          static void main ( String [] args ) {
65
66          // instanciation directement au sein de la classe
67          Ex22 myFrame = new Ex22("Exemple de menu");
68          }
69          // Classe interne
```

```
70          public class WindowCloser extends WindowAdapter {
71               public void windowClosing(WindowEvent we) {
72                         System.exit(0);
73               }
74          }
75 }
```

Ex22.java

> Vous constatez que le programme précédent n'est guère adapté à une application ayant beaucoup de menus. Heureusement, les environnements de développement proposent des interfaces graphiques qui vous assistent dans la préparation de menus.

E.5. Evénements

Pour qu'un menu réagisse au clic de souris, il incombe à la fenêtre d'écouter l'élément de menu :

```
1  JMenuItem cutItem = new JMenuItem("Couper");
cutItem.addActionListener(this); // this étant la fenêtre
```

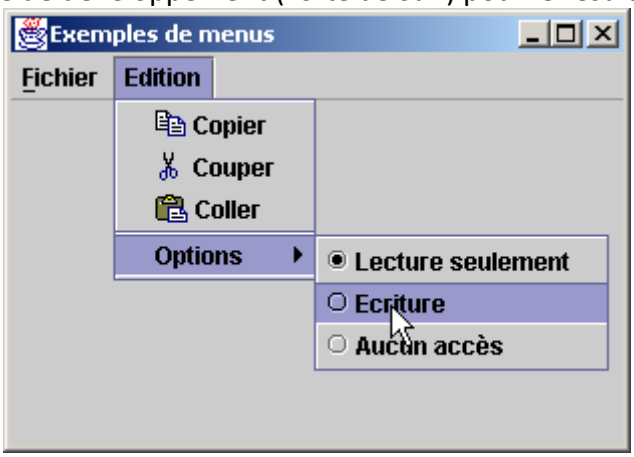
Lorsque le menu sera sélectionné, la méthode *actionPerformed* sera appelée :

```
public void actionPerformed( Event evt ) {
            if (evt.getSource() == cutItem)
                 // .. Action à réaliser
}
```

> L'objet écouteur, ici, la fenêtre, doit implémenter l'interface ***ActionListener*** , ce qu'il l'oblige à implémenter la méthode ***actionPerformed*.**

E.6. Exemple complet

Voici un programme permettant de gérer des menus. La majeure partie de cet exemple a été généré par une interface de développement (*Forte de Sun*) pour le résultat suivant :



Programme Ex23.java

Le code source du programme est dans le fichier *Ex23.java*.

➲ Ce genre d'interface de développement prend très souvent des libertés par rapport au code source qu'il génère. De plus, le code n'est pas toujours optimisé et clair. Il vous faudra cependant comprendre le code généré pour pouvoir le modifier à bon escient. Dans l'exemple du fichier ***Ex23.java*** , vous observerez que l'interface préfère générer une méthode par événement plutôt que de gérer l'ensemble des évènements comme nous le faisions dans les exemples précédents.

F. Boîtes de dialogue

F.1. Classes

La gestion des boîtes de dialogue est intégrée à Java qui propose une multitude de fenêtres différentes correspondant à un usage spécifique :

- ***JDialog*** : Utilisée pour la création de boîtes de dialogue personnalisées
- ***JFileChooser*** : Utilisée pour sélectionner des fichiers ou des répertoires
- ***JColorChooser*** : Même chose pour les couleurs
- ***JOptionPane*** : Utilisée au travers de ses méthodes statiques pour créer des boîtes de dialogue à usage instantané pour informer ou interagir avec l'utilisateur

F.2. Modale ou non modale

Toutes ces boîtes de dialogue sont des fenêtres. Une fenêtre *modale* ne permet à l'utilisateur d'utiliser d'autres fenêtres tant que celle-ci n'a pas été fermée. A l'inverse, une fenêtre non modale peut être mise en arrière-plan tandis que l'utilisateur travaille avec une autre. C'est le cas de la plupart des fenêtres principales des application type *word , excel, …*

F.3. Objet parent

Dans presque tous les cas, les boîtes de dialogue ont un objet parent. Cet objet correspond en général à la fenêtre qui a provoqué son ouverture.

F.4. JOptionPane

Les méthodes **statiques** suivantes seront très utiles pour créer des boîtes de dialogue simples :

- **showMessageDialog** : Affiche un message et attend que l'utilisateur clique sur OK
- **showConfirmDialog** : Affiche une message et attend que l'utilisateur donne une réponse parmi deux choix , OK ou Cancel
- **showOptionDialog** : Affiche un message et récupère l'option choisie par l'utilisateur
- **showInputDialog** : Affiche un message et récupère un ligne d'entrée saisie par l'utilisateur

Chaque boîte contient les éléments suivants :

- Un icône
- Un message

- Un ou plusieurs boutons d'option

a. showMessageDialog

Voici les différentes signatures de cette méthode :

Méthodes showMessageDialog	
static void	**showMessageDialog**(Component parentComponent, Object message)
static void	**showMessageDialog**(Component parentComponent, Object message, String title, int messageType)
static void	**showMessageDialog**(Component parentComponent, Object message, String title, int messageType, Icon icon)

Le code suivant affiche la boîte qui suit :


```
1  import javax.swing.JOptionPane;
2
3  public class Ex25  {
4
5      public static void main (String args[]) {
6
7           int type = JOptionPane.WARNING_MESSAGE;
8           JOptionPane.showMessageDialog(null,"Exemple de
9                                boîte","showMessageDialog",type);
10
11     }
12 }
```

Ex25.java

b. showConfirmDialog

Voici les différentes signatures de cette méthode :

Méthode showConfirmDialog	
static int	**showConfirmDialog**(Component parentComponent, Object message) Affiche une boîte modale avec les options Yes, No et Cancel et le titre "Select An Option"
static int	**showConfirmDialog**(Component parentComponent, Object message, String title, int optionType) Affiche une boîte de dialogue modale dont le nombre de choix dépend de l'entier *optionType*.
static int	**showConfirmDialog**(Component parentComponent,

	Object message, String title, int optionType, int messageType) Même boîte que la précédente avec en plus le paramètre *messageType* qui détermine le type d'icône à afficher
static int	**showConfirmDialog**(Component parentComponent, Object message, String title, int optionType, int messageType, Icon icon) Même boîte que la précédente avec en plus l'icône à afficher. La paramètre *messageType* n'est là que pour fournir un icône par défaut pour le Look and Feel

Le code suivant affiche la boîte qui suit :


```
1  import javax.swing.JOptionPane;
2
3  public class Ex26 {
4
5      public static void main (String args[]) {
6
7          int optionType = JOptionPane.YES_NO_CANCEL_OPTION;
8          int messageType = JOptionPane.INFORMATION_MESSAGE;
9          String titre = "showConfirmDialog";
10         String message = "Exemple de boîte de confirmation";
11         JOptionPane.showConfirmDialog(null,message,titre,optionType,
12                                                         messageType);
13     }
14 }
```

Ex26.java

c. showOptionDialog

Method Summary	
static int	**showOptionDialog**(Component parentComponent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object initialValue)

Le code suivant affiche la boîte qui suit :


```
1  import javax.swing.JOptionPane;
2  import javax.swing.JRadioButton;
3  import javax.swing.ButtonGroup;
4
5  public class Ex27 {
6
7      public static void main (String args[]) {
8
9                 int optionType = JOptionPane.YES_NO_OPTION;
10                int messageType = JOptionPane.QUESTION_MESSAGE;
11                String titre = "showOptionDialog";
12                String [] options = { "Sympa ?" , "Pas sympa ?"};
13
14                String message = "Exemple de boîte de confirmation";
15
           JOptionPane.showOptionDialog(null,message,titre,optionType,
16                              messageType,null,options, options[0]);
17         System.exit(0);
18     }
19 }
```

Ex27.java

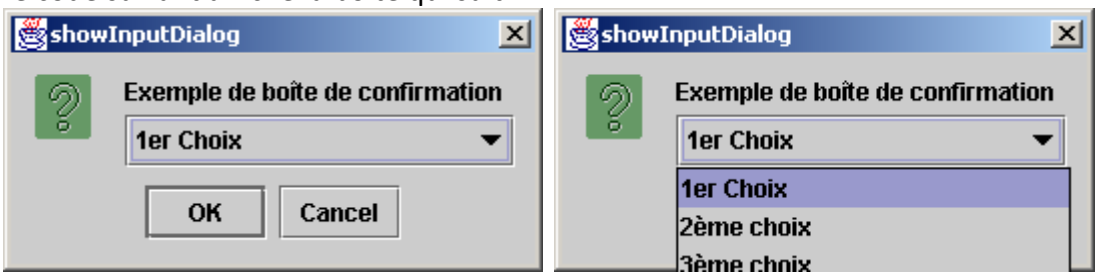
➲ La méthode ***showOptionDialog*** renvoie un entier correspondant au choix de l'utilisateur. Par contre , cette méthode renvoie ***JOptionPane.CLOSED_OPTION*** si

d. showInputDialog

Signatures de *showInputDialog*	
`static String`	**showInputDialog**(Component parentComponent, Object message)
`static String`	**showInputDialog**(Component parentComponent, Object message, String title, int messageType)
`static Object`	**showInputDialog**(Component parentComponent, Object message, String title, int messageType, Icon icon, Object[] selectionValues, Object initialSelectionValue) Affiche une boîte de dialgue où l'utilisateur peut effectuer un certain nombre de choix caractérisés par le paramètre *selectionValues* qui représente un tableau d'objets. Renvoie

	l'objet séléectionné ou null si l'utilisateur à annulé
`static String`	**`showInputDialog`**`(Object message)`

Le code suivant affiche la boîte qui suit :




```
1  import javax.swing.JOptionPane;
2  import javax.swing.JRadioButton;
3  import javax.swing.ButtonGroup;
4
5  public class Ex28 {
6
7      public static void main (String args[]) {
8
9                 int optionType = JOptionPane.YES_NO_OPTION;
10                int messageType = JOptionPane.QUESTION_MESSAGE;
11                String titre = "showInputDialog";
12                String [] options = { "1er Choix" , "2ème choix",
13                                                    "3ème choix"};
14
15                String message = "Exemple de boîte de confirmation";
16
17                JOptionPane.showInputDialog(null,message,titre,
18                                messageType,null,options, options[0]);
19           System.exit(0);
20     }
21 }
```

Ex28.java

F.5. JFileChooser

Conçue tout spécialement pour proposer à l'utilisateur une boîte de dialogue de sélection de fichiers ou de répertoires, la classe *JFileChooser* est assez simple à utiliser et propose toute sorte d'options pour sélectionner plusieurs fichiers ou encore pour filtrer les fichiers affichés.

a. Constructeurs

Constructeurs intéressants de *JFileChooser*
`JFileChooser``()` Construit un objet JFileChooser pointant sur le répertoire utilisateur.
`JFileChooser``(File currentDirectory)` Construit un objet *JFileChooser* pointant sur le répertoire désigné par l'objet *File*
`JFileChooser``(String currentDirectoryPath)` Construit un objet *JFileChooser* pointant sur le répertoire désigné par la chaîne

Liste partielle des constructeurs de JFileChooser

Voilà un exemple de création :

```
1  JFileChooser fc = new JFileChooser("C:\\jdk1.3");
```

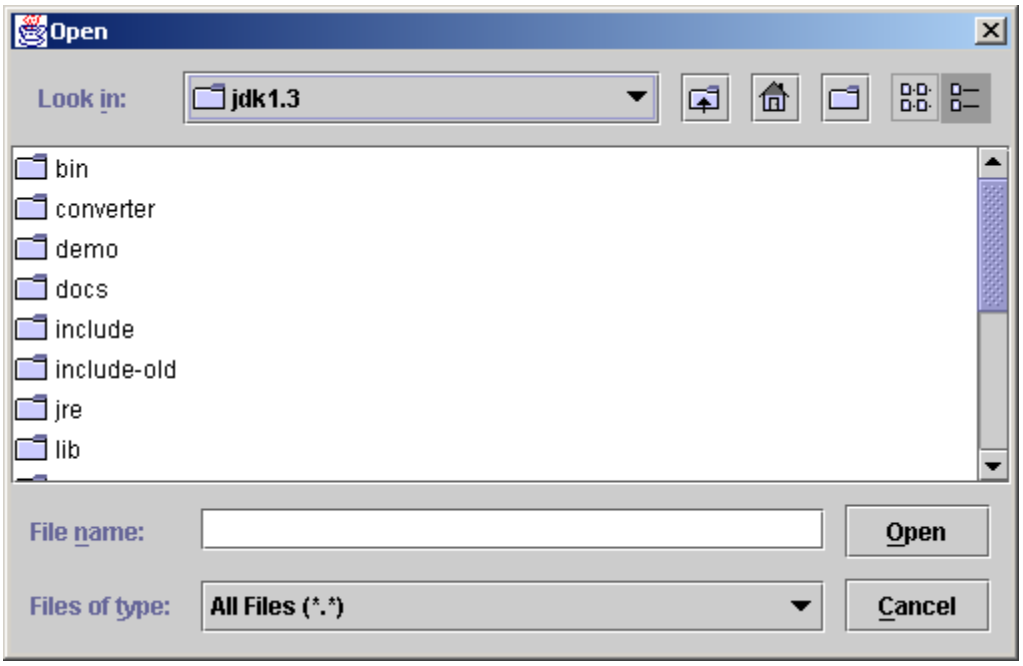
Ou en construisant un objet *File* :

```
1  File f = new File("C:\\jdk1.3");
2  JFileChooser fc = new JFileChooser(f);
```

Ensuite l'appel à la méthode showO*penDialog* affiche l'objet à partir de l'objet parent qui peut-être une fenêtre ou *null* :

```
fc.showOpenDialog(parent);
```

Le résultat ressemblera à ceci sous Windows :



Résultats du fichier Ex29.java

b. Filtrer les fichiers

A l'aide de la classe *FileFilter* , vous avez la possibilité de filtrer les fichiers affichés ou encore de fournir à l'utilisateur une liste de filtres disponibles. Il vous faut donc créer une classe qui hérite de cette classe et qui masque les deux méthodes *accept* et *getDescription*.

Méthodes de FileFilter	
abstract boolean	**accept**(File f) Détermine si le fichier fourni est accepté par ce filtre
abstract String	**getDescription**() Renvoie la description du filtre

Voici la classe *JPGFilter* permettant de filtrer les fichiers images de type *JPeg* :

```
1  import javax.swing.filechooser.FileFilter;
2
3  public class JPGFileFilter extends FileFilter {
```

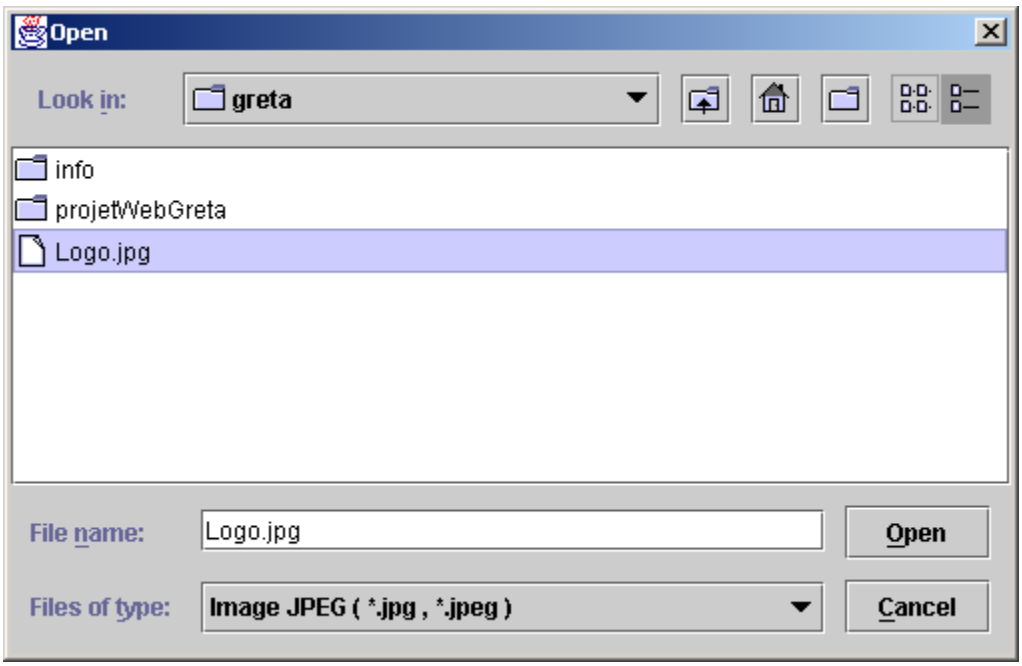
```
4
5     public boolean accept( File f ) {
6           String ext;
7           ext = f.getName().toLowerCase();
8           if ((ext.endsWith(".jpg")||ext.endsWith(".jpeg"))
9                                                       ||f.isDirectory())
10                return true;
11          else
12                return false;
13    }
14
15    public String getDescription() {
16          return "Image JPEG ( *.jpg , *.jpeg )";
17               }
18   }
```

Voici un exemple d'utilisation des filtres :

```
1  import javax.swing.JFileChooser;
2  import java.io.File;
3
4  public class Ex30 {
5
6      public static void main (String args[]) {
7
8           Ex30 myEx30 = new Ex30();
9           System.exit(0);
10     }
11
12          public Ex30 () {
13
14               String fs = System.getProperty("file.separator");
15
16               File f = new File("C:\\jdk1.3");
17               JFileChooser fc = new JFileChooser("C:\\jdk1.3");
18               fc.setFileFilter(new JPGFileFilter());
19               fc.showOpenDialog(null);
20          }
```

Ex30.java

Voilà la fenêtre résultante :



c. Récupérer le(s) fichier(s) sélectionné(s)

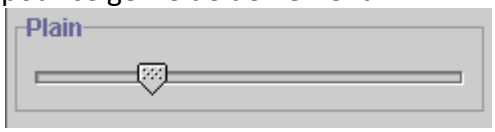
La méthode *getSelectedFile* permet de récupérer un objet *File* si l'utilisateur a sélectionné un seul fichier. La méthode *getSelectedFiles* renvoie un tableau de *File* si l'utilisateur a effectué une sélection multiple.

> Pour récupérer une sélection multiple, il faut avoir activé la sélection multiple grâce à la méthode ***setMultiSelectionEnabled(boolean b)***

G. Barres de défilement

G.1. Préambule

Les barres de défilement sont utiles pour faire défiler le contenu d'une fenêtre ou de n'importe quel conteneur. Elles peuvent être également utilisées comme un contrôle permettant d'atteindre une valeur sur une échelle prédéfinie mais la classe *JSlider* semble plus appropriée pour ce genre de défilement.

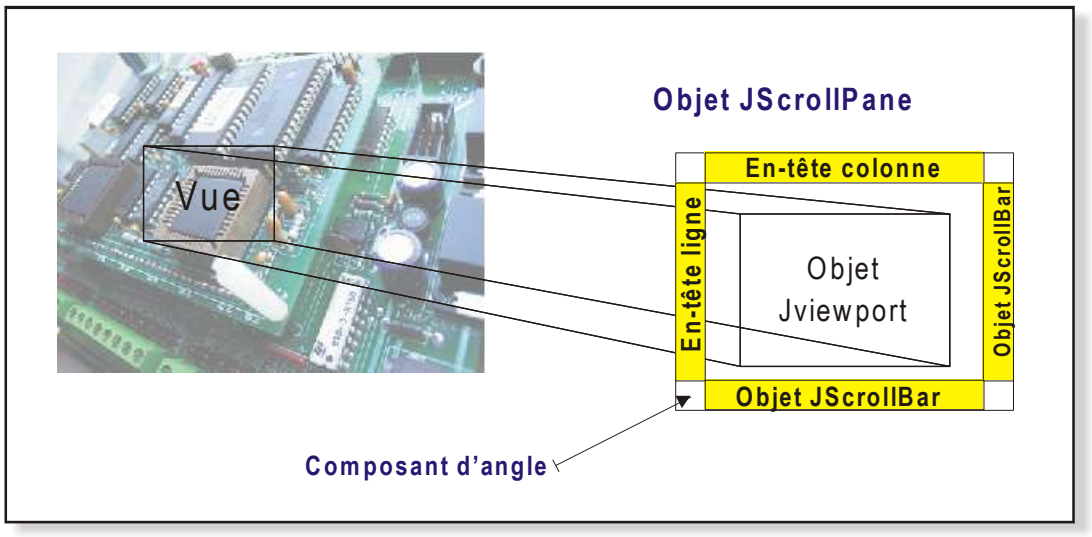


Utilisation de JSlider

Nous étudierons ici les barres de défilement appliquées aux fenêtres ou aux panneaux.

G.2. Panneaux de défilement

Le modèle objet utilisé pour représenter des barres de défilement est le suivant :



A partir du moment où le contenu est plus grand que la vue, les barres de défilement entrent en action. Un objet *JScrollPane* est alors associé au composant par le constructeur suivant :

```
1  JPanel jp= new JPanel();
2  JScrollPane sp = new JScrollPane(jp);
```

Dans cet exemple , le *JPanel* disposera de barres de défilement. En fait, lors de l'appel au constructeur de *JScrollPane* , un objet *JViewPort* est crée autour du composant à visualiser. Ensuite, le panneau défilant sera ajouté à la fenêtre courante :

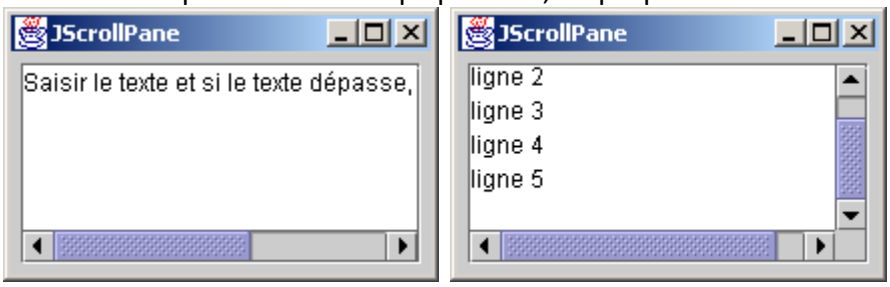
```
contentPane.add(sp);
```

La dernière étape consiste à fixer une taille préférée pour le panneau défilant :

```
sp.setPreferredSize(new Dimension(300,200));
```

Si le composant est plus grand que cette taille, les barres de défilement apparaîtront si besoin.

Voici un exemple illustrant ce qui précède, en proposant une fenêtre comme celle-ci :



Voici le code source de cet exemple :

```
1  import javax.swing.*;
2  import java.awt.event.*;
3  import java.awt.*;
4
5  public class Ex31 extends JFrame {
6
7  JScrollPane sp;
8  JTextArea champ;
9  JPanel panel;
10
11      public Ex31(String titre) {
12
13          setTitle(titre);
14
15          panel = new JPanel();
16          champ = new JTextArea("Saisir le texte");
17          sp = new JScrollPane(champ);
```

```
18          // Fixe la taille du panneau défilant
19          sp.setPreferredSize(new Dimension(200,100));
20          panel.add(sp);
21
22          Container contentPane = getContentPane();
23
24          contentPane.add(panel);
25          // Ajuste la fenêtre à la taille préférée des composants
26          pack();
27
28          setVisible(true);
29
30          // Ajout du gestionnaire d'évènement auprès de la fenêtre
31          addWindowListener(new WindowCloser());
32          }
33
34          // Programme principal
35          static void main ( String [] args ) {
36
37          // instanciation directement au sein de la classe
38          Ex31 myFrame = new Ex31("JScrollPane");
39          }
40
41          // Classe interne gérant la fermeture
42          public class WindowCloser extends WindowAdapter {
43               public void windowClosing(WindowEvent we) {
44                         System.exit(0);
45               }
46     }
47 }
```

Ex31.java

Pour modifier la façon dont les barres de défilement fonctionnent, il est fort utile d'utiliser les constantes disponibles au sein de *JOptionPane* . Ces champs ne sont pas documentés dans la classe *JOptionPane* puisqu'ils appartiennent à l'interface *ScrollPaneConstants* que cette dernière implémente :

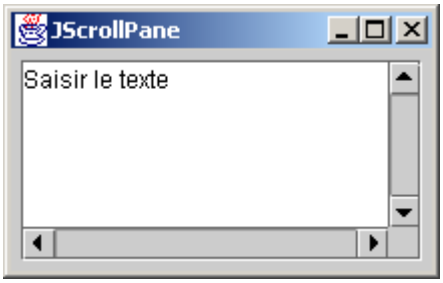
Champs de *ScrollPaneConstants*	
static String	**HORIZONTAL_SCROLLBAR** Identifies a horizonal scrollbar.
static int	**HORIZONTAL_SCROLLBAR_ALWAYS** Used to set the horizontal scroll bar policy so that horizontal scrollbars are always displayed.
static int	**HORIZONTAL_SCROLLBAR_AS_NEEDED** Used to set the horizontal scroll bar policy so that horizontal scrollbars are displayed only when needed.
static int	**HORIZONTAL_SCROLLBAR_NEVER** Used to set the horizontal scroll bar policy so that horizontal scrollbars are never displayed.
static int	**VERTICAL_SCROLLBAR_ALWAYS** Used to set the vertical scroll bar policy so that vertical scrollbars are always displayed.

static int	**VERTICAL_SCROLLBAR_AS_NEEDED** Used to set the vertical scroll bar policy so that vertical scrollbars are displayed only when needed.
static int	**VERTICAL_SCROLLBAR_NEVER** Used to set the vertical scroll bar policy so that vertical scrollbars are never displayed.

Pour profiter de ces champs, vous pouvez utiliser les méthodes de l'exemple qui suit :

```
1  sp.setHorizontalScrollBarPolicy(
2                JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
3  sp.setVerticalScrollBarPolicy (
4                JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
```

Cet exemple fait apparaître les barres verticale et horizontale tout le temps, même quand cela n'est pas nécessaire :



➲ La classe ***JScrollPane*** implémentant l'interface ***ScrollPaneConstants*** , elle hérite donc de tous ses champs statiques. C'est la raison pour laquelle, vous pouvez appeler les champs statiques directement à partir de la classe ***JScrollPane***.

Chapitre 4 : Gestion des évènements

A. Introduction

Les évènements sont une partie incontournable d'un système d'exploitation. Ils sont le cœur de l'intervention de l'utilisateur et des périphériques au sein du système. L'appui sur un bouton, le clic de souris ou la fermeture d'une fenêtre sont des évènements typiques.
Java permet de gérer de manière efficace ces évènements au sein d'applications graphiques. Le modèle des évènements permet à chaque composant *SWING* de prendre en charge les évènements qui se produisent.

B. Modèle objet Java pour les évènements

B.1. Introduction

En Java, la gestion des évènements se fait par les classes contenues dans le package *java.awt.event* , ce qui signifie qu'*AWT* prend à sa charge cette gestion. Les composants *SWING* , qui ne sont que des composants *AWT* améliorés bénéficient donc de ce savoir-faire.

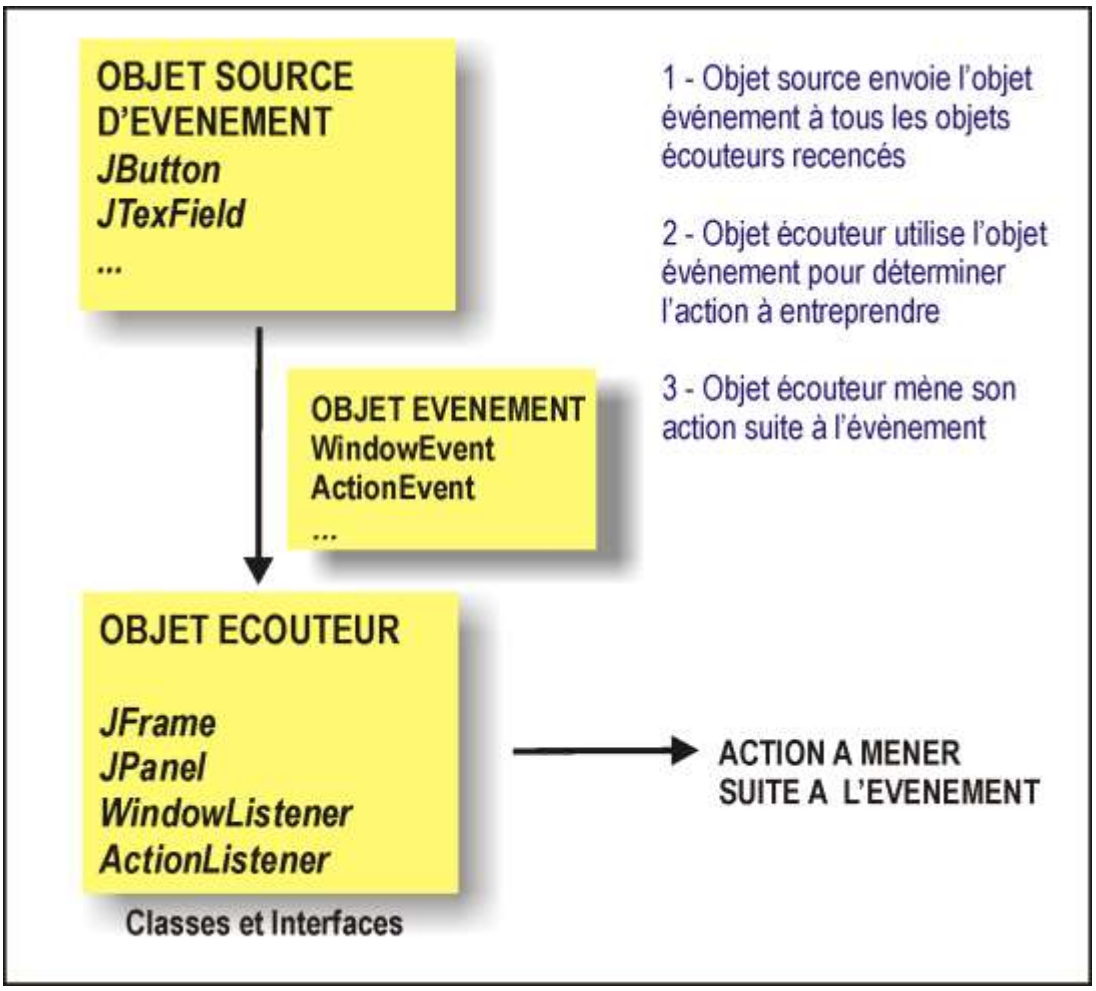
B.2. Mécanisme des évènements

Les éléments constituant la gestion des évènements sont les suivants :

- **Un objet écouteur** : Un objet capable d'écouter les évènements et qui par conséquent implémente une interface spéciale appelée *interface écouteur* (*Listener Interface*).
- **Un objet événement** : Objet représentant la source de l'événement capable de réagir à des évènements spécifiques à l'objet écouteur. Par exemple une fenêtre de type *JFrame* est un objet écouteur recevra un objet événement de type *WindowListener* ou *ActionListener* , envoyé par l'objet source de l'événement.
- **Un objet source d'événement** : Objet à l'origine de l'événement. Un objet de type *JButton* est la source de l'événement provoqué par le clic de souris de l'utilisateur.

> Un objet source d'événement peut très bien être recensé auprès du plusieurs écouteurs qui réagiront en même temps à l'événement.

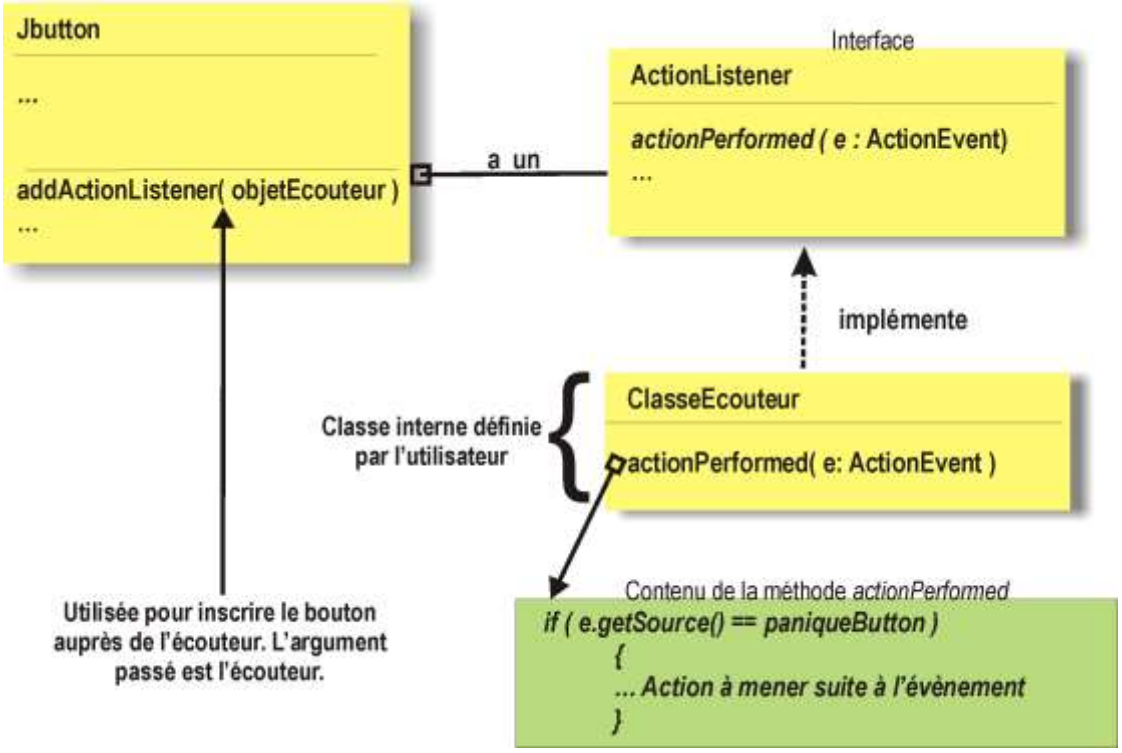
Voici un schéma résumant le mécanisme des évènements en Java.



Modèle des évènements en Java

B.3. Exemple de gestion de l'appui sur un bouton

Voici un exemple d'objets utilisés dans le cadre de la gestion des évènements.



Il s'agit d'un bouton de type *JButton* , écouté par un objet de type *ObjetEcouteur* dont la méthode *actionPerformed* exécutera le code appelé lorsqu'on appuiera sur le bouton. Voici comment coder un tel modèle dans le programme suivant :

```
1  import java.awt.event.*;
2  import javax.swing.*;
3
4  public class MaClasse extends JFrame {
5
6  private JButton paniqueButton = new JButton("Panique !!!");
7
8           public MaClasse () {
9
10          setTitle ("Petite fenêtre");
11          setSize (300,200);
12          ClasseEcouteur monListener = new ClasseEcouteur();
13          paniqueButton.addActionListener( monListener );
14          }
15
16          class ClasseEcouteur implements ActionListener {
17               public void actionPerformed ( ActionEvent e ) {
18                    if ( e.getSource() == paniqueButton)
19                         …// Action à réaliser
20                    }
21          }
22 }
```

- **Lignes 16 à 21** : Définition d'une classe interne dont les instances seront *écouteurs d'actions* (*action listener*). La méthode *actionPerformed* sera exécutée lors de l'appui sur un bouton. Le paramètre passé à cette méthode est l'objet événement de type *ActionEvent*. La méthode *getSource* de l'objet issu de *ActionEvent* renvoie la référence à l'objet ayant provoqué l'événement.
- **Ligne 13** : Recensement du bouton auprès de l'écouteur grâce à la méthode *addActionListener* qui prend comme paramètre l'objet écouteur.

C. Classes évènements

Les classes évènements sont destinées à créer des objets évènements. Il en existe beaucoup en Java. Voici quelques classes évènements :

- *ActionEvent* : Destiné à gérer les évènements de type action tel que l'appui sur un bouton par exemple.
- *TextEvent* : Destiné à gérer les évènements qui se produisent sur des objets qui manipulent du texte
- *WindowEvent* : Gère les évènements qui se produisent lorsqu'une fenêtre change d'état. Voir les API Java pour connaître les différents états d'une fenêtre.
- *MouseEvent* : Gère les événements bas niveau de l'action de l'utilisateur sur la souris.

➲ La gestion des évènements en Java ne vous oblige pas à utiliser les objets évènements fournis. Comme nous le verrons dans le cas des fenêtres, aucun objet événement n'est réellement utilisé.

D. Interfaces écouteurs (Listener Interface)

Il existe, dans les API Java , des interfaces utilisées pour gérer le comportement des objets écouteurs. En effet, si vous souhaitez créer vos propres classes écouteurs, il faudra qu'elles implémentent une interface spécialisée dans l'interception des évènements qui vous intéressent.
Par exemple, une classe souhaitant intercepter les évènements relatifs aux fenêtres devra implémenter l'interface *WindowListener*. Cela l'obligera à implémenter toutes les méthodes suivantes :

Méthodes de l'interface *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)

Une classe souhaitant intercepter les évènements relatifs à la souris devra implémenter l'interface *MouseListener*. Cela l'obligera à implémenter toutes les méthodes suivantes :

Méthodes de l'interface *MouseListener*	
void	**mouseClicked**(MouseEvent e)
void	**mouseEntered**(MouseEvent e)
void	**mouseExited**(MouseEvent e)
void	**mousePressed**(MouseEvent e)
void	**mouseReleased**(MouseEvent e)

Il est important de considérer qu'à partir du moment où une classe implémente une interface, cette classe doit implémenter TOUTES les méthodes définies dans une interface.

E. Classes adaptateurs

E.1. Principe

Nous avons vu que tout objet issu d'une classe désireuse d'écouter les évènements relatifs aux fenêtres devait implémenter l'interface *WindowListener*. Ceci peu devenir très fastidieux puisque cette dernière possède 7 méthodes. De plus, implémenter 7 méthodes alors que seule la méthode *windowClosing* nous intéresse , est encore plus rageant.
C'est pourquoi, les concepteurs des API Java ont crée les classes adaptateurs qui implémentent l'interface concernée et toutes les méthodes de celle-ci. Pour créer des classes écouteurs, il ne nous reste plus alors qu'à masquer la méthode qui nous intéresse.
C'est le cas de la classe *WindowAdapter* que l'on va dériver pour obtenir nos propres classes écouteurs.

E.2. Exemple

Rappelez-vous ce que nous faisions pour permettre à une application Java de se fermer correctement. Nous avions crée une classe *WindowCloser* qui héritait de *WindowAdapter* dans laquelle nous masquions seulement la méthode *windowClosing* :

```
1  public class WindowCloser extends WindowAdapter {
2            public void windowClosing(WindowEvent we) {
3                     System.exit(0);
4            }
5  }
```

WindowCloser.java

L'objet issu de cette classe est alors prêt à être écouté et à réagir à la fermeture de la fenêtre.
Voici le code d'une classe héritant de *JFrame* qui va se faire recensé auprès d'un objet de type *WindowCloser* qui saura réagir à la fermeture en quittant le programme :

```
1  import javax.swing.*;
2
3  public class Ex33 extends JFrame {
4
5  public Ex33(String titre) {
6
7            setTitle(titre);
8            setVisible(true);
9            // Recensement auprès d'un nouvel objet écouteur
10           addWindowListener(new WindowCloser());
11           }
12
13 // Programme principal
14 static void main ( String [] args ) {
15           // instanciation directement au sein de la classe
16           Ex33 myFrame = new Ex33("Fenêtre écoutée");
17 }
18
19 }
```

Ex33.java

➲ La plupart du temps, les classes destinées à écouter les objets sont des classes internes à la classe principale. Ceci pour des raisons de simplicité évidente vu que cese classes ne sont pas très volumineuses.

Index