

User Interface components with swing

A **graphical user interface (GUI)** presents a user-friendly mechanism for interacting with an application. A GUI (**pronounced “GOO-ee”**) gives an application a distinctive “look and feel.” GUIs are built from GUI components. These are sometimes called **controls or widgets**—short for window gadgets. A GUI component is an object with which the user interacts via the mouse, the keyboard or another form of input, such as voice recognition. The Swing GUI components are defined in the **javax.swing package**.

Different Java GUI controls

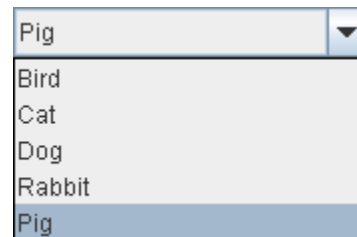
Basic Controls



JButton



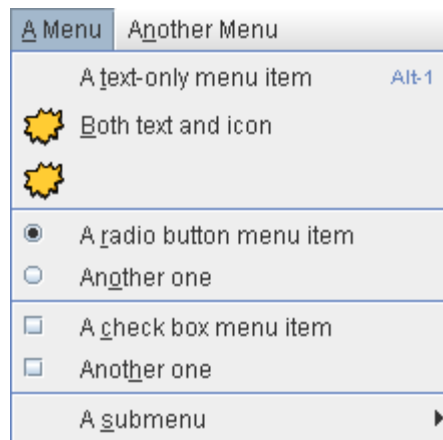
JCheckBox



JComboBox



JList



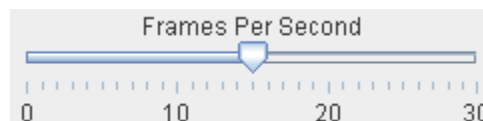
JMenu



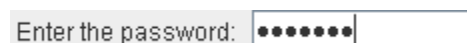
JRadioButton



JTextField

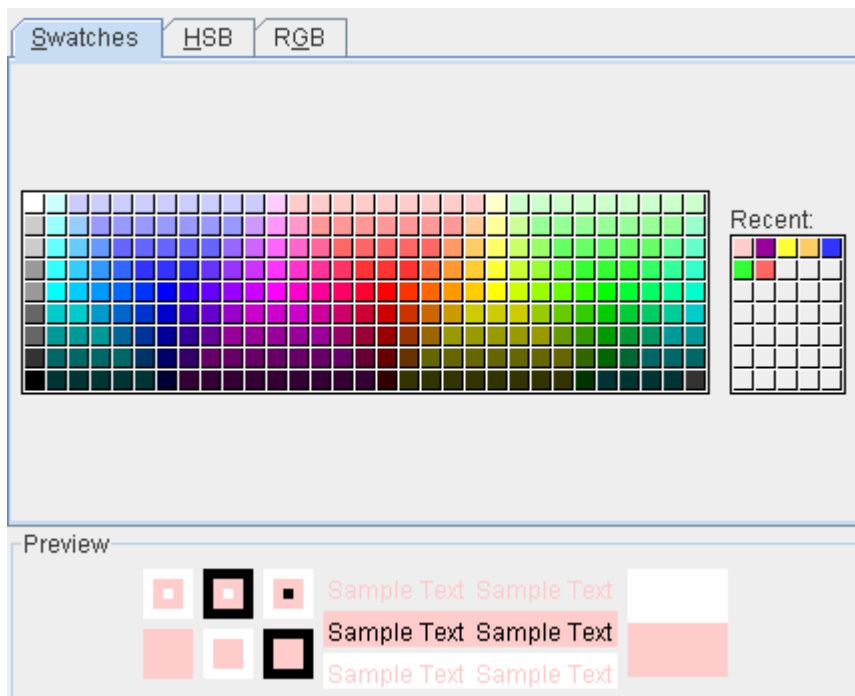


JSlider

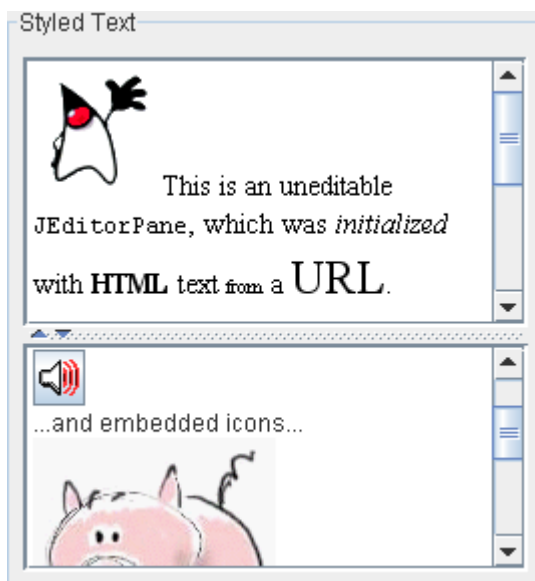


JPasswordField

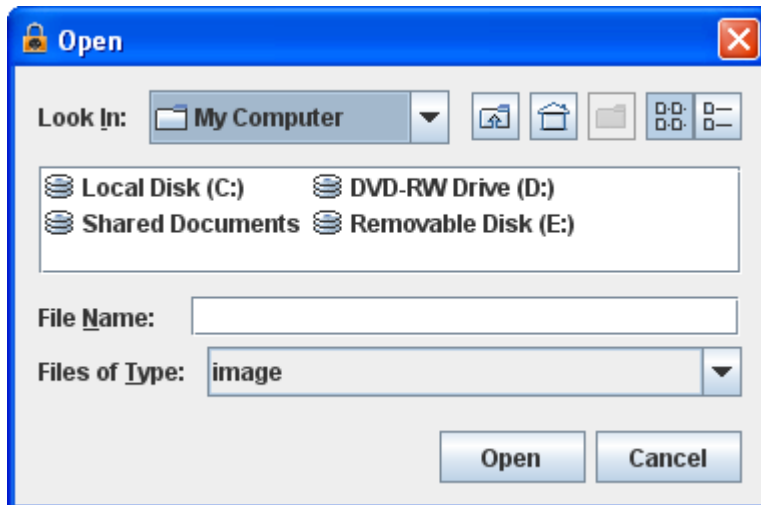
Interactive Controls



JColorChooser



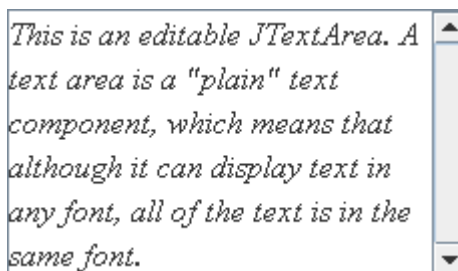
JEditorPane



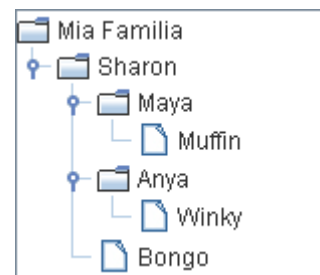
JFileChooser

Host	User	Password	Last Modified
Biocca Games	Freddy	!#asf6Awwzb	Mar 16, 2006
zabble	ichabod	Tazb!34\$fZ	Mar 6, 2006
Sun Developer	fraz@hotmail.co...	AasW541!fbZ	Feb 22, 2006
Heirloom Seeds	shams@gmail....	bkz[ADF78!	Jul 29, 2005
Pacific Zoo Shop	seal@hotmail.c...	vbAf124%z	Feb 22, 2006

JTable



JTextArea

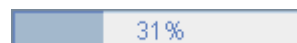


JTree

Uneditable Controls



JLabel

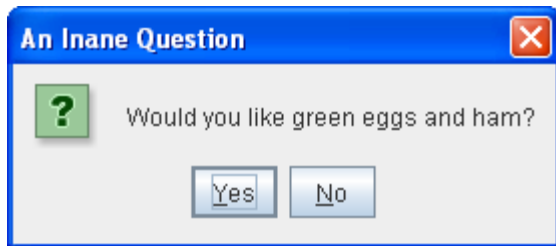


JProgressBar

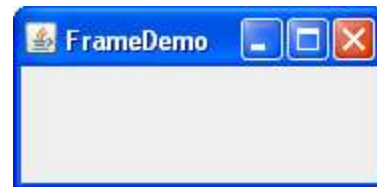


JToolTip

Top-Level Containers

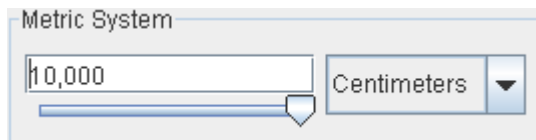


JDialog



JFrame

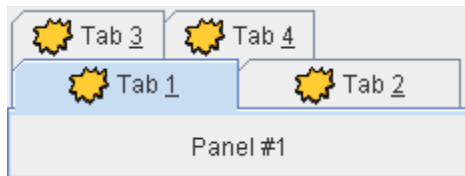
General Purpose Containers



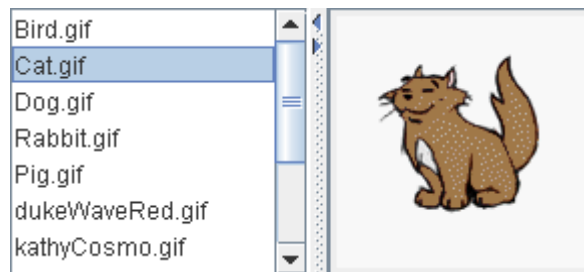
JPanel



JScrollPane



JTabbedPane



JSplitPane

Pls Visit the URL <http://docs.oracle.com/javase/tutorial/uiswing/components/index.html> for detailed study of Java GUI using Swing.

Overview of Swing Components

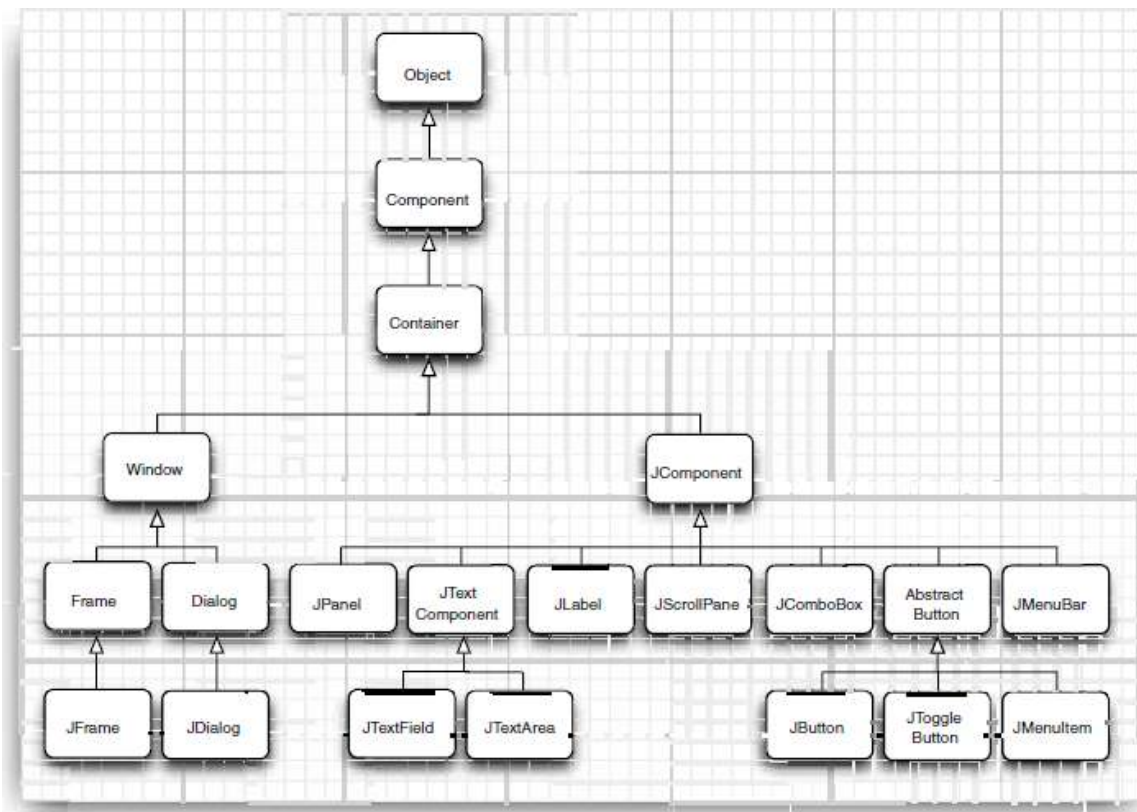


Fig 3. Inheritance hierarchy for the Component class

Component	Description
JLabel	Displays uneditable text and/or icons.
JTextField	Typically receives input from the user.
JButton	Triggers an event when clicked with the mouse.
JCheckBox	Specifies an option that can be selected or not selected.
JComboBox	A drop-down list of items from which the user can make a selection.
JList	A list of items from which the user can make a selection by clicking on any one of them. Multiple elements can be selected.
JPanel	An area in which components can be placed and organized.

fig. Some basic GUI components

Swing vs. AWT

There are actually two sets of Java GUI components. In Java's early days, GUIs were built with components from the **Abstract Window Toolkit (AWT)** in **package java.awt**. These look like the native GUI components of the platform on which a Java program executes. For example, a Button object displayed in a Java program running on Microsoft Windows looks like those in other Windows

applications. On Apple Mac OS X, the Button looks like those in other Mac applications. Sometimes, even the manner in which a user can interact with an AWT component differs between platforms. The component's appearance and the way in which the user interacts with it are known as its **look-and-feel**.

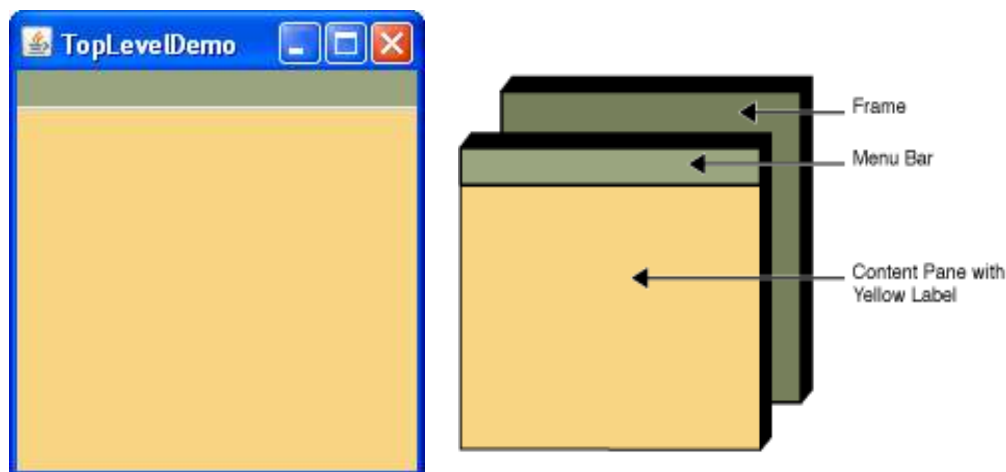
Swing GUI components allow you to specify a uniform look-and-feel for your application across all platforms or to use each platform's custom look-and-feel. An application can even change the look-and-feel during execution to enable users to choose their own preferred look-and-feel.

Most Swing components are **lightweight components**—they're written, manipulated and displayed completely in Java. AWT components are **heavyweight components**, because they rely on the local platform's windowing system to determine their functionality and their look-and-feel. Several Swing components are heavyweight components.

Java Top-Level Containers

Swing provides three generally useful **top-level container classes**: **JFrame**, **JDialog**, and **JApplet**. When using these classes, you should keep these facts in mind:

1. To appear onscreen, every GUI component must be part of a **containment hierarchy**. A containment hierarchy is a tree of components that has a **top-level container as its root**.
2. Each GUI component can be contained only once. If a component is already in a container and you try to add it to another container, the component will be removed from the first container and then added to the second.
3. Each top-level container has a **content pane** that, generally speaking, contains (directly or indirectly) the visible components in that top-level container's GUI.
4. You can optionally add a **menu bar** to a top-level container. The menu bar is by convention positioned within the top-level container, but outside the content pane.



Top-Level Containers and Containment Hierarchies

Each program that uses Swing components has **at least one top-level container**. This top-level container is the **root** of a **containment hierarchy** — the hierarchy that contains all of the Swing components that appear inside the top-level container.

As a rule, **a standalone application with a Swing-based GUI** has at least one containment hierarchy with **a JFrame as its root**. For example, if an application has **one main window** and **two dialogs**, then the application has **three containment hierarchies**, and thus **three top-level containers**. **One** containment hierarchy has a **JFrame as its root**, and **each of the other two** has a **JDialog object as its root**.

A Swing-based applet has at least one containment hierarchy, exactly one of which is rooted by a **JApplet** object. For example, an applet that brings up a dialog has two containment hierarchies. The components in the browser window are in a containment hierarchy rooted by a JApplet object. The dialog has a containment hierarchy rooted by a JDialog object.

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

public class TopLevelDemo {
    private static void createAndShowGUI() {
        //Create and set up the window.
        JFrame frame = new JFrame("TopLevelDemo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Create the menu bar. Make it have a green background.
        JMenuBar greenMenuBar = new JMenuBar();
        greenMenuBar.setOpaque(true);
        greenMenuBar.setBackground(new Color(154, 165, 127));
        greenMenuBar.setPreferredSize(new Dimension(200, 20));

        //Create a yellow label to put in the content pane.
        JLabel yellowLabel = new JLabel();
        yellowLabel.setOpaque(true);
        yellowLabel.setBackground(new Color(248, 213, 131));
        yellowLabel.setPreferredSize(new Dimension(200, 180));

        //Set the menu bar and add the label to the content pane.
        frame.setJMenuBar(greenMenuBar);
        frame.getContentPane().add(yellowLabel, BorderLayout.CENTER);

        /*
        //Create a panel and add components to it.
        JPanel contentPane = new JPanel(new BorderLayout());
        contentPane.setBorder(someBorder);
        contentPane.add(someComponent, BorderLayout.CENTER);
        contentPane.add(anotherComponent, BorderLayout.PAGE_END);
        topLevelContainer.setContentPane(contentPane);
        */
    }
}
```

```

//Display the window.
    frame.pack();
    frame.setVisible(true);
}

public static void main(String[] args) {
    //Schedule a job for the event-dispatching thread:
    //creating and showing this application's GUI.
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            createAndShowGUI();
        }
    });
}
} //end of class TopLevelDemo

```

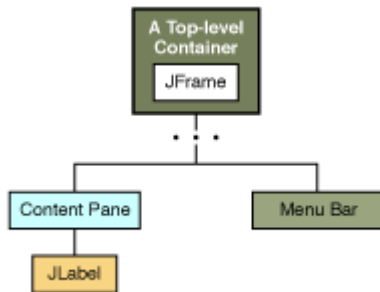


fig. the containment hierarchy for this example's GUI

Creating a Frame

A top-level window (that is, a window that is not contained inside another window) is called a **frame** in Java. The AWT library has a class, called `Frame`, for this top level. The Swing version of this class is called `JFrame` and extends the `Frame` class.

```

import javax.swing.*;

public class SimpleFrameTest
{
    public static void main(String[] args)
    {
        SimpleFrame frame = new SimpleFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

class SimpleFrame extends JFrame

```



```

{
public SimpleFrame()
{
setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
}
public static final int DEFAULT_WIDTH = 300;
public static final int DEFAULT_HEIGHT = 200;
}

```



fig. A simple frame

Swing GUI components are instances of class `JFrame` or a subclass of `JFrame`. `JFrame` is an indirect subclass of class `java.awt.Window` that provides the basic attributes and behaviors of a window—a title bar at the top, and buttons to minimize, maximize and close the window.

Buttons, text fields, and other user interface elements extend the class `Component`. Components can be placed inside containers such as panels.

By default, closing a window simply hides the window. However, when the user closes the frame, we would like the application to terminate. **setDefaultCloseOperation method (inherited from class `JFrame`) with constant `JFrame.EXIT_ON_CLOSE` as the argument** indicate that the program should terminate when the window is closed by the user.

Dialog Box Using Swing

Dialog boxes are windows in which programs display important messages to the user or obtain information from the user. Most applications you use on a daily basis use windows or dialog boxes (also called **dialogs**) to interact with the user.

Java's **`JOptionPane` class (package `javax.swing`)** provides prebuilt dialog boxes for both input and output. These are displayed by invoking **static `JOptionPane` methods**. Program below presents a simple addition application that uses two input dialogs to obtain integers from the user and a message dialog to display the sum of the integers the user enters.

// Fig. 14.2: Addition.java

// Addition program that uses `JOptionPane` for input and output.

```

import javax.swing.JOptionPane;           // program uses JOptionPane
public class Addition

```

```

{
public static void main( String[] args )
{
    // obtain user input from JOptionPane input dialogs
    String firstNumber =
        JOptionPane.showInputDialog( "Enter first integer" );
    String secondNumber =
        JOptionPane.showInputDialog( "Enter second integer" );
    // convert String inputs to int values for use in a calculation
    int number1 = Integer.parseInt( firstNumber );
    int number2 = Integer.parseInt( secondNumber );
    int sum = number1 + number2; // add numbers
    // display result in a JOptionPane message dialog
    JOptionPane.showMessageDialog( null, "The sum is " + sum,
        "Sum of Two Integers", JOptionPane.PLAIN_MESSAGE );
} // end method main
} // end class Addition

```

Output

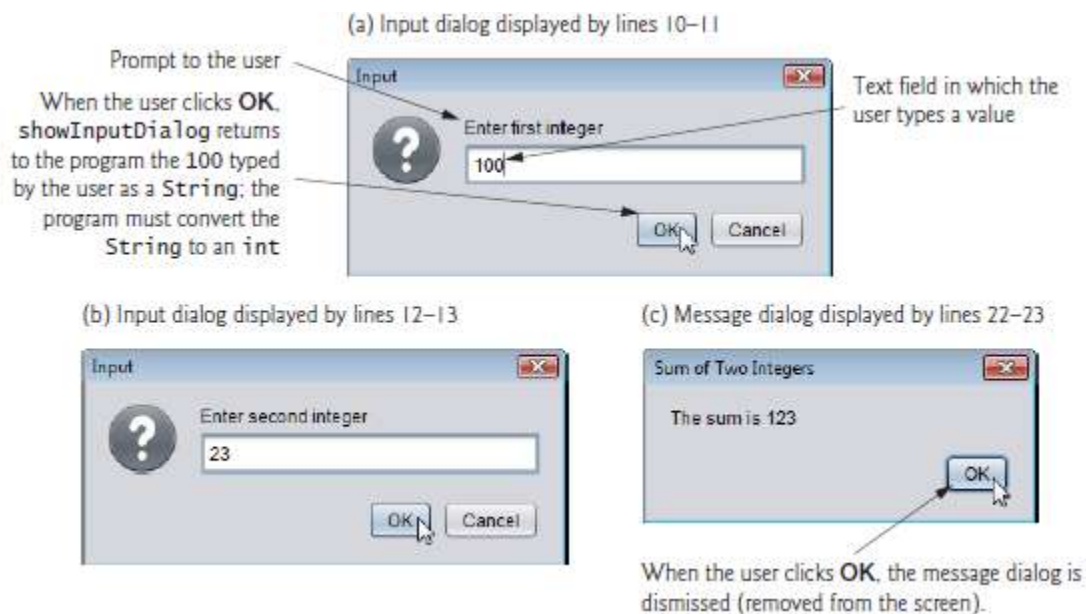


fig 1

Input Dialogs

Line 3 imports class `JOptionPane`. Lines 10–11 declare the local `String` variable `firstNumber` and assign it the result of the call to **`JOptionPane` static method `showInputDialog`**. This method displays an input dialog using the method's `String` argument ("Enter first integer") as a prompt.

Message Dialogs

Lines 22–23 use `JOptionPane` static method `showMessageDialog` to display a message dialog (the last screen of Fig. 1) containing the sum. The **first argument** helps the Java application determine **where to**

position the dialog box. A dialog is typically displayed from a GUI application with its own window. The first argument refers to that window (known as the parent window) and causes the dialog to appear centered over the parent. If the first argument is null, the dialog box is displayed at the center of your screen. The **second argument** is the message to display—in this case, **the result of concatenating the String "The sum is " and the value of sum.** The **third argument**—"Sum of Two Integers"—is the String that should appear in the title bar at the top of the dialog. The fourth argument—**JOptionPane.PLAIN_MESSAGE**—is the type of message dialog to display. A PLAIN_MESSAGE dialog does not display an icon to the left of the message.

JOptionPane Message Dialog Constants

The constants that represent the message dialog types are shown in Fig. 2. All message dialog types except PLAIN_MESSAGE display an icon to the left of the message. These icons provide a visual indication of the message's importance to the user. A QUESTION_MESSAGE icon is the default icon for an input dialog box (see Fig. 2).





Message dialog type	Icon	Description
ERROR_MESSAGE		Indicates an error.
INFORMATION_MESSAGE		Indicates an informational message.
WARNING_MESSAGE		Warns of a potential problem.
QUESTION_MESSAGE		Poses a question. This dialog normally requires a response, such as clicking a Yes or a No button.
PLAIN_MESSAGE	no icon	A dialog that contains a message, but no icon.

fig. 2

JLabel

A typical GUI consists of many components. GUI designers often provide text stating the purpose of each components. Such text is known as a **label** and is created with a **JLabel**—a subclass of JComponent. **A JLabel displays read-only text, an image, or both text and an image.** Applications rarely change a label's contents after creating it.

```
// LabelFrame.java
// Demonstrating the JLabel class.
import java.awt.FlowLayout;           // specifies how components are arranged
import javax.swing.JFrame;            // provides basic window features
import javax.swing.JLabel;            // displays text and images
import javax.swing.SwingConstants;    // common constants used with Swing
import javax.swing.Icon;              // interface used to manipulate images
import javax.swing.ImageIcon;         // loads images
```

```

public class LabelFrame extends JFrame
{
    private JLabel label1;           // JLabel with just text
    private JLabel label2;           // JLabel constructed with text and icon
    private JLabel label3;           // JLabel with added text and icon

    // LabelFrame constructor adds JLabels to JFrame
    public LabelFrame()
    {
        super( "Testing JLabel" );
        setLayout( new FlowLayout() );    // set frame layout

        // JLabel constructor with a string argument
        label1 = new JLabel( "Label with text" );
        label1.setToolTipText( "This is label1" );
        add( label1 ); // add label1 to JFrame

        // JLabel constructor with string, Icon and alignment arguments
        Icon bug = new ImageIcon( getClass().getResource( "bug1.png" ) );
        label2 = new JLabel( "Label with text and icon", bug,
            SwingConstants.LEFT );
        label2.setToolTipText( "This is label2" );
        add( label2 );    // add label2 to JFrame

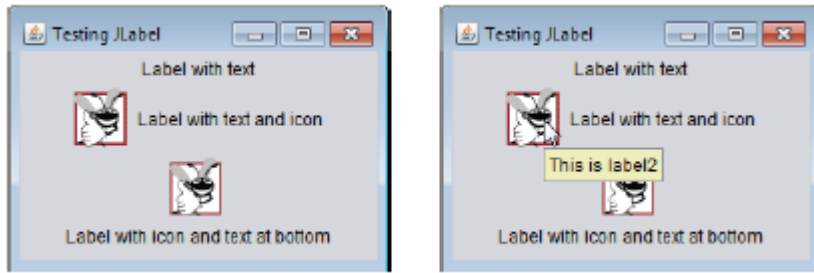
        label3 = new JLabel();           // JLabel constructor no arguments
        label3.setText( "Label with icon and text at bottom" );
        label3.setIcon( bug );           // add icon to JLabel
        label3.setHorizontalTextPosition( SwingConstants.CENTER );
        label3.setVerticalTextPosition( SwingConstants.BOTTOM );
        label3.setToolTipText( "This is label3" );
        add( label3 );    // add label3 to JFrame
    } // end LabelFrame constructor
} // end class LabelFrame

// LabelTest.java
// Testing LabelFrame.
import javax.swing.JFrame;

public class LabelTest
{
    public static void main( String[] args )
    {
        LabelFrame labelFrame = new LabelFrame();    // create LabelFrame
        labelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        labelFrame.setSize( 260, 180 );    // set frame size
        labelFrame.setVisible( true );    // display frame
    } // end main
} // end class LabelTest

```

output



Introduction to event handling

Normally, a user interacts with an application's GUI to indicate the tasks that the application should perform. For example, when you write an e-mail in an e-mail application, clicking the Send button tells the application to send the e-mail to the specified e-mail addresses. **GUIs are event driven.** When the user interacts with a GUI component, the interaction—known as an **event**—**drives the program to perform a task.** Some common user interactions that cause an application to perform a task include **clicking a button, typing in a text field, selecting an item from a menu, closing a window and moving the mouse.** The code that performs a task in response to an event is called an **event handler**, and the overall process of responding to events is known as **event handling.**

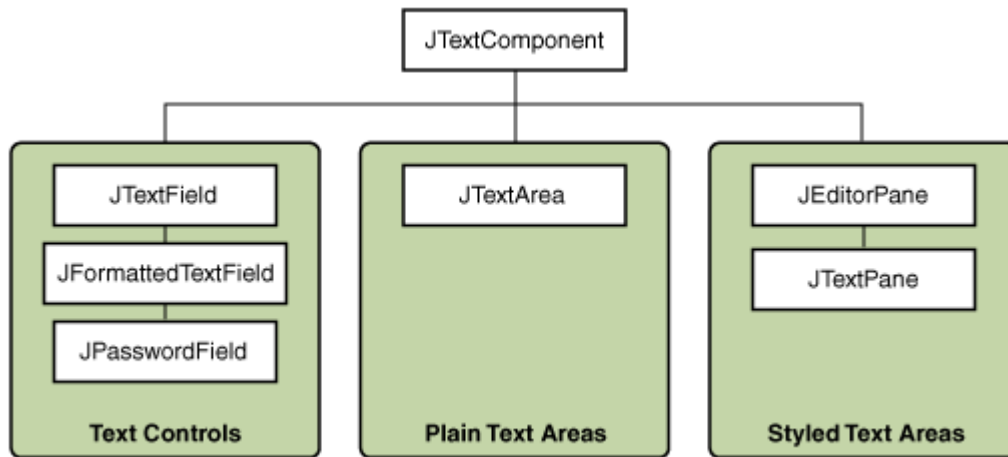
Steps Required to Set Up Event Handling for a GUI Component

Before an application can respond to an event for a particular GUI component, you must:

1. Create a class that represents the event handler and implements an appropriate interface—known as an **event-listener interface.**
2. Indicate that an object of the class from Step 1 should be notified when the event occurs—known as **registering the event handler.**

Text Field with event handling

A **text field** is a basic text control that enables the user to type a small amount of text. When the user indicates that text entry is complete (usually by pressing Enter), the text field fires an **action event.** If you need to obtain **more than one line** of input from the user, use a **text area.**



// Demonstrating the JTextField class.

```

import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JPasswordField;
import javax.swing.JOptionPane;

```

```

public class TextFieldFrame extends JFrame
{
    private JTextField textField1; // text field with set size
    private JTextField textField2; // text field constructed with text
    private JTextField textField3; // text field with text and size
    private JPasswordField passwordField; // password field with text

```

// TextFieldFrame constructor adds JTextFields to JFrame

```

public TextFieldFrame()
{
    super( "Testing JTextField and JPasswordField" );
    setLayout( new FlowLayout() ); // set frame layout

```

// construct textfield with 10 columns

```

textField1 = new JTextField( 10 );
add( textField1 ); // add textField1 to JFrame

```

// construct textfield with default text

```

textField2 = new JTextField( "Enter text here" );
add( textField2 ); // add textField2 to JFrame

```

// construct textfield with default text and 21 columns

```

textField3 = new JTextField( "Uneditable text field", 21 );
textField3.setEditable( false ); // disable editing
add( textField3 ); // add textField3 to JFrame

```

```

// construct passwordfield with default text
passwordField = new JPasswordField( "Hidden text" );
add( passwordField ); // add passwordField to JFrame

// register event handlers
TextFieldHandler handler = new TextFieldHandler();
textField1.addActionListener( handler );
textField2.addActionListener( handler );
textField3.addActionListener( handler );
passwordField.addActionListener( handler );

} // end TextFieldFrame constructor

// private inner class for event handling
private class TextFieldHandler implements ActionListener
{
    // process text field events
    public void actionPerformed((ActionEvent event) )
    {
        String string = ""; // declare string to display

        // user pressed Enter in JTextField textField1
        if(event.getSource() == textField1)
            string = String.format( "textField1: %s",event.getActionCommand());

        // user pressed Enter in JTextField textField2
        else if(event.getSource() == textField2 )
            string = String.format( "textField2: %s",event.getActionCommand() );

        // user pressed Enter in JTextField textField3
        else if(event.getSource() == textField3 )
            string = String.format( "textField3: %s",event.getActionCommand());

        // user pressed Enter in JTextField passwordField
        else if(event.getSource() == passwordField )
            string = String.format( "passwordField: %s",event.getActionCommand() );

        // display JTextField content
        JOptionPane.showMessageDialog( null, string );
    } // end method actionPerformed
} // end private inner class TextFieldHandler
} // end class TextFieldFrame

// Testing TextFieldFrame.
import javax.swing.JFrame;

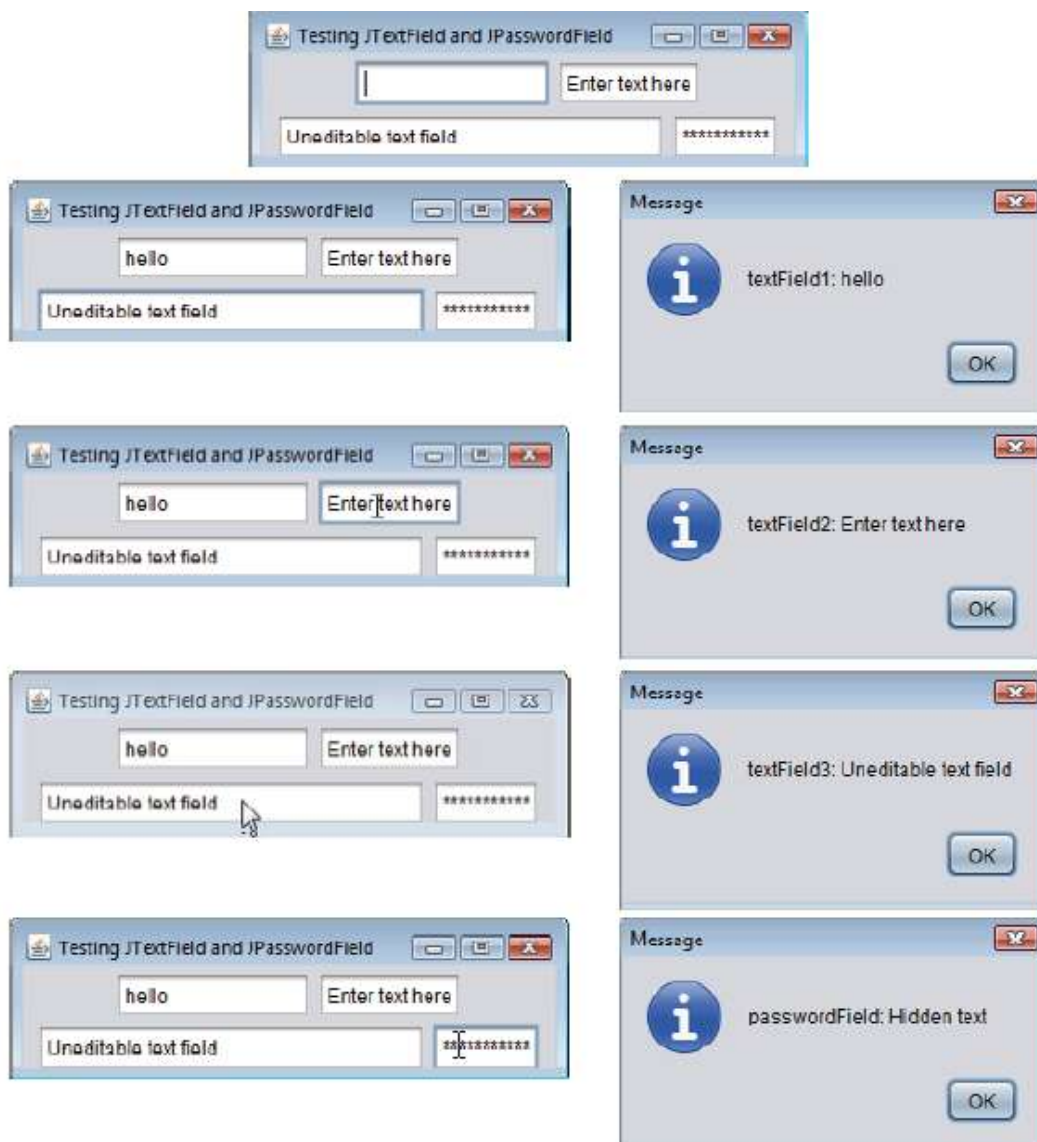
```

```

public class TextFieldTest
{
public static void main( String[] args )
{
TextFieldFrame textFieldFrame = new TextFieldFrame();
textFieldFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
textFieldFrame.setSize( 350, 100 ); // set frame size
textFieldFrame.setVisible( true ); // display frame
} // end main
} // end class TextFieldTest

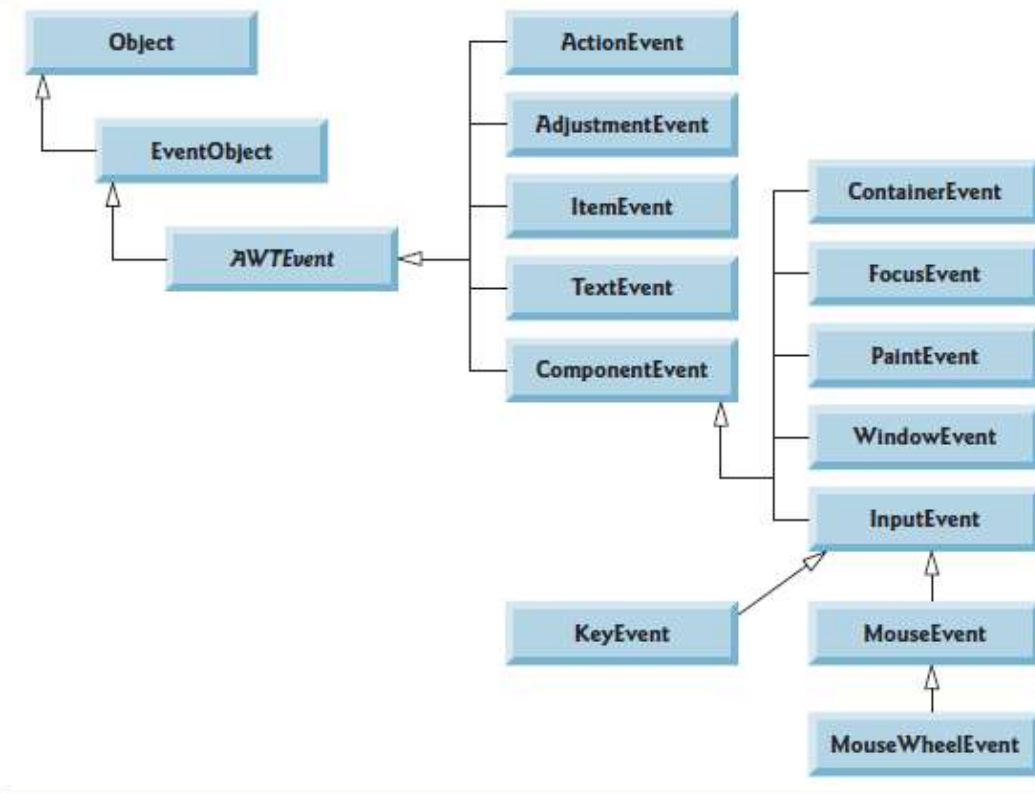
```

output



Common GUI Event Types and Listener Interfaces

Many different types of events can occur when the user interacts with a GUI. The event information is stored in an object of a class that extends **AWTEvent** (from package **java.awt**). Figure below illustrates a hierarchy containing many event classes from the package **java.awt.event**. These event types are used with both AWT and Swing components. Additional event types that are specific to Swing GUI components are declared in package **javax.swing.event**.



Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract super class for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.

KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved. (Added by Java 2, version 1.4)
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

fig. Some event classes of package java.awt.event

The event-handling mechanism mainly consist of three parts—the **event source**, the **event object** and the **event listener**. The **event source** is the GUI component with which the user interacts. The **event object** encapsulates information about the event that occurred, such as a reference to the event source and any event-specific information that may be required by the event listener for it to handle the event. The **event listener** is an object that's notified by the event source when an event occurs; in effect, it "listens" for an event, and one of its methods executes in response to the event. A method of the event listener receives an event object when the event listener is notified of the event. The event listener then uses the event object to respond to the event. This event-handling model is known as the **delegation event model**—an event's processing is delegated to an object (the event listener) in the application.

Event Source	Description
Button	Generates action events when the button is pressed.
Checkbox	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scrollbar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.


Table  *Event Source Examples*

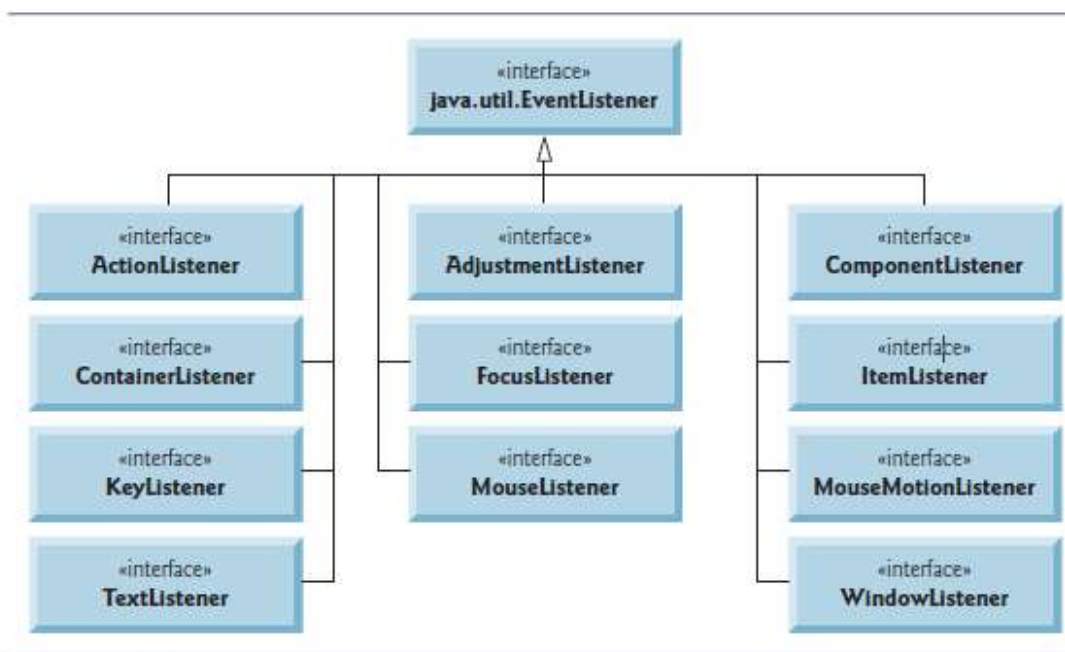
fig. Event Source Examples

For each event-object type, there's typically a **corresponding event-listener interface**. An event listener for a GUI event is an object of a class that implements one or more of the event-listener interfaces from

packages `java.awt.event` and `javax.swing.event`. Many of the event-listener types are common to both Swing and AWT components. Such types are declared in package `java.awt.event`, and some of them are shown in Fig. below. Additional event-listener types that are specific to Swing components are declared in package `javax.swing.event`.

Each event-listener interface specifies one or more event-handling methods that must be declared in the class that implements the interface. **Any class which implements an interface must declare all the abstract methods of that interface;** otherwise, the class is an abstract class and cannot be used to create objects.

When an event occurs, the GUI component with which the user interacted **notifies its registered listeners by calling each listener's appropriate event-handling method**. For example, when the user presses the Enter key in a `JTextField`, the registered listener's `actionPerformed` method is called.



Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved. (Added by Java 2, version 1.4)
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus. (Added by Java 2, version 1.4)
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.


Table  *Commonly Used Event Listener Interfaces*

fig. Some common event-listener interfaces of package java.awt.event

The ActionListener Interface

This interface defines the **actionPerformed()** method that is invoked when an action event occurs. Its general form is shown here:

```
void actionPerformed(ActionEvent ae)
```

The AdjustmentListener Interface

This interface defines the **adjustmentValueChanged()** method that is invoked when an adjustment event occurs. Its general form is shown here:

```
void adjustmentValueChanged(AdjustmentEvent ae)
```

The ComponentListener Interface

This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:

```
void componentResized(ComponentEvent ce)
void componentMoved(ComponentEvent ce)
void componentShown(ComponentEvent ce)
void componentHidden(ComponentEvent ce)
```

Note

*The AWT processes the resize and move events. The **componentResized()** and **componentMoved()** methods are provided for notification purposes only.*

The ContainerListener Interface

This interface contains two methods. When a component is added to a container, **componentAdded()** is invoked. When a component is removed from a container, **componentRemoved()** is invoked. Their general forms are shown here:

```
void componentAdded(ContainerEvent ce)
void componentRemoved(ContainerEvent ce)
```

The FocusListener Interface

This interface defines two methods. When a component obtains keyboard focus, **focusGained()** is invoked. When a component loses keyboard focus, **focusLost()** is called. Their general forms are shown here:

```
void focusGained(FocusEvent fe)
void focusLost(FocusEvent fe)
```


The ItemListener Interface

This interface defines the **itemStateChanged()** method that is invoked when the state of an item changes. Its general form is shown here:

```
void itemStateChanged(ItemEvent ie)
```

The KeyListener Interface

This interface defines three methods. The **keyPressed()** and **keyReleased()** methods are invoked when a key is pressed and released, respectively. The **keyTyped()** method is invoked when a character has been entered.

For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released. If a user presses and releases the HOME key, two key events are generated in sequence: key pressed and released.

The general forms of these methods are shown here:

```
void keyPressed(KeyEvent ke)
void keyReleased(KeyEvent ke)
void keyTyped(KeyEvent ke)
```

The MouseListener Interface

This interface defines five methods. If the mouse is pressed and released at the same point, **mouseClicked()** is invoked. When the mouse enters a component, the **mouseEntered()** method is called. When it leaves, **mouseExited()** is called. The **mousePressed()** and **mouseReleased()** methods are invoked when the mouse is pressed and released, respectively.

The general forms of these methods are shown here:

```
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
```

The MouseMotionListener Interface

This interface defines two methods. The **mouseDragged()** method is called multiple times as the mouse is dragged. The **mouseMoved()** method is called multiple times as the mouse is moved. Their general forms are shown here:

```
void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)
```

JCheckBox

The MouseWheelListener Interface

This interface defines the **mouseWheelMoved()** method that is invoked when the mouse wheel is moved. Its general form is shown here.

```
void mouseWheelMoved(MouseWheelEvent mwe)
```

MouseWheelListener was added by Java 2, version 1.4.

The TextListener Interface

This interface defines the **textChanged()** method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

```
void textChanged(TextEvent te)
```

The WindowFocusListener Interface

This interface defines two methods: **windowGainedFocus()** and **windowLostFocus()**. These are called when a window gains or losses input focus. Their general forms are shown here.

```
void windowGainedFocus(WindowEvent we)
void windowLostFocus(WindowEvent we)
```

WindowFocusListener was added by Java 2, version 1.4.

The WindowListener Interface

This interface defines seven methods. The **windowActivated()** and **windowDeactivated()** methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the **windowIconified()** method is called. When a window is deiconified, the **windowDeiconified()** method is called. When a window is opened or closed, the **windowOpened()** or **windowClosed()** methods are called, respectively. The **windowClosing()** method is called when a window is being closed. The general forms of these methods are

```
void windowActivated(WindowEvent we)
void windowClosed(WindowEvent we)
void windowClosing(WindowEvent we)
void windowDeactivated(WindowEvent we)
void windowDeiconified(WindowEvent we)
void windowIconified(WindowEvent we)
void windowOpened(WindowEvent we)
```

JButton

A **button** is a component **the user clicks to trigger a specific action**. A Java application can use several types of buttons, including **command buttons, checkboxes, toggle buttons and radio buttons**. Figure below shows the inheritance hierarchy of the **Swing buttons**. As you can see, all the button types are subclasses of **AbstractButton** (package `javax.swing`), which declares the common features of Swing buttons.

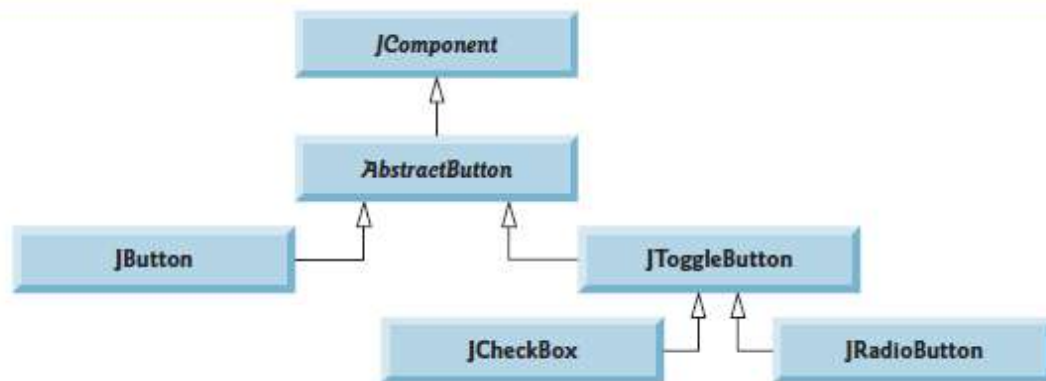


fig. Swing button hierarchy.

A **command button** generates an `ActionEvent` when the user clicks it. Command buttons are created with class `JButton`. The text on the face of a `JButton` is called a **button label**. A GUI can have many `JButtons`, but each button label should be unique in the portion of the GUI that's currently displayed.

The application below creates two `JButtons` and demonstrates that `JButtons` support the display of icons. Event handling for the buttons is performed by a single instance of inner class `ButtonHandler`.

```

// ButtonFrame.java
// Creating JButtons.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JOptionPane;

public class ButtonFrame extends JFrame
{
    private JButton plainJButton; // button with just text
    private JButton fancyJButton; // button with icons

    // ButtonFrame adds JButtons to JFrame
    public ButtonFrame()
    {
        super( "Testing Buttons" );
        setLayout( new FlowLayout() ); // set frame layout

        plainJButton = new JButton( "Plain Button" ); // button with text
        add( plainJButton ); // add plainJButton to JFrame
    }
}

```



```

Icon bug1 = new ImageIcon( getClass().getResource( "bug1.gif" ) );
Icon bug2 = new ImageIcon( getClass().getResource( "bug2.gif" ) );
fancyJButton = new JButton( "Fancy Button", bug1 ); // set image
fancyJButton.setRolloverIcon( bug2 ); // set rollover image
add( fancyJButton ); // add fancyJButton to JFrame

// create new ButtonHandler for button event handling
ButtonHandler handler = new ButtonHandler();
fancyJButton.addActionListener( handler );
plainJButton.addActionListener( handler );
} // end ButtonFrame constructor

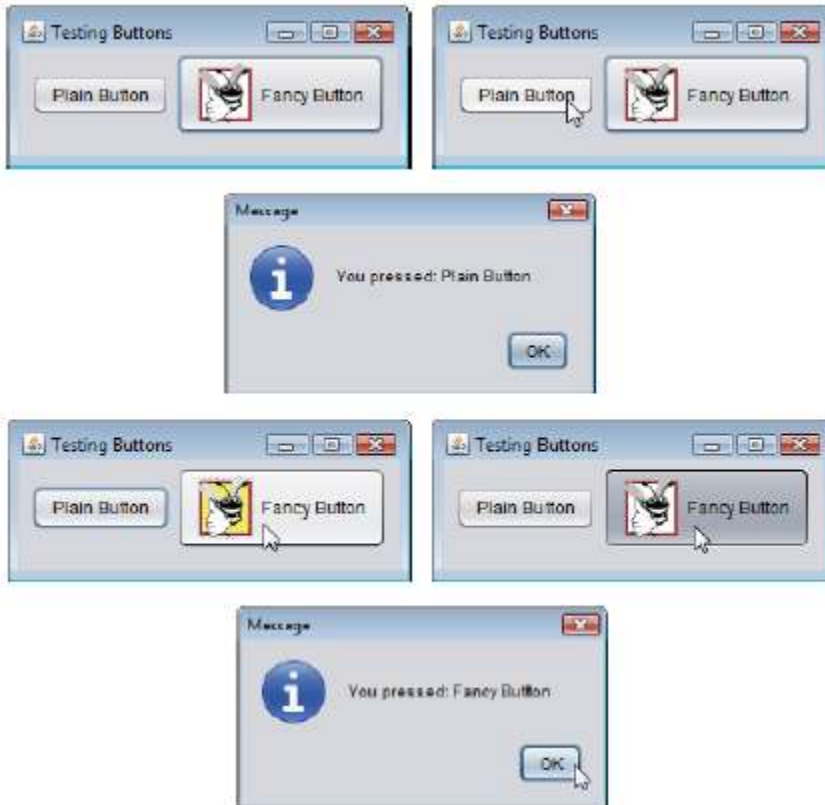
// inner class for button event handling
private class ButtonHandler implements ActionListener
{
    // handle button event
    public void actionPerformed((ActionEvent event) )
    {
        JOptionPane.showMessageDialog(ButtonFrame.this, String.format(
            "You pressed: %s", event.getActionCommand() ));
    } // end method actionPerformed
} // end private inner class ButtonHandler
} // end class ButtonFrame

// ButtonTest.java
// Testing ButtonFrame.
import javax.swing.JFrame;

public class ButtonTest
{
    public static void main( String[] args )
    {
        ButtonFrame buttonFrame = new ButtonFrame(); // create ButtonFrame
        buttonFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        buttonFrame.setSize( 275, 110 ); // set frame size
        buttonFrame.setVisible( true ); // display frame
    } // end main
} // end class ButtonTest

```

output



JCheckBox

Classes **JCheckBox** and **JRadioButton** are subclasses of **JToggleButton**. A **JRadioButton** is different from a **JCheckBox** in that normally several **JRadioButtons** are grouped together and are mutually exclusive—only one in the group can be selected at any time where as more than one check box can be selected at a time.

```
//CheckBoxFrame.java
// Creating JCheckBox buttons.
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JCheckBox;

public class CheckBoxFrame extends JFrame
{
    private JTextField textField; // displays text in changing fonts
    private JCheckBox boldJCheckBox; // to select/deselect bold
    private JCheckBox italicJCheckBox; // to select/deselect italic

    // CheckBoxFrame constructor adds JCheckBoxes to JFrame
    public CheckBoxFrame()
```

```

{
    super( "JCheckBox Test" );
    setLayout( new FlowLayout() ); // set frame layout

    // set up JTextField and set its font
    textField = new JTextField( "Watch the font style change", 20 );
    textField.setFont( new Font( "Serif", Font.PLAIN, 14 ) );
    add( textField ); // add textField to JFrame

    boldJCheckBox = new JCheckBox( "Bold" ); // create bold checkbox
    italicJCheckBox = new JCheckBox( "Italic" ); // create italic
    add( boldJCheckBox ); // add bold checkbox to JFrame
    add( italicJCheckBox ); // add italic checkbox to JFrame

    // register listeners for JCheckBoxes
    CheckBoxHandler handler = new CheckBoxHandler();
    boldJCheckBox.addItemListener( handler );
    italicJCheckBox.addItemListener( handler );

} // end CheckBoxFrame constructor

// private inner class for ItemListener event handling
private class CheckBoxHandler implements ItemListener
{
    // respond to checkbox events
    public void itemStateChanged( ItemEvent event )
    {
        Font font = null; // stores the new Font
        // determine which CheckBoxes are checked and create Font
        if ( boldJCheckBox.isSelected() && italicJCheckBox.isSelected() )
            font = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
        else if ( boldJCheckBox.isSelected() )
            font = new Font( "Serif", Font.BOLD, 14 );
        else if ( italicJCheckBox.isSelected() )
            font = new Font( "Serif", Font.ITALIC, 14 );
        else
            font = new Font( "Serif", Font.PLAIN, 14 );
        textField.setFont( font ); // set textField's font
    } // end method itemStateChanged
} // end private inner class CheckBoxHandler
} // end class CheckBoxFrame

// CheckBoxTest.java
// Testing CheckBoxFrame.
import javax.swing.JFrame;

public class CheckBoxTest
{

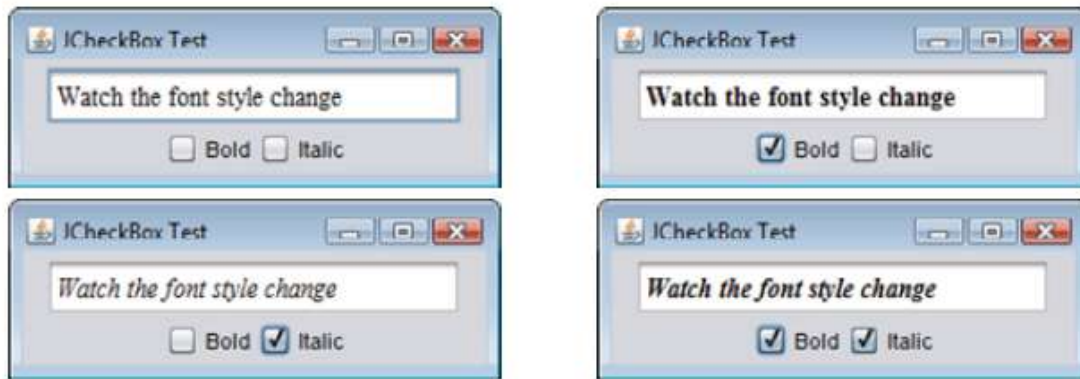
```

```

public static void main( String[] args )
{
    CheckBoxFrame checkBoxFrame = new CheckBoxFrame();
    checkBoxFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    checkBoxFrame.setSize( 275, 100 ); // set frame size
    checkBoxFrame.setVisible( true ); // display frame
} // end main
} // end class CheckBoxTest

```

Output



JRadioButton

Radio buttons (declared with class JRadioButton) are similar to checkboxes in that they have **two states**—**selected and not selected (also called deselected)**. However, **radio buttons** normally appear as a group **in which only one button can be selected at a time**. Selecting a different radio button forces all others to be deselected. **Radio buttons are used to represent mutually exclusive options (i.e., multiple options in the group cannot be selected at the same time).**

// Fig. 14.19: RadioButtonFrame.java
 // Creating radio buttons using ButtonGroup and JRadioButton.

```

import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JRadioButton;
import javax.swing.ButtonGroup;

public class RadioButtonFrame extends JFrame
{
    private JTextField textField; // used to display font changes
    private Font plainFont; // font for plain text

```

```

private Font boldFont; // font for bold text
private Font italicFont; // font for italic text
private Font boldItalicFont; // font for bold and italic text
private JRadioButton plainJRadioButton; // selects plain text
private JRadioButton boldJRadioButton; // selects bold text
private JRadioButton italicJRadioButton; // selects italic text
private JRadioButton boldItalicJRadioButton; // bold and italic
private ButtonGroup radioGroup; // buttongroup to hold radio buttons

```

```

// RadioButtonFrame constructor adds JRadioButtons to JFrame
public RadioButtonFrame()

```

```

{
super( "RadioButton Test" );
setLayout( new FlowLayout() ); // set frame layout
textField = new JTextField( "Watch the font style change", 25 );
add( textField ); // add textField to JFrame

```

```

// create radio buttons

```

```

plainJRadioButton = new JRadioButton( "Plain", true );
boldJRadioButton = new JRadioButton( "Bold", false );
italicJRadioButton = new JRadioButton( "Italic", false );
boldItalicJRadioButton = new JRadioButton( "Bold/Italic", false );
add( plainJRadioButton ); // add plain button to JFrame
add( boldJRadioButton ); // add bold button to JFrame
add( italicJRadioButton ); // add italic button to JFrame
add( boldItalicJRadioButton ); // add bold and italic button

```

```

// create logical relationship between JRadioButtons

```

```

radioGroup = new ButtonGroup(); // create ButtonGroup
radioGroup.add( plainJRadioButton ); // add plain to group
radioGroup.add( boldJRadioButton ); // add bold to group
radioGroup.add( italicJRadioButton ); // add italic to group
radioGroup.add( boldItalicJRadioButton ); // add bold and italic

```

```

// create font objects

```

```

plainFont = new Font( "Serif", Font.PLAIN, 14 );
boldFont = new Font( "Serif", Font.BOLD, 14 );
italicFont = new Font( "Serif", Font.ITALIC, 14 );
boldItalicFont = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
textField.setFont( plainFont ); // set initial font to plain

```

```

// register events for JRadioButtons

```

```

plainJRadioButton.addItemListener(
new RadioButtonHandler( plainFont ) );
boldJRadioButton.addItemListener(
new RadioButtonHandler( boldFont ) );

```

```

italicJRadioButton.addItemListener(
new RadioButtonHandler( italicFont ) );
boldItalicJRadioButton.addItemListener(
new RadioButtonHandler( boldItalicFont ) );
} // end RadioButtonFrame constructor

// private inner class to handle radio button events
private class RadioButtonHandler implements ItemListener
{
private Font font; // font associated with this listener
public RadioButtonHandler( Font f )
{
font = f; // set the font of this listener
} // end constructor RadioButtonHandler

// handle radio button events
public void itemStateChanged( ItemEvent event )
{
textField.setFont( font ); // set font of textField
} // end method itemStateChanged
} // end private inner class RadioButtonHandler
} // end class RadioButtonFrame

//RadioButtonTest.java
// Testing RadioButtonFrame.
import javax.swing.JFrame;

public class RadioButtonTest
{
public static void main( String[] args )
{
RadioButtonFrame radioButtonFrame = new RadioButtonFrame();
radioButtonFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
radioButtonFrame.setSize( 300, 100 ); // set frame size
radioButtonFrame.setVisible( true ); // display frame
} // end main
} // end class RadioButtonTest

```

output





JComboBox (Using an Anonymous Inner Class for Event Handling)

A combo box (sometimes called a **drop-down list**) enables the user to select one item from a list . Combo boxes are implemented with class **JComboBox**, which extends class **JComponent**. JComboBoxes generate **ItemEvents** just as **JCheckBoxes** and **JRadioButtons** do. This example also demonstrates a special form of inner class that's used frequently in event handling.

// Fig. 14.21: JComboBoxFrame.java

// JComboBox that displays a list of image names.

```
import java.awt.FlowLayout;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JComboBox;
import javax.swing.Icon;
import javax.swing.ImageIcon;

public class JComboBoxFrame extends JFrame
{
    private JComboBox imagesJComboBox; // combobox to hold names of icons
    private JLabel label; // label to display selected icon

    private static final String[] names =
    { "bug1.gif", "bug2.gif", "travelbug.gif", "buganim.gif" };
    private Icon[] icons = {
        new ImageIcon( getClass().getResource( names[ 0 ] ) ),
        new ImageIcon( getClass().getResource( names[ 1 ] ) ),
        new ImageIcon( getClass().getResource( names[ 2 ] ) ),
        new ImageIcon( getClass().getResource( names[ 3 ] ) ) };

    // JComboBoxFrame constructor adds JComboBox to JFrame
    public JComboBoxFrame()
    {
        super( "Testing JComboBox" );
        setLayout( new FlowLayout() ); // set frame layout
        imagesJComboBox = new JComboBox( names ); // set up JComboBox
        imagesJComboBox.setMaximumRowCount( 3 ); // display three rows
        imagesJComboBox.addItemListener(
            new ItemListener() // anonymous inner class
            {

```

```

// handle JComboBox event
public void itemStateChanged( ItemEvent event )
{
    // determine whether item selected
    if ( event.getStateChange() == ItemEvent.SELECTED )
        label.setIcon( icons[
            imagesJComboBox.getSelectedIndex() ] );
    } // end method itemStateChanged
} // end anonymous inner class
); // end call to addItemListener
add( imagesJComboBox ); // add combobox to JFrame
label = new JLabel( icons[ 0 ] ); // display first icon
add( label ); // add label to JFrame
} // end ComboBoxFrame constructor
}

```

// Fig. 14.22: ComboBoxTest.java

// Testing ComboBoxFrame.

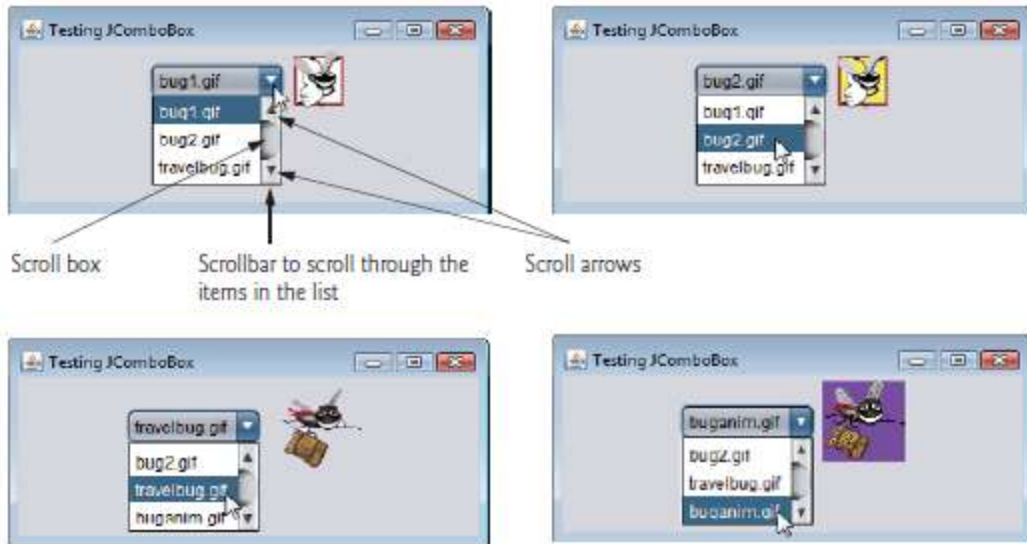
```
import javax.swing.JFrame;
```

```

public class ComboBoxTest
{
    public static void main( String[] args )
    {
        ComboBoxFrame comboBoxFrame = new ComboBoxFrame();
        comboBoxFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        comboBoxFrame.setSize( 350, 150 ); // set frame size
        comboBoxFrame.setVisible( true ); // display frame
    } // end main
} // end class ComboBoxTest

```

Output



JList

A list displays a series of items from which the user may select one or more items . Lists are created with class **JList**, which directly extends class **JComponent**. Class **JList** supports **single selection lists** (which allow only one item to be selected at a time) and **multiple-selection lists**(which allow any number of items to be selected).

single-selection lists

The application below creates a **JList** containing 13 color names. When a color name is clicked in the **JList**, a **ListSelectionEvent** occurs and the application changes the background color of the application window to the selected color.

```
// ListFrame.java
// JList that displays a list of colors.
import java.awt.FlowLayout;
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.event.ListSelectionListener;
import javax.swing.event.ListSelectionEvent;
import javax.swing.ListSelectionModel;

public class ListFrame extends JFrame
{
    private JList colorJList; // list to display colors
    private static final String[] colorNames = { "Black", "Blue", "Cyan",
        "Dark Gray", "Gray", "Green", "Light Gray", "Magenta",
        "Orange", "Pink", "Red", "White", "Yellow" };
    private static final Color[] colors = { Color.BLACK, Color.BLUE,
        Color.CYAN, Color.DARK_GRAY, Color.GRAY, Color.GREEN,
```

```

Color.LIGHT_GRAY, Color.MAGENTA, Color.ORANGE, Color.PINK,
Color.RED, Color.WHITE, Color.YELLOW };
// ListFrame constructor add JScrollPane containing JList to JFrame
public ListFrame()
{
    super( "List Test" );
    setLayout( new FlowLayout() ); // set frame layout
    colorJList = new JList( colorNames ); // create with colorNames
    colorJList.setVisibleRowCount( 5 ); // display five rows at once
    // do not allow multiple selections
    colorJList.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
    // add a JScrollPane containing JList to frame
    add( new JScrollPane( colorJList ) );

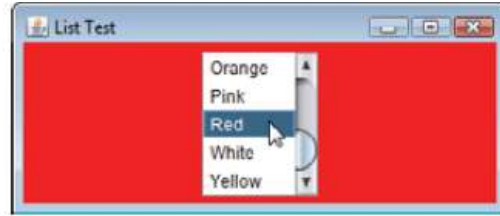
    colorJList.addListSelectionListener(
        new ListSelectionListener() // anonymous inner class
        {
            // handle list selection events
            public void valueChanged( ListSelectionEvent event )
            {
                getContentPane().setBackground(
                    colors[colorJList.getSelectedIndex() ] );
            } // end method valueChanged
        } // end anonymous inner class
    ); // end call to addListSelectionListener
} // end ListFrame constructor
} // end class ListFrame

// ListTest.java
//Selecting colors from a JList.
import javax.swing.JFrame;

public class ListTest
{
    public static void main( String[] args )
    {
        ListFrame listFrame = new ListFrame(); // create ListFrame
        listFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        listFrame.setSize( 350, 150 ); // set frame size
        listFrame.setVisible( true ); // display frame
    } // end main
} // end class ListTest

```

output



Multiple-Selection Lists

A multiple-selection list **enables the user to select many items from a JList.**

```
// MultipleSelectionFrame.java
//Copying items from one List to another.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JButton;
import javax.swing.JScrollPane;
import javax.swing.ListSelectionModel;

public class MultipleSelectionFrame extends JFrame
{
    private JList colorJList; // list to hold color names
    private JList copyJList; // list to copy color names into
    private JButton copyJButton; // button to copy selected names
    private static final String[] colorNames = { "Black", "Blue", "Cyan",
        "Dark Gray", "Gray", "Green", "Light Gray", "Magenta", "Orange",
        "Pink", "Red", "White", "Yellow" };

    // MultipleSelectionFrame constructor
    public MultipleSelectionFrame()
    {
        super( "Multiple Selection Lists" );
        setLayout( new FlowLayout() ); // set frame layout

        colorJList = new JList( colorNames ); // holds names of all colors
        colorJList.setVisibleRowCount( 5 ); // show five rows
        colorJList.setSelectionMode(
            ListSelectionModel.MULTIPLE_INTERVAL_SELECTION );
        add( new JScrollPane( colorJList ) ); // add list with scrollpane

        copyJButton = new JButton( "Copy >>>" ); // create copy button
        copyJButton.addActionListener(

            new ActionListener() // anonymous inner class
            {
```

```

// handle button event
public void actionPerformed((ActionEvent event)
{
// place selected values in copyJList
copyJList.setListData( colorJList.getSelectedValues() );
} // end method actionPerformed
} // end anonymous inner class
); // end call to addActionListener

add( copyJButton ); // add copy button to JFrame

copyJList = new JList(); // create list to hold copied color names
copyJList.setVisibleRowCount( 5 ); // show 5 rows
copyJList.setFixedCellWidth( 100 ); // set width
copyJList.setFixedCellHeight( 15 ); // set height
copyJList.setSelectionMode(
ListSelectionModel.SINGLE_INTERVAL_SELECTION );
add( new JScrollPane( copyJList ) ); // add list with scrollpane
} // end MultipleSelectionFrame constructor
} // end class MultipleSelectionFrame

```

```

//MultipleSelectionTest.java
//Testing MultipleSelectionFrame.
import javax.swing.JFrame;

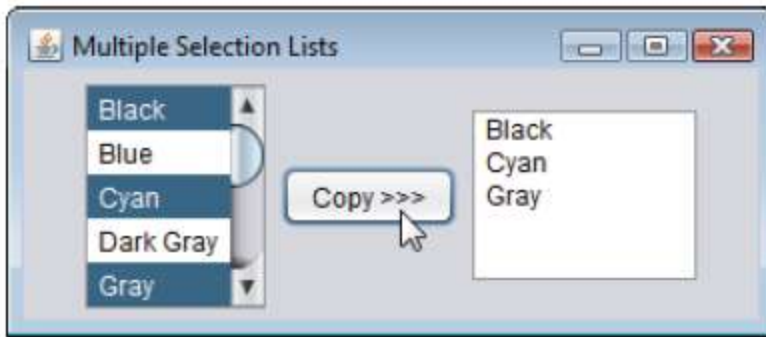
```

```

public class MultipleSelectionTest
{
public static void main( String[] args )
{
MultipleSelectionFrame multipleSelectionFrame =
new MultipleSelectionFrame();
multipleSelectionFrame.setDefaultCloseOperation(
JFrame.EXIT_ON_CLOSE );
multipleSelectionFrame.setSize( 350, 150 ); // set frame size
multipleSelectionFrame.setVisible( true ); // display frame
} // end main
} // end class MultipleSelectionTest

```

output



Mouse Event Handling

MouseListener and MouseMotionListener interface methods

Methods of interface *MouseListener*

`public void mousePressed(MouseEvent event)`

Called when a mouse button is *pressed* while the mouse cursor is on a component.

`public void mouseClicked(MouseEvent event)`

Called when a mouse button is *pressed and released* while the mouse cursor remains stationary on a component. This event is always preceded by a call to `mousePressed`.

`public void mouseReleased(MouseEvent event)`

Called when a mouse button is *released after being pressed*. This event is always preceded by a call to `mousePressed` and one or more calls to `mouseDragged`.

`public void mouseEntered(MouseEvent event)`

Called when the mouse cursor *enters* the bounds of a component.

`public void mouseExited(MouseEvent event)`

Called when the mouse cursor *leaves* the bounds of a component.

Methods of interface *MouseMotionListener*

`public void mouseDragged(MouseEvent event)`

Called when the mouse button is *pressed* while the mouse cursor is on a component and the mouse is *moved* while the mouse button *remains pressed*. This event is always preceded by a call to `mousePressed`. All drag events are sent to the component on which the user began to drag the mouse.

`public void mouseMoved(MouseEvent event)`

Called when the mouse is *moved* (with no mouse buttons pressed) when the mouse cursor is on a component. All move events are sent to the component over which the mouse is currently positioned.

```
//MouseTrackerFrame.java
//Demonstrating mouse events.
import java.awt.Color;
import java.awt.BorderLayout;
```

```

import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class MouseTrackerFrame extends JFrame
{
    private JPanel mousePanel; // panel in which mouse events will occur
    private JLabel statusBar; // label that displays event information

    // MouseTrackerFrame constructor sets up GUI and
    // registers mouse event handlers
    public MouseTrackerFrame()
    {
        super( "Demonstrating Mouse Events" );
        mousePanel = new JPanel(); // create panel
        mousePanel.setBackground( Color.WHITE ); // set background color
        add( mousePanel, BorderLayout.CENTER ); // add panel to JFrame
        statusBar = new JLabel( "Mouse outside JPanel" );
        add( statusBar, BorderLayout.SOUTH ); // add label to JFrame
        // create and register listener for mouse and mouse motion events
        MouseHandler handler = new MouseHandler();
        mousePanel.addMouseListener( handler );
        mousePanel.addMouseMotionListener( handler );
    } // end MouseTrackerFrame constructor
    private class MouseHandler implements MouseListener,
        MouseMotionListener
    {
        // MouseListener event handlers
        // handle event when mouse released immediately after press
        public void mouseClicked( MouseEvent event )
        {
            statusBar.setText( String.format( "Clicked at [%d, %d]",
                event.getX() event.getY() ));
        } // end method mouseClicked

        // handle event when mouse pressed

        public void mousePressed( MouseEvent event )
        {
            statusBar.setText( String.format( "Pressed at [%d, %d]",
                event.getX() event.getY() ));
        } // end method mousePressed

        // handle event when mouse released
        public void mouseReleased( MouseEvent event )
        {

```

```

statusBar.setText( String.format( "Released at [%d, %d]",
event.getX() event.getY() ));
} // end method mouseReleased

// handle event when mouse enters area
public void mouseEntered( MouseEvent event )
statusBar.setText( String.format( "Mouse entered at [%d, %d]",
event.getX() ,event.getY() ));
mousePanel.setBackground( Color.GREEN );
} // end method mouseEntered

// handle event when mouse exits area
public void mouseExited( MouseEvent event )
{
statusBar.setText( "Mouse outside JPanel" );
mousePanel.setBackground( Color.WHITE );
} // end method mouseExited

// MouseMotionListener event handlers
// handle event when user drags mouse with button pressed
public void mouseDragged( MouseEvent event )
{
statusBar.setText( String.format( "Dragged at [%d, %d]",
event.getX() event.getY()));
} // end method mouseDragged

// handle event when user moves mouse
public void mouseMoved( MouseEvent event )
{
statusBar.setText( String.format( "Moved at [%d, %d]",
event.getX() ,event.getY() ));
} // end method mouseMoved
} // end inner class MouseHandler
} // end class MouseTrackerFrame

// MouseTrackerFrame.java
// Testing MouseTrackerFrame.
import javax.swing.JFrame;

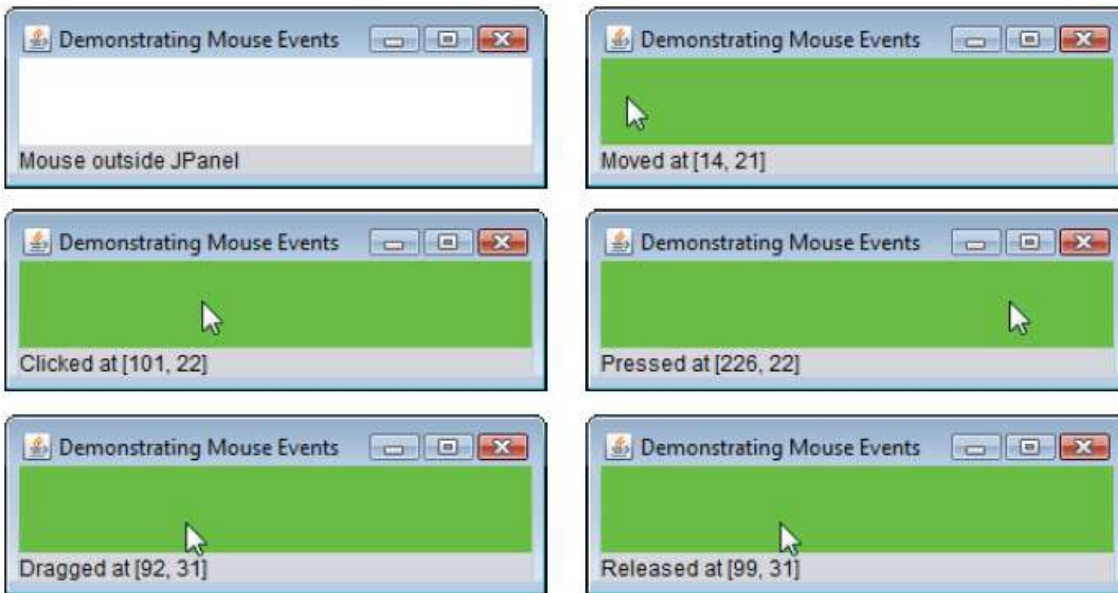
public class MouseTracker
{
public static void main( String[] args )
{
MouseTrackerFrame mouseTrackerFrame = new MouseTrackerFrame();
mouseTrackerFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
mouseTrackerFrame.setSize( 300, 100 ); // set frame size
mouseTrackerFrame.setVisible( true ); // display frame
} // end main

```



```
} // end class MouseTracker
```

output



Adapter Classes

Many event-listener interfaces, such as `MouseListener` and `MouseMotionListener`, contain multiple methods. It's not always desirable to declare every method in an event-listener interface. For instance, an application may need only the `mouseClicked` handler from `MouseListener` or the `mouseDragged` handler from `MouseMotionListener`. Interface `WindowListener` specifies seven window event-handling methods. For many of the listener interfaces that have multiple methods, packages `java.awt.event` and `javax.swing.event` provide **event-listener adapter classes**. An adapter class implements an interface and provides a default implementation (with an empty method body) of each method in the interface. Figure below shows several `java.awt.event` adapter classes and the interfaces they implement. You can extend an adapter class to inherit the default implementation of every method and subsequently override only the method(s) you need for event handling.

Event-adapter class in java.awt.event	Implements interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

fig. Event-adapter classes and the interfaces they implement in package java.awt.event.

```
//MouseDetailsFrame.java
// Demonstrating mouse clicks and distinguishing between mouse buttons.
import java.awt.BorderLayout;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MouseDetailsFrame extends JFrame
{
    private String details; // String that is displayed in the statusBar
    private JLabel statusBar; // JLabel that appears at bottom of window

    // constructor sets title bar String and register mouse listener
    public MouseDetailsFrame()
    {
        super( "Mouse clicks and buttons" );

        statusBar = new JLabel( "Click the mouse" );
        add( statusBar, BorderLayout.SOUTH );
        addMouseListener( new MouseClickHandler() ); // add handler
    } // end MouseDetailsFrame constructor

    // inner class to handle mouse events
    private class MouseClickHandler extends MouseAdapter
    {
        // handle mouse-click event and determine which button was pressed
        public void mouseClicked( MouseEvent event )
        {
            int xPos = event.getX(); // get x-position of mouse
            int yPos = event.getY(); // get y-position of mouse
        }
    }
}
```

```

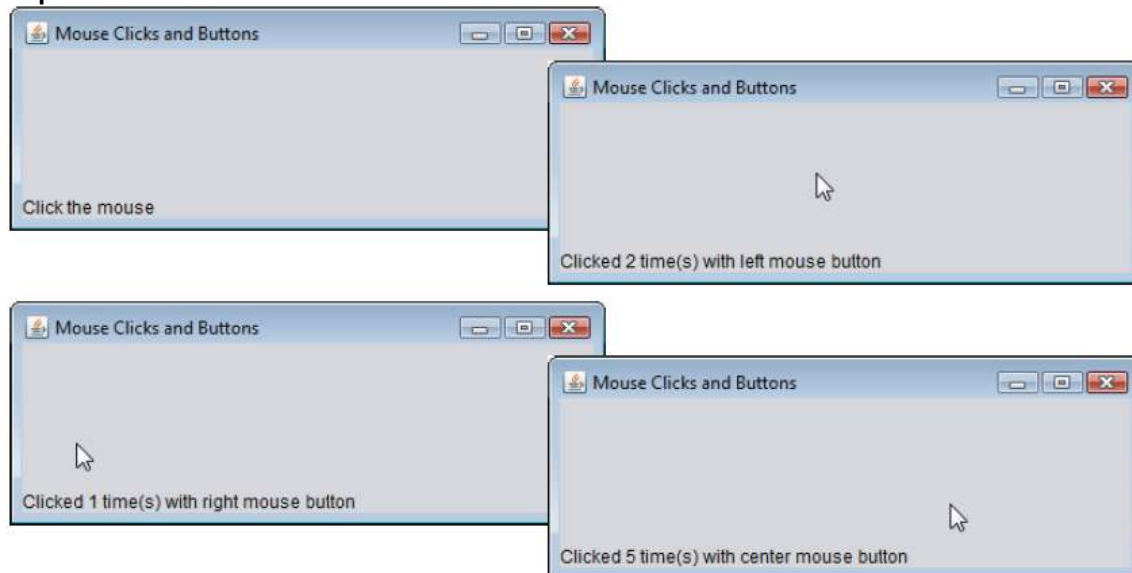
details = String.format( "Clicked %d time(s)",
event.getClickCount() );
if ( event.isMetaDown());
details += " with right mouse button";
else if( event.isAltDown())// middle mouse button
details += " with center mouse button";
else // left mouse button
details += " with left mouse button";

statusBar.setText( details ); // display message in statusBar
} // end method mouseClicked
} // end private inner class MouseClickHandler
} // end class MouseDetailsFrame

//MouseDetails.java
// Testing MouseDetailsFrame.
import javax.swing.JFrame;
public class MouseDetails
{
public static void main( String[] args )
{
MouseDetailsFrame mouseDetailsFrame = new MouseDetailsFrame();
mouseDetailsFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
mouseDetailsFrame.setSize( 400, 150 ); // set frame size
mouseDetailsFrame.setVisible( true ); // display frame
} // end main
} // end class MouseDetails

```

output



InputEvent method	Description
isMetaDown()	Returns true when the user clicks the <i>right mouse button</i> on a mouse with two or three buttons. To simulate a right-mouse-button click on a one-button mouse, the user can hold down the <i>Meta</i> key on the keyboard and click the mouse button.
isAltDown()	Returns true when the user clicks the <i>middle mouse button</i> on a mouse with three buttons. To simulate a middle-mouse-button click on a one- or two-button mouse, the user can press the <i>Alt</i> key and click the only or left mouse button, respectively.

Key Event Handling with JTextArea

Key events are generated when keys on the keyboard are pressed and released. A class that implements KeyListener must provide declarations for methods **keyPressed**, **keyReleased** and **keyTyped**, each of which receives a KeyEvent as its argument. Class KeyEvent is a subclass of InputEvent. **Method keyPressed** is called in response to **pressing any key**. **Method key- Typed** is called in response to pressing any key that is **not an action key**. (The action keys are any arrow key, Home, End, Page Up, Page Down, any function key, etc.) **Method key- Released** is called **when the key is released after any keyPressed or keyTyped event**.

The application below demonstrates the KeyListener methods. Class KeyDemoFrame implements the KeyListener interface, **so all three methods are declared in the application**. The constructor registers the application to handle its own key events by using method addKeyListener . Method addKey- Listener is declared in class Component, so every subclass of Component can notify Key- Listener objects of key events for that Component.

setDisabledTextColor to change the text color in the JTextArea to black for readability.

getKeyCode to get the virtual key code of the pressed key. Class KeyEvent contains virtual key-code constants that represent every key on the keyboard.

getKey-Text the value returned by getKeyCode is passed to static KeyEvent method getKey- Text, which returns a string containing the name of the key that was pressed.

get-KeyChar (which returns a char) to get the Unicode value of the character typed.

isActionKey to determine whether the key in the event was an action key.

getModifiers is called to determine whether any **modifier keys (such as Shift, Alt and Ctrl)** were pressed when the key event occurred.

getKeyModifiersText which produces a string containing the names of the pressed modifier keys.

```
//KeyDemoFrame.java
//Demonstrating keystroke events.
import java.awt.Color;
import java.awt.event.KeyListener;
import java.awt.event.KeyEvent;
import javax.swing.JFrame;
import javax.swing.JTextArea;
```

```

public class KeyDemoFrame extends JFrame implements KeyListener
{
    private String line1 = ""; // first line of textarea
    private String line2 = ""; // second line of textarea
    private String line3 = ""; // third line of textarea
    private JTextArea textArea; // textarea to display output
    // KeyDemoFrame constructor
    public KeyDemoFrame()
    {
        super( "Demonstrating Keystroke Events" );

        textArea = new JTextArea( 10, 15 ); // set up JTextArea
        textArea.setText( "Press any key on the keyboard..." );
        textArea.setEnabled( false ); // disable textarea
        textArea.setDisabledTextColor( Color.BLACK ); // set text color
        add( textArea ); // add textarea to JFrame
        addKeyListener( this ); // allow frame to process key events
    } // end KeyDemoFrame constructor

    // handle press of any key
    public void keyPressed( KeyEvent event )
    {
        line1 = String.format( "Key pressed: %s", KeyEvent.getKeyText( event.getKeyCode() ) ); // show pressed
        key // show pressed
        setLines2and3( event ); // set output lines two and three
    } // end method keyPressed

    // handle release of any key
    public void keyReleased( KeyEvent event )
    {
        line1 = String.format( "Key released: %s", KeyEvent.getKeyText( event.getKeyCode() ) ); // show released
        key // show released
        setLines2and3( event ); // set output lines two and three
    } // end method keyReleased

    // handle press of an action key
    public void keyTyped( KeyEvent event )
    {
        line1 = String.format( "Key typed: %s", event.getKeyChar() );
        setLines2and3( event ); // set output lines two and three
    } // end method keyTyped

    // set second and third lines of output
    private void setLines2and3( KeyEvent event )
    {
        line2 = String.format( "This key is %san action key",
            (event.isActionKey() ? "" : "not " ) );
        String temp = KeyEvent.getKeyModifiersText( event.getModifiers() );
    }
}

```

```

line3 = String.format( "Modifier keys pressed: %s",
( temp.equals( "" ) ? "none" : temp ) ); // output modifiers

textArea.setText( String.format( "%s\n%s\n%s\n",
line1, line2, line3 ) ); // output three lines of text
} // end method setLines2and3
} // end class KeyDemoFrame

//KeyDemo.java
// Testing KeyDemoFrame.
import javax.swing.JFrame;
public class KeyDemo
{
public static void main( String[] args )
{
KeyDemoFrame keyDemoFrame = new KeyDemoFrame();
keyDemoFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
keyDemoFrame.setSize( 350, 100 ); // set frame size
keyDemoFrame.setVisible( true ); // display frame
} // end main
} // end class KeyDemo

```

output



Introduction to Layout Managers

Layout managers arrange GUI components in a container for presentation purposes. You can use the layout managers for basic layout capabilities instead of determining every GUI component's exact

position and size. This functionality enables you to concentrate on the basic look-and-feel and lets the layout managers process most of the layout details. All layout managers implement the interface **LayoutManager** (in package `java.awt`). Class `Container`'s **setLayout** method takes an object that implements the `LayoutManager` interface as an argument. There are basically three ways for you to arrange components in a GUI:

1. **Absolute positioning:** This provides the greatest level of control over a GUI's appearance. By setting a `Container`'s layout to null, you can specify the absolute position of each GUI component with respect to the upper-left corner of the `Container` by using `Component` methods `setSize` and `setLocation` or `setBounds`. If you do this, you also must specify each GUI component's size. Programming a GUI with absolute positioning can be tedious, unless you have an integrated development environment (IDE) that can generate the code for you.
2. **Layout managers:** Using layout managers to position elements can be simpler and faster than creating a GUI with absolute positioning, but you lose some control over the size and the precise positioning of GUI components.
3. **Visual programming in an IDE:** IDEs provide tools that make it easy to create GUIs. Each IDE typically provides a GUI design tool that allows you to drag and drop GUI components from a tool box onto a design area. You can then position, size and align GUI components as you like. The IDE generates the Java code that creates the GUI. In addition, you can typically add event-handling code for a particular component by double-clicking the component.

Layout manager	Description
<code>FlowLayout</code>	Default for <code>javax.swing.JPanel</code> . Places components sequentially (left to right) in the order they were added. It's also possible to specify the order of the components by using the <code>Container</code> method <code>add</code> , which takes a <code>Component</code> and an integer index position as arguments.
<code>BorderLayout</code>	Default for <code>JFrames</code> (and other windows). Arranges the components into five areas: NORTH, SOUTH, EAST, WEST and CENTER.
<code>GridLayout</code>	Arranges the components into rows and columns.

FlowLayout

`FlowLayout` is the simplest layout manager. **GUI components are placed on a container from left to right in the order in which they're added to the container.** When the edge of the container is reached, components continue to display on the next line.

Class `FlowLayout` allows GUI components to be **left aligned, centered (the default) and right aligned.**

The application below creates three `JButton` objects and adds them to the application, using a `FlowLayout` layout manager. The components are center aligned by default. When the user clicks Left, the alignment for the layout manager is changed to a left-aligned `FlowLayout`. When the user clicks Right, the alignment for the layout manager is changed to a right-aligned `FlowLayout`. When the user clicks Center, the alignment for the layout manager is changed to a center-aligned `FlowLayout`.

```
//FlowLayoutFrame.java  
//Demonstrating FlowLayout alignments.
```



```

import java.awt.FlowLayout;
import java.awt.Container;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;

public class FlowLayoutFrame extends JFrame
{
    private JButton leftJButton; // button to set alignment left
    private JButton centerJButton; // button to set alignment center
    private JButton rightJButton; // button to set alignment right
    private FlowLayout layout; // layout object
    private Container container; // container to set layout

    // set up GUI and register button listeners
    public FlowLayoutFrame()
    {
        super( "FlowLayout Demo" );
        layout = new FlowLayout(); // create FlowLayout
        container = getContentPane(); // get container to layout
        setLayout( layout ); // set frame layout
        // set up leftJButton and register listener
        leftJButton = new JButton( "Left" ); // create Left button
        add( leftJButton ); // add Left button to frame
        leftJButton.addActionListener(
            new ActionListener() // anonymous inner class
            {
                // process leftJButton event
                public void actionPerformed( ActionEvent event )
                {
                    layout.setAlignment( FlowLayout.LEFT );
                    // realign attached components
                    layout.layoutContainer( container );
                } // end method actionPerformed
            } // end anonymous inner class
        ); // end call to addActionListener

        // set up centerJButton and register listener
        centerJButton = new JButton( "Center" ); // create Center button
        add( centerJButton ); // add Center button to frame
        centerJButton.addActionListener(
            new ActionListener() // anonymous inner class
            {
                // process centerJButton event
                public void actionPerformed( ActionEvent event )
                {
                    layout.setAlignment( FlowLayout.CENTER );
                }
            }
        );
    }
}

```

```

// realign attached components
layout.layoutContainer( container );
} // end method actionPerformed
} // end anonymous inner class
); // end call to addActionListener

// set up rightJButton and register listener
rightJButton = new JButton( "Right" ); // create Right button
add( rightJButton ); // add Right button to frame
rightJButton.addActionListener(
new ActionListener() // anonymous inner class
{
// process rightJButton event
public void actionPerformed((ActionEvent event)
{
layout.setAlignment( FlowLayout.RIGHT );
// realign attached components
layout.layoutContainer( container );
} // end method actionPerformed
} // end anonymous inner class
); // end call to addActionListener
} // end FlowLayoutFrame constructor
} // end class FlowLayoutFrame

```

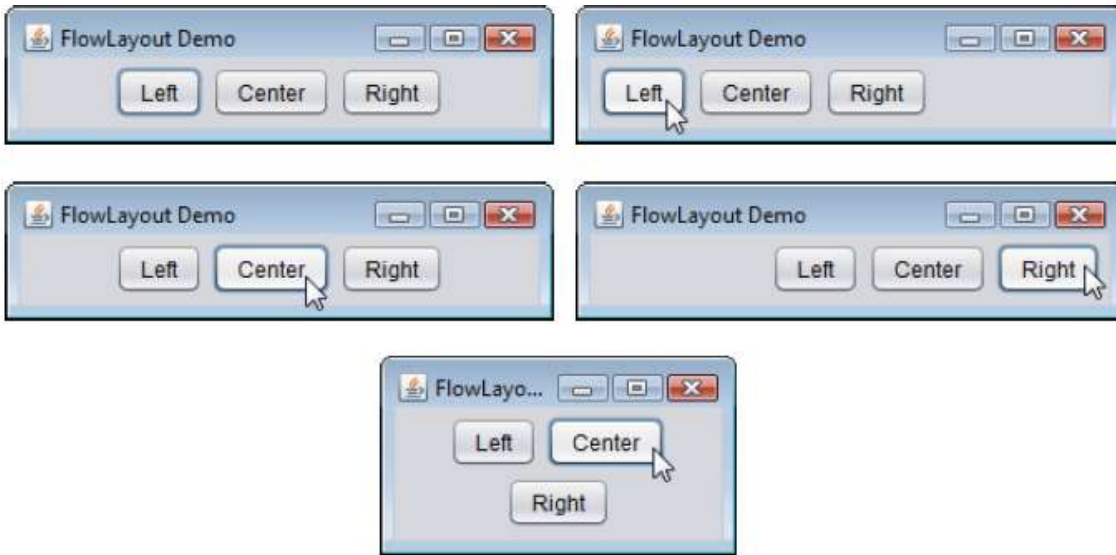
```

// FlowLayoutDemo.java
// Testing FlowLayoutFrame.
import javax.swing.JFrame;

public class FlowLayoutDemo
{
public static void main( String[] args )
{
FlowLayoutFrame flowLayoutFrame = new FlowLayoutFrame();
flowLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
flowLayoutFrame.setSize( 300, 75 ); // set frame size
flowLayoutFrame.setVisible( true ); // display frame
} // end main
} // end class FlowLayoutDemo

```

output



BorderLayout

The **BorderLayout** layout manager (the default layout manager for a **JFrame**) arranges components into **five regions: NORTH, SOUTH, EAST, WEST and CENTER**. **NORTH** corresponds to the **top** of the container.

A BorderLayout limits a Container to containing at most five components—one in each region. The components placed in the **NORTH and SOUTH regions extend horizontally** to the sides of the container and are as tall as the components placed in those regions. The **EAST and WEST regions expand vertically between the NORTH and SOUTH regions** and are as wide as the components placed in those regions. The component placed in the **CENTER region expands to fill all remaining space in the layout**.

If all five regions are occupied, the entire container's space is covered by GUI components. If the NORTH or SOUTH region is not occupied, the GUI components in the EAST, CENTER and WEST regions expand vertically to fill the remaining space. If the EAST or WEST region is not occupied, the GUI component in the CENTER region expands horizontally to fill the remaining space. If the CENTER region is not occupied, the area is left empty—the other GUI components do not expand

```
// BorderLayoutFrame.java
// Demonstrating BorderLayout.
```

```
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;
```

```
public class BorderLayoutFrame extends JFrame implements ActionListener
{
    private JButton[] buttons; // array of buttons to hide portions
    private static final String[] names = { "Hide North", "Hide South",
        "Hide East", "Hide West", "Hide Center" };
    private BorderLayout layout; // borderlayout object
    // set up GUI and event handling
    public BorderLayoutFrame()
```

```

{
super( "BorderLayout Demo" );
layout = new BorderLayout( 5, 5 ); // 5 pixel gaps
setLayout( layout ); // set frame layout
buttons = new JButton[ names.length ]; // set size of array

// create JButtons and register listeners for them
for ( int count = 0; count < names.length; count++ )
{
buttons[ count ] = new JButton( names[ count ] );
buttons[ count ].addActionListener( this );
} // end for
add( buttons[ 0 ], BorderLayout.NORTH ); // add button to north
add( buttons[ 1 ], BorderLayout.SOUTH ); // add button to south
add( buttons[ 2 ], BorderLayout.EAST ); // add button to east
add( buttons[ 3 ], BorderLayout.WEST ); // add button to west
add( buttons[ 4 ], BorderLayout.CENTER ); // add button to center
} // end BorderLayoutFrame constructor

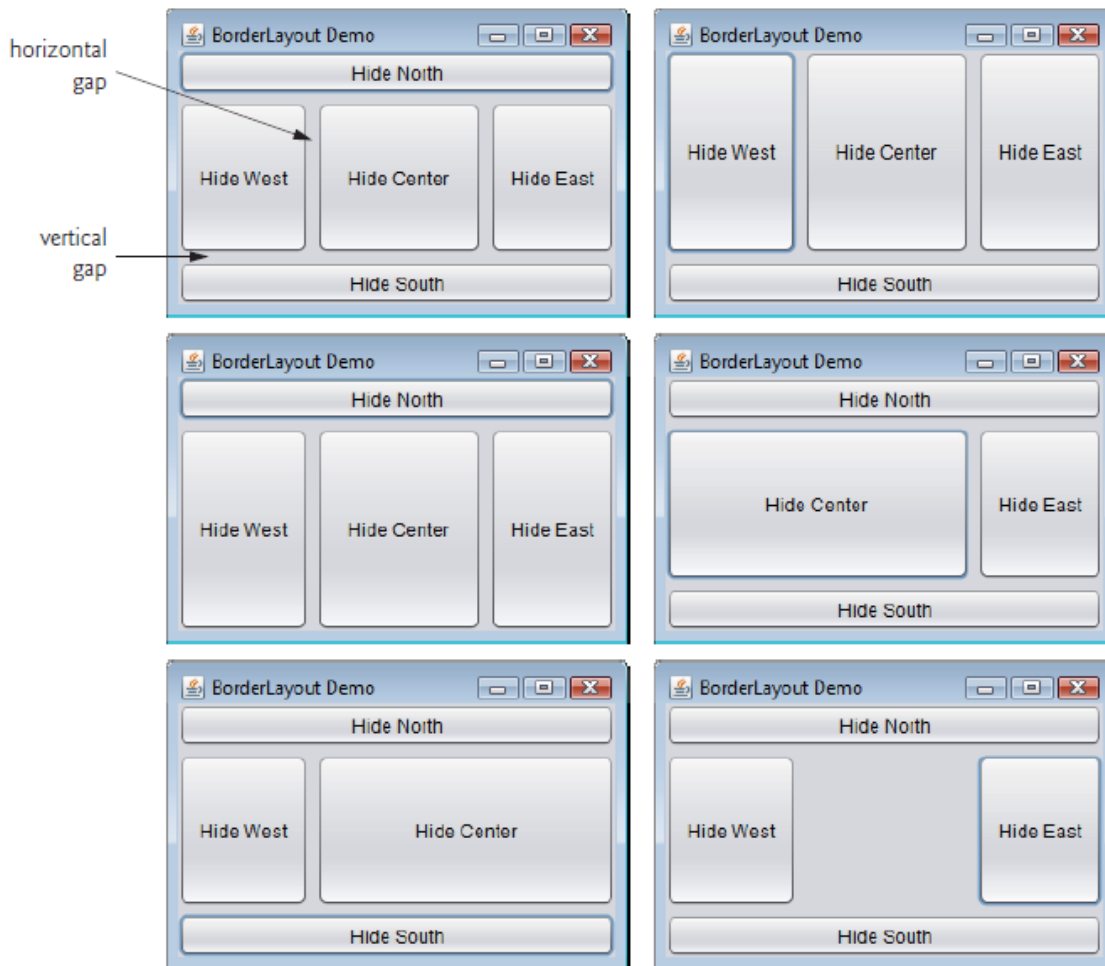
// handle button events
public void actionPerformed((ActionEvent event)
{
// check event source and lay out content pane correspondingly
for ( JButton button : buttons )
{
if ( event.getSource() == button )
button.setVisible( false ); // hide button clicked
else
button.setVisible( true ); // show other buttons
} // end for
layout.layoutContainer( getContentPane() ); // lay out content pane
} // end method actionPerformed
} // end class BorderLayoutFrame

// BorderLayoutDemo.java
// Testing BorderLayoutFrame.
import javax.swing.JFrame;

public class BorderLayoutDemo
{
public static void main( String[] args )
{
BorderLayoutFrame borderLayoutFrame = new BorderLayoutFrame();
borderLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
borderLayoutFrame.setSize( 300, 200 ); // set frame size
borderLayoutFrame.setVisible( true ); // display frame
} // end main
} // end class BorderLayoutDemo

```

output



GridLayout

The **GridLayout** layout manager divides the container into a grid so that components can be placed in **rows and columns**. Class `GridLayout` inherits directly from class `Object` and implements interface `LayoutManager`. **Every Component in a GridLayout has the same width and height.** Components are added to a `GridLayout` starting at the top-left cell of the grid and proceeding left to right until the row is full. Then the process continues left to right on the next row of the grid, and so on.

```
// GridLayoutFrame.java  
// Demonstrating GridLayout.
```

```
import java.awt.GridLayout;  
import java.awt.Container;  
import java.awt.event.ActionListener;  
import java.awt.event.ActionEvent;  
import javax.swing.JFrame;  
import javax.swing.JButton;
```

```
public class GridLayoutFrame extends JFrame implements ActionListener  
{
```

```

private JButton[] buttons; // array of buttons
private static final String[] names =
{ "one", "two", "three", "four", "five", "six" };
private boolean toggle = true; // toggle between two layouts
private Container container; // frame container
private GridLayout gridLayout1; // first gridlayout
private GridLayout gridLayout2; // second gridlayout
// no-argument constructor
public GridLayoutFrame()
{
    super( "GridLayout Demo" );
    gridLayout1 = new GridLayout( 2, 3, 5, 5 ); // 2 by 3; gaps of 5
    gridLayout2 = new GridLayout( 3, 2 ); // 3 by 2; no gap
    container = getContentPane(); // get content pane
    setLayout( gridLayout1 ); // set JFrame layout
    buttons = new JButton[ names.length ]; // create array of JButtons

    for ( int count = 0; count < names.length; count++ )
    {

        buttons[ count ] = new JButton( names[ count ] );
        buttons[ count ].addActionListener( this ); // register listener
        add( buttons[ count ] ); // add button to JFrame
    } // end for
} // end GridLayoutFrame constructor

// handle button events by toggling between layouts
public void actionPerformed((ActionEvent event) )
{
    if ( toggle )
        container.setLayout( gridLayout2 ); // set layout to second
    else
        container.setLayout( gridLayout1 ); // set layout to first
    toggle = !toggle; // set toggle to opposite value
    container.validate(); // re-lay out container
} // end method actionPerformed
} // end class GridLayoutFrame

// GridLayoutDemo.java
// Testing GridLayoutFrame.
import javax.swing.JFrame;

public class GridLayoutDemo
{
    public static void main( String[] args )
    {
        GridLayoutFrame gridLayoutFrame = new GridLayoutFrame();

```

```

gridLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
gridLayoutFrame.setSize( 300, 200 ); // set frame size
gridLayoutFrame.setVisible( true ); // display frame
} // end main
} // end class GridLayoutDemo

```

output



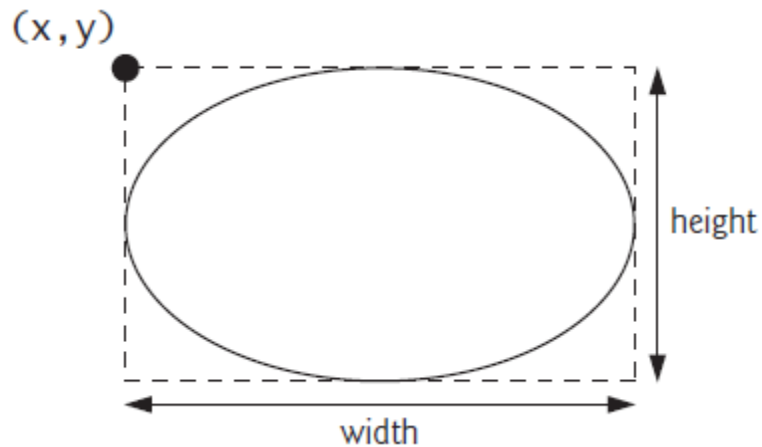
JSlider

JSliders enable a user to select from a range of **integer values**. Class JSlider inherits from JComponent. Figure below shows a horizontal JSlider with **tick marks** and the **thumb** that allows a user to select a value.



if a JSlider has the focus (i.e., it's the currently selected GUI component in the user interface), the **left arrow key** and **right arrow key** cause the thumb of the JSlider to decrease or increase by 1, respectively. The **down arrow key** and **up arrow key** also cause the thumb to decrease or increase by 1 tick, respectively. The **PgDn** (page down) key and **PgUp** (page up) key cause the thumb to decrease or increase by block increments of one-tenth of the range of values, respectively. The **Home** key moves the thumb to the minimum value of the JSlider, and the **End** key moves the thumb to the maximum value of the JSlider.

- **paintComponent** method that can be used to draw graphics. Method paintComponent takes a Graphics object as an argument. This object is passed to the paintComponent method by the system when a lightweight Swing component needs to be repainted.
- **repaint** method can be used if you want to update the graphics drawn on a Swing component.
- **fillOval(int x, int y, int width, int height)** draws a filled oval in the current color with the specified width and height. The bounding rectangle's top-left corner is located at (x, y). The oval touches the center of all four sides of the bounding rectangle.



- The **JSlider constructor** takes four arguments. The first argument specifies the orientation of Slider, which is HORIZONTAL or VERTICAL (a constant in interface SwingConstants). The second and third arguments indicate the minimum and maximum integer values in the range of values for this JSlider. The last argument indicates that the initial value of the JSlider (i.e., where the thumb is displayed).
- Method **setMajorTickSpacing** indicates that each major tick mark represents 10 values in the range of values.
- Method **setPaintTicks** with a true argument indicates that the tick marks should be displayed (they aren't displayed by default).

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Dimension;
import javax.swing.JPanel;

public class OvalPanel extends JPanel
{
    private int diameter = 10; // default diameter of 10

    // draw an oval of the specified diameter
    public void paintComponent( Graphics g )
    {
        super.paintComponent( g );
        g.setColor(Color.red);
        g.fillOval( 10, 10, diameter, diameter ); // draw circle
    } // end method paintComponent

    // validate and set diameter, then repaint
    public void setDiameter( int newDiameter )
    {
        // if diameter invalid, default to 10
        diameter = ( newDiameter >= 0 ? newDiameter : 10 );
        repaint(); // repaint panel
    }
}
```

```

} // end method setDiameter

// used by layout manager to determine preferred size
public Dimension getPreferredSize()
{
    return new Dimension( 200, 200 );
} // end method getPreferredSize

// used by layout manager to determine minimum size
public Dimension getMinimumSize()
{
    return getPreferredSize();
} // end method getMinimumSize
} // end class OvalPanel

import java.awt.BorderLayout;
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JSlider;
import javax.swing.SwingConstants;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;

public class SliderFrame extends JFrame
{
    private JSlider diameterJSlider; // slider to select diameter
    private OvalPanel myPanel; // panel to draw circle

    // no-argument constructor
    public SliderFrame()
    {
        super( "Slider Demo" );

        myPanel = new OvalPanel(); // create panel to draw circle
        myPanel.setBackground( Color.YELLOW ); // set background to yellow

        // set up JSlider to control diameter value
        diameterJSlider =
            new JSlider( SwingConstants.HORIZONTAL, 0, 200, 10 );
        diameterJSlider.setMajorTickSpacing( 10 ); // create tick every 10
        diameterJSlider.setPaintTicks( true ); // paint ticks on slider

        // register JSlider event listener
        diameterJSlider.addChangeListener(

            new ChangeListener() // anonymous inner class
            {
                // handle change in slider value

```



```

        public void stateChanged(ChangeEvent e)
        {
            myPanel.setDiameter( diameterJSlider.getValue() );
        } // end method stateChanged
    } // end anonymous inner class
}; // end call to addChangeListener

add( diameterJSlider, BorderLayout.SOUTH ); // add slider to frame
add( myPanel, BorderLayout.CENTER ); // add panel to frame
} // end SliderFrame constructor
} // end class SliderFrame

```

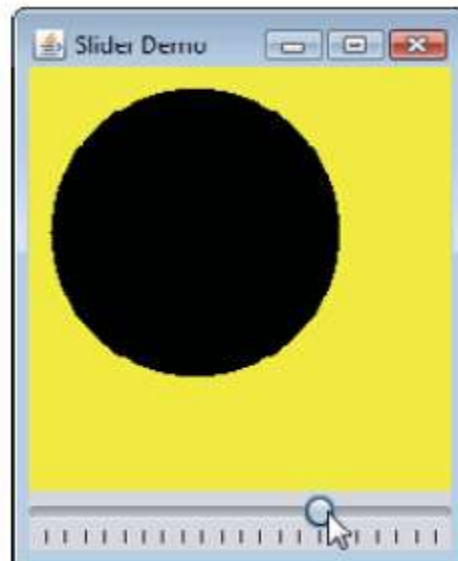
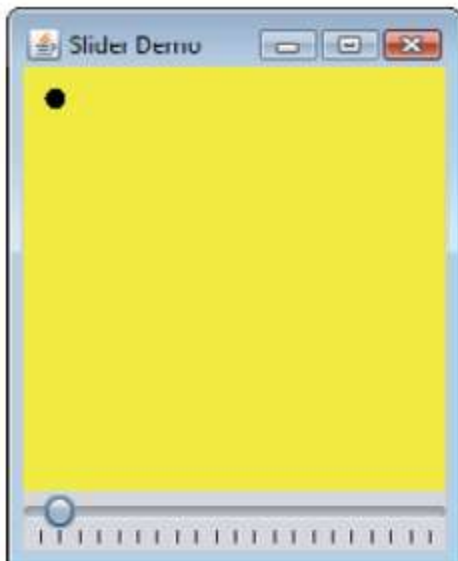
```
import javax.swing.JFrame;
```

```

public class SliderDemo
{
    public static void main( String[] args )
    {
        SliderFrame sliderFrame = new SliderFrame();
        sliderFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        sliderFrame.setSize( 220, 270 ); // set frame size
        sliderFrame.setVisible( true ); // display frame
    } // end main
} // end class SliderDemo

```

Output



JColorChooser

Package javax.swing provides the JColorChooser GUI component that enables application users to select colors. The application below demonstrates a JColorChooser dialog. When you click the Change Color

button, a JColorChooser dialog appears. When you select a color and press the dialog's OK button, the background color of the application window changes.

```
//ShowColors2JFrame.java
//Choosing colors with JColorChooser.
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JColorChooser;
import javax.swing.JPanel;

public class ShowColors2JFrame extends JFrame
{
    private JButton changeColorJButton;
    private Color color = Color.LIGHT_GRAY;
    private JPanel colorJPanel;

    // set up GUI
    public ShowColors2JFrame()
    {
        super( "Using JColorChooser" );

        // create JPanel for display color
        colorJPanel = new JPanel();
        colorJPanel.setBackground( color );

        // set up changeColorJButton and register its event handler
        changeColorJButton = new JButton( "Change Color" );
        changeColorJButton.addActionListener(
            new ActionListener() // anonymous inner class
            {
                // display JColorChooser when user clicks button
                public void actionPerformed( ActionEvent event )
                {
                    color = JColorChooser.showDialog(
                        ShowColors2JFrame.this, "Choose a color", color );
                    // set default color, if no color is returned
                    if ( color == null )
                        color = Color.LIGHT_GRAY;
                    // change content pane's background color
                    colorJPanel.setBackground( color );
                } // end method actionPerformed
            } // end anonymous inner class
        ); // end call to addActionListener
    }
}
```

```
add( colorJPanel, BorderLayout.CENTER ); // add colorJPanel
add( changeColorJButton, BorderLayout.SOUTH ); // add button
```

```
setSize( 400, 130 ); // set frame size
setVisible( true ); // display frame
} // end ShowColor2JFrame constructor
} // end class ShowColors2JFrame
```

//ShowColors2.java

// Choosing colors with JColorChooser.

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

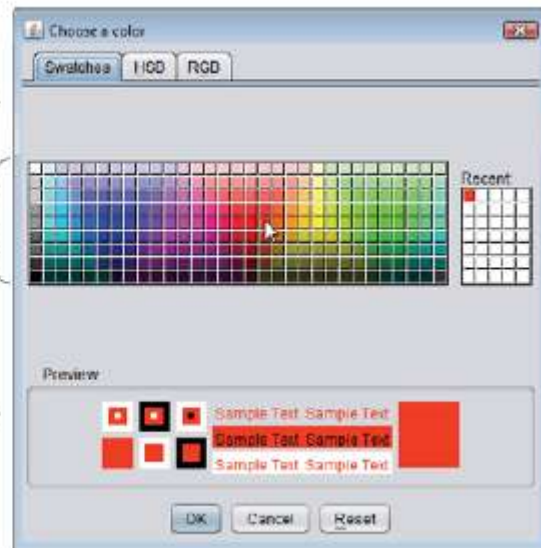
```
public class ShowColors2
{
    // execute application
    public static void main( String[] args )
    {
        ShowColors2JFrame application = new ShowColors2JFrame();
        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    } // end main
} // end class ShowColors2
```

output

(a) Initial application window



(b) JColorChooser window



Select a color from
one of the color
swatches

(c) Application window after changing JPanel's
background color



Using Menus with Frames

Menus are an integral part of GUIs. They allow the user to perform actions without unnecessarily cluttering a GUI with extra components. In Swing GUIs, menus can be attached only to objects of the classes that provide method `setJMenuBar`. Two such classes are **JFrame** and **JApplet**. The classes used to declare menus are **JMenuBar**, **JMenu**, **JMenuItem**, **JCheckBoxMenuItem** and class **JRadioButtonMenuItem**.

Class JMenuBar (a subclass of JComponent) contains the methods necessary to manage a menu bar, which is a container for menus.

Class JMenu (a subclass of javax.swing.JMenuItem) contains the methods necessary for managing menus. Menus contain **menu items** and are added to **menu bars** or to other menus as submenus. When a menu is clicked, it expands to show its list of menu items.

Class JMenuItem (a subclass of javax.swing.AbstractButton) contains the methods necessary to manage menu items. A **menu item** is a GUI component **inside a menu** that, when selected, causes an **action event**. A menu item can be used to initiate an action, or it can be a submenu that provides more menu items from which the user can select. Submenus are useful for grouping related menu items in a menu.

Class JCheckBoxMenuItem (a subclass of javax.swing.JMenuItem) contains the methods necessary to manage **menu items** that can be toggled on or off. When a JCheck-BoxMenuItem is selected, a check appears to the left of the menu item. When the JCheck-BoxMenuItem is selected again, the check is removed.

Class JRadioButtonMenuItem (a subclass of javax.swing.JMenuItem) contains the methods necessary to manage menu items that can be toggled on or off like JCheckBox-Menus. When multiple JRadioButtonMenuItems are maintained as part of a Button-Group, only one item in the group can be selected at a given time. When a JRadioButtonMenuItem is selected, a filled circle appears to the left of the menu item. When another JRadioButtonMenuItem is selected, the filled circle of the previously selected menu item is removed.

```
// MenuFrame.java
// Demonstrating menus.
import java.awt.Color;
import java.awt.Font;
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JLabel;
import javax.swing.SwingConstants;
import javax.swing.ButtonGroup;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import javax.swing.JMenuBar;

public class MenuFrame extends JFrame
{
    private final Color[] colorValues =
        { Color.BLACK, Color.BLUE, Color.RED, Color.GREEN };
    private JRadioButtonMenuItem[] colorItems; // color menu items
    private JRadioButtonMenuItem[] fonts;      // font menu items
    private JCheckBoxMenuItem[] styleItems;     // font style menu items
```

```

private JLabel displayJLabel;           // displays sample text
private ButtonGroup fontButtonGroup;    // manages font menu items
private ButtonGroup colorButtonGroup;   // manages color menu items
private int style;                      // used to create style for font

// no-argument constructor set up GUI
public MenuFrame()
{
    super( "Using JMenus" );

    JMenu fileMenu = new JMenu( "File" );    // create file menu
    fileMenu.setMnemonic( 'F' );             // set mnemonic to F

    // create About... menu item
    JMenuItem aboutItem = new JMenuItem( "About..." );
    aboutItem.setMnemonic( 'A' );           // set mnemonic to A
    fileMenu.add( aboutItem );              // add about item to file menu
    aboutItem.addActionListener(
        new ActionListener() // anonymous inner class
        {
            // display message dialog when user selects About...
            public void actionPerformed((ActionEvent event) )
            {
                JOptionPane.showMessageDialog( MenuFrame.this,
                    "This is an example\nof using menus",
                    "About", JOptionPane.PLAIN_MESSAGE );
            } // end method actionPerformed
        } // end anonymous inner class
    ); // end call to addActionListener

    JMenuItem exitItem = new JMenuItem( "Exit" ); // create exit item
    exitItem.setMnemonic( 'x' );                 // set mnemonic to x
    fileMenu.add( exitItem );                    // add exit item to file menu
    exitItem.addActionListener(

        new ActionListener() // anonymous inner class
        {
            // terminate application when user clicks exitItem
            public void actionPerformed((ActionEvent event) )
            {
                System.exit( 0 ); // exit application
            } // end method actionPerformed
        } // end anonymous inner class
    ); // end call to addActionListener

    JMenuBar bar = new JMenuBar(); // create menu bar
    setJMenuBar( bar ); // add menu bar to application
    bar.add( fileMenu ); // add file menu to menu bar

```

```

JMenu formatMenu = new JMenu( "Format" ); // create format menu
formatMenu.setMnemonic( 'r' ); // set mnemonic to r

// array listing string colors
String[] colors = { "Black", "Blue", "Red", "Green" };

JMenu colorMenu = new JMenu( "Color" ); // create color menu
colorMenu.setMnemonic( 'C' ); // set mnemonic to C

// create radio button menu items for colors
colorItems = new JRadioButtonMenuItem[ colors.length ];
colorButtonGroup = new ButtonGroup(); // manages colors
ItemHandler itemHandler = new ItemHandler(); // handler for colors

// create color radio button menu items
for ( int count = 0; count < colors.length; count++ )
{
    colorItems[ count ] =
        new JRadioButtonMenuItem( colors[ count ] ); // create item
    colorMenu.add( colorItems[ count ] ); // add item to color menu
    colorButtonGroup.add( colorItems[ count ] ); // add to group
    colorItems[ count ].addActionListener( itemHandler );
} // end for

colorItems[ 0 ].setSelected( true ); // select first Color item

formatMenu.add( colorMenu ); // add color menu to format menu
formatMenu.addSeparator(); // add separator in menu

// array listing font names
String[] fontNames = { "Serif", "Monospaced", "SansSerif" };
JMenu fontMenu = new JMenu( "Font" ); // create font menu
fontMenu.setMnemonic( 'n' ); // set mnemonic to n

// create radio button menu items for font names
fonts = new JRadioButtonMenuItem[ fontNames.length ];
fontButtonGroup = new ButtonGroup(); // manages font names

// create Font radio button menu items
for ( int count = 0; count < fonts.length; count++ )
{
    fonts[ count ] = new JRadioButtonMenuItem( fontNames[ count ] );
    fontMenu.add( fonts[ count ] ); // add font to font menu
    fontButtonGroup.add( fonts[ count ] ); // add to button group
    fonts[ count ].addActionListener( itemHandler ); // add handler
} // end for

```

```

fonts[ 0 ].setSelected( true ); // select first Font menu item
fontMenu.addSeparator(); // add separator bar to font menu

String[] styleNames = { "Bold", "Italic" }; // names of styles
styleItems = new JCheckBoxMenuItem[ styleNames.length ];
StyleHandler styleHandler = new StyleHandler(); // style handler

// create style checkbox menu items
for ( int count = 0; count < styleNames.length; count++ )
{
    styleItems[ count ] =
        new JCheckBoxMenuItem( styleNames[ count ] ); // for style
    fontMenu.add( styleItems[ count ] ); // add to font menu
    styleItems[ count ].addItemListener( styleHandler ); // handler
} // end for

formatMenu.add( fontMenu ); // add Font menu to Format menu
bar.add( formatMenu ); // add Format menu to menu bar

// set up label to display text
displayJLabel = new JLabel( "Sample Text", SwingConstants.CENTER );
displayJLabel.setForeground( colorValues[ 0 ] );
displayJLabel.setFont( new Font( "Serif", Font.PLAIN, 72 ) );

getContentPane().setBackground( Color.CYAN ); // set background
add( displayJLabel, BorderLayout.CENTER ); // add displayJLabel
} // end MenuFrame constructor

// inner class to handle action events from menu items
private class ItemHandler implements ActionListener
{
    // process color and font selections
    public void actionPerformed((ActionEvent event) )
    {
        // process color selection
        for ( int count = 0; count < colorItems.length; count++ )
        {
            if ( colorItems[ count ].isSelected() )
            {
                displayJLabel.setForeground( colorValues[ count ] );
                break;
            } // end if
        } // end for

        // process font selection
        for ( int count = 0; count < fonts.length; count++ )
        {
            if ( event.getSource() == fonts[ count ] )

```



```

        {
            displayJLabel.setFont(
                new Font( fonts[ count ].getText(), style, 72 ) );
        } // end if
    } // end for

    repaint(); // redraw application
} // end method actionPerformed
} // end class ItemHandler

// inner class to handle item events from checkbox menu items
private class StyleHandler implements ItemListener
{
    // process font style selections
    public void itemStateChanged( ItemEvent e )
    {
        String name = displayJLabel.getFont().getName(); // current Font
        Font font; // new font based on user selections

        // determine which CheckBoxes are checked and create Font
        if ( styleItems[ 0 ].isSelected() &&
            styleItems[ 1 ].isSelected() )
            font = new Font( name, Font.BOLD + Font.ITALIC, 72 );
        else if ( styleItems[ 0 ].isSelected() )
            font = new Font( name, Font.BOLD, 72 );
        else if ( styleItems[ 1 ].isSelected() )
            font = new Font( name, Font.ITALIC, 72 );
        else
            font = new Font( name, Font.PLAIN, 72 );

        displayJLabel.setFont( font );
        repaint(); // redraw application
    } // end method itemStateChanged
} // end class StyleHandler
} // end class MenuFrame

//MenuTest.java
//Testing MenuFrame.
import javax.swing.JFrame;

public class PopupTest
{
    public static void main( String[] args )
    {
        PopupFrame popupFrame = new PopupFrame(); // create PopupFrame
        popupFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        popupFrame.setSize( 300, 200 ); // set frame size
        popupFrame.setVisible( true ); // display frame
    }
}

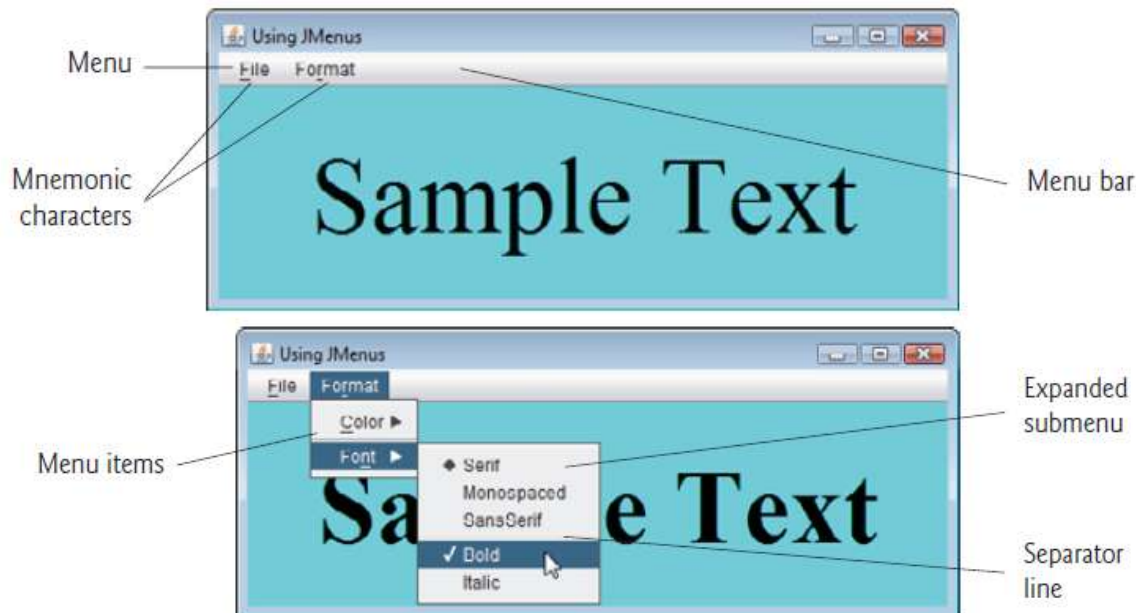
```

```

    } // end main
} // end class PopupTest

```

output



JPopupMenu

Many of today's computer applications provide so-called context-sensitive pop-up menus. In Swing, such menus are created with class **JPopupMenu** (a subclass of JComponent). These menus provide options that are specific to the component for which the popup trigger event was generated. On most systems, **the pop-up trigger event occurs when the user presses and releases the right mouse button.**

```

//PopupFrame.java
//Demonstrating JPopupMenu.
import java.awt.Color;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.JPopupMenu;
import javax.swing.ButtonGroup;

public class PopupFrame extends JFrame
{
    private JRadioButtonMenuItem[] items; // holds items for colors
    private final Color[] colorValues =
    { Color.BLUE, Color.YELLOW, Color.RED }; // colors to be used
    private JPopupMenu popupMenu; // allows user to select color
    // no-argument constructor sets up GUI

```

```

public PopupFrame()
{
    super( "Using JPopupMenu" );

    ItemHandler handler = new ItemHandler(); // handler for menu items
    String[] colors = { "Blue", "Yellow", "Red" }; // array of colors

    ButtonGroup colorGroup = new ButtonGroup(); // manages color items
    popupMenu = new JPopupMenu(); // create pop-up menu
    items = new JRadioButtonMenuItem[ colors.length ]; // color items

    // construct menu item, add to pop-up menu, enable event handling
    for ( int count = 0; count < items.length; count++ )
    {
        items[ count ] = new JRadioButtonMenuItem( colors[ count ] );
        popupMenu.add( items[ count ] ); // add item to pop-up menu
        colorGroup.add( items[ count ] ); // add item to button group
        items[ count ].addActionListener( handler ); // add handler
    } // end for

    setBackground( Color.WHITE ); // set background to white

    // declare a MouseListener for the window to display pop-up menu
    addMouseListener(

        new MouseAdapter() // anonymous inner class
        {
            // handle mouse press event
            public void mousePressed( MouseEvent event )
            {
                checkForTriggerEvent( event ); // check for trigger
            } // end method mousePressed

            // handle mouse release event
            public void mouseReleased( MouseEvent event )
            {
                checkForTriggerEvent( event ); // check for trigger
            } // end method mouseReleased

            // determine whether event should trigger pop-up menu
            private void checkForTriggerEvent( MouseEvent event )
            {
                if ( event.isPopupTrigger() )
                    popupMenu.show(
                        event.getComponent(), event.getX(), event.getY() );
            } // end method checkForTriggerEvent
        } // end anonymous inner class
    ); // end call to addMouseListener

```

```

} // end PopupFrame constructor

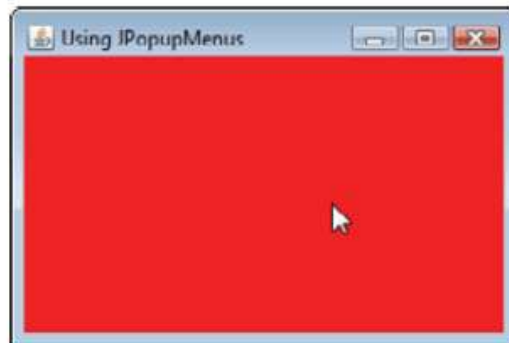
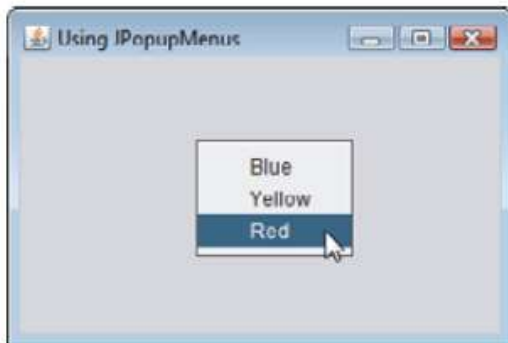
// private inner class to handle menu item events
private class ItemHandler implements ActionListener
{
    // process menu item selections
    public void actionPerformed((ActionEvent event) )
    {
        // determine which menu item was selected
        for ( int i = 0; i < items.length; i++ )
        {
            if ( event.getSource() == items[ i ] )
            {
                getContentPane().setBackground( colorValues[ i ] );
                return;
            } // end if
        } // end for
    } // end method actionPerformed
} // end private inner class ItemHandler
} // end class PopupFrame

//PopupTest.java
// Testing PopupFrame.
import javax.swing.JFrame;

public class PopupTest
{
    public static void main( String[] args )
    {
        PopupFrame popupFrame = new PopupFrame(); // create PopupFrame
        popupFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        popupFrame.setSize( 300, 200 ); // set frame size
        popupFrame.setVisible( true ); // display frame
    } // end main
} // end class PopupTest

```

output



JTabbedPane

A **JTabbedPane** arranges GUI components into layers, of which only one is visible at a time. Users access each layer via a tab—similar to folders in a file cabinet. When the user clicks a tab, the appropriate layer is displayed. The tabs appear at the top by default but also can be positioned at the left, right or bottom of the JTabbedPane. Any component can be placed on a tab. If the component is a container, such as a panel, it can use any layout manager to lay out several components on the tab. Class JTabbedPane is a subclass of JComponent. The application below creates one tabbed pane with three tabs. Each tab displays one of the JPanels—panel1, panel2 or panel3.

```
// JTabbedPaneFrame.java
// Demonstrating JTabbedPane
import java.awt.BorderLayout;
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JTabbedPane;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JButton;
import javax.swing.SwingConstants;

public class JTabbedPaneFrame extends JFrame
{
    // set up GUI
    public JTabbedPaneFrame()
    {
        super( "JTabbedPane Demo " );

        JTabbedPane tabbedPane = new JTabbedPane(); // create JTabbedPane

        // set up pane11 and add it to JTabbedPane
        JLabel label1 = new JLabel( "panel one", SwingConstants.CENTER );
        JPanel panel1 = new JPanel(); // create first panel
        panel1.add( label1 ); // add label to panel
        tabbedPane.addTab( "Tab One", null, panel1, "First Panel" );

        // set up panel2 and add it to JTabbedPane
        JLabel label2 = new JLabel( "panel two", SwingConstants.CENTER );
        JPanel panel2 = new JPanel(); // create second panel
        panel2.setBackground( Color.YELLOW ); // set background to yellow
        panel2.add( label2 ); // add label to panel
        tabbedPane.addTab( "Tab Two", null, panel2, "Second Panel" );

        // set up panel3 and add it to JTabbedPane
        JLabel label3 = new JLabel( "panel three" );
        JPanel panel3 = new JPanel(); // create third panel
        panel3.setLayout( new BorderLayout() ); // use borderlayout
        panel3.add( new JButton( "North" ), BorderLayout.NORTH );
```

```

panel3.add( new JButton( "West" ), BorderLayout.WEST );
panel3.add( new JButton( "East" ), BorderLayout.EAST );
panel3.add( new JButton( "South" ), BorderLayout.SOUTH );
panel3.add( label3, BorderLayout.CENTER );
tabbedPane.addTab( "Tab Three", null, panel3, "Third Panel" );

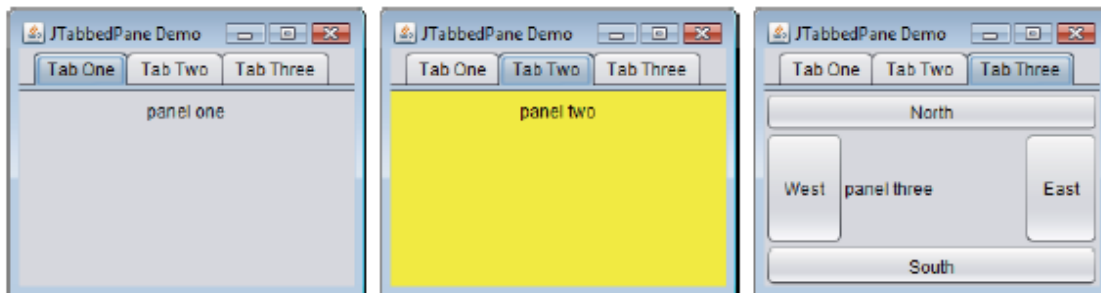
add( tabbedPane ); // add JTabbedPane to frame
} // end JTabbedPaneFrame constructor
} // end class JTabbedPaneFrame

// JTabbedPaneDemo.java
// Demonstrating JTabbedPane.
import javax.swing.JFrame;

public class JTabbedPaneDemo
{
    public static void main( String[] args )
    {
        JTabbedPaneFrame tabbedPaneFrame = new JTabbedPaneFrame();
        tabbedPaneFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        tabbedPaneFrame.setSize( 250, 200 ); // set frame size
        tabbedPaneFrame.setVisible( true ); // display frame
    } // end main
} // end class JTabbedPaneDemo

```

output



Tables

A table is a component that displays rows and columns of data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position. Tables are implemented by the **JTable class**, which extends JComponent.

One of its **constructors** is shown here:

JTable(Object data[][], Object colHeads[])

Here, **data** is a **two-dimensional array of the information to be presented**, and **colHeads** is a **one-dimensional array with the column headings**.

Here are the steps for using a table in a Frame:

1. Create a JTable object.

- 2. Create a JScrollPane object. (The arguments to the constructor specify the table and the policies for vertical and horizontal scroll bars.)**
- 3. Add the table to the scroll pane.**
- 4. Add the scroll pane to JFrame**

//TableFrame.java

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import java.awt.BorderLayout;
import javax.swing.*;

public class TableFrame extends JFrame{
    public TableFrame()
    {
        super("Testing Table");
        setLayout(new BorderLayout());
        // Initialize column headings
        final String[] colHeads = { "Name", "Phone", "Fax" };
        // Initialize data
        final Object[][] data = {
            { "Gail", "4567", "8675" },
            { "Ken", "7566", "5555" },
            { "Viviane", "5634", "5887" },
            { "Melanie", "7345", "9222" },
            { "Anne", "1237", "3333" },
            { "John", "5656", "3144" },
            { "Matt", "5672", "2176" },
            { "Claire", "6741", "4244" },
            { "Erwin", "9023", "5159" },
            { "Ellen", "1134", "5332" },
            { "Jennifer", "5689", "1212" },
            { "Ed", "9030", "1313" },
            { "Helen", "6751", "1415" }
        };
        // Create the table
        JTable table = new JTable(data, colHeads);
        // Add table to a scroll pane
        int v = JScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
        int h = JScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
        JScrollPane scrollPane = new JScrollPane(table, v, h);
        // Add scroll pane to Frame
        add(scrollPane, BorderLayout.CENTER);
    }
}
```

//TableDemo.java

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



```

public class TableDemo
{
    public static void main( String[] args )
    {
        TableFrame tableFrame = new TableFrame();
        tableFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        tableFrame.setSize( 250, 200 ); // set frame size
        tableFrame.setVisible( true ); // display frame
    } // end main
} // end class Table

```

output



Name	Fax	Phone
Gail	8675	4567
Ken	5555	7566
Viviane	5887	5634
Melanie	9222	7345
Anne	3333	1237
John	3144	5656
Matt	2176	5672
Claire	4244	6741
Erwin	5159	9023

Trees

A tree is a component that presents a hierarchical view of data. A user has the ability to expand or collapse individual subtrees in this display. Trees are implemented in Swing by the **JTree class**, which extends JComponent.

A JTree object generates events when a node is expanded or collapsed. The **addTreeExpansionListener()** and **removeTreeExpansionListener()** methods allow listeners to register and unregister for these notifications. The signatures of these methods are :

```

void addTreeExpansionListener(TreeExpansionListener tel)
void removeTreeExpansionListener(TreeExpansionListener tel)

```

The **getPathForLocation()** method is used to translate a mouse click on a specific point of the tree to a tree path. Its signature is shown here:

```

TreePath getPathForLocation(int x, int y)

```

Here, x and y are the coordinates at which the mouse is clicked. The return value is a TreePath object that encapsulates information about the tree node that was selected by the user.

The **TreeNode interface** declares methods that obtain information about a tree node. For example, it is possible to obtain a reference to the parent node or an enumeration of the child nodes. The **MutableTreeNode interface extends TreeNode**. It declares methods that can insert and remove child nodes or change the parent node.

The **DefaultMutableTreeNode** class implements the **MutableTreeNode** interface. It represents a node in a tree. One of its constructors is shown here:

DefaultMutableTreeNode(Object obj)

Here, obj is the object to be enclosed in this tree node. The new tree node doesn't have a parent or children.

To create a hierarchy of tree nodes, the **add()** method of DefaultMutableTreeNode can be used. Its signature is shown here:

void add(DefaultMutableTreeNode child)

Here, child is a mutable tree node that is to be added as a child to the current node.

Here are the steps that should be followed to use a tree in a JFrame:

- 1. Create a JTree object.**
- 2. Create a JScrollPane object.** (The arguments to the constructor specify the tree and the policies for vertical and horizontal scroll bars.)
- 3. Add the tree to the scroll pane.**
- 4. Add the scroll pane to the JFrame.**

//TreeFrame.java

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import java.awt.BorderLayout;
import javax.swing.*;
import javax.swing.tree.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

public class TreeFrame extends JFrame{

    private JTextField textField ;
    private JTree tree;
    public TreeFrame(){
        super("Testing Tree");
        setLayout(new BorderLayout());
        // Create top node of tree
        DefaultMutableTreeNode top = new DefaultMutableTreeNode("Options");
        // Create subtree of "A"
        DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");
        top.add(a);
        DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("A1");
        a.add(a1);
        DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("A2");
        a.add(a2);
        // Create subtree of "B"
        DefaultMutableTreeNode b = new DefaultMutableTreeNode("B");
        top.add(b);
        DefaultMutableTreeNode b1 = new DefaultMutableTreeNode("B1");
        b.add(b1);
        DefaultMutableTreeNode b2 = new DefaultMutableTreeNode("B2");
        b.add(b2);
        DefaultMutableTreeNode b3 = new DefaultMutableTreeNode("B3");
        b.add(b3);
```

```

// Create tree
tree = new JTree(top);
// Add tree to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane scrollPane = new JScrollPane(tree, v, h);
// Add scroll pane to Frame
add(scrollPane, BorderLayout.CENTER);
// Add text field to Frame
textField = new JTextField("", 20);
add(textField, BorderLayout.SOUTH);
// Anonymous inner class to handle mouse clicks
tree.addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent me) {
        doMouseClicked(me);
    }
});
}

void doMouseClicked(MouseEvent me) {
    TreePath tp = tree.getPathForLocation(me.getX(), me.getY());
    if(tp != null)
        textField.setText(tp.toString());
    else
        textField.setText("");
}
}

```

//TreeDemo.java

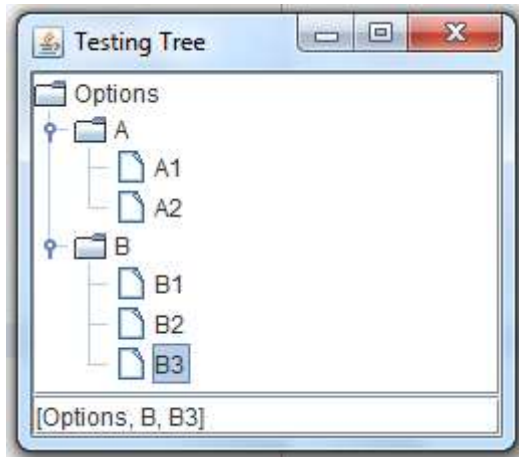
```
import javax.swing.JFrame;
```

```

public class TreeDemo
{
    public static void main( String[] args )
    {
        TreeFrame treeFrame = new TreeFrame();
        treeFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        treeFrame.setSize( 250, 200 ); // set frame size
        treeFrame.setVisible( true ); // display frame
    } // end main
} // end class Tree

```

output



Creating GUI components in NetBeans using drag and drop

Step 1: Create a New Project

Step 2: Choose General -> Java Application

Step 3: Set a Project Name "CelsiusConverterProject". Make sure to deselect the "Create Main Class" checkbox; leaving this option selected generates a new class as the main entry point for the application, but our main GUI window (created in the next step) will serve that purpose, so checking this box is not necessary. Click the "Finish" button when you are done.

Step 4: Add a JFrame Form-Now right-click the CelsiusConverterProject name and choose New -> JFrame Form (JFrame is the Swing class responsible for the main frame for your application.)

Step 5: Name the GUI Class-Next type CelsiusConverterGUI as the class name, and CelsiusConverter as the package name. The remainder of the fields should automatically be filled in, as shown above. Click the Finish button when you are done.

When the IDE finishes loading, the right pane will display a design-time, graphical view of the CelsiusConverterGUI. It is on this screen that you will visually drag, drop, and manipulate the various Swing components.

NetBeans IDE Basics

It is not necessary to learn every feature of the NetBeans IDE before exploring its GUI creation capabilities. In fact, the only features that you really need to understand are **the Palette, the Design Area, the Property Editor, and the Inspector.**

The Palette

The Palette contains all of the components offered by the Swing API (JLabel is a text label, JList is a drop-down list, etc.).

The Design Area

The Design Area is where you will visually construct your GUI. It has two views: **source view, and design view.** Design view is the default, as shown below. You can toggle between views at any time by clicking their respective tabs.

The Property Editor

The Property Editor does what its name implies: it allows you to edit the properties of each component. The Property Editor is intuitive to use; in it you will see a series of rows — one row per property — that you can click and edit without entering the source code directly.

The Inspector

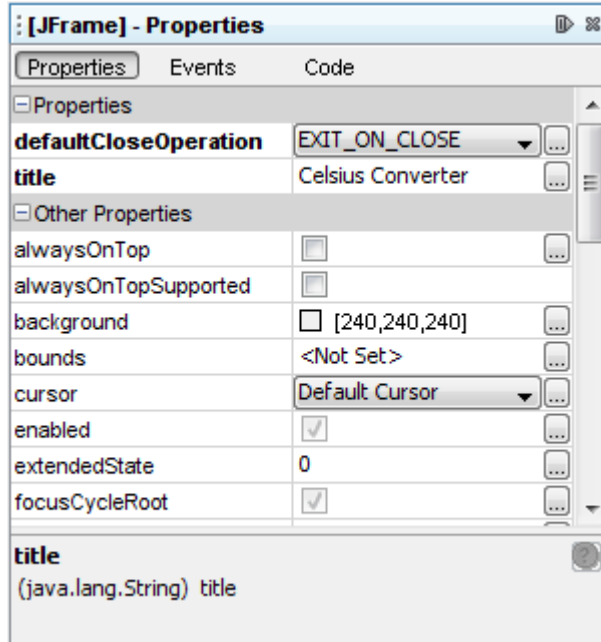
The Inspector provides a graphical representation of your application's components. We will use the Inspector only once, to change a few variable names to something other than their defaults.

Creating the CelsiusConverter GUI

This section explains how to use the NetBeans IDE to create the application's GUI. As you drag each component from the Palette to the Design Area, the IDE auto-generates the appropriate source code.

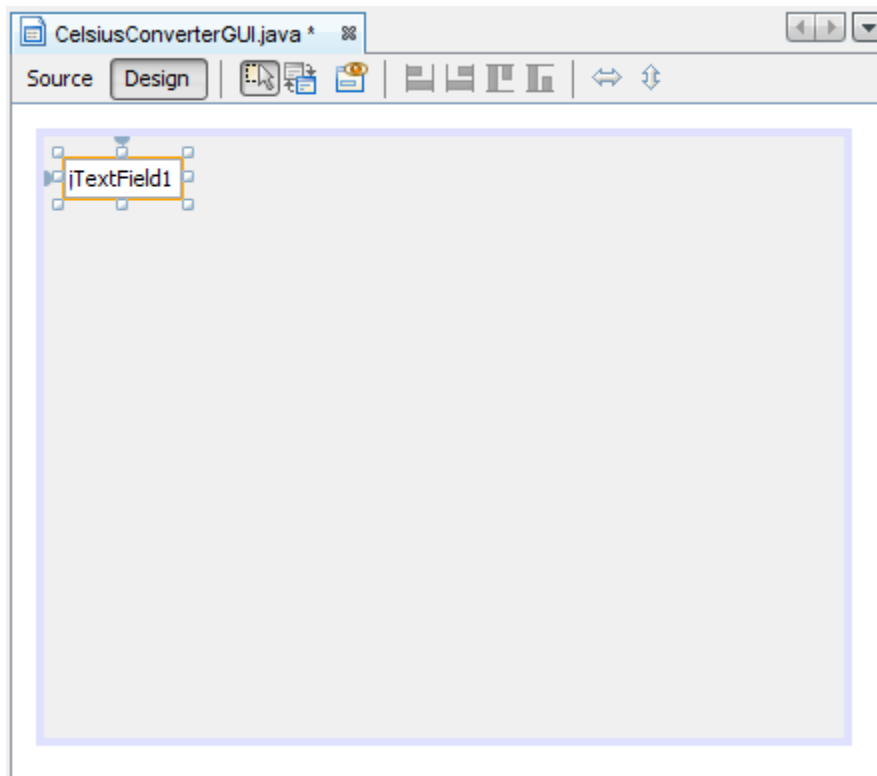
Step 1: Set the Title

First, set the title of the application's JFrame to "Celsius Converter", by single-clicking the JFrame in the Inspector: Then, set its title with the Property Editor:



Step 2: Add a JTextField

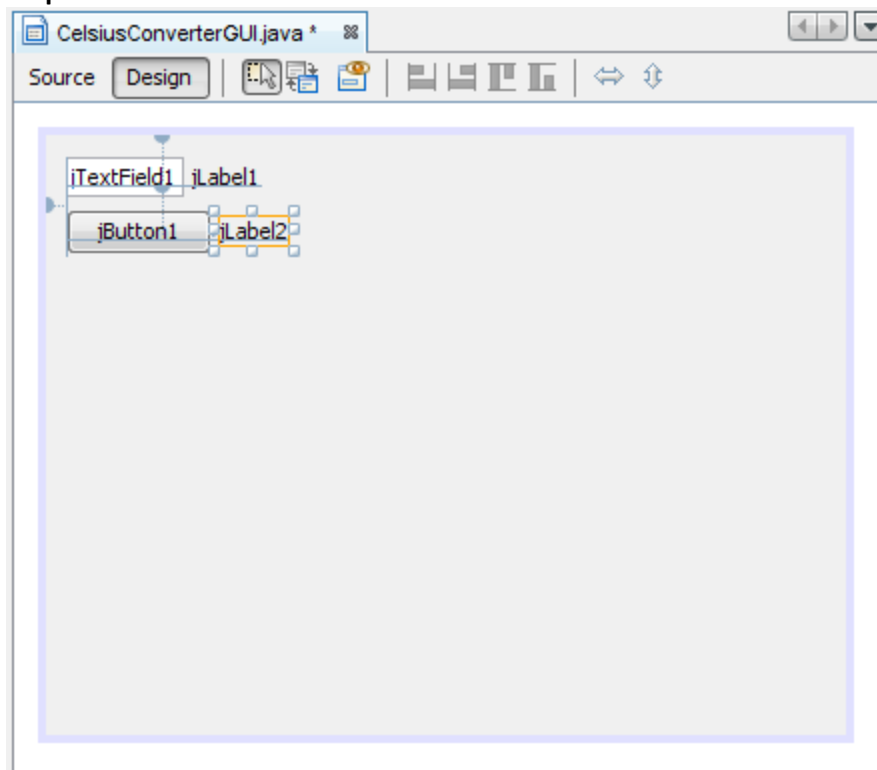
Next, drag a JTextField from the Palette to the upper left corner of the Design Area. As you approach the upper left corner, the GUI builder provides visual cues (dashed lines) that suggest the appropriate spacing. Using these cues as a guide, drop a JTextField into the upper left hand corner of the window as shown below:



Step 3: Add a JLabel

Step 4: Add a JButton

Step 5: Add a Second JLabel



Adjusting the CelsiusConverter GUI

Step 1: Set the Component Text

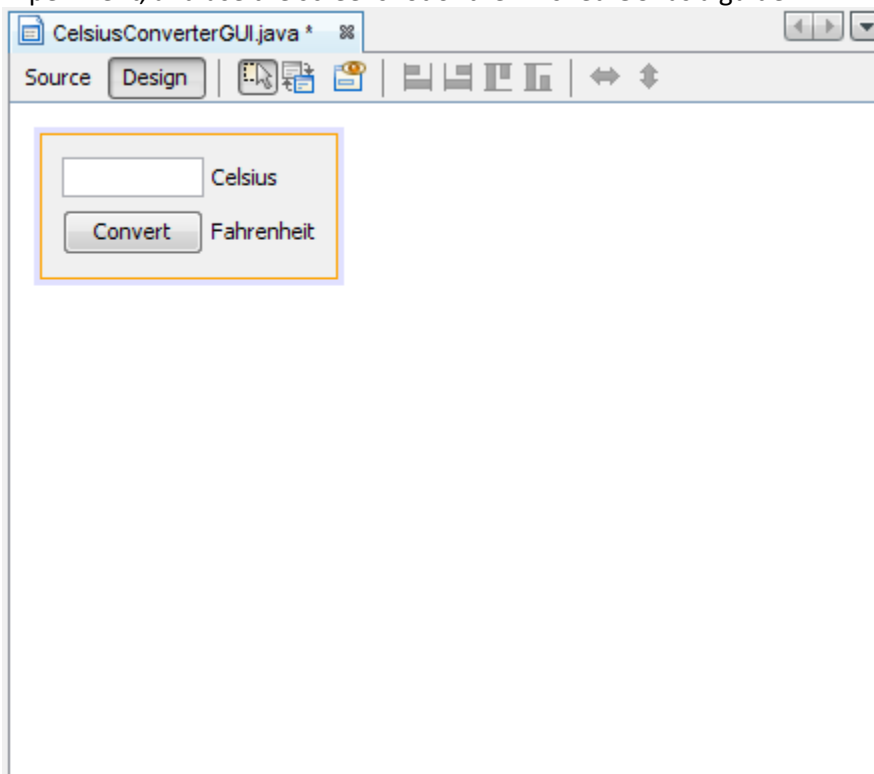
First, double-click the JTextField and JButton to change the default text that was inserted by the IDE. When you erase the text from the JTextField, it will shrink in size as shown below. Change the text of the JButton from "JButton1" to "Convert." Also change the top JLabel text to "Celsius" and the bottom to "Fahrenheit."

Step 2: Set the Component Size

Next, shift-click the JTextField and JButton components. This will highlight each showing that they are selected. Right-click (control-click for mac users) Same Size -> Same Width. The components will now be the same width, as shown below. When you perform this step, make sure that JFrame itself is not also selected. If it is, the Same Size menu will not be active.

Step 3: Remove Extra Space

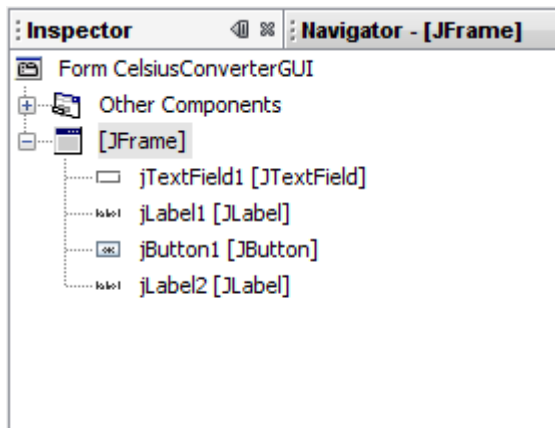
Finally, grab the lower right-hand corner of the JFrame and adjust its size to eliminate any extra whitespace. Note that if you eliminate all of the extra space (as shown below) the title (which only appears at runtime) may not show completely. The end-user is free to resize the application as desired, but you may want to leave some extra space on the right side to make sure that everything fits correctly. Experiment, and use the screenshot of the finished GUI as a guide.



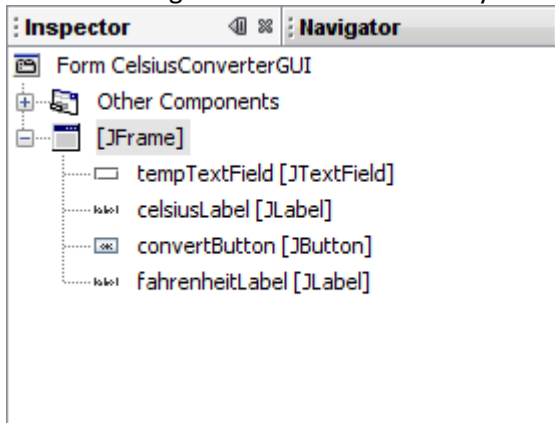
Adding the Application Logic

Step 1: Change the Default Variable Names

The figure below shows the default variable names as they currently appear within the Inspector. For each component, the variable name appears first, followed by the object's type in square brackets. For example, jTextField1 [JTextField] means that "jTextField1" is the variable name and "JTextField" is its type.

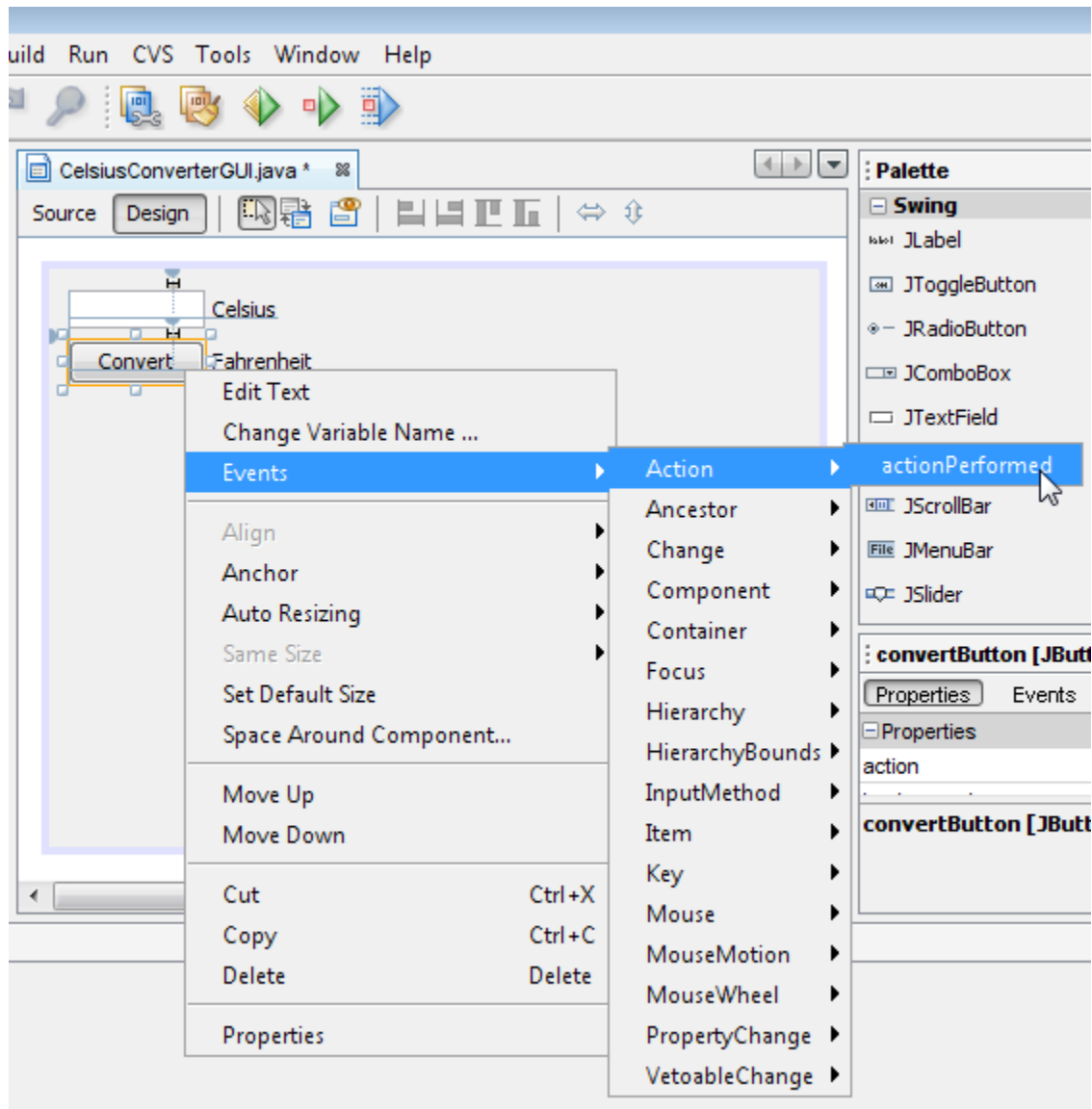


The default names are not very relevant in the context of this application, so it makes sense to change them from their defaults to something that is more meaningful. Right-click each variable name and choose "Change variable name." When you are finished, the variable names should appear as follows:

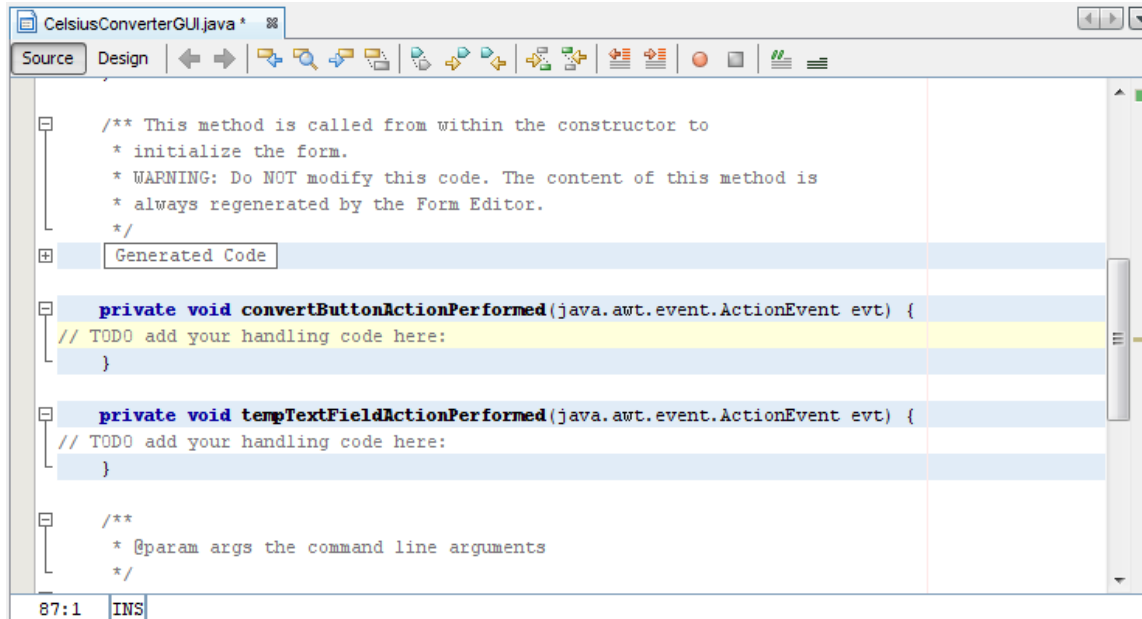


Step 2: Register the Event Listeners

When an end-user interacts with a Swing GUI component (such as clicking the Convert button), that component will generate a special kind of object — called an event object — which it will then broadcast to any other objects that have previously registered themselves as listeners for that event. The NetBeans IDE makes event listener registration extremely simple:



In the Design Area, click on the Convert button to select it. Make sure that only the Convert button is selected (if the JFrame itself is also selected, this step will not work.) Right-click the Convert button and choose Events -> Action -> actionPerformed. This will generate the required event-handling code, leaving you with empty method bodies in which to add your own functionality:



```
/** This method is called from within the constructor to
 * initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is
 * always regenerated by the Form Editor.
 */
Generated Code

private void convertButtonActionPerformed(java.awt.event.ActionEvent evt) {
// TODO add your handling code here:
}

private void tempTextFieldActionPerformed(java.awt.event.ActionEvent evt) {
// TODO add your handling code here:
}

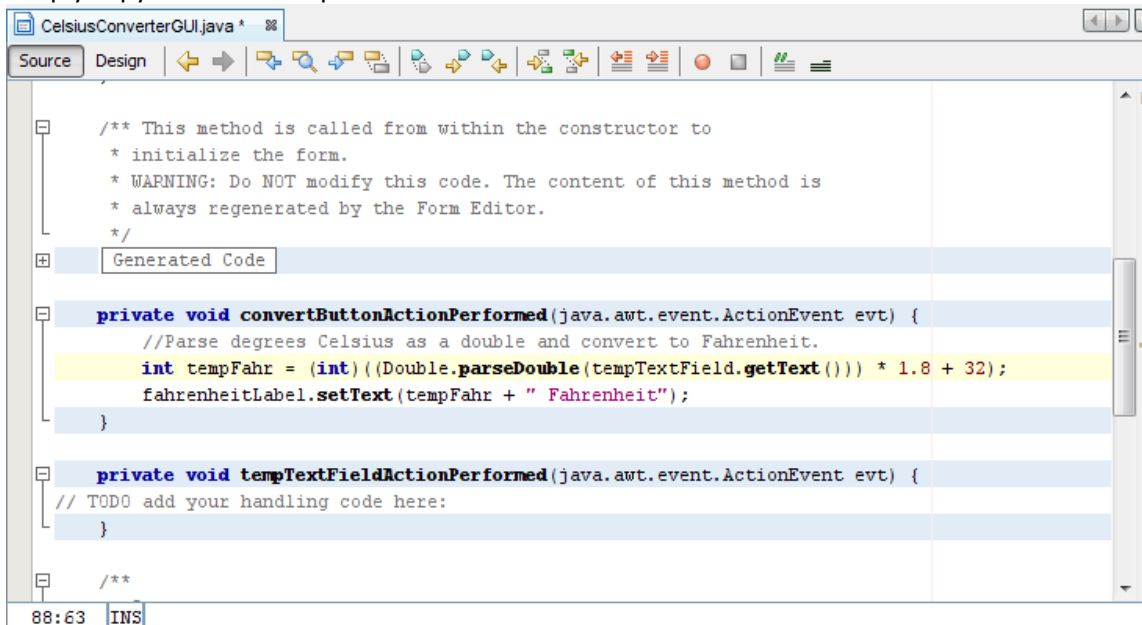
/**
 * @param args the command line arguments
 */
```

Step 3: Add the Temperature Conversion Code

The final step is to simply paste the temperature conversion code into the empty method body.

```
//Parse degrees Celsius as a double and convert to Fahrenheit.
int tempFahr = (int)((Double.parseDouble(tempTextField.getText()))
    * 1.8 + 32);
fahrenheitLabel.setText(tempFahr + " Fahrenheit");
```

Simply copy this code and paste it into the convertButtonActionPerformed method as shown below:



```
/** This method is called from within the constructor to
 * initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is
 * always regenerated by the Form Editor.
 */
Generated Code

private void convertButtonActionPerformed(java.awt.event.ActionEvent evt) {
//Parse degrees Celsius as a double and convert to Fahrenheit.
int tempFahr = (int)((Double.parseDouble(tempTextField.getText())) * 1.8 + 32);
fahrenheitLabel.setText(tempFahr + " Fahrenheit");
}

private void tempTextFieldActionPerformed(java.awt.event.ActionEvent evt) {
// TODO add your handling code here:
}

/**
```

Step 4: Run the Application