CNT 4714: Enterprise Computing Spring 2009

GUI Components: Part 1

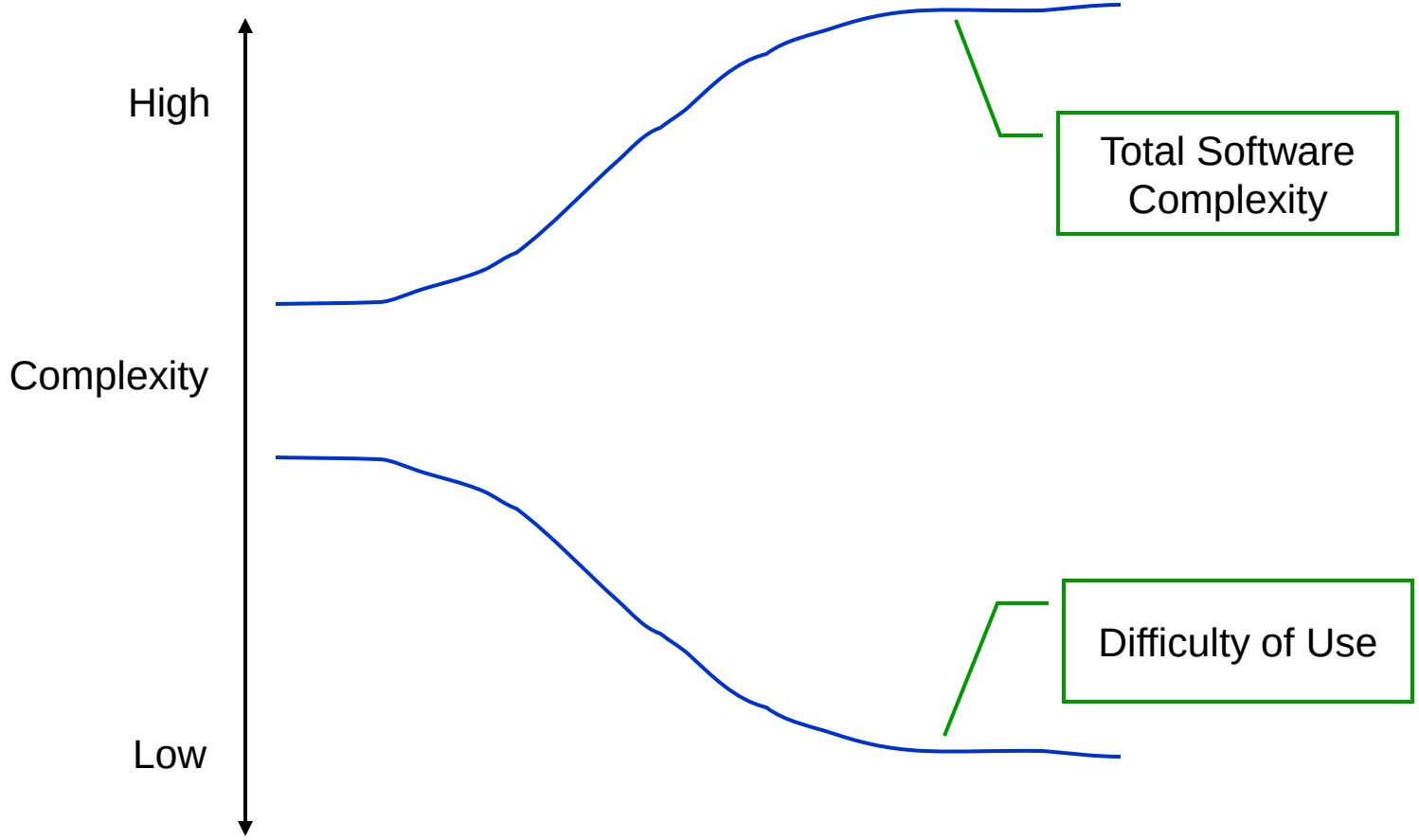
Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cnt4714/spr2009

School Electrical Engineering and Computer Science
University of Central Florida

GUI and Event-Driven Programming

- Most users of software will prefer a graphical user-interface (**GUI**) -based program over a console-based program any day of the week.
- A GUI gives an application a distinctive "look" and "feel".
- Providing different applications with consistent, intuitive user interface components allows users to be somewhat familiar with an application, so that they can learn it more quickly and use it more productively.
- Studies have found that users find GUIs easier to manipulate and more forgiving when misused.
- The GUI ease of functionality comes at a programming price – GUI-based programs are more complex in their structure than console-based programs.

The Trade-off Between Ease of Use and Software Complexity



Popularity of GUIs

- Despite the complexity of GUI programming, its dominance in real-world software development makes it imperative that GUI programming be considered.
- Do not confuse GUI-based programming with applets. Although some of the features of the first few GUIs that we look at will be similar to those used in applet programs, notice that we are developing application programs here not applets.
  - The execution of a GUI-based application also begins in its method main(). However, method main() is normally responsible only for creating an *instance* of the GUI.
  - After creating the GUI, the flow of control will shift from the main() method to an event-dispatching loop that will repeatedly check for user interactions with the GUI.

Components of the GUI

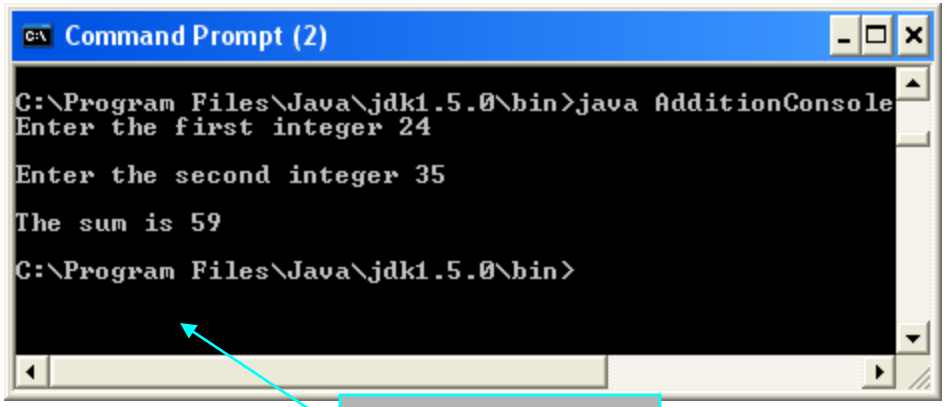
- GUI's are built from GUI components. These are sometimes called controls or widgets (short for *windows gadgets*) in languages other than Java.
- A GUI component is an object with which the user interacts via the mouse, keyboard, or some other input device (voice recognition, light pen, etc.).
- Many applications that 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 prepackaged dialog boxes for both input and output.
  - These dialogs are displayed by invoking static JOptionPane methods.
- The simple example on the next page illustrates this concept.

```
// A simple integer addition program that uses JOptionPane for input and output.
import javax.swing.JOptionPane; // program uses JOptionPane class

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.INFORMATION_MESSAGE );
   } // end method main
} // end class Addition
```

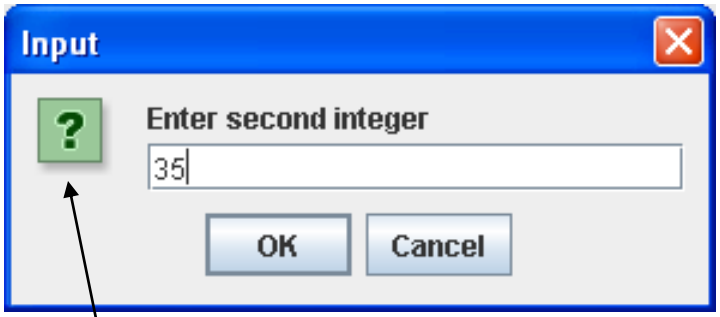
Example GUI illustrating The JOptionPane class

This parameter, called the Message Dialog Constant, indicates the type of information that the box is displaying to the user and will cause the appropriate icon to appear in the dialog box (see next page).

Output from execution of the Addition Example

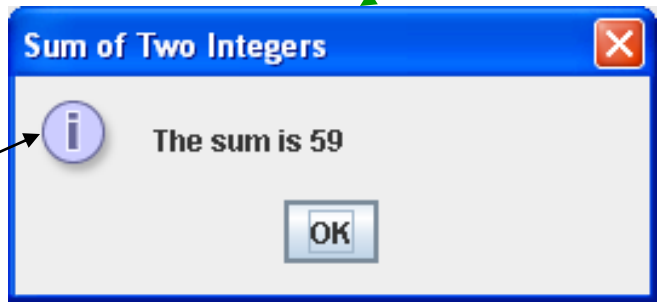


Console-based version



User enters their integers in the dialog box and clicks OK after entering each.

Result is displayed in a third dialog box.



The "?" icons appear because of the input dialog, the "I" icon appears because of a specific parameter.

Overview of Swing Components

- Java GUI-based programming typically makes use of the `swing` API. The `swing` API provides over 40 different types of graphical components and 250 classes to support the use of these components.
  - JFrame: Represents a titled, bordered window.
  - JTextArea: Represents an editable multi-line text entry component.
  - JLabel: Displays a single line of uneditable text or icons.
  - JTextField: Enables the user to enter data from the keyboard. Can also be used to display editable or uneditable text.
  - JButton: Triggers and event when clicked with the mouse.
  - JCheckBox: Specifies an option that can be selected or not selected.
  - JComboBox: Provides a drop-down list of items from which the user can make a selection by clicking an item or possibly by typing into the box.
  - JList: Provides a list of items from which the user can make a selection by clicking on any item in the list. Multiple elements can be selected.
  - JPanel: Provides an area in which components can be placed and organized. Can also be used as a drawing area for graphics.

Displaying Text and Images in a Window

- Most windows that you will create will be an instance of class `JFrame` or a subclass of `JFrame`.
- `JFrame` provides the basic attributes and behaviors of a window that you expect – a title bar at the top of the window, and buttons to minimize, maximize, and close the window.
- Since an application's GUI is typically specific to the application, most of the examples in this section of the notes will consist of two classes – a subclass of `JFrame` that illustrates a GUI concept and an application class in which main creates and displays the application's primary window.

Labeling GUI Components

- A typical GUI consists of many components. In a large GUI, it can be difficult to identify the purpose of every component unless the GUI designer provides text instructions or information stating the purpose of each component.
- Such text is known as a label and is created with class `JLabel` (which is a subclass of `JComponent`).
- A `JLabel` displays a single line of read-only (noneditable) text, an image, or both text and an image.
- The sample code on the next page illustrates some of the features of the `JLabel` class.

```
// 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 label #1" );
      add( label1 ); // add label #1 to JFrame
```

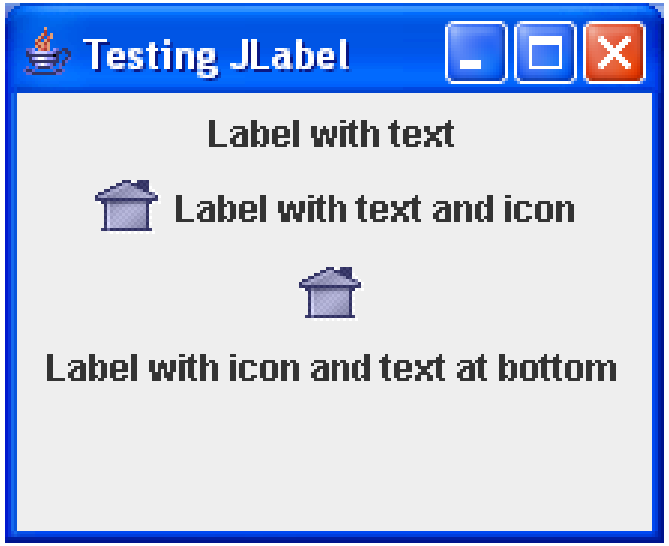
Example GUI illustrating The JLabel class

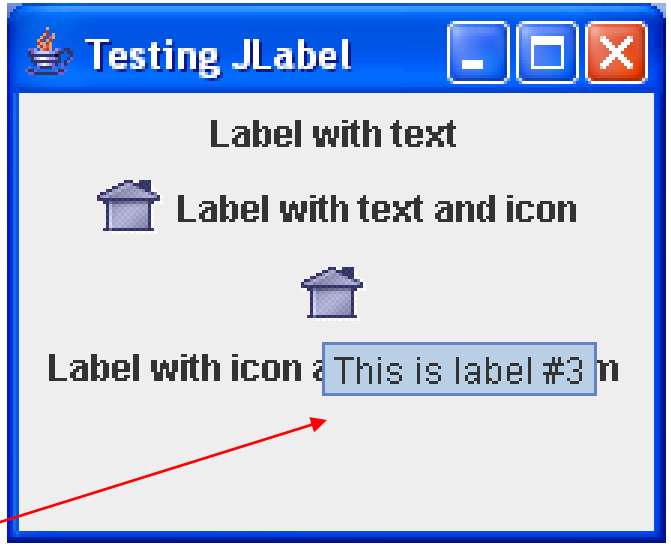
```
    // JLabel constructor with string, Icon and alignment arguments
    Icon home = new ImageIcon( getClass().getResource( "home.gif" ) );
    label2 = new JLabel( "Label with text and icon", home,
       SwingConstants.LEFT );
    label2.setToolTipText( "This is label #2" );
    add( label2 ); // add label #2 to JFrame

    label3 = new JLabel(); // JLabel constructor no arguments
    label3.setText( "Label with icon and text at bottom" );
    label3.setIcon( home ); // add icon to JLabel
    label3.setHorizontalTextPosition( SwingConstants.CENTER );
    label3.setVerticalTextPosition( SwingConstants.BOTTOM );
    label3.setToolTipText( "This is label #3" );
    add( label3 ); // add label3 to JFrame
  } // end LabelFrame constructor
} // end class LabelFrame
```

LabelFrame class continues

```
// Driver class for 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( 220, 180 ); // set frame size
      labelFrame.setVisible( true ); // display frame
   } // end main
} // end class LabelTest
```


Output from execution of the LabelTest Example





Moving cursor over the label will display ToolTipText if GUI designer provided it.

GUI Programming

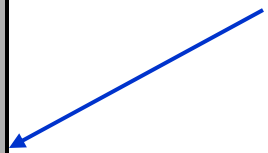
- Besides having a different look and feel from console-based programs, GUI-based programs follow a different program execution paradigm – *event-driven programming*.
- A console-based program begins and ends in its `main()` method. To solve the problem, the main method statements are executed in order. After the last statement in `main()` executes, the program terminates.
- The execution of a GUI-based program also begins in its `main()` method. Normally the main method is responsible only for creating an instance of the GUI. After creating the GUI, the flow of control passes to an *event-dispatching loop* that repeatedly checks for user interactions with the GUI through *action events*. When an event occurs, an *action performer* method is invoked for each *listener* of that event. The performer then processes the interaction. After the performer handles the event, control is given again to the event-dispatching loop to watch for future interactions. This process continues until an action performer signals that the program has completed its task.

Console-based Execution

Console program

```
Method main() {
statement1;
statement2;
...
statementm;
}
```

Console programs begin and end in main() method.



GUI-based Execution

GUI Program

```
main() {
   GUI gui = new GUI();
}

GUI Constructor() {
   constructor1;
   constructor2;
   ...
   constructorn;
}

Action Performer() {
  action1;
  action2;
   ...
  actionk;
}
```

GUI program begins in method main(). The method creates a new instance of the GUI by invoking the GUI constructor. On completion, the event dispatching loop is started.

The constructor configures the components of the GUI. Part of the configuration is registering the listener-performer for user interactions.

The action performer implements the task of the GUI. After it completes, the event-dispatching loop is restarted.

Event Dispatching Loop

```
do
   if an event occurs then
      signal its action listeners
until program ends
```

The event dispatching loop watches for user interactions with the GUI. When a user event occurs, the listener-performers for that event are notified.

GUI Program Structure

- GUI-based programs typically have at least three methods.
  - One method is the class method main() that defines an instance of the GUI.
  - The creation of that object is accomplished by invoking a constructor method that creates and initializes the components of the GUI. The constructor also registers any event listeners that handle any program-specific responses to user interactions.
  - The third method is an action performer instance method that processes the events of interest. For many GUIs there is a separate listener-performer object for each of the major components of the GUI.
  - An action performer is always a **public** instance method with name `actionPerformed()`.

GUI Program Structure (cont.)

- GUI-based programs also have instance variables for representing the graphical components and the values necessary for its task.
- Thus, a GUI is a true object. Once constructed, a GUI has attributes and behaviors.
  - The attributes are the graphical component instance variables.
  - The behaviors are the actions taken by the GUI when events occur.

Specifying A GUI Layout

- When building a GUI, each GUI component must be attached to a container, such as a window created with a `JFrame`. Typically, you must also decide where to position each GUI component within the container. This is known as specifying the layout of the GUI components.
- Java provides several layout managers that can help you position components if you don't wish for a truly custom layout.
- Many IDEs provide GUI design tools in which you specify the exact size and location of each component in a visual manner using the mouse. The IDE then generates the GUI code automatically.

GridLayout Manager

- As with all layout managers, you can call the setLayout() method of the container (or panel) and pass in a layout manager object. This is done as follows:

```
//Just use new in the method call because we don't need
//a reference to the layout manager.
Container canvas = getContentPane();
canvas.setLayout(new GridLayout(4,2));
```

- Or you could create an object and pass the object to the setLayout() method as follows:

```
//Create a layout manager object and pass it to the
//setLayout() method.
Container canvas = getContentPane();
GridLayout grid = new GridLayout(4,2);
canvas.setLayout(grid);
```

GridLayout Manager (cont.)

- The following program illustrates how you can change the layout dimension if necessary and repaint the window.
- This is accomplished by calling the JFrame's validate() method to layout current components and repaint() calls paint().
- Program `GridDemo.java` fills a frame's container with twelve buttons, and ten of the button labels are the names of cities. Only two buttons are active. When the "Show Florida Cities" button is clicked, the layout then shows the buttons with Florida cities. When the "Show Maryland Cities" button is clicked, the layout changes to show only the cities in Maryland. Each of these two dialogs also has one active button, "Show All Cities". This button toggles back to the four by three view of all the city buttons.

Example GUI illustrating the GridLayout Manager

```
//File:  GridDemo.java
//This program sets a 4x3 grid layout and then changes it
//to a different grid layout based on the user's choice.

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

public class GridDemo extends JFrame   {
        //Set up an array of 13 buttons.
        JButton buttonArray [] = new JButton [13];
        //Set up a String array for the button labels.
        String buttonText[] =  { "Orlando", "New York","Rock Creek", "Miami",
                "Bethesda", "Santa Fe", "Baltimore", "Oxon Hill", "Chicago",
        "Sarasota",   "Show Florida Cities", "Show Maryland Cities", "Show All
        Cities" };
        Container canvas = getContentPane();
        public GridDemo()  {
                //Here's where we make our buttons and set their text.
                for(int i = 0; i<buttonArray.length; ++i)
                        buttonArray[i] = new JButton( buttonText[i] );
                addAllTheCities();
                buttonArray[10].setBackground(Color.cyan);
                buttonArray[11].setBackground(Color.magenta);
                buttonArray[12].setBackground(Color.green);
```


```
//Just going to show the Florida cities.
buttonArray[10].addActionListener( new ActionListener()        {
                public void actionPerformed( ActionEvent ae)        {
                                addFloridaCities();
                                //validate() causes a container to lay out its
                                //components again after the components it
                                //contains have been added to or modified.
                                canvas.validate();
                                //repaint() forces a call to paint() so that the
                                //window is repainted.
                                canvas.repaint();
                }
} );

//Just going to show the Maryland cities.
buttonArray[11].addActionListener( new ActionListener()   {
                public void actionPerformed( ActionEvent ae)        {
                                addMarylandCities();
                                //validate() causes a container to lay out its
                                //components again after the components it
                                //contains have been added to or modified.
                                canvas.validate();
                                //repaint() forces a call to paint() so that the
                                //window is repainted.
                                canvas.repaint();
                }
} );
```

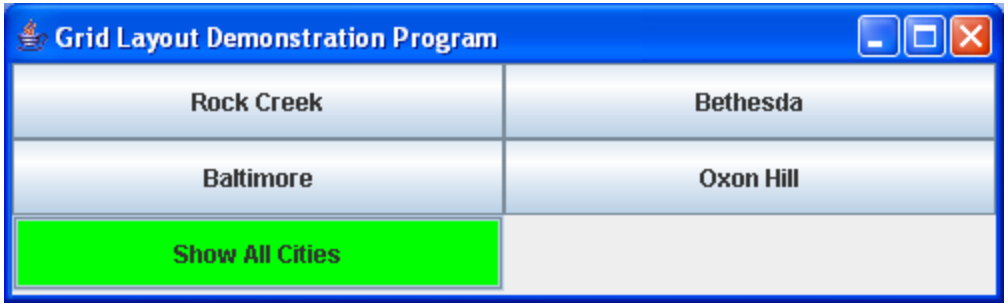
```
                //Now show all the cities.
                buttonArray[12].addActionListener( new ActionListener()   {
                        public void actionPerformed( ActionEvent ae)        {
                                addAllTheCities();
                                canvas.validate();
                                canvas.repaint();
                        }
                } );
                this.setSize(500,150);
                this.setTitle("Grid Layout Demonstration Program ");
                this.show();
}

//Sets the container's canvas to 4x3 and adds all cities.
public void addAllTheCities()  {
                canvas.removeAll();
                canvas.setLayout(new GridLayout(4, 3));
                for(int i = 0; i < 12; ++i)  {
                      canvas.add(buttonArray[i]);
                }
}
```


```
        //Sets the container's canvas to 2x2 and adds Florida cities.
        public void addFloridaCities()  {
                canvas.removeAll();
                canvas.setLayout(new GridLayout(2, 2));
                canvas.add(buttonArray[0]);
                canvas.add(buttonArray[3]);
                canvas.add(buttonArray[9]);
                canvas.add(buttonArray[12]);
        }
        //Sets the container's canvas to 3x2 and adds Maryland cities.
        public void addMarylandCities()         {
                canvas.removeAll();
                canvas.setLayout(new GridLayout(3, 2));
                canvas.add(buttonArray[2]);
                canvas.add(buttonArray[4]);
                canvas.add(buttonArray[6]);
                canvas.add(buttonArray[7]);
                canvas.add(buttonArray[12]);
        }


        public static void main( String args[] )
        {
                GridDemo app = new GridDemo();
                app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
}
```


Initial window

Output from execution of the GridDemo Example

Resized grid after clicking “Show Florida Cities” button



Resized grid after clicking “Show Maryland Cities” button

More on Event Handling

- The previous example illustrates the basic concepts in event handling.
- Before an application can respond to an event for a particular GUI component, you must perform several coding steps:

1. Create a class than represents the event handler.
2. Implement an appropriate interface, known as an event-listener interface, in the class from Step 1.
3. Indicate that an object of the class from Steps 1 and 2 should be notified when the event occurs. This is known as registering the event handler.

Using A Nested Class to Implement an Event Handler

- The examples so far have all utilized only top-level classes, i.e., the classes were not nested inside another class.
- Java allows for nested classes (a class declared inside another class) to be either static or non-static.
- Non-static nested classes are called inner classes and are frequently used for event handling.
  - An inner class is allowed to directly access its top-level class's variables and methods, even if they are private.
- Before an object of an inner class can be created, there must first be an object of the top-level class that contains the inner class. This is required because an inner class object implicitly has a reference to an object of its top-level class.
  - A nested class that is static does not require an object of its top-level class and has no implicit reference to an object in the top-level class.

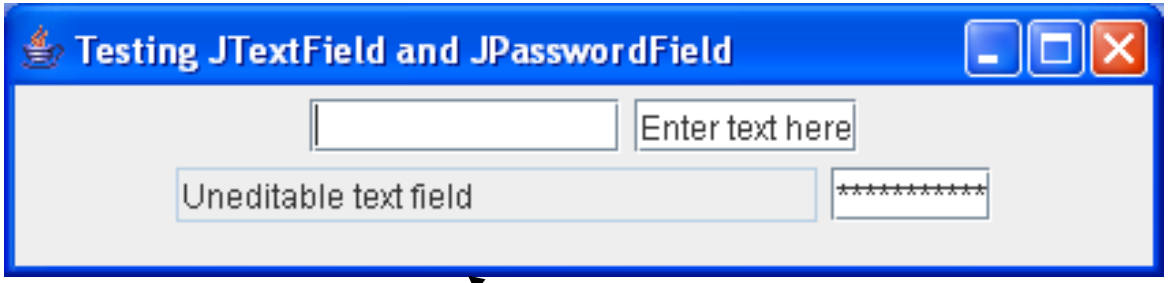
Example GUI illustrating Nested Classes

```
// Demonstrating the JTextField class and nested classes
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 textfield 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",
        new String( passwordField.getPassword() ) );
    // display JTextField content
    JOptionPane.showMessageDialog( null, string );
  } // end method actionPerformed
} // end private inner class TextFieldHandler
} // end class TextFieldFrame
```


```
 // Driver class for 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( 425, 100 ); // set frame size
      textFieldFrame.setVisible( true ); // display frame
   } // end main
} // end class TextFieldTest
```



Initial window when executing TextFieldTest

Summary of Event-Handling Mechanism

- As we've just seen, there are three parts to the event-handling mechanism – the event source, the event object, and the event listener.

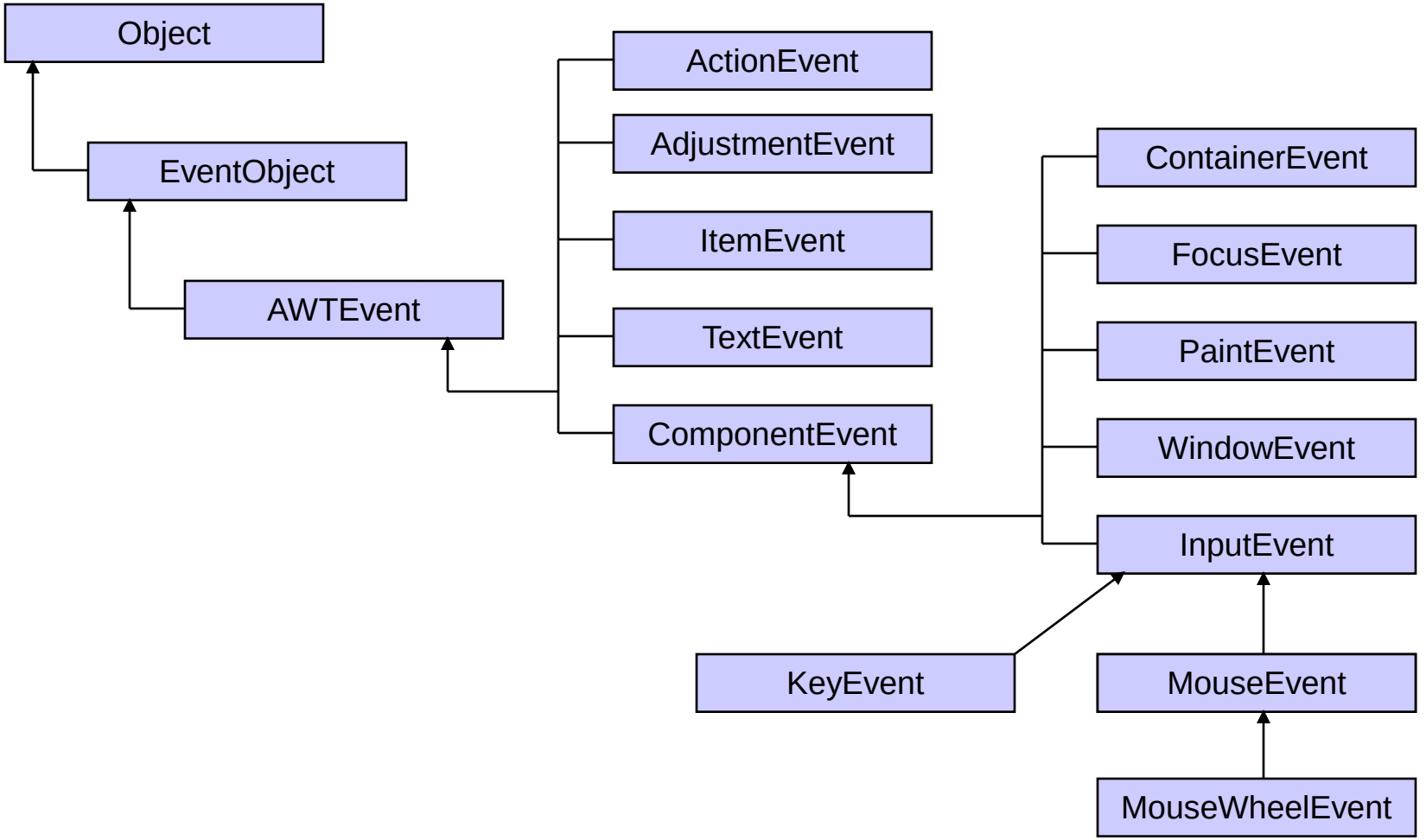
1. The event source is the particular GUI component with which the user interacts.

2. 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.

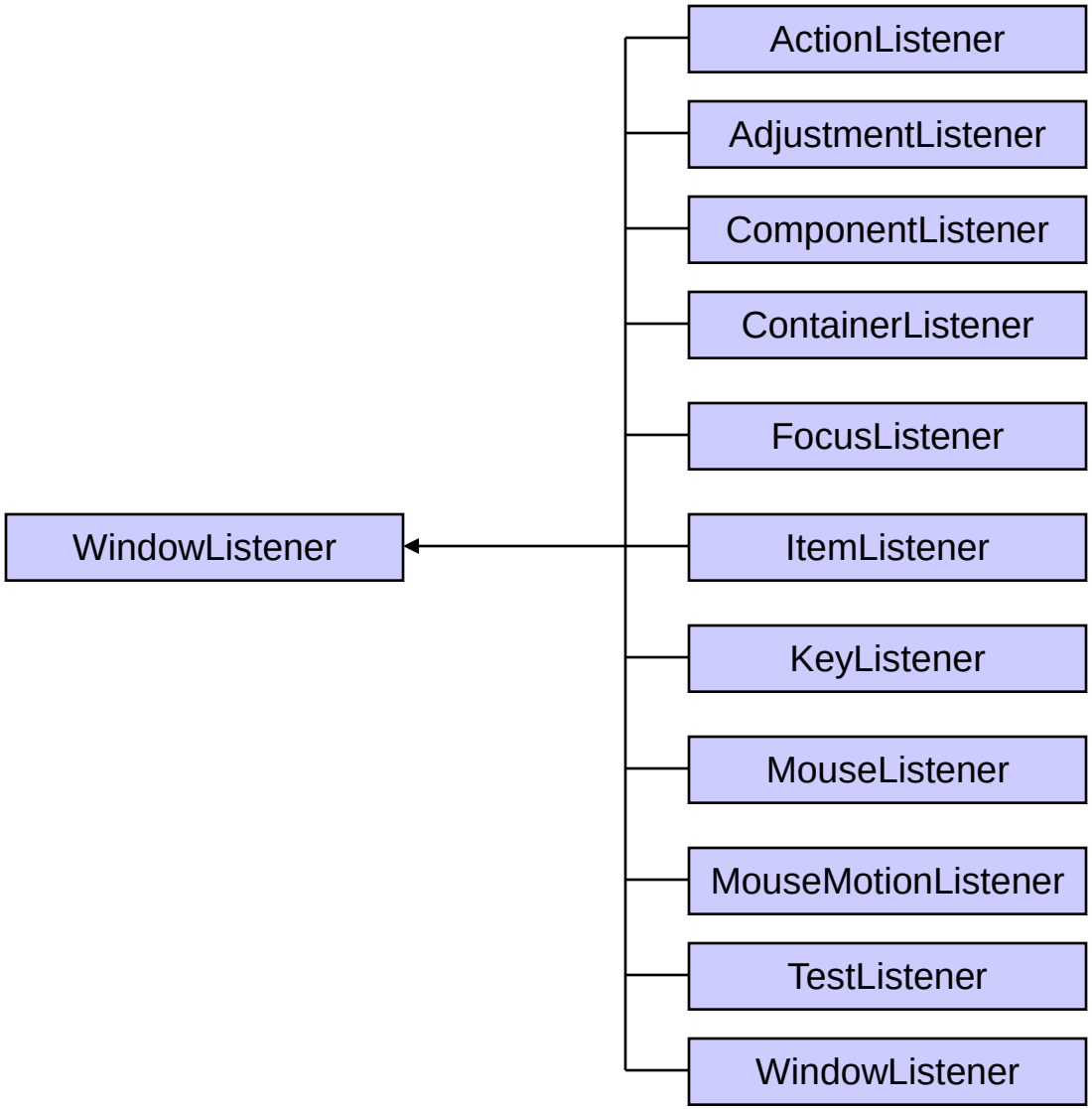
3. The event listener is an object that is notified by the event source when an event occurs; in effect, it "listens" for an event and one of its methods executes in response to the event. A method of the event listener receives an event object when the event listener is notified of the event.

Summary of Event-Handling Mechanism
(cont.)

- The event-handling model as used by Java is known as the delegation event model. In this model an event's processing is delegated to a particular object (the event listener) in the application.
- For each event-object type, there is 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`.
- 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.



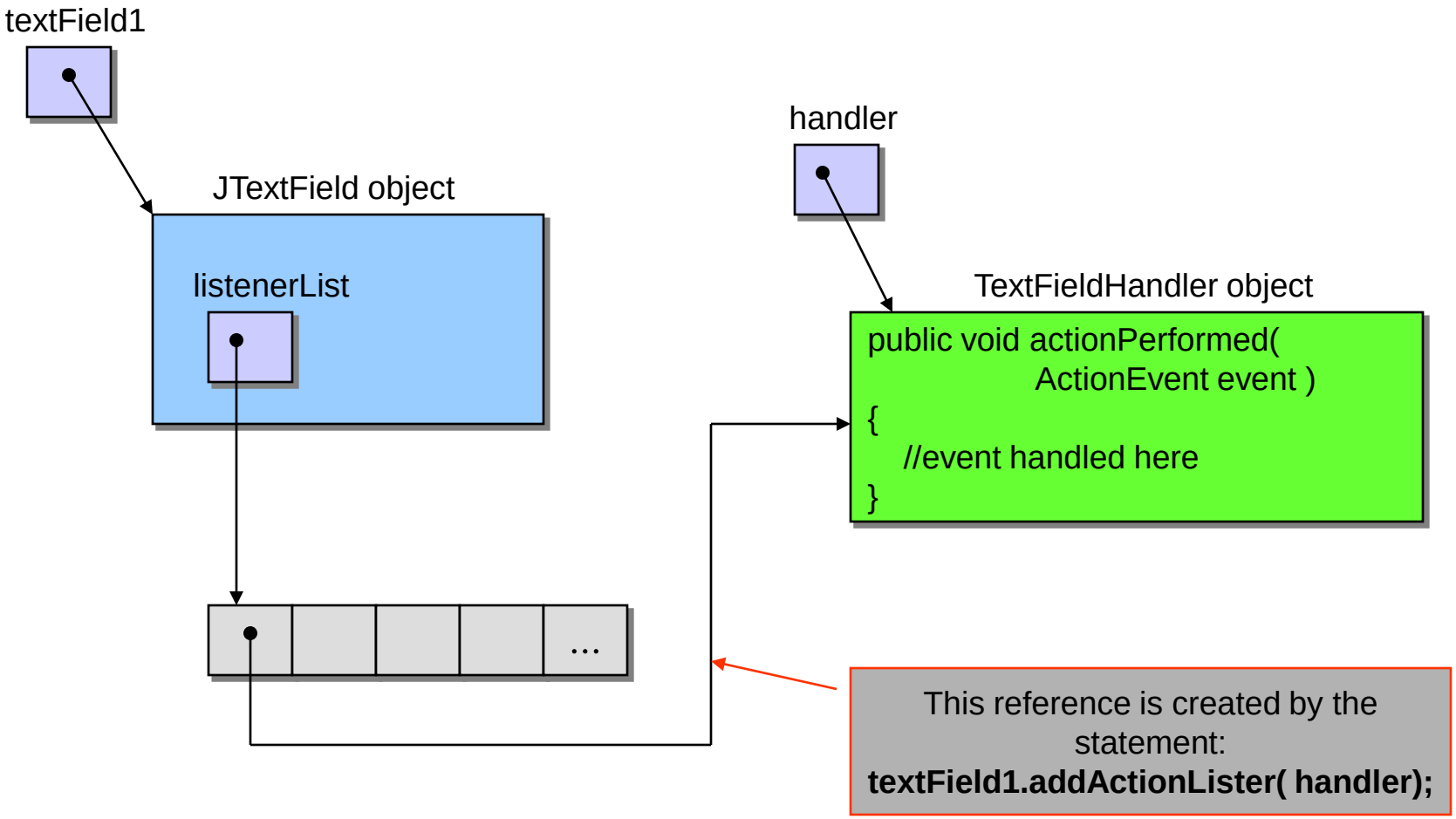
Some event classes of package `java.awt.event`



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

How Event Handling Works

- Registering Events (How the event handler gets registered)
  - Every JComponent has an instance variable called `listenerList` that refers to an object of class `EventListenerList` (package `javax.swing.event`).
  - Each object of a `JComponent` subclass maintains a reference to all its registered listeners in the `listenerList` (for simplicity think of listenerList as an array).
  - Using the code which begins on page 29 (`TextFieldFrame` class) as an example, when the line: `textField1.addActionListener( handler );` executes, a new entry containing a reference to the `TextFieldHandler` object is placed in `textField1's listenerList`. This new entry also includes the listener's type (in this case `ActionListener`).
  - Using this mechanism, each lightweight Swing GUI component maintains its own list of listeners that were registered to handle the component's events.



Event registration for the JTextField textField1 in class TextFieldFrame

How Event Handling Works (cont.)

- Event-Handling Invocation
  - How does the GUI component know which event-handling method to invoke? In our example, how did the GUI component `textField1` know to call `actionPerformed` rather than some other method?
  - Every GUI component supports several event types, including mouse events, key events, and others. When an event occurs, the event is dispatched to only the event listeners of the appropriate type.
  - Dispatching is the process by which the GUI component calls an event-handling method on each of its listeners that are registered for the particular event type that occurred.

How Event Handling Works (cont.)

- Event-Handling Invocation (cont.)
  - Each event type has one or more corresponding event-listener interfaces.
    - For example, `ActionEvents` are handled by `ActionListeners`, `MouseEvents` are handled by `MouseListeners` and `KeyEvents` are handled by `KeyListeners`.
  - When an event occurs, the GUI component receives, from the JVM, a unique event ID specifying the event type. The GUI component uses the event ID to decide the listener type to which the event should be dispatched and to decide which method to call on each listener object.

How Event Handling Works (cont.)

- Event-Handling Invocation (cont.)
  - For an `ActionEvent`, the event is dispatched to every registered `ActionListener`'s `actionPerformed` method (the only method in interface `ActionListener`).
  - For a `MouseEvent`, the event is dispatched to every registered `MouseListener` or `MouseMotionListener`, depending on the mouse event that occurs. The `MouseEvent`'s event ID determined which of the several mouse event-handling methods are called.

JButton

- A button is a component the user clicks to trigger a specific action.
- There are several different types of buttons available to Java applications including, command buttons, check boxes, toggle buttons, and radio buttons.
- The example program on the next page, creates three different buttons, one plain button and two with icons on the buttons. Event handling for the buttons is performed by a single instance of inner class `ButtonHandler`.

GUI illustrating JButton

```
// Playing with 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 saveJButton; // button with save icon
   private JButton helpJButton; //button with help icon

   // 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
```

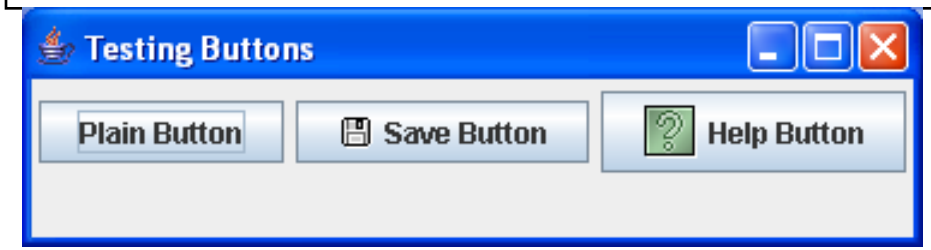

```
    Icon icon1 = new ImageIcon( getClass().getResource( "images/save16.gif" ) );
    Icon icon2 = new ImageIcon( getClass().getResource( "images/help24.gif" ) );
    saveJButton = new JButton( "Save Button", icon1 ); // set image
    helpJButton = new JButton("Help Button", icon2); //set image
    //helpButton.setRolloverIcon( icon1 );
    add( saveJButton ); // add saveJButton to JFrame
    add( helpJButton ); //add helpJButton to JFrame

    // create new ButtonHandler for button event handling
    ButtonHandler handler = new ButtonHandler();
    saveJButton.addActionListener( handler );
    helpJButton.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
```

Command buttons generate an ActionEvent when the user clicks the button.

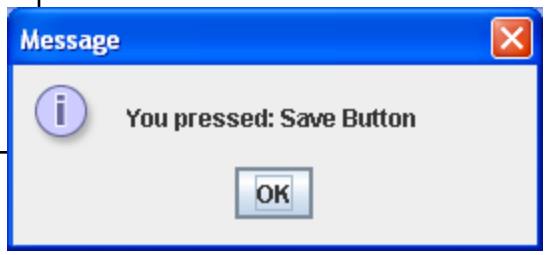
See page 46.

```
// Testing ButtonFrame class.
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( 375,100); // set frame size
      buttonFrame.setVisible( true ); // display frame
   } // end main
} // end class ButtonTest
```

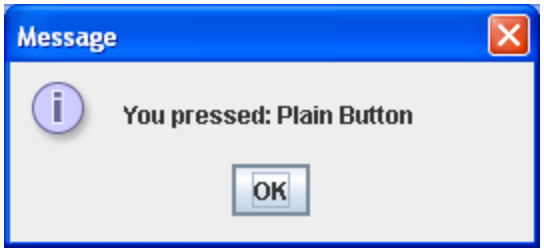
Driver class to test ButtonFrame



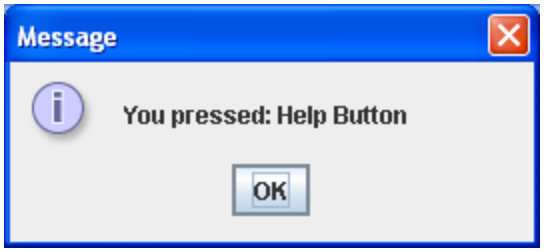
Initial GUI



GUI after user clicks the Save Button



GUI after user clicks the Plain Button



GUI after user clicks the Help Button

Accessing the *this* Reference in an Object of a Top-Level Class From an Inner Class

- When you execute the previous application and click one of the buttons, notice that the message dialog that appears is centered over the application's window.
- This occurs because the call to `JOptionPane` method `showMessageDialog` uses `ButtonFrame.this` rather than `null` as the first argument. When this argument is not null, it represents the parent GUI component of the message dialog (in this case the application window is the parent component) and enables the dialog to be centered over that component when the dialog is displayed.

Buttons That Maintain State

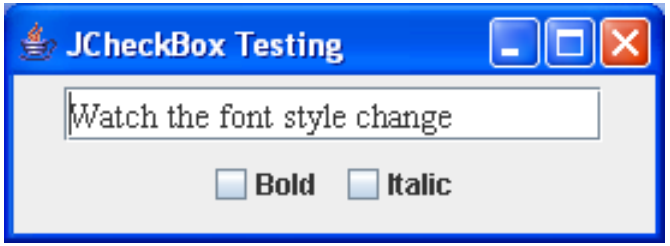
- The Swing GUI components contain three types of state buttons – JToggleButton, JCheckBox and JRadioButton that have on/off or true/false values.
- Classes JCheckBox and JRadioButton are subclasses of JToggleButton.
- A JRadioButton is different from a JCheckBox in that normally several JRadioButtons are grouped together, and are mutually exclusive – only one in the group can be selected at any time.
- The example on the next page illustrates the JCheckBox class.

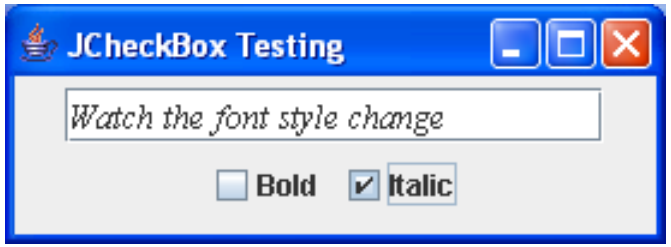
GUI illustrating JCheckBox

```
// Playing with 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 Testing" );
      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
{
    private int valBold = Font.PLAIN; // controls bold font style
    private int valItalic = Font.PLAIN; // controls italic font style
    // respond to checkbox events
    public void itemStateChanged( ItemEvent event )
    {
      // process bold checkbox events
      if ( event.getSource() == boldJCheckBox )
        valBold =
          boldJCheckBox.isSelected() ? Font.BOLD : Font.PLAIN;

      // process italic checkbox events
      if ( event.getSource() == italicJCheckBox )
        valItalic =
          italicJCheckBox.isSelected() ? Font.ITALIC : Font.PLAIN;
```


```
// set text field font
        textField.setFont(
           new Font( "Serif", valBold + valItalic, 14 ) );
     } // end method itemStateChanged
   } // end private inner class CheckBoxHandler
} // end class CheckBoxFrame
```






```
// Driver Class for 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
```


JRadioButton

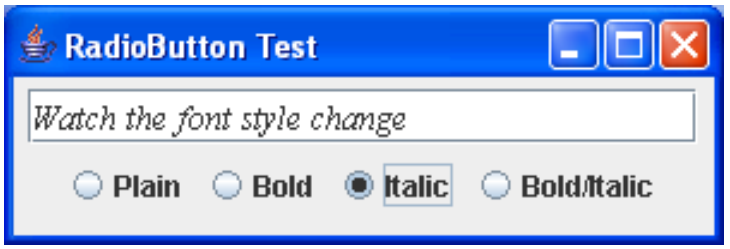
- Radio buttons (declared with class JRadioButton) are similar to checkboxes in that they have two states – selected or deselected. However, radio buttons normally appear as a group in which only one button can be selected at a time.
- Selecting a different radio button forces all others to be deselected.
- Radio buttons are used to represent mutually exclusive options.
- The example application on the following pages illustrates radio buttons. The driver class is not shown but is available on the code page of the course website.

GUI illustrating JRadioButton

```
// Illustration of 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
```



JComboBox

- A combo box (sometimes called a drop-down list) provides a list of items from which the user can make a single selection.
- `JComboBox`es generate `ItemEvents` like `JCheckBox`es and `JRadioButton`s.
- The code for illustrating a JComboBox begins on the next page. Notice that the event handler in this application is an anonymous inner class. This is highlighted in the code.

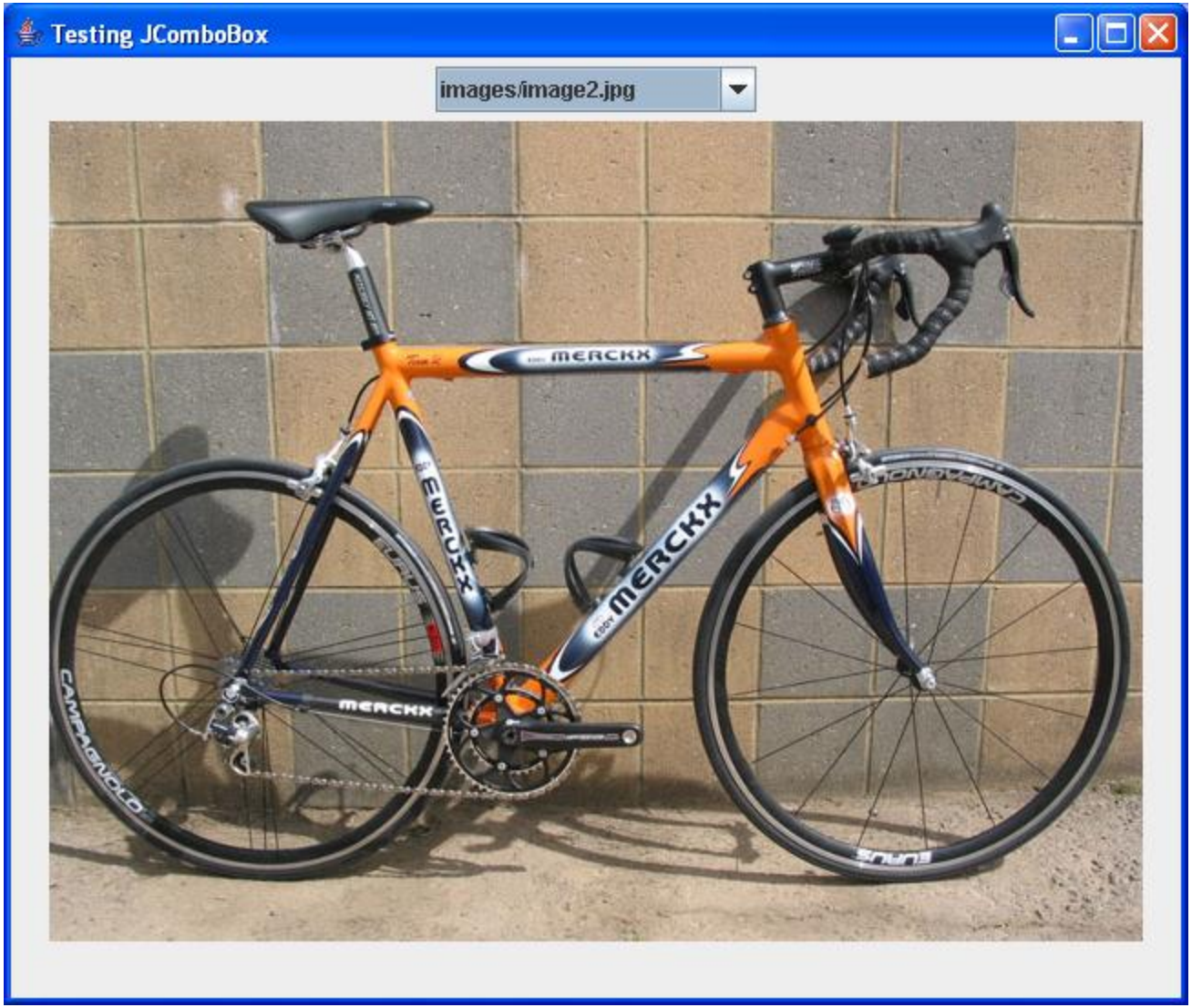
GUI illustrating JComboBox

```
// Illustrating a JComboBox to select an image to display.
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 ComboBoxFrame extends JFrame
{
   private JComboBox imagesJComboBox; // combobox to hold names of icons
   private JLabel label; // label to display selected icon

   private String names[] =
      { "images/image2.jpg", "images/image5.jpg",  "images/image03.jpg",
"images/yellowflowers.png", "images/redflowers.png", "images/purpleflowers.png" };
   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 ] ) ),
      new ImageIcon( getClass().getResource( names[ 4 ] ) ),
      new ImageIcon( getClass().getResource( names[ 5 ] ) )  };
```

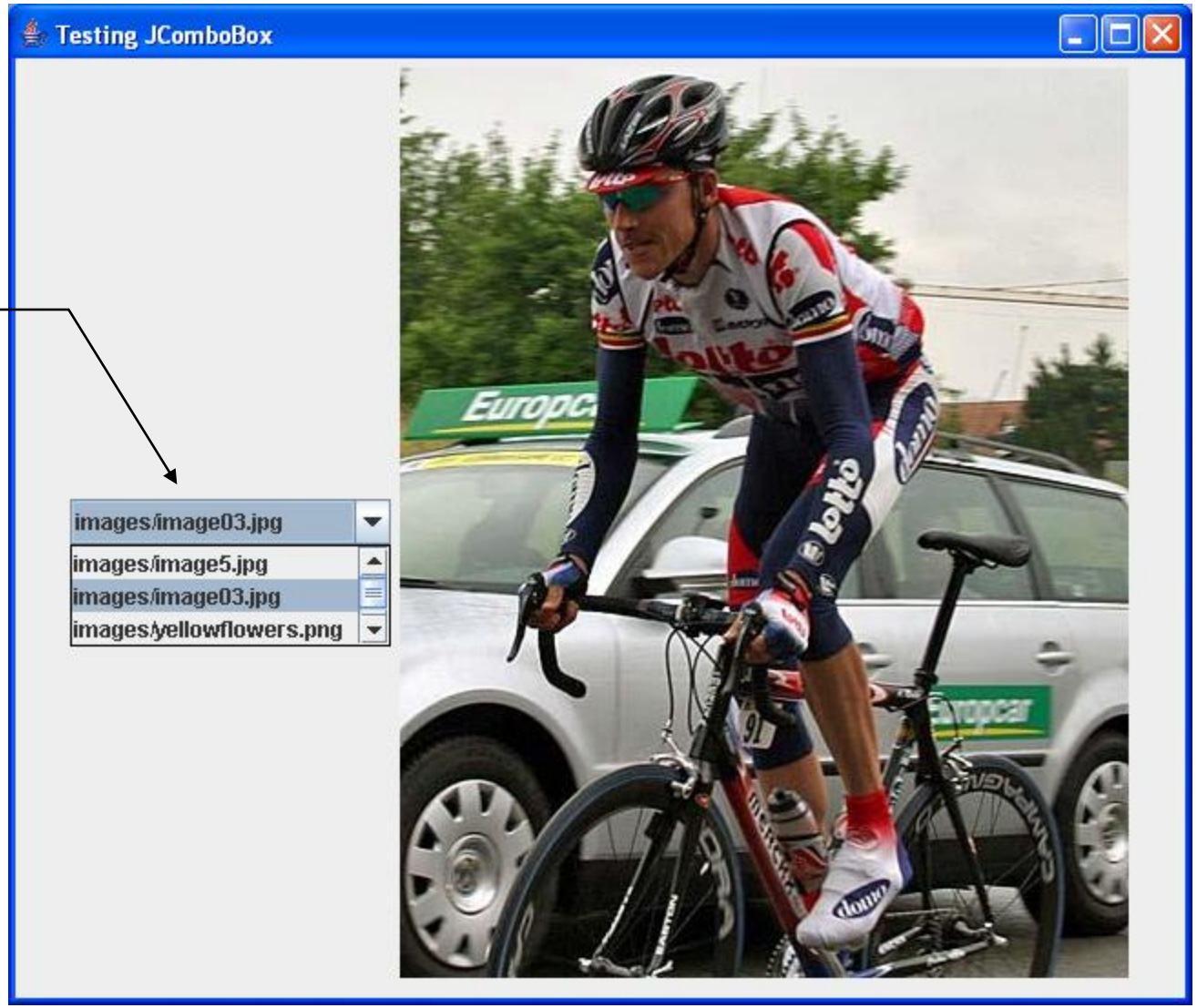

```
// ComboBoxFrame constructor adds JComboBox to JFrame
public ComboBoxFrame()
{
   super( "Testing JComboBox" );
   setLayout( new FlowLayout() ); // set frame layout
   imagesJComboBox = new JComboBox( names ); // set up JComboBox
   imagesJComboBox.setMaximumRowCount( 3 ); // display three rows

   imagesJComboBox.addItemListener(
      new ItemListener() // anonymous inner class
      {
         // handle JComboBox event
         public void itemStateChanged( ItemEvent event )
         {
            // determine whether check box 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
} // end class ComboBoxFrame
```

Anonymous inner class.

A special form of an inner class that is declared without a name and typically appears inside a method declaration. As with other inner classes an anonymous inner class can access its top-level class's members. However, an anonymous inner class has limited access to variables of the method in which the anonymous inner class is declared.


Testing JComboBox
images/image2.jpg
MERCKX
EWRUXX
MERCKX
MERCKX
CAMPAGNOLO

Drop-down portion of the combobox is active when the user moves the mouse over the box.



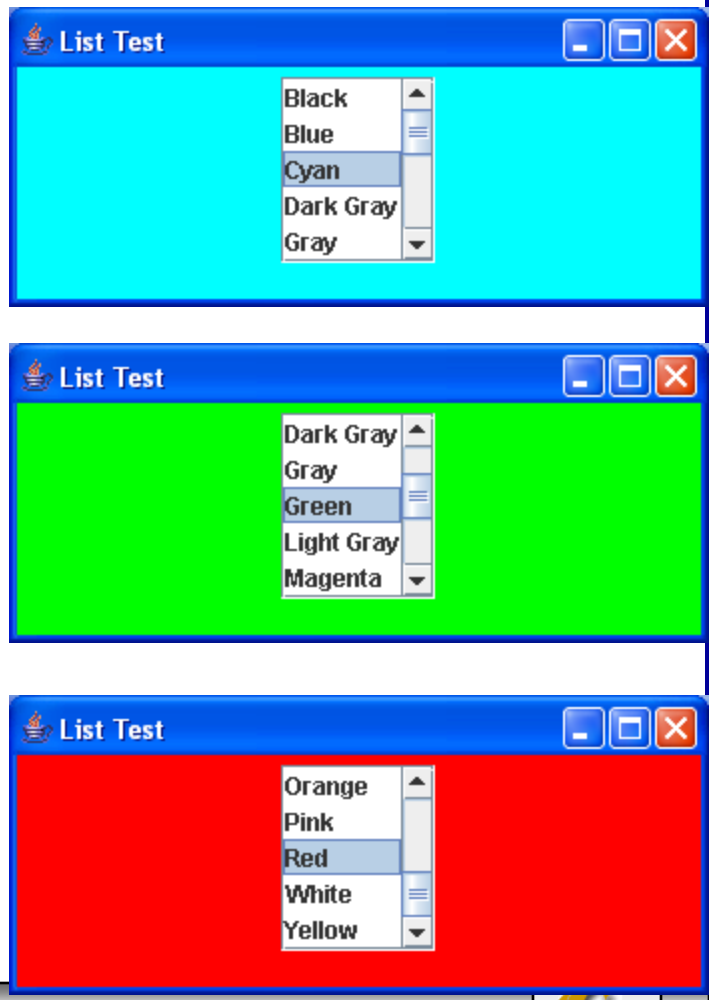
JList

- A list displays a series of items from which the user may select one or more items.
- Java's `JList` class supports single-selection lists (which allow only one item to be selected) and multiple-selection lists (which allow any number of items to be selected).
- The following example illustrates a simple use of a JList to select the background color for a dialog box. Again, the driver class is not listed but can be found on the code page of the course website.
- The example beginning on page 63 illustrates a multiple selection list.

GUI illustrating JList

```
// Example selecting colors from a JList.
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 final String colorNames[] = { "Black", "Blue", "Cyan",
      "Dark Gray", "Gray", "Green", "Light Gray", "Magenta",
      "Orange", "Pink", "Red", "White", "Yellow" };
   private 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
```



GUI illustrating Multiple-selection lists

```
// Multiple-selection lists: 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 toppingJList; // list to hold toppings
   private JList copyJList; // list to copy toppings into
   private JButton copyJButton; // button to copy selected toppings
   private final String toppingNames[] = { "chocolate syrup", "peanuts", "colored sprinkles",
      "chocolate sprinkles", "pineapple syrup", "strawberries", "caramel", "pecans", "more ice cream",
"whipped cream", "gummi bears", "chocolate hard shell", "raisins" };

   // MultipleSelectionFrame constructor
   public MultipleSelectionFrame()
   {
      super( "Multiple Selection Lists - Favorite Ice Cream Toppings" );
      setLayout( new FlowLayout() ); // set frame layout

      toppingJList = new JList( toppingNames ); // holds all the toppings
      toppingJList.setVisibleRowCount( 5 ); // show five rows
```


```
      toppingJList.setSelectionMode(
      ListSelectionModel.MULTIPLE_INTERVAL_SELECTION );
      toppingJList.setToolTipText("Hold CONTROL key down to select     multiple items");
    add( new JScrollPane( toppingJList ) ); // add list with scrollpane

    copyJButton = new JButton( "Favorites >>>" ); // 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( toppingJList.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
```

A single interval selection list allows selecting a contiguous range of items. To do this, click on the first item, then press and hold the SHIFT key while clicking on the last item in the range.

A multiple interval selection list allows contiguous selection as in the single interval list and miscellaneous items to be selected by pressing and holding the CONTROL key.

Initial GUI

GUI after user's selections