Ecole d'Ingénieurs de l'Etat de Vaud

Résumé Swing - awt

VERSION NO 2

5 Janvier 2003

Prof. Eric Lefrançois

Contenu

1 Composants graphiques

1.1 Conception d'une interface utilisateur

Vous pouvez assembler facilement et rapidement les éléments de l'interface utilisateur d'une application ou d'une applet Java. En utilisant les outils de conception visuelle de NetBeans, JBuilder, Eclise ou autres.. vous construisez simplement l'interface avec divers blocs choisis dans une palette contenant des composants : boutons, zones de texte, listes, dialogues et menus.

Vous définissez ensuite les valeurs des propriétés des composants et vous attachez le code des gestionnaires aux événements des composants, pour indiquer au programme comment répondre aux événements de l'interface utilisateur.

1.1.1 Qu'est-ce qu'un composant graphique ?

Les composants sont les blocs de construction utilisés par les outils de conception visuelle de JBuilder ou autres pour construire un programme. Chaque composant représente un élément de programme, tel un objet de l'interface utilisateur, une base de données ou un utilitaire système. Vous construisez un programme en choisissant et en

reliant ces éléments.

Les environnements de programmation cités précédemment fournissent un ensemble de composants prêts à l'emploi dans la palette des composants. Vous pouvez compléter cet ensemble en créant vos propres composants ou en installant des composants tiers.

Chaque composant, encapsule certains éléments d'un programme, tels une fenêtre ou une boîte de dialogue, un champ d'une base de données, un timer système. Les composants d'interface utilisateur doivent étendre la classe `java.awt.Component` ou toute autre classe dérivée d'elle, comme `javax.swing.JPanel` ou encore `javax.swing.JFrame`.

1.1.2 Les conteneurs

Les conteneurs contiennent et gèrent des composants graphiques. Ils dérivent `java.awt.Container`. A l'exécution, ils apparaissent généralement sous forme de panneaux, de fenêtres ou de boîtes de dialogues. La totalité de votre travail de conception d'interface utilisateur se fait dans des conteneurs. Les conteneurs sont aussi des composants. En tant que tels, ils vous laissent interagir avec eux, c'est-à-dire définir leurs propriétés, appeler leurs méthodes et répondre à leurs événements.

- **JFrame**

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

- **JPanel**

  Panneau graphique, jouant principalement le rôle de **conteneur**. Les `JPanel` peuvent s'imbriquer les uns dans les autres sans limite aucune.C'est la brique de base qui permet au programmeur de mettre en place les interfaces graphiques les plus compliqués qui soient.

  Un `JPanel` peut également jouer le rôle de surface graphique dans laquelle le programmeur peut dessiner ou afficher des images. Il suffit pour cela de redéfinir la méthode **`paintComponent(Graphics)`**.

- **JApplet**

  Une sous-classe de la classe `java.awt.Panel` utilisée pour construire un programme devant être incorporé dans une page HTML et exécuté dans un navigateur HTML ou dans un visualiseur d'applet. `JApplet` étant une sous-classe de `Panel`, elle peut contenir des composants, mais elle ne possède ni bordure, ni titre.

1.1.3 Connaître la taille utile d'une fenêtre

La méthode `getInsets()`, héritée de la classe `Container`, est intéressante pour connaître la ***surface utile*** mise à disposition par une fenêtre. Par «surface utile», nous entendons la surface mise à disposition par la fenêtre sans compter la bordure et la barre de titre.

Cette méthode retourne un objet de type `Insets`, caractérisé par 4 attributs publics: `top`, `bottom`, `left` et `right`, qui indique la taille occupée de manière générale par la bordure du conteneur. Par exemple, l'attribut `top` d'un `JFrame` caractérise la hauteur en pixels occupée par la barre de titre.

Ainsi, pour définir une fenêtre de taille utilisable de 300*200 pixels:

```
f.setSize(f.getInsets().left + f.getInsets().right  + 300,
          f.getInsets().top  + f.getInsets().bottom + 200);
```

1.2 LES COMPOSANTS DE LA LIBRAIRIE SWING

Voici la hiérarchie des composants de Swing:

Figure 1: Composants similaires à l'awt

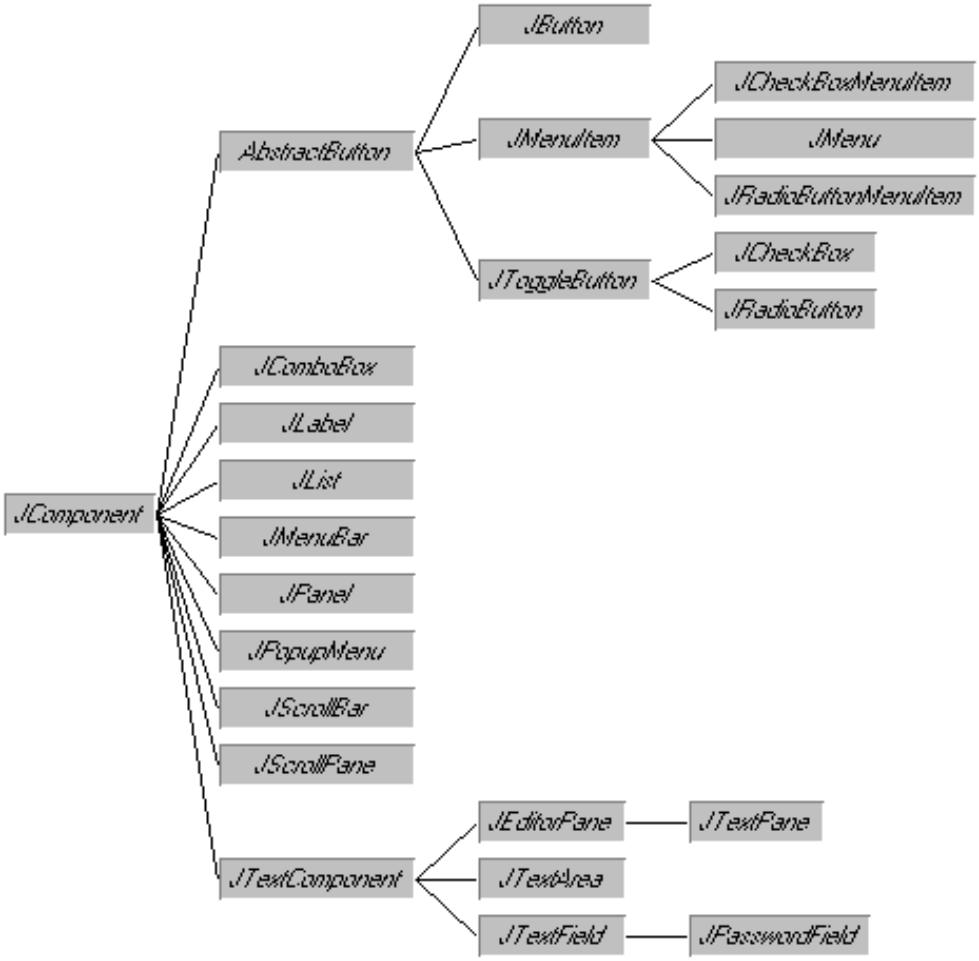
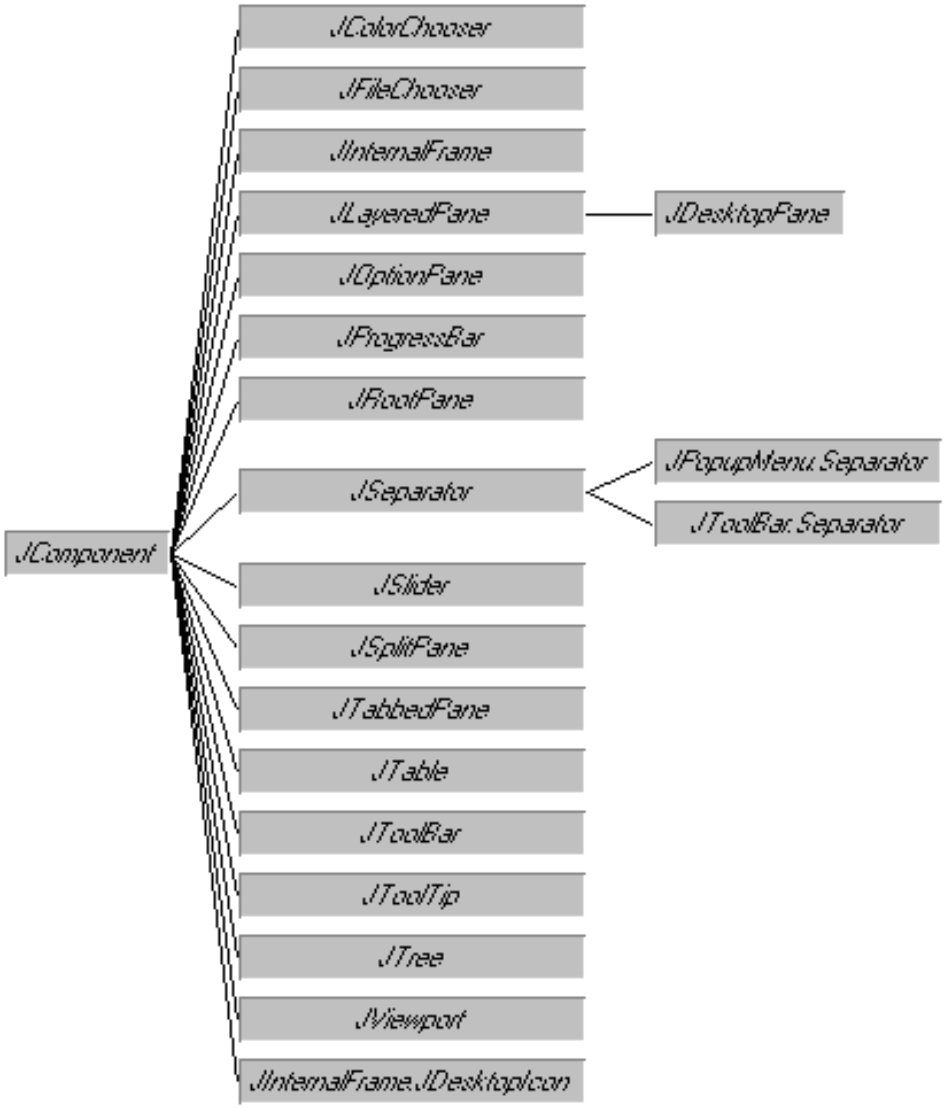


Figure 2: Nouveaux composants

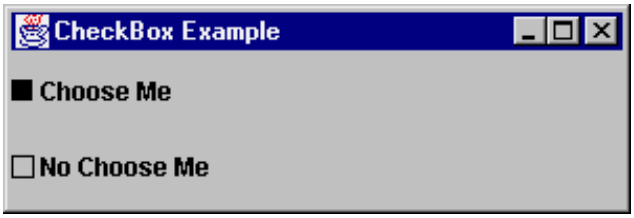


1.2.1 JCheckbox (boîte à cocher)

Un «check box» est un composant graphique pouvant être soit dans l'état «on» (true) ou dans l'état «off» (false). C'est en cliquant sur la boîte à cocher que son état passe de la valeur «on» à la valeur «off» et vice-et-versa.

Il est possible de spécifier les deux icônes qui seront utilisées pour afficher les états «on» et «off».

.




```
public class CheckboxPanel extends JPanel {

   Icon unchecked = new ToggleIcon (false);
   Icon checked = new ToggleIcon (true);


   public CheckboxPanel() {

      // Spécifier le gestionnaire de dipsosition du JPanel
      setLayout(new GridLayout(2, 1));

      // Création du premier CheckBox, initialisé à «true»
      JCheckBox cb1 = new JCheckBox("Choose Me", true);
      cb1.setIcon(unchecked);
      cb1.setSelectedIcon(checked);

      // Création du deuxième CheckBox, initialisé à «true»
      JCheckBox cb2 = new JCheckBox("No Choose Me", false);
      cb2.setIcon(unchecked);
      cb2.setSelectedIcon(checked);
      add(cb1);
      add(cb2);
   }
   class ToggleIcon implements Icon {
      boolean state;

      public ToggleIcon (boolean s) {
         state = s;
      }

      public void paintIcon (Component c, Graphics g,
                             int x, int y) {
         int width = getIconWidth();
         int height = getIconHeight();
         g.setColor (Color.black);
         if (state)
            g.fillRect (x, y, width, height);
         else
            g.drawRect (x, y, width, height);
      }

      public int getIconWidth() {
         return 10;
      }
```

```
        public int getIconHeight() {
            return 10;
        }
    }
}
```

Pour créer un `JCheckBox`, les constructeurs

- **`JCheckBox`**`()`
  Pour créer une boîte à cocher vierge
- `public` **`JCheckBox`**`(Icon icon)`
  Création en spécifiant une image
- `public` **`JCheckBox`**`(Icon icon, boolean selected)`
  Création en spécifiant une image et un état initial
- `public` **`JCheckBox`**`(String text)`
  Création en spécifiant un texte
- `public` **`JCheckBox`**`(String text, boolean selected)`
  Création en spécifiant un texte et un état initial
- `public` **`JCheckBox`**`(String text, Icon icon, boolean selected)`
  Création en spécifiant un texte, une image et un état initial

Evénements spécifiques

- void **addActionListener**(ActionListener l)
  Pour inscrire un écouteur d'événement `ActionEvent` auprès de la boîte à cocher. L'écouteur recevra le message **`actionPerformed`**`(ActionEvent)` à chaque fois que l'état de la boîte à cocher aura été modifié.
- void **removeActionListener**(ActionListener l)
  Pour désinscrire un écouteur d'événement

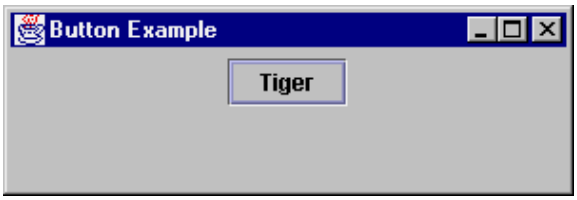
Autres méthodes intéressantes (héritées de `AbstractButton`)

- `void` **`setText`**`(String text)`
  Pour spécifier le texte de la boîte à cocher
- `void` **`setIcon`**`(Icon icon)`
  Pour spécifier l'image de la boîte à cocher
- `public void` **`setSelected`**`(boolean b)`
  Pour spécifier l'état de la boîte à cocher
- `public boolean` **`isSelected`**`()`
  Retourne l'état de la boîte à cocher

1.2.2 JButton

Un bouton est un composant graphique, susceptible d'être «cliqué» par l'utilisateur, et qui répondra à cet événement en envoyant le message `actionPerformed(..)` à tous les objets qui se seront inscrits auprès de lui en tant qu'écouteur.

Voici ce à quoi ressemble un bouton:




```
public class JButtonTest extends JPanel {
    public JButtonTest () {
        JButton myButton = new JButton("Tiger");
        add(myButton);
    }
}
```

- **Possibilité d'insérer une image dans un bouton (`Icon`)**

  Il est possible de placer une image à l'intérieur d'un bouton, l'image devant être manipulée au travers d'un objet de type `Icon`. L'image peut être spécifiée dans le constructeur lui-même, ou en utilisant la méthode **`setIcon`**`(`*`uneIcone`*`)`.

  Comme par exemple:


```
public class ButtonPanel extends JPanel {
    public ButtonPanel() {
        Icon tigerIcon =
            new ImageIcon("SmallTiger.gif");
        JButton myButton =
            new JButton("Tiger", tigerIcon);
        add(myButton);
  }
}
```

- **Pour créer un JButton, les constructeurs**
  - **`JButton`**`()`
    Pour créer un bouton, sans étiquette ni image
  - **`JButton`**`(Icon icon)`
    Pour créer un bouton avec une image
  - **`JButton`**`(String text)`
    Pour créer un bouton avec une étiquette
  - **`JButton`**`(String text, Icon icon)`
    Pour créer un bouton avec une étiquette et une image

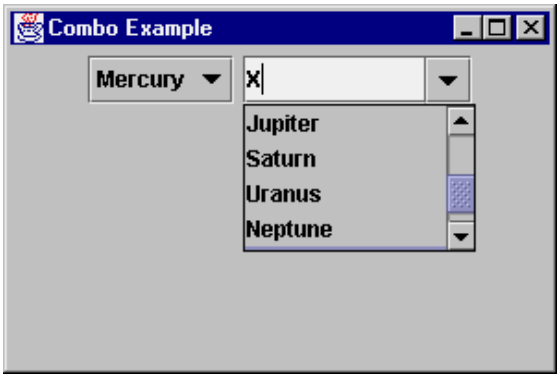
- **Evénements spécifiques**
  - void **addActionListener**(ActionListener l)
    Pour inscrire un écouteur d'événement `ActionEvent` auprès d'un bouton. L'écouteur recevra le message **`actionPerformed`**`(ActionEvent)` à chaque fois que le bouton aura été cliqué.
  - void **removeActionListener**(ActionListener l)
    Pour désinscrire un écouteur d'événement

- **Autres méthodes intéressantes (héritées de `AbstractButton`)**
  - `void` **`setIcon`**`(Icon defaultIcon)`
    Pour spécifier l'image du bouton.
  - `void` **`setText`**`(String text)`
    Pour spécifier l'étiquette du bouton.
  - `void` **`doClick`**`(int pressTime)`

1.2.3 `JComboBox` (menu déroulant)

La classe JComboBox représente un menu déroulant comprenant une liste de choix pré-établie, à la manière de la classe Choice de l'awt, mais donnant en plus la possibilité à l'utilisateur de saisir une nouvelle valeur.

Si l'utilisateur presse une touche correspondant à la première lettre de l'un des choix possibles, le composant le sélectionne automatiquement.




```
public class ComboPanel extends JPanel {
    String choices[] = {
        "Mercury", "Venus", "Earth",
        "Mars", "Jupiter", "Saturn",
        "Uranus","Neptune", "Pluto"};

    public ComboPanel() {
        JComboBox combo1 = new JComboBox();
        JComboBox combo2 = new JComboBox();

        for (int i=0;i<choices.length;i++) {
            combo1.addItem (choices[i]);
            combo2.addItem (choices[i]);
        }

        combo2.setEditable(true);
        combo2.setSelectedItem("X");
        combo2.setMaximumRowCount(4);

        add(combo1);
        add(combo2);
    }
}
```

Pour créer un JComboBox, les constructeurs

- `public` **`JComboBox`**`()`
  Pour créer un ComboBox, avec une liste de choix vide
- `public` **`JComboBox`**`(Vector items)`
  Pour créer un ComboBox avec une liste de choix stockée dans un Vector
- `public` **`JComboBox`**`(Object[] items)`
  Pour créer un ComboBox avec une liste de choix stockée dans un tableau

Pour ajouter des éléments dans la liste de choix

- public **addItem**(Object anObject)

Pour ajouter un nouvel élément dans la liste de choix

Evénements spécifiques

- public void **addActionListener**(ActionListener l)
  Pour inscrire un écouteur d'événement auprès du `JComboBox`. L'écouteur recevra le message `actionPerformed(ActionEvent)` quand une sélection sera opérée par l'utilisateur. Un événement sera également notifié si le ComboBox possède un champ d'édition et qu'une saisie aura été opérée par l'utilisateur.

- void **removeActionListener**(ActionListener l)
  Pour désinscrire un écouteur d'événement

Autres méthodes intéressantes

- `int` **`getSelectedIndex`**`()`
  Pour retourner l'indice de l'élément sélectionné

- `void` **`setSelectedIndex`**`(int anIndex)`
  Pour pré-sélectionner un élément en spécifiant son indice

- `Object` **`getSelectedItem`**`()`
  Pour retourner l'élément sélectionné

- `void` **`setSelectedItem`**`(Object anItem)`
  Pour pré-sélectionner un élément

- `void` **`setEditable`**`(boolean aFlag)`
  Pour déterminer si le Combox possède ou non un champ de saisie pour compléter la liste de choix.

- void **setMaximumRowCount**(int count)
  Pour spécifier le nombre maximum de choix qui seront affichés dans le menu déroulant. Si ce nombre n'est pas suffisant pour afficher tous les choix possibles, une barre de défilement sera automatiquement rajoutée.

1.2.4 JLabel

Un `JLabel` affiche du texte, ou une une image, sur une simple ligne en lecture uniquement.

Possibilités:

1. ajouter une icône
2. spécifier la position horizontale et verticale du texte par rapport à l'icône

3. Set the relative position of contents within component


```
public class LabelPanel extends JPanel {
  public LabelPanel() {
   // Créer et ajouter un JLabel
    JLabel plainLabel = new JLabel("Plain Small Label");
    add(plainLabel);

   // Créer un deuxième JLabel

    JLabel fancyLabel = new JLabel("Fancy Big Label");

    // Instantier une fonte
    Font fancyFont =
          new Font("Serif", Font.BOLD | Font.ITALIC, 32);

   //Associer la fonte avec le label
    fancyLabel.setFont(fancyFont);

    // Créer une icône
    Icon tigerIcon = new ImageIcon("SmallTiger.gif");

   // Placer l'icône dans le label
    fancyLabel.setIcon(tigerIcon);

    // Aligner le texte à la droite de l'icône
    fancyLabel.setHorizontalAlignment(JLabel.RIGHT);

    // Ajouter le label au panell
    add(fancyLabel);
  }
}
```

1.2.5 JTextComponents

`JTextComponent` est une classe générique qui contient toutes les caractéristiques qui pourraient être utiles à la création d'un petit éditeur simplifié.

Voici quelques méthodes, à titre d'illustration:

- `copy()`
- `cut()`
- `paste()`

- `getSelectedText()`
- `setSelectionStart()`
- `setSelectionEnd()`
- `selectAll()`
- `replaceSelection()`
- `getText()`
- `setText()`
- `setEditable()`
- `setCaretPosition()`

La classe `JTextComponent` ne sera jamais instantiée directement. Il existe en revanche 3 sous-classes particulièrement importantes: **`JTextField`**, **`JTextArea`**, and **`JEditorPane`**. **`JPasswordField`** et **`JTextPane`** sont également deux sous-classes présentant un certain intérêt.

Si l'utilisateur doit avoir la possibilité de visualiser un contenu qui dépasse l'espace mis à disposition dans la fenêtre d'affichage, le composant texte devra être placé à l'intérieur d'un `JScrollPane` qui offre la possibilité de visualiser tout le contenu au moyen d'une barre de navigation.

JTextField & JTextArea

```
// Instancier un champ de saisie (limité à une ligne de texte)
JTextField tf = new JTextField();

// Instancier une zone de saisie (qui peut comprendre plusieurs lignes de texte)

JTextArea ta = new JTextArea();

// Initialiser leur contenu
tf.setText("TextField");
ta.setText("JTextArea");


add(tf);
add(new JScrollPane(ta));// Avec une barre de défilement
```

Pour plus de détails, voir l'API Java du JDK

1.2.6 JToolTip

Un «tooltip» est un texte fugitif, dépendant du contexte, qui s'affiche dans une petite fenêtre quand la souris s'attarde sur un objet spécifique de l'écran. La classe `JToolTip` elle-même sera rarement utilisée directement. Il suffit en effet d'utiliser la méthode **`setToolTipText()`**héritée de la classe `JComponent`.

- **Exemple**


```
public class TooltipPanel extends JPanel {
    public TooltipPanel() {
        JButton myButton = new JButton("Hello");
        myButton.setToolTipText ("World");
        add(myButton);
    }
```

1.2.7 JToolBar

Le composant `JToolBar` est une espèce de *conteneur* qui affiche ses composants à la manière d'une barre d'outils, verticalement ou horizontalement, en s'adaptant à la surface de l'écran dans laquelle il se trouve placé.

Une barre d'outils peut être **flottante:** l'utilisateur pouvant la déplacer et la placer où bon lui semble en draguant la barre d'outils dans une autre région de l'écran, ou même en déplaçant cette dernière dans une fenêtre externe au conteneur d'origine.

Pour que cette fonctionnalité fonctionne correctement, il est recommendé d'ajouter la barre d'outils à l'un ou l'autre des 4 côtés d'un conteneur dont le gestionnaire de dispostion est de type BorderLayout, et de ne rien placer dans les 3 côtés restants.

Par défaut, une barre d'outils a la faculté d'être flottante.

Pour supprimer la faculté «floatable» d'une barre d'outils:

```
unToolBar.setFlotable(false);
```

Pour créer un JToolBar

- **`JToolBar`**`()`
- **`JToolBar`**`(int)`
  Le paramètre permet de spécifier l'orientation de la barre d'outils: **`HORIZONTAL`** ou **`VERTICAL`**, ces deux constantes étant définies dans l'interface `javax.swing.SwingConstants`.
- **`JToolBar`**`(String)`
  Le paramètre permet de spécifier un titre, qui apparaîtra lorsque la barre d'outils aura quitté son port d'attache.

- **`JToolBar`**`(String, int)`
  Les deux paramètres sont expliqués dans les lignes qui précèdent

Ajouter des composants dans un JToolBar

Les composants d'un `JToolBar` sont disposés à la manière d'un gestionnaire de type `BoxLayout`. Concrètement, il suffit d'utiliser la méthode `add`:

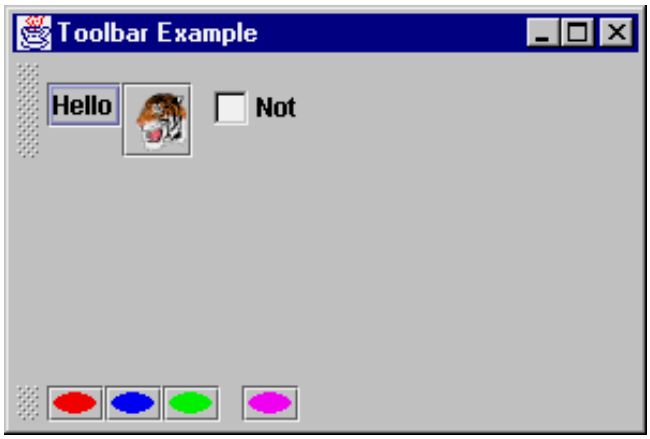
```
unToolBar.add(unComposant);
```

Un espace vide peut être rajouter entre les composants au moyen de la méthode `addSeparator()`:

```
unToolBar.addSeparator();
```

Un exemple

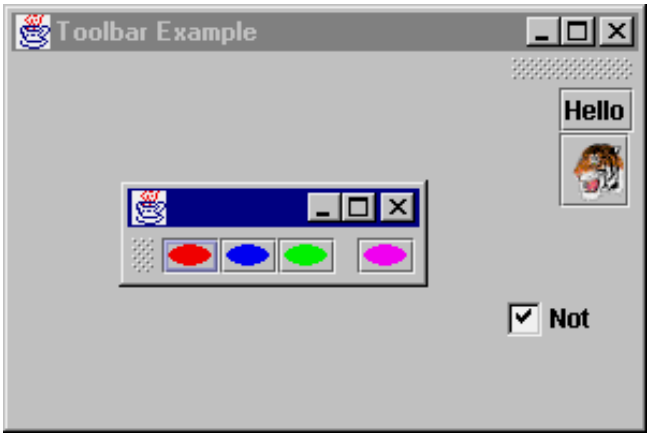
Voici un petit exemple présentant un `JPanel` dans lequel deux barres d'outil ont été placées, la première au nord, et la deuxième au sud. L'effet obtenu serait bien entendu plus esthétique si les composants placés dans les barre d'outils étaient de type et de taille identiques.




```
public class ToolbarPanel extends JPanel {
    public ToolbarPanel() { // Constructeur du JPanel
        // Gestionnaire de disposition de type BorderLayout
        setLayout (new BorderLayout());


        // Barre d'outils placée au nord
        JToolBar toolbar = new JToolBar();
        JButton myButton = new JButton("Hello");
        toolbar.add(myButton);

        Icon tigerIcon = new ImageIcon("SmallTiger.gif");
        myButton = new JButton(tigerIcon);
        toolbar.add(myButton);
        toolbar.addSeparator();
        toolbar.add (new Checkbox ("Not"));
        add (toolbar, BorderLayout.NORTH);
```

```
        // Barre d'outils placée au sud
        toolbar = new JToolBar();
        Icon icon = new AnOvalIcon(Color.red);
        myButton = new JButton(icon);
        toolbar.add(myButton);

        icon = new AnOvalIcon(Color.blue);
        myButton = new JButton(icon);
        toolbar.add(myButton);

        icon = new AnOvalIcon(Color.green);
        myButton = new JButton(icon);
        toolbar.add(myButton);
        toolbar.addSeparator();

        icon = new AnOvalIcon(Color.magenta);
        myButton = new JButton(icon);
        toolbar.add(myButton);
        add (toolbar, BorderLayout.SOUTH);
    }

    class AnOvalIcon implements Icon {
        Color color;

        public AnOvalIcon (Color c) {
            color = c;
        }

        public void paintIcon (Component c, Graphics g,
                               int x, int y) {
            g.setColor(color);
            g.fillOval (x, y, getIconWidth(), getIconHeight());
        }

        public int getIconWidth() {
            return 20;
        }

        public int getIconHeight() {
            return 10;
        }
    }
}
```

Il est alors possible de déplacer la barre d'outils:



2 Gestionnaires de disposition

2.1 En préliminaire

Un programme écrit en Java peut être déployé sur plusieurs plates-formes. Si vous utilisiez des techniques standard de conception d'interface utilisateur en spécifiant la position et la taille de vos composants en valeur absolue, votre interface n'apparaîtrait pas correctement sur toutes les plates-formes. Ce qui paraît bon avec le système sur lequel vous développez peut s'avérer inutilisable sur une autre plate-forme. Pour résoudre ce problème, Java fournit un système portable de gestionnaires de disposition. Utilisez ces gestionnaires de disposition pour spécifier les règles et les contraintes de disposition au sein de votre interface utilisateur, de sorte que cette dernière puisse être portée sur une autre machine.

Les gestionnaires de disposition vous procurent les avantages suivants:

1. Des composants correctement positionnés, car indépendants des polices, des résolutions d'écrans et des différences de plates-formes.
2. Un positionnement intelligent des conteneurs, qui sont redimensionnés à l'exécution de façon dynamique.
3. La traduction des chaînes facilitée. Si une chaîne s'agrandit, les composants restent correctement alignés.

2.2 A propos des gestionnaires de disposition

Un conteneur d'interface utilisateur Java (java.awt.Container) utilise un objet spécial, appelé ***gestionnaire de disposition***, pour contrôler l'emplacement et la taille des composants chaque fois qu'ils sont affichés. Un gestionnaire de disposition arrange automatiquement les composants dans un conteneur, selon un ensemble de règles propre à chaque gestionnaire de disposition.

Le gestionnaire de disposition définit la taille et l'emplacement des composants selon divers facteurs, comme:

- Les règles de disposition du gestionnaire de disposition;
- Les définitions des propriétés du gestionnaire de disposition, s'il y en a;
- Les *contraintes* de disposition associées à chaque composant;

- Certaines propriétés communes à tous les composants, comme `preferredSize`, `minimumSize`, `maximumSize`, `alignmentX` et `alignmentY`.
- La taille du conteneur.

Le gestionnaire par défaut des conteneurs

Certains types de conteneurs utilisent par défaut des gestionnaires de disposition spécifiques:

- Tous les panneaux, - c'est-à-dire les «**`JPanel`**» -, (y compris les applets) utilisent par défaut le gestionnaire **`FlowLayout`**.
- Toutes les fenêtres (y compris les cadres et les dialogues) utilisent **`BorderLayout`**.

Choisir un autre gestionnaire pour un conteneur

Lorsque vous créez un conteneur dans un programme Java, vous pouvez accepter le gestionnaire de disposition par défaut pour ce type de conteneur, ou bien vous pouvez le remplacer par un autre type de gestionnaire au moyen de la méthode **`setLayout(..)`** comme par exemple:

```
leContainer.setLayout(null);
leContainer.setLayout(new BorderLayout());
leContainer.setLayout(new FlowLayout(FlowLayout.RIGHT, 5, 5));
```

2.3 EXPLICATION DES PROPRIÉTÉS DE DIMENSIONNEMENT

Les gestionnaires de disposition utilisent diverses informations pour déterminer la façon de positionner et de dimensionner les composants dans leurs conteneurs. Les composants graphiques disposent d'un ensemble de méthodes qui permettent aux gestionnaires de disposer intelligemment les composants. Toutes ces méthodes sont fournies de sorte qu'un composant puisse communiquer la taille qu'il désire au composant responsable de son dimensionnement (généralement un gestionnaire de disposition).

Ces méthodes, qui ressemblent à des acquéreurs de propriétés, sont énumérées ci-dessous.

`public Dimension getPreferredSize();`

La **taille préférée**, celle que le composant choisirait d'avoir, c'est-à-dire la taille lui donnant le meilleur aspect. Selon les règles du gestionnaire de disposition, la taille préférée, preferredSize, *peut être ou ne pas être prise en compte lors de la disposition du conteneur.* Pour plus de détails, se référer au paragraphe correspondant [Voir paragraphe - Les principaux gestionnaires - page 23] se trouvent décrits chacun des gestionnaires de disposition.

Pour spécifier la taille préférée d'un composant, le programmeur a deux possibilités:

1. envoyer le message **`setPreferredSize`** au composant

2. ou redéfinir la méthode **`getPreferredSize()`**.

Nous recommendons la dernière méthode, plus souple, qui permet un meilleur contrôle dynamique de la taille du composant: la méthode preferredSize peut retourner l'état de variables d'instance du composantsaccessibles de l'extérieur.

Comme par exemple:

```
public Dimension getPreferredSize(){
       return new Dimension (largeur, hauteur);
}
```

- **`public Dimension getMinimumSize();`**
  La **taille minimale** du composant pour qu'il soit encore utilisable. La taille minimale, minimumSize, d'un composant peut être limitée, par exemple, par la taille d'un libellé. Pour la plupart des contrôles, minimumSize est identique à preferredSize. Les gestionnaires de disposition respectent généralement davantage minimumSize que preferredSize.

- **`public Dimension getMaximumSize();`**
  La **taille maximale** du composant pour qu'il soit encore utilisable. Elle sert à empêcher le gestionnaire de disposition d'attribuer trop d'espace à un composant qui ne l'utilisera pas de façon efficace et à le donner, à la place, à un composant qui est seulement à sa taille minimale, `minimumSize`. Par exemple, `BorderLayout` limite la taille du composant central à sa taille maximale, puis attribue la place aux composants de côté ou limite la taille de la fenêtre externe lors de son redimensionnement.

  Pour spécifier la taille minimale ou la taille maximale d'un composant, le programmeur a deux possibilités:

  1. envoyer les messages **`setMinimumSize`** ou **`setMaximumSize`** au composant
  2. ou redéfinir les méthodes **`getMinimumSize`** et **`getMaximumSize`**

  Comme pour PreferredSize, la dernière méthode est préférée.

- **`getAlignmentX()`**
  L'alignement horizontal que voudrait le composant, par rapport aux autres composants.

- **`getAlignmentY()`**
  L'alignement vertical que voudrait le composant, par rapport aux autres composants.

2.4 TAILLE ET EMPLACEMENT DE L'INTERFACE UTILISATEUR

Si votre classe interface utilisateur est un descendant de java.awt.Window (comme la clas-

se **JFrame**), vous pouvez contrôler sa taille et son emplacement à l'exécution. La taille et l'emplacement de la fenêtre interface utilisateur sont déterminés à la fois par le code lors de sa création et par l'utilisateur qui la redimensionne ou la déplace.

Lorsque la fenêtre interface utilisateur est créée, et que divers composants lui sont ajoutés, chaque composant ajouté affecte la taille `preferredSize` de la fenêtre globale, et généralement la `preferredSize` du conteneur fenêtre est agrandie au fur et à mesure que les composants sont ajoutés. L'effet exact sur la `preferredSize` dépend du gestionnaire de disposition du conteneur externe, ainsi que des dispositions des conteneurs imbriqués. Pour plus de détails sur la façon dont est calculée `preferredLayoutSize` pour les diverses dispositions, voir les sections de ce document relatives à chaque type de disposition.

La **taille** de la fenêtre interface utilisateur, telle qu'elle est définie par votre programme (avant toute intervention de l'utilisateur), est déterminée par la méthode qui est appelée en dernier dans le code, parmi les deux suivantes:

- **`pack()`**
- **`setSize(`*`largeur, hauteur`*`)`**
- **`setLocation(int x, int y)`**
  L'emplacement de votre interface utilisateur à l'exécution sera 0,0 jusqu'à ce que vous définissiez la valeur de la propriété location du conteneur (par exemple en appelant setLocation(..) avant que l'interface ne soit affichée).

2.4.1 Dimensionnement automatique d'une fenêtre avec `pack()`

Quand vous appelez la méthode **`pack`**`()` sur une fenêtre, vous lui demandez de calculer sa taille `preferredSize`, en fonction des composants qu'elle contient, puis de se redimensionner elle-même en adoptant cette taille. Cela a généralement pour effet de la rendre la plus petite possible tout en respectant la taille `preferredSize` des composants contenus.

Vous pouvez appeler la méthode `pack()` pour donner automatiquement à la fenêtre une taille minimale en préservant l'aspect de tous ses contrôles et sous-conteneurs.

2.4.2 Calcul de la taille préférée d'un conteneur

La taille `preferredSize` est calculée différemment pour des conteneurs ayant différentes dispositions.

Dispositions portables

Les dispositions portables, telles `FlowLayout` et `BorderLayout`, calculent leur preferredSize en fonction des règles de disposition et de la taille `preferredSize` de chaque composant ajouté au conteneur. Si certains de ces composants sont eux-mêmes des conteneurs (un composant Panel, par exemple), alors la taille `preferredSize` de chacun d'eux est calculée en fonction de sa disposition et de ses composants, un calcul récurrent

s'effectuant dans toutes les couches d'imbrication des conteneurs, si nécessaire.

Pour plus d'informations sur le calcul de la taille `preferredSize` selon les dispositions, voir la description de chaque disposition.

2.4.3 Définition explicite de la taille d'une fenêtre en utilisant `setSize()`

Si vous appelez **`setSize()`** sur le conteneur (au lieu de **`pack`**() ou après un appel à **`pack()`**), alors la taille du conteneur sera définie par une valeur explicite, en pixels.

Important

N'oubliez pas que même s'il est possible de définir la taille de votre conteneur par une largeur et une hauteur données, cela diminue la portabilité de votre interface utilisateur, puisque la taille des pixels diffère selon les écrans.

`validate()`

Si vous définissez la taille explicitement en utilisant `setSize`() de manière dynamique, - alors par exemple que la fenêtre est déjà affichée -, vous devez appeler **`validate()`** pour que les enfants puissent être correctement disposés (le **`validate()`** revient à opérer une espèce de rafraîchissement).

Notez que **`pack()`** appelle **`validate()`**.

2.4.4 Portabilité de l'interface utilisateur

Généralement, si vous voulez que votre interface utilisateur soit portable, vous devez soit utiliser `pack()` et pas `setSize()`, soit prendre en compte les tailles des pixels des différents écrans et évaluer raisonnablement la taille à définir.

Par exemple, vous pouvez décider que, au lieu d'appeler `pack()`, vous afficherez l'interface utilisateur de sorte qu'elle occupe toujours 75 % de la largeur et de la hauteur de l'écran. Pour ce faire, ajoutez les lignes de code suivantes à votre application, à la place de l'appel à `pack()`.

```
Dimension screenSize =
   Toolkit.getDefaultToolkit().getScreenSize();

frame.setSize( screenSize.width * 3 / 4,
screenSize.height * 3 / 4);
```

2.4.5 Positionnement d'une fenêtre au milieu de l'écran

Si vous ne donnez pas explicitement la position à l'écran de l'interface utilisateur, elle apparaîtra dans le coin supérieur gauche. C'est parfois bien mieux de centrer l'interface utilisateur. Pour cela, il faut connaître la largeur et la hauteur de l'écran, soustraire respectivement la largeur et la hauteur de votre interface utilisateur, diviser par deux la différence (afin de créer des marges égales pour les côtés opposés de l'interface utilisateur), et utiliser ces valeurs pour désigner l'emplacement du coin supérieur gauche de l'interface utilisateur.

```
//Centrer la fenêtre
Dimension screenSize =
      Toolkit.getDefaultToolkit().getScreenSize();

Dimension frameSize = frame.getSize();

if (frameSize.height > screenSize.height)
   frameSize.height = screenSize.height;

if (frameSize.width > screenSize.width)
   frameSize.width = screenSize.width;

frame.setLocation((screenSize.width - frameSize.width)/2,
                  (screenSize.height - frameSize.height) /2);
```

2.5 LES PRINCIPAUX GESTIONNAIRES

Voici les principaux gestionnaires de disposition mis à disposition par Swing et l'awt:

- `BorderLayout [awt]`
- `FlowLayout[awt]`
- `GridLayout [awt]`
- `BoxLayout [swing]`
- `GridBagLayout [awt]`
- `null`

Vous pouvez créer des dispositions personnalisées, ou essayer d'autres gestionnaires de disposition, comme ceux de Sun ou ceux d'autres fournisseurs. Un grand nombre d'entre eux se trouvent sur le Web et appartiennent au domaine public.

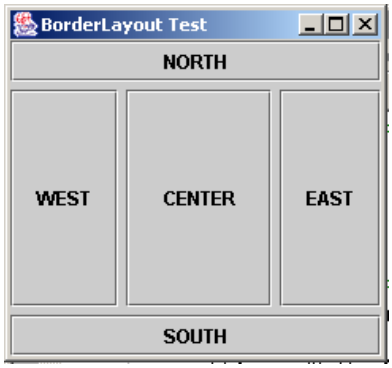
La plupart des interfaces utilisateur seront constituées de différentes dispositions, sous forme de panneaux imbriqués les uns dans les autres.

2.5.1 BorderLayout

Le gestionnaire `BorderLayout` organise les composants d'un conteneur en diverses zones, appelées `NORTH`, `SOUTH`, `EAST`, `WEST` et `CENTER`.

- Les composants situés dans les zones NORTH et SOUTH adoptent leur hauteur préférée (`preferredSize`) et s'étalent sur toute la largeur du conteneur.
- Les composants situés dans `EAST` et `WEST` adoptent leur largeur préférée et s'étalent verticalement sur tout l'espace restant entre les zones nord et sud.
- Un composant situé dans `CENTER` remplit tout l'espace restant.

Un exemple avec 5 boutons (`JButton`):



BorderLayout convient pour obliger les composants à se placer le long des côtés d'un conteneur et pour remplir le centre du conteneur par un composant . **C'est la disposition que l'on utilisera très fréquemment pour qu'un seul composant occupe toute la surface d'un conteneur.**

Vous trouverez probablement que BorderLayout est le gestionnaire de disposition le plus pratique pour les plus grands conteneurs de votre interface utilisateur. En imbriquant un panneau dans chaque zone du conteneur BorderLayout, puis en remplissant chacun de ces panneaux avec d'autres panneaux de diverses dispositions, vous pourrez concevoir les interfaces utilisateur les plus complexes.

Associer un gestionnaire de type `BorderLayout`

- `setLayout(new BorderLayout());`
  Constructeur par défaut : les composants sont placés dans chaque en occupant la place disponible, il n'y a pas de marge précue entre les composants.
- `setLayout(new BorderLayout(hgap, vgap));`
  BorderLayout possède des propriétés appelées `hgap` (espace horizontal) et `vgap` (espace vertical) qui déterminent la distance entre les composants, et

qu'il est possible de spécifier au moment de la création du BorderLayout:

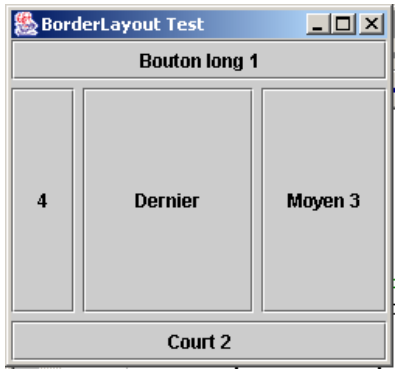
- **Pour ajouter des composants**

Un composant est placé dans une des cinq zones d'un BorderLayout, en spécifiant sa position: `NORTH, SOUTH,..`

Par exemple:

```
leContainer.add (unComposant, BorderLayout.NORTH);
```

- **Exemple**




```
class BorderLayoutTest extends JPanel {

    public BorderLayoutTest() {
        // Spécifier un gestionnaire FlowLayout
        setLayout(new BorderLayout(5, 5));

        String [] etiquettes = {"Bouton long 1", "Court 2",
                                "Moyen 3", "4", "Dernier"};

        add (new JButton(etiquettes[0]), BorderLayout.NORTH);
        add (new JButton(etiquettes[1]), BorderLayout.SOUTH);
        add (new JButton(etiquettes[2]), BorderLayout.EAST);
        add (new JButton(etiquettes[3]), BorderLayout.WEST);
        add (new JButton(etiquettes[4]), BorderLayout.CENTER);
    }
}
```

2.5.2 FlowLayout

Le gestionnaire FlowLayout organise les composants en lignes, de gauche à droite, puis de haut en bas, en utilisant la taille préférée de chaque composant (preferredSize): la taille prévue pour les composants est donc respectée par ce gestionnaire !!

FlowLayout place en ligne autant de composants que possible, avant de recommencer sur une nouvelle ligne. Généralement, FlowLayout est utilisé pour disposer des boutons sur un panneau.

Associer un gestionnaire de type `FlowLayout`

- `setLayout(new FlowLayout());`
  Constructeur par défaut : les composants sont centrés
- `setLayout(new FlowLayout(FlowLayout.LEFT));`
  Pour spécifier le type d'alignement: **`LEFT`**, **`RIGHT`** ou **`CENTER`** (centré par défaut)
- `setLayout(new FlowLayout(`*`align, hgap, vgap`*`));`
  Pour spécifier en plus la valeur des espacements entre composants (valant `5` par défaut)

Pour créer par exemple un `FlowLayout` aligné à gauche avec des espacements de 10 pixels horizontalement et verticalement:

```
new FlowLayout (FlowLayout.LEFT, 10, 10);
```

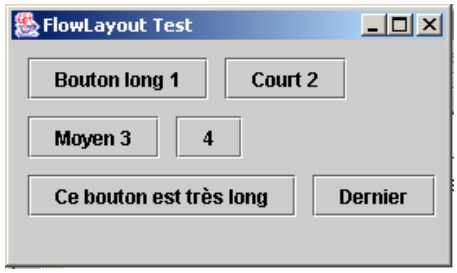
Pour ajouter des composants

Un composant est placé dans un `FlowLayout` en utilisant simplement la méthode **`add`**. Les composants sont placés sont l'ordre, de gauche à droite, en allant à la ligne suivante si nécessaire.

Par exemple:

leContainer`.add (`*unComposant*`);`

- **Exemple**




```
class FlowLayoutTest extends JPanel {

    public FlowLayoutTest() {
        // Spécifier un gestionnaire FlowLayout
        setLayout(new FlowLayout(FlowLayout.LEFT, 10, 10));

        String [] etiquettes = {"Bouton long 1", "Court 2",
                                "Moyen 3", "4",
                                "Ce bouton est très long", "Dernier"};

        for (int i = 0; i < etiquettes.length ; i++) {
            add (new JButton(etiquettes[i]));
        }
    }
}
```

2.5.3 GridLayout

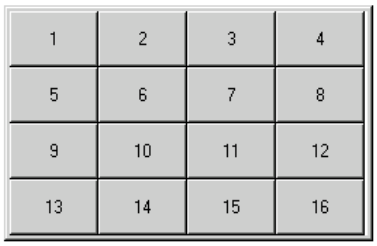
Le gestionnaire `GridLayout` place les composants sur une grille de cellules en lignes et en colonnes.

`GridLayout` agrandit chaque composant de façon qu'il remplisse l'espace disponible dans la cellule. Toutes les cellules sont exactement de même taille et la grille est uniforme.

Le «`PreferredSize`» d'un `GridLayout` est calculé en fonction du `PreferredSize` des composants les plus grands.

Lorsque vous redimensionnez un conteneur `GridLayout`, `GridLayout` change la taille des cellules de sorte qu'elles soient aussi grandes que possible étant donné l'espace disponible dans le conteneur.

Exemple:



Utilisez **`GridLayout`** si vous concevez un conteneur dont tous les composants doivent avoir la même taille, par exemple un pavé numérique ou une barre d'outils.

Vous pouvez spécifier le nombre de colonnes et le nombre de lignes de la grille. Pour `GridLayout`, le nombre de lignes ou le nombre de colonnes peut être nul, mais pas les deux à la fois. Vous devez indiquer au moins une valeur pour que le gestionnaire de `GridLayout` puisse calculer la seconde.

Par exemple, si vous spécifiez quatre colonnes et zéro ligne pour une grille de 15 composants, `GridLayout` crée quatre colonnes de 4 lignes, la dernière ligne ne contenant que trois composants.

Ou, si vous spécifiez trois lignes et zéro colonne, `GridLayout` crée trois lignes et cinq colonnes complètement remplies.

Espacement vertical et horizontal des composants

Outre le nombre de lignes et de colonnes, vous pouvez spécifier le nombre de pixels **entre les cellules** en utilisant les paramètres d'espace horizontal (`hgap`) et d'espace vertical (`vgap`). Ceci ne concerne pas les marges qui resteront nulles de toute manière. L'espace horizontal et l'espace vertical sont nuls par défaut.

Associer un gestionnaire de type `GridLayout`

- `setLayout (new GridLayout(`*lignes*`, `*colonnes*`);`
- `setLayout (new GridLayout(`*lignes*`, `*colonnes*`,`
  hgap`, `*vgap*`);`

Pour créer par exemple un `GridLayout` de 5 lignes, 8 colonnes, et des espacements horizontaux et verticaux de valeurs respectives 10 et 20:

```
unConteneur.setLayout (new GridLayout (5, 8, 10, 20));
```

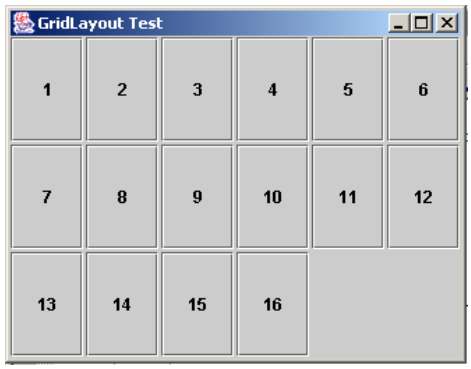
Pour ajouter des composants

Un composant est placé dans un `GridLayout` en utilisant simplement la méthode **add**. Les composants sont placés sont l'ordre, ligne par ligne, en remplissant chaque ligne.

Par exemple:

leContainer`.add (`*unComposant*`);`

Exemple




```
class GridLayoutTest extends JPanel {

       public GridLayoutTest() {

       // Spécifier un gestionnaire GridLayout .
       setLayout(new GridLayout(3, 6, 2, 2));

       for (int i = 1; i <= 16 ; i++) {
           add (new JButton(i+""));
       }
}
```

2.5.4 BoxLayout

Le gestionnaire de disposition «BoyLayout» est prévu pour aligner les composants selon un axe horizontal ou un axe vertical (que l'on désignera plus bas comme étant «**l'axe primaire**», l'autre axe étant désigné par le terme «**axe complémentaire**»). Par exemple, si on choisit l'axe vertical comme axe primaire, les composants seront placés de haut en bas, en respectant l'ordre d'addition des composants.

Arrangement selon l'axe primaire

Au contraire du gestionnaire GridLayout, ce gestionnaire permet à chaque composant d'occuper un espace spécifique selon l'axe primaire. Par exemple, si l'on choisit l'axe vertical comme axe primaire, un `JTextField` occupera moins de place verticalement qu'un `JTextArea`. Cet espace vertical sera déterminé par envoi du message `getPreferredSize()` au composant.

Arrangement selon l'axe complémentaire

S'il s'agit d'un BoxLayout horizontal, le gestionnaire tentera de rendre chaque composant aussi haut que le plus haut parmi tous les composants.

Au contraire, s'il s'agit d'un BoxLayout vertical, le gestionnaire tentera de rendre chaque composant aussi large que le large parmi tous les composants.

Si le composant ne peut s'accroître de cette manière (getMaximumSize()), le gestionnaire appliquera les propriétés Y-alignment ou X-alignmentdu composant afin de déterminer comment placer le composant au sein de l'espace disponible.

 Alignement des composants

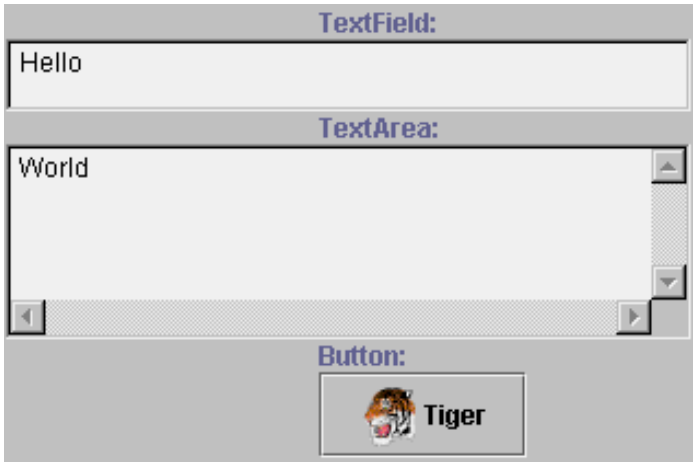
Notons à ce sujet que les composants de type `JComponent` héritent d'une propriété d'alignement valant `0.5`, indiquant par là qu'ils seront centrés. Mais cette propriété peut être redéfinie par le composant: il lui suffit de redéfinir les méthodes **`getAlignmentX()`** et **`getAlignmentY()`**héritées de la classe `Container`, qui retournent une valeur comprise entre `0.0` et `1.0`, où `0.0` signifie un alignement à gauche, et 1.0 un alignement à droite.

- **Associer un gestionnaire de type BoxLayout**
  - **`setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));`**
    Le premier paramètre correspond au conteneur (on écrira «this» en général), et le second indique l'axe primaire: `X_AXIS ou Y_AXIS`

- **Pour ajouter des composants**

  Il suffit d'envoyer le message **`add`**`(`*`unComposant`*`)au conteneur`

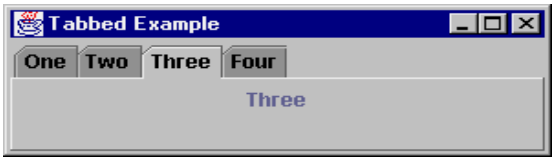
- **Exemple**




```
class BoxLayoutTest extends JPanel {

    public BoxLayoutTest() {

        // Spécifier un gestionnaire BoxLayout selon l'axe primaire vertical.
        setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));

        // Création des 3 composants
        TextField textField = new TextField();
        TextArea textArea = new TextArea(4, 20);
        JButton button = new JButton(
                             "Tiger",
                             new ImageIcon("SmallTiger.gif"));

        // Add the three components to the BoxLayout
        add(new JLabel("TextField:"));
        add(textField);
        add(new JLabel("TextArea:"));
        add(textArea);
        add(new JLabel("Button:"));
        add(button);
}}
```

2.5.5 JTabbedPane (panneaux à onglets)

`Le JTabbedPane` empile les composants (généralement des panneaux) les uns sur les autres, comme dans un jeu de cartes. Vous n'en voyez qu'un à la fois, et vous pouvez feuilleter les panneaux à l'aide des onglets mis à votre disposition.



Création d'un JTabbedPane

- **`JTabbedPane`**`()`
- **`JTabbedPane`**`(int tabPlacement)`

  Pour créer un panneau à onglets avec un emplacement spécifique des onglets. Pour plus de détails, voir le 3ème constructeur.
- **`JTabbedPane`**`(int tabPlacement, int tabLayoutPolicy)`

  Pour créer un panneau à onglets avec un emplacement spécifique des onglets et une politique de disposition des onglets.

  L'emplacement des onglets peut prendre l'une ou l'autre des valeurs suivantes: `JTabbedPane.`**`TOP`**, `JTabbedPane.`**`BOTTOM`**, `JTabbedPane.`**`LEFT`**, ou `JTabbedPane.`**`RIGHT`**. Par défaut (1er constructeur): `JTabbedPane.`**`TOP`**

  La politique de disposition des onglets peut prendre l'une ou l'autre des valeurs suivantes: `JTabbedPane.`**`WRAP_TAB_LAYOUT`** ou `JTabbed-Pane.`**`SCROLL_TAB_LAYOUT`**. Cette politique est mise en oeuvre dès l'instant qu'il n'y a pas assez de place pour afficher les onglets sur une seule ligne ou une seule verticale. Par défaut (1er et 2ème constructeur): **`TOP`** et **`WRAP_TAB_LAYOUT.`**

Ajouter une «carte» à un TabbedPane

La classe met à disposition la méthode addTab, qui existe sous 3 formes différentes.

- **`addTab`**`(String title, Component component)`
  Pour ajouter une nouvelle carte en spécifiant son étiquette (un texte) et le composant qui sera affiché une fois sélectionné.
- **`addTab`**`(String title, Icon icon, Component component)`
  Permet de spécifier une image de type Icon qui serait associée au texte de l'étiquette. Le texte, tout comme l'image, peuvent prendre la valeur `null`.
- ```
  addTab(String title,
         Icon icon,
         Component component,
         String tip)
  ```
  Permet en plus de spécifier un «tooltip» (texte d'aide fugitif) qui serait associé à l'onglet.
- ```
  insertTab(String title,
            Icon icon,
            Component component,
            String tip,
            int index)
  ```
  Pour insérer une carte, qui occupera la position indiquée par l'index, passé en paramètre.

Pour retirer des cartes

- **`remove`**`(int index)`

Pour supprimer l'onglet correspond à l'indice passé en paramètre.

- **`removeAll()`**
  Pour supprimer tous les onglets.

Un exemple


```
public class TabbedPanel extends JPanel {
    String tabs[] = {"One", "Two", "Three", "Four"};

    public JTabbedPane tabbedPane = new JTabbedPane();

    public TabbedPanel() {
        //Conteneur associé à un gestionnaire de disposition de type BorderLayout
        setLayout (new BorderLayout());


        // Création des 4 onglets
        for (int i=0; i<tabs.length; i++) {
            // Ajout d'un panneau, créé au moyen de la méthode
            // «createPane», spécifiée un peu plus bas
            tabbedPane.addTab(tabs[i], null, createPane(tabs[i]));
        }
        // Sélection d'office du premier panneau
        tabbedPane.setSelectedIndex(0);


        // Ajout du panneau à onglets au centre du conteneur
        add (tabbedPane, BorderLayout.CENTER);
    }


    private JPanel createPane(String s) {
    // Retourne un nouveau panneau muni d'un label initialisé avec le texte «s»
        JPanel p = new JPanel();
        p.add(new JLabel(s));
        return p;
    }
}
```

Evénements spécifiques

- **`addChangeListener`**`(ChangeListener l)`
  Pour inscrire un écouteur de «changement»

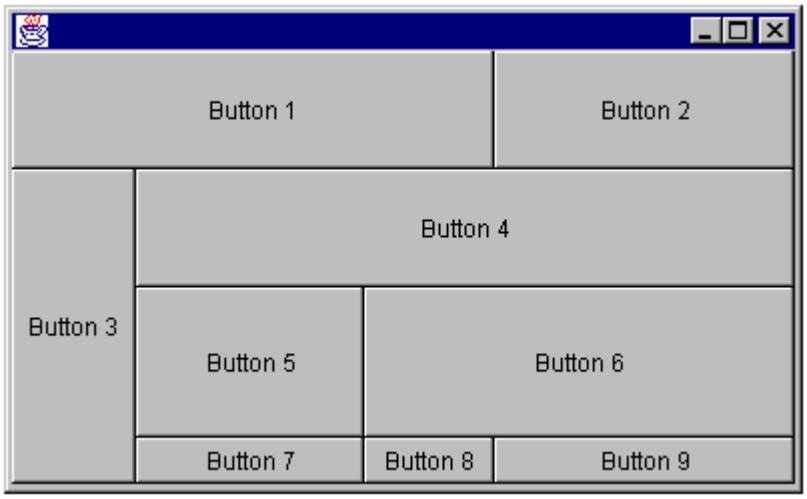
- **`removeChangeListener`**`(ChangeListener l)`

Pour retirer un écouteur de «changement»

2.5.6 GridBagLayout

`GridBagLayout` est une disposition extrêmement souple et puissante qui permet un meilleur contrôle du positionnement des composants sur la grille que celui fourni par GridLayout. GridBagLayout positionne les composants horizontalement et verticalement sur une grille rectangulaire. Il n'est pas nécessaire que les composants soient de même taille, chacun pouvant remplir plusieurs cellules.

Exemple:



`GridBagLayout` détermine l'emplacement des composants qu'il contient en se basant sur les contraintes (constraints) et la taille minimum de chaque composant, ainsi que sur la taille préférée (preferred size) du conteneur.

Bien que `GridBagLayout` puisse supporter des grilles complexes, il se comportera mieux (de manière plus prévisible) si vous avez organisé vos composants en panneaux plus petits et imbriqués dans le conteneur `GridBagLayout`. Ces panneaux imbriqués peuvent utiliser d'autres dispositions, et contenir d'autres panneaux de composants si nécessaire.

Cette méthode présente deux avantages :

- Elle permet un contrôle plus précis de l'emplacement et de la taille des composants individuels car vous pouvez utiliser les dispositions les plus appropriées à chaque zone, par exemple à une barre de boutons
- Elle utilise moins de cellules en simplifiant l'utilisation de GridBagLayout et en facilitant son contrôle.

2.5.7 Null

La disposition Null signifie qu'aucun gestionnaire de disposition n'est assigné au conteneur. Vous pouvez placer les composants dans le conteneur à des coordonnées spécifiques, x et y, relatives au coin supérieur gauche du conteneur.

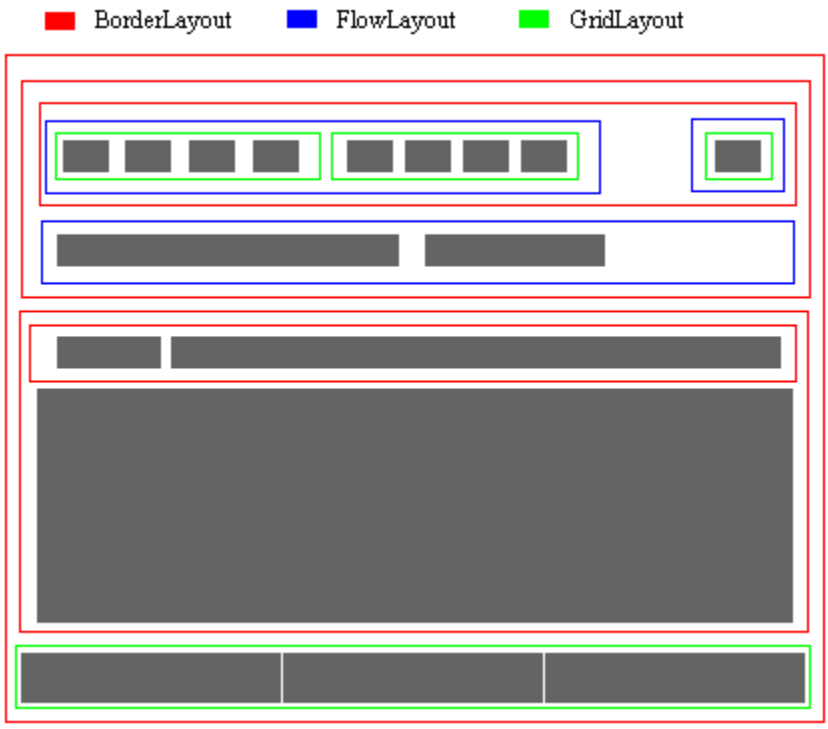
Par exemple:

```
setLayout(null);      // Pas de gestionnaire de disposition
JButton b = new JButton("OK");// Un composant ..
b.setBounds (x, y, largeur, hauteur);// Pour spécifier la position
                                     //  et la taille du composant
b.setLocation (x, y);       // Pour spécifier uniquement la position
```

2.6 UTILISATION DE PANNEAUX IMBRIQUÉS

La plupart des interfaces conçues en Java utilisent plusieurs types de dispositions pour arriver au résultat souhaité, en imbriquant plusieurs panneaux de dispositions différentes dans le conteneur principal. Vous choisirez également d'imbriquer des panneaux dans d'autres panneaux pour mieux contrôler la position des composants. En créant une conception composite utilisant le gestionnaire de disposition approprié à chaque panneau, vous pouvez regrouper les composants pour améliorer leur utilisation et assurer la portabilité.

Par exemple, le schéma qui suit montre l'utilisation de panneaux imbriqués avec différentes dispositions. Les objets en gris représentent des boutons ou d'autres composants visibles dans l'interface utilisateur imbriqués dans divers niveaux de panneaux.

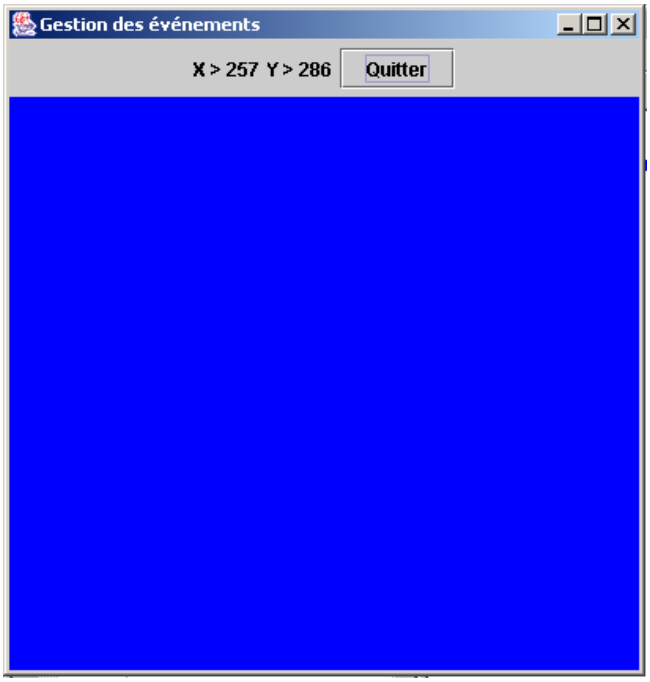


3 La gestion des événements

La mécanique décrite dans ce document correspond au modèle introduit dans Java 1.1, modèle sur lequel s'appuie la librairie Swing.

Le mécanisme de gestion des événements, basé sur le modèle des ***listeners***, sera présenté au travers d'un petit exemple dont la fenêtre d'exécution comprendrait les éléments suivants:

- Au sud: une zone graphique (surface d'un `JPanel`);
- Au nord: une barre d'outils composée du bouton «Quitter» et de deux labels qui affichent en temps réel la position courante de la souris dans la zone graphique.



Dans cet exemple, il nous faudra gérer deux types d'événements: le déplacement de la souris (événements de type `MouseMotion`) et l'actionnement d'un bouton (événement de type `Action`).

Voici le code du programme qui, dans une première étape, se limite à l'affichage de la fenêtre, sans effectuer aucune gestion d'événement.

Le programme, version limitée à l'affichage

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

//-----------------------------------------------------------------------------
interface Configuration {
    public static int TAILLE_X = 400;          // Taille fenêtre en X
    public static int TAILLE_Y = 400;          // Taille fenêtre en Y
}
//-----------------------------------------------------------------------------

class PanneauPrincipal extends JPanel
                       implements Configuration {
// Variables d'instance
    private JPanel pNord = new JPanel();
    private JButton bQuit = new JButton("Quitter");
    private JLabel lX = new JLabel("X > 000");
    private JLabel lY = new JLabel("Y > 000");

// Constructeur
    public PanneauPrincipal() {
           setLayout(new BorderLayout());
           add(pNord, BorderLayout.NORTH);
           pNord.add (lX);
           pNord.add (lY);
           pNord.add (bQuit);

           setBackground (Color.BLUE);
           setPreferredSize
              (new Dimension(TAILLE_X, TAILLE_Y));

    }

//-----------------------------------------------------------------------------

public class Amorce {
    public static void main (String [] args) {
        JFrame f = new JFrame ("Gestion des ÈvÈnements");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // L'application se termine lorsque la fenÍtre est fermÈe

        // Mise en place du panneau principal au centre de la fenÍtre
        f.getContentPane().add(new PanneauPrincipal(),
                               BorderLayout.CENTER);
        f.pack();
        f.setVisible(true);
    }
}
```

3.1 GESTION DES ÉVÉNEMENTS: PRINCIPE

3.1.1 Le événements

Les ***événements*** auxquels nous nous intéressons correspondent à des **changements d'état** subits par les objets qui composent l'interface graphique avec l'utilisateur (menus, boutons, comboBox, ..).

En voici quelques exemples:

- «le bouton x a été actionné»;
- «le menu y a été sélectionné»;
- «la souris a été déplacée en position 30, 40»;
- ..

3.1.2 Les sources d'événements

Les événements que nous venons de décrire sont occasionnés par l'utilisateur. Par abus de langage, pour simplifier le discours, on dira que ces événements sont **générés** par les composants graphiques eux-mêmes. Pour cette raison, ces objets sont appelés des ***sources d'événements***.

Par exemple, les objets de type `JButton` sont susceptibles de générer les événements suivants:

`Action`	Lorsque le bouton a été actionné par l'utilisateur (appuyé et relâché)
`MouseMotion`	Lorsque la souris est déplacée ou draguée sur la surface du bouton
`Mouse`	Lorsque le bouton de la souris est appuyé, relâché, ou clické, ou encore lorsque la souris sort ou entre dans la surface du bouton.
`Focus`	Lorsque le bouton obtient ou perd le focus (suite par exemple à la pression de la touche TAB)
`Component`	Lorsque le bouton a été déplacé ou caché
`Key`	Lorsqu'une touche du clavier est pressée alors que le bouton posséde le focus.

3.1.3 Les écouteurs

De manière générale, lorsqu'ils sont déclenchés par une action de l'utilisateur, ces événements sont ignorés, perdus dans la nature. A moins toutefois qu'un objet ait décidé de s'y intéresser en s'y **abonnant** de manière explicite.

Par exemple, pour s'abonner auprès d'un bouton à un événement de type `Action`, il suffira à un objet «*objetX*» de s'enregistrer en tant qu'auditeur pour ce type d'événement:

```
leBouton.addActionListener(objetX);
```

Si le même objet désire s'abonner aux événements de type `Component`, il le fera ainsi:

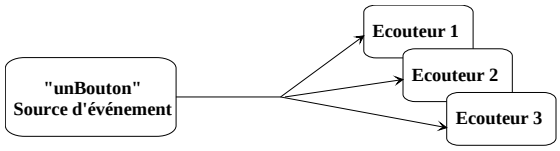
```
leBouton.addComponentListener(objetX);
```

Un tel objet est appelé **auditeur**, ou encore **écouteur.**

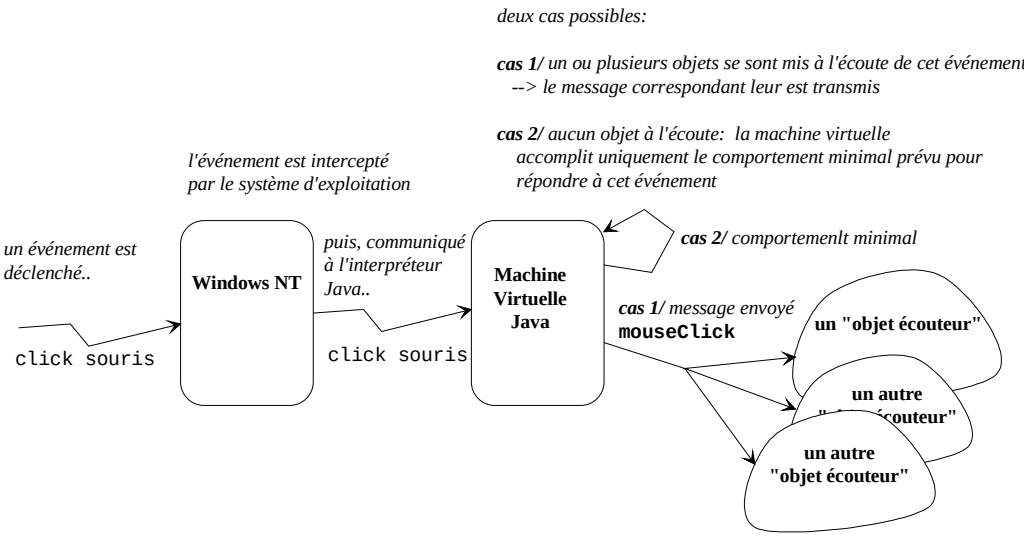
Un objet source d'événements peut être écouté par plusieurs écouteurs !

Lorsqu'un événement est déclenché, le **message** correspondant sera **diffusé** par l'objet source à tous les écouteurs qui se seront inscrits au préalable pour le type d'événement concerné. Si aucun écouteur ne s'est inscrit, l'événement sera simplement ignoré, sans occasionner aucune perte de temps.

Par exemple, l'actionnement du bouton `unBouton` provoquera la diffusion du message `actionPerformed` à tous ses écouteurs.



En résumé, on peut représenter cette mécanique à l'aide de la figure suivante:



3.1.4 Le gestionnaire d'événement

Reprenons notre exemple, et supposons que le bouton soit actionné. L'écouteur (`objetX`) en sera notifié en recevant le message `actionPerformed`, avec l'événement passé en paramètre. La classe de l'écouteur (`objetX`) devra donc implémenter la méthode correspondante, que l'on appellera un **gestionnaire d'événement**. Cette méthode contiendra les instructions qui seront exécutées pour traiter l'événement:

```
public void actionPerformed (ActionEvent e) {
   /* instructions pour traiter l'événement */
}
```

3.1.5 Les types d'événements

Le gestionnaire d'événement reçoit un paramètre, l'événement lui-même, un objet qui est propagé de la source vers le destinataire. Ainsi, le traitement de l'événement est **délégué au destinataire** (d'où le nom donné parfois au concept: le modèle «par délégation»).

Les événements sont des instances de classes spécifiques que l'on appelle des ***types d'événements.***

Par exemple, la pression d'un bouton va générer un événement de type `ActionEvent`.

Un type d'événement correspond à une ***catégorie d'événements***, et regroupe en général plusieurs événements spécifiques, qui correspondront chacun à un message particulier qui sera envoyé à l'auditeur.

Par exemple, le type d'événement `MouseMotionEvent` regroupe deux événements spécifiques:

- `mouseMoved`: la souris a été déplacée;
- `mouseDragged`: la souris a été draguée.

Notons, et nous y reviendrons un peu plus loin, qu'un auditeur devra s'abonner à **toute une catégorie d'événements à la fois**, même si certains événements de cette catégorie ne l'intéressent pas. Il devra ainsi prévoir un gestionnaire d'événements pour chacun des événements de la catégorie.

Voici la liste des différentes catégories d'événements, présentée en deux tableaux:

1. ***Les événements «de base»***
   Ces événements sont générés par tous les composants de type **`Component`**, notamment **par tous les composants Swing** (qui héritent indirectement de la classe `Component`).
2. ***Les événements «sémantiques»***
   Ces événements sont dits «sémantiques» en raison du fait qu'ils représentent une action logique de l'utilisateur, de haut-niveau, dont la réalisation correspond à une séquence spécifique d'éléments primitifs «de base» (voir le tableau précédent). Par exemple, un événement **`Action`**, - qui, pour un bouton, modélise le fait qu'il a été appuyé -, est déclenché après que la souris ait été successi-

vement pressée puis relâchée sur le même bouton (deux événements de bas niveau).

A chaque catégorie d'événement correspond une classe de même nom, terminé par le mot **`Event`**, comme par exemple `Action`**`Event`** pour la catégorie `Action`. Pour obtenir plus d'informations sur l'événement, il est recommandé de consulter la documentation de la classe correspondante, directement dans l'API du JDK.

Dans les tableaux qui suivent, les événements dont l'explication commence par [awt] sont issus historiquement de l'awt, la première librairie graphique de Java. Les autres appartiennent exclusivement à la librairie Swing (à partir de Java 1.2).

Tableau 1: Voici les événements dits «de base»

`Mouse`	[awt] Evénements associés à la souris : click, bouton enfoncé, bouton relâché, souris entrée dans le composant, ou encore souris sortie du composant.
`MouseMotion`	[awt] Evénements relatifs au déplacement de la souris: move ou drag, générés en flot continu.
`MouseWheel`	[awt] Evénements relatifs à la rotation de la roue centrale d'une souris.
`Key`	[awt] Evénements associés au clavier : touche tapée, enfoncée ou relâchée alors que le composant concerné possède le focus.
`Component`	[awt] Changement dans la visibilité du composant (caché, affiché), dans la position du composant, ou encore dans la taille du composant.
`Container`	[awt] Un composant a été rajouté ou retiré du container
`Focus`	[awt] Les entrées du clavier et de la souris sont destinées au composant qui possède le focus. Les événements de type «`Focus`» correspondent soit à un gain, soit à une perte de focus. Le focus peut être modifié si le composant est «cliqué», si la touche TAB a été pressée, ou encore si la fenêtre qui contient le composant a été désactivée ou activée. Une fenêtre qui possède le «focus» est une fenêtre «en premier plan» a qui sont envoyés tous les caractéres tapés au clavier.
`Window`	[awt] Modification dans le statut de la fenêtre : activée, désactivée, icônifiée, désicônifiée, fermée ou ouverte.

Notons que les événements de type **`Container`** n'ont de sens qu'auprès des composants Swing qui jouent le rôle de conteneur: **`JFrame`**, **`JPanel`**, **`JTabbedPane`**, etc.. ou de composite, comme par exemple **`JComboBox`**.

Tableau 2: Evénements dits «sémantiques»

`Action`	[awt] Evénement fréquemment récupéré. Il existe différentes sources: ❍ Un bouton est actionné (**`JButton`**, **`JRadioButton`**, **`JCheckBox`**); ❍ Un élément est sélectionné dans un **`JComboBox`**; ❍ La touche <Return> est pressée dans un **`JTextField;`** ❍ Un fichier est sélectionné dans un **`JFileChooser`**; ❍ Un élément de menu est sélectionné (**`JMenu`**, **`JMenuItem`**, **`JRadioButtonMenuItem`**, **`JCheckBoxMenuItem`**)
`Adjustment`	[awt] Nouvelle valeur sélectionnée dans un composant de type «**`JScrollBar`**»
`AncestorEvent`	*Ajout, déplacement, ou suppression d'un ancêtre. Evénements générés par tous les composants de type **`JComponent`**. Ayant trait à l'implémentation, ce type d'événement ne concerne pas le programmeur ordinaire.*
`CaretEvent`	Modification de la position du curseur, pour composants de type **`JTextComponent`**. Pour connaître la position actuelle du curseur, pour connaître l'étendue d'une sélection opérée par l'utilisateur au sein d'un texte.
`ChangeEvent`	Changement d'état en général, relatif aux composants suivants: ❍ Boutons et boîtes à cocher: **`JButton`**, **`JRadioButton`**, **`JRadioButtonMenuItem`**, **`JCheckBoxMenuItem`** ❍ **`JMenuItem`** (élément de menu sélectionné) ❍ **`JProgressBar`** (déplacement du curseur) ❍ **`JSlider`** (déplacement du curseur) ❍ **`JTabbedPane`** (sélection d'un nouveau panneau au sein d'un panneau à onglets) ❍ **`JViewPort`** ❍ **`JTable`** & **`JTree:`** modification opérée dans la cellule d'édition d'une cellule de table, resp. du noeud d'un arbre
`DocumentEvent`	Modification des attributs d'un document, insertion de texte, suppression de texte. Ces événements sont générés par des composants de type **`Document`**, le modèle associé à un **`JTextComponent`**.
`HyperlinkEvent`	Concerne les composants d'édition de type **`JEditorPane`**: activation d'un hyperlien, entrée ou sortie dans un hyperlien.
`InternalFrameEvent`	Activation, désactivation, fermeture, en phase de fermeture, iconification, désiconification, ou encore ouverture d'une fenêtre. Pour composants de type **`JInternalFrame`**.
`Item`	[awt] Nouvel élément sélectionné dans une liste de type **`JComboBox`**, ou sélection d'une case à cocher (**`JCheckBox`**).
`ListDataEvent`	Modification du contenu, ajout ou suppression d'un groupe de valeurs au sein du modèle associé à un **`JList`**.

ListSelectionEvent	Modification d'état de la sélection au sein d'un **JList**.
MenuDragMouseEvent	Concerne les **JMenuItem** de type «Menu drag»: souris draguée, entrée, sortie ou encore relâchée.
MenuEvent	Sélection/déselection/annulation opérée dans un **JMenu.**
MenuKeyEvent	Touche de menu pressée, relâchée ou encore saisie, pour composants de type **JMenuItem**.
PopupMenuEvent	❍ Sélection opérée dans un menu fugitif (popup menu) ❍ Affichage ou masquage (invisible) d'un menu fugitif ❍ Sélection opérée dans le menu fugitif d'un **JComboBox**.
TableColumnModelEvent	Concerne les composants de type **JTable** (tables) ❍ Rajout ou suppression d'une colonne dans une table ❍ Déplacement ou changement de taille de la colonne d'une table ❍ Modification de la sélection dans une table
TableModelEvent	Concerne les composants de type **JTable** (tables): modification du modèle associé à la table.
TreeExpansionEvent	Déploiement ou repli d'un **JTree** (arbre)
TreeModelEvent	Modification, insertion ou suppression du noeud d'un **JTree** (arbre)
TreeSelectionEvent	Modification de la sélection au sein d'un **JTree** (arbre)
UndoableEditEvent	Occurrence d'une opération «undo» dans le modèle associé à un **JTextComponent.**

3.1.6 Interroger un événement

Un gestionnaire d'événement reçoit en paramètre l'objet événement auquel il doit répondre. Chaque événement est donc représenté par un objet que le programmeur peut interroger, comme par exemple:

- En lui envoyant le message `getSource()` pour connaître l'objet source de l'événement;

```
public void actionPerformed (ActionEvent e) {
    if (e.getSource()== boutonQuitter) {
        /* instruction pour répondre à l'événement quitter */
    }
}
```

- pour les événements de type `MouseEvent` ou `MouseMotionEvent`, les messages `getX()` et `getY()` retournent la position de la souris dans le composant au moment où l'événement a été déclenché;

```
public void mousePressed (MouseEvent e) {
        System.out.println (e.getX() + " " + e.getY());
}
```

- etc..

Du point de vue hiérarchique, les événements héritent de la classe `java.util.EventObject`.

3.1.7 Threads et gestion des événements

La totalité du code de gestion des événements s'exécute dans un thread unique: le «event-dispatching thread». Le fait qu'il soit unique nous assure que chaque gestionnaire d'événement aura fini son exécution avant qu'un autre ne commence à s'exécuter.

Par exemple, la méthode `actionPerformed` s'exécute dans le thread «event-dispatching». Pendant l'exécution de cette méthode, aucun autre événement ne peut être traité!

Comme tout le code de «dessin» (`paint`, `paintComponent`) s'exécute également dans le même thread, on est sûr que lorsque la méthode `actionPerformed` s'exécute, le GUI est gelé et ne répondra à aucun message de type «repaint» ou «mouse click».

Les gestionnaires d'événements doivent être programmés de manière à s'exécuter **très rapidement** ! Faute de quoi, les peformances du programme pourraient s'en trouver fortement dégradées. Si, par aventure, le traitement à effectuer dans le gestionnaire présentait une certaine compléxité, il est conseillé de démarrer l'exécution d'un nouveau thread responsable d'accomplir cette opération.

3.2 PROGRAMMATION: COMMENT GÉRER UN ÉVÉNEMENT ?

3.2.1 1re étape: choisir une catégorie d'événements

La figure présentée ci-après dresse la liste exhaustive de tous les *types d'événements* qu'un composant graphique est susceptible de générer. Cette liste est représentée sous la forme d'un arbre d'héritage.

A titre d'exemple, la consultation de cet arbre nous indique que le composant `JButton` peut générer les types d'événements suivants:

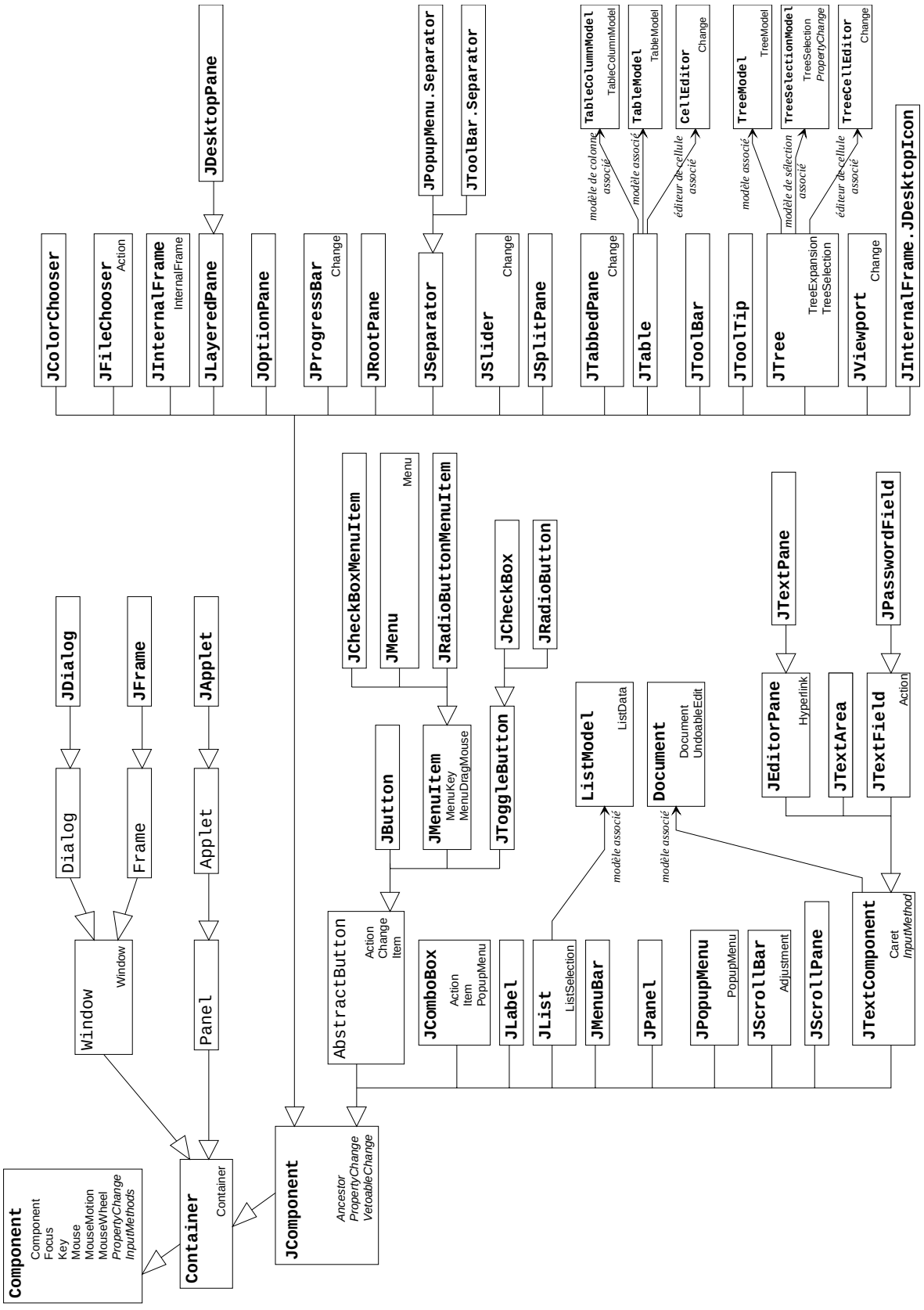
- **`Action`**, **`Change`** et **`Item`**: héritées de `AbstractButton`;
- **`Ancestor`**, **`PropertyChange`** et **`VetoableChange`**: héritées de `JComponent`;
- **`Container`**: héritée de `Container`;

- **`Component`**, **`Focus`**, **`Key`**, etc..: héritées de `Component`

Se référer aux deux tableaux présentés plus haut pour une brève explication de chacun de ces types d'événements[Voir tableau- Voici les événements dits «de base» - page 41], [Voir tableau- Evénements dits «sémantiques» - page 42].

Des événements qui ne nous concernent pas directement..

- Héritée de `JComponent`, la catégorie d'événements **`Ancestor`** a trait à des détails d'implémentation. Le programmeur «normal» ne les utilisera pas.
- Héritées de `Component` ou de `JComponent`, les catégories d'événements **`PropertyChange`**, **`VetoableChange`** et **`InputMethods`** ont trait au fait que les composants Swing sont compatibles avec la technologie «JavaBean». Ces événements intéressent essentiellement les concepteurs d'outils de développement.


Component
Component
Focus
Key
Mouse
MouseMotion
MouseWheel
PropertyChange
InputMethods
Container
Container
JComponent
Ancestor
PropertyChange
VetoableChange
Window
Window
Panel
Applet
Dialog
Frame
JDialog
JFrame
JApplet
AbstractButton
Action
Change
Item
JButton
JMenuItem
MenuKey
MenuDragMouse
JToggleButton
JCheckBoxMenuItem
JMenu
Menu
JRadioButtonMenuItem
JCheckBox
JRadioButton
JComboBox
Action
Item
PopupMenu
JLabel
JList
ListSelection
JMenuBar
JPanel
JPopupMenu
PopupMenu
JScrollBar
Adjustment
JScrollPane
ListModel
ListData
modèle associé
Document
Document
UndoableEdit
modèle associé
JTextComponent
Caret
InputMethod
JEditorPane
Hyperlink
JTextArea
JTextField
Action
JTextPane
JPasswordField
JDesktopPane
JLayeredPane
JInternalFrame
InternalFrame
JColorChooser
JFileChooser
Action
JOptionPane
JProgressBar
Change
JRootPane
JPopupMenu.Separator
JToolBar.Separator
JSeparator
JSlider
Change
JSplitPane
JTabbedPane
Change
JTable
JToolBar
JToolTip
JTree
TreeExpansion
TreeSelection
JViewport
Change
JInternalFrame.JDesktopIcon
TableColumnModel
modèle de colonne associé
TableModel
modèle associé
CellEditor
Change
éditeur de cellule associé
TreeModel
modèle associé
TreeSelectionModel
TreeSelection
PropertyChange
modèle de sélection associé
TreeCellEditor
Change
éditeur de cellule associé

3.2.2 2ème étape: inscrire un auditeur pour une certaine catégorie d'événements

Il suffit d'envoyer le message suivant:

leComposant.add***XXX***Listener* (*auditeur*);

où

- *leComposant* désigne le composant graphique auprès duquel on désire s'abonner
- *XXX* désigne la catégorie d'événements concernée (`Action`, `MouseMotion`, ..)
- *auditeur* désigne l'objet qui se met à l'écoute des événements

3.2.3 3ème étape: écrire les gestionnaires d'événement dans la classe de l'auditeur

Les procédures `add`***XXX***`Listener` sont spécifiées par les signatures suivantes:

```
public void addXXXListener (XXXListener l)
```

Comme par exemple:

```
public void addActionListener (ActionListener l)
public void addMouseMotionListener (MouseMotionListener l)
etc..
```

Les types `ActionListener`, `MouseMotionListener`, `ComponentListener`, etc.. sont des **interfaces** qui doivent **impérativement être implémentés** par la classe de l'auditeur concerné. Ces interfaces décrivent les gestionnaires d'événement qui sont susceptibles d'être invoqués lors du déclenchement de l'événement.

Voici à titre d'exemple les interfaces **`ActionListener`** et **`MouseMotionListener`**:

```
interface ActionListener {
    public void actionPerformed (ActionEvent e);
}

interface MouseMotionListener {
    public void mouseMoved (MouseEvent e);
    public void mouseDragged (MouseEvent e);
}
```

Comme nous l'avons remarqué précédemment, tous les gestionnaires de l'interface devront être implémentés, même ceux qui correspondent à des événements qui n'intéressent pas l'auditeur.

Voici deux nouveaux tableaux qui décrivent toutes les interfaces mises à disposition:

- La première colonne indique le nom de l'interface;
- La deuxième colonne indique le nom de la classe d'adaptation correspondante (si elle existe), [Voir paragraphe - Exemple: solution no3 - page 53];
- La troisième, enfin, donne la liste des méthodes contenues dans l'interface, implémentées par l'auditeur

Tableau 3: Les interfaces «Listener» des événements «de base»

Interface	*Classe d'adaptation*	*Méthodes*
ComponentListener	ComponentAdapter	**componentHidden** (ComponentEvent) **componentMoved** (ComponentEvent) **componentResized** (ComponentEvent) **componentShown** (ComponentEvent)
ContainerListener	ContainerAdapter	**componentAdded** (ContainerEvent) **componentRemoved** (ContainerEvent)
FocusListener	FocusAdapter	**focusGained** (FocusEvent) **focusLost** (FocusEvent)
KeyListener	KeyAdapter	**keyPressed** (KeyEvent) **keyReleased** (KeyEvent) **keyTyped** (KeyEvent)
MouseListener	MouseAdapter	**mouseClicked** (MouseEvent) **mouseEntered** (MouseEvent) **mouseExited** (MouseEvent) **mousePressed** (MouseEvent) **mouseReleased** (MouseEvent)
MouseMotionListener	MouseMotionAdapter	**mouseDragged** (MouseEvent) **mouseMoved** (MouseEvent)
MouseWheelListener	MouseWheelAdapter	**mouseWheelMoved**(MouseWheelEvent)

Nous reviendrons sur la notion de classe d'adaptation [Voir paragraphe - Exemple: solution no3 - page 53].

Tableau 4: Les interfaces «Listener» des événements sémantiques

Interface	*Methodes*
ActionListener	**actionPerformed**(ActionEvent)
AdjustmentListener	**ajustmentValueChanged**(AjustmentEvent)
AncestorListener	**ancestorAdded** (AncestorEvent) **ancestorMoved** (AncestorEvent) **ancestorRemoved** (AncestorEvent)
CaretListener	**caretUpdate** (CaretEvent)
CellEditorListener	**editingCanceled** (ChangeEvent) **editingStopped** (ChangeEvent)
ChangeListener	**stateChanged** (ChangeEvent)
DocumentListener	**changedUpdate** (DocumentEvent) **insertUpdate** (DocumentEvent) **removeUpdate** (DocumentEvent)
HyperlinkListener	**hyperlinkUpdate** (HyperlinkEvent)
InternalFrameListener	**internalFrameActivated** (InternalFrameEvent) **internalFrameClosed** (InternalFrameEvent) **internalFrameClosing** (InternalFrameEvent) **internalFrameDeactivated** (InternalFrameEvent) **internalFrameDeiconified** (InternalFrameEvent) **internalFrameIconified** (InternalFrameEvent) **internalFrameOpened** (InternalFrameEvent)
ItemListener	**itemStateChanged** (ItemEvent)
ListDataListener	**contentsChanged** (ListDataEvent) **intervalAdded** (ListDataEvent) **intervalRemoved** (ListDataEvent)
ListSelectionListener	**valueChanged** (ListSelectionEvent)
MenuDragMouseListener	**menuDragMouseDragged** (MenuDragMouseEvent) **menuDragMouseEntered** (MenuDragMouseEvent) **menuDragMouseExited** (MenuDragMouseEvent)
	menuDragMouseReleased (MenuDragMouseEvent)
MenuKeyListener	**menuKeyTyped** (MenuKeyEvent)
MenuListener	**menuCanceled** (MenuEvent) **menuDeselected** (MenuEvent) **menuSelected** (MenuEvent)
MouseInputListener (implements MouseListener & MouseMotionListener)	**mouseClicked** (MouseEvent) **mouseDragged** (MouseEvent) **mouseEntered** (MouseEvent) **mouseExited** (MouseEvent) **mouseMoved** (MouseEvent) **mousePressed** (MouseEvent) **mouseReleased** (MouseEvent)
PopupMenuListener	**popupmenuCanceled** (PopupMenuEvent) **popupMenuWillBecomeInvisible** (PopupMenuEvent) **popupMenuWillBecomeVisible** (PopupMenuEvent)
TableColumnModelListener	**columnAdded** (TableColumnModelEvent) **columnMarginChanged** (ChangeEvent) **columnMoved** (TableColumnModelEvent) **columnRemoved** (TableColumnModelEvent) **columnSelectionChanged** (ListSelectionEvent)

TableModelListener	**tableChanged** (TableModelEvent)
TreeExpansionListener	**treeCollapsed** (TreeExpansionEvent)
	treeExpanded (TreeExpansionEvent)
TreeModelListener	**treeNodesChanged** (TreeModelEvent) **treeNodesInserted** (TreeModelEvent) **treeNodesRemoved** (TreeModelEvent) **treeStructureChanged** (TreeModelEvent)
TreeSelectionListener	**valueChanged** (TreeSelectionEvent)
UndoableEditListener	**undoableEditHappened** (UndoableEditEvent)
WindowListener	**windowActivated** (WindowEvent) **windowDeactivated** (WindowEvent) **windowIconified** (WindowEvent) **windowDeiconified** (WindowEvent) **windowClosing** (WindowEvent) **windowClosed** (WindowEvent) **windowOpened** (WindowEvent)

Au contraire des événements dits «de base», il existe peu de classes `Adapter` pour les événements sémantiques. Ainsi, si le programmeur ne s'intéresse qu'à un seul des événements d'une même catégorie, il aura quand même l'obligation d'implémenter toutes les méthodes de l'interface. Les seules classes `Adapter` prévues sont les suivantes:

- `InternalFrameAdapter`,
- `MouseInputAdapter`,
- `WindowAdapter`

Pour plus d'informations, [Voir paragraphe - Exemple: solution no3 - page 53].

3.3 EXEMPLE: SOLUTION NO1

Dans cette première solution, l'auditeur est le container lui-même (la fenêtre).

En début de programme, il est nécessaire de rajouter la liste d'importation suivante:

```
import java.awt.event.*;          // Gestion des événements

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

//--------------------------------------------------------------------------
interface Configuration {
    public static int TAILLE_X = 400;          // Taille fenêtre en X
    public static int TAILLE_Y = 400;          // Taille fenêtre en Y
}
//--------------------------------------------------------------------------
class PanneauPrincipal extends JPanel
                       implements Configuration,
                          ActionListener, MouseMotionListener{
// Variables d'instance
    private JPanel pNord = new JPanel();
    private JButton bQuit = new JButton("Quitter");
    private JLabel lX = new JLabel("X > 000");
    private JLabel lY = new JLabel("Y > 000");

// Constructeur
    public PanneauPrincipal() {
           setLayout(new BorderLayout());
           add(pNord, BorderLayout.NORTH);
           pNord.add (lX);
           pNord.add (lY);
           pNord.add (bQuit);

           setBackground (Color.BLUE);
           setPreferredSize
              (new Dimension(TAILLE_X, TAILLE_Y));

           bQuit.addActionListener(this);
           addMouseMotionListener(this);
    }

// Gestionnaires d'événements
    public void actionPerformed(ActionEvent e) {
       System.exit(0);
    }

    public void mouseMoved (MouseEvent e) {
       lX.setText ("X > " + e.getX());
       lY.setText ("Y > " + e.getY());
    }

    public void mouseDragged (MouseEvent e){}
}

//--------------------------------------------------------------------------
public class Amorce {
    public static void main (String [] args) {
       JFrame f = new JFrame ("Gestion des ÈvÈnements");
       f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       // L'application se termine lorsque la fenÍtre est fermÈe
```

```
        // Mise en place du panneau principal au centre de la fenÍtre
        f.getContentPane().add(new PanneauPrincipal(),
                               BorderLayout.CENTER);
        f.pack();
        f.setVisible(true);
    }
}
```

On notera les 3 éléments:

- implémentation des interfaces `ActionListener` et `MouseMotionListener`
- l'événement `mouseDragged`, ignoré par notre programme, est associé à un gestionnaire qui ne fait rien (méthode «do-nothing»)
- l'enregistrement de la fenêtre en tant qu'écouteur (paramètre `this`)

3.4 EXEMPLE: SOLUTION NO2

Cette solution, plus claire, est une amélioration qui consiste à créer des classes spécifiques pour chacun des écouteurs.

Ces dernières seront en général déclarées sous la forme de classes imbriquées dans la classe du container. L'imbrication leur permet d'accéder directement aux variables d'instance du container. Dans notre exemple, la classe `EcouteurSouris` accède directement aux deux labels `lX` et `lY`.

```
class PanneauPrincipal extends JPanel implements Configuration{

// Variables d'instance
        private JPanel pNord = new JPanel();
        private JButton bQuit = new JButton("Quitter");
        private JLabel lX = new JLabel("X > 000");
        private JLabel lY = new JLabel("Y > 000");

// Gestionnaires d'événements sous forme de classes imbriquées
        class EcouteurBouton implements ActionListener {
                public void actionPerformed(ActionEvent e) {
                        System.exit(0);
                }
        }

        class EcouteurSouris implements MouseMotionListener {
                public void mouseMoved (MouseEvent e) {
                        lX.setText ("X > " + e.getX());
                        lY.setText ("Y > " + e.getY());
                }
                 public void mouseDragged (MouseEvent e) {}
        }
```

```
// Constructeur
       public PanneauPrincipal() {
           setLayout(new BorderLayout());
           add(pNord, BorderLayout.NORTH);
           pNord.add (lX);
           pNord.add (lY);
           pNord.add (bQuit);

           setBackground (Color.BLUE);
           setPreferredSize (new Dimension(TAILLE_X, TAILLE_Y));

           bQuit.addActionListener(new EcouteurBouton());
           addMouseMotionListener(new EcouteurSouris());
       }
}
```

3.5 EXEMPLE: SOLUTION NO3

Si le programmeur ne s'intéresse qu'à un seul des événements d'une même catégorie, il lui faudra pourtant implémenter tous les gestionnaires de la catégorie, quitte à écrire des-gestionnaires qui ne font rien...

C'est parfois un peu lourd..

Le langage met à disposition un certain nombre de ***classes d'adaptation***, associées chacune à une catégorie particulière d'événements. Une classe d'adaptation contient la totalité des gestionnaires d'événements correspondant à la catégorie, implémentés sous la forme de méthodes qui ne font rien (méthodes «do-nothing»).

Il suffit alors de définir la classe de l'écouteur en dérivant la classe d'adaptation par héritage et en se contentant de redéfinir les gestionnaires intéressants.

Le tableau présenté plus haut [Voir tableau- Les interfaces «Listener» des événements «de base» - page 48] donne la liste des classes d'adaptation pour les événements «de base».

Par exemple, on trouve la classe d'adaptation `MouseMotionAdapter`:

```
class MouseMotionAdapter implements MouseMotionListener {
       public void mouseMoved (MouseEvent e) {}
       public void mouseDragged (MouseEvent e) {}
}
```

En appliquant cette solution à notre exemple, la classe `EcouteurSouris` devient:

```
class EcouteurSouris extends MouseMotionAdapter {
    public void mouseMoved (MouseEvent e) {
        lX.setText ("X > " + e.getX());
        lY.setText ("Y > " + e.getY());
    }
}
```

On remarquera que le gestionnaire `mouseDragged` a pu ainsi être laissé de côté.

C'est aussi cette solution que l'on peut utiliser pour gérer la fermeture de la fenêtre directement au niveau de la barre de titre. L'interface `WindowListener` comporte en effet pas moins de 7 gestionnaires qu'il faudrait implémenter si on mettait en oeuvre la solution no 2 !!

```
// Classes internes
class EcouteurFenetre extends WindowAdapter {
      public void windowClosing (WindowEvent e) {
            System.exit(0);
      }
}
  :
// Constructeur
public Fenetre () {
   this.addWindowListener (new EcouteurFenetre ());
    :
```

3.6 EXEMPLE: SOLUTION NO 4

Cette dernière solution, encore peu usitée, utilise le mécanisme de classe anonyme: la classe de l'écouteur est écrite au moment même de l'instanciation de l'objet écouteur. En opérant ainsi, on localise très fortement les instructions qui ont trait à la même fonctionnalité:

1. Création du composant graphique;
2. Choix de son emplacement dans le conteneur;
3. Descriptions du ou de ses gestionnaires d'événements associés.

Concept donc très intéressant, même s'il peut étonner de prime abord.

```
class PanneauPrincipal extends JPanel implements Configuration{
// Variables d'instance
    private JPanel pNord = new JPanel();
    private JButton bQuit = new JButton("Quitter");
    private JLabel lX = new JLabel("X > 000");
    private JLabel lY = new JLabel("Y > 000");

// Constructeur
    public PanneauPrincipal() {
        setLayout(new BorderLayout());
        add(pNord, BorderLayout.NORTH);
        pNord.add (lX);
        pNord.add (lY);
        pNord.add (bQuit);

        setBackground (Color.BLUE);
        setPreferredSize (new Dimension(TAILLE_X, TAILLE_Y));
```

```
        bQuit.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });
        addMouseMotionListener(new MouseMotionAdapter(){
            public void mouseMoved (MouseEvent e) {
                lX.setText ("X > " + e.getX());
                lY.setText ("Y > " + e.getY());
            }
        });
    }
}
```

Annexe A Liste des figures

Annexe B Liste des tableaux

Annexe C Index