



JAVA 8 – APPLICATIONS GRAPHIQUES

JSE – SEQUENCE 4

LYCEE PASTEUR MONT-ROLAND
Julian.courbez@gmail.com

Table des matières

1.	Introduction.....	2
1.2.	Les bibliothèques graphiques.....	2
	La bibliothèque AWT	2
	La bibliothèque Swing	2
1.2.	Constitution de l'interface graphique d'une application	3
2.	Conception d'une interface graphique	3
2.1.	Les fenêtres	3
2.2.	La gestion des événements	7
	Utilisation d'une classe « normale » implémentant l'interface	9
	Implémenter l'interface dans un classe déjà existante.....	11
	Créer une classe interne implémentant l'interface.....	13
	Créer une classe interne anonyme implémentant l'interface	15
	Créer une classe « normale » héritant d'une classe XXXXAdapter	17
	Créer une classe interne héritant d'une classe XXXXAdapter	18
	Créer une classe interne anonyme héritant d'une classe XXXXAdapter	20
2.3.	Aspect des composants.....	36
2.4.	Le positionnement des composants.....	37
	FlowLayout	37
	BorderLayout.....	39
	GridLayout	45
	BoxLayout.....	46
	GridBagLayout	49
	Sans gestionnaire de mise en page	53
2.5.	Les composants graphiques	56
	La classe JComponent.....	56
	Affichage d'information	59
	Les composants d'édition de texte	62
	Les composants de déclenchement d'actions.....	68
	JMenuBar, JMenu, JMenuItem, JPopupMenu, JSeparator	69
	Les composatsns de sélection.....	75
2.6.	Les boîtes de dialogue	83
	La boîte de saisie	83
	La boîte de message	85
	La boîte de confirmation	86

1. Introduction

La plupart des utilisateurs de vos futures applications s'attendent certainement à avoir une interface un petit peu moins austère qu'un écran en mode caractères. Nous allons donc étudier dans ce chapitre comment fonctionnent les interfaces graphiques avec Java. Vous allez rapidement vous apercevoir que la conception d'interfaces graphiques en Java n'est pas très simple et nécessite l'écriture de nombreuses lignes de code.

Dans la pratique, de nombreux outils de développement sont capables de prendre en charge la génération d'une grande partie de ce code en fonction de la conception graphique de l'application que vous dessinez. Il est cependant important de bien comprendre les principes de fonctionnement de ce code pour éventuellement intervenir dessus et l'optimiser.

1.2. Les bibliothèques graphiques

Le langage Java propose deux bibliothèques dédiées à la conception d'interfaces graphiques. La bibliothèque AWT et la bibliothèque SWING.

Les principes d'utilisation sont quasiment identiques pour ces deux bibliothèques. L'utilisation simultanée de ces deux bibliothèques dans une même application peut provoquer des problèmes de fonctionnement et doit être évitée.

La bibliothèque AWT

Cette bibliothèque est la toute première disponible pour le développement d'interfaces graphiques. Elle contient une multitude de classes et interfaces permettant la définition et la gestion d'interfaces graphiques. Cette bibliothèque utilise en fait les fonctionnalités graphiques du système d'exploitation. Ce n'est donc pas le code présent dans cette bibliothèque qui assure le rendu graphique des différents composants. Ce code sert uniquement d'intermédiaire avec le système d'exploitation. Son utilisation est de ce fait relativement économe en ressources. Par contre elle souffre de plusieurs inconvénients.

- L'aspect visuel de chaque composant étant lié à la représentation qu'en fait le système d'exploitation, il est parfois délicat de développer une application ayant une apparence cohérente sur tous les systèmes. La taille et la position des différents composants étant les deux éléments principalement affectés par ce problème.
- Pour que cette bibliothèque soit compatible avec tous les systèmes d'exploitation, les composants qu'elle contient sont donc limités aux plus courants (boutons, zone de texte, listes...).

La bibliothèque Swing

Cette bibliothèque a été conçue pour pallier les principales insuffisances de la bibliothèque AWT. Cette amélioration a été obtenue en écrivant entièrement cette bibliothèque en Java sans pratiquement faire appel aux services du système d'exploitation. Seuls quelques éléments graphiques (fenêtres et boîtes de dialogue) sont encore en relation avec le système d'exploitation. Pour les autres composants, c'est le code de la bibliothèque Swing qui détermine entièrement leurs aspects et comportements.

La bibliothèque Swing contient donc une quantité impressionnante de classes servant à redéfinir les composants graphiques. Il ne faut cependant pas penser que la bibliothèque Swing rend complètement obsolète la bibliothèque AWT. Beaucoup d'éléments de la bibliothèque AWT sont d'ailleurs repris dans la bibliothèque Swing.

1.2. Constitution de l'interface graphique d'une application

La conception de l'interface graphique d'une application consiste essentiellement à créer des instances des classes représentant les différents éléments nécessaires, modifier les caractéristiques de ces instances de classe, les assembler et prévoir le code de gestion des différents événements pouvant intervenir au cours du fonctionnement de l'application.

Une application graphique est donc constituée d'une multitude d'éléments superposés ou imbriqués. Parmi ces éléments, l'un d'entre eux joue un rôle prépondérant dans l'application. Il est souvent appelé conteneur de premier niveau. C'est lui qui va interagir avec le système d'exploitation et contenir tous les autres éléments. En général ce conteneur de premier niveau ne contient pas directement les composants graphiques mais d'autres conteneurs sur lesquels sont placés les composants graphiques. Pour faciliter la disposition de ces éléments les uns par rapport aux autres, nous pouvons utiliser l'aide d'un gestionnaire de mise en page. Cette superposition d'éléments peut être assimilée à une arborescence au sommet de laquelle nous avons le conteneur de premier niveau et dont les différentes branches sont constituées d'autres conteneurs. Les feuilles de l'arborescence correspondant aux composants graphiques.

Le conteneur de premier niveau étant l'élément indispensable de toute application graphique, nous allons donc commencer par étudier en détail ces caractéristiques et son utilisation puis nous étudierons les principaux composants graphiques.

2. Conception d'une interface graphique

Toute application graphique est au moins constituée d'un conteneur de premier niveau. La bibliothèque Swing dispose de trois classes permettant de jouer ce rôle :

- **JApplet** : représente une fenêtre graphique embarquée à l'intérieur d'une page html pour être prise en charge par un navigateur.
- **JWindow** : représente une fenêtre graphique la plus rudimentaire qui soit. Celle-ci ne dispose pas de barre de titre, de menu système, pas de bordure, c'est en fait un simple rectangle sur l'écran. Cette classe est rarement utilisée sauf pour l'affichage d'un écran d'accueil lors du démarrage d'une application (*splash screen*).
- **JFrame** : représente une fenêtre graphique complète et pleinement fonctionnelle. Elle dispose d'une barre de titre, d'un menu système, d'une bordure, elle peut facilement accueillir un menu, c'est bien sûr cet élément que nous allons utiliser dans la très grande majorité des cas.

2.1. Les fenêtres

La classe **JFrame** est l'élément indispensable de toute application graphique. Comme pour une classe normale nous devons créer une instance, modifier éventuellement les propriétés et utiliser les méthodes. Voici donc le code de la première application graphique.

```
package jse.s4;

import javax.swing.JFrame;

public class Principale {

    public static void main(String[] args)

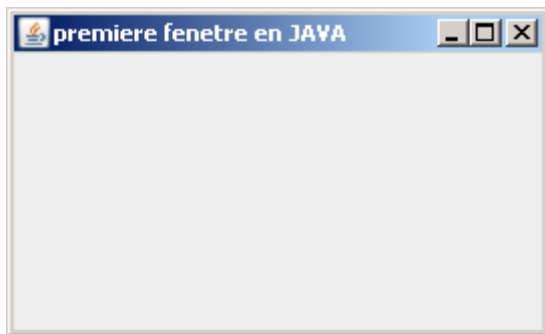
    {

        JFrame fenetre;

        // création de l'instance de la classe JFrame
```

```
fenetre=new JFrame();  
  
// modification de la position et de la  
// taille de la fenetre  
fenetre.setBounds(0,0,300,400);  
  
// modification du titre de la fenetre  
fenetre.setTitle("première fenetre en JAVA");  
  
// affichage de la fenetre  
fenetre.setVisible(true);  
  
}
```

Et le résultat de son exécution :



C'est simple d'utilisation et très efficace. C'est d'ailleurs tellement efficace que vous ne pouvez pas arrêter l'application. En effet même si la fenêtre est fermée par l'utilisateur, cette fermeture ne provoque pas la suppression de l'instance de la **JFrame** de la mémoire. La seule solution pour arrêter l'application est de stopper la machine virtuelle Java avec la combinaison de touches [Ctrl] **C**. Il faut bien sûr prévoir une autre solution pour terminer plus facilement l'exécution de l'application et faire en sorte que celle-ci s'arrête à la fermeture de la fenêtre.

La première solution consiste à gérer les événements se produisant lors de la fermeture de la fenêtre et dans un de ces événements provoquer l'arrêt de l'application. Cette solution sera étudiée dans le paragraphe consacré à la gestion des événements.

La deuxième solution utilise des comportements prédéfinis pour la fermeture de la fenêtre. Ces comportements sont déterminés par la méthode **setDefaultCloseOperation**. Plusieurs constantes sont définies pour déterminer l'action entreprise à la fermeture de la fenêtre.

DISPOSE_ON_CLOSE : cette option provoque l'arrêt de l'application lors de la fermeture de la dernière fenêtre prise en charge par la machine virtuelle.

DO_NOTHING_ON_CLOSE : avec cette option, il ne se passe rien lorsque l'utilisateur demande la fermeture de la fenêtre. Il est dans ce cas obligatoire de gérer les événements pour que l'action de l'utilisateur provoque un effet sur la fenêtre ou l'application.

EXIT_ON_CLOSE : cette option provoque l'arrêt de l'application même si d'autres fenêtres sont encore visibles.

HIDE_ON_CLOSE : avec cette option, la fenêtre est simplement masquée par un appel à sa méthode `setVisible(false)`.

La classe **JFrame** se trouve située à la fin d'une hiérarchie de classes assez importante et implémente de nombreuses interfaces. De ce fait elle possède donc de nombreuses méthodes et attributs.

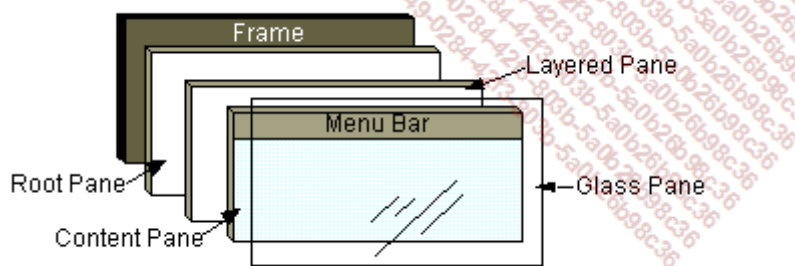
Class JFrame

```
java.lang.Object
├── java.awt.Component
│   ├── java.awt.Container
│   │   ├── java.awt.Window
│   │   │   ├── java.awt.Frame
│   │   │   └── javax.swing.JFrame
```

All Implemented Interfaces:

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

Maintenant que nous sommes capables d'afficher une fenêtre, le plus gros de notre travail va consister à ajouter un contenu à la fenêtre. Avant de pouvoir ajouter quelque chose sur une fenêtre, il faut bien comprendre sa structure qui est relativement complexe. Un objet **JFrame** est composé de plusieurs éléments superposés jouant chacun un rôle bien spécifique dans la gestion de la fenêtre.



L'élément **RootPane** correspond au conteneur des trois autres éléments.

L'élément **LayeredPane** est lui responsable de la gestion de la position des éléments aussi bien sur les axes X et Y que sur l'axe Z ce qui permet la superposition de différents éléments.

L'élément **ContentPane** est le conteneur de base de tous les éléments ajoutés sur la fenêtre. C'est bien sûr à lui que nous allons confier les différents composants de l'interface de l'application. Par-dessous le **ContentPane**, se superpose le **GlassPane** comme on le fait avec une vitre placée sur une photo. Il présente d'ailleurs beaucoup de similitudes avec la vitre.

- Il est transparent par défaut.
- Ce qui est dessiné sur le **GlassPane** masque les autres éléments.
- Il est capable d'intercepter les événements liés à la souris avant que ceux-ci n'atteignent les autres composants.

De tous ces éléments, c'est incontestablement le **ContentPane** que nous allons le plus utiliser.

Celui-ci est accessible par la méthode **getContentPane** de la classe **JFrame**. Il est techniquement possible de placer des composants directement sur l'objet **ContentPane** mais c'est une pratique déconseillée par Oracle. Il est préférable d'intercaler un conteneur intermédiaire qui lui va contenir les composants et de placer ce conteneur sur le **ContentPane**. Le composant **JPanel** est le plus couramment utilisé dans ce rôle.

Le scénario classique de conception d'une interface graphique consiste donc à créer les différents composants puis à les placer sur un conteneur et enfin à placer ce conteneur sur le **ContentPane** de

la fenêtre. L'exemple suivant met cela en application en créant une interface utilisateur composée de trois boutons.

```
package jse.s4;

import java.awt.Graphics;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Principale {
    public static void main(String[] args)

    {
        // création de la fenêtre
        JFrame fenetre;
        fenetre=new JFrame();
        fenetre.setTitle("première fenêtre en JAVA");
        fenetre.setBounds(0,0,300,100);
        fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // création des trois boutons
        JButton b1,b2,b3;
        b1=new JButton("Rouge");
        b2=new JButton("Vert");
        b3=new JButton("Bleu");
        // création du conteneur intermédiaire
        JPanel pano;
        pano=new JPanel();
        // ajout des boutons sur le conteneur intermédiaire
        pano.add(b1);
        pano.add(b2);
        pano.add(b3);
        // ajout du conteneur intermédiaire sur le ContentPane
        fenetre.getContentPane().add(pano);
        // affichage de la fenêtre
        fenetre.setVisible(true);
    }
}
```

À l'exécution, ce code affiche la fenêtre suivante :



L'étape suivante de notre étude va nous permettre de déterminer ce que doit faire l'application lorsque l'utilisateur va cliquer sur l'un des boutons.

2.2. La gestion des événements

L'élément à l'origine de l'événement est appelé source d'événement et l'élément contenant la portion de code chargée de gérer l'événement est appelée écouteur d'événement. Les sources d'événements gèrent, pour chaque événement qu'elles peuvent déclencher, une liste leur permettant de savoir quels écouteurs doivent être avertis si l'événement se produit. Les sources d'événements et les écouteurs d'événements sont bien entendu des objets.

Il faut bien sûr que les écouteurs soient prévus pour gérer les événements que va leur transmettre la source d'événement. Pour garantir cela, à chaque type d'événement correspond une interface que doit implémenter un objet s'il veut être candidat pour la gestion de cet événement. Pour éviter la multiplication des interfaces (déjà très nombreuses), les événements sont regroupés par catégories. Le nom de ces interfaces respecte toujours la convention suivante :

La première partie du nom est représentative de la catégorie d'événements pouvant être gérés par les objets implémentant cette interface. Le nom se termine toujours par **Listener**.

Nous avons par exemple l'interface **MouseEventListener** correspondant aux événements déclenchés par les déplacements de la souris ou l'interface **ActionListener** correspondant à un clic sur un bouton. C'est dans chacune de ces interfaces que nous trouvons les signatures des différentes méthodes associées à chaque événement.

```
public interface MouseEventListener
extends EventListener
{
    void mouseDragged(MouseEvent e);
    void mouseMoved(MouseEvent e);
}
```

Chacune des méthodes attend comme argument un objet représentant l'événement lui-même. Cet objet est créé automatiquement lors du déclenchement de l'événement puis passé comme argument à la méthode chargée de gérer l'événement dans l'écouteur d'événement. Il contient généralement des informations complémentaires concernant l'événement et il est spécifique à chaque type d'événement.

Nous avons donc besoin de créer des classes implémentant ces interfaces. De ce point de vue, nous avons une multitude de possibilités :

- Créer une classe "normale" implémentant l'interface.
- Implémenter l'interface dans une classe déjà existante.
- Créer une classe interne implémentant l'interface.

- Créer une classe interne anonyme implémentant l'interface.

L'utilisation d'une classe interne anonyme est la solution la plus fréquemment utilisée avec le petit inconvénient d'avoir une syntaxe difficilement lisible lorsque l'on n'y est pas habitué.

Pour clarifier tout cela, nous allons illustrer chacune de ces possibilités par un petit exemple. Cet exemple va nous permettre de terminer proprement l'application lors de la fermeture de la fenêtre principale en appelant la méthode **System.exit(0)**. Cette solution permet d'effectuer des vérifications avant l'arrêt de l'application (sauvegarde, affichage d'un message de confirmation, déconnexion de l'utilisateur...). Il faut, dans ce cas, modifier la propriété **DefaultCloseOperation** de la fenêtre avec la valeur **DO_NOTHING_ON_CLOSE** pour qu'il n'y ait plus d'action par défaut.

Cette méthode doit être appelée lors de la détection de la fermeture de la fenêtre. Pour cela, nous devons gérer les événements liés à la fenêtre et plus particulièrement l'événement **windowClosing** qui est déclenché au moment où l'utilisateur demande la fermeture de la fenêtre par le menu système. L'interface **WindowListener** est tout à fait adaptée pour ce genre de travail.

Notre base de travail est constituée des deux classes suivantes :

```
package jse.s4;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Ecran
extends JFrame
{
    public Ecran()
    {
        setTitle("première fenêtre en JAVA");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // création des trois boutons
        JButton b1,b2,b3;
        b1=new JButton("Rouge");
        b2=new JButton("Vert");
        b3=new JButton("Bleu");
        // création du conteneur intermédiaire
        JPanel pano;
        pano=new JPanel();
        // ajout des boutons sur le conteneur intermédiaire
        pano.add(b1);
        pano.add(b2);
```

```

        pano.add(b3);
        // ajout du conteneur sur le ContentPane
        getContentPane().add(pano);
    }
}

package jse.s4;

public class Principale {

    public static void main(String[] args)

    {
        // création de la fenêtre
        Ecran fenetre;
        fenetre=new Ecran();
        // affichage de la fenêtre
        fenetre.setVisible(true);
    }
}

```

Si nous exécutons ce code, la fenêtre apparaît mais il n'est plus possible de la fermer et encore moins d'arrêter l'application. Voyons maintenant comment remédier à ce problème avec les différentes solutions évoquées plus haut.

Utilisation d'une classe « normale » implémentant l'interface

```

package jse.s4;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

public class EcouteurFenetre implements WindowListener {

    public void windowActivated(WindowEvent arg0)
    {
    }

    public void windowClosed(WindowEvent arg0)
    {
    }

    public void windowClosing(WindowEvent arg0)

```

```

    {
        System.exit(0);
    }
    public void windowDeactivated(WindowEvent arg0)
    {
    }
    public void windowDeiconified(WindowEvent arg0)
    {
    }
    public void windowIconified(WindowEvent arg0)
    {
    }
    public void windowOpened(WindowEvent arg0)
    {
    }
}

package jse.s4;

public class Principale {

    public static void main(String[] args)

    {
        // création de la fenêtre
        Ecran fenetre;
        fenetre=new Ecran();
        // création d'une instance de la classe chargée
        // de gérer les événements
        EcouteurFenetre ef;
        ef=new EcouteurFenetre();
        // référencement de cette instance de classe
        // comme écouteur d'événement pour la fenêtre
        fenetre.addWindowListener(ef);
        // affichage de la fenêtre
        fenetre.setVisible(true);
    }
}

```

Implémenter l'interface dans une classe déjà existante

Dans cette solution, nous allons confier à la classe représentant la fenêtre le soin de gérer ses propres événements en lui faisant implémenter l'interface **WindowListener**.

```
package jse.s4;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Ecran extends JFrame
    implements WindowListener
{
    public Ecran()
    {
        setTitle("première fenêtre en JAVA");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // création des trois boutons
        JButton b1,b2,b3;
        b1=new JButton("Rouge");
        b2=new JButton("Vert");
        b3=new JButton("Bleu");
        // création du conteneur intermédiaire
        JPanel pano;
        pano=new JPanel();
        // ajout des boutons sur le conteneur intermédiaire
        pano.add(b1);
        pano.add(b2);
        pano.add(b3);
        // ajout du conteneur intermédiaire sur le ContentPane
        getContentPane().add(pano);
        // référencement de la fenêtre elle-même
        // comme écouteur de ses propres événements

        addWindowListener(this);
    }
}
```

```

    }

    public void windowActivated(WindowEvent arg0)
    {
    }

    public void windowClosed(WindowEvent arg0)
    {
    }

    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }

    public void windowDeactivated(WindowEvent arg0)
    {
    }

    public void windowDeiconified(WindowEvent arg0)
    {
    }

    public void windowIconified(WindowEvent arg0)
    {
    }

    public void windowOpened(WindowEvent arg0)
    {
    }
}
package jse.s4;

public class Principale {

    public static void main(String[] args)

    {
        // création de la fenêtre
        Ecran fenetre;
        fenetre=new Ecran();
        // affichage de la fenêtre
        fenetre.setVisible(true);
    }
}

```

Avec cette solution, le code est centralisé dans une seule et unique classe. S'il y a de nombreux événements à gérer, cette classe va comporter une multitude de méthodes.

Créer une classe interne implémentant l'interface

Cette solution est un mélange des deux précédentes puisque nous avons une classe spécifique pour la gestion des événements mais celle-ci est définie à l'intérieur de la classe correspondant à la fenêtre.

```
package jse.s4;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Ecran extends JFrame

{
    public Ecran()
    {
        setTitle("première fenêtre en JAVA");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // création des trois boutons
        JButton b1,b2,b3;
        b1=new JButton("Rouge");
        b2=new JButton("Vert");
        b3=new JButton("Bleu");
        // création du conteneur intermédiaire
        JPanel pano;
        pano=new JPanel();
        // ajout des boutons sur le conteneur intermédiaire
        pano.add(b1);
        pano.add(b2);
        pano.add(b3);
        // ajout du conteneur intermédiaire sur le ContentPane
        getContentPane().add(pano);
        // création d'une instance de la classe chargée
        // de gérer les événements
    }
}
```

```

        EcouteurFenetre ef;

        ef=new EcouteurFenetre();

        // référencement de cette instance de classe
        // comme écouteur d'événement pour la fenêtre

        addWindowListener(ef);
    }

    public class EcouteurFenetre implements WindowListener
    {
        public void windowActivated(WindowEvent arg0)
        {
        }

        public void windowClosed(WindowEvent arg0)
        {
        }

        public void windowClosing(WindowEvent arg0)
        {
            System.exit(0);
        }

        public void windowDeactivated(WindowEvent arg0)
        {
        }

        public void windowDeiconified(WindowEvent arg0)
        {
        }

        public void windowIconified(WindowEvent arg0)
        {
        }

        public void windowOpened(WindowEvent arg0)
        {
        }
    }
}

package jse.s4;

public class Principale {

    public static void main(String[] args)

```

```

    {
        // création de la fenêtre
        Ecran fenetre;
        fenetre=new Ecran();
        // affichage de la fenêtre
        fenetre.setVisible(true);
    }
}

```

Avec cette solution les responsabilités sont bien réparties entre plusieurs classes, par contre nous allons avoir une multiplication du nombre de classes.

Créer une classe interne anonyme implémentant l'interface

Cette solution est une petite variante de la précédente puisque nous avons toujours une classe spécifique chargée de la gestion des événements mais celle-ci est déclarée au moment de son instantiation.

```

package jse.s4;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Ecran extends JFrame
{
    public Ecran()
    {
        setTitle("première fenêtre en JAVA");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // création des trois boutons
        JButton b1,b2,b3;
        b1=new JButton("Rouge");
        b2=new JButton("Vert");
        b3=new JButton("Bleu");
        // création du conteneur intermédiaire
        JPanel pano;
    }
}

```



```

        pano=new JPanel();
        // ajout des boutons sur le conteneur intermédiaire
        pano.add(b1);
        pano.add(b2);
        pano.add(b3);
        // ajout du conteneur intermédiaire sur le ContentPane
        getContentPane().add(pano);
        // création d'une instance d'une classe anonyme
        // chargée de gérer les événements
        addWindowListener(new WindowListener()
        // début de la définition de la classe
        {
            public void windowActivated(WindowEvent arg0)
            {
            }
            public void windowClosed(WindowEvent arg0)
            {
            }
            public void windowClosing(WindowEvent arg0)
            {
                System.exit(0);
            }
            public void windowDeactivated(WindowEvent arg0)
            {
            }
            public void windowDeiconified(WindowEvent arg0)
            {
            }
            public void windowIconified(WindowEvent arg0)
            {
            }
            public void windowOpened(WindowEvent arg0)
            {
            }
        } // fin de la définition de la classe
    ); // fin de l'appel de la méthode addWindowListener
} // fin du constructeur
} // fin de la classe Ecran

```

```

package jse.s4;

public class Principale {

    public static void main(String[] args)

    {

        // création de la fenêtre
        Ecran fenetre;

        fenetre=new Ecran();

        // affichage de la fenêtre
        fenetre.setVisible(true);

    }

}

```

Le seul reproche que l'on puisse faire à cette solution réside dans la relative complexité de la syntaxe. Les commentaires placés sur les différentes lignes fournissent une aide précieuse pour s'y retrouver dans les accolades et parenthèses.

Par contre, il y a un reproche que l'on peut faire de manière globale à toutes ces solutions. Pour une seule méthode réellement utile, nous sommes obligés d'en écrire sept.

Pour éviter ce code inutile, nous pouvons travailler avec une classe implémentant déjà la bonne interface et simplement redéfinir uniquement les méthodes nous intéressant.

Créer une classe « normale » héritant d'une classe XXXXAdapter

```

package jse.s4;

import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class Ecouteurfenetre extends WindowAdapter
{
    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
}

package jse.s4;

public class Principale {

    public static void main(String[] args)

```

```

{
    // création de la fenêtre
    Ecran fenetre;

    fenetre=new Ecran();

    // création d'une instance de la classe chargée
    // de gérer les événements
    EcouteurFenetre ef;
    ef=new EcouteurFenetre();
    // référencement de cette instance de classe
    // comme écouteur d'événement pour la fenêtre
    fenetre.addWindowListener(ef);
    // affichage de la fenêtre
    fenetre.setVisible(true);
}
}

```

Créer une classe interne héritant d'une classe XXXXAdapter

```

package jse.s4;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Ecran extends JFrame

{
    public Ecran()
    {
        setTitle("première fenêtre en JAVA");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // création des trois boutons
        JButton b1,b2,b3;
        b1=new JButton("Rouge");
        b2=new JButton("Vert");
        b3=new JButton("Bleu");
    }
}

```

```

        // création du conteneur intermédiaire
        JPanel pano;
        pano=new JPanel();
        // ajout des boutons sur le conteneur intermédiaire
        pano.add(b1);
        pano.add(b2);
        pano.add(b3);
        // ajout du conteneur intermédiaire sur le ContentPane
        getContentPane().add(pano);
        // création d'une instance de la classe chargée
        // de gérer les événements
        EcouteurFenetre ef;
        ef=new EcouteurFenetre();
        // comme écouteur d'événement pour la fenêtre

        addWindowListener(ef);
    }

    public class Ecouteurfenetre extends WindowAdapter
    {
        public void windowClosing(WindowEvent arg0)
        {
            System.exit(0);
        }
    }
}

package jse.s4;

public class Principale {

    public static void main(String[] args)

    {
        // création de la fenêtre
        Ecran fenetre;
        fenetre=new Ecran();
        // affichage de la fenêtre
        fenetre.setVisible(true);
    }
}

```

```
}  
}
```

Créer une classe interne anonyme héritant d'une classe XXXXAdapter

```
package jse.s4;  
  
import java.awt.event.WindowEvent;  
import java.awt.event.WindowListener;  
  
import javax.swing.JButton;  
import javax.swing.JFrame;  
import javax.swing.JPanel;  
  
public class Ecran extends JFrame  
{  
    public Ecran()  
    {  
        setTitle("première fenêtre en JAVA");  
        setBounds(0,0,300,100);  
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);  
        // création des trois boutons  
        JButton b1,b2,b3;  
        b1=new JButton("Rouge");  
        b2=new JButton("Vert");  
        b3=new JButton("Bleu");  
        // création du conteneur intermédiaire  
        JPanel pano;  
        pano=new JPanel();  
        // ajout des boutons sur le conteneur intermédiaire  
        pano.add(b1);  
        pano.add(b2);  
        pano.add(b3);  
        // ajout du conteneur intermédiaire sur le ContentPane  
        getContentPane().add(pano);  
        // création d'une instance d'une classe anonyme  
        // chargée de gérer les événements  
        addWindowListener(new WindowAdapter()  
        // début de la définition de la classe  
        {
```

```

        public void windowClosing(WindowEvent arg0)
        {
            System.exit(0);
        }
    } // fin de la définition de la classe
); // fin de l'appel de la méthode addWindowListener
} // fin du constructeur
} // fin de la classe Ecran

package jse.s4;

public class Principale {

    public static void main(String[] args)

    {
        // création de la fenêtre
        Ecran fenetre;
        fenetre=new Ecran();
        // affichage de la fenêtre
        fenetre.setVisible(true);
    }
}

```

Cette solution est bien sûr la plus économe en nombre de lignes et c'est celle qui est utilisée par de nombreux outils de développement générant automatiquement du code. La relative complexité du code peut parfois être troublante lorsque l'on n'y est pas habitué.

Jusqu'à présent, nous avons une source d'événement et un écouteur pour cette source d'événement. Nous pouvons avoir dans certains cas la situation où nous avons plusieurs sources d'événements et souhaiter utiliser le même écouteur ou avoir une source d'événement et prévenir plusieurs écouteurs. La situation classique où nous avons plusieurs sources d'événements et un seul écouteur se produit lorsque nous fournissons à l'utilisateur plusieurs solutions pour lancer l'exécution d'une même action (menu et barre d'outils ou boutons).

Quel que soit le moyen utilisé pour lancer l'action, le code à exécuter reste le même.

Nous pouvons dans ce cas de figure utiliser le même écouteur pour les deux sources d'événements. Pour illustrer cela, nous allons ajouter un menu à l'application et faire en sorte que l'utilisation du menu ou d'un des boutons lance la même action en modifiant la couleur de fond correspondante au bouton ou au menu utilisé. Comme nous devons utiliser le même écouteur pour deux sources d'événements il est préférable d'utiliser une classe interne pour la création de l'écouteur. Voici ci-dessous le code correspondant.

```

package jse.s4;

```

```
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;

public class Ecran extends JFrame

{
    JPanel pano;
    public Ecran ()
    {
        setTitle("première fenêtre en JAVA");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // création des trois boutons
        JButton btnRouge,btnVert,btnBleu;
        btnRouge=new JButton("Rouge");
        btnVert=new JButton("Vert");
        btnBleu=new JButton("Bleu");
        // création des trois écouteurs
        EcouteurRouge ecR;
        EcouteurVert ecV;
        EcouteurBleu ecB;
        ecR=new EcouteurRouge();
        ecV=new EcouteurVert();
        ecB=new EcouteurBleu();
        // association de l'écouteur à chaque bouton
        btnRouge.addActionListener(ecR);
        btnVert.addActionListener(ecV);
        btnBleu.addActionListener(ecB);
        // Création du menu
```

```

JMenuBar barreMenu;

barreMenu=new JMenuBar();

JMenu mnuCouleurs;

mnuCouleurs=new JMenu("Couleurs");

barreMenu.add(mnuCouleurs);

JMenuItem mnuRouge,mnuVert,mnuBleu;

mnuRouge=new JMenuItem("Rouge");

mnuVert=new JMenuItem("Vert");

mnuBleu=new JMenuItem("Bleu");

mnuCouleurs.add(mnuRouge);

mnuCouleurs.add(mnuVert);

mnuCouleurs.add(mnuBleu);

// association de l'écouteur à chaque menu
// (les mêmes que pour les boutons)
mnuRouge.addActionListener(ecR);
mnuVert.addActionListener(ecV);
mnuBleu.addActionListener(ecB);

// ajout du menu sur la fenêtre
setJMenuBar(barreMenu);

// création du conteneur intermédiaire
pano=new JPanel();

// ajout des boutons sur le conteneur intermédiaire
pano.add(btnRouge);

pano.add(btnVert);

pano.add(btnBleu);

// ajout du conteneur intermédiaire sur le ContentPane
getContentPane().add(pano);

// création d'une instance d'une classe anonyme
// chargée de gérer les événements de la fenêtre
addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
});
}

public class EcouteurRouge implements ActionListener

```



```

{
    public void actionPerformed(ActionEvent arg0)
    {
        pano.setBackground(Color.RED);
    }
}

public class EcouteurVert implements ActionListener
{
    public void actionPerformed(ActionEvent arg0)
    {
        pano.setBackground(Color.GREEN);
    }
}

public class EcouteurBleu implements ActionListener
{
    public void actionPerformed(ActionEvent arg0)
    {
        pano.setBackground(Color.BLUE);
    }
}
}

```

Dans ce code, nous avons nos trois classes écouteur qui sont très similaires. Avec une petite astuce, nous allons pouvoir simplifier le code pour n'avoir plus qu'une seule classe écouteur pour les trois boutons. La même méthode **actionPerformed** sera appelée suite à un clic sur n'importe lequel des boutons. Le choix de l'action à exécuter sera fait à l'intérieur de cette méthode. Pour cela, nous allons utiliser le paramètre **ActionEvent** fourni à cette méthode. Celui-ci permet d'obtenir une référence sur objet à l'origine de l'événement par l'intermédiaire de la méthode **getSource**. Le code simplifié est présenté ci-dessous :

```

package jse.s4;

import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;

```

```
import javax.swing.JPanel;

public class Ecran extends JFrame

{
    JPanel pano;
    JButton btnRouge, btnVert, btnBleu;
    JMenuItem mnuRouge, mnuVert, mnuBleu;
    public Ecran ()
    {
        setTitle("première fenêtre en JAVA");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // création des trois boutons
        btnRouge=new JButton("Rouge");
        btnVert=new JButton("Vert");
        btnBleu=new JButton("Bleu");
        // création des trois écouteurs
        EcouteurCouleur ec;
        ec=new EcouteurCouleur();
        // association de l'écouteur à chaque bouton
        btnRouge.addActionListener(ec);
        btnVert.addActionListener(ec);
        btnBleu.addActionListener(ec);
        // Création du menu
        JMenuBar barreMenu;
        barreMenu=new JMenuBar();
        JMenu mnuCouleurs;
        mnuCouleurs=new JMenu("Couleurs");
        barreMenu.add(mnuCouleurs);

        mnuRouge=new JMenuItem("Rouge");
        mnuVert=new JMenuItem("Vert");
        mnuBleu=new JMenuItem("Bleu");
        mnuCouleurs.add(mnuRouge);
        mnuCouleurs.add(mnuVert);
        mnuCouleurs.add(mnuBleu);
        // association de l'écouteur à chaque menu
        // (le même que pour les boutons)
```

```

mnuRouge.addActionListener(ec);
mnuVert.addActionListener(ec);
mnuBleu.addActionListener(ec);
// ajout du menu sur la fenetre
setJMenuBar(barreMenu);
// création du conteneur intermédiaire
pano=new JPanel();
// ajout des boutons sur le conteneur intermédiaire
pano.add(btnRouge);
pano.add(btnVert);
pano.add(btnBleu);
// ajout du conteneur intermédiaire sur le ContentPane
getContentPane().add(pano);
// création d'une instance d'une classe anonyme
// chargée de gérer les événements
addWindowListener(new WindowAdapter()

{
    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
});
}

public class EcouteurCouleur implements ActionListener
{
    public void actionPerformed(ActionEvent arg0)
    {
        if (arg0.getSource()==btnRouge | arg0.getSource()==mnuRouge)
        {
            pano.setBackground(Color.RED);
        }
        if (arg0.getSource()==btnVert | arg0.getSource()==mnuVert)
        {
            pano.setBackground(Color.GREEN);
        }
        if (arg0.getSource()==btnBleu | arg0.getSource()==mnuBleu)
        {
            pano.setBackground(Color.BLUE);
        }
    }
}

```

```

        }

    }

}

}

```

À noter que pour que cette solution puisse fonctionner, les objets source d'événements doivent être accessibles depuis la classe écouteur d'événements. La déclaration des boutons et des éléments de menu a donc été déplacée au niveau de la classe elle-même et non plus dans le constructeur comme c'était le cas dans la version précédente. Cette solution est possible uniquement si la classe écouteur est une classe interne. Dans le cas où la classe écouteur est parfaitement indépendante de la classe où sont créés les objets source d'événement, il faut revoir le code de la méthode **actionPerformed**. Le paramètre **ActionEvent** de la méthode **actionPerformed** nous fournit une autre solution pour contourner ce problème. Par l'intermédiaire de la méthode **getActionCommand** nous avons accès à une chaîne de caractères représentant l'objet à l'origine de l'événement. Par défaut cette chaîne de caractères correspond au libellé du composant ayant déclenché l'événement mais elle peut être modifiée par la méthode **setActionCommand** de chaque composant. C'est d'ailleurs une pratique recommandée puisqu'elle nous permet d'avoir un code identique pour une application fonctionnant en plusieurs langues. Voici les modifications correspondantes.

```

package jse.s4;

import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;

public class Ecran extends JFrame

{
    JPanel pano;

    JButton btnRouge, btnVert, btnBleu;

    JMenuItem mnuRouge, mnuVert, mnuBleu;

    public Ecran ()

```

```

{
    setTitle("première fenêtre en JAVA");
    setBounds(0,0,300,100);
    setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
    // création des trois boutons
    btnRouge=new JButton("Rouge");
    btnRouge.setActionCommand("red");
    btnVert=new JButton("Vert");
    btnVert.setActionCommand("green");
    btnBleu=new JButton("Bleu");
    btnBleu.setActionCommand("blue");
    // création des trois écouteurs
    EcouteurCouleur ec;
    ec=new EcouteurCouleur();
    // association de l'écouteur à chaque bouton
    btnRouge.addActionListener(ec);
    btnVert.addActionListener(ec);
    btnBleu.addActionListener(ec);
    // Création du menu
    JMenuBar barreMenu;
    barreMenu=new JMenuBar();
    JMenu mnuCouleurs;
    mnuCouleurs=new JMenu("Couleurs");
    barreMenu.add(mnuCouleurs);
    mnuRouge=new JMenuItem("Rouge");
    mnuRouge.setActionCommand("red");
    mnuVert=new JMenuItem("Vert");
    mnuVert.setActionCommand("green");
    mnuBleu=new JMenuItem("Bleu");
    mnuBleu.setActionCommand("blue");
    mnuCouleurs.add(mnuRouge);
    mnuCouleurs.add(mnuVert);
    mnuCouleurs.add(mnuBleu);
    // association de l'écouteur à chaque menu
    // (le même que pour les boutons)
    mnuRouge.addActionListener(ec);
    mnuVert.addActionListener(ec);
    mnuBleu.addActionListener(ec);
    // ajout du menu sur la fenêtre

```

```

        setJMenuBar(barreMenu);

        // création du conteneur intermédiaire
        pano=new JPanel();

        // ajout des boutons sur le conteneur intermédiaire
        pano.add(btnRouge);
        pano.add(btnVert);
        pano.add(btnBleu);

        // ajout du conteneur intermédiaire sur le ContentPane
        getContentPane().add(pano);

        // création d'une instance d'une classe anonyme
        // chargée de gérer les événements
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent arg0)
            {
                System.exit(0);
            }
        });
    }

    public class EcouteurCouleur implements ActionListener
    {
        public void actionPerformed(ActionEvent arg0)
        {
            String commande;
            commande=arg0.getActionCommand();
            if (commande.equals("red"))
            {
                pano.setBackground(Color.RED);
            }
            if (commande.equals("green"))
            {
                pano.setBackground(Color.GREEN);
            }
            if (commande.equals("blue"))
            {
                pano.setBackground(Color.BLUE);
            }
        }
    }

```

```
}  
  
}
```

À noter que dans cette solution, la déclaration des boutons et des éléments de menu peut être réintégrée dans le constructeur puisque nous n'en avons plus besoin au niveau de la classe.

La dernière étape de notre marathon sur les événements va nous permettre d'avoir plusieurs écouteurs pour une même source d'événements et éventuellement supprimer un écouteur existant. Pour cela, nous allons créer une nouvelle classe écouteur qui va nous permettre d'afficher sur la console la date et heure de l'événement et l'objet à l'origine de l'événement.

```
package jse.s4;  
  
import java.awt.event.ActionEvent;  
import java.awt.event.ActionListener;  
import java.text.SimpleDateFormat;  
import java.util.Date;  
  
import javax.swing.AbstractButton;  
import javax.swing.JButton;  
import javax.swing.JMenuItem;  
  
public class ConsoleLog implements ActionListener  
{  
    public void actionPerformed(ActionEvent e)  
    {  
        String message;  
        SimpleDateFormat sdf;  
        sdf=new SimpleDateFormat("dd/MM/yyyy hh:mm:ss");  
        message=sdf.format(new Date());  
        message=message + " clic sur le ";  
        if (e.getSource() instanceof JButton)  
        {  
            message=message+ "bouton ";  
        }  
        if (e.getSource() instanceof JMenuItem)  
        {  
            message=message+ "menu ";  
        }  
        message=message + ((AbstractButton)e.getSource()).getText();  
        System.out.println(message);  
    }  
}
```

```
}
```

Dans notre application, nous ajoutons ensuite une case à cocher permettant de choisir si les événements sont affichés sur la console. En fonction de l'état de cette case à cocher nous ajoutons, avec la méthode **addActionListener**, ou supprimons, avec la méthode **removeActionListener**, un écouteur aux boutons et menus. Ces deux méthodes attendent comme argument l'instance de l'écouteur à ajouter ou supprimer.

```
package jse.s4;

import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;

public class Ecran extends JFrame

{
    JPanel pano;
    JButton btnRouge, btnVert, btnBleu;
    JMenuItem mnuRouge, mnuVert, mnuBleu;
    ConsoleLog lg;
    public Ecran ()
    {
        setTitle("premiere fenetre en JAVA");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // création des trois boutons

        btnRouge=new JButton("Rouge");
        btnRouge.setActionCommand("red");
        btnVert=new JButton("Vert");
        btnVert.setActionCommand("green");
```



```

btnBleu=new JButton("Bleu");
btnBleu.setActionCommand("blue");
// création des trois écouteurs
EcouteurCouleur ec;
ec=new EcouteurCouleur();
// association de l'écouteur à chaque bouton
btnRouge.addActionListener(ec);
btnVert.addActionListener(ec);
btnBleu.addActionListener(ec);
// création de la case à cocher
JCheckBox chkLog;
chkLog=new JCheckBox("log sur console");
// ajout d'un écouteur à la case à cocher
chkLog.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        JCheckBox chk;
        chk=(JCheckBox) arg0.getSource();
        if (chk.isSelected())
        {
            // ajout d'un écouteur supplémentaire
            // aux boutons et menus
            lg=new ConsoleLog();
            btnBleu.addActionListener(lg);
            btnRouge.addActionListener(lg);
            btnVert.addActionListener(lg);
            mnuBleu.addActionListener(lg);
            mnuRouge.addActionListener(lg);
            mnuVert.addActionListener(lg);
        }
        else
        {
            // suppression de l'écouteur supplémentaire
            // des boutons et menus
            btnBleu.removeActionListener(lg);
            btnRouge.removeActionListener(lg);
            btnVert.removeActionListener(lg);
            mnuBleu.removeActionListener(lg);

```

```

        mnuRouge.removeActionListener(lg);
        mnuVert.removeActionListener(lg);
    }

}

});
// Création du menu
JMenuBar barreMenu;
barreMenu=new JMenuBar();
JMenu mnuCouleurs;
mnuCouleurs=new JMenu("Couleurs");
barreMenu.add(mnuCouleurs);
mnuRouge=new JMenuItem("Rouge");
mnuRouge.setActionCommand("red");
mnuVert=new JMenuItem("Vert");
mnuVert.setActionCommand("green");
mnuBleu=new JMenuItem("Bleu");
mnuBleu.setActionCommand("blue");
mnuCouleurs.add(mnuRouge);
mnuCouleurs.add(mnuVert);
mnuCouleurs.add(mnuBleu);
// association de l'écouteur à chaque menu
// (le même que pour les boutons)
mnuRouge.addActionListener(ec);
mnuVert.addActionListener(ec);
mnuBleu.addActionListener(ec);
// ajout du menu sur la fenêtre
setJMenuBar(barreMenu);
// création du conteneur intermédiaire
pano=new JPanel();
// ajout des boutons sur le conteneur intermédiaire
pano.add(btnRouge);
pano.add(btnVert);
pano.add(btnBleu);
pano.add(chkLog);
// ajout du conteneur intermédiaire sur le ContentPane
getContentPane().add(pano);
// création d'une instance d'une classe anonyme
// chargée de gérer les événements

```

```

        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent arg0)
            {
                System.exit(0);
            }
        });
    }

    public class EcouteurCouleur implements ActionListener
    {
        public void actionPerformed(ActionEvent arg0)
        {
            String commande;
            commande=arg0.getActionCommand();
            if (commande.equals("red"))
            {
                pano.setBackground(Color.RED);
            }
            if (commande.equals("green"))
            {
                pano.setBackground(Color.GREEN);
            }
            if (commande.equals("blue"))
            {
                pano.setBackground(Color.BLUE);
            }
        }
    }
}

```

À l'exécution et suivant l'état de la case à cocher, nous voyons apparaître les messages suivants sur la console.

```

24/01/2017 01:02:43 clic sur le bouton Rouge
24/01/2017 01:02:45 clic sur le bouton Vert
24/01/2017 01:02:47 clic sur le bouton Bleu
24/01/2017 01:02:51 clic sur le menu Rouge
24/01/2017 01:02:54 clic sur le menu Vert

```

Tous ces principes de gestion sont identiques quels que soient les événements et les objets les déclenchant. Le seul problème que vous pourrez rencontrer au début c'est de savoir quels sont les événements disponibles pour un objet particulier. À ce niveau, seule la documentation Java va pouvoir vous venir en aide. Une petite astuce consiste à chercher dans la classe correspondant à l'objet concerné, les méthodes nommées **addXXXXXListener** puis à partir de cette méthode remonter jusqu'à l'interface correspondante et découvrir, grâce aux méthodes définies dans l'interface, les différents événements possibles. L'analyse du type d'argument reçu dans les différentes méthodes vous permettra de savoir quelles informations supplémentaires sont disponibles pour un événement particulier. Par exemple, vous avez repéré la méthode **addMouseListener**.

addMouseListener

```
public void addMouseListener(MouseListener l)
```

Adds the specified mouse listener to receive mouse events from this component. If listener *l* is null, no exception is thrown and no action is performed.

Refer to [AWT Threading Issues](#) for details on AWT's threading model.

Parameters:

l - the mouse listener

Since:

JDK1.1

See Also:

[MouseEvent](#), [MouseListener](#), [removeMouseListener\(java.awt.event.MouseListener\)](#), [getMouseListeners\(\)](#)

Vous pouvez déjà déterminer que cette méthode va vous permettre de travailler avec des événements liés à la souris. Pour avoir plus d'informations sur les différents événements vous devez aller voir l'interface **MouseListener**.

Method Summary

void	mouseClicked(MouseEvent e) Invoked when the mouse button has been clicked (pressed and released) on a component.
void	mouseEntered(MouseEvent e) Invoked when the mouse enters a component.
void	mouseExited(MouseEvent e) Invoked when the mouse exits a component.
void	mousePressed(MouseEvent e) Invoked when a mouse button has been pressed on a component.
void	mouseReleased(MouseEvent e) Invoked when a mouse button has been released on a component.

La documentation de cette interface vous apprend que cinq méthodes sont prévues donc que cinq types d'événements sont disponibles et dans quelles circonstances ils sont déclenchés. Pour savoir quelles informations seront disponibles lors du déclenchement d'un de ces événements, vous poursuivez votre périple vers la classe correspondant au type des arguments reçus par ces méthodes.

Method Summary	
int	getButton() Returns which, if any, of the mouse buttons has changed state.
int	getClickCount() Returns the number of mouse clicks associated with this event.
Point	getLocationOnScreen() Returns the absolute x, y position of the event.
static String	getMouseModifiersText(int modifiers) Returns a String describing the modifier keys and mouse buttons that were down during the event, such as "Shift", or "Ctrl+Shift".
Point	getPoint() Returns the x,y position of the event relative to the source component.
int	getX() Returns the horizontal x position of the event relative to the source component.
int	getXOnScreen() Returns the absolute horizontal x position of the event.
int	getY() Returns the vertical y position of the event relative to the source component.
int	getYOnScreen() Returns the absolute vertical y position of the event.
boolean	isPopupTrigger() Returns whether or not this mouse event is the popup menu trigger event for the platform.
String	 paramString() Returns a parameter string identifying this event.
void	translatePoint(int x, int y) Translates the event's coordinates to a new position by adding specified x (horizontal) and y (vertical) offsets.

Vous pensez certainement que tout ceci est fastidieux mais vous allez rapidement vous rendre compte que ce sont pratiquement toujours les mêmes événements dont on a besoin et vous allez très rapidement les connaître par cœur.

2.3. Aspect des composants

Une application Java étant susceptible d'être exécutée sur n'importe quelle plate-forme, l'aspect des composants doit s'adapter à cette plate-forme. Java fournit une solution avec le mécanisme du look and feel. Chaque composant graphique est en fait constitué de deux classes. L'une gère la partie fonctionnelle du composant et la deuxième concerne uniquement l'aspect visuel du composant. Cette dernière est adaptée en fonction de la plate-forme sur laquelle s'exécute l'application. La classe **UIManager** est responsable du fonctionnement de ce mécanisme. Lorsqu'un composant graphique est créé, le constructeur du composant utilise la classe **UIManager** pour obtenir une instance de la classe chargée de gérer l'aspect graphique du composant. La classe **UIManager** doit donc être informée du look and feel qu'elle doit utiliser.

L'indication du look and feel devant être utilisé peut être effectuée de trois façons différentes :

- Par programmation : vous devez utiliser la méthode `setLookAndFeel` de la classe `UIManager` en fournissant comme paramètre le nom complet de la classe implémentant le look and feel.

```
UIManager.setLookAndFeel("javax.swing.plaf.nimbus.NimbusLookAndFeel");
```

Il est recommandé d'effectuer la modification du look and feel dès le début de l'application avant même la création du premier composant graphique.

- Par un paramètre de la ligne de commande utilisée pour lancer l'application.

```
java -Dswing.defaultlaf=javax.swing.plaf.nimbus.NimbusLookAndFeel applicationDemo
```

- Par modification d'un fichier de configuration.

Vous pouvez également utiliser un fichier de configuration **swing.properties** pour définir le look and feel par défaut qu'appliquera la machine virtuelle Java lors du lancement d'une application graphique. Ce fichier doit être placé dans le répertoire lib de la machine virtuelle. Il doit contenir la ligne suivante :

```
swing.defaultlaf=javax.swing.plaf.nimbus.NimbusLookAndFeel
```

Les exemples ci-dessus utilisent le look and feel nimbus disponible depuis la version 7 de Java. Celui-ci se démarque des autres look and feel par son mode de gestion graphique. Les versions précédentes utilisaient des images bitmap pour la représentation des composants. Cette nouvelle version utilise un mode graphique vectoriel qui procure un rendu plus précis que le mode bitmap.

2.4. Le positionnement des composants

Si vous avez déjà utilisé un autre langage de programmation permettant de développer des interfaces graphiques, une chose doit vous paraître étrange dans les exemples de code que nous avons utilisés.

- Nulle part, nous n'avons indiqué la taille et la position des composants utilisés.
- Si vous redimensionnez la fenêtre de l'application, les composants changent de place.

Et pourtant aucune ligne de code n'est prévue pour effectuer ce traitement. Ce petit miracle est lié à un concept très pratique de Java : les gestionnaires de mise en page (layout manager). En fait lorsque vous confiez des composants à un conteneur, celui-ci délègue à son gestionnaire de mise en page le soin d'organiser la disposition des composants sur sa surface.

De nombreux types de gestionnaire de mise en page sont disponibles, chacun d'eux ayant une stratégie différente pour le placement des composants. Chaque type de conteneur dispose d'un gestionnaire de mise en page par défaut, différent en fonction du type de conteneur. Si ce gestionnaire par défaut ne vous convient pas, vous êtes libre de le remplacer par un autre en appelant la méthode **setLayout** et en fournissant à cette méthode le nouveau gestionnaire à utiliser par ce conteneur.

Pour pouvoir travailler et organiser la disposition des composants, le gestionnaire de mise en page doit connaître la taille de chaque composant. La meilleure façon d'obtenir cette information est de s'adresser au composant lui-même. Pour cela, le gestionnaire de mise en page questionne chaque composant en appelant sa méthode **getPreferredSize**. Cette méthode calcule la taille du composant en fonction de son contenu, par exemple la longueur du libellé pour un bouton. Pour court-circuiter ce calcul, vous pouvez fixer une taille par défaut à chaque composant avec la méthode **setPreferredSize**.

La réussite de la conception d'une interface utilisateur nécessite donc de bien connaître comment fonctionnent les différents gestionnaires de mise en page. Nous allons donc étudier les caractéristiques des plus utilisés d'entre eux.

FlowLayout

Ce gestionnaire de mise en page est certainement le plus simple à utiliser. Il est associé par défaut à un composant **JPanel**. Sa stratégie de placement des composants consiste à les placer les uns à la suite des autres sur une ligne jusqu'à ce qu'il n'y ait plus de place sur cette ligne. Après le remplissage de la première ligne, les composants suivants sont placés sur une nouvelle ligne et ainsi

de suite. C'est l'ordre d'ajout des composants sur le conteneur qui détermine leurs positions sur les différentes lignes. Chaque composant est séparé horizontalement de son voisin par un espace de cinq pixels par défaut et chaque ligne de composants est séparée de sa voisine par un espace de cinq pixels par défaut également. Lorsque ce gestionnaire place les composants, il les aligne par défaut sur le centre du conteneur. Tous ces paramètres peuvent être modifiés pour chaque gestionnaire de mise en page **FlowLayout**. Vous pouvez le faire dès la création du **FlowLayout** en indiquant dans le constructeur les informations correspondantes. Trois constructeurs sont disponibles pour cette classe. Le premier n'attend aucun argument et crée un **FlowLayout** avec les caractéristiques par défaut décrites ci-dessus. Le deuxième attend comme argument une des constantes suivantes permettant de spécifier le type d'alignement utilisé.

- **FlowLayout.CENTER** : chaque ligne de composant est centrée dans le conteneur (valeur par défaut).
- **FlowLayout.LEFT** : chaque ligne de composant est alignée sur la gauche du conteneur.
- **FlowLayout.RIGHT** : chaque ligne de composant est alignée sur la droite du conteneur.

Le dernier constructeur disponible attend comme arguments deux entiers en plus de la constante indiquant l'alignement. Ces deux entiers indiquent l'espacement horizontal et vertical entre les composants. La ligne de code suivante crée un **FlowLayout** qui aligne les lignes de composants sur le centre du conteneur, laisse un espace horizontal de cinquante pixels et un espace vertical de vingt pixels entre les composants.

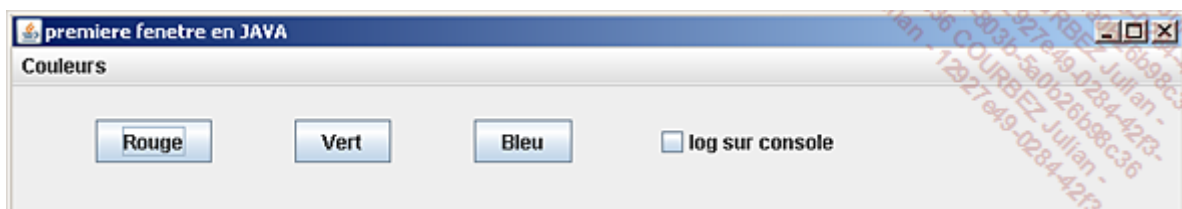
```
fl=new FlowLayout (FlowLayout.LEFT,50,20);
```

Si vous ne créez pas vous-même de **FlowLayout** mais que vous utilisez celui fourni par défaut avec un **Jpanel**, vous pouvez intervenir sur ces paramètres de fonctionnement avec les méthodes suivantes :

- **setAlignment** : pour spécifier l'alignement des composants.
- **setHgap** : pour spécifier l'espacement horizontal entre les composants.
- **setVgap** : pour spécifier l'espacement vertical entre les composants.

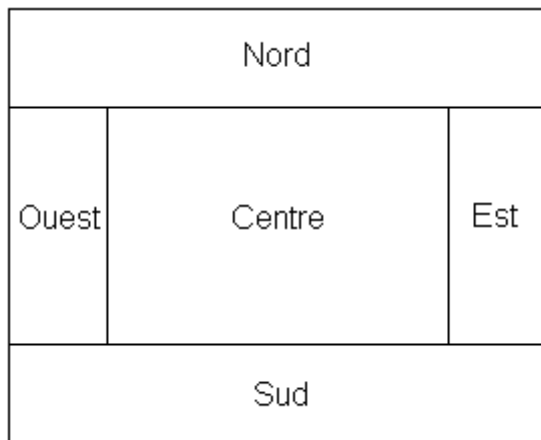
Il est dans ce cas nécessaire d'obtenir une référence sur le **FlowLayout** associé au **JPanel** en utilisant la méthode **getLayout** de celui-ci. Une opération de transtypage est dans ce cas obligatoire pour pouvoir utiliser les méthodes de la classe **FlowLayout** sur la référence obtenue.

```
((FlowLayout)pano.getLayout()).setAlignment (FlowLayout.LEFT);  
((FlowLayout)pano.getLayout()).setHgap (50);  
((FlowLayout)pano.getLayout()).setVgap (20);
```

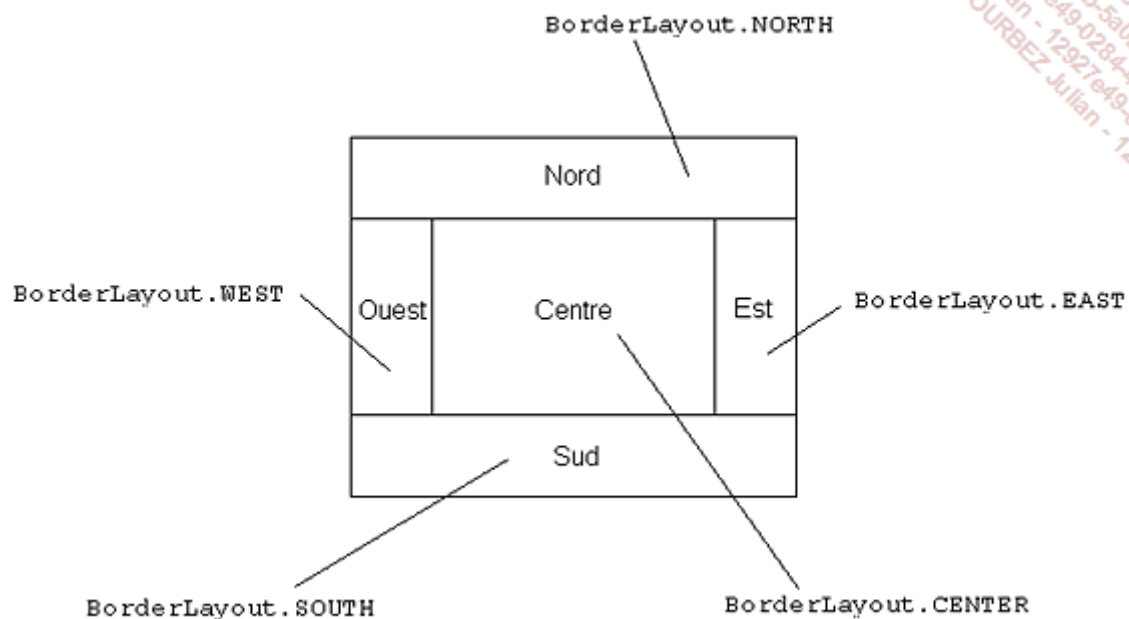


BorderLayout

Ce gestionnaire de mise en page découpe la surface qui lui est confiée en cinq zones suivant le schéma ci-dessous.



Lorsqu'un composant est confié à un **BorderLayout**, il convient d'indiquer dans quelle zone celui-ci doit être placé. Cette indication est fournie comme deuxième argument de la méthode **add**, le premier argument étant toujours le composant à insérer sur le conteneur. Les constantes suivantes sont disponibles pour identifier une zone du **BorderLayout**.



Si aucune information n'est indiquée pour identifier une zone lors de l'ajout d'un composant celui-ci est systématiquement placé dans la zone Centre.

Le **BorderLayout** n'aime pas le vide et de ce fait il redimensionne automatiquement tous les composants que vous lui confiez pour occuper tout l'espace disponible. Lorsque le conteneur est redimensionné, les composants placés en bordure ne changent pas de largeur pour les zones Ouest et Est, de hauteur pour les zones Nord et Sud. L'espace est donc gagné ou perdu par la zone Centre. Chaque zone du **BorderLayout** ne peut contenir qu'un seul composant. Si plusieurs composants

sont malgré tout ajoutés à une même zone d'un **BorderLayout**, seul le dernier ajouté sera pris en charge par le **BorderLayout**, les autres ne seront pas visibles.

Du fait de cette limitation, le **BorderLayout** est généralement utilisé pour positionner des conteneurs les uns par rapport aux autres plutôt que des composants isolés.

Le **BorderLayout** est le gestionnaire de mise en page par défaut de l'élément **ContentPane** d'une **JFrame**. L'utilisation classique consiste à placer dans la zone Nord la ou les barres d'outils, dans la zone Sud la barre d'état de l'application et dans la zone Centre le document sur lequel l'utilisateur doit travailler. Comme pour le **FlowLayout** il est possible d'indiquer au **BorderLayout** qu'il doit ménager un espace vertical ou horizontal entre ces différentes zones. Ceci peut être fait à la création du **BorderLayout** en indiquant l'espace horizontal et vertical dans l'appel du constructeur ou, si vous utilisez un **BorderLayout** existant, en appelant les méthodes **setHgap** et **setVgap** du **BorderLayout**. Pour illustrer tout cela, nous allons remanier l'interface de notre application en plaçant les boutons dans la zone Nord, la case à cocher dans la zone Sud et en modifiant la couleur de la zone Centre lors du clic sur les boutons ou menus.

```
package jse.s4;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;

public class Ecran9 extends JFrame

{
    JPanel panoBoutons;
    JPanel panoChk;
    JPanel panoCouleur;
    JButton btnRouge, btnVert, btnBleu;
    JMenuItem mnuRouge, mnuVert, mnuBleu;
    ConsoleLog lg;
```

```

public Ecran9()
{
    setTitle("première fenêtre en JAVA");
    setBounds(0,0,300,100);
    setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
    // création des trois boutons

    btnRouge=new JButton("Rouge");
    btnRouge.setActionCommand("red");
    btnVert=new JButton("Vert");
    btnVert.setActionCommand("green");
    btnBleu=new JButton("Bleu");
    btnBleu.setActionCommand("blue");
    // création des trois écouteurs
    EcouteurCouleur ec;
    ec=new EcouteurCouleur();
    // association de l'écouteur à chaque bouton
    btnRouge.addActionListener(ec);
    btnVert.addActionListener(ec);
    btnBleu.addActionListener(ec);
    // création de la case à cocher
    JCheckBox chkLog;
    chkLog=new JCheckBox("log sur console");
    // ajout d'un écouteur à la case à cocher
    chkLog.addActionListener(new ActionListener()

    public void actionPerformed(ActionEvent arg0)
    {
        JCheckBox chk;
        chk=(JCheckBox) arg0.getSource();
        if (chk.isSelected())
        {
            // ajout d'un écouteur supplémentaire
            // aux boutons et menus
            lg=new ConsoleLog();
            btnBleu.addActionListener(lg);
            btnRouge.addActionListener(lg);
            btnVert.addActionListener(lg);
            mnuBleu.addActionListener(lg);
        }
    }
}

```

```

        mnuRouge.addActionListener(lg);
        mnuVert.addActionListener(lg);
    }
    else
    {
        // suppression de l'écouteur supplémentaire
        // des boutons et menus
        btnBleu.removeActionListener(lg);
        btnRouge.removeActionListener(lg);
        btnVert.removeActionListener(lg);
        mnuBleu.removeActionListener(lg);
        mnuRouge.removeActionListener(lg);
        mnuVert.removeActionListener(lg);
    }

    }

});

// Création du menu
JMenuBar barreMenu;
barreMenu=new JMenuBar();

JMenu mnuCouleurs;
mnuCouleurs=new JMenu("Couleurs");
barreMenu.add(mnuCouleurs);

mnuRouge=new JMenuItem("Rouge");
mnuRouge.setActionCommand("red");
mnuVert=new JMenuItem("Vert");
mnuVert.setActionCommand("green");
mnuBleu=new JMenuItem("Bleu");
mnuBleu.setActionCommand("blue");
mnuCouleurs.add(mnuRouge);
mnuCouleurs.add(mnuVert);
mnuCouleurs.add(mnuBleu);

// association de l'écouteur à chaque menu
// (le même que pour les boutons)
mnuRouge.addActionListener(ec);
mnuVert.addActionListener(ec);
mnuBleu.addActionListener(ec);

// ajout du menu sur la fenêtre
setJMenuBar(barreMenu);

```

```

// création du conteneur intermédiaire
panoBoutons=new JPanel();
// ajout des boutons sur le conteneur intermédiaire
panoBoutons.add(btnRouge);
panoBoutons.add(btnVert);
panoBoutons.add(btnBleu);
// ajout du conteneur intermédiaire sur le ContentPane
// zone nord

getContentPane().add(panoBoutons,BorderLayout.NORTH);
// création du conteneur pour la case à cocher
panoChk=new JPanel();
panoChk.add(chkLog);
// ajout du conteneur dans la zone sud
getContentPane().add(panoChk,BorderLayout.SOUTH);
// création du conteneur pour affichage de la couleur
panoCouleur=new JPanel();
// ajout du conteneur dans la zone centre
getContentPane().add(panoCouleur,BorderLayout.CENTER);
// création d'une instance d'une classe anonyme
// chargée de gérer les événements
addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
});
((FlowLayout)panoBoutons.getLayout()).setAlignment
(FlowLayout.LEFT);
((FlowLayout)panoBoutons.getLayout()).setHgap(50);
((FlowLayout)panoBoutons.getLayout()).setVgap(20);
}

public class EcouteurCouleur implements ActionListener
{
    public void actionPerformed(ActionEvent arg0)
    {
        String commande;
        commande=arg0.getActionCommand();
    }
}

```

```
        if (commande.equals("red"))
        {
            panoCouleur.setBackground(Color.RED);
        }
        if (commande.equals("green"))
        {
            panoCouleur.setBackground(Color.GREEN);
        }
        if (commande.equals("blue"))
        {
            panoCouleur.setBackground(Color.BLUE);
        }
    }
}
}
```

Notre fenêtre présente maintenant la disposition suivante :



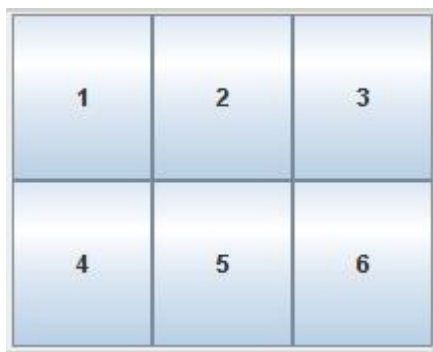
GridLayout

Ce gestionnaire de mise en page découpe la surface qui lui est confiée sous forme d'une grille invisible. Le nombre de lignes et de colonnes doit être indiqué dans l'appel du constructeur. Lorsque ce gestionnaire place les composants qu'on lui confie, il commence par remplir toutes les cases de la première ligne puis passe à la ligne suivante et ainsi de suite. Toutes les cases ont une taille identique et leurs contenus sont donc redimensionnés pour occuper tout l'espace disponible de chaque case. Par défaut les composants sont placés côte à côte. Un espace vertical et horizontal peut être inséré entre chaque composant en spécifiant deux arguments supplémentaires lors de l'appel du constructeur. Ces deux arguments indiquent l'espacement horizontal et l'espacement vertical. Ces espacements peuvent également être modifiés avec les méthodes `setHgap` et `setVgap`.

Si vous insérez plus de composants qu'il y a de cases dans le `GridLayout`, celui-ci crée des cases supplémentaires en augmentant le nombre de colonnes et en respectant le nombre de lignes d'origine. Par exemple avec le code suivant, nous prévoyons une grille de 2x2 cases mais nous ajoutons six composants.

```
JPanel grille;  
grille=new JPanel();  
GridLayout grl;  
grl=new GridLayout(2,2);  
grille.setLayout(grl);  
grille.add(new JButton("1"));  
grille.add(new JButton("2"));  
grille.add(new JButton("3"));  
grille.add(new JButton("4"));  
grille.add(new JButton("5"));  
grille.add(new JButton("6"));  
pano.add(grille);
```

Les composants sont disposés selon le schéma suivant :



Si vous souhaitez que le `GridLayout` respecte le nombre de colonnes, vous devez indiquer que le nombre de lignes vous est indifférent en spécifiant la valeur 0 pour cet argument dans le constructeur. Dans ce cas, le gestionnaire de mise en page ajoute des lignes supplémentaires pour les composants en surplus.

```

JPanel grille;

grille=new JPanel();

GridLayout grl;

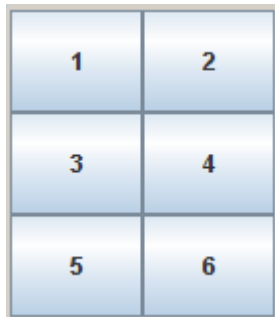
grl=new GridLayout(0,2);

grille.setLayout(grl);

grille.add(new JButton("1"));
grille.add(new JButton("2"));
grille.add(new JButton("3"));
grille.add(new JButton("4"));
grille.add(new JButton("5"));
grille.add(new JButton("6"));

pano.add(grille);

```



BoxLayout

Ce gestionnaire est utilisé lorsque l'on a besoin de placer des composants sur une ligne ou une colonne. Bien que l'on puisse obtenir le même résultat avec un **GridLayout** d'une seule ligne ou d'une seule colonne, ce gestionnaire procure plus de fonctionnalités. Le prix à payer pour ces fonctionnalités supplémentaires est une relative complexité d'utilisation. Le constructeur de la classe **BoxLayout** attend comme premier argument le conteneur dont il va gérer le contenu et comme deuxième argument une constante indiquant si ce conteneur gère une ligne (**BoxLayout.X_AXIS**) ou une colonne (**BoxLayout.Y_AXIS**).

Bien que l'on fournisse une référence sur le conteneur lors de la création de gestionnaire de mise en page, il est tout de même obligatoire d'associer le gestionnaire au conteneur avec la méthode **setLayout** comme nous l'avons déjà fait avec les autres gestionnaires.

Étudions maintenant la stratégie de fonctionnement de ce gestionnaire. Nous allons prendre le cas d'un gestionnaire horizontal.

- Le gestionnaire recherche le composant le plus haut.
- Il tente d'agrandir les autres composants jusqu'à cette hauteur.
- Si un composant ne peut pas atteindre cette hauteur, il est aligné verticalement sur les autres composants en fonction de la valeur renvoyée par la méthode **getAlignementY** du composant.
 - 0 pour un alignement sur le haut des autres composants.
 - 1 pour un alignement sur le bas des autres composants.
 - 0.5 pour un centrage sur les autres composants.

- Les largeurs préférées de chaque composant sont additionnées.
- Si la somme est inférieure à la largeur du conteneur, les composants sont agrandis jusqu'à leur largeur maximale.
- Si la somme est supérieure à la largeur du conteneur, les composants sont réduits jusqu'à leur largeur minimale. Si ce n'est pas suffisant, les derniers composants ne seront pas affichés.
- Les composants sont ensuite placés sur le conteneur de gauche à droite sans espace de séparation.

Le fonctionnement est tout à fait similaire pour un gestionnaire vertical.

```

JPanel ligne;
ligne=new JPanel();
BoxLayout bl;
bl=new BoxLayout(ligne,BoxLayout.X_AXIS);
ligne.setLayout(bl);
JButton b1,b2,b3,b4,b5;
// création d'un bouton avec alignement sur le haut
b1=new JButton("petit");
b1.setAlignmentY(0);
ligne.add(b1);
// création d'un bouton avec alignement sur le bas
b2=new JButton("    moyen    ");
b2.setAlignmentY(1);
ligne.add(b2);
// utilisation de html pour le libellé du bouton
b3=new JButton("<html>très<BR>haut</html>");
ligne.add(b3);
b4=new JButton("        très        large        ");
ligne.add(b4);
b5=new JButton("<html>très haut<br>et<br>très large</html>");
ligne.add(b5);
getContentPane().add(ligne);

```

Ce code nous donne l'interface utilisateur suivante.



Par défaut il n'y a pas d'espace entre les composants gérés par un **BoxLayout** et l'aspect résultant n'est très aéré. Pour pouvoir ajouter un espace entre les composants, vous devez insérer des composants de réservation d'espace. Trois types sont disponibles :

Strut : cet élément est utilisé pour ajouter un espace fixe entre deux composants. Un **Strut** peut être horizontal ou vertical suivant le type de **BoxLayout** auquel il est destiné. Ces deux types d'éléments sont créés par les méthodes

statiques **createVerticalStrut** et **createHorizontalStrut** de la classe **Box**. Elles attendent toutes les deux comme argument le nombre de pixels pour la largeur ou la hauteur.

```
// création d'un bouton avec alignement sur le haut
b1=new JButton("petit");
b1.setAlignmentY(0);
ligne.add(b1);
ligne.add(Box.createHorizontalStrut(10));
// création d'un bouton avec alignement sur le bas
b2=new JButton("    moyen    ");
b2.setAlignmentY(1);
ligne.add(b2);
```

RigidArea : cet élément a un fonctionnement similaire à celui d'un **Strut** en séparant les composants les uns des autres. Il force également une hauteur minimale pour le conteneur. Si un composant contenu dans le conteneur a une hauteur supérieure à celle-ci, c'est dans ce cas lui qui impose la hauteur du conteneur. Dans le cas contraire, c'est le composant le plus haut qui s'impose. Comme pour un **Strut**, c'est la méthode statique **createRigidArea** de la classe **Box** qui nous permet la création d'un **RigidArea**. Cette méthode attend comme argument un objet **Dimension** spécifiant l'espacement entre les composants et la hauteur minimale du conteneur.

```
// création d'un bouton avec alignement sur le haut
b1=new JButton("petit");
b1.setAlignmentY(0);
ligne.add(b1);
ligne.add(Box.createRigidArea(new Dimension(50,150)));
// création d'un bouton avec alignement sur le bas
b2=new JButton("    moyen    ");
b2.setAlignmentY(1);
ligne.add(b2);
```

Glue : le but de cet élément est toujours de séparer les composants mais cette fois l'espace entre les composants n'a pas une taille fixe. On peut comparer un **Glue** à un ressort que l'on place entre deux composants et qui les éloigne toujours le plus possible. Pour avoir un fonctionnement correct du **Glue**, les composants doivent avoir une taille fixe ou une taille maximale fixée. Comme les autres éléments de séparation, il est créé à partir de la méthode statique **createGlue** de la classe **Box**.

```

JPanel ligne;

    ligne=new JPanel();

    BoxLayout bl;

    bl=new BoxLayout(ligne,BoxLayout.X_AXIS);

    ligne.setLayout(bl);

    JButton b1,b2;

    // création d'un bouton avec alignement sur le haut

    b1=new JButton("petit");

    b1.setAlignmentY(0);

    b1.setMaximumSize(new Dimension(50,20));

    ligne.add(b1);

    ligne.add(Box.createGlue());

    // création d'un bouton avec alignement sur le bas

    b2=new JButton("    moyen    ");

    b2.setAlignmentY(1);

    b2.setMaximumSize(new Dimension(50,20));

    ligne.add(b2);

```

GridBagLayout

Vous pouvez considérer le **GridBagLayout** comme le "super" gestionnaire de mise en page tant du point de vue des performances que de la complexité d'utilisation. Il fonctionne à la base comme un **GridLayout** en plaçant les composants à l'intérieur d'une grille mais la comparaison s'arrête là. Dans un **GridBagLayout** les lignes et les colonnes ont des tailles variables, les cellules adjacentes peuvent être fusionnées pour accueillir les composants les plus grands. Les composants n'occupent pas obligatoirement toute la surface de leurs cellules et leurs positions à l'intérieur de la cellule peuvent être modifiées. La spécificité de ce gestionnaire se situe principalement dans la méthode d'ajout des composants. Cette méthode attend bien sûr comme argument le composant à ajouter mais également un objet **GridBagConstraints** indiquant la façon de positionner le composant dans le conteneur. C'est donc les caractéristiques de cet objet qui vont permettre le positionnement du composant par le **GridBagLayout**. Regardons donc les principales caractéristiques de cet objet :

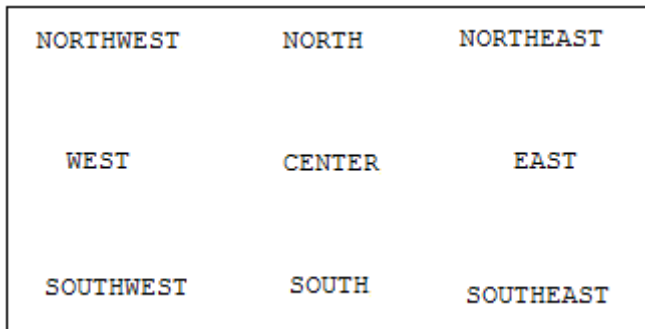
- **gridx** : colonne du coin supérieur gauche du composant.
- **gridy** : ligne du coin supérieur gauche du composant.

Pour ces deux propriétés la numérotation commence à zéro.

- **gridwidth** : nombre de colonnes occupées par le composant.
- **gridheight** : nombre de lignes occupées par le composant.
- **weightx** : indique comment est réparti l'espace supplémentaire disponible en largeur lorsque le conteneur est redimensionné. La répartition est faite au prorata de cette valeur. Si cette valeur est égale à zéro le composant n'est pas redimensionné. Si cette valeur est identique pour tous les composants l'espace est réparti équitablement.
- **weighty** : cette propriété joue le même rôle que la précédente mais sur l'axe vertical.
- **fill** : cette propriété est utilisée lorsque la zone allouée à un composant est supérieure à sa taille. Elle détermine si le composant est redimensionné et comment.

Les constantes suivantes sont possibles :

- **NONE** : le composant n'est pas redimensionné.
- **HORIZONTAL** : le composant est redimensionné en largeur et sa hauteur reste inchangée.
- **VERTICAL** : le composant est redimensionné en hauteur et sa largeur reste inchangée.
- **BOTH** : le composant est redimensionné en largeur et en hauteur pour remplir complètement la surface disponible.
- **anchor** : indique comment est positionné le composant dans l'espace disponible s'il n'est pas redimensionné. Les constantes suivantes sont disponibles.



Un même objet **GridBagConstraints** peut être utilisé pour placer plusieurs composants sur un **GridBagLayout**. Il faut bien penser dans ce cas à initialiser correctement les différents champs lors de l'ajout de chaque composant.

À titre d'exemple voici l'interface d'une application simple et le code correspondant.



```
package jse.s4;

import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
```

```

import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;

public class Ecran extends JFrame

{
    JPanel pano;
    JCheckBox chkGras,chkItalique;
    JLabel lblTaille,lblExemple;
    JComboBox cboTaille;
    JList lstPolices;
    JScrollPane defilPolices;

    public Ecran()
    {
        setTitle("choix d\'une police");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // création des composants

        pano=new JPanel();
        chkGras=new JCheckBox("gras");
        chkItalique=new JCheckBox("italique");
        lblTaille=new JLabel("taille");
        lblExemple=new JLabel("essai de police de caractères");
        cboTaille=new JComboBox(new String[]
{"10","12","14","16","18","20"});
        lstPolices=new JList(new String[]{"arial","courier",
"letter","helvetica","times roman", "symbole","antique"});
        defilPolices=new JScrollPane(lstPolices);

        GridBagLayout gbl;
        gbl=new GridBagLayout();
        pano.setLayout(gbl);

        GridBagConstraints gbc;
        gbc=new GridBagConstraints();
        // position dans la case 0,0

```

```
gbc.gridy=0;
// sur une colonne de largeur
gbc.gridwidth=1;
// et sur trois lignes en hauteur
gbc.gridheight=3;
// pondération en cas d'agrandissement du conteneur
gbc.weightx=100;
gbc.weighty=100;
// le composant est redimensionné pour occuper
// tout l'espace disponible dans son conteneur
gbc.fill=GridBagConstraints.BOTH;
pano.add(defilPolices,gbc);
// position dans la case 1,0
gbc.gridx=1;
gbc.gridy=0;
// sur deux colonnes de largeur
gbc.gridwidth=2;
// et sur une ligne en hauteur
gbc.gridheight=1;
// pondération en cas d'agrandissement du conteneur
gbc.weightx=100;
gbc.weighty=100;
// le composant n'est pas redimensionné pour occuper
// tout l'espace disponible dans son conteneur
gbc.fill=GridBagConstraints.NONE;
pano.add(chkGras,gbc);
// position dans la case 1,1
gbc.gridx=1;
gbc.gridy=1;
// sur deux colonnes de largeur
gbc.gridwidth=2;
// et sur une ligne en hauteur
gbc.gridheight=1;
// pondération en cas d'agrandissement du conteneur
gbc.weightx=100;
gbc.weighty=100;
pano.add(chkItalique,gbc);
// position dans la case 1,2
gbc.gridx=1;
```

```

        gbc.gridy=2;
        // sur une colonne de largeur
        gbc.gridwidth=1;
        // et sur une ligne en hauteur
        gbc.gridheight=1;
        // pondération en cas d'agrandissement du conteneur
        gbc.weightx=100;
        gbc.weighty=100;
        pano.add(lblTaille,gbc);
        // position dans la case 2,2
        gbc.gridx=2;
        gbc.gridy=2;
        // sur une colonne de largeur
        gbc.gridwidth=1;
        // et sur une ligne en hauteur
        gbc.gridheight=1;
        // pondération en cas d'agrandissement du conteneur
        gbc.weightx=100;
        gbc.weighty=100;
        pano.add(cboTaille,gbc);
        // position dans la case 0,3
        gbc.gridx=0;
        gbc.gridy=3;
        // sur trois colonnes de largeur
        gbc.gridwidth=3;
        // et sur une ligne en hauteur
        gbc.gridheight=1;
        // pondération en cas d'agrandissement du conteneur
        gbc.weightx=100;
        gbc.weighty=100;
        pano.add(lblExemple,gbc); x
        getContentPane().add(pano);

    }
}

```

Sans gestionnaire de mise en page

Si vous n'êtes pas convaincu de l'utilité d'un gestionnaire de mise en page, vous avez la possibilité de gérer vous-même la taille et position des composants de l'interface utilisateur. Vous devez d'abord indiquer que vous renoncez à utiliser un gestionnaire de mise en forme en appelant la méthode **setLayout** du conteneur et en lui passant la valeur **null**. Vous pouvez ensuite ajouter les composants au conteneur avec la méthode **add** de celui-ci. Et finalement vous devez positionner et

dimensionner les composants. Ces deux opérations peuvent être effectuées en deux étapes avec les méthodes **setLocation** et **setSize** ou en une seule étape avec la méthode **setBounds**.

```
package jse.s4;

import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;

public class Ecran extends JFrame

{
    JPanel pano;
    JCheckBox chkGras,chkItalique;
    JLabel lblTaille,lblExemple;
    JComboBox cboTaille;
    JList lstPolices;
    JScrollPane defilPolices;

    public Ecran()
    {
        setTitle("choix d'une police");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // création des composants
        pano=new JPanel();
        chkGras=new JCheckBox("gras");
        chkItalique=new JCheckBox("italique");
        lblTaille=new JLabel("taille");
        lblExemple=new JLabel("essai de police de caractères");
        cboTaille=new JComboBox(new String[]{"10","12","14","16","18","20"});
        lstPolices=new JList(new
        String[]{"arial","courier","letter","helvetica","times roman","symbole",
        "antique"});
```

```

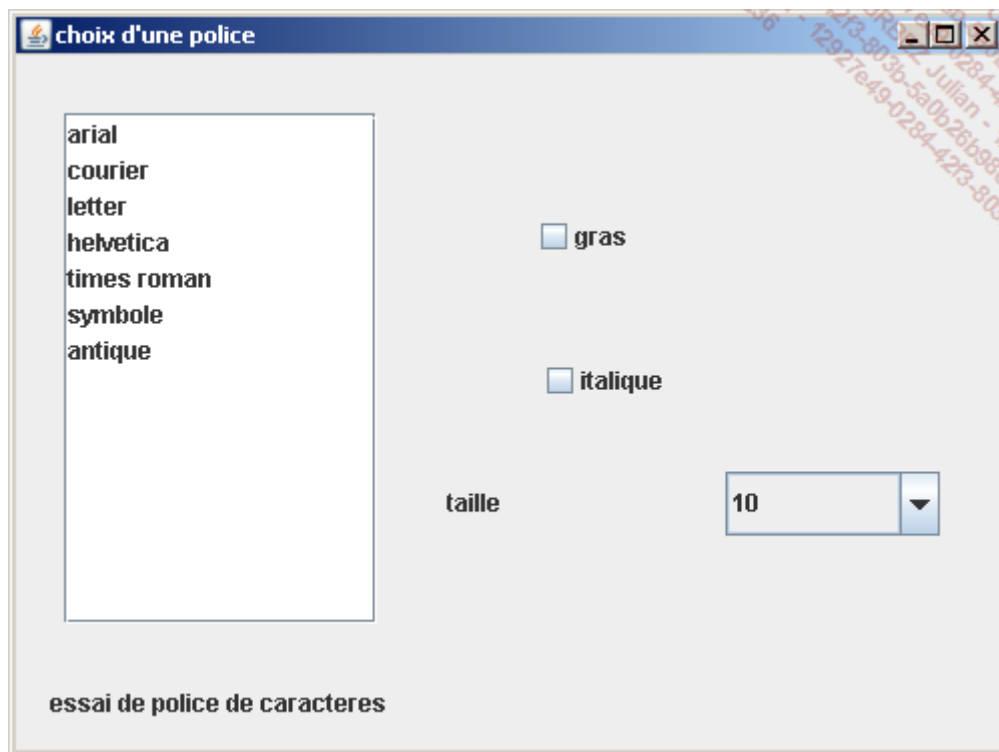
defilPolices=new JScrollPane(lstPolices);

// ajout sur le conteneur
pano.setLayout(null);
pano.add(defilPolices);
pano.add(chkGras);
pano.add(chkItalique);
pano.add(lblTaille);
pano.add(cboTaille);
pano.add(lblExemple);

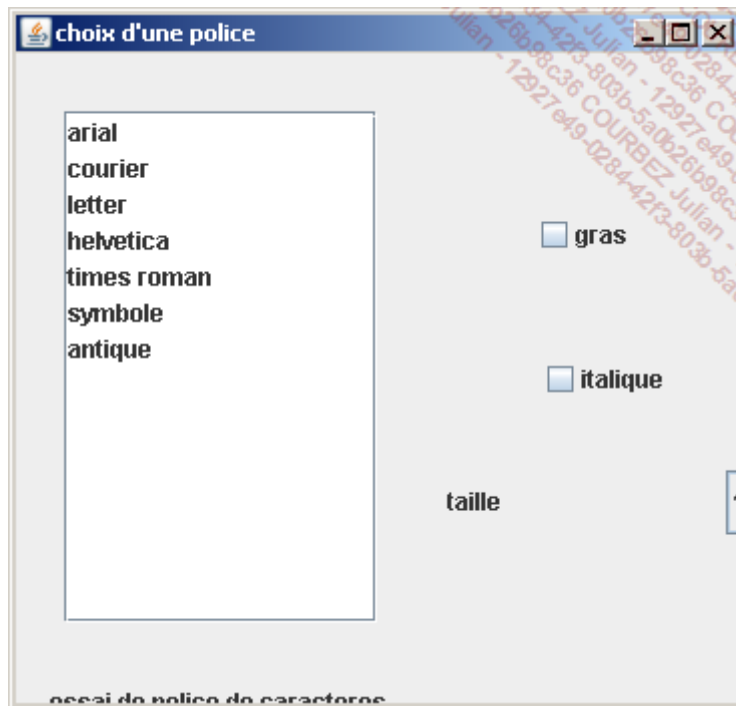
// position des composants.
defilPolices.setBounds(24, 29, 156, 255);
chkGras.setBounds(258, 78, 170, 25);
chkItalique.setBounds(261, 139, 167, 46);
lblTaille.setBounds(215, 211, 106, 24);
cboTaille.setBounds(354, 208, 107, 32);
lblExemple.setBounds(17, 309, 459, 28);
getContentPane().add(pano);

```

Notre application présente bien le même aspect que si elle utilisait un gestionnaire de mise en page.



Par contre si la fenêtre est redimensionnée, les composants conservent leurs tailles et positions et l'interface peut alors prendre un aspect incohérent.



2.5. Les composants graphiques

Chaque composant utilisable avec Java est représenté par une classe dont nous allons pouvoir créer des instances pour concevoir l'interface de l'application. La majorité des composants **Swing** dérivent de la classe **JComponent** et de ce fait, héritent d'un bon nombre de propriétés et de méthodes de cette classe ou des classes situées au-dessus dans la hiérarchie d'héritage.

`javax.swing`

Class JComponent

```

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

La classe JComponent

Dimensions et position

Une multitude de méthodes interviennent dans la gestion de la position et de la taille des composants. Deux catégories d'informations sont disponibles pour la taille des composants :

- La taille actuelle est accessible par la méthode **getSize**. Cette dernière obtient la taille du composant au moment de l'appel de la méthode. Si le composant est pris en charge par un gestionnaire de mise en page, celui-ci est susceptible de modifier la taille du composant et donc l'appel de cette méthode peut renvoyer un résultat différent à chaque appel. La taille peut aussi être modifiée par la méthode **setSize**. Les effets de cette méthode peuvent être de courte durée car, si le composant est pris en charge par un gestionnaire de mise en page, celui-ci peut à tout moment redessiner le composant avec une taille différente.
- La taille préférée est accessible par la méthode **getPreferredSize**. Par défaut cette taille est calculée en fonction du contenu du composant. Cette méthode est surtout utilisée en interne par les gestionnaires de mise en page. Pour éviter que la taille du composant soit

recalculée en fonction de son contenu, vous pouvez utiliser la méthode `setPreferredSize`. Les informations fournies par cette méthode sont ensuite utilisées par la méthode `getPreferredSize` lorsque le conteneur a besoin d'informations sur la taille du composant.

La position d'un composant à l'intérieur de son conteneur peut être obtenue ou définie avec les méthodes `getLocation` ou `setLocation`. Comme pour la méthode `setSize` l'effet de la méthode `setLocation` peut être à tout moment annulé par le gestionnaire de mise en page, si celui-ci doit redessiner les composants.

Les méthodes `getBounds` et `setBounds` permettent de combiner la modification de la taille et de la position du composant. Ces méthodes n'ont vraiment d'intérêt que si vous assumez complètement la présentation des composants à l'intérieur du conteneur sans passer par les services d'un gestionnaire de mise en page.

Apparence des composants

La couleur de fond du composant peut être modifiée par la méthode `setBackground` tandis que la couleur du texte du composant est modifiée par la propriété `setForeground`. Ces méthodes attendent comme argument un objet `Color`. Cet objet `Color` peut être obtenu par les constantes définies dans la classe `Color`.

```
lstPolices.setBackground(Color.red);  
lstPolices.setForeground(Color.GREEN);
```

La classe `Color` propose également de nombreux constructeurs permettant d'obtenir une couleur particulière en effectuant un mélange des couleurs de base.

```
lstPolices.setBackground(new Color(23,67,89));  
lstPolices.setForeground(new Color(167,86,23));
```

La police est modifiable par la méthode `setFont` du composant. On peut, pour l'occasion, créer une nouvelle instance de la classe `Font` et l'affecter au composant. Pour cela, nous utiliserons un des constructeurs de la classe `Font` en indiquant le nom de la police, le style de la police et sa taille.

```
lstPolices.setFont(new Font("Serif",Font.BOLD,24));
```

Ces deux propriétés s'appliquent sur l'ensemble du texte affiché sur le composant. Il est possible d'utiliser plusieurs couleurs et polices en formatant le texte du composant à l'aide de balises html. Il suffit simplement d'encadrer le texte du composant entre des balises `<html>` et `</html>`. À l'intérieur de ces balises, vous pouvez utiliser ensuite n'importe quelle balise de formatage html. L'exemple suivant modifie la couleur du texte et affiche en italique (sauf la première lettre) le libellé du bouton.

```
// création des trois boutons  
JButton b1,b2,b3;  
b1=new JButton("<html><font color=red>R<i>ouge</i></font>  
</html>");  
b2=new JButton("<html><font color=green>V<i>ert</i></font>  
</html>");  
b3=new JButton("<html><font color=blue>B<i>leu</i></font>  
</html>");
```

La méthode **setCursor** permet de choisir l'apparence du curseur lorsque la souris se trouve sur la surface du composant. Plusieurs curseurs sont prédéfinis et sont accessibles en créant une instance de la classe **Cursor** avec une des constantes prédéfinies.

```
lstPolices.setCursor(new Cursor(Cursor.HAND_CURSOR));
```

La détection de l'entrée et de la sortie de la souris sur le composant et la modification du curseur en conséquence est gérée automatiquement par le composant lui-même.

Comportement des composants

Les composants placés sur un conteneur peuvent être masqués en appelant la méthode **setVisible** ou désactivés en appelant la méthode **setEnabled**. Dans ce cas, le composant est toujours visible mais apparaît avec un aspect grisé pour indiquer à l'utilisateur que ce composant est inactif pour le moment.

```
chkGras.setVisible(false);
```

```
chkItalique.setEnabled(false);
```

Les composants dans cet état ne peuvent bien sûr pas recevoir le focus dans l'application. Vous pouvez vérifier cela en appelant la méthode **isFocusable** qui renvoie un boolean. Vous pouvez également vérifier si un composant détient actuellement le focus, en utilisant la méthode **isFocusOwner**. Le focus peut être placé sur un composant sans l'intervention de l'utilisateur, en appelant la méthode **requestFocus** du composant.

Pour surveiller le passage du focus d'un composant à l'autre, deux événements sont à votre disposition :

- **focusGained** indique qu'un composant particulier a reçu le focus.
- **focusLost** indique qu'un composant a perdu le focus.

Par exemple, pour bien visualiser qu'un composant détient le focus, on peut utiliser le code suivant qui modifie la couleur du texte lorsque le composant reçoit ou perd le focus :

```
lstPolices.addFocusListener(new FocusListener()
{
    public void focusGained(FocusEvent arg0)
    {
        lstPolices.setForeground(Color.RED);
    }

    public void focusLost(FocusEvent arg0)
    {
        lstPolices.setForeground(Color.BLACK);
    }
});
```

Affichage d'information

Le composant JLabel

Le composant **JLabel** est utilisé pour afficher sur un formulaire, un texte qui ne sera pas modifiable par l'utilisateur. Il sert essentiellement à fournir une légende à des contrôles qui n'en possèdent pas (zones de texte par exemple, liste déroulante...). Dans ce cas, il permettra également de fournir un raccourci-clavier pour atteindre ce composant.

Le texte affiché par le composant est indiqué lors de sa création ou par la méthode **setText**. Par défaut, le composant **JLabel** n'a pas de bordure. Vous pouvez en ajouter une en appelant la méthode **setBorder** en lui passant la bordure à utiliser.

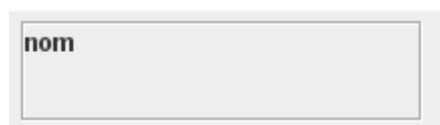
```
JLabel legende;  
legende=new JLabel("nom");  
legende.setBorder(BorderFactory.createEtchedBorder());
```

Vous avez aussi la possibilité d'indiquer la position du texte dans le composant par l'intermédiaire des méthodes **setHorizontalAlignment** et **setVerticalAlignment** en spécifiant une des constantes prédéfinies.

```
SwingConstants.RIGHT : alignement sur la droite  
SwingConstants.CENTER : alignement sur le centre  
SwingConstants.LEFT : alignement sur la gauche  
SwingConstants.TOP : alignement sur le haut  
SwingConstants.BOTTOM : alignement sur le bas
```

Ces méthodes n'ont aucun effet visible si le composant est dimensionné automatiquement par son conteneur puisque dans ce cas sa taille est automatiquement ajustée à son contenu. Pour que ces méthodes aient un effet visible, vous devez indiquer une taille préférée pour le composant.

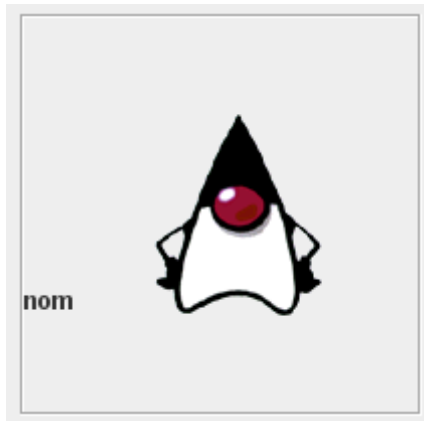
```
legende=new JLabel("nom");  
legende.setBorder(BorderFactory.createEtchedBorder());  
legende.setPreferredSize(new Dimension(200,50));  
legende.setHorizontalAlignment(SwingConstants.LEFT);  
legende.setVerticalAlignment(SwingConstants.TOP);
```



Les composants **JLabel** peuvent également afficher des images. Vous pouvez indiquer l'image à afficher à l'aide de la méthode **setIcon**. L'image doit bien sûr être créée au préalable. Si du texte est également affiché sur le composant, sa position peut être spécifiée par rapport à l'image en utilisant les méthodes **setHorizontalTextPosition** et **setVerticalTextPosition**, ainsi que l'espace entre le texte et l'image avec la méthode **setIconTextGap**.

```
JLabel legende;  
legende=new JLabel("nom");  
legende.setBorder(BorderFactory.createEtchedBorder());
```

```
legende.setPreferredSize(new Dimension(200,200));
legende.setIcon(new ImageIcon("duke.gif"));
legende.setHorizontalTextPosition(SwingConstants.LEFT);
legende.setVerticalTextPosition(SwingConstants.BOTTOM);
legende.setIconTextGap(40);
```



Nous avons indiqué que le composant **JLabel** pouvait être utilisé comme raccourci-clavier pour un autre composant. Pour cela, deux précautions sont à prendre.

Vous devez indiquer, avec la méthode **setDisplayMnemonic**, le caractère utilisé comme raccourci-clavier.

Vous devez également indiquer au **JLabel** pour quel composant il joue le rôle de légende en utilisant la méthode **setLabelFor**.

```
legende.setDisplayedMnemonic('n');
legende.setLabelFor(lstPolices);
```

Le composant JProgressBar

Ce composant est utilisé pour informer l'utilisateur sur la progression d'une action lancée dans l'application. Il affiche cette information sous la forme d'une zone rectangulaire, qui sera plus ou moins remplie en fonction de l'état d'avancement de l'action exécutée.

La position de la barre de progression est contrôlée par la méthode **setValue**. Cette méthode accepte un argument pouvant évoluer entre les deux extrêmes indiqués par les méthodes **setMinimum** et **setMaximum**.

Une légende peut également être affichée en incrustation sur la barre de progression. Le texte de la légende est indiqué par la méthode **setString** et son affichage est contrôlé par la méthode **setStringPainted**.

S'il est impossible d'évaluer la progression d'une opération et donc d'obtenir une valeur à fournir à la barre de progression, celle-ci peut être placée dans un état indéterminé avec la méthode **setIndeterminate**. Un curseur se déplaçant en permanence entre les deux extrêmes est dans ce cas affiché.

L'exemple suivant présente une horloge originale où l'heure est affichée par trois **JProgressBar** :

```
package jse.s4;
```

```
import java.util.GregorianCalendar;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JProgressBar;

public class Ecran13 extends JFrame

{
    JPanel pano;
    JProgressBar pgbHeure,pgbMinutes,pgbSeconde,pgbDefil;

    public Ecran13()
    {
        setTitle("horloge");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // création des composants
        pgbHeure =new JProgressBar();
        pgbMinutes=new JProgressBar();
        pgbSeconde=new JProgressBar();
        pgbDefil=new JProgressBar();
        pgbHeure.setMinimum(0);
        pgbHeure.setMaximum(23);
        pgbMinutes.setMinimum(0);
        pgbMinutes.setMaximum(59);
        pgbSeconde.setMinimum(0);
        pgbSeconde.setMaximum(59);
        pgbHeure.setString("heure");
        pgbHeure.setStringPainted(true);
        pgbMinutes.setString("minute");
        pgbMinutes.setStringPainted(true);
        pgbSeconde.setString("seconde");
        pgbSeconde.setStringPainted(true);
        pgbDefil.setString("le temps passe");
        pgbDefil.setStringPainted(true);
        pgbDefil.setIndeterminate(true);
        pano=new JPanel();
```

```

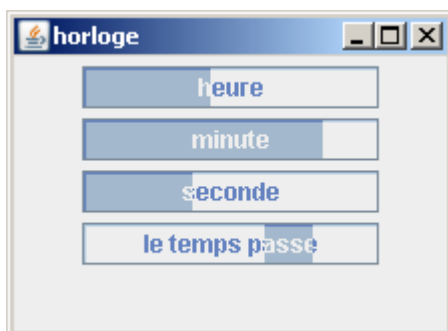
        pano.add(pgbHeure);
        pano.add(pgbMinutes);
        pano.add(pgbSeconde);
        pano.add(pgbDefil);
        getContentPane().add(pano);

        Thread th;
        th=new Thread()
        {
            public void run()
            {
                while (true)
                {
                    LocalTime d;
                    d=LocalTime.now();
                    pgbHeure.setValue(d.getHour());
                    pgbMinutes.setValue(d.getMinute());
                    pgbSeconde.setValue(d.getSecond());

                    try
                    {
                        sleep(500);
                    }
                    catch (InterruptedException e)
                    {
                    }
                }
            }
        };

        th.start();
    }
}

```



Les composants d'édition de texte

JTextField

Le composant **JTextField** est utilisé pour permettre à l'utilisateur de saisir des informations. Ce composant ne permet que la saisie de texte sur une seule ligne. Ce composant est également capable

de gérer la sélection de texte et les opérations avec le Presse-papiers. De nombreuses méthodes sont disponibles pour travailler avec ce composant. Par défaut, ce composant adapte automatiquement sa taille à son contenu ce qui peut provoquer un réaffichage permanent du composant. Pour pallier ce problème, il est préférable de spécifier une taille pour le composant soit en utilisant sa méthode **setPreferredSize** soit en indiquant le nombre de colonnes désiré pour l’affichage du texte dans ce composant. Ceci ne limite absolument pas le nombre de caractères pouvant être saisis mais uniquement la largeur du composant. Le texte affiché dans le composant peut être modifié ou récupéré par les méthodes **setText** ou **getText**.

La gestion de la sélection du texte se fait automatiquement par le composant. La méthode **getSelectedText** permet de récupérer la chaîne de caractères actuellement sélectionnée dans le contrôle. Les méthodes **getSelectionStart** et **getSelectionEnd** indiquent respectivement le caractère de début de la sélection (le premier caractère a l’indice 0) et le dernier caractère de la sélection.

La sélection de texte peut également s’effectuer avec la méthode **select**, en indiquant le caractère de début de la sélection et le caractère de fin la sélection.

La sélection de la totalité du texte peut être effectuée avec la méthode **selectAll**. Par exemple, on peut forcer la sélection de tout le texte lorsque le composant reçoit le focus.

```
txt=new JTextField(10);
txt.addFocusListener(new FocusListener()
{
    public void focusGained(FocusEvent e)
    {
        txt.selectAll();
    }
    public void focusLost(FocusEvent e)
    {
        txt.setSelectionStart(0);
        txt.setSelectionEnd(0);
    }
});
```

Pour la gestion du Presse-papiers, le composant **JTextField** gère automatiquement les raccourcis-clavier du système pour les opérations de copier, couper, coller. Vous avez cependant la possibilité d’appeler les méthodes **copy**, **cut** et **paste** pour gérer les opérations de copier, couper, coller d’une autre manière, par exemple par un menu de l’application ou un menu contextuel comme dans l’exemple ci-dessous :

```
txt=new JTextField(10);
mnu=new JPopupMenu();
JMenuItem mnuCopier;
JMenuItem mnuCouper;
JMenuItem mnuColler;
```



```
mnuCopier=new JMenuItem("Copier");
mnuCouper=new JMenuItem("Couper");
mnuColler=new JMenuItem("Coller");
mnuCopier.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.copy();
    }
});
mnuCouper.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.cut();
    }
});
mnuColler.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.paste();
    }
});
mnu.add(mnuCopier);
mnu.add(mnuCouper);
mnu.add(mnuColler);
txt.addMouseListener(new MouseAdapter()
{
    public void mousePressed(MouseEvent e)
    {
        if (e.getButton()==MouseEvent.BUTTON3)
        {
            mnu.show((Component)e.getSource(), e.getX(),e.getY());
        }
    }
});
```

```
}  
);  
  
}
```

Les opérations couper et coller ne seront cependant pas possibles si le composant **JTextField** est configuré en lecture seule avec la méthode **setEditable(false)**, la modification du texte par l'utilisateur est bien sûr dans ce cas également impossible.

Tout le monde ayant droit à l'erreur, il est possible de provoquer l'annulation de la dernière modification de texte effectuée sur le contrôle. Pour cela, nous devons nous adjoindre l'aide d'un objet **UndoManager**. C'est ce dernier qui prend effectivement en charge l'annulation de la dernière modification grâce à sa méthode `undo`. Cette méthode peut être appelée par une option du menu contextuel ou par le raccourci-clavier [Ctrl] **Z**. Il n'y a qu'un seul niveau de "Undo", vous ne pourrez revenir au texte que vous avez saisi, il y a deux heures !

```
// création du UndoManager  
UndoManager udm;  
  
udm=new UndoManager();  
  
// association avec le JTextField  
txt.getDocument().addUndoableEditListener(udm);  
  
// ajout d'un écouteur pour intercepter le ctrl Z  
txt.addKeyListener(new KeyAdapter()  
{  
    public void keyPressed(KeyEvent e)  
    {  
        if (e.getKeyChar()=='z' & e.isControlDown())  
        {  
            udm.undo();  
        }  
    }  
});
```

JPasswordField

Ce composant est à la base un **JTextField** normal légèrement modifié pour être spécialisé dans la saisie de mot de passe. Les deux seuls ajouts réellement utiles sont la méthode **setEchoChar** permettant d'indiquer le caractère utilisé comme caractère de substitution lors de l'affichage et la méthode **getPassword** permettant d'obtenir le mot de passe saisi par l'utilisateur.

JTextArea

Ce composant offre des fonctionnalités plus évoluées qu'un simple **JTextField**. La première amélioration importante apportée par ce composant réside dans sa capacité à gérer la saisie sur plusieurs lignes. C'est d'ailleurs pour cette raison que lors de la construction d'un objet d'un objet **JTextArea** nous devons spécifier le nombre de lignes et le nombre de colonnes qu'il va comporter. Ces deux informations sont utilisées uniquement pour le dimensionnement du composant et n'influencent pas la quantité de texte que peut contenir le composant. Par contre, ce

composant ne gère pas lui-même le défilement du texte qu'on lui confie. Pour lui adjoindre cette fonctionnalité, celui-ci doit être placé sur un conteneur de type `JScrollPane` qui va prendre en charge le défilement du contenu du `JTextArea`.

Si cette solution n'est pas utilisée, le composant `JTextArea` peut être configuré pour scinder automatiquement une ligne lors de son affichage. La méthode `setLineWrap` permet d'activer ce mode de fonctionnement. Par défaut le découpage se produit sur le dernier caractère visible d'une ligne avec le risque de voir certains mots affichés sur deux lignes. Pour éviter ce problème, le composant `JTextArea` peut être configuré pour effectuer son découpage sur les espaces séparant les mots avec la méthode `setWrapStyleWord`. Les sauts de ligne ajoutés par le composant ne font pas partie du texte et sont utilisés uniquement pour l'affichage de celui-ci.

```
package fr.eni;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

public class Ecran extends JFrame
{
    JPanel pano;
    JTextArea txt;
    JCheckBox chkWrap, chkWrapWord;
    JScrollPane defil;

    public Ecran()
    {
        setTitle("éditeur de texte");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // création des composants
        pano=new JPanel();
        txt=new JTextArea(10,40);
        defil=new JScrollPane(txt);
        pano.add(defil);

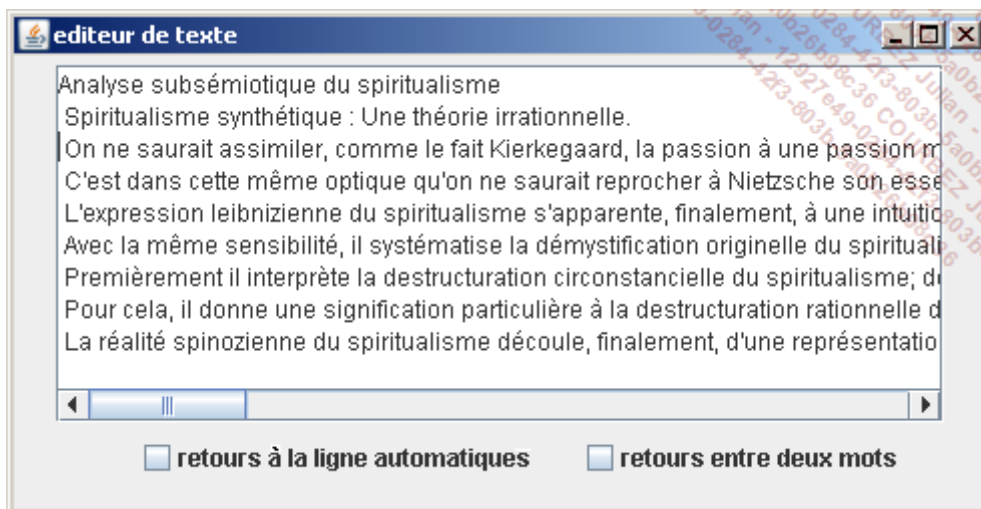
        chkWrap=new JCheckBox("retours à la ligne automatiques");
        chkWrapWord=new JCheckBox("retours entre deux mots");
        chkWrap.addActionListener(new ActionListener()
```

```

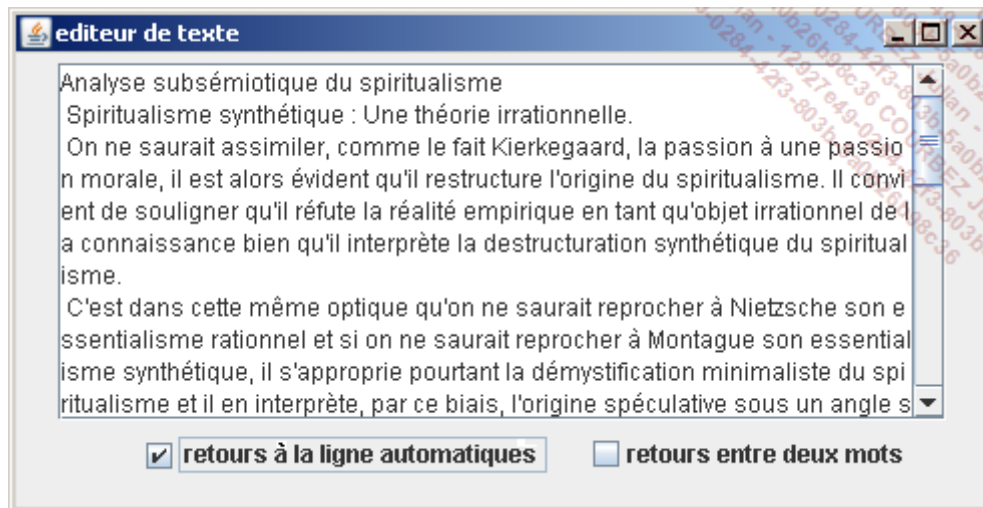
{
    public void actionPerformed(ActionEvent e)
    {
        txt.setLineWrap(chkWrap.isSelected());
    }
});
chkWrapWord.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.setWrapStyleWord(chkWrapWord.isSelected());
    }
});
pano.add(chkWrap);
pano.add(chkWrapWord);
getContentPane().add(pano);
}
}

```

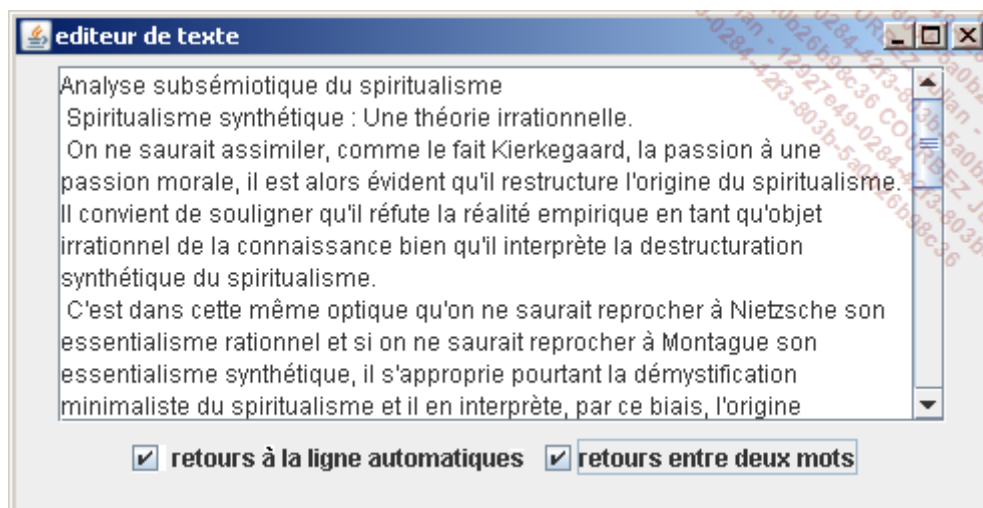
Avec les options par défaut les lignes trop longues ne sont pas visibles entièrement et la barre de défilement horizontal est activée automatiquement.



Avec l'option **LineWrap**, les lignes sont découpées automatiquement à la largeur du composant. Il n'y a plus besoin de barre de défilement horizontal. Par contre une barre de défilement vertical est maintenant nécessaire puisque le nombre de lignes est maintenant supérieur à la capacité du composant.



Pour améliorer la lisibilité du texte, le découpage peut être fait sur des mots entiers avec l'option **WrapStyleWord**.



La gestion du texte contenu dans ce composant est également facilitée grâce à plusieurs méthodes spécifiques.

- **append**(String chaîne) : ajoute la chaîne de caractères passée comme argument au texte déjà présent dans le composant.
- **insert**(String chaîne, int position) : insère la chaîne de caractères passée comme premier argument à la position indiquée par le deuxième argument.
- **replaceRange**(String chaîne, int debut, int fin) : remplace la portion de texte comprise entre la valeur fournie par le deuxième argument et celle fournie par le troisième argument par la chaîne indiquée comme premier argument.

Les composants de déclenchement d'actions

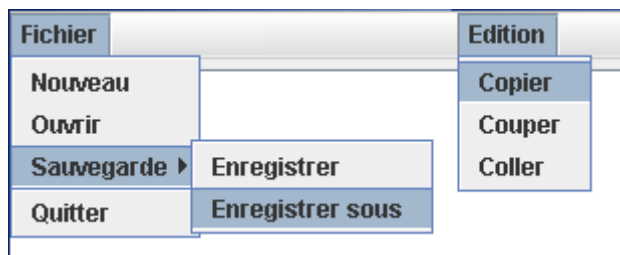
JButton

Le composant **JButton** est principalement utilisé dans une application, pour lancer l'exécution d'une action. Cette action peut être l'exécution d'une portion de code ou la fermeture d'une boîte de dialogue. Comme pour les contrôles vus jusqu'à présent, le libellé du bouton est modifiable par la méthode **setText** du composant. Ce composant peut également contenir une image. Celle-ci est gérée exactement de la même façon que pour le composant **JLabel**. Ce composant est

pratiquement toujours associé à un écouteur implémentant l'interface **ActionListener** afin de gérer les clics sur le bouton.

JMenuBar, JMenu, JMenuItem, JPopupMenu, JSeparator

Cet ensemble de composants va permettre la gestion des menus d'application ou des menus contextuels. Le composant **JMenuBar** constitue le conteneur qui va accueillir les menus représentés par des **JMenu** qui eux-mêmes vont contenir des éléments de menus représentés par des **JMenuItem**. Le composant **JMenuBar** se comporte comme un conteneur pour les **JMenu**. Les **JMenu** vont eux aussi se comporter comme un conteneur pour des **JMenuItem** ou d'autres **JMenu**. La conception d'un menu va donc consister à créer des instances de ces différentes classes puis de les associer les unes aux autres. Au final le **JMenuBar** obtenu est placé sur son conteneur qui n'est autre que la **JFrame** de l'application. Il faut bien sûr également penser à traiter les événements déclenchés par ces différents éléments. En général, seuls les **JMenuItem** sont associés à des écouteurs. Ils se comportent comme des **JBUTTON**. Ils ont d'ailleurs un ancêtre commun puisque tous deux héritent de la classe **AbstractButton**. La conception d'un menu n'est pas très complexe. La seule difficulté réside dans la quantité de code relativement importante nécessaire. Pour illustrer cela, voici le code d'un éditeur de texte rudimentaire possédant les menus suivants :



```
package fr.eni;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.PrintWriter;

import javax.swing.JCheckBox;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
```

```

import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSeparator;
import javax.swing.JTextArea;

public class Ecran14 extends JFrame
{
    JPanel pano;
    JTextArea txt;
    JScrollPane defil;
    JMenuBar barre;
    JMenu mnuFichier,mnuEdition,mnuSauvegarde;
    JMenuItem mnuNouveau,mnuOuvrir,mnuEnregister,
mnuEnregistrerSous,mnuQuitter;
    JMenuItem mnuCopier,mnuCouper,mnuColler;
    File fichier;

    public Ecran14()
    {
        setTitle("éditeur de texte");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // création des composants
        pano=new JPanel();
        pano.setLayout(new BorderLayout());
        txt=new JTextArea();
        defil=new JScrollPane(txt);
        pano.add(defil,BorderLayout.CENTER);
        getContentPane().add(pano);
        // création des composants des menus
        barre=new JMenuBar();
        mnuFichier=new JMenu("Fichier");
        mnuEdition=new JMenu("Edition");
        mnuSauvegarde=new JMenu("Sauvegarde");
        mnuNouveau=new JMenuItem("Nouveau");
        mnuOuvrir=new JMenuItem("Ouvrir");
    }
}

```

```

mnuEnregister=new JMenuItem("Enregistrer");
mnuEnregister.setEnabled(false);
mnuEnregistrerSous=new JMenuItem("Enregistrer sous");
mnuCopier=new JMenuItem("Copier");
mnuCouper=new JMenuItem("Couper");
mnuColler=new JMenuItem("Coller");
mnuQuitter=new JMenuItem("Quitter");
// association des éléments
barre.add(mnuFichier);
barre.add(mnuEdition);
mnuFichier.add(mnuNouveau);
mnuFichier.add(mnuOuvrir);
mnuFichier.add(mnuSauvegarde);
mnuSauvegarde.add(mnuEnregister);
mnuSauvegarde.add(mnuEnregistrerSous);
mnuFichier.add(new JSeparator());
mnuFichier.add(mnuQuitter);
mnuEdition.add(mnuCopier);
mnuEdition.add(mnuCouper);
mnuEdition.add(mnuColler);
// association du menu avec la JFrame
setJMenuBar(barre);
// les écouteurs associés aux différents menus
mnuNouveau.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        fichier=null;
        txt.setText("");
        mnuEnregister.setEnabled(false);
    }
});
mnuOuvrir.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        JFileChooser dlg;
        dlg=new JFileChooser();

```



```

        dlg.showDialog(null, "Ouvrir");
        fichier=dlg.getSelectedFile();
        FileInputStream in;
        try
        {
            in=new FileInputStream(fichier);
            BufferedReader br;
            br=new BufferedReader(new
InputStreamReader(in));

            String ligne;
            txt.setText("");
            while ((ligne=br.readLine())!=null)
            {
                txt.append(ligne+"\r\n");
            }
            br.close();
            mnuEnregister.setEnabled(true);
        }
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }

});
mnuQuitter.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
});
mnuCopier.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {

```

```

        txt.copy();
    }
});
mnuCouper.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.cut();
    }
});
mnuColler.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.paste();
    }
});
mnuEnregister.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            PrintWriter pw;
            pw=new PrintWriter(fichier);
            pw.write(txt.getText());
            pw.close();
        }
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
        }
    }
});
mnuEnregistrerSous.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try

```

```

        {
            JFileChooser dlg;

            dlg=new JFileChooser();

            dlg.showDialog(null,"enregistrer sous");

            fichier=dlg.getSelectedFile();

            PrintWriter pw;

            pw=new PrintWriter(fichier);

            pw.write(txt.getText());

            pw.close();

        }

        catch (FileNotFoundException e)

        {

            e.printStackTrace();

        }

    }

});

}

}

```

JToolBar

Ce composant sert en fait de conteneur pour les éléments constituant une barre d'outils. Ces éléments sont ajoutés au composant **JToolBar** comme pour n'importe quel autre conteneur. Des **JButton** sans libellé mais affichant des images sont généralement placés sur ce composant. Ils fournissent un accès à une fonctionnalité déjà disponible par un menu. À ce titre, ils partagent généralement le même écouteur. Ce composant présente la particularité de pouvoir être déplacé par l'utilisateur sur n'importe laquelle des bordures de la fenêtre, voire dans certains cas à l'extérieur de la fenêtre. Pour que ce mécanisme fonctionne correctement, le composant **JToolBar** doit être placé dans un conteneur utilisant un **BorderLayout** comme gestionnaire de mise en page. Il est dans ce cas placé sur une des bordures (NORTH, SOUTH, WEST, EAST) et doit être le seul composant placé dans cette zone. Cette fonctionnalité peut être désactivée en appelant la méthode **setFloatable(false)**. La barre d'outils reste alors à sa position d'origine. Le code suivant ajoute une barre d'outils à notre éditeur de texte.

```

JToolBar tlbr;

    tlbr=new JToolBar();

    JButton btnNouveau,btnOuvrir,btnEnregister;

    JButton btnCopier,btnCouper,btnColler;

    // création des boutons

    btnNouveau=new JButton(new ImageIcon("new.jpg"));

    btnOuvrir=new JButton(new ImageIcon("open.jpg"));

    btnEnregister=new JButton(new ImageIcon("save.jpg"));

    btnCopier=new JButton(new ImageIcon("copy.jpg"));

    btnColler=new JButton(new ImageIcon("paste.jpg"));

```

```

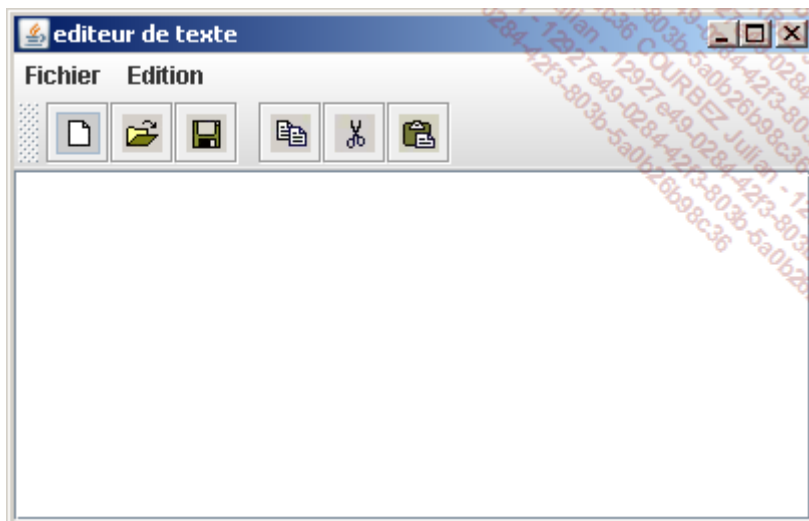
    btnCouper=new JButton(new ImageIcon("cut.jpg"));

    // ajout des boutons à la barre d'outils
    tlbr.add(btnNouveau);
    tlbr.add(btnOuvrir);
    tlbr.add(btnEnregister);
    tlbr.addSeparator();
    tlbr.add(btnCopier);
    tlbr.add(btnCouper);
    tlbr.add(btnColler);

    // ajout de la barre d'outils sur son conteneur
    pano.add(tlbr, BorderLayout.NORTH);

    // réutilisation écouteurs déjà associés aux menus
    btnNouveau.addActionListener(mnuNouveau.getActionListeners()[0]);
    btnOuvrir.addActionListener(mnuOuvrir.getActionListeners()[0]);
    btnEnregister.addActionListener(mnuEnregister.getActionListeners()[0]);
    btnCopier.addActionListener(mnuCopier.getActionListeners()[0]);
    btnCouper.addActionListener(mnuCouper.getActionListeners()[0]);
    btnColler.addActionListener(mnuColler.getActionListeners()[0]);

```



Les composatsns de sélection

JCheckBox

Le composant **JCheckBox** est utilisé pour proposer à l'utilisateur différentes options, parmi lesquelles il pourra en choisir aucune, une, ou plusieurs. Le composant **JCheckBox** peut prendre deux états : coché lorsque l'option est sélectionnée ou non coché lorsque l'option n'est pas sélectionnée. L'état de la case à cocher peut être vérifié ou modifié par les méthodes **isSelected** et **setSelected**. Comme beaucoup d'autres composants, celui-ci peut avoir un libelle et une image. La gestion de l'image est cependant un petit peu différente des **JLabel** puisqu'elle vient remplacer le dessin de la case à cocher. Pour pouvoir dans ce cas faire une distinction entre une **JCheckBox** cochée et une non cochée, vous devez lui associer une

deuxième image correspondant à l'état coché. Cette deuxième image est indiquée par la méthode `setSelectedIcon`.

Le code suivant permet de choisir le style de la police utilisée dans notre éditeur de texte.

```
JPanel options;

    GridLayout gl;
    options=new JPanel();
    gl=new GridLayout(2,1);
    options.setLayout(gl);
    chkGras=new JCheckBox("Gras");
    chkItalique=new JCheckBox("Italique");
    options.add(chkGras);
    options.add(chkItalique);
    chkGras.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            changePolice();
        }
    });
    chkItalique.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            changePolice();
        }
    });

    pano.add(options,BorderLayout.SOUTH);
}

public void changePolice()
{
    int attributs;
    attributs=0;
    if (chkGras.isSelected())
    {
        attributs=attributs+Font.BOLD;
    }
    if (chkItalique.isSelected())
```

```

        {
            attributs=attributs+Font.ITALIC;
        }

        Font police;
        police=new
Font(txt.getFont().getName(),attributs,txt.getFont().getSize());
        txt.setFont(police);
    }

```

JRadioButton

Le composant **JRadioButton** permet également de proposer à l'utilisateur différentes options parmi lesquelles il ne pourra en sélectionner qu'une seule. Comme son nom l'indique, ce contrôle fonctionne comme les boutons permettant de sélectionner une station sur un poste de radio (vous ne pouvez pas écouter trois stations de radio en même temps !). Les caractéristiques de ce composant sont identiques à celles disponibles dans le composant **JCheckBox**. Pour pouvoir fonctionner correctement, les composants **JRadioButton** doivent être regroupés logiquement. La classe **ButtonGroup** fournit cette fonctionnalité. Elle fait en sorte que parmi tous les **JRadioButton** qui lui sont confiés, grâce à sa méthode `add`, un seul d'entre eux pourra être sélectionné à la fois. Pour regrouper physiquement des **JRadioButon** ceux-ci doivent être placés sur un conteneur tel qu'un **JPanel**. Pour bien visualiser ce regroupement, on ajoute généralement une bordure sur le **JPanel** et éventuellement une légende.

C'est ce que nous allons faire en ajoutant deux ensembles de boutons permettant de choisir la couleur du texte et la couleur de fond de notre éditeur.

```

JRadioButton optFondRouge,optFondVert,optFondBleu;

JRadioButton optRouge,optVert,optBleu;
JPanel couleur,couleurFond;
ButtonGroup grpCouleur,grpCouleurFond;
// création des boutons
optRouge=new JRadioButton("Rouge");
optVert=new JRadioButton("Vert");
optBleu=new JRadioButton("Bleu");
optFondRouge=new JRadioButton("Rouge");
optFondVert=new JRadioButton("Vert");
optFondBleu=new JRadioButton("Bleu");
// regroupement logique des boutons
grpCouleur=new ButtonGroup();
grpCouleur.add(optRouge);
grpCouleur.add(optVert);
grpCouleur.add(optBleu);
grpCouleurFond=new ButtonGroup();
grpCouleurFond.add(optFondRouge);

```

```

    grpCouleurFond.add(optFondVert);
    grpCouleurFond.add(optFondBleu);
    // regroupement physique des boutons
    couleur=new JPanel();
    couleur.setLayout(new GridLayout(0,1));
    couleur.add(optRouge);
    couleur.add(optVert);
    couleur.add(optBleu);
    couleurFond=new JPanel();
    couleurFond.setLayout(new GridLayout(0,1));
    couleurFond.add(optFondRouge);
    couleurFond.add(optFondVert);
    couleurFond.add(optFondBleu);
    // ajout d'une bordure avec titre
    couleur.setBorder(BorderFactory.createTitledBorder
(BorderFactory.createEtchedBorder(),"couleur police"));
    couleurFond.setBorder(BorderFactory.createTitledBorder
(BorderFactory.createEtchedBorder(),"couleur fond"));
    // référencement des écouteurs
    optBleu.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            txt.setForeground(Color.BLUE);
        }
    });
    optVert.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            txt.setForeground(Color.GREEN);
        }
    });
    optRouge.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            txt.setForeground(Color.RED);
        }
    });

```

```

    });
    optFondBleu.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            txt.setBackground(Color.BLUE);
        }
    });
    optFondRouge.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            txt.setBackground(Color.RED);
        }
    });
    optFondVert.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            txt.setBackground(Color.GREEN);
        }
    });

```

JList

Le composant **JList** propose à l'utilisateur une liste de choix dans laquelle il pourra en sélectionner un ou plusieurs. Le composant gère automatiquement l'affichage des éléments cependant, il n'est pas capable d'en assurer le défilement. Ce composant doit donc être placé sur un conteneur prenant en charge cette fonctionnalité comme un **JScrollPane**. Les éléments affichés dans la liste peuvent être fournis au moment de la création du composant en indiquant comme argument au constructeur un tableau d'objet ou un **Vector**. Les méthodes **toString** de chacun des objets seront utilisées par le composant **JList** pour pouvoir afficher l'élément. Avec cette solution, la liste est créée en lecture seule et il n'est pas possible d'y ajouter d'autres éléments par la suite. Pour avoir une liste dans laquelle il sera possible d'ajouter d'autres éléments, il faut utiliser un objet de type **DefaultListModel** à qui l'on va confier le soin de gérer les éléments de la liste. Cet objet possède de nombreuses méthodes, similaires aux méthodes disponibles pour un **Vector**, permettant la gestion des éléments. Si vous souhaitez utiliser une liste dynamique, vous devez passer au constructeur de la classe **JList** un objet de ce type.

La liste peut être conçue pour autoriser différents types de sélection. Le choix du type de sélection s'effectue avec la méthode **setSelectionMode** à laquelle l'on passe comme argument une des constantes suivantes :

- **ListSelectionModel.SINGLE_SELECTION** : un seul élément de la liste peut être sélectionné à la fois.
- **ListSelectionModel.SINGLE_INTERVAL_SELECTION** : plusieurs éléments peuvent être sélectionnés mais ils doivent se suivre dans la liste.
- **ListSelectionModel.MULTIPLE_INTERVAL_SELECTION** : plusieurs éléments peuvent être sélectionnés dans la liste et ils ne sont pas forcément contigus.

La récupération de l'élément ou des éléments sélectionnés peut être effectuée avec deux techniques différentes. Vous pouvez obtenir l'index ou les index des éléments sélectionnés avec les méthodes **getSelectedIndex** ou **getSelectedIndices**. Ces méthodes retournent un entier ou un tableau d'entiers représentant l'index ou les index des éléments sélectionnés.

L'objet ou les objets sélectionnés sont eux disponibles grâce aux méthodes **getSelectedValue** et **getSelectedValues**. De manière similaire, ces deux méthodes retournent l'objet sélectionné ou un tableau contenant les objets sélectionnés. La présence d'un élément sélectionné dans la liste peut être testée avec la méthode **isSelectedEmpty**. La sélection peut être annulée avec la méthode **clearSelection**. La modification de la sélection dans la liste déclenche un événement de type **valueChanged**. Cet événement peut être géré en associant à la liste un écouteur de type **ListSelectionListener** avec la méthode **addListSelectionListener**. Il faut être vigilant avec cet événement car il se produit plusieurs fois de suite lorsqu'un élément est sélectionné. Le premier événement correspond à la désélection de l'élément précédent et le deuxième à la sélection de l'élément actuel. C'est donc ce dernier qu'il faut prendre en compte. Pour cela, l'argument **ListSelectionEvent** disponible avec cet événement dispose de la méthode **getValueIsAdjusting** permettant de savoir si la sélection est terminée ou si elle se trouve dans un état transitoire. Pour tester le fonctionnement, nous allons ajouter à notre éditeur de texte une liste proposant le choix d'une police de caractères.

```
JList polices ;
JScrollPane defilPolices;

String[] nomsPolices={"Dialog","DialogInput","Monospaced","Serif",
"SansSerif"};
polices=new JList(nomsPolices);
polices.setSelectedIndex(0);
defilPolices=new JScrollPane(polices);
defilPolices.setPreferredSize(new Dimension(100,60));
options.add(defilPolices);
polices.addListSelectionListener(new ListSelectionListener()
{
    public void valueChanged(ListSelectionEvent e)
    {
        if (!e.getValueIsAdjusting())
        {
            changePolice();
        }
    }
})
```

```

    }
});
...
...
...
public void changePolice()
{
    int attributs;
    attributs=0;
    if (chkGras.isSelected())
    {
        attributs=attributs+Font.BOLD;
    }
    if (chkItalique.isSelected())
    {
        attributs=attributs+Font.ITALIC;
    }
    Font police;
    police=new Font(polices.getSelectedValue().toString(),
attributs,txt.getFont().getSize(
));
    txt.setFont(police);
}

```

JCombobox

Ce composant peut être assimilé à l'association de trois autres composants :

- un JTextField
- un JButton
- un JList

L'utilisateur peut choisir un élément dans la liste qui apparaît lorsqu'il clique sur le bouton ou bien saisir directement dans la zone de texte l'information attendue si celle-ci n'est pas disponible dans la liste. Deux modes de fonctionnement sont disponibles. Vous pouvez n'autoriser que la sélection d'un élément dans la liste en configurant la zone de texte en lecture seule avec la méthode `setEditable(false)` ou autoriser soit la sélection dans la liste, soit la saisie dans la zone de texte avec la méthode `setEditable(true)`. La création d'une instance de ce composant suit les mêmes principes que la création d'un `JList`. La récupération de l'élément sélectionné est effectuée de manière semblable à une `JList`. Attention tout de même car il n'y a pas de sélections multiples dans ce type de composant. La détection de la sélection d'un nouvel élément peut être faite en gérant l'événement `actionPerformed`. Cet événement se déclenche lors de la sélection d'un élément dans la liste ou lorsque l'utilisateur valide la saisie avec la touche Enter. Nous terminons notre éditeur de texte en proposant à l'utilisateur le choix d'une taille de police de caractères.

```

JComboBox cboTaille ;

String tailles[]={"10","12","14","16","20"};

cboTaille=new JComboBox(tailles);

cboTaille.setEditable(true);

options.add(cboTaille);

cboTaille.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        changePolice();
    }

});

...

...

...

    public void changePolice()
    {
        int attributs;
        attributs=0;
        if (chkGras.isSelected())
        {
            attributs=attributs+Font.BOLD;
        }
        if (chkItalique.isSelected())
        {
            attributs=attributs+Font.ITALIC;
        }

        Font police;
        System.out.println(cboTaille.getSelectedItem().toString());
        police=new Font(polices.getSelectedValue().toString(),
attributs,Integer.parseInt (cboTaille.getSelectedItem().toString()));
        txt.setFont(police);
    }
}

```

2.6. Les boîtes de dialogue

Les boîtes de dialogue sont des fenêtres qui ont une fonction spéciale dans une application. Elles sont en général utilisées pour demander la saisie d'informations à l'utilisateur, lui présenter un message ou lui poser une question. Pour s'assurer que la boîte de dialogue a bien été prise en compte par l'utilisateur avant qu'il continue à utiliser l'application, celle-ci est affichée en mode modal, c'est-à-dire que le reste de l'application, est bloqué tant que la boîte de dialogue est affichée. Pour nous éviter d'avoir à recréer à chaque fois une nouvelle boîte de dialogue, nous avons à notre disposition plusieurs boîtes de dialogue prédéfinies.

Elles sont disponibles par l'intermédiaire de méthodes **static** dans la classe **JOptionPane**.

La boîte de saisie

La boîte de saisie permet de demander à l'utilisateur la saisie d'une chaîne de caractères. Cette fonctionnalité est disponible par l'intermédiaire de la fonction **showInputDialog** de la classe **JOptionPane**. Plusieurs versions de cette fonction sont à notre disposition pour nous permettre de configurer différemment l'affichage de la boîte de dialogue. La plus simple n'attend comme paramètre qu'une seule chaîne de caractères représentant le message affiché sur la boîte de saisie. Généralement, ce message indique la nature des informations que l'utilisateur doit saisir. Le code suivant :





```
String nom;  
  
nom=JOptionPane.showInputDialog("saisir le nom du client");
```

Affiche cette boîte de dialogue :



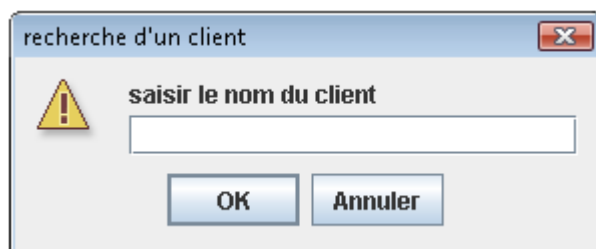
Une version plus complète permet de choisir :

- Le mode d'affichage : fenêtre autonome ou fenêtre interne à un composant. Dans ce cas, il faut fournir comme premier paramètre le composant concerné et la valeur `null` si la fenêtre est autonome.
- Le message affiché sur la boîte de saisie. C'est le deuxième paramètre qui est utilisé à cet effet.
- Le titre de la boîte de saisie comme troisième paramètre.
- Le type d'icône affichée sur la boîte de saisie. Une série de constantes définies dans la classe permet d'effectuer ce choix.

Constante	Icône
ERROR_MESSAGE	
INFORMATION_MESSAGE	
WARNING_MESSAGE	
QUESTION_MESSAGE	
PLAIN_MESSAGE	Pas d'icône

Voici un exemple d'utilisation de cette version plus évoluée et le résultat correspondant.

```
String nom;
nom=JOptionPane.showInputDialog(null,"saisir le nom du client",
"recherche d'un client",JOptionPane.WARNING_MESSAGE);
```



Une autre version ayant un fonctionnement légèrement différent est également disponible. Celle-ci ne propose pas une saisie libre dans une zone de texte, mais une liste d'objets parmi lesquels l'utilisateur devra faire son choix. La liste représente les objets en appelant la méthode **toString** de chacun d'eux. Elle attend comme paramètres les informations suivantes :

- Le mode d'affichage : fenêtre autonome ou fenêtre interne à un composant. Dans ce cas, il faut fournir comme premier paramètre le composant concerné et la valeur `null` si la fenêtre est autonome.
- Le message affiché sur la boîte de saisie. C'est le deuxième paramètre qui est utilisé à cet effet.
- Le titre de la boîte de saisie comme troisième paramètre.
- Le type d'icône affichée sur la boîte de saisie. Une série de constantes définies dans la classe permet d'effectuer ce choix.
- Une icône permettant le remplacement de l'icône par défaut ou `null` pour conserver l'icône par défaut.
- Un tableau d'objets constituant la liste des choix présentés à l'utilisateur.
- L'objet représentant le choix par défaut sélectionné à l'affichage de la boîte de saisie.

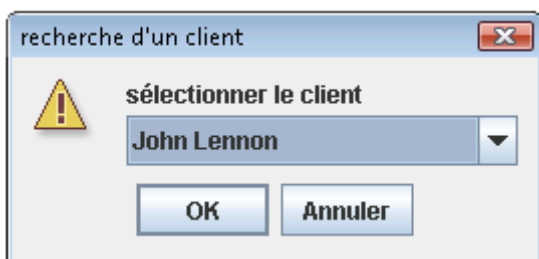
En retour, cette fonction fournit l'objet sélectionné ou `null` si le bouton **Annuler** est utilisé pour fermer la boîte de saisie.

L'exemple ci-dessous propose la sélection d'une personne parmi toutes celles proposées dans la liste.

```
Personne[] choix;
choix=new Personne[5];
choix[0] = new Personne("Wayne", "John",LocalDate.of(1907,5,26));
choix[1] = new Personne("McQueen", "Steeve",LocalDate.of(1930,3,24));
choix[2] = new Personne("Lennon", "John",LocalDate.of(1940,10,9));
choix[3] = new Personne("Gibson", "Mel",LocalDate.of(1956,1,3));
choix[4] = new Personne("Willis", "Bruce",LocalDate.of(1955,3,19));

Personne choisie;

choisie=(Personne) JOptionPane.showInputDialog(null,
"sélectionner le client","recherche d'un client",
JOptionPane.WARNING_MESSAGE,null,choix,choix[1]);
```



La boîte de message

La boîte de message permet de passer une information à l'utilisateur. La boîte de message est disponible par l'intermédiaire de la fonction `showMessageDialog`.

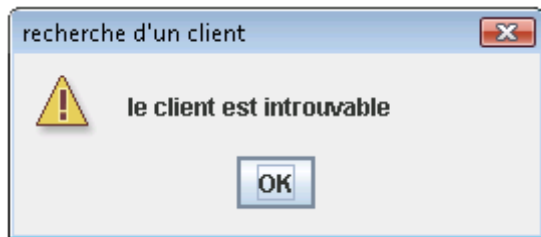
Cette fonction accepte les paramètres suivants :

- Le mode d'affichage : fenêtre autonome ou fenêtre interne à un composant. Dans ce cas, il faut fournir comme premier paramètre le composant concerné et la valeur `null` si la fenêtre est autonome.
- Le message affiché sur la boîte de message. C'est le deuxième paramètre qui est utilisé à cet effet.
- Le titre de la boîte de message comme troisième paramètre.
- Le type d'icône affichée sur la boîte de saisie. Une série de constantes définies dans la classe permet d'effectuer ce choix. Ces constantes sont identiques à celles utilisables pour la boîte de saisie.

Le code suivant :

```
JOptionPane.showMessageDialog(null,"le client est introuvable",
"recherche d'un client",JOptionPane.WARNING_MESSAGE);
```

Affiche cette boîte de message :



La boîte de confirmation

Cette boîte de dialogue est utilisée pour poser une question à l'utilisateur et lui permet de répondre par l'intermédiaire de l'un des boutons qu'elle propose. Les boutons disponibles sont prédéfinis par la boîte de confirmation. La fonction **showConfirmDialog** provoque l'affichage de cette boîte de dialogue. Elle retourne un entier permettant d'identifier le bouton utilisé pour fermer la boîte de dialogue, et donc obtenir la réponse de l'utilisateur. Elle accepte les paramètres suivants :

- Le mode d'affichage : fenêtre autonome ou fenêtre interne à un composant. Dans ce cas, il faut fournir comme premier paramètre le composant concerné et la valeur **null** si la fenêtre est autonome.
- Le message affiché sur la boîte de message. C'est le deuxième paramètre qui est utilisé à cet effet.
- Le titre de la boîte de message comme troisième paramètre.
- La combinaison de boutons affichés. Les constantes **YES_NO_OPTION**, **YES_NO_CANCEL_OPTION** ou **OK_CANCEL_OPTION** sont utilisables pour ce paramètre.
- Le type d'icône affichée sur la boîte de saisie. Une série de constantes définies dans la classe permet d'effectuer ce choix. Ces constantes sont identiques à celles utilisables pour la boîte de saisie.

Cette fonction retourne un entier qu'il faut comparer aux constantes suivantes **YES_OPTION**, **OK_OPTION**, **NO_OPTION**, **CANCEL_OPTION** ou **CLOSE_OPTION** pour déterminer la réponse de l'utilisateur.

```
int reponse;

reponse=JOptionPane.showConfirmDialog(null,"voulez vous quitter
sans enregistrer",
    "fermeture de l'application", JOptionPane.YES_NO_OPTION,
    JOptionPane.WARNING_MESSAGE);
```