# COP 4610L: Applications in the Enterprise
# Spring 2005

## GUI Components: Part 2

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CSB 242, 823-2790
http://www.cs.ucf.edu/courses/cop4610L/spr2005

School of Computer Science
University of Central Florida

## Summary of Event-Handling Mechanism

- As we illustrated in the previous set of notes, there are three parts to the event-handling mechanism – the event source, the event object, and the event listener.

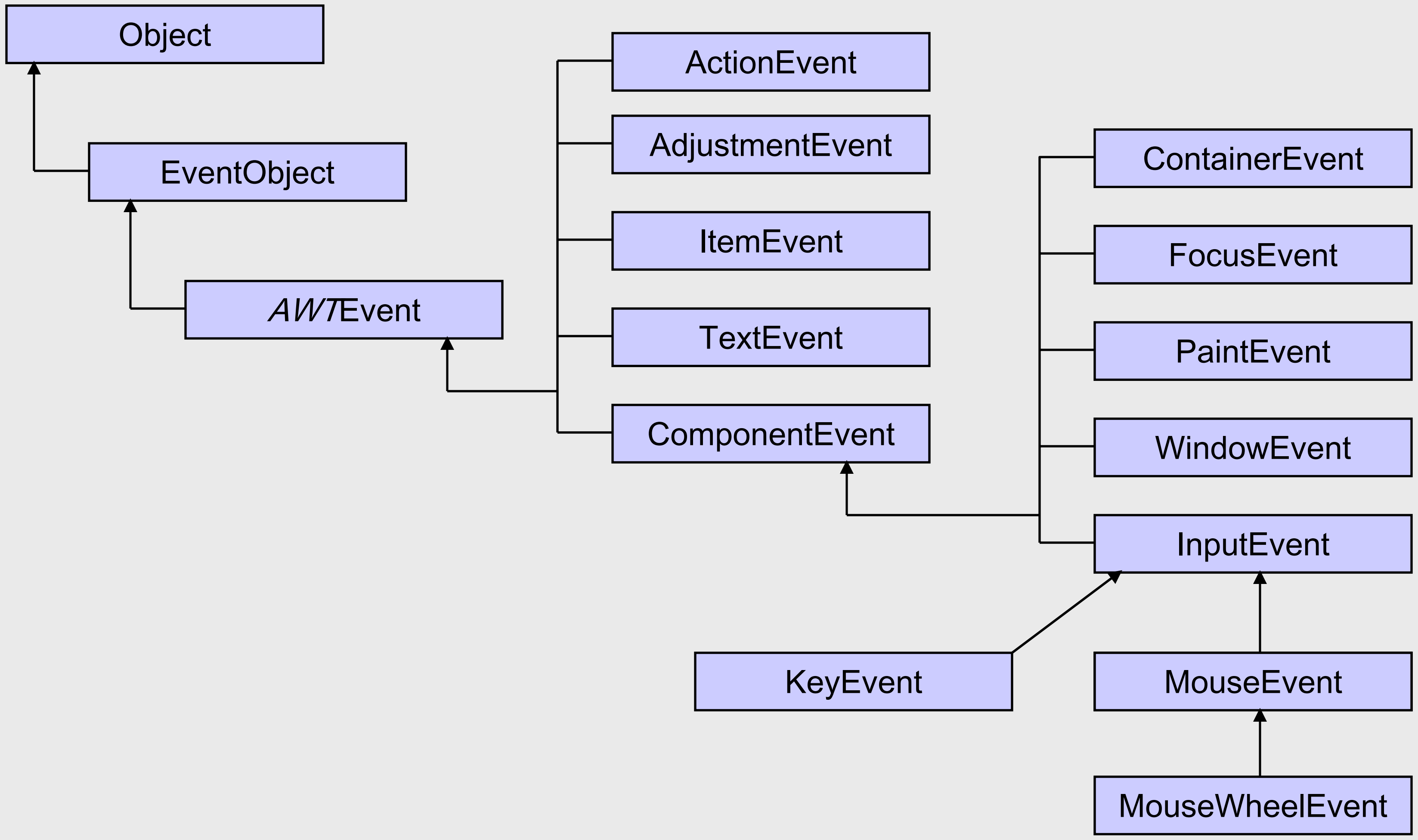
1. The event source is the particular GUI component with which the user interacts.

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.

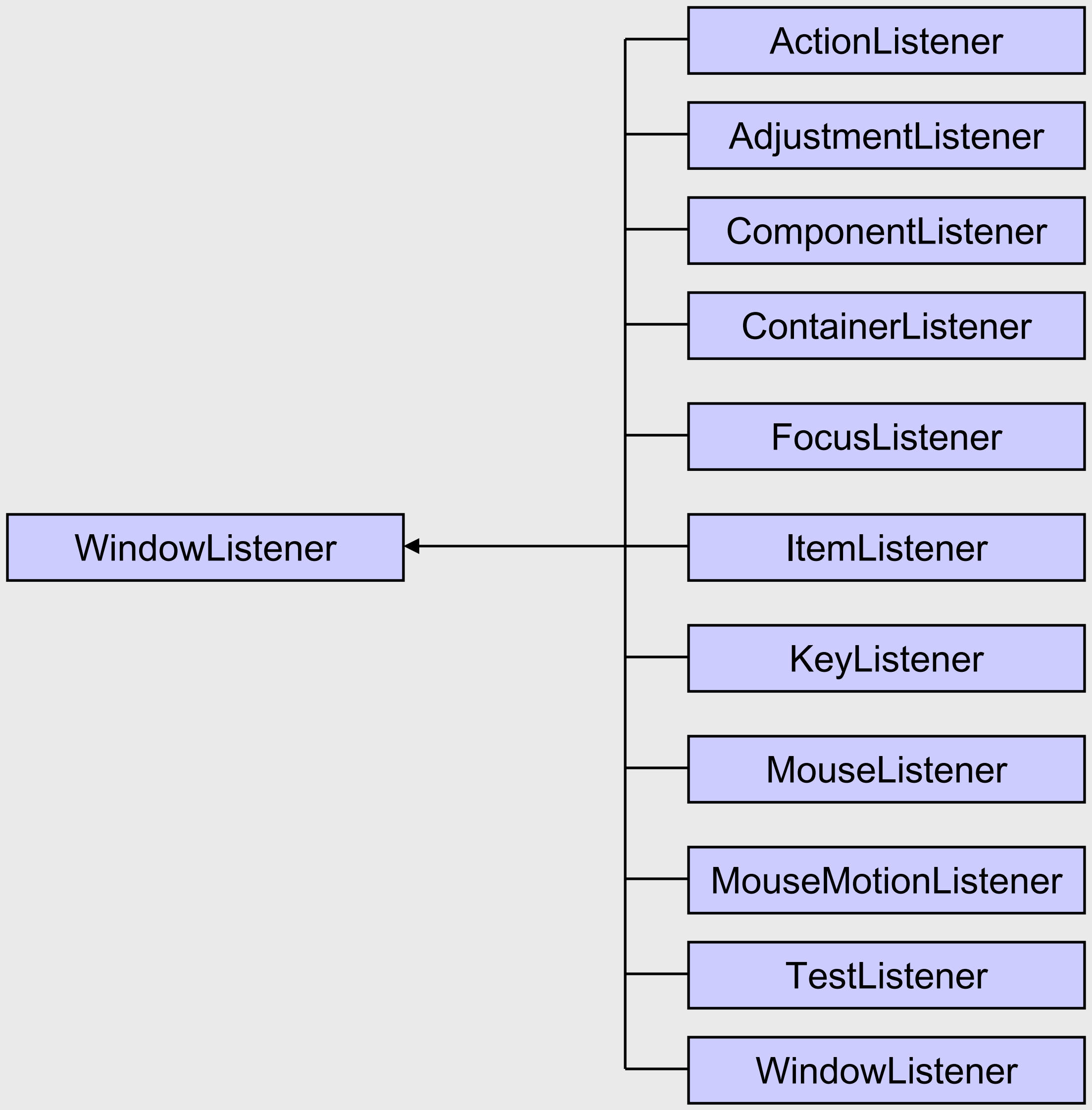
3. The event listener is an object that is notified by the event source when an event occurs; in effect, it "listens" for an event and one of its methods executes in response to the event. A method of the event listener receives an event object when the event listener is notified of the event.

## 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.



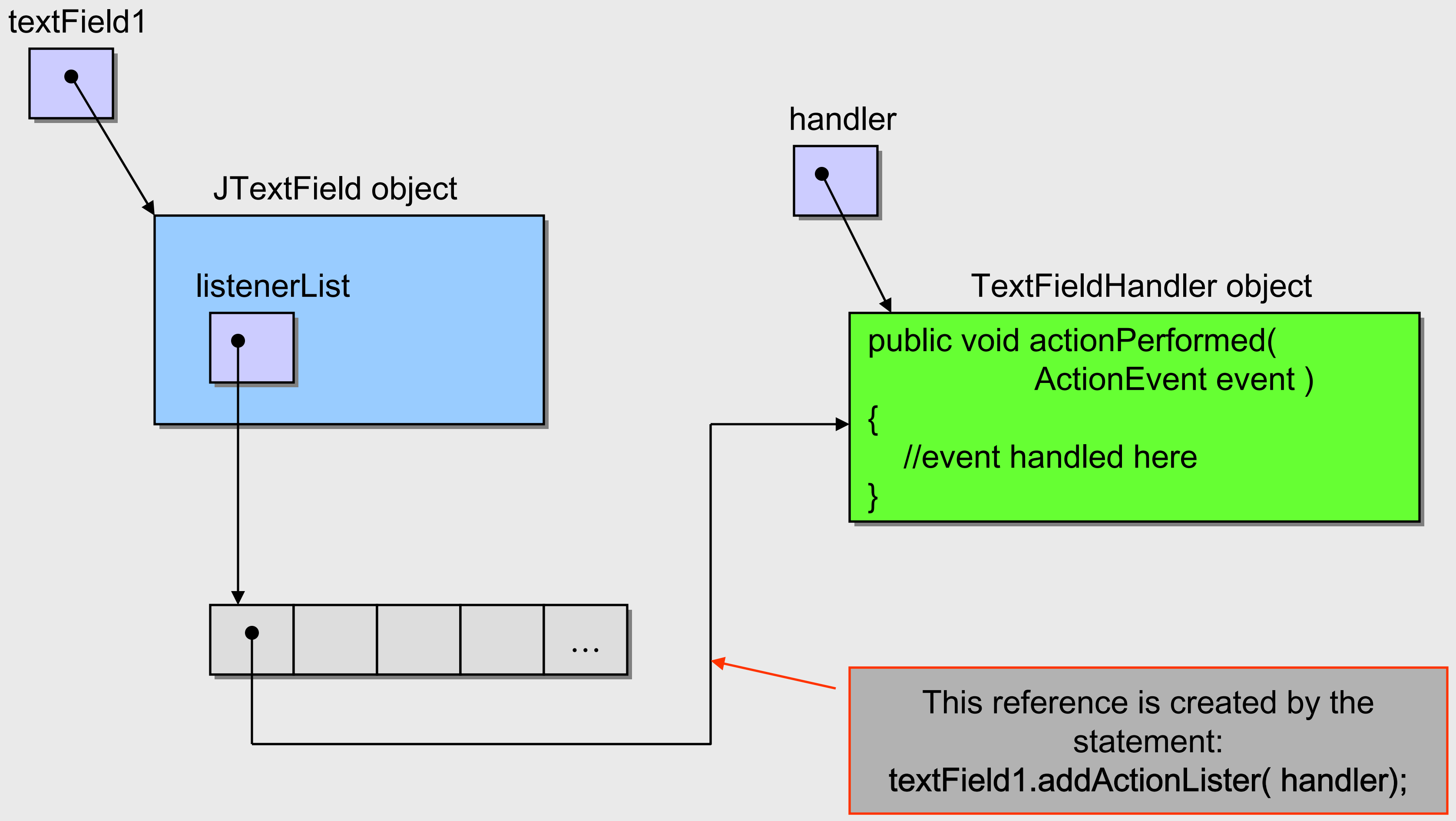
Some event classes of package `java.awt.event`



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 in part 1 of the notes (`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.



```
public void actionPerformed(
                 ActionEvent event )
{
   //event handled here
}
```

This reference is created by the statement:
textField1.addActionLister( handler);

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
```

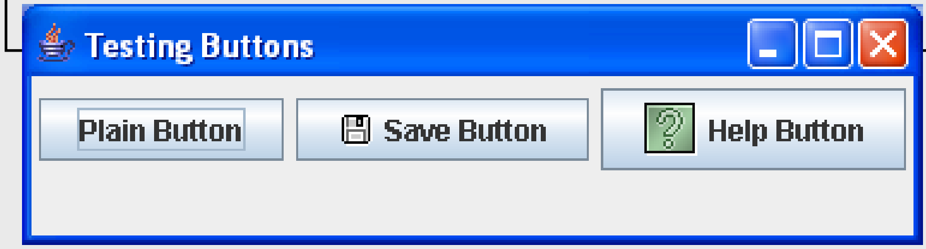
```
      Icon icon1 = new ImageIcon( getClass().getResource( "images/save16.gif" ) );
      Icon icon2 = new ImageIcon( getClass().getResource( "images/help24.gif" ) );
      saveJButton = new JButton( "Save Button", icon1 ); // set image
      helpJButton = new JButton("Help Button", icon2); //set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

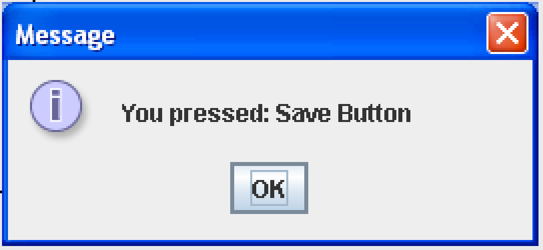
See page 15.

```
// 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

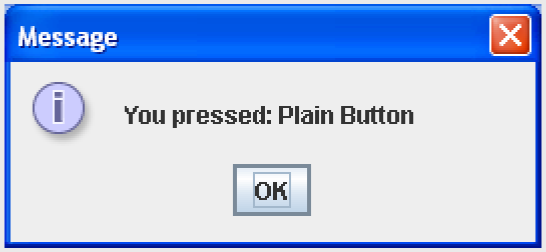


Initial GUI



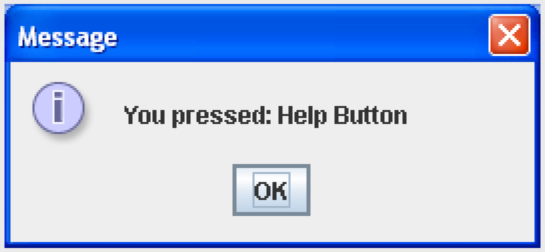
You pressed: Save Button

GUI after user clicks the Save Button



You pressed: Plain Button

GUI after user clicks the Plain Button



You pressed: Help 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

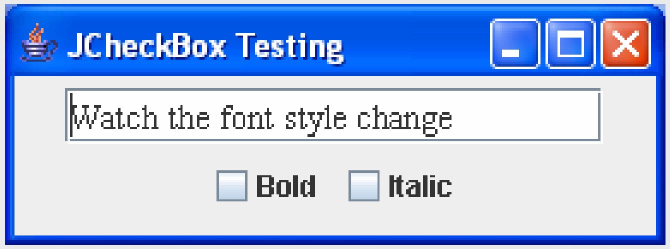
- The Swing GUI components contain three types of state buttons – JToggleButton, JCheckBox and JRadioButton that have on/off or true/false values.
- Classes JCheckBox and JRadioButton are subclasses of JToggleButton.
- A JRadioButton is different from a JCheckBox in that normally several JRadioButtons 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.

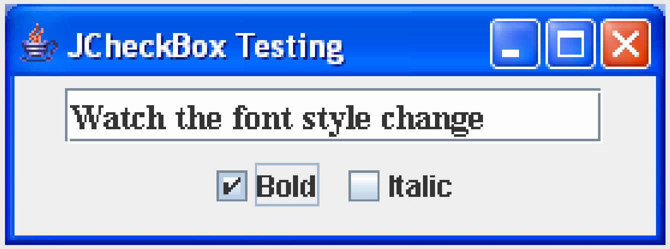
GUI illustrating JCheckBox

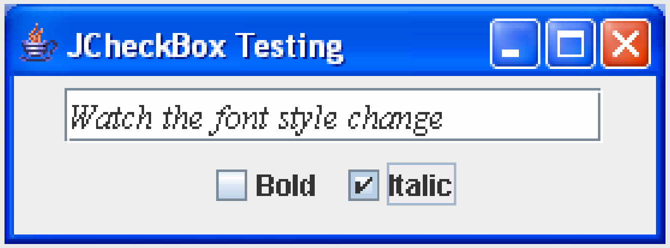
```
// Playing with JCheckBox buttons.
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JCheckBox;

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;
```

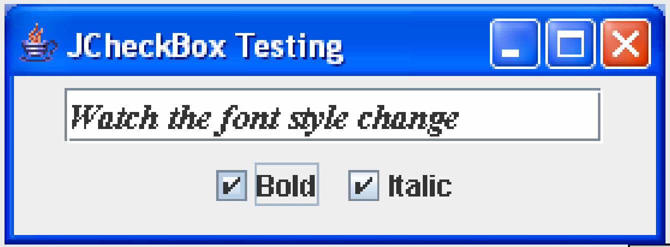
```
// set text field font
         textField.setFont(
            new Font( "Serif", valBold + valItalic, 14 ) );
      } // end method itemStateChanged
   } // end private inner class CheckBoxHandler
} // end class CheckBoxFrame
```

```
// 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

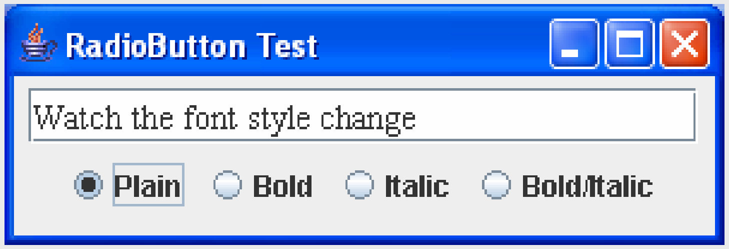
- Radio buttons (declared with class JRadioButton) are similar to checkboxes in that they have two states – selected or deselected. However, radio buttons normally appear as a group in which only one button can be selected at a time.
- Selecting a different radio button forces all others to be deselected.
- 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" );
```

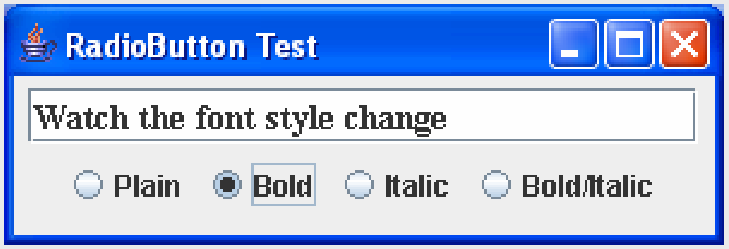
```
setLayout( new FlowLayout() ); // set frame layout
textField = new JTextField( "Watch the font style change", 25 );
add( textField ); // add textField to JFrame
// create radio buttons
plainJRadioButton = new JRadioButton( "Plain", true );
boldJRadioButton = new JRadioButton( "Bold", false );
italicJRadioButton = 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
```


RadioButton Test

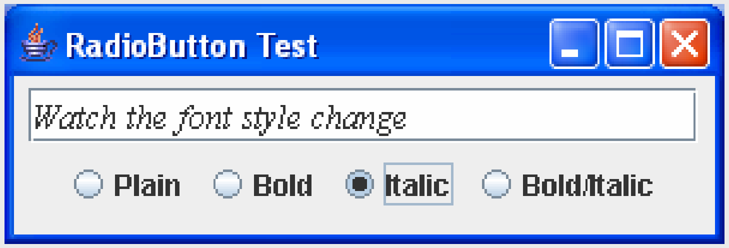
Watch the font style change

[✓ Plain] Bold Italic Bold/Italic


RadioButton Test

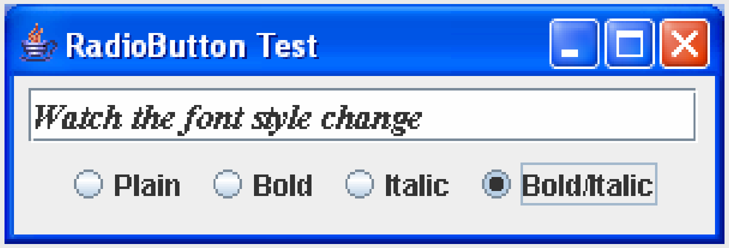
**Watch the font style change**

Plain [✓ Bold] Italic Bold/Italic


RadioButton Test

*Watch the font style change*

Plain Bold [✓ Italic] Bold/Italic


RadioButton Test

***Watch the font style change***

Plain Bold Italic [✓ Bold/Italic]

# 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 `ItemEvent`s 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.

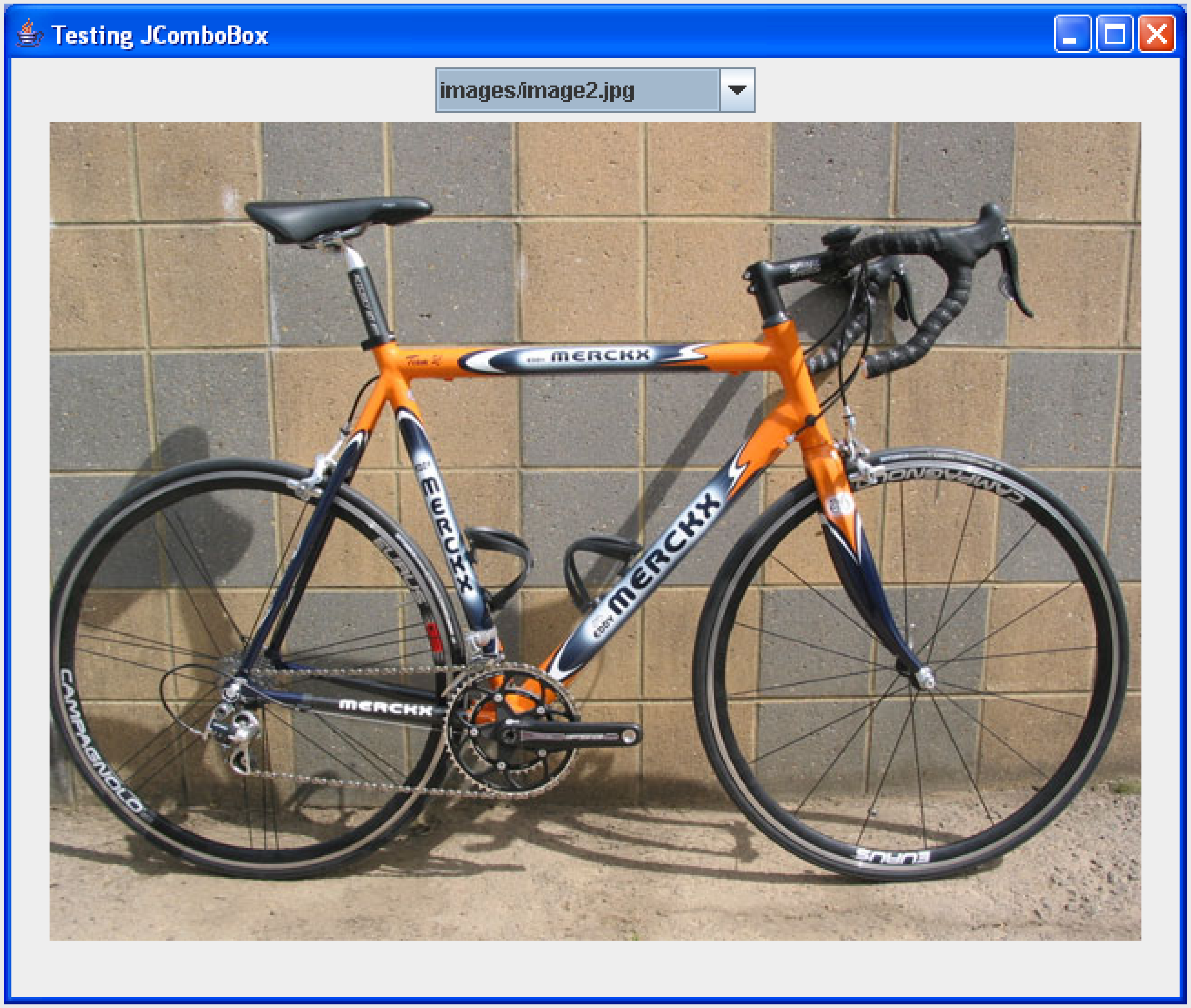
## GUI illustrating JComboBox

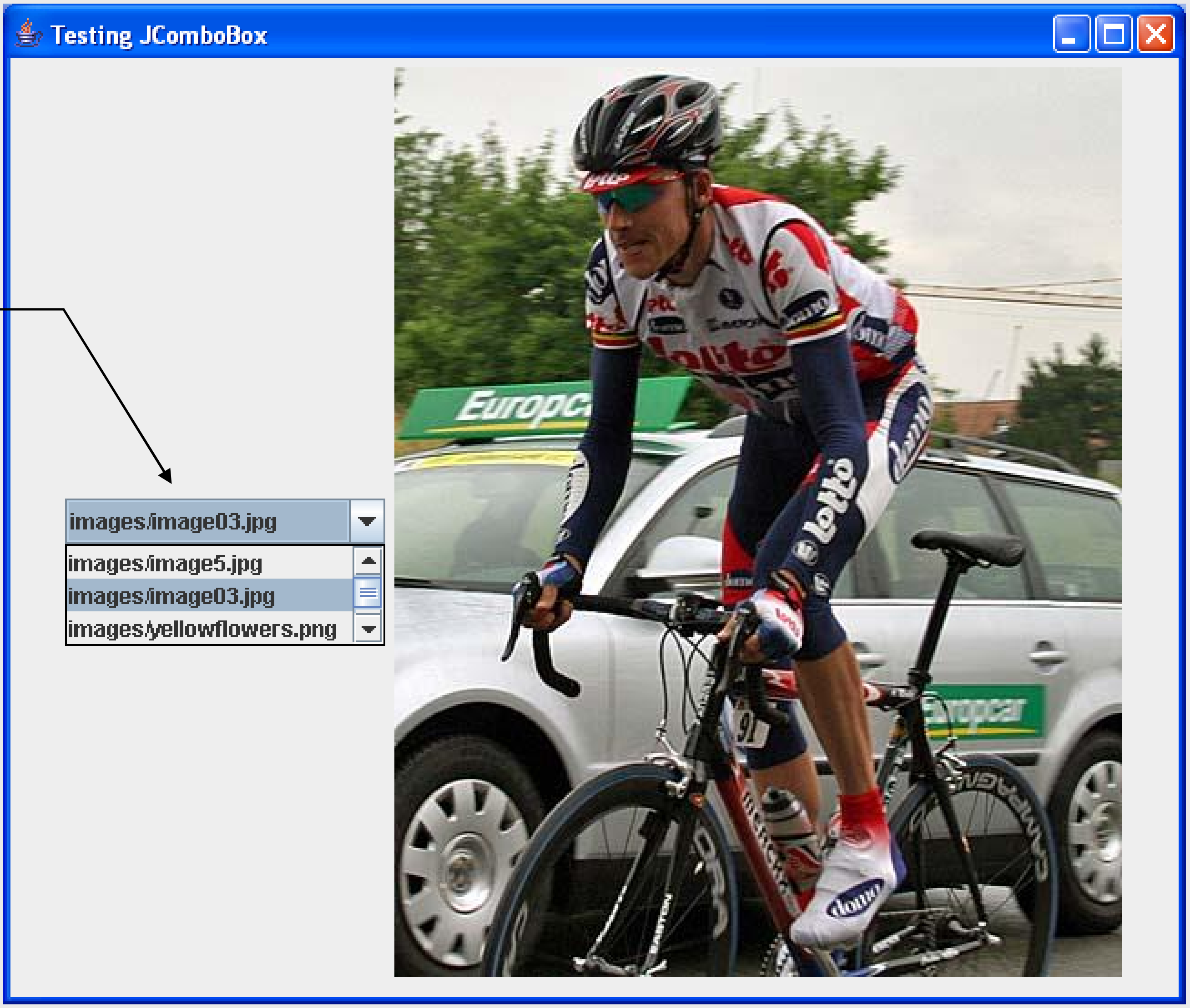
```
// Illustrating a JComboBox to select an image to display.
import java.awt.FlowLayout;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JComboBox;
import javax.swing.Icon;
import javax.swing.ImageIcon;

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



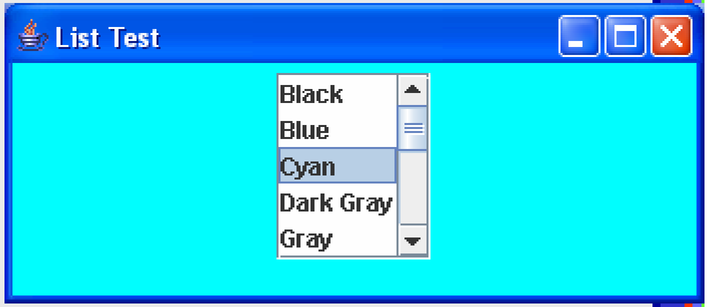
Drop-down portion of the combobox is active when the user moves the mouse over the box.

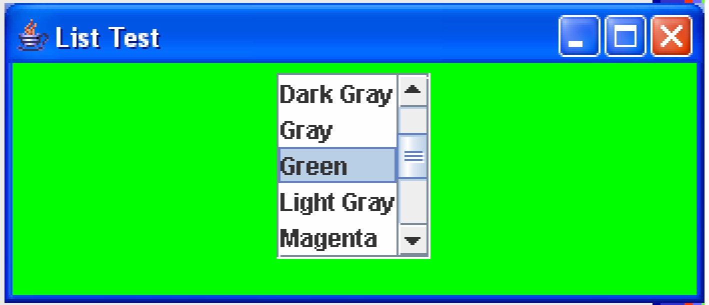
## JList

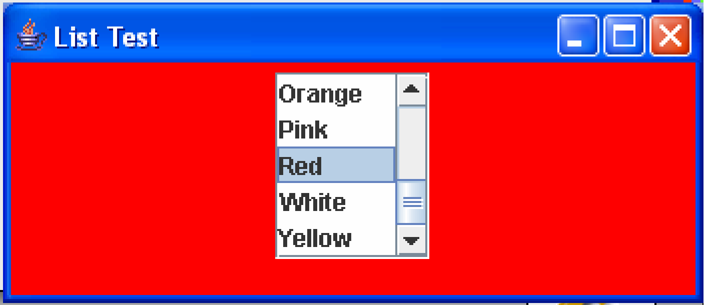
- A list displays a series of items from which the user may select one or more items.
- Java's `JList` class supports single-selection lists (which allow only one item to be selected) and multiple-selection lists (which allow any number of items to be selected).
- The following example illustrates a 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 32 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
```





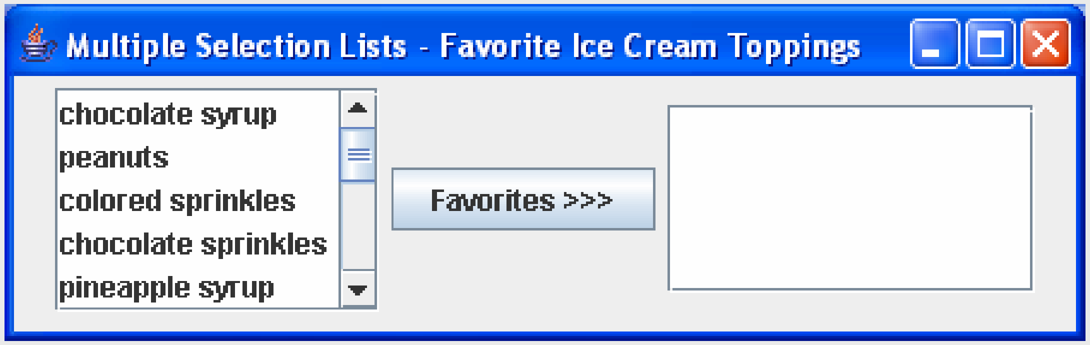


GUI illustrating Multiple-selection lists

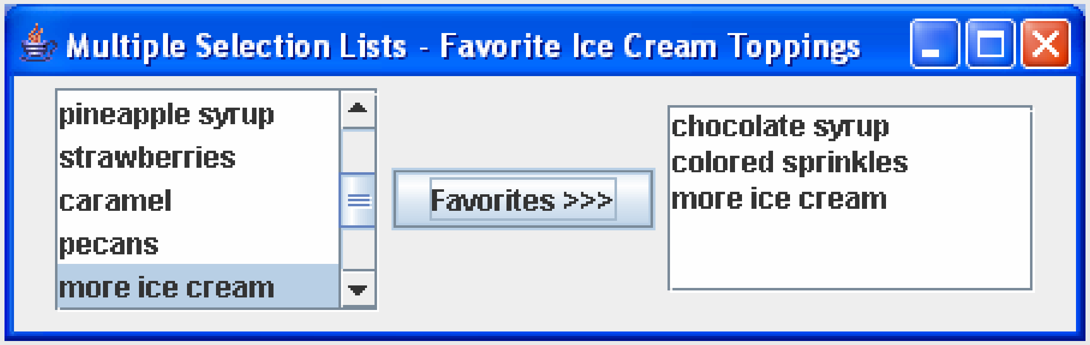
```
// Multiple-selection lists: Copying items from one List to another.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JButton;
import javax.swing.JScrollPane;
import javax.swing.ListSelectionModel;

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