# Chapter 6: JFC Components

# Outline

Goals and Objectives....4
Section 1: Layout Managers....5
Layout Managers....6
FlowLayout....7
GridLayout....8
BorderLayout....9
CardLayout and GridBagLayout....10
BoxLayout....11
ScrollPaneLayout....12
Absolute Position....13
Section 2: Components....14
AbstractButton....15
JLabel....17
JButton....18
JCheckBox....19
JRadioButton....20
JToggleButton....21
JList....22
JComboBox....24
JScrollPane....25
JTextComponent....27
Reading and writing Using JTextComponent Objects....28
Cursor Position within a JTextComponent....29
JTextArea....30
JEditorPane....32
JSlider....36
JProgressBar....37
JTabbedPane....38
JSplitPane....39
JTable....40
JTable TableModel....42
Borders....46
Section 3: Menus....47
Menus....48
JMenuBar....49
Menu Demonstration....50
Section 4: Tool Bars....53
Creating Tool Bars....54
Section 5: Event Handling....57

Events ........................................................................................................58
Listener Interfaces ........................................................................................59
Event Interface ..............................................................................................60
Event Objects .................................................................................................62
getSource Method ..........................................................................................63
Action Commands...........................................................................................64
MouseListener and MouseMotionListener .....................................................65
WindowListener Interface...............................................................................66
Actions ............................................................................................................67
Section 6: Look and Feel ..................................................................................70
Look and Feel.................................................................................................71
Summary ........................................................................................................73

# Goals and Objectives

## *After completing this unit the student will:*

- Learn about the Java Foundation Classes
- Add components to containers
- Understand how layout managers work
- Provide a GUI interface for Java applications
- Manipulate most common controls
- Handle application events

# Section 1: Layout Managers

# Layout Managers

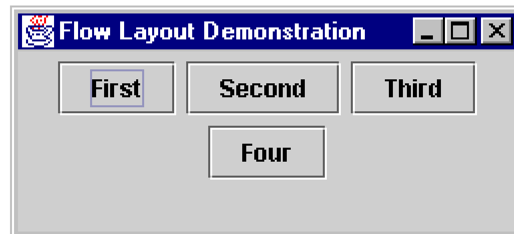
Layout managers determine how components added to a container are positioned within that container. Each container has a default layout manager. The default layout manager can be changed using the **setLayout** method. There are 8 layout managers available:

- FlowLayout
- GridLayout
- BorderLayout
- CardLayout
- GridBagLayout
- BoxLayout
- ScrollPaneLayout
- Absolute

# FlowLayout

FlowLayout permits each component to assume its preferred size. Preferred size means that the component such as a JButton will be large enough to display it text plus a little additional room. Components are arranged left to right and top to bottom. FlowLayout is the default layout manager of a JPanel.

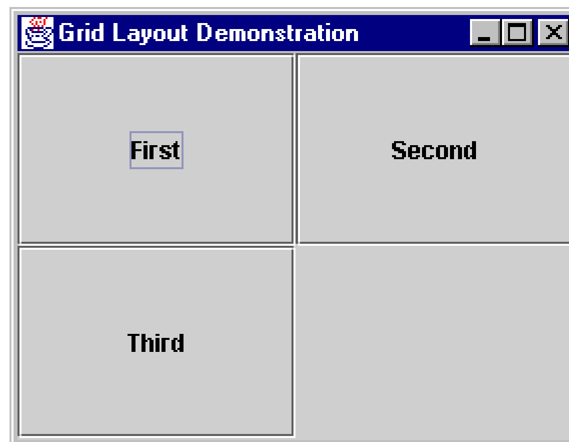
```
JPanel topPanel = new JPanel();
topPanel.setLayout(new FlowLayout());
getContentPane().add(topPanel);

JButton btnFirst = new JButton("First");
JButton btnSecond = new JButton("Second");
JButton btnThird = new JButton("Third");
JButton btnFour = new JButton("Four");
topPanel.add(btnFirst);
topPanel.add(btnSecond);
topPanel.add(btnThird);
topPanel.add(btnFour);
```



# GridLayout

The GridLayout manager arranges components in a grid format. Each component occupies an entire cell. Components added in a row column manner.

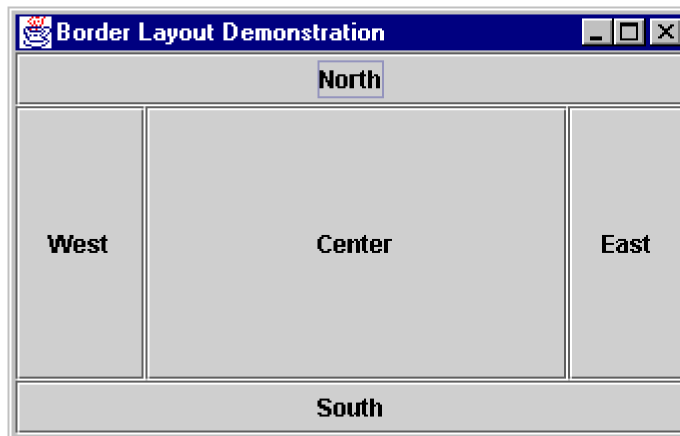
```
JPanel topPanel = new JPanel();
topPanel.setLayout(new GridLayout(2,2));
getContentPane().add(topPanel);

JButton btnFirst = new JButton("First");
JButton btnSecond = new JButton("Second");
JButton btnThird = new JButton("Third");

topPanel.add(btnFirst);
topPanel.add(btnSecond);
topPanel.add(btnThird);
```



# BorderLayout

This layout manager uses the four compass points and the center to arrange components. Components fill each region completely.

```
JPanel topPanel = new JPanel();
topPanel.setLayout(new BorderLayout());
getContentPane().add(topPanel);

JButton btnNorth = new JButton("North");
JButton btnSouth = new JButton("South");
JButton btnEast = new JButton("East");
JButton btnWest = new JButton("West");
JButton btnCenter = new JButton("Center");
topPanel.add(btnNorth,"North");
topPanel.add(btnSouth,"South");
topPanel.add(btnEast,"East");
topPanel.add(btnWest,"West");
topPanel.add(btnCenter,"Center");
```



# CardLayout and GridBagLayout

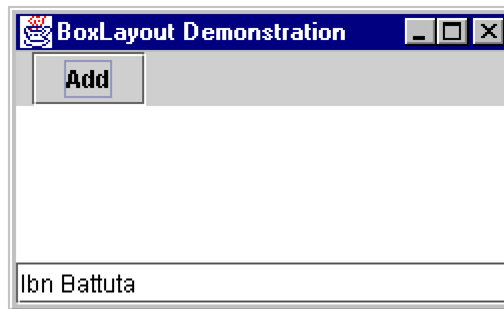
These two layout managers are not used frequently. They are more difficult to use than the FlowLayout or BorderLayout managers. The CardLayout manager provides an interface similar to that of a tabbed dialog sheet. The GridBagLayout manager is similar to the GridLayout manager except that more detail control is possible.

A great degree of control can be exercised over the layout using the GridBagLayout. http://java.sun.com/docs/books/tutorial/uiswing/layout/gridbag.html provides a good tutorial on how to use it.

# BoxLayout

The BoxLayout aligns components along an X or Y axis. Along the X-axis, components are provided with the same vertical height but different widths. Along the Y-axis, components are provided with the same horizontal width but different heights. The first argument of the constructor specifies the container while the second argument is the orientation.

```
JPanel topPanel = new JPanel();
topPanel.setLayout(new BoxLayout(topPanel, BoxLayout.Y_AXIS));
getContentPane().add(topPanel);

JButton buttonAdd = new JButton("Add");
JTextArea taComment = new JTextArea(5, 30);
JTextField tfName = new JTextField("Ibn Battuta");
topPanel.add(buttonAdd);
topPanel.add(taComment);
topPanel.add(tfName);
```



# ScrollPaneLayout

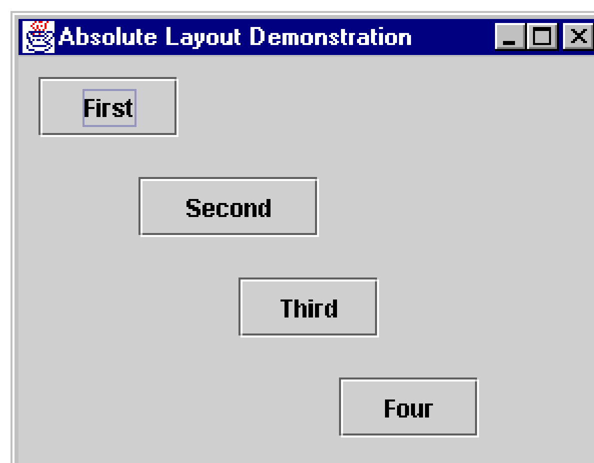
The ScrollPaneLayout is built into the JScrollPane. There are nine different areas that make up the JScrollPane. The center area is a JViewport that automatically positions the internal component. http://java.about.com/library/swing/bl-Swing_Chapter_7-1.htm provides a good tutorial on how to use this manager.

# Absolute Position

Setting the absolute position can be more difficult to use than the FlowLayout or other managers. To specify the absolute position use a null argument in the **setLayout** method. The **setSize** and **setLocation** methods can then be used to control the size and position of the component.

```
public void setSize(int width, int height);
public void setLocation(int x, int y);
```

```
JPanel topPanel = new JPanel();
topPanel.setLayout(null);
getContentPane().add(topPanel);

JButton btnFirst = new JButton("First");
btnFirst.setSize(70,30);
btnFirst.setLocation(10,10);

JButton btnSecond = new JButton("Second");
btnSecond.setSize(90,30); btnSecond.setLocation(60,60);

JButton btnThird = new JButton("Third");
btnThird.setSize(70,30); btnThird.setLocation(110,110);

JButton btnFour = new JButton("Four");
btnFour.setSize(70,30); btnFour.setLocation(160,160);

topPanel.add(btnFirst);
topPanel.add(btnSecond);
topPanel.add(btnThird);
topPanel.add(btnFour);
```

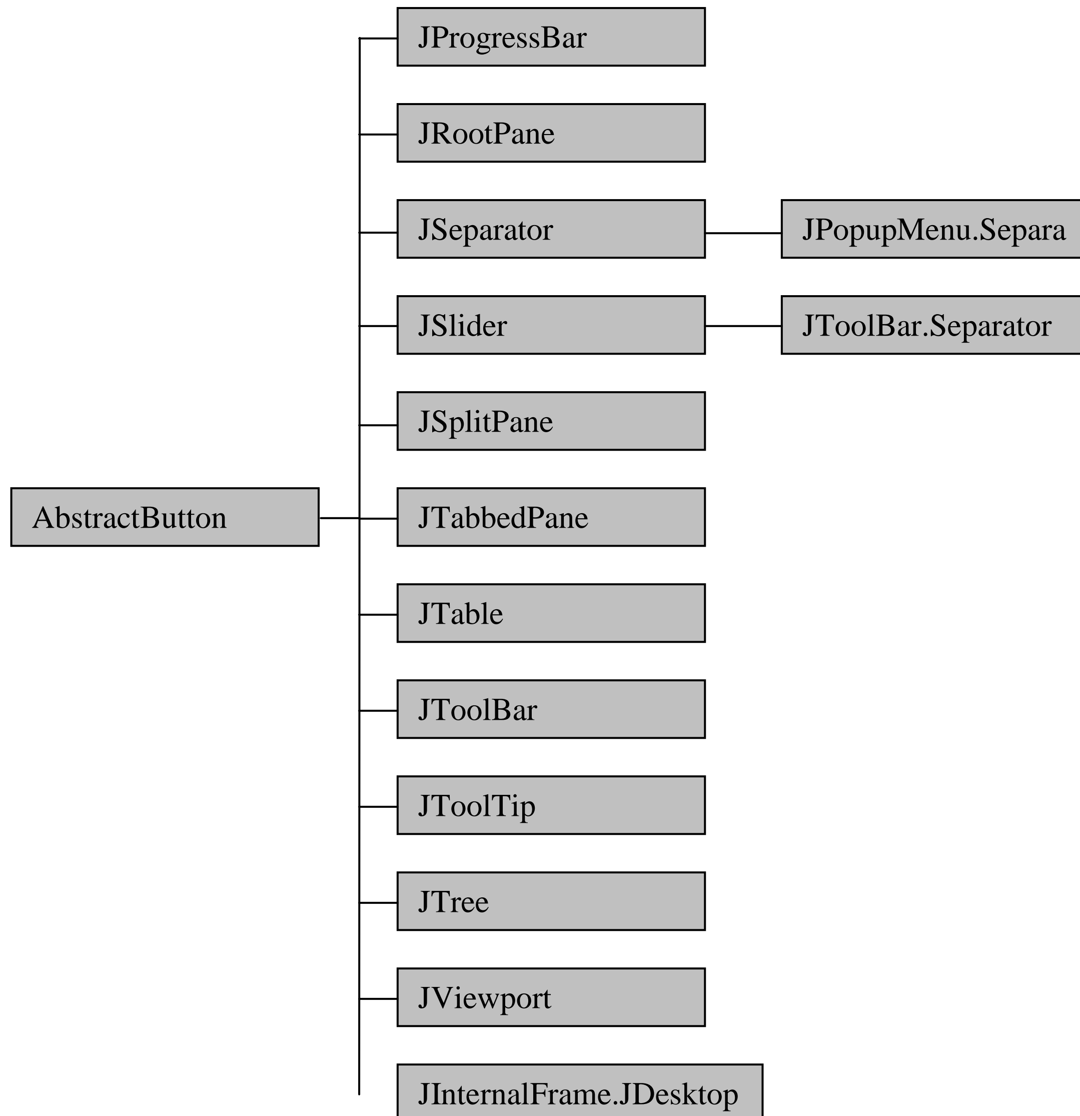


# Section 2: Components

# AbstractButton

The AbstractButton class possesses a number of common methods inherited by its derived classes. Classes derived from this class exhibit button like behavior. They can:

- Displays text and/or icon
- Can be enabled/disabled
- Can possess a mnemonic
- HTML text can be displayed



Useful methods include:

- **setMnemonic** – Sets a mnemonic
- **doClick** – Activates the button
- **setDisabledIcon** – Set the icon to be used when the button is disabled
- **setHorizontalAlignment**, **setVerticalAlignment** – Set the alignment for icons
- **setHorizontalTextPosition**, **setVerticalTextPosition** – Sets the alignment for text

# JLabel

The purpose of JLabel is to display static text on the screen. General characteristics include:

- The text displayed is static
- The user does not interact with this component
- The text can be positioned within that label
- Images can be added to a label
- The JLabel cannot receive focus

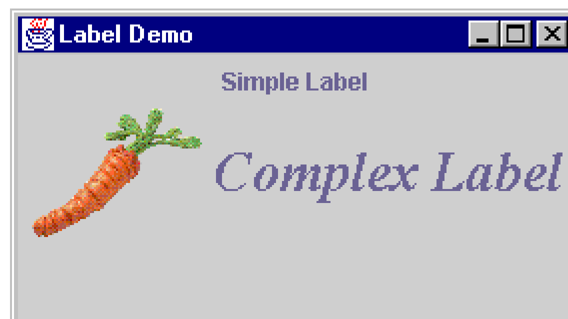
```
JLabel labelSimple = new JLabel("Simple Label");
JLabel labelIcon = new JLabel(iconCarrot);
```

The **setText** method can be used to set a new text string during execution.

```
labelSimple.setText("New Label");
```

Alignment is controlled either at creation or using various alignment methods.

```
JPanel topPanel = new JPanel();
getContentPane().add(topPanel);

JLabel labelSimple = new JLabel("Simple Label");
topPanel.add(labelSimple);
JLabel labelComplex = new JLabel("Complex Label");
Font fontComplex = new Font("Times New Roman", Font.BOLD | Font.ITALIC, 28);
labelComplex.setFont(fontComplex);
Icon iconCarrot = new ImageIcon("carrot.gif");
labelComplex.setIcon(iconCarrot);
labelComplex.setHorizontalAlignment(JLabel.RIGHT);
topPanel.add(labelComplex);
```



# JButton

The JButton component is a frequently used component.  Buttons are used to allow the user to perform some action.  They possess a text title that conveys their purpose.  Buttons will accept focus allowing a button to fire events.

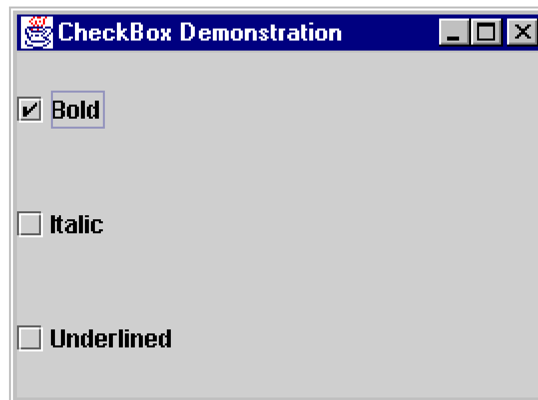
```
public JButton();
public JButton(String label);
public String getText()
public synchronized void setText(String label);
```

```
JPanel topPanel = new JPanel();
getContentPane().add(topPanel);

JButton btnExit = new JButton("Exit");
topPanel.add(btnExit);
```

# JCheckBox

The JCheckBox component represents check boxes. Zero or more check boxes can be selected. Several methods provide access to this component. Overloaded constructors allow the assignment of an icon to a label.

```
JPanel topPanel = new JPanel();
topPanel.setLayout(new GridLayout(3,1));
getContentPane().add(topPanel);

JCheckBox cbBold = new JCheckBox("Bold",true);
JCheckBox cbItalic = new JCheckBox("Italic");
JCheckBox cbUnderlined = new JCheckBox("Underlined");
topPanel.add(cbBold);
topPanel.add(cbItalic);
topPanel.add(cbUnderlined);
```




Useful methods of the JCheckBox include:

- **getText** – To obtain the current label
- **setText** – Used to set the label
- **getState** – Returns a boolean value reflecting the state of the check box
- **setState** – Sets the checkbox

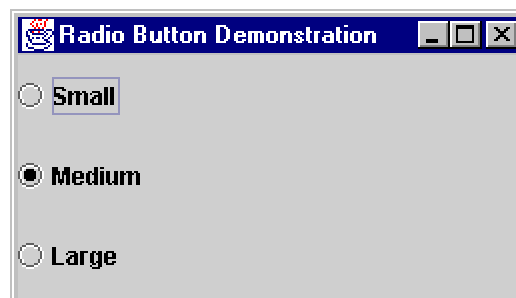
# JRadioButton

The JCheckBox component is used to create radio buttons. JRadioButtons are grouped together by adding them to a ButtonGroup object. Only one radio button may be selected at a time by the user.

Several radio buttons can be added to a GUI interface. However, for them to work together there has to be some way to group them. Visually the might be grouped by placing them in a box or physically placing them close together. However, we also need to programmatically group them. This is done using the ButtonGroup object. Its only purpose is to tell Java that the buttons in this group work together.

Icons can also be added to radio buttons

```
JPanel topPanel = new JPanel();
topPanel.setLayout(new GridLayout(3,1));
getContentPane().add(topPanel);

JRadioButton rbSmall = new JRadioButton("Small");
JRadioButton rbMedium = new JRadioButton("Medium",true);
JRadioButton rbLarge = new JRadioButton("Large");

ButtonGroup rbg = new ButtonGroup();

rbg.add(rbSmall);
rbg.add(rbMedium);
rbg.add(rbLarge);

topPanel.add(rbSmall);
topPanel.add(rbMedium);
topPanel.add(rbLarge);
```



# JToggleButton

The JToggleButton component is the parent class for both the JCheckBox and the JRadioButton. When it is pressed it stays in until pressed again.  Icons can also be assigned to this button.

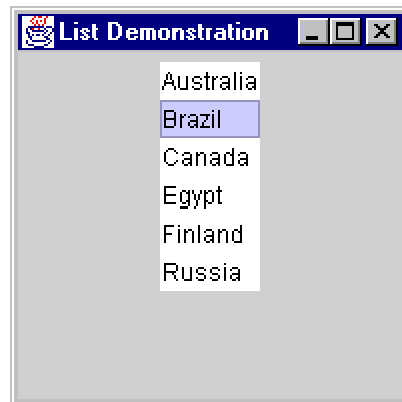
```
JPanel topPanel = new JPanel();
topPanel.setLayout(new GridLayout(1,3));
getContentPane().add(topPanel);

JToggleButton tbPower = new JToggleButton("Power");
JToggleButton tbOnLine = new JToggleButton("On-Line");
JToggleButton tbHi = new JToggleButton("Hi");
topPanel.add(tbPower);
topPanel.add(tbOnLine);
topPanel.add(tbHi);
```

# JList

The JList component is used to display a list of elements. The user can select one or more items from the list. Scrollbars are added by placing the JList object inside of a JScrollPane object. Useful methods include:

- **getSelectedIndex** – Returns the index of the element selected
- **setSelectedIndex** – Sets the selected item
- **getSelectedValue** – Returns the item selected
- **getItemCount** – returns the number of elements in the list
- **isSelectionEmpty** – Returns true if there are no items in the list
- **setListData** – Replaces the content of the list

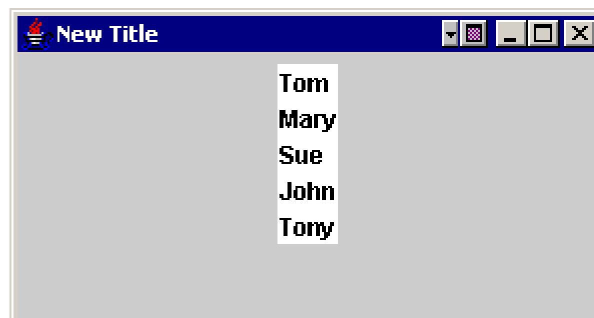
```
JPanel topPanel = new JPanel();
topPanel.setLayout(new FlowLayout());
getContentPane().add(topPanel);

String Countries[] = { "Australia", "Brazil", "Canada", "Egypt",
   "Finland", "Russia"};
JList listCountries = new JList(Countries);
topPanel.add(listCountries);
```




```
JScrollPane listPane = new JScrollPane(listCountries);
topPanel.add(listPane);
```

Notice that the constructor takes an array of Objects. The array does not have to be an array of Strings.

```
public JListDemo(String frameTitle) {
        super(frameTitle);
        // Create GUI interface
        Container c = getContentPane();
        JPanel demoPanel = new JPanel();

        demoPanel.setLayout(new FlowLayout());
        getContentPane().add(demoPanel);

        Employee arr[] = new Employee[5];
        arr[0] = new Employee("Tom");
        arr[1] = new Employee("Mary");
        arr[2] = new Employee("Sue");
        arr[3] = new Employee("John");
        arr[4] = new Employee("Tony");
        JList listCountries = new JList(arr);
        demoPanel.add(listCountries);

        c.add(demoPanel);
}
```



# JComboBox

The JComboBox component provides a list of choices the user can select from. The user can also edit the contents of the component. Useful methods include:

- **addItem** – Used to add an object to the list
- **getItem** – Returns the specified item
- **removeItem** – Removes the specified item
- **removeAllItems** – Removes all items from the list
- **getSelectedIndex** – Returns the index of the selected item
- **getSelectedItem** – Returns the item selected
- **getItemCount** – returns the number of items in the list
- **isEditable** – Returns whether the list can be edited or not
- **setEditable** – Specifies whether the list can be edited

```
JPanel topPanel = new JPanel();
topPanel.setLayout(new FlowLayout());
getContentPane().add(topPanel);

String Countries[] = { "Australia", "Brazil", "Canada", "Egypt", "Finland", "Russia"};
JComboBox cbCountries = new JComboBox(Countries);
topPanel.add(cbCountries);
```

Using **setEditable** to enable editing

# JScrollPane

The JScrollPane class permits the automatic scrolling of components. Internal to JScrollPane is a JViewport to which components are added. JViewport can be used standalone if the developer needs greater control on how scrolling occurs.

```
JScrollPane spMain = new JScrollPane();
JViewport viewPort = spMain.getViewport();
viewPort.add(someComponent);
```

or

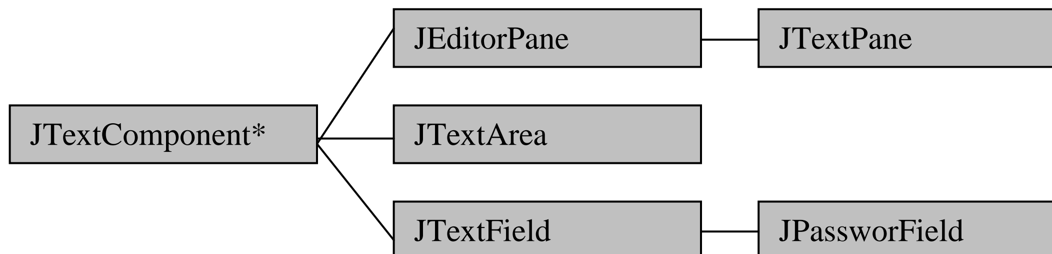
```
JScrollPane spMain = new JScrollPane(someComponent);
```

```
JPanel topPanel = new JPanel();
topPanel.setLayout(new BorderLayout());
getContentPane().add(topPanel);

Icon iconBigTiger = new ImageIcon("BigTiger.gif");
JLabel labelBigTiger = new JLabel(iconBigTiger);
JScrollPane spMain = new JScrollPane(labelBigTiger);
topPanel.add(spMain, BorderLayout.CENTER);
```

Scroll Pane Demonstration

# JTextComponent

The JTextComponent class is the base class for the JTextArea, JTextField and JEditorPane classes. It is an abstract class found in the javax.swing.text package. It provides a number of useful text methods shared by its derived classes.




- **getText** – Returns the component's contents
- **setText** – Sets the component's contents
- **getSelectedText** – Returns the selected text
- **getSelectedEnd** – Returns an index to the end of the selected text
- **getSelectedStart** – Returns an index to the start of the selected text
- **selectAll** – Selects all of the text
- **replaceSelection** – Replaces the selected text
- **isEditable** – Determines if the component is editable
- **setCaretPosition** – Sets the position of the caret

The JTextComponent also has read and write methods that are used to read and write to IO streams. This makes it easier to populate a JTextComponent from a file.

# Reading and writing Using JTextComponent Objects

The JTextComponent class provides an easy way of moving data between itself and a file. The read method first argument is a FileReader object. The second argument is an object that might be used by the underlying document.

```
try {
      FileReader fr = new FileReader(currentFile);
      txtComponent.read(fr, "file");
      fr.close();
}
catch (IOException e) {
      e.printStackTrace();
}
```

The write method takes a single object that represents a Writer stream.

```
try {
      FileWriter fw = new FileWriter(currentFile);
      txtComponent.write(fw);

      fw.close();
}
catch (IOException e) {
      e.printStackTrace();
}
```

The setCaret method is used to set the position of the caret within a JTextComponent.

# Cursor Position within a JTextComponent

When the caret, that represents the position of the cursor in a JTextComponent, is moved the caretUpdate method of the CaretListener interface is invoked. The cursor position within a JTextComponent is returned by the getDot method. This method returns an index into the string that composes the text component. The initial position is 0. For example, if the text component consists of the two lines:

```
First line
Second line
```

If the caret is after the ‘s’ of the word ‘First, the index returns is 4. If the caret is after the ‘c’ of the word ‘Second, the index returns is 14. The end of line character is counted as a single line.

In order to determine the row and column position of the caret in a text component the length of each line needs to be calculated. The following

```
editRegion.addCaretListener(new CaretListener() {
      public void caretUpdate(CaretEvent e) {
            int caretPosition = e.getDot();
            String buffer = textComponent.getText();
            int row = 0;
            int col = 0;

            for(int i=0; i<caretPosition; i++) {
                  if(buffer.charAt(i)=='\n') {
                        row++;
                        col = 0;
                  } else if(buffer.charAt(i)=='\t') {
                        col += textComponent.getTabSize();
                  }else {
                        col++;
                  }
            }
            // Use the row and col variables as desired
      }
);
```

Instead of using the getDot method, the getCaretPosition method can be used. This is a method of the JTextComponent class and can be used without using CaretListener..

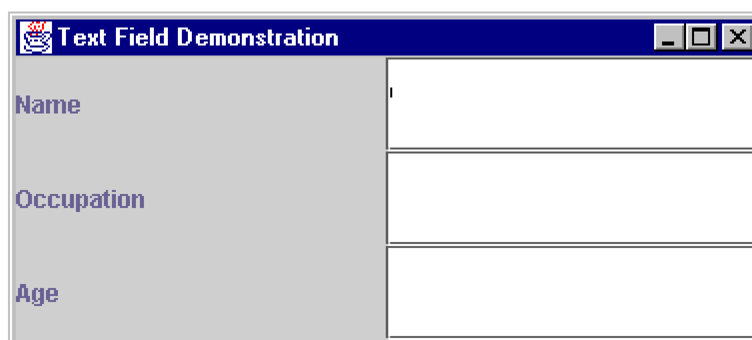
```
            int dot = textComponent.getCaretPosition();
```

# JTextArea

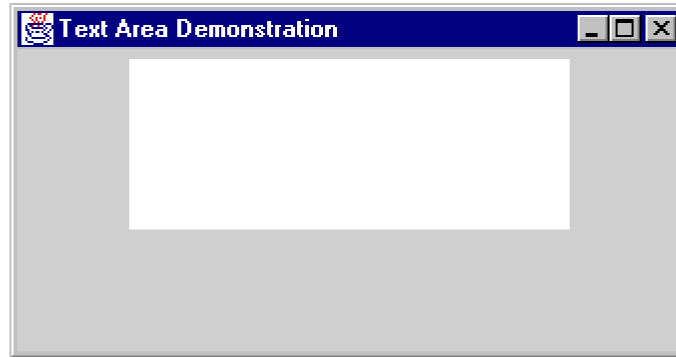
The JTextArea class displays a text input box with multiple lines. The constructor specifies the number of rows and columns used by the component. A JTextField component uses only a single row. Otherwise it is very similar to the JTextArea component.

- **getColumns** – Returns the number of columns
- **getRows** – Returns the number of rows
- **setColumns** – Sets the number of columns
- **setRows** – Sets the number of rows
- **insert** – Inserts a string at a specified position
- **setFont** – Sets the font to be used

```
JPanel topPanel = new JPanel();
topPanel.setLayout(new GridLayout(3,2));
getContentPane().add(topPanel);
JLabel labelName = new JLabel("Name");
JLabel labelOccupation = new JLabel("Occupation");
JLabel labelAge = new JLabel("Age");
JTextField tfName = new JTextField();
JTextField tfOccupation = new JTextField();
JTextField tfAge = new JTextField();
topPanel.add(labelName);
topPanel.add(tfName);
topPanel.add(labelOccupation);
topPanel.add(tfOccupation);
topPanel.add(labelAge);
topPanel.add(tfAge);
```



```
JPanel topPanel = new JPanel();
topPanel.setLayout(new FlowLayout());
getContentPane().add(topPanel);

JTextArea taComment = new JTextArea(5,20);
topPanel.add(taComment);
```



# JEditorPane

The JEditorPane component provides a means to display and edit specially formatted text. A javax.swing.text.EditorKit class provides the means to manage different format types. The javax.swing.text.html.HTMLEditorKit and javax.swing.text.rtf.RTFEditorKit classes are derived from javax.swing.text.EditorKit and supply the basic display capabilities for HTML and RTF respectively.

The full source code for the demo is found in JEditorDemo.java. Only the JEditorPane features are exampled here. The currentURL and oldURL variables are used to hold the current and old URL strings. This is needed to implement the browser's back button.

```
private String currentURL = "http://www.java.sun.com";
private String oldURL;
```

A URL object is needed to hold the URL of interest. It is build using the string referenced by currentURL. The JEditorPane then uses the URL object in its constructor. The **setEditable** method issues that the user cannot modify the page being displayed. The editor is enclosed in a JScrollPane and the program adds itself to a HyperlinkListener.

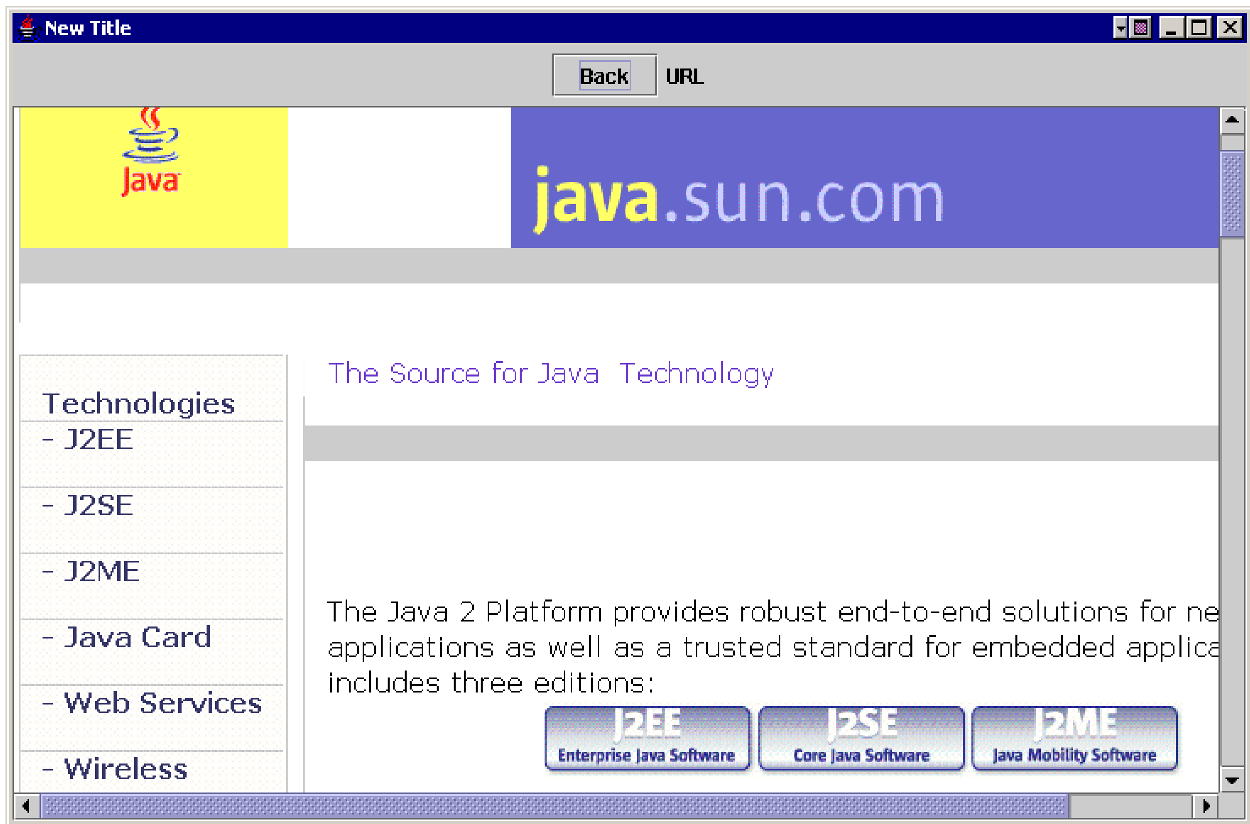
```
URL url = new URL(currentURL);
urlAddress = url.toString();
htmlEditor = new JEditorPane(url);
htmlEditor.setEditable(false);
JScrollPane scrollPane = new JScrollPane(htmlEditor);
htmlEditor.addHyperlinkListener(this);
```

The HyperlinkListener has a single method **hyperlinkUpdate**. When the requested HTML page is received from a web server, this method is called. An hour glass cursor is temporarily displayed. The **getPage** method of the HyperlinkEvent object returns the URL being displayed. The **setPage** method of the JEditorPane will display its URL object.

```
public void hyperlinkUpdate(HyperlinkEvent e) {
        if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED) {
                Cursor cursor = htmlEditor.getCursor();
                Cursor waitCursor =
                        Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR);
                 htmlEditor.setCursor(waitCursor);
                try {
                        System.out.println("Current URL: " + htmlEditor.getPage());
                        System.out.println("    New URL: " + e.getURL());
                        htmlEditor.setPage(e.getURL());
                }
                catch (IOException err) {
                        System.out.println("IO Exception in hyperlinkUpdate");
                        System.out.println("Attempted to access URL: " +
                                htmlEditor.getPage());
                }
                htmlEditor.setCursor(cursor);
        }
}
```

The code in the actionPerformed method also uses the setPage method to accept a new URL as entered by the used and to affect the back button.

```
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == txtURL) {
        Cursor cursor = htmlEditor.getCursor();
        Cursor waitCursor =
            Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR);
         htmlEditor.setCursor(waitCursor);
        try {
            oldURL = currentURL;
            currentURL = txtURL.getText();
            htmlEditor.setPage(currentURL);
            System.out.println("    New URL: " + currentURL);
        }
        catch (IOException err) {
            System.out.println("IO Exception in hyperlinkUpdate");
            System.out.println("Attempted to access URL: " +
                htmlEditor.getPage());
        }
        htmlEditor.setCursor(cursor);
    } else if (e.getSource() == btnBack) {
        Cursor cursor = htmlEditor.getCursor();
        Cursor waitCursor =
            Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR);
         htmlEditor.setCursor(waitCursor);
        try {
            htmlEditor.setPage(oldURL);
            System.out.println("    Old URL: " + oldURL);
        }
        catch (IOException err) {
            System.out.println("IO Exception in hyperlinkUpdate");
            System.out.println("Attempted to access URL: " +
                htmlEditor.getPage());
        }
        htmlEditor.setCursor(cursor);

    } else {

    }
}
```

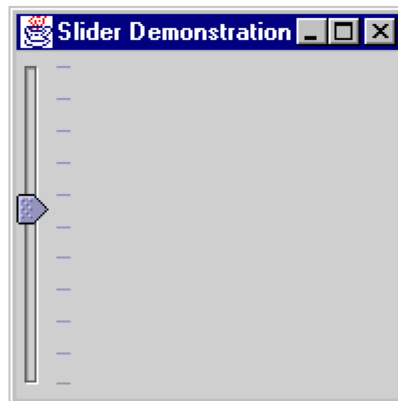

New Title
Back
URL
Java
java.sun.com
Technologies
- J2EE
- J2SE
- J2ME
- Java Card
- Web Services
- Wireless
The Source for Java Technology
The Java 2 Platform provides robust end-to-end solutions for ne
applications as well as a trusted standard for embedded applica
includes three editions:
J2EE
Enterprise Java Software
J2SE
Core Java Software
J2ME
Java Mobility Software

# JSlider

The JSlider component provides a means for the user to move a slider control. Major and minor tick marks are supported. The slider component can be positioned horizontally or vertically. Useful methods include:

- **setPaintTicks** – Specifies whether ticks marks are displayed or not
- **setMajorTickSpacng** – Specifies the spacing of the major ticks
- **setMinorTickSpacng** – Specifies the spacing of the minor ticks

```
JPanel topPanel = new JPanel();
topPanel.setLayout(new BorderLayout());
getContentPane().add(topPanel);

JSlider slider1 = new JSlider(JSlider.VERTICAL, 0, 100, 0);
slider1.setPaintTicks(true);
slider1.setMajorTickSpacing(10);
topPanel.add(slider1);
```

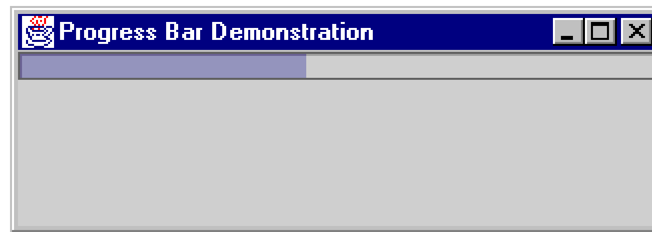


# JProgressBar

The JProgressBar component provides visual feedback to the user as to the progress of an activity. The orientation, minimum and maximum values can be set when created or later. Useful methods include:

- **setValue** – Sets the current value of the progress bar
- **setOrientation** - Set the orientation
- **setMinimum** – Sets the minimum value
- **setMaximum** – Sets the maximum value

```
JPanel topPanel = new JPanel();
topPanel.setLayout(new BorderLayout());
getContentPane().add(topPanel);

JProgressBar bar1 = new JProgressBar(JSlider.HORIZONTAL, 0, 100);
bar1.setValue(45);
topPanel.add("North",bar1);
```

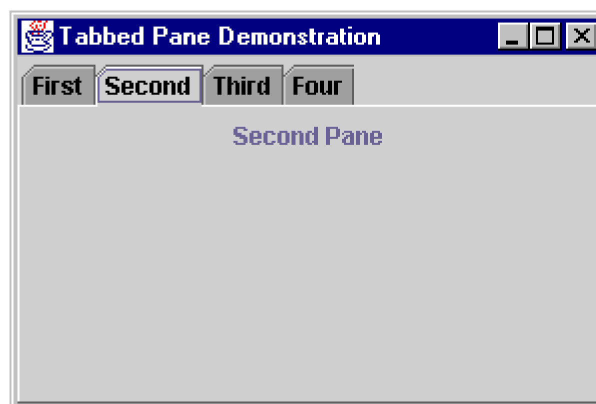


# JTabbedPane

The JTabbedPane component provides a tabbed control that uses multiple panels. Panels are added each representing a new tab. Text and/or an Icon can appear on a tab. The overloaded **addTab** method will add a new pane. Useful methods include:

- **insertTab** – Used to insert a new tab
- **removeTabAt** – Removes a specific tab
- **getSelectedIndex** – Returns the index of the tab selected (index starts at 0)
- **setSelectedIndex** – Sets the tab to have focus
- **getTabCount** – Returns the number of tabs

```
JPanel topPanel = new JPanel();
topPanel.setLayout(new BorderLayout());
getContentPane().add(topPanel);

JTabbedPane tabbedPane = new JTabbedPane();
JPanel pane11 = new JPanel();
JPanel pane12 = new JPanel();
JPanel pane13 = new JPanel();
JPanel pane14 = new JPanel();

pane11.add(new JLabel("First Pane"));
pane12.add(new JLabel("Second Pane"));
pane13.add(new JLabel("Third Pane"));
pane14.add(new JLabel("Fourth Pane"));

tabbedPane.addTab("First", pane11);
tabbedPane.addTab("Second", pane12);
tabbedPane.addTab("Third", pane13);
tabbedPane.addTab("Four", pane14);

topPanel.add(tabbedPane);
```

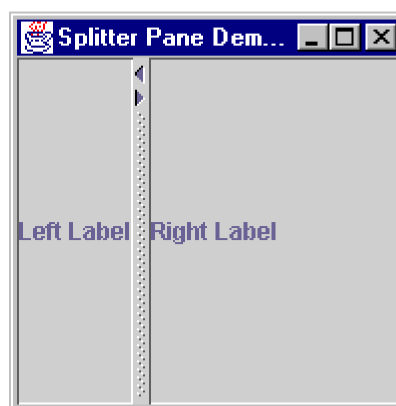


# JSplitPane

The JSplitPane component provides a means for the user to control the resizing of two components. These components can be arranged either horizontally or vertically. Scrollable panes are not automatically provided. Useful methods include:

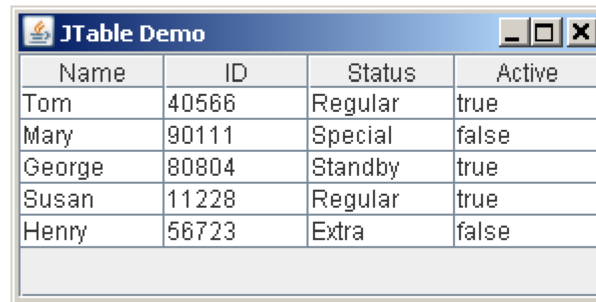
- **setContinuousLayout** – setting to true results in each pane to be updated continuously
- **setDividerLocation** – Uses a floating point number between 0 and 1 to set the position of the splitter bar
- **setLeftComponent** – Sets the left component
- **setRightComponent** – Sets the right component
- **setTopComponent** – Sets the top component
- **setBottomComponent** – Sets the bottom component
- **setOneTouchExpandable** – When set to true will add arrows on the splitter bar
- **setMinimumSize** – Specifies the smallest size a component can assume

```
JPanel topPanel = new JPanel();
topPanel.setLayout(new BorderLayout());
getContentPane().add(topPanel);

JLabel labelLeft = new JLabel("Left Label");
JLabel labelRight = new JLabel("Right Label");
JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
        labelLeft, labelRight);
splitPane.setOneTouchExpandable(true);
topPanel.add(splitPane);
```



# JTable

The JTable component provides a table interface for the display of tabular data.




| Name | ID | Status | Active |
|---|---|---|---|
| Tom | 40566 | Regular | true |
| Mary | 90111 | Special | false |
| George | 80804 | Standby | true |
| Susan | 11228 | Regular | true |
| Henry | 56723 | Extra | false |

A simple JTable con be created using two arrays:

```
JTable table = new JTable(data, columnNames);
```

The first argument of the constructor is a one dimension array containing header information.  Te second argument is a two dimensional array that holds the data to be displayed.

```
String[] columnNames = {"Name", "ID", "Status", “Active"};

Object[][] data = {
      {"Tom", 40566, "Regular", "true"},
      {"Mary", 90111,"Special", "false"},
      {"George", 80804, "Standby", "true"},
      {"Susan", 11228, "Regular", "true"},
      {"Henry", 56723, "Extra", "false"}
};
```

If the JTable is placed inside of a JScrollPane the header is automatically displayed.

```
JScrollPane scrollPane = new JScrollPane(table);
```

Otherwise if you want the header to be present it will be necessary to used a BorderLayout type manager and place the header information in the top region and the table itself in the center.

This simple JTable allows the users to modify the data in the individual cells.  In order to capture these changes it is necessary to implement the TableModelListener interface and its tableChanged method.  Within the method the changes can be captured.

```
public void tableChanged(TableModelEvent e) {
      int row = e.getFirstRow();
      int column = e.getColumn();
      TableModel model = (TableModel)e.getSource();
      String columnName = model.getColumnName(column);
      Object data = model.getValueAt(row, column);
```

```
            // Process the data
      }
```

The complete code for the simple JTable application follows:

```
import java.awt.*;

import javax.swing.*;
import javax.swing.table.TableColumn;

public class JTableDemoWindow extends JFrame implements TableModelListener {

      public JTableDemoWindow() {
            super("JTable Demo");
            String[] columnNames = {"Name",
                        "ID",
                        "Status",
            "Active"};

            Object[][] data = {
                        {"Tom", 40566, "Regular", "true"},
                        {"Mary", 90111,"Special", "false"},
                        {"George", 80804, "Standby", "true"},
                        {"Susan", 11228, "Regular", "true"},
                        {"Henry", 56723, "Extra", "false"}
            };

            JTable table = new JTable(data, columnNames);
            JScrollPane scrollPane = new JScrollPane(table);

            Container container = this.getContentPane();
            container.setLayout(new BorderLayout());

            container.add(scrollPane,BorderLayout.CENTER);

            this.setBounds(100,100,300,150);
            this.setVisible(true);
      }

      public void tableChanged(TableModelEvent e) {
            int row = e.getFirstRow();
            int column = e.getColumn();
            TableModel model = (TableModel)e.getSource();
            String columnName = model.getColumnName(column);
            Object data = model.getValueAt(row, column);
            // Process the data
      }

}
```

# JTable TableModel

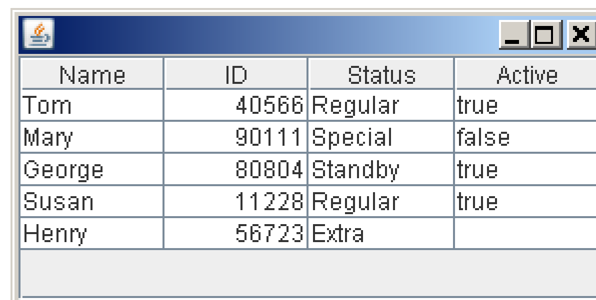
The JTable component uses a table model object to hold the table's data. This object provides data to table for display and stores cell changes. The model is essentially a repository for the data. How data is stored is up the implementer of the model. The data may be stored in a file, a database, randomly generated, or use whatever storage mechanism that makes sense for the application.

The table model has all of the information needed by the table. It informs the table of the number of rows and columns, the content of a given cell and even the way the data is to be displayed. The table knows nothing about the data, the data model provides this information as requested.

The important methods of the model include:

- getColumnCount – Returns the number of columns that make up the table
- getRowCount – Returns the number of rows that make up the table
- getColumnName – Returns the name of a specified column
- getValueAt – Returns the value for a specified cell
- setValueAt – Sets the value for a specified cell
- isCellEditable – Determines if a cell can be modified or not
- getColumnClass – Is concerned with how the data in a cell is displayed

To duplicate the previous JTable example we will need to create a class that implements the AbstractTableModel.



| Name | ID | Status | Active |
|---|---|---|---|
| Tom | 40566 | Regular | true |
| Mary | 90111 | Special | false |
| George | 80804 | Standby | true |
| Susan | 11228 | Regular | true |
| Henry | 56723 | Extra | |

Inside of this class we will use the same arrays that we used before though we could have chosen any other data source we needed. The declaration and population of the arrays is different because we will use different data values in later examples.

```
import javax.swing.table.AbstractTableModel;

public class CustomTableModel extends AbstractTableModel {

      private String[] columnNames;
      private Object[][] data;

      public CustomTableModel() {
            columnNames = new String[4];
            data = new Object[5][4];
```

```
            columnNames[0] = "Name";
            columnNames[1] = "ID";
            columnNames[2] = "Status";
            columnNames[3] = "Active";

            data[0][0] = "Tom";
            data[0][1] = 40566;
            data[0][2] = "Regular";
            data[0][3] = "true";

            data[1][0] = "Mary";
            data[1][1] = 90111;
            data[1][2] = "Special";
            data[1][3] = "false";

            data[2][0] = "George";
            data[2][1] = 80804;
            data[2][2] = "Standby";
            data[2][3] = "true";

            data[3][0] = "Susan";
            data[3][1] = 11228;
            data[3][2] = "Regular";
            data[3][3] = "true";

            data[4][0] = "Henry";
            data[4][1] = 56723;
            data[4][2] = "Extra";
      }
}
```

The complete example is as follows:

```
package cs230;

import javax.swing.table.AbstractTableModel;

public class CustomTableModel extends AbstractTableModel {

      private String[] columnNames;
      private Object[][] data;

      public CustomTableModel() {
            columnNames = new String[4];
            data = new Object[5][4];

            columnNames[0] = "Name";
            columnNames[1] = "ID";
            columnNames[2] = "Status";
            columnNames[3] = "Active";

            data[0][0] = "Tom";
            data[0][1] = 40566;
```

```
            data[0][2] = "Regular";
            data[0][3] = "true";

            data[1][0] = "Mary";
            data[1][1] = 90111;
            data[1][2] = "Special";
            data[1][3] = "false";

            data[2][0] = "George";
            data[2][1] = 80804;
            data[2][2] = "Standby";
            data[2][3] = "true";

            data[3][0] = "Susan";
            data[3][1] = 11228;
            data[3][2] = "Regular";
            data[3][3] = "true";

            data[4][0] = "Henry";
            data[4][1] = 56723;
            data[4][2] = "Extra";

//          data[0][3] = new Boolean(true);
//          data[1][3] = new Boolean(false);
//          data[2][3] = new Boolean(true);
//          data[3][3] = new Boolean(true);
//          data[4][3] = new Boolean(false);

      }

      public int getColumnCount() {
            return 4;
      }

      public String getColumnName(int col) {
            return columnNames[col];
      }


      public int getRowCount() {
            return 5;
      }

      public Object getValueAt(int rowIndex, int columnIndex) {
            return data[rowIndex][columnIndex];
      }

    public boolean isCellEditable(int rowIndex, int mColIndex) {
        return true;
    }

    public void setValueAt(Object value, int rowIndex, int columnIndex) {
      data[rowIndex][columnIndex] = value;
      System.out.println("Cell["+rowIndex+"]["+columnIndex+
                  "] changed to " + value);
    }

    public Class getColumnClass(int c) {
      return getValueAt(0,c).getClass();
    }
```

```
}
```

# Borders

javax.swing.border package consists of objects that will draw borders around components. These borders are useful in providing different appearances to an application. Three methods are supported:

- **getBorderInsets** – Used to define the area to draw the border around
- **isBorderOpaque** – Determines if the border is opaque or not
- **paintBorder** – Provides the ability to draw into the border area

The component **setBorder** method is used to specify the border to be used.

```
import javax.swing.border.*;
        …
        JPanel topPanel = new JPanel();
        topPanel.setLayout(new GridLayout(2,2));
        getContentPane().add(topPanel);

        JLabel labelEmpty = new JLabel("Empty", SwingConstants.CENTER);
        JLabel labelBevelUp = new JLabel("Bevel Up", SwingConstants.CENTER);
        JLabel labelBevelDown = new JLabel("Bevel Down", SwingConstants.CENTER);
        JLabel labelEtched = new JLabel("Etched", SwingConstants.CENTER);

        labelEmpty.setBorder(new EmptyBorder(1,1,1,1));
        labelBevelUp.setBorder(new BevelBorder(BevelBorder.RAISED));
        labelBevelDown.setBorder(new BevelBorder(BevelBorder.LOWERED));
        labelEtched.setBorder(new EtchedBorder());

        topPanel.add(labelEmpty);
        topPanel.add(labelBevelUp);
        topPanel.add(labelBevelDown);
        topPanel.add(labelEtched);
```

# Section 3: Menus

# Menus

Menus are an important part of most GUI applications. They allow the user to easily select from a number of different commands. A menu bar is normally found at the top of the application directly below the window's caption. When a menu is selected, a drop-down list of menu items appears for the user to choose from. These menu items may be cascaded in the form of sub-menus.

Characteristics of menus include:

- Menus are not normally a part of splash screens or dialog boxes
- Toolbars are often used for menu shortcuts
- Associated with menus are short-cut keys and accelerators or mnemonics.

Short-cut keys are frequently a keystroke combination, frequently using the control key. Mnemonics are identified with one letter of the menu name underscored and accessed using the Alt key. java supports both of these techniques.

The major parts of a menu system in Java include:

- Menu Bar – Used to hold the menus (JMenuBar)
- Menus – These are the drop-down list which hold menu items (JMenu)
- Menu items – The members of the menu that effect commands (JMenuItem)

# JMenuBar

The JMenuBar class is used to hold menus. A JMenuBar is created and then added to the container, usually a Frame or JFrame. The **setJMenuBar** is used to add a JMenuBar to a JFrame.

```
private JMenuBar menuBar = new JMenuBar();
…
setJMenuBar(menuBar);
```

The setJMenuBar method runs against a JFrame object.

A top level menu, such as File, Edit and such, are represented in Java by the JMenu class. These menus typically contain text and have drop-down menus. It is possible to add images to the menus though we don't see this done that frequently. When a JMenu object is created, its text is specified in the constructor.

```
private JMenu fileMenu = new JMenu("File");
```

The JMenu objects are added to a JMenuBar using the add method. The order in which JMenu objects are added to the JMenuBar is the same order that they will appear on the menu bar.

```
menuBar.add(fileMenu);
```

To create the drop-down menus, JMenuItems are added to a JMenu using the add method. The text used in the JMenuItem constructor will appear in the drop-down menu.

```
private JMenuItem fileNew = new JMenuItem("New");
private JMenuItem fileOpen = new JMenuItem("Open");

…

fileMenu.add(fileNew);
fileMenu.add(fileOpen);
fileMenu.add(fileSave);
fileMenu.add(fileExit);
```

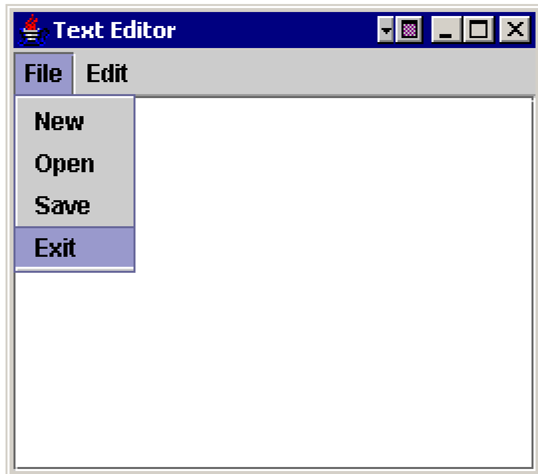
When a user selects a JMenuItem an ActionEvent object is created and passed to the **actionPerformed** method. By implementing the ActionListener interface, the programmer can capture and respond to menu selection.

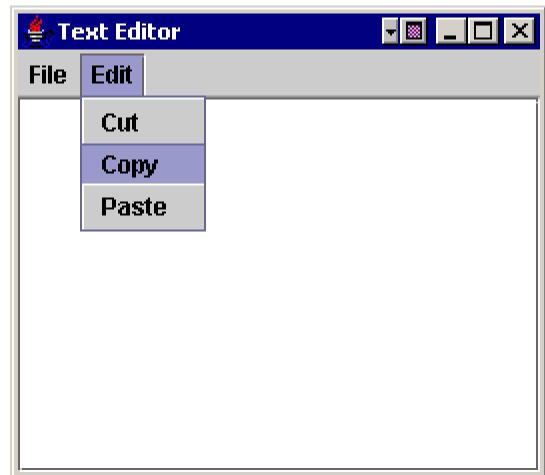
# Menu Demonstration

The TextEditor demonstrates the creation of a simple menu and how menu events can be handled.

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

class TextEditor extends JFrame implements WindowListener, ActionListener{
	private JTextArea textBox;

	private JMenuBar menuBar = new JMenuBar();
	private JMenu fileMenu = new JMenu("File");
	private JMenuItem fileNew = new JMenuItem("New");
	private JMenuItem fileOpen = new JMenuItem("Open");
	private JMenuItem fileSave = new JMenuItem("Save");
	private JMenuItem fileExit = new JMenuItem("Exit");
	private JMenu editMenu = new JMenu("Edit");
	private JMenuItem editCut = new JMenuItem("Cut");
	private JMenuItem editCopy = new JMenuItem("Copy");
	private JMenuItem editPaste = new JMenuItem("Paste");
	private JMenu specialMenu = new JMenu("Special");
	 private File currentFile;

	public TextEditor () {
		super("Text Editor");

		fileMenu.add(fileNew);
		fileMenu.add(fileOpen);
		fileMenu.add(fileSave);
		fileMenu.add(fileExit);
		editMenu.add(editCut);
		editMenu.add(editCopy);
		editMenu.add(editPaste);
		menuBar.add(fileMenu);
		menuBar.add(editMenu);
		setJMenuBar(menuBar);
		textBox = new JTextArea();
		JScrollPane s = new JScrollPane(textBox);
		Container a = getContentPane();
		a.setLayout(new BorderLayout());
		a.add(s);

		setSize(300,400);
		setVisible(true);
		this.addWindowListener(this);
```

```
		fileExit.addActionListener(this);
		fileOpen.addActionListener(this);
		fileSave.addActionListener(this);
		fileNew.addActionListener(this);
	}
	public void actionPerformed(ActionEvent e) {
		JFileChooser fc;

		Object o = e.getSource();
		JMenuItem mi = (JMenuItem) o;

		if (mi == fileExit) {
			System.exit(0);

		} else if(mi == fileOpen) {

		} else if(mi == fileSave) {

		} else if(mi == fileNew) {

		}
	}

	public void windowOpened(WindowEvent e) {}
	public void windowClosing(WindowEvent e) {
		System.exit(0);
	}
	public void windowClosed(WindowEvent e) {}
	public void windowIconified(WindowEvent e) {}
	public void windowDeiconified(WindowEvent e) {}
	public void windowActivated(WindowEvent e) {}
	public void windowDeactivated(WindowEvent e) {}

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


Text Editor
File
Edit
New
Open
Save
Exit

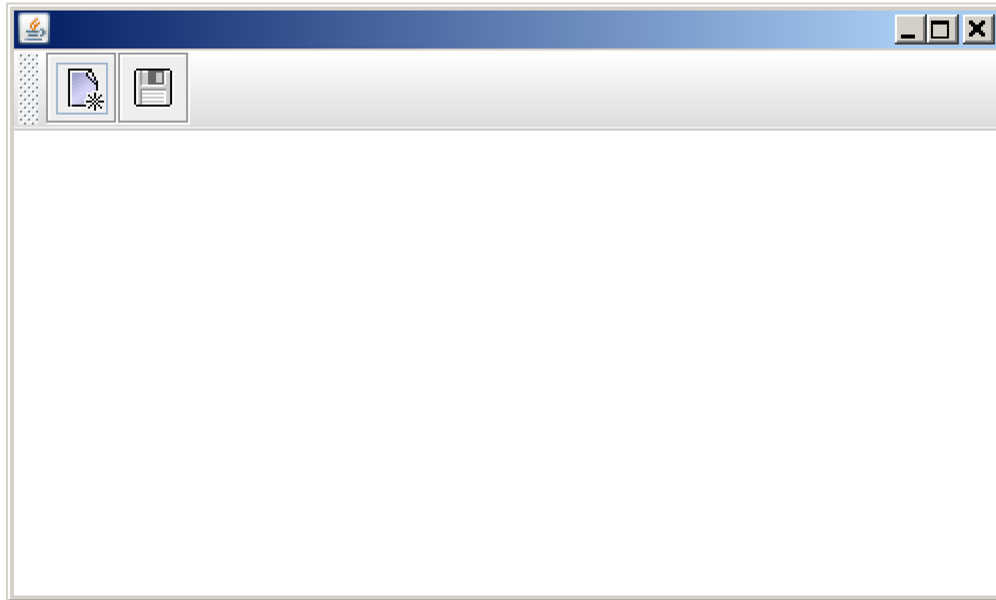

Text Editor
File
Edit
Cut
Copy
Paste

# Section 4: Tool Bars

# Creating Tool Bars

Tool bars are designed to provide a graphics representation of a series of choices. Tool bars often mimic menus and often duplicate the more commonly used menus.

Tool bars can be displayed either horizontally or vertically. The most common orientation is horizontally. Tool bars often permit themselves to be changed from one orientation to another by dragging the toolbar to a different border of the window. Tool bars can also be dragged outside of the container.



Tool bars normally contain a series of buttons though other controls can also be used. Tool bars can often be dragged, docked and undocked. In order to permit a tool bar to be dragged from one border to another the BorderLayout manager must be used. The toolbar must be the only component in the container and it must not be centered.

The JToolBar class is the basis for a tool bar. The following illustrates how a tool bar is created using a JPanel. In the constructor:

- BorderLayout is set
- A new JToolBar is created
- Buttons are added to the tool bar
- The tool bar is added to the panel

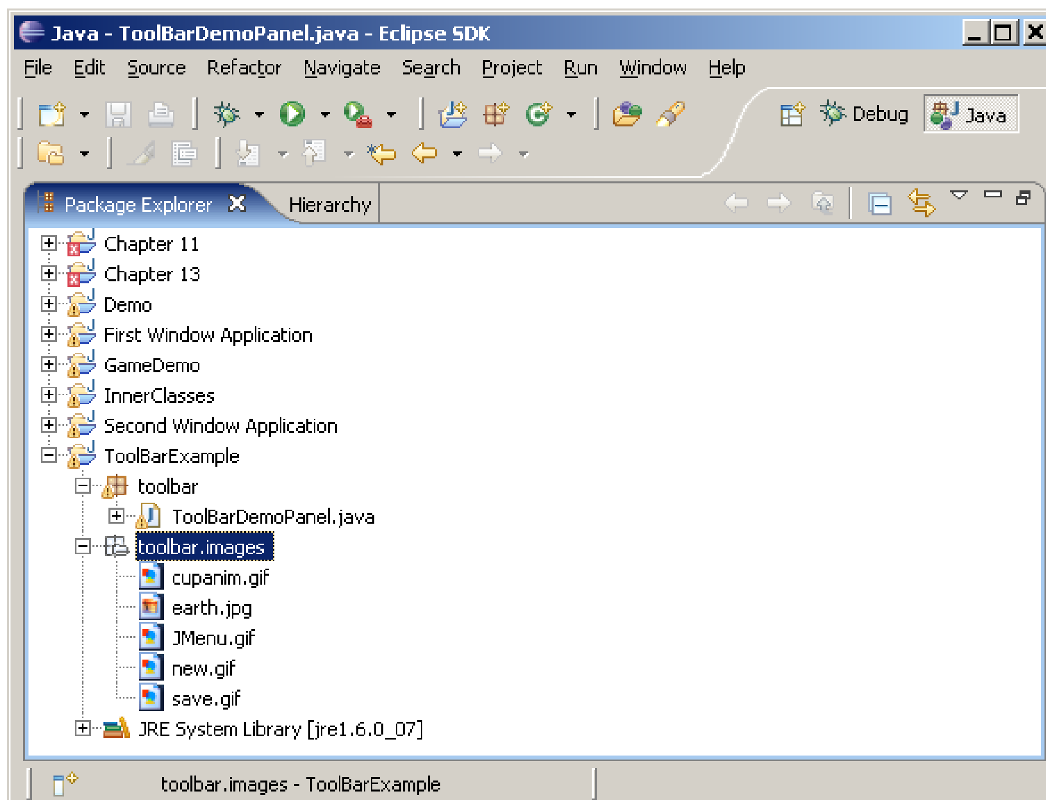
In this example an instance of the JToolBar is created and a method called addToolBarButtons is used to initialize the tool bar. The tool bar is then added to the application.

```
public class ToolBarDemoPanel extends JPanel implements ActionListener
{
	JButton exitButton;
	…

	public ToolBarDemoPanel() {
		super(new BorderLayout());
		JToolBar toolBar = new JToolBar();
		addToolBarButtons(toolBar);
		add(toolBar, BorderLayout.PAGE_START);
	}
```

The process for creating a single button is a multistep process. Depending on the requirements of the application the button can be customized with:

- An image
- Tooltip
- Actions

The creation of an exit button illustrates this basic process. In this example the image resources are founding the images subdirectory which is normally found at the same level as the applications package:



Images can be found at the Java look and fee Graphics repository at http://java.sun.com/developer/techDocs/hi/repository/.

Two buttons are added: a New button and a Save button.

```
public void addToolBarButtons(JToolBar toolBar) {
      toolBar.add(createToolBarButton(
                  "new.gif","New","New File",this));
      toolBar.add(createToolBarButton(
                  "save.gif","Save","Save the File",this));
}
```

The creation of a button can be encapsulated into a standard method.

```
JButton createToolBarButton(String imageName,
            String alternativeText, String toolTipText,
            ActionListener action) {

      // Create the JButton
      JButton button = new JButton();

      // Create and set the image
      String imageLocation = "images\\" + imageName;
      URL imageURL = getClass().getResource(imageLocation);
      if(imageURL != null) {
            ImageIcon imageIcon = new
                        ImageIcon(imageURL,alternativeText);
            button.setIcon(imageIcon);
      } else {
            System.out.println("Could not find the file");
      }

      // Set the tool tip text
      button.setToolTipText(toolTipText);

      // Add an action
      button.addActionListener(action);

      // Add the button to the tool bar
      return button;

}
```

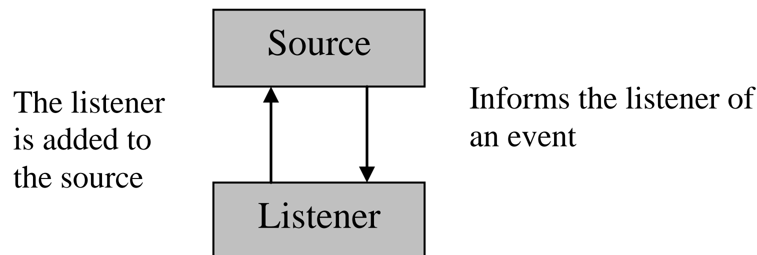
# Section 5: Event Handling

# Events

When a button or similar component is used as part of our interface, the question arises as to how the programmer knows when the user has pressed the button. Programs are written to respond to events. Events include actions such as:

- Clicking or movement of the mouse
- Keyboard entry
- Window closing or resizing

Java 1.1 added a source/listener paradigm to model events. The source will inform listeners when specific events occur.




In the previous application, the button is the source of events. The user may press the button, move the mouse over the button, or perform similar actions. The listener is the application itself. The application is interested when a specific event occurs.

In Java, the listener must tell the source of the event that it is:

1) Interested in listening to the source, and
2) Which events it is interested in listening to

Once the source is aware that a listener is interested in its events, it will add the listener to its list of listeners. When the event of interest occurs, the source will examine its list and then inform each listener that the event has occurred. While it may seem that there is a need for only one listener per program, later we will see that multiple listeners are useful at times.

# Listener Interfaces

Interfaces are an important part of the event model. The add*Some*Listener informs the source that an object is interested in certain events. The naming convention is quite consistent among the different interfaces. Each event interface supports a specific number and types of events.

Each interface is of the form *SomeListener.* Each method of an interface passes an object derived from the Event class and is of the form *SomeEvent.* This object can be used to obtain more information regarding the event and its specific capability is determined by the nature of the event. For example, a MouseEvent object has the X and Y position of the mouse.

One of the advantages to this approach is that a listener can specify only those types of events that it is interested in listening to and will not be troubled by other non-relevant events. It can listen to mouse events but ignore key board events.

# Event Interface

| Event Interface | Class Listener | Listener Methods |
|---|---|---|
| ActionEvent | ActionListener | actionPerformed() |
| AdjustmentEvent | AdjustmentListener | adjustmentValueChanged() |
| ComponentEvent | ComponentListener | componentHidden() |
| | | componentMoved() |
| | | componentResized() |
| | | componentShown() |
| ContainerEvent | ContainerListener | componentAdded() |
| | | componentRemoved() |
| FocusEvent | FocusListener | focusGained() |
| | | focusLost() |
| ItemEvent | ItemListener | itemStateChanged() |
| KeyEvent | KeyListener | keyPressed() |
| | | keyReleased() |
| | | keyTyped() |
| MouseEvent | MouseListener | mouseClicked() |
| | | mouseEntered() |
| | | mouseExited() |
| | | mousePressed() |
| | | mouseReleased() |
| | MouseMotionListener | mouseDragged() |
| | | mouseMoved() |
| TextEvent | TextListener | textValueChanged() |
| WindowEvent | WindowListener | windowActivated() |
| | | windowClosed() |
| | | windowClosing() |
| | | windowDeactivated() |
| | | windowDeiconified() |
| | | windowIconified() |
| | | windowOpened() |
| | | |
| | | |
| AncestorEvent | AncestorListener | ancestorAdded |
| | | ancestorMoved |
| | | ancestorRemoved |
| CaretEvent | CaretListener | caretUpdate |
| CellEditorEvent | CellEditorListener | editingCanceled |
| | | editingStopped |
| ChangeEvent | ChangeListener | stateChanged |
| HyperlinkEvent | HyperlinkListener | hyperlinkUpdate |

| Event Interface | Class Listener | Listener Methods |
|---|---|---|
| InternalFrameEvent | InternalFrameListener | internalFrameActivated |
| | | internalFrameClosed |
| | | internalFrameClosing |
| | | internalFrameDeactivated |
| | | internalFrameDeiconified |
| | | internalFrameIconified |
| | | internalFrameOpened |
| ListDataEvent | ListDataListener | contentChanged |
| | | intervalAdded |
| | | intervalRemoved |
| ListSelectionEvent | ListSelectionListener | valueChanged |
| MenuDragMouseEvent | MenuDragMouseListener | menuDragMouseDragged |
| | | menuDragMouseEntered |
| | | menuDragMouseExited |
| | | menuDragMouseReleased |
| MenuKeyEvent | MenuKeyListener | menuKeyPressed |
| | | menuKeyReleased |
| | | menuKeyTyped |
| MenuEvent | MenuListener | menuCanceled |
| | | menuDeselected |
| | | menuSelected |
| PopupMenuEvent | PopupMenuListener | popupMenuCanceled |
| | | popupMenuWillBecomeInvisible |
| | | popupMenuWillBecomeVisible |
| TreeExpansionEvent | TreeExpansionListener | treeCollapsed |
| | | treeExpanded |
| TreeSelection | TreeSelectionListener | valueChanged |
| TreeWillExpandEvent | TreeWillExpandListener | treeWillCollapse |
| | | treeWillExpand |
| VetoableChangeEvent | java.beans.<br>PropertyChangeListener.-<br>VetoableChangeListener | vetoableChange |

# Event Objects

Each event results in a method of a listener interface being called. Each method is invoked with an event object. The event object contains information regarding the event. Each event has a specific meaning.

| **Component** | **Events** | **Generated Meaning** |
|---|---|---|
| Button | ActionEvent | User clicked on the button |
| Checkbox | ItemEvent | User selected or deselected an item |
| CheckboxMenuItem | ItemEvent | User selected or deselected an item |
| Choice | ItemEvent | User selected or deselected an item |
| Component | ComponentEvent | Component moved, resized, hidden, or shown |
| | FocusEvent | Component gained or lost focus |
| | KeyEvent | User pressed or released a key |
| | MouseEvent | User pressed or released mouse button, mouse entered or exited component, or user moved or dragged mouse. |
| Container | ContainerEvent | Component added to or removed from container |
| List | ActionEvent | User double-clicked on list item |
| | ItemEvent | User selected or deselected an item |
| MenuItem | ActionEvent | User selected a menu item |
| Scrollbar | AdjustmentEvent | User moved the scrollbar |
| TextComponent | TextEvent | User changed text |
| TextField | ActionEvent | User finished editing text |
| Window | WindowEvent | Window opened, closed, iconified, deiconified, or close requested |

# getSource Method

If multiple buttons are used then there needs to be a way to distinguish between the buttons. The **getSource** method returns a reference to the source object. This can then be used with the **instanceof** and/or **equals** methods to determine the specific object that caused the event to occur.

```
public Object getSource();
```

```
public void actionPerformed(ActionEvent e) {
	if (e.getSource()  instanceof  Button)
		b.setLabel("Clicked!");
}
```

It is sometimes convenient to cast the object to its specific type and then compare it to an actual object. This will simplify the selection of multiple types of components of the same type.

```
public void actionPerformed(ActionEvent e) {
	if (e.getSource() instanceof Button) {
		Button SourceButton = (Button) e.getSource();
		if (SourceButton.equals(b))
			b.setLabel("Clicked!");
	}
}
```

# Action Commands

The **setActionCommand** method assigns a unique String to an object. The **getActionCommand** method returns the String. These are used as an alternate technique for determining the source of an event.

```
public void init () {
        b = new Button("Click Here!");
        b.addActionListener(this);
        b.setActionCommand("DemoButton");
        add(b);
}

public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("DemoButton")) {
                b.setLabel("Clicked!");
        }
}
```

By default, the action command for most components is the text string assigned to that component.

# MouseListener and MouseMotionListener

The MouseListener and MouseMotionListener interfaces are commonly used. They include a variety of methods that address common mouse events. Each method of these interfaces is passed a MouseEvent object. This object provides access to information such as the position of the mouse.

| Event Interface | Class Listener | Listener Methods |
|---|---|---|
| MouseEvent | MouseListener | mouseClicked() |
| | | mouseEntered() |
| | | mouseExited() |
| | | mousePressed() |
| | | mouseReleased() |
| | MouseMotionListener | mouseDragged() |
| | | mouseMoved() |

The **getX** and **getY** methods of the MouseEvent object return the x and y coordinates respectively.

```
public class EventDemo extends JFrame implements MouseListener {

    Button b;

    public EventDemo () {
        Container c = getContentPane();
        b = new Button("     Click Here!     ");
        b.addMouseListener(this);
        c.add(b);
    }

    public void mouseClicked(MouseEvent e) {}

    public void mouseEntered(MouseEvent e) {
        b.setLabel("Entered at x: " + e.getX());
    }

    public void mouseExited(MouseEvent e) {
        b.setLabel("Exited at x: " + e.getX());
    }

    public void mousePressed(MouseEvent e) {}

    public void mouseReleased(MouseEvent e) {}

}
```

# WindowListener Interface

The WindowListener interface provides a means of detecting and responding to window events:

- windowActivated
- windowClosed
- windowDeactivated
- windowDeiconified
- windowIconified
- windowOpened

To dispose of a window the window native resources need to be deleted and the program must be exited.

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

public class Demo extends JFrame implements WindowListener {

    public static void main(String args[]) {
        Demo app = new Demo("Window Listener Demonstration");
        app.setSize(300,100); app.show();
    }

    public Demo(String frameTitle) {
        super(frameTitle);
        JButton btnExit;
        JPanel bottomPanel = new JPanel();
        btnExit = new JButton("Exit");
        bottomPanel.add(btnExit);
        this.add(bottomPanel);
        addWindowListener(this);
        btnExit.addActionListener(this);
    }

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

    public void windowActivated(WindowEvent event) { }
    public void windowClosed(WindowEvent event) { }
    public void windowDeactivated(WindowEvent event) { }
    public void windowDeiconified(WindowEvent event) { }
    public void windowIconified(WindowEvent event) { }
    public void windowOpened(WindowEvent event) { }

}
```

# Actions

Actions are convenient ways to associate more than one event with an action. For example, the menu cut item command can be invoked in many applications with a button click or within the same application using different techniques. The code does not need to be duplicated. Instead an Action object can be created that encapsulates the command intended by the event along with other related information.

Classes that are declared within another class are called inner classes. There are several types of inner classes. The Action objects explained next use a simple inner class to define the action. They are contained within the outer application class. Being an inner class, they have access to all of the members of the outer class including private members.

An Action object provides a centralized location for handling activities. The object handles not only the action but also common text, icons, mnemonics, and the enabled or disabled status of the action component. Actions tend to be more expensive than the code used for typical ActionListener interface implementation but provide the benefit of centralized control.

The actionPerformed method of an Action object is executed in the same way that it is performed when implementing the ActionListener interface. Some implementations of the Action object use a doAction method that is invoked from the actionPerformed method.

The advantage of the doAction method is that it allows the actions to be executed without having to be able to create an ActionEvent and then calling the actionPerformed method. This restricts the activities in the doAction method to those that can be performed without the aid of the ActionEvent object.

A more detailed coverage of Actions can be found at http://java.sun.com/docs/books/tutorial/uiswing/misc/action.html.

The ExitAction class is used to capture the application exiting activities. Key points include:

- **putValue** – Sets the value for an action. The table below (http://java.sun.com/docs/books/tutorial/uiswing/misc/action.html) list the properties supported, the class it is used for, and its purpose

| **Property** | **Auto-Applied to: Class** *(Method Called)* | **Purpose** |
|---|---|---|
| ACCELERATOR_KEY | JMenuItem (*setAccelerator*) | The KeyStroke to be used as the accelerator for the action. For a discussion of accelerators versus mnemonics, see Enabling Keyboard Operation.. |
| ACTION_COMMAND_KEY | AbstractButton, JCheckBox, JRadioButton (*setActionCommand*) | The command string associated with the ActionEvent. |
| LONG_DESCRIPTION | None | The longer description for the action. Can be used for context-sensitive help. |
| MNEMONIC_KEY | AbstractButton, JMenuItem, JCheckBox, JRadioButton (*setMnemonic*) | The mnemonic for the action. For a discussion of accelerators versus mnemonics, see Enabling Keyboard Operation. |
| NAME | AbstractButton, JMenuItem, JCheckBox, JRadioButton (*setText*) | The name of the action. You can set this property when creating the action using the AbstractAction(String) or AbstractAction(String, Icon) constructors. |
| SHORT_DESCRIPTION | AbstractButton, JCheckBox, JRadioButton (*setToolTipText*) | The short description of the action. |
| SMALL_ICON | AbstractButton, JMenuItem (*setIcon*) | The icon for the action used in the tool bar or on a button. You can set this property when creating the action using the AbstractAction(name, icon) constructor. |

A more detailed description of actions can be found at http://java.sun.com/docs/books/tutorial/uiswing/misc/action.html.

```
class ExitAction extends AbstractAction {
	public ExitAction() {
		super("Exit");
		putValue(MNEMONIC_KEY,new Integer(KeyEvent.VK_X));
		putValue(SHORT_DESCRIPTION, "Exit the application");
	}

	public void actionPerformed(ActionEvent e) {
		doAction();
	}

	public void doAction() {
		System.exit(0);
	}
}
```

The ExitAction object is defined.

```
private ExitAction exitAction;
```

The ExitAction object is created.

```
exitAction = new ExitAction();
```

The ExitAction object is used instead of the System.exit(0) statement.

```
	frame.addWindowListener(new WindowAdapter() {
		public void windowClosing(WindowEvent e) {
			exitAction.doAction();
		}
	});
```

# Section 6: Look and Feel

# Look and Feel

The look and feel of an application refers to its visual appearance and how the user interacts with the GUI using mouse and keyboard events. In Swing a Model-View-Controller framework is used to implement the swing components. For each component there is:

- Model – That presents the data to the component
- View – The appearance of the component
- Controller – The user interaction

In swing the view and controller is somewhat bound together in a delegate. The data is usually presented in an underlying data model.

There are different standard look and feels available on most platforms. For example, on XP the following are available:

- MetalLookAndFeel
- MotifLookAndFeel
- WindowsLookAndFeel
- WindowsClassicLookAndFeel

Each of these provides a different look and behavior depending on the component being displayed. Custom look and feels can be created but this is a difficult task and is not covered here.

When the look and feel is changed it normally done for the entire application and not individual components. How this change is accomplished is explained here.

The UIManager class method, getInstalledLookAndFeels method returns an array of all look and feels installed on a platform:

```
	LookAndFeelInfo[] arr = UIManager.getInstalledLookAndFeels();
```

These can be displayed easily:

```
	for(LookAndFeelInfo laf : arr) {
			System.out.println(laf.getClassName());
	}

// Output
javax.swing.plaf.metal.MetalLookAndFeel
com.sun.java.swing.plaf.motif.MotifLookAndFeel
com.sun.java.swing.plaf.windows.WindowsLookAndFeel
com.sun.java.swing.plaf.windows.WindowsClassicLookAndFeel
```

To change the look and feel the setLookAndFeel method is used:

```
try {
      UIManager.setLookAndFeel(arr[i].getClassName());
} catch (ClassNotFoundException e) {
      e.printStackTrace();
} catch (InstantiationException e) {
      e.printStackTrace();
} catch (IllegalAccessException e) {
      e.printStackTrace();
} catch (UnsupportedLookAndFeelException e) {
      e.printStackTrace();
}
```

The following is a more complete example that illustrates adding a dynamic menu to an

```
LookAndFeelInfo[] arr = UIManager.getInstalledLookAndFeels();
for(LookAndFeelInfo laf : arr) {
      System.out.println(laf.getClassName());
      final LookAndFeelInfo tmp = laf;
      int index = tmp.getClassName().lastIndexOf('.')+1;
      String name = tmp.getClassName().substring(index);
      JMenuItem menu = new JMenuItem(name);
      menu.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ev) {
                  try {
                        UIManager.setLookAndFeel(tmp.getClassName());
                  } catch (ClassNotFoundException e) {
                        e.printStackTrace();
                  } catch (InstantiationException e) {
                        e.printStackTrace();
                  } catch (IllegalAccessException e) {
                        e.printStackTrace();
                  } catch (UnsupportedLookAndFeelException e) {
                        e.printStackTrace();
                  }
                  repaint();
            }
      });
      lookAndFeelMenu.add(menu);
}
```

# Summary

- The JFC provides a large set of objects that are used to provide a GUI interface
- Components are the building blocks of this interface
- Containers are objects that hold components
- There are numerous components available in Java
- The position of components is controlled by a layout manager
- Events are intercepted using various listener interfaces