CSC 306: Human-Computer Interaction and User Interface Design

Java GUI Programming I

Denis L. Nkweteyim

Outline

- Introduction
- JOptionPane
- Overview of Swing Components
- Text Fields and Introduction to Event Handling with Nested Classes
- JButton
- Buttons that Maintain State
- JComboBox and Using Anonymous Inner Class for Event Handling
- JList
- Multiple Selection Lists
- Mouse Event Handling
- Adapter Classes
- Key-Event Handling
- Layouts

Introduction

- We focus on building desktop interfaces (using Java)
- Similar ideas hold for other applications
  - Web interfaces (using HTML forms and related technologies)
  - Interfaces for mobile applications
  - etc.
- We start by hand coding our GUI components
  - So that we understand the mechanics well
- Later we use NetBeans GUI Builder to visually place our widgets on screen
  - Other IDEs support similar GUI builder technologies to do visual programming

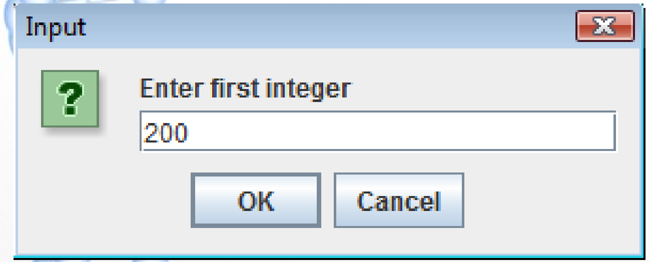
Outline

JOptionPane

- Java’s JOptionPane class (package javax.swing) provides prepackaged dialog boxes for both input and output
  - Dialogs are displayed by invoking static JOptionPane methods
- Example program makes use of three dialog boxes
  - Two to obtain integers from user
  - The third to display the sum of the integers

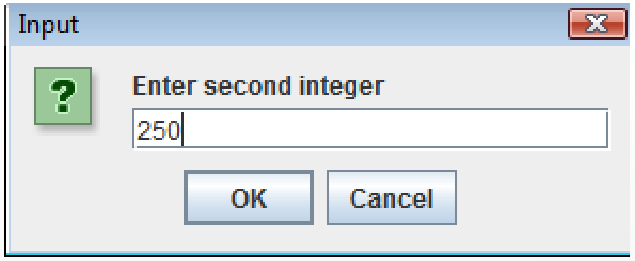
JOptionPane

```
package sampleapplications;
import javax.swing.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);
    }
}
```



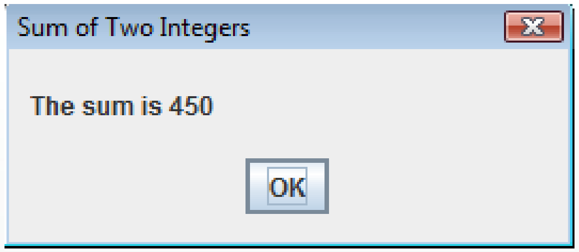
Enter first integer

200



Enter second integer

250



The sum is 450

JOptionPane

- Program notes
  - Unlike Scanner, which can be used to input data of several types, the input dialog (like most GUI components) can only input Strings
  - If the user types data that is not a valid integer, an exception will be thrown
  - If the user clicks the Cancel button, ShowInputDialog returns null
  - Converting Strings to int values
    - Use the Integer class’s static method parseInt

JOptionPane

- Program notes
  - JOptionPane Dialog
    - First parameter: component on which the dialog box is displayed; null means centre of current screen
    - Constants
      - PLAIN_MESSAGE: No icon displayed with message
      - Try other constants to see different icons displayed, e.g.,
        - INFORMATION_MESSAGE
        - WARNING_MESSAGE
        - QUESTION_MESSAGE (the default icon we saw for input dialog)

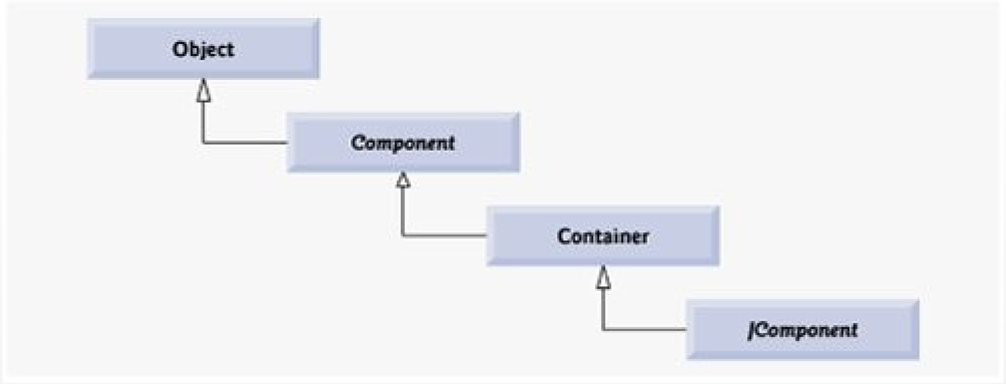
Outline

Overview of Swing Components

- Most GUI applications require more elaborate user interfaces than dialog boxes
  - Swing GUI components (package javax.swing) commonly used
    - Most Swing components are part of Java Foundation Classes (JFC) – Java's libraries for cross-platform GUI development
  - Before Swing, there was AWT (Abstract Window Toolkit)
    - Still supported in package java.awt
- Swing vs AWT
  - Application with AWT GUI executes differently on different platforms (Windows, Apple MAC OS, etc.)
    - i.e., look-and-feel is the same as for components of other applications in that platform, which is different across platforms
  - With Swing, we can make the look-and-feel the same for all platforms

Overview of Swing Components

- Swing components inheritance hierarchy
  - Class Component (package java.awt)
    - Declares many of the attributes and behaviors common to the GUI components in packages java.awt and javax.swing
    - Most GUI components extend class Component directly or indirectly
  - Class Container (package java.awt)
    - Components are attached to Containers (such as windows)
    - Any object that is a Container can be used to organize other Components in a GUI
    - Because a Container is a Component , you can attach Containers to other Containers to help organize a GUI
  - Class JComponent (package javax.swing)
    - JComponent is the superclass of all lightweight Swing components (lightweight because their operation does not depend on the underlying platform)
    - Because JComponent is a subclass of Container, all lightweight Swing components are also Containers



Outline

- Introduction
- JOptionPane
- Overview of Swing Components
- Text Fields and Introduction to Event Handling with Nested Classes
- JButton
- Buttons that Maintain State
- JComboBox and Using Anonymous Inner Class for Event Handling
- JList
- Multiple Selection Lists
- Mouse Event Handling
- Adapter Classes
- Key-Event Handling
- Layouts

Displaying Text and Images in a Window

- Objectives
  - Introducing a framework for building GUI applications
    - Sample application will run in its own window
      - As with most windows, our window will be instantiated from class JFrame
      - JFrame provides basic window attributes and behaviours: title bar; maximize, minimize and close buttons

Displaying Text and Images in a Window

- Labels
  - Used to identify purpose of components on the GUI
  - Created from class JLabel, a subclass of JComponent
  - Label displays any of the following
    - A line of read-only text
    - An image
    - Both text and image

Displaying Text and Images in a Window

```
package hci;

import java.awt.FlowLayout;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingConstants;

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
```

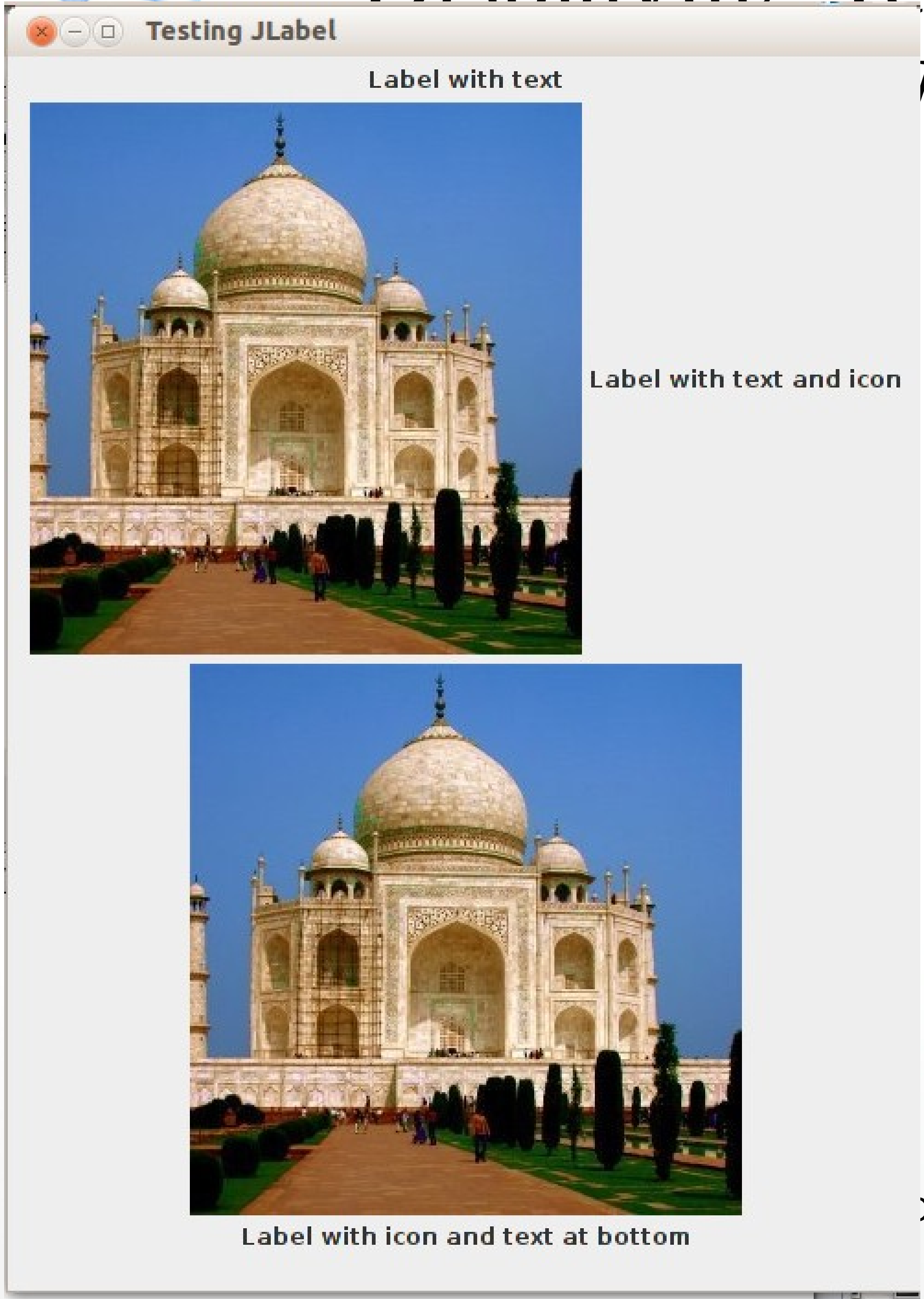
Displaying Text and Images in a Window

```
public class LabelFrame contd.
     // 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 tajLogo = new ImageIcon(getClass().
                 getResource("color-taj-filterpack.jpg"));
      label2 = new JLabel("Label with text and icon", tajLogo,
                 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(tajLogo); // add icon to JLabel
      label3.setHorizontalTextPosition(SwingConstants.CENTER);
      label3.setVerticalTextPosition(SwingConstants.BOTTOM);
      label3.setToolTipText("This is label3");
      add(label3); // add label3 to JFrame
   }
}
```

Displaying Text and Images in a Window

```
package hci;

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(500, 700); // set frame size
    labelFrame.setVisible(true); // display frame
  }
}
```

Displaying Text and Images in a Window


Testing JLabel

Label with text

Label with text and icon

Label with icon and text at bottom

Displaying Text and Images in a Window

- Program notes
  - Class LabelFrame is a subclass of JFrame
  - We use an instance of class LabelFrame to display a window containing three JLabels
  - The JLabels are instantiated in the constructor for LabelFrame
  - LabelFrame’s constructor also invokes its superclass’s (i.e., Jframe’s) constructor with the argument "Testing JLabel"
    - This text appears in the window's title bar

Displaying Text and Images in a Window

- Program notes
  - Specifying the layout
    - Java specifies several layout managers (used to position GUI components in a container)
    - A tool like Netbeans provides design tools that can be used to exactly specify the position of components in containers, and then generate the corresponding code
    - In this example, we do this manually using the FlowLayout manager
      - FlowLayout places components on a container from left to right, and then continue on the next line if there is not enough space

Displaying Text and Images in a Window

- Program notes
  - Notes on JLabels
    - Setting tool tip
    - Support for images (most components can display images)
      - An Icon is an object of any class that implements interface Icon
      - color-taj-filterpack.png is a graphic that will be loaded
    - When label2 is created, we use the constructor that includes a graphic to accompany the label text
      - The last constructor argument (SwingConstants.LEFT) indicates that the label's contents are left justified (i.e. text and icon appear on the left of the area available to display the label)
      - Other horizontal Swing constants: SwingConstants.CENTER, SwingConstants.RIGHT
      - And vertical SwingConstants: SwingConstants.TOP, SwingConstants.CENTER SwingConstants.BOTTOM
    - Note from Label3 that methods are available to set horizontal and vertical alignments of labels

Displaying Text and Images in a Window

- Program notes
  - Creating and displaying the LabelFrame window (in class LabelTest)
    - Default close operation for the window
    - By default, closing a window simply hides the window
    - When the user closes the LabelFrame window, we would like the application to terminate
    - Hence, default close operation set to constant argument JFrame.EXIT_ON_CLOSE
  - LabelFrame 's setSize method used to specify the width and height of the window
  - LabelFrame 's setVisible method with the argument true used to display the window on the screen
  - Try resizing the window to see how the FlowLayout changes the JLabel positions as the window width change

Outline

- Introduction
- JOptionPane
- Overview of Swing Components
- Text Fields and Introduction to Event Handling with Nested Classes
- JButton
- Buttons that Maintain State
- JComboBox and Using Anonymous Inner Class for Event Handling
- JList
- Multiple Selection Lists
- Mouse Event Handling
- Adapter Classes
- Key-Event Handling
- Layouts

Text Fields and Introduction to Event Handling with Nested Classes

- GUIs are event driven
  - When a user interacts with a GUI component, the interaction – also known as an event – drives the program to perform a task
  - Some common events (user interactions) that might 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

Text Fields and Introduction to Event Handling with Nested Classes

- In the following program
  - We consider JTextFields and JPasswordFields (package javax.swing)
    - Class JTextField extends class JTextComponent (package javax.swing.text)
    - Class JPasswordField extends JTextField and adds several methods that are specific to processing passwords
    - Each of these components is a single-line area in which the user can enter text via the keyboard
    - Applications can also display text in a JTextField
    - A JPasswordField shows that characters are being typed as the user enters them, but hides the actual characters with an echo character, assuming that they represent a password that should remain known only to the user
  - When the user types data into a JTextField or a JPasswordField, then presses Enter, an event occurs
  - We see how a program can perform a task in response to that event
  - The techniques shown here are applicable to all GUI components that generate events

Text Fields and Introduction to Event Handling with Nested Classes

- In the program
  - When the user types in one of the text fields, then presses Enter, the application displays a message dialog box containing the text the user typed
  - You can only type in the text field that is in focus
    - A component receives the focus when the user clicks the component
    - The text field with the focus is the one that generates an event when the user presses Enter

Text Fields and Introduction to Event Handling with Nested Classes

```
package sampleapplications;
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;  private JTextField textField2;
  private JTextField textField3;  private JPasswordField passwordField;
  public TextFieldFrame() {
     super("Testing JTextField and JPasswordField"); setLayout(new FlowLayout());
     // construct textfield with 10 columns
     textField1 = new JTextField(10); add(textField1);
     // construct textfield with default text
     textField2 = new JTextField("Enter text here"); add(textField2);
     // construct textfield with default text and 21 columns
     textField3 = new JTextField("Uneditable text field", 21);
     textField3.setEditable(false); add(textField3);
     // construct passwordfield with default text
     passwordField = new JPasswordField("Hidden text"); add(passwordField);
     // register event handlers
     TextFieldHandler handler = new TextFieldHandler();
     textField1.addActionListener(handler); textField2.addActionListener(handler);
     textField3.addActionListener(handler); passwordField.addActionListener(handler);
 }
```

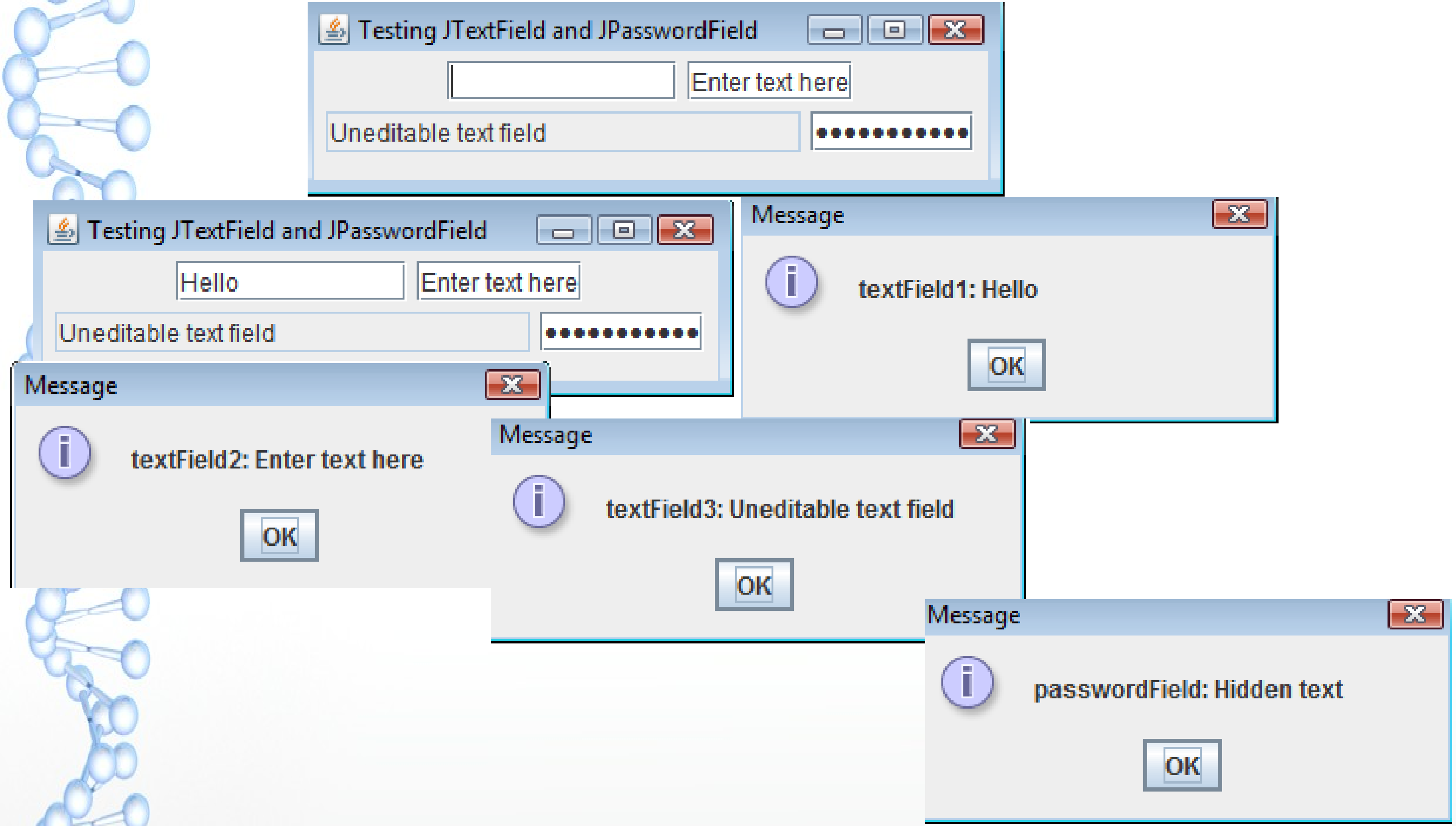
Text Fields and Introduction to Event Handling with Nested Classes

```
// 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
            if (event.getSource() == textField1) {
                string = String.format("textField1: %s", event.getActionCommand());
            }
            else if (event.getSource() == textField2) {
                string = String.format("textField2: %s", event.getActionCommand());
            }
            else if (event.getSource() == textField3) {
                string = String.format("textField3: %s", event.getActionCommand());
            }
            else if (event.getSource() == passwordField) {
                string = String.format("passwordField: %s",
                        new String(passwordField.getPassword()));
            }
            JOptionPane.showMessageDialog(null, string);// display JTextField content
        } // end method actionPerformed
    } // end private inner class TextFieldHandler
} // end class TextFieldFrame
```

Text Fields and Introduction to Event Handling with Nested Classes

```
package sampleapplications;
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(375, 100); // set frame size
      textFieldFrame.setVisible(true); // display frame
   }
}
```

Text Fields and Introduction to Event Handling with Nested Classes


Testing JTextField and JPasswordField

Enter text here

Uneditable text field

••••••••••••

Testing JTextField and JPasswordField

Hello

Enter text here

Uneditable text field

••••••••••••

Message

textField1: Hello

OK

Message

textField2: Enter text here

OK

Message

textField3: Uneditable text field

OK

Message

passwordField: Hidden text

OK

Text Fields and Introduction to Event Handling with Nested Classes

- Program notes
  - The code to create the text fields and password fields is not complicated
  - Steps required to set up event handling
    - Create a class that represents the event handler
    - Implement an appropriate interface, known as an event-listener interface, in the class from Step 1
    - 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

Text Fields and Introduction to Event Handling with Nested Classes

- Program notes
  - Using a nested class to implement an event handler
    - All the classes we have seen up to this point are top-level classes
      - Top-level classes are classes that are not declared inside another class
    - Java allows you to declare classes inside other classes
      - These are called nested classes
      - Nested classes can be static or non-static
      - Non-static nested classes are called inner classes and are frequently used for event handling
    - Note
      - An inner class is allowed to directly access its top-level class's variables and methods, even if they are private

Text Fields and Introduction to Event Handling with Nested Classes

- Program notes
  - Event handling is performed by an object of the private inner class TextFieldHandler
    - Private because it will be used only to create event handlers for the text fields in top-level class TextFieldFrame
      - But could also have been public or protected
  - GUI components can generate a variety of events in response to user interactions
    - Each event is represented by a class and can be processed only by the appropriate type of event handler
      - When the user presses Enter in a JTextField or JPasswordField, the GUI component generates an ActionEvent (package java.awt.event)
      - Such an event is processed by an object that implements the interface ActionListener (package java.awt.event)
  - To prepare to handle the events in this example, inner class TextFieldHandler implements interface ActionListener and declares the only method in that interface, viz. actionPerformed
    - This method specifies the tasks to perform when an ActionEvent occurs

Text Fields and Introduction to Event Handling with Nested Classes

- Program notes
  - Registering the event handler for each text field
    - The statement: TextFieldHandler handler = new TextFieldHandler();
    - Creates a TextFieldHandler object and assigns it to variable handler
    - We want this object's actionPerformed method to be called when the user presses Enter in any of the text fields
    - So we register the object as the event handler for each of the text fields
      - That is the purpose of the 4 lines that follow the statement above that call the JTextFields' method addActionListener
  - The object handler can now listen to events – generated when the Enter key is pressed when any of the text fields is in focus

Text Fields and Introduction to Event Handling with Nested Classes

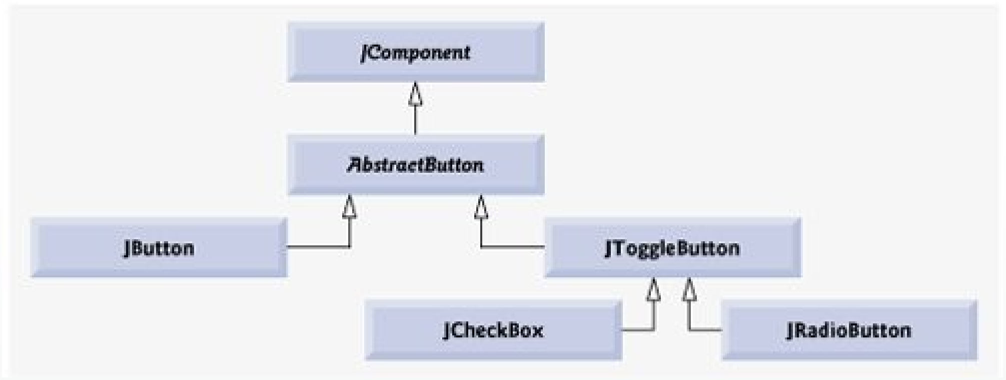
- Program notes
  - TextFieldHandler’s actionPerformed method
    - Since we are using one event-handling object's actionPerformed method to handle the events generated by four text fields, we must determine which text field generated the event each time actionPerformed is called
    - The GUI component with which the user interacts is the event source
    - When the user presses Enter while one of the text fields has focus, the system creates a unique ActionEvent object that contains information about the event that just occurred, such as the event source and the text in the text field
    - The system calls the event listener’s actionPerformed method and passes this ActionEvent object to it
    - We store the text field’s content in the string variable that is declared in the method for display later
    - ActionEvent method getSource returns a reference to the event source
    - ActionEvent method getActionCommand obtains the text the user typed in the text field that generated the event
      - If the user interacts with the JPasswordField JPasswordField method getPassword is used to obtain the password and create the String to display

Outline

- Introduction
- JOptionPane
- Overview of Swing Components
- Text Fields and Introduction to Event Handling with Nested Classes
- JButton
- Buttons that Maintain State
- JComboBox and Using Anonymous Inner Class for Event Handling
- JList
- Multiple Selection Lists
- Mouse Event Handling
- Adapter Classes
- Key-Event Handling
- Layouts

JButton

- A user clicks a button to trigger a specific action
- Several types of buttons
  - Command buttons, toggle buttons (check boxes and radio buttons)
  - Class AbstractButton (package javax.swing) declares the common features of Swing buttons



JButton

- Command button
  - Generates an ActionEvent when the user clicks the button
  - Created from class JButton
  - The text on the face of a JButton is called a button label
-
- Example program
  - 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

JButton

```
package sampleapplications;

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
```

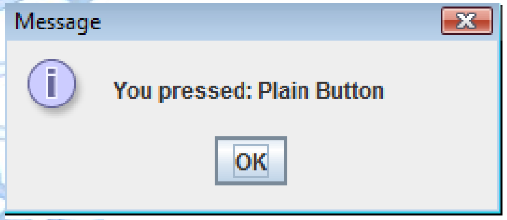
JButton

```
        Icon button1 = new ImageIcon(getClass().getResource("buttons1.png"));
        Icon button2 = new ImageIcon(getClass().getResource("buttons2.png"));
        fancyJButton = new JButton("Fancy Button", button1); // set image
        fancyJButton.setRolloverIcon(button2); // 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 {

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

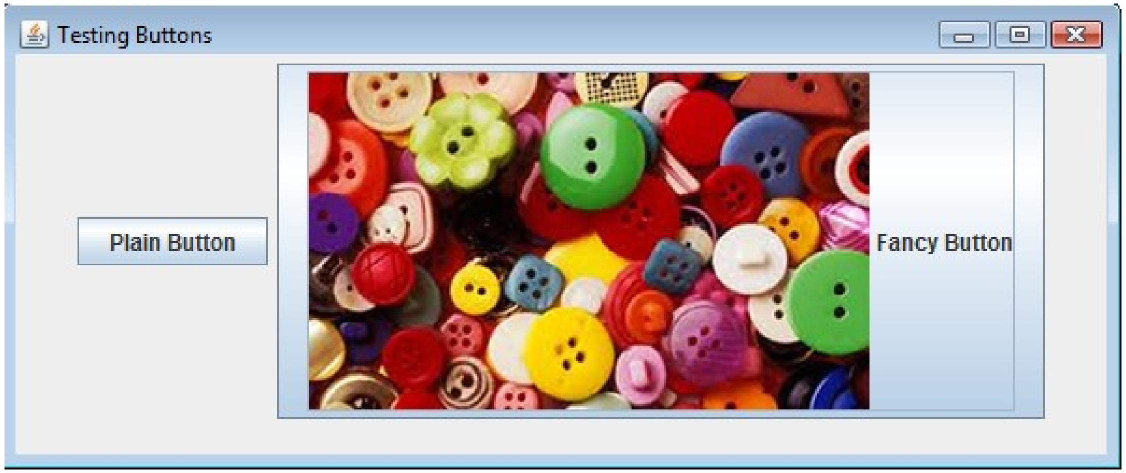
JButton

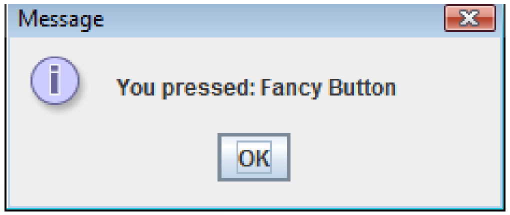
```
package sampleapplications;
import javax.swing.JFrame;

public class ButtonTest {
   public static void main(String args[]) {
      ButtonFrame buttonFrame = new ButtonFrame(); // create 
ButtonFrame
      buttonFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      buttonFrame.setSize(600, 250); // set frame size
      buttonFrame.setVisible(true); // display frame
   } // end main
}
```

JButton

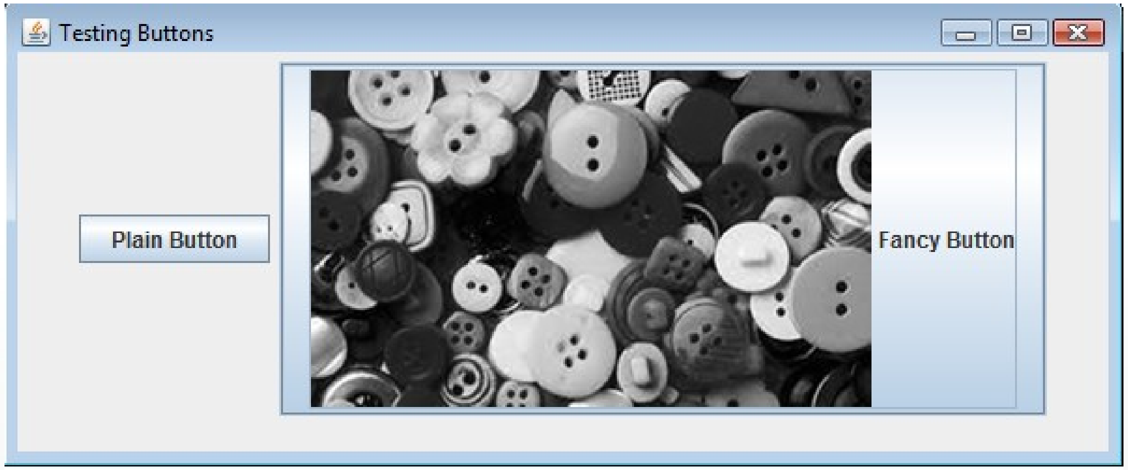


You pressed: Plain Button





You pressed: Fancy Button



JButton

- Program notes
  - The code should be understandable
  - Notice that JButtons support image rollover (using method setRolloverIcon)
  - Accessing the **this** reference in an object of a top-level class from an inner class
    - Notice that when a button is clicked, the message dialog appears centred over the application’s window
    - This is because the call to JOptionPane showMessageDialog uses ButtonFrame.**this** and not null as the first argument
    - When **this** argument is not null, it represents the parent GUI of the message dialog (in this case the ButtonFrame application window) and causes the dialog to be centred over that component

Outline

Buttons that Maintain State

- The Swing GUI components contain three types of state buttons
  - JToggleButton, JCheckBox and JRadioButton
  - These buttons have on/off or true/false values
  - Classes JCheckBox and JRadioButton are subclasses of JToggleButton
  - A JRadioButton is different from a JCheckBox in that usually (though not necessarily) several JRadioButtons are grouped together, and are mutually exclusive
    - Only one in the group can be selected at any time

Buttons that Maintain State

- JCheckBox example
  - Two JCheckBox objects used to select the desired font style of the text displayed in a JTextField
  - When selected, one applies a bold style and the other an italic style
  - If both are selected, the style of the font is bold and italic
  - When the application initially executes, neither JCheckBox is checked (i.e., they are both false), so the font is plain

Buttons that Maintain State - JCheckBox

```
package sampleapplications;
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
        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
```

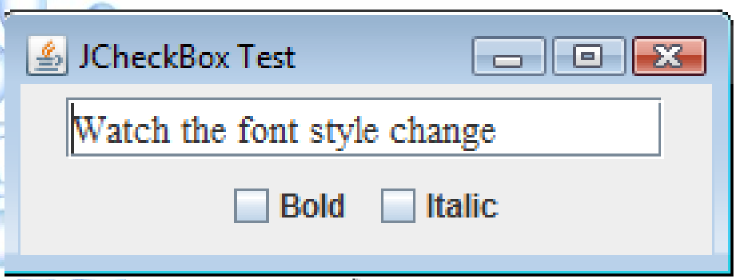
Buttons that Maintain State - JCheckBox

```
      CheckBoxHandler handler = new CheckBoxHandler();
      boldJCheckBox.addItemListener(handler);
      italicJCheckBox.addItemListener(handler);
  }

  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
}
```

Buttons that Maintain State - JCheckBox

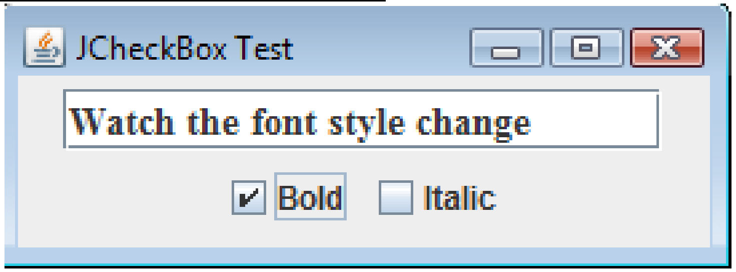
```
package sampleapplications;
import javax.swing.JFrame;
public class CheckBoxTest {
    public static void main(String args[]) {
        CheckBoxFrame checkBoxFrame = new CheckBoxFrame();
        checkBoxFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        checkBoxFrame.setSize(275, 100);
        checkBoxFrame.setVisible(true);
    } // end main
}
```

Buttons that Maintain State - JCheckBox


JCheckBox Test

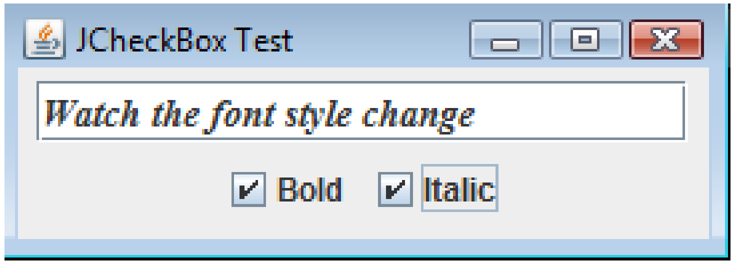
Watch the font style change

☐ Bold ☐ Italic


JCheckBox Test

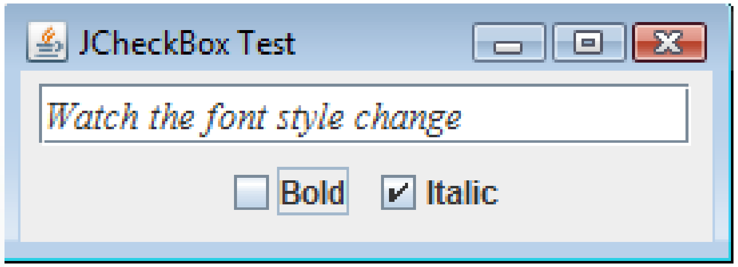
Watch the font style change

☑ Bold ☐ Italic


JCheckBox Test

Watch the font style change

☑ Bold ☑ Italic


JCheckBox Test

Watch the font style change

☐ Bold ☑ Italic

Buttons that Maintain State - JCheckBox

- Program notes
  - After the JTextField is created and initialized, method setFont is used to set the font of the JTextField to a new object of class Font (package java.awt)
    - The new Font is initialized with "Serif" (a generic font name representing a font such as Times), Font.PLAIN style and 14-point size.
    - The string passed to the JCheckBox constructor when each check box is created is the checkbox label that appears to the right of the JCheckBox by default
  - When the user clicks a JCheckBox, an ItemEvent occurs
    - This event can be handled by an ItemListener object, which must implement method itemStateChanged
    - In this example, the event handling is performed by an instance of private inner class CheckBoxHandler
    - In the code, variable handler is an instance of class CheckBoxHandler and is registered with method addItemListener as the listener for both JCheckBox objects

Buttons that Maintain State - JCheckBox

- Program notes
  - Instance variables valBold and valItalic are used to represent the font style for the text displayed in the JTextField
  - Initially both are Font.PLAIN
  - Method itemStateChanged is called when the user clicks the bold or the italic JCheckBox
    - The method uses event.getSource() to determine which JCheckBox the user clicked
    - If it was the boldJCheckBox, JCheckBox method isSelected is used to determine if the JCheckBox is selected (i.e., it is checked)
      - If the checkbox is selected, local variable valBold is assigned Font.BOLD ; otherwise , it is assigned Font.PLAIN
      - A similar statement executes for variable valItalic if the user clicks the italicJCheckBox

Buttons that Maintain State - JCheckBox

- Program notes
  - The setFont method of the textField object is used to change the font of the JTextField, using the same font name, style, and point size
  - The sum of valBold and valItalic represents the JTextField 's new font style. Each of the Font constants represents a unique value
    - Font.PLAIN has the value 0, so if both valBold and valItalic are set to Font.PLAIN, the font will have the plain style
    - If one of the values is Font.BOLD or Font.ITALIC, the font will be bold or italic accordingly
    - If one is BOLD and the other is ITALIC , the font will be both bold and italic.

Buttons that Maintain State

- JRadioButton example
  - Radio buttons (class JRadioButton) are similar to check boxes
    - Two states (selected and not selected)
  - 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
  - Typically used to represent mutually exclusive options (i.e., multiple options in the group cannot be selected at the same time)
  - Logical relationship between radio buttons is maintained by a ButtonGroup object (package javax.swing), which itself is NOT a GUI component
    - A ButtonGroup object organizes a group of buttons and is not itself displayed in a user interface
    - Rather, the individual JRadioButton objects from the group are displayed in the GUI.

Buttons that Maintain State - JRadioButton

```
package sampleapplications;
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
```

Buttons that Maintain State - JRadioButton

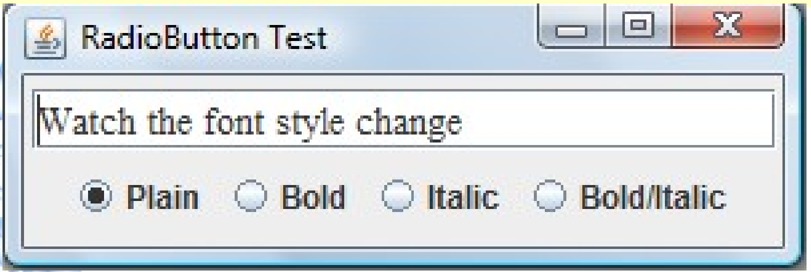
```
textField = new JTextField("Watch the font style change", 25);
add(textField); // add textField to JFrame
// create radio buttons
plainJRadioButton = new JRadioButton("Plain", true);
boldJRadioButton = new JRadioButton("Bold", false);
italicJRadioButton = new JRadioButton("Italic", false);
boldItalicJRadioButton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

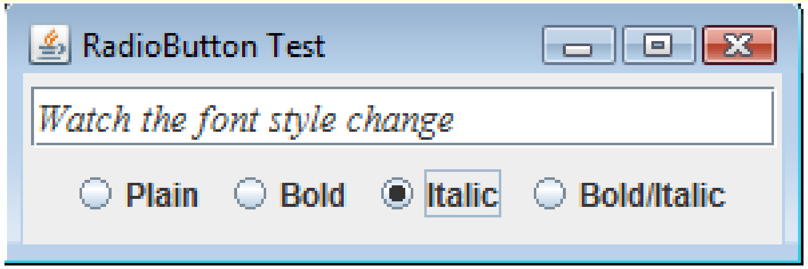
Buttons that Maintain State - JRadioButton

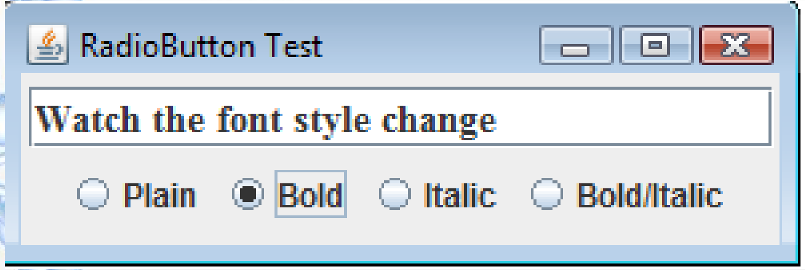
```
        // register events for JRadioButtons
        plainJRadioButton.addItemListener(new RadioButtonHandler(plainFont));
        boldJRadioButton.addItemListener(new RadioButtonHandler(boldFont));
        italicJRadioButton.addItemListener(new RadioButtonHandler(italicFont));
        boldItalicJRadioButton.addItemListener(new 
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
```

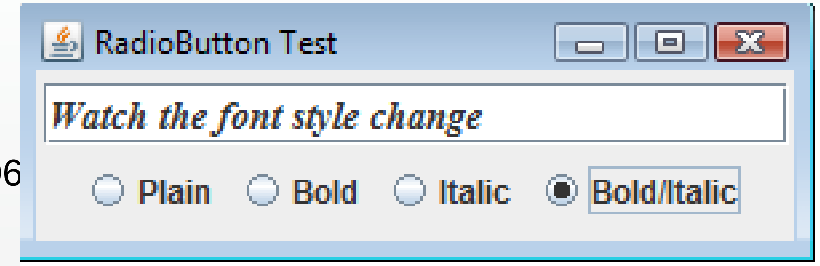
Buttons that Maintain State - JRadioButton

```
package sampleapplications;
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
}
```









Buttons that Maintain State - JRadioButton

- Program notes
  - The radio button constructor specifies the label that appears to the right of the JRadioButton by default and the initial state of the JRadioButton
    - A TRUE second argument indicates that the JRadioButton should appear selected when it is displayed, and a FALSE value indicates that the button should not be selected
  - The statement: radioGroup = new ButtonGroup()
    - Instantiates ButtonGroup object radioGroup
    - This object is the "glue" that forms the logical relationship between the four JRadioButton objects and allows only one of the four to be selected at a time

Buttons that Maintain State - JRadioButton

- Program notes
  - JRadioButtons, like JCheckBoxes, generate ItemEvents when they are clicked
  - An RadioButtonHandler instance is created for each of the four radio buttons of inner class RadioButtonHandler (declared at lines 7084)
    - Each event listener object is registered to handle the ItemEvent generated when the user clicks a particular JRadioButton
    - In this example, each RadioButtonHandler object is initialized with a particular Font object when the constructor for class RadioButtonListener runs
  - Class RadioButtonHandler implements interface ItemListener so it can handle ItemEvent s generated by the JRadioButtons
    - When the user clicks a JRadioButton, radioGroup turns off the previously selected JRadioButton and method itemStateChanged sets the font in the JTextField to the Font stored in the JRadioButton's corresponding event-listener object

Outline

- Introduction
- JOptionPane
- Overview of Swing Components
- Text Fields and Introduction to Event Handling with Nested Classes
- JButton
- Buttons that Maintain State
- JComboBox and Using Anonymous Inner Class for Event Handling
- JList
- Multiple Selection Lists
- Mouse Event Handling
- Adapter Classes
- Key-Event Handling
- Layouts

JComboBox and Using Anonymous Inner Class for Event Handling

- We demonstrate
  - How to handle events generated when the user makes a selection from combo boxes (class JComboBox)
    - A combo box (or drop-down list) provides a list of items from which the user can make a single selection
  - Introduce the use of anonymous inner classes
    - An anonymous inner class is an unnamed inner class and typically appears inside a method declaration
  - The demo program
    - Uses a JComboBox to provide a list of four image file names from which the user can select one image to display
    - When the user selects a name, the application displays the corresponding image as an Icon on a JLabel

JComboBox and Using Anonymous Inner Class for Event Handling

```
package sampleapplications;
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[]={"taj1.jpg","taj2.jpg","oak1.jpg","oak2.jpg"};
  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]))};
```

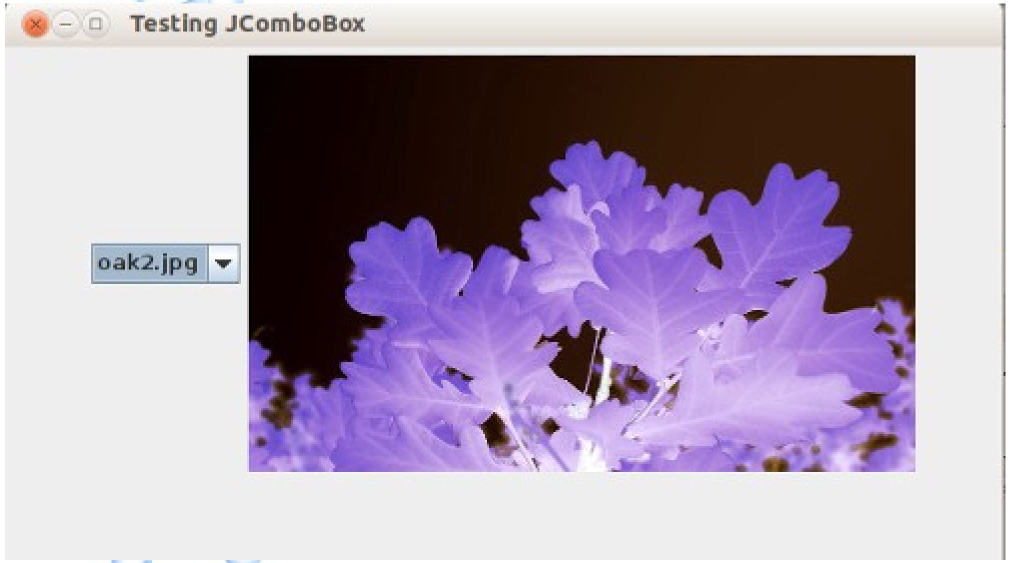
JComboBox and Using Anonymous Inner Class for Event Handling

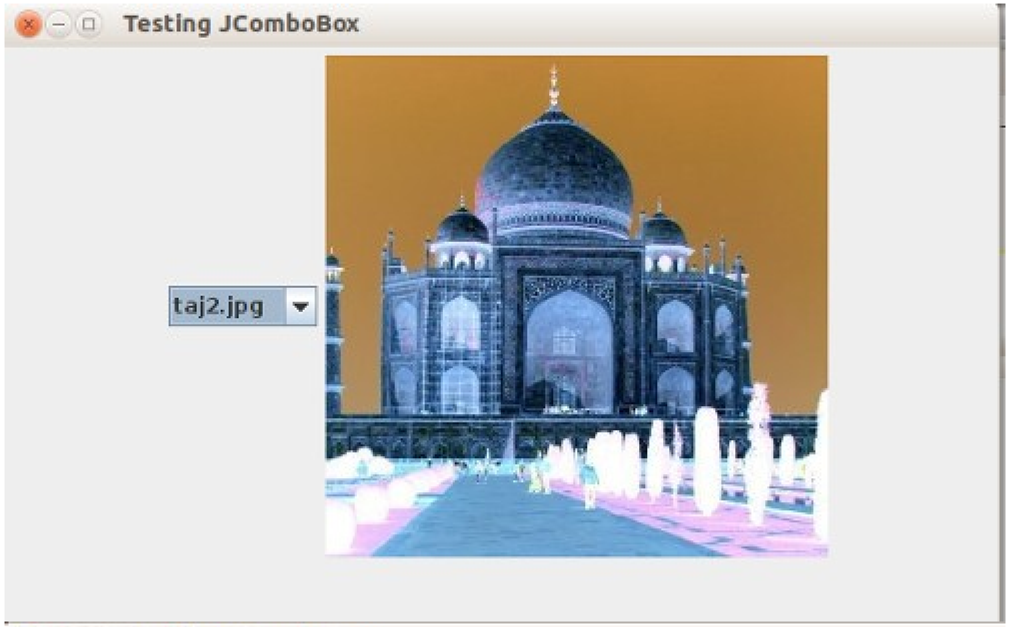
```
   // ComboBoxFrame constructor adds JComboBox to JFrame
   public ComboBoxFrame() {
     super("Testing JComboBox");
     setLayout(new FlowLayout());

     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
  }
}
```

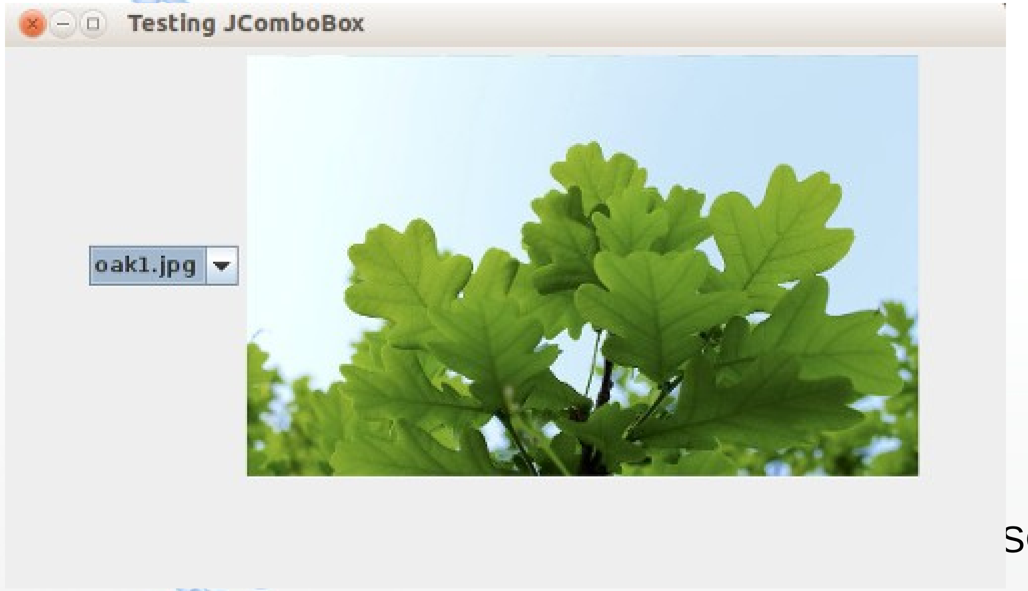
JComboBox and Using Anonymous Inner Class for Event Handling

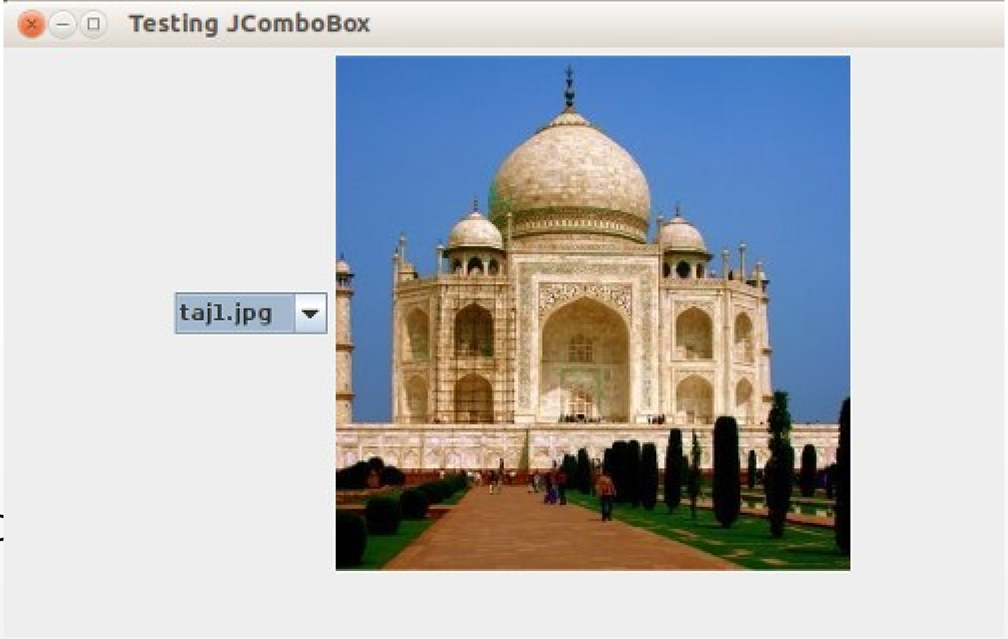
```
package sampleapplications;
import javax.swing.JFrame;
public class ComboBoxTest {
    public static void main(String args[]) {
        ComboBoxFrame comboBoxFrame = new ComboBoxFrame();
        comboBoxFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        comboBoxFrame.setSize(450, 250); // set frame size
        comboBoxFrame.setVisible(true); // display frame
    }
}
```

JComboBox and Using Anonymous Inner Class for Event Handling









SC

JComboBox and Using Anonymous Inner Class for Event Handling

- Program notes
  - String array names contains the names of the four image files that are stored in the same directory as the application
  - In the JComboBox inner class (ComboBoxFrame), the constructor creates a JComboBox object, using the Strings in array names as the elements in the list
  - Each item in the list has an index: the first item is added at index 0, the next at index 1 …
  - The first item added to a JComboBox appears as the currently selected item when the JComboBox is displayed
    - Although this can be changed by calling the combo box's setSelectedIndex or setSelectedItem method
  - Other items are selected by clicking the JComboBox, which expands into a list from which the user can make a selection

JComboBox and Using Anonymous Inner Class for Event Handling

- Program notes
  - JComboBox's method setMaximumRowCount sets the maximum number of elements that are displayed when the user clicks the JComboBox
    - If there are additional items, the JComboBox provides a scrollbar that allows the user to scroll through all the elements in the list
  - The JLabel is used to display each icon, and is initialized to the first icon

JComboBox and Using Anonymous Inner Class for Event Handling

- Program notes
  - Using an anonymous inner class for event handling
    - The lines starting from: **imagesJComboBox.addItemListener(**and ending at**)**; are a single statement that
      - declares the event listener's class
      - creates an object of that class, and
      - registers that object as the listener for imagesJComboBox's ItemEvents
    - In this example, the event-listener object is an instance of an anonymous inner class (a special form of inner class that is declared without a name and typically appears inside a method declaration)

Outline

- Introduction
- JOptionPane
- Overview of Swing Components
- Text Fields and Introduction to Event Handling with Nested Classes
- JButton
- Buttons that Maintain State
- JComboBox and Using Anonymous Inner Class for Event Handling
- JList
- Multiple Selection Lists
- Mouse Event Handling
- Adapter Classes
- Key-Event Handling
- Layouts

JList

- A list (class JList) displays a series of items from which the user may select one or more items
- 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)
- Example below is for single selection list
  - Program creates a JList containing 13 colour names
  - When a colour name is clicked in the JList , a ListSelectionEvent occurs and the application changes the background colour of the application window to the selected colour

JList

```
package sampleapplications;
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};
```

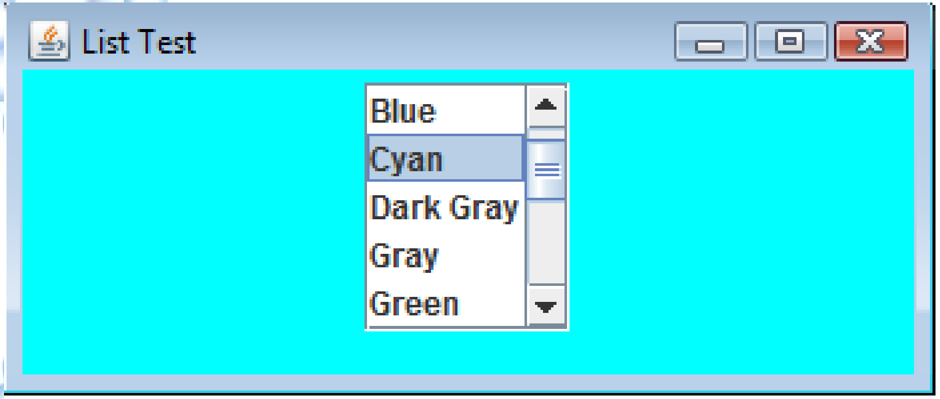
JList

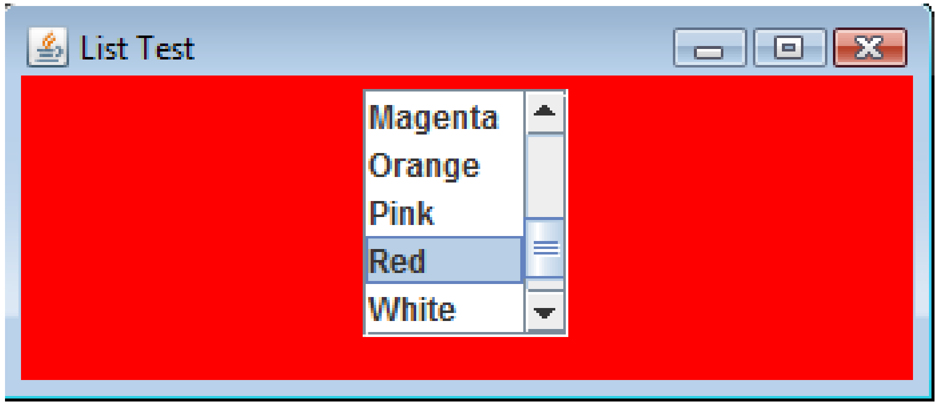
```
 // ListFrame constructor add JScrollPane containing JList to JFrame
 public ListFrame() {
     super("List Test");
     setLayout(new FlowLayout()); // set frame layout
     colorJList = new JList(colorNames); // create with colorNames
     colorJList.setVisibleRowCount(5); // display five rows 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
```

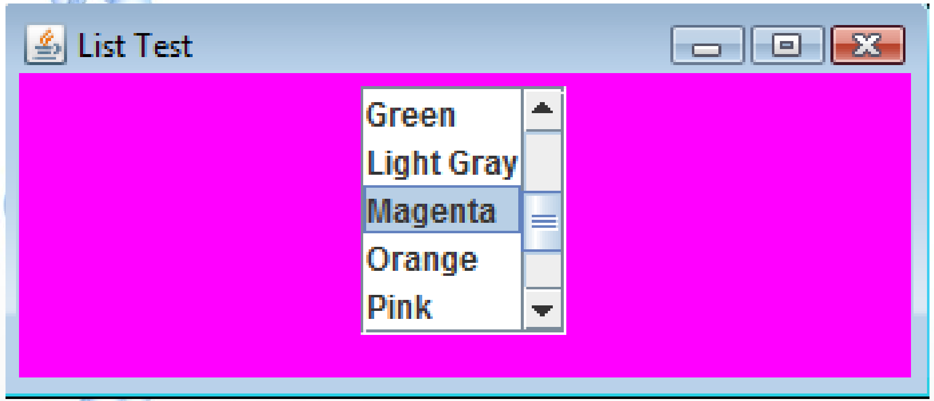
JList

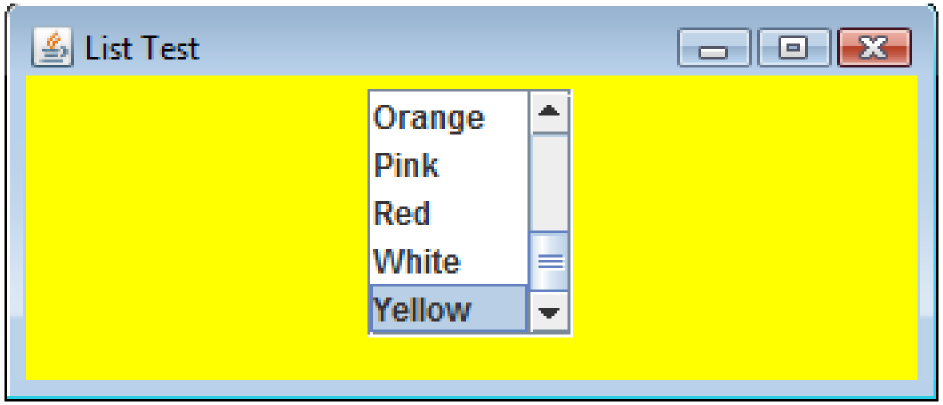
```
package sampleapplications;
import javax.swing.JFrame;
public class ListTest {
  public static void main(String args[]) {
     ListFrame listFrame = new ListFrame(); // create ListFrame
     listFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     listFrame.setSize(350, 150); // set frame size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

JList


List Test
Blue
Cyan
Dark Gray
Gray
Green


List Test
Magenta
Orange
Pink
Red
White


List Test
Green
Light Gray
Magenta
Orange
Pink


List Test
Orange
Pink
Red
White
Yellow

JList

- Program notes
  - The JList object created is colorJList
    - The argument to the JList constructor is the array of Objects (in this case Strings) to display in the list
    - JList method setVisibleRowCount determine the number of items that are visible in the list
    - JList method setSelectionMode specifies the list's selection mode via class ListSelectionModel (package javax.swing)
    - Class ListSelectionModel declares three constants that specify a JList's selection mode
      - SINGLE_SELECTION (only one item can be selected at a time)
      - SINGLE_INTERVAL_SELECTION (for a multiple-selection list that allows selection of several contiguous items to be selected)
      - MULTIPLE_INTERVAL_SELECTION (for a multiple-selection list that does not restrict the items that can be selected)

JList

- Program notes
  - Unlike a JComboBox , a JList does not provide a scrollbar if there are more items in the list than the number of visible rows
  - Hence, we use a JScrollPane object to provide the scrolling capability
  - JScrollPane’s constructor receives as its argument the JComponent that needs scrolling functionality (in this case, colorList)

JList

- Program notes
  - JList method addListSelectionListener is used to register an object that implements ListSelectionListener (package javax.swing.event) as the listener for the JList 's selection events
  - When the user makes a selection from colorList , method valueChanged changes the background color of the ListFrame to the selected colour

JList

- How?
  - Through JFrame method getContentPane
    - Each JFrame actually consists of three layers: the background, the content pane and the glass pane
    - The content pane appears in front of the background and is where the GUI components in the JFrame are displayed
    - The glass pane is used to display tool tips and other items that should appear in front of the GUI components on the screen
    - The content pane completely hides the background of the Jframe
      - To change the background colour behind the GUI components, we change the content pane's background color
      - Method getContentPane returns a reference to the JFrame 's content pane
      - Method setBackground of that reference is then used to set the content pane's background color to an element in the colors array
      - The colour is selected from the array by using the selected item's index (JList method getSelectedIndex)

Outline

- Introduction
- JOptionPane
- Overview of Swing Components
- Text Fields and Introduction to Event Handling with Nested Classes
- JButton
- Buttons that Maintain State
- JComboBox and Using Anonymous Inner Class for Event Handling
- JList
- Multiple Selection Lists
- Mouse Event Handling
- Adapter Classes
- Key-Event Handling
- Layouts

Multiple Selection Lists

- Multiple-selection list
  - Enables the user to select many items from a JList
    - SINGLE_INTERVAL_SELECTION list allows selecting a contiguous range of items
      - User clicks the first item, then press and hold the Shift key while clicking the last item in the range
    - MULTIPLE_INTERVAL_SELECTION list allows continuous range selection as described for a SINGLE_INTERVAL_SELECTION list
      - In addition, such a list allows miscellaneous items to be selected by pressing and holding the Ctrl key while clicking each item to select
      - To deselect an item, press and hold the Ctrl key while clicking the item a second time

Multiple Selection Lists

- Example program uses multiple-selection lists to copy items from one JList to another
- One list is a MULTIPLE_INTERVAL_SELECTION list and the other is a SINGLE_INTERVAL_SELECTION list
- Experiment with using the selection techniques described previously to select items in both lists

Multiple Selection Lists

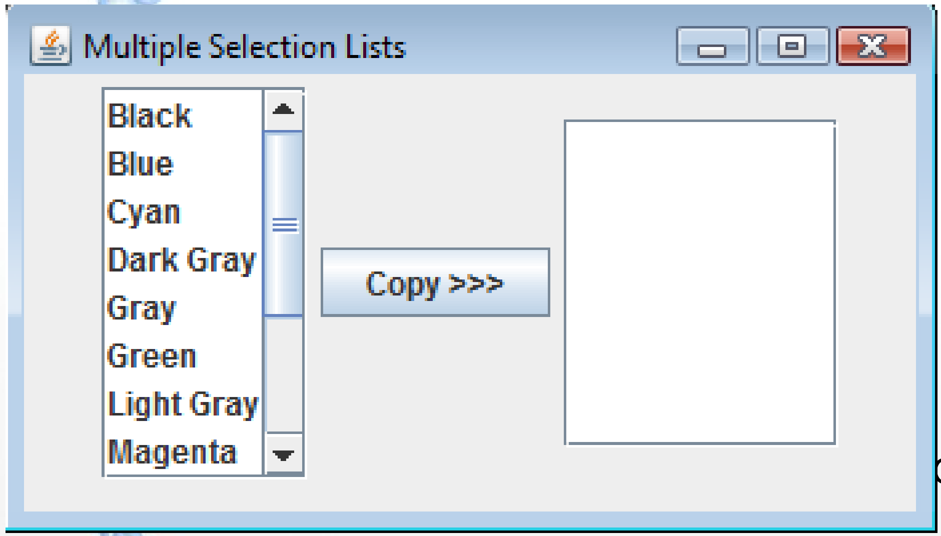
```
package sampleapplications;

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 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(8); // show 8 rows
        colorJList.setSelectionMode(ListSelectionModel.
            MULTIPLE_INTERVAL_SELECTION);
        add(new JScrollPane(colorJList)); // add list with scrollpane
```

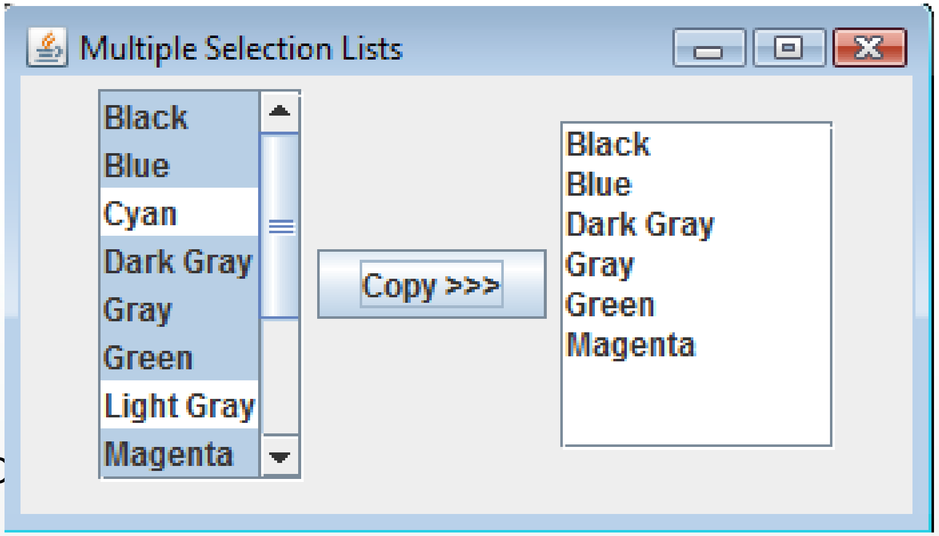
Multiple Selection Lists

```
        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(8); // show 8 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
    }
}
```

Multiple Selection Lists

```
package sampleapplications;
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, 200); // set frame size
        multipleSelectionFrame.setVisible(true); // display frame
    }
}
```





Multiple Selection Lists

- Program notes
  - The left hand list (colorJList) is a MULTIPLE_INTERVAL_SELECTION list and the right hand list (copyJList) a SINGLE_INTERVAL_SELECTION list
  - JList method setFixedCellWidth to sets a list’s width to a value in pixels
  - There are no events to indicate that a user has made multiple selections in a multiple-selection list
    - Normally, an event generated by another GUI component (known as an external event) specifies when the multiple selections in a JList should be processed
    - In this example, the user clicks the JButton called copyButton to trigger the event that copies the selected items in colorList to copyList

Multiple Selection Lists

- Program notes
  - For event handling, the program declares, creates and registers an ActionListener for the copyButton
  - When the user clicks copyButton, method actionPerformed uses JList method setListData to set the items displayed in copyList
  - colorList 's method getSelectedValues returns an array of Object s representing the selected items in colorList
  - This array is passed as the argument to copyList's setListData method
  - Note:
    - It is OK that copyList is used in the program before the copyList object is created
    - This is because method actionPerformed does not execute until the user presses the copyButton, which cannot occur until after the constructor completes execution and the application displays the GUI
    - At that point, copyList is already initialized with a new JList object

Outline

- Introduction
- JOptionPane
- Overview of Swing Components
- Text Fields and Introduction to Event Handling with Nested Classes
- JButton
- Buttons that Maintain State
- JComboBox and Using Anonymous Inner Class for Event Handling
- JList
- Multiple Selection Lists
- Mouse Event Handling
- Adapter Classes
- Key-Event Handling
- Layouts

Mouse Event Handling

- The MouseListener and MouseMotionListener event-listener interfaces are used for handling mouse events
  - Mouse events can be trapped for any GUI component that derives from java.awt.Component
  - Interface MouseInputListener (package javax.swing.event) extends interfaces MouseListener and MouseMotionListener to create a single interface containing all the MouseListener and MouseMotionListener methods
  - The corresponding methods are summarized below

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

Mouse Event Handling

MouseListener and MouseMotionListener interface methods	
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 when the mouse cursor is on a component. All move events are sent to the component over which the mouse is currently positioned

Mouse Event Handling

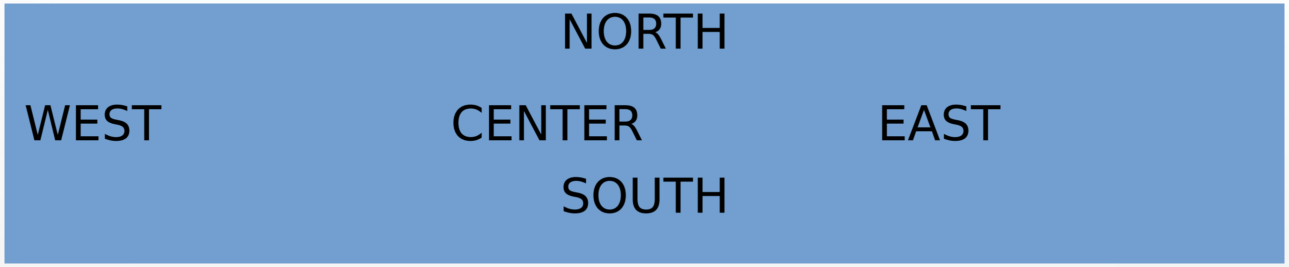
- Mouse event-handling methods
  - Take a MouseEvent object as argument
  - A MouseEvent object contains information about the mouse event that occurred, including the x- and y -coordinates of the location where the event occurred
  - In addition, the methods and constants of class InputEvent (MouseEvent's superclass) enable an application to determine which mouse button the user clicked
- Java also provides interface MouseWheelListener to enable applications to respond to the rotation of a mouse wheel
  - This interface declares method mouseWheelMoved, which receives a MouseWheelEvent as its argument
    - Class MouseWheelEvent (a subclass of MouseEvent) contains methods that enable the event handler to obtain information about the amount of wheel rotation

Mouse Event Handling

- Example program
  - Demonstrates the MouseListener and MouseMotionListener interface methods
  - Note that being interfaces, all seven methods from these two interfaces MUST be declared by the programmer when a class implements both interfaces
  - Each mouse event in this example displays a string corresponding to the mouse action, in the JLabel called statusBar at the bottom of the window

Mouse Event Handling

- Example program
  - The program creates a JPanel object (mousePanel)
  - mousePanel's background colour is initially set to white
  - When the mouse moves into mousePanel, mousePanel's background colour is changed to green
  - When the mouse moves out of mousePanel, the background colour is set back to white
  - We do not use the FlowLayout manager in this program
  - Instead, we use Jframe’s default content layout manager, BorderLayout
  - BorderLayout arranges components into five regions NORTH, SOUTH, EAST, WEST and CENTER
  - BorderLayout automatically sizes the component in the CENTER to use all the space in the JFrame that is not occupied by components in the other regions



Mouse Event Handling

```
package sampleapplications;
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
        MouseTrackerFrame.MouseHandler handler = new
MouseTrackerFrame.MouseHandler();
        mousePanel.addMouseListener(handler);
        mousePanel.addMouseMotionListener(handler);
    } // end MouseTrackerFrame constructor
```

Mouse Event Handling

```
    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()));
        }
        // handle event when mouse pressed
        public void mousePressed(MouseEvent event) {
            statusBar.setText(String.format("Pressed at [%d, %d]",
                    event.getX(), event.getY()));
        }
        // handle event when mouse released after dragging
        public void mouseReleased(MouseEvent event) {
            statusBar.setText(String.format("Released at [%d, %d]",
                    event.getX(), event.getY()));
        }
        // 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);
        }
```

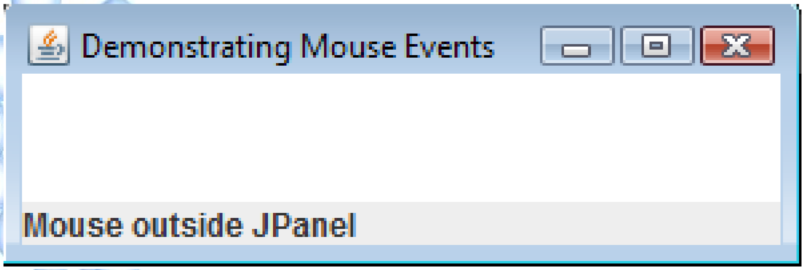
Mouse Event Handling

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

        // 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()));
        }

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

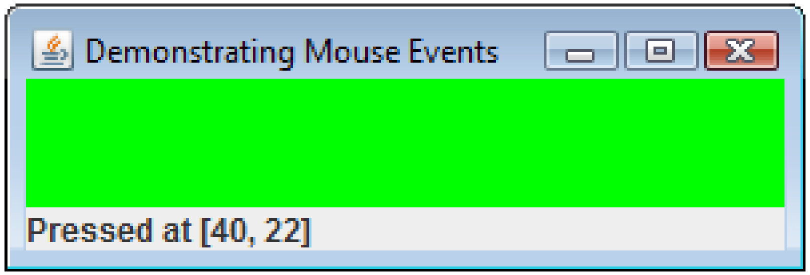
Mouse Event Handling

```
package sampleapplications;
import javax.swing.JFrame;
public class MouseTrackerTest {
    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
    }
}
```

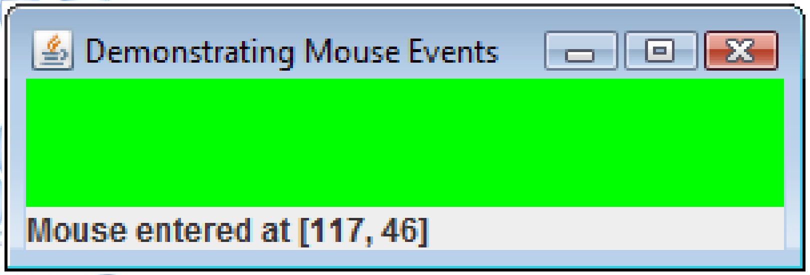
Mouse Event Handling



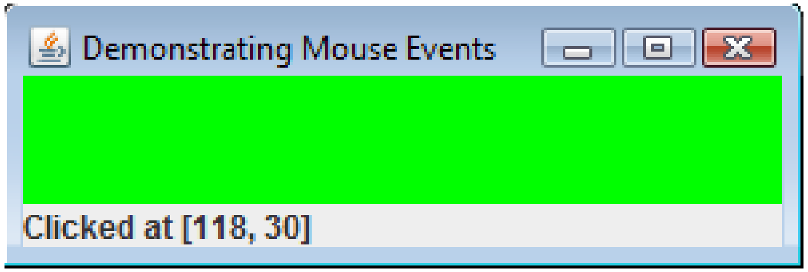
Mouse outside JPanel



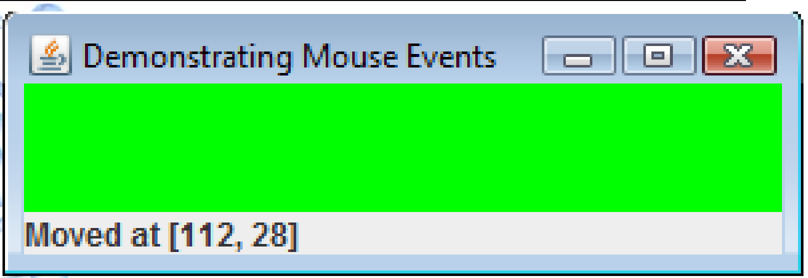
Pressed at [40, 22]



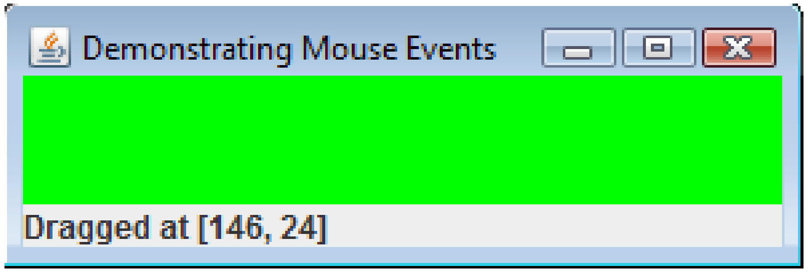
Mouse entered at [117, 46]



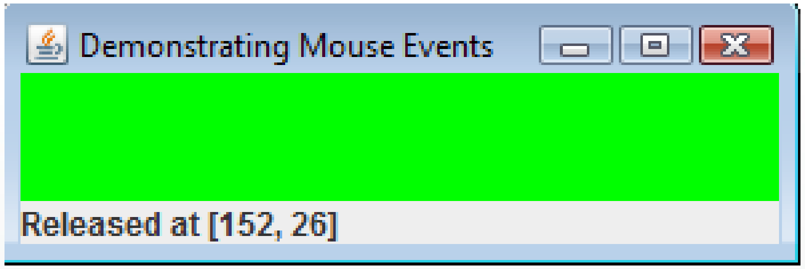
Clicked at [118, 30]



Moved at [112, 28]



Dragged at [146, 24]



Released at [152, 26]

Mouse Event Handling

- Program notes
  - The program attaches JLabel statusBar to the JFrame 's SOUTH region
    - This JLabel occupies the width of the Jframe
    - The region's height is determined by the JLabel
  - Inner class MouseHandler’s handler object responds to mouse events since it is registered as the listener for mousePanel's mouse events
    - This is done through methods addMouseListener (for interface MouseListener) and addMouseMotionListener (for interface MouseMotionListener) respectively
    - Notice that in this example, a MouseHandler object is both a MouseListener and a MouseMotionListener because the class implements both interfaces
    - We could also have opted to implemented interface MouseInputListener instead (which as we saw earlier, implements the same two interfaces)

Outline

Adapter Classes

- Several event-listener interfaces (MouseListener, MouseMotionListener, etc.) contain multiple methods
- Not always desirable to declare every method in an event-listener interface
  - We may need for example, only the mouseClicked handler from MouseListener or the mouseDragged handler from MouseMotionListener
  - 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.
    - 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

Adapter Classes

- Example program
  - Determines the number of mouse clicks
  - Distinguish between the different mouse buttons
  - Event listener is an object of inner class MouseClickHandler that extends MouseAdapter
    - Hence, we only need declare the mouseClicked (and not the other methods in interface MouseListener)

Adapter Classes

```
package sampleapplications;
import java.awt.BorderLayout;
import java.awt.Graphics;
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;
    private JLabel statusBar;
    // 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
    }
```

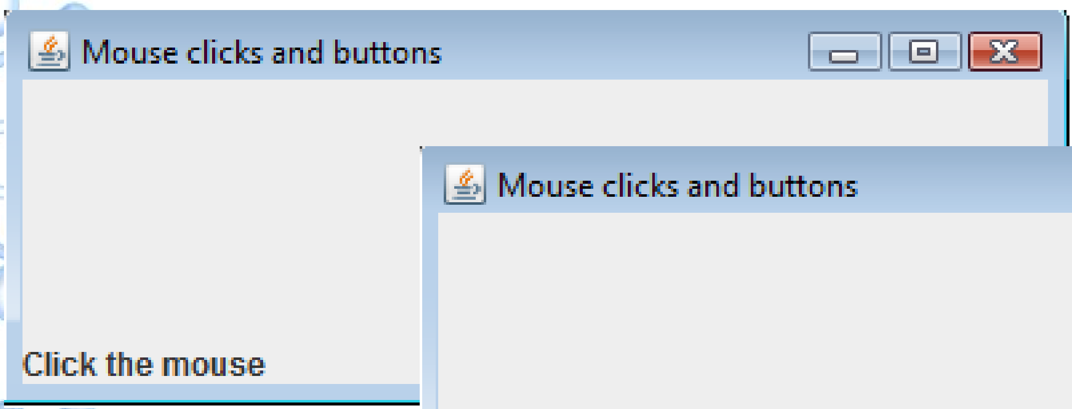
Adapter Classes

```
// 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()) { // right mouse button
                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
    }
}
```

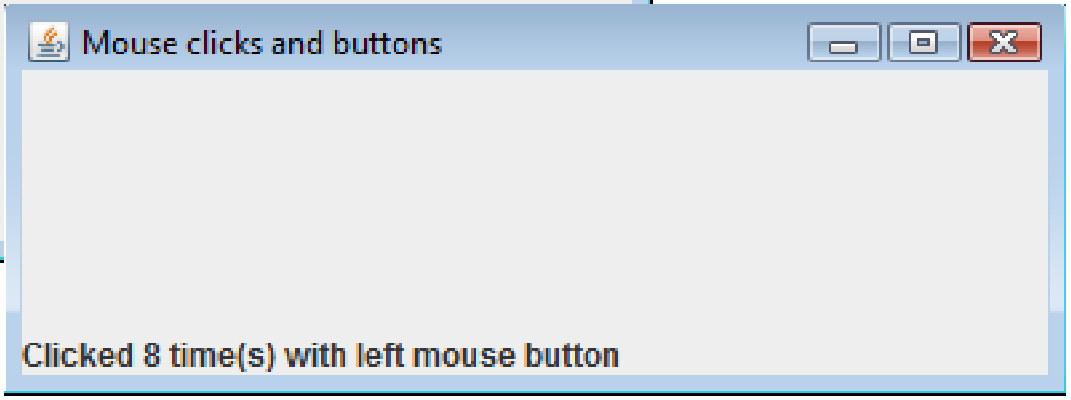
Adapter Classes

```
package sampleapplications;
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);
        mouseDetailsFrame.setVisible(true);
    }
}
```

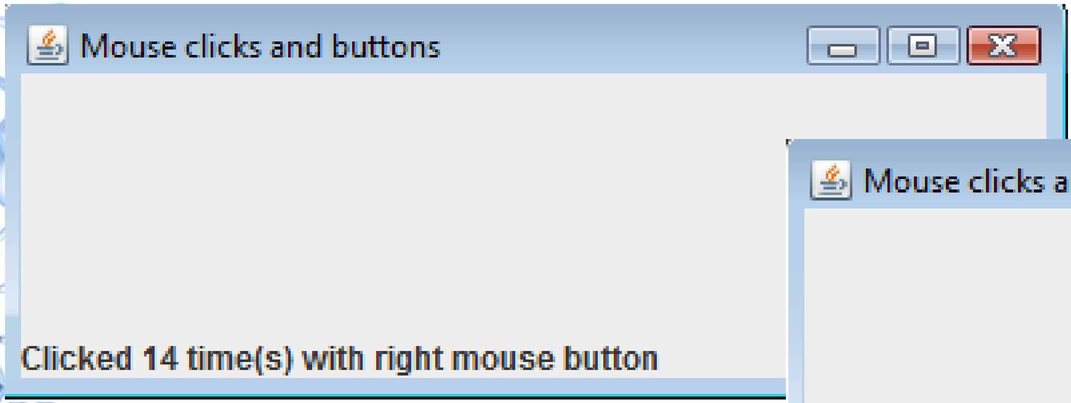
Adapter Classes



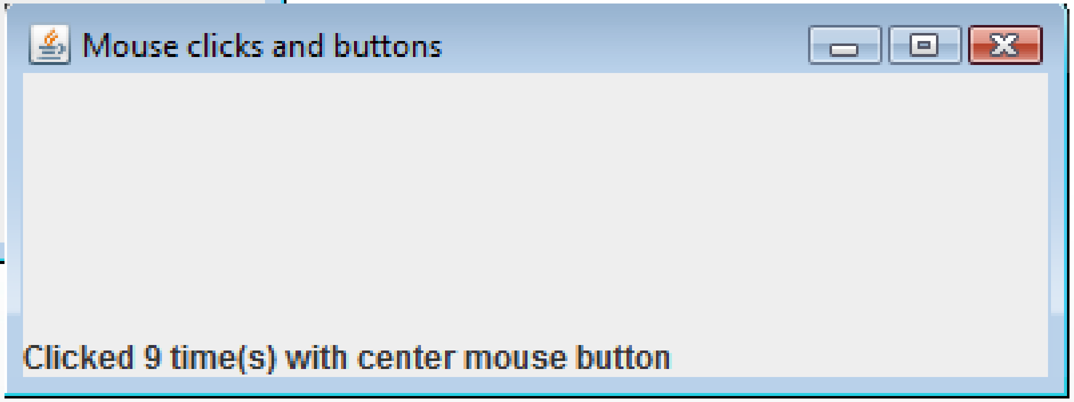
Click the mouse



Clicked 8 time(s) with left mouse button



Clicked 14 time(s) with right mouse button



Clicked 9 time(s) with center mouse button

Outline

- Introduction
- JOptionPane
- Overview of Swing Components
- Text Fields and Introduction to Event Handling with Nested Classes
- JButton
- Buttons that Maintain State
- JComboBox and Using Anonymous Inner Class for Event Handling
- JList
- Multiple Selection Lists
- Mouse Event Handling
- Adapter Classes
- Key-Event Handling
- Layouts

Key-Event Handling

- KeyListener interface
  - Handles key events
    - i.e., events that 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
    - Method keyPressed: called in response to pressing any key
    - Method keyTyped: called in response to pressing any key that is not an action key
      - Action keys are any arrow key, Home, End, Page Up, Page Down, any function key, Num Lock, Print Screen, Scroll Lock, Caps Lock and Pause)
    - Method keyReleased: called when the key is released after any keyPressed or keyTyped event
  - Example program
    - Demonstrates the KeyListener methods

Key-Event Handling

```
package sampleapplications;
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
    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
        getContentPane().add(textArea); // add textarea to JFrame

        //Note the object declared as the event handler is this object!
        addKeyListener(this); // allow frame to process key events
    }
```

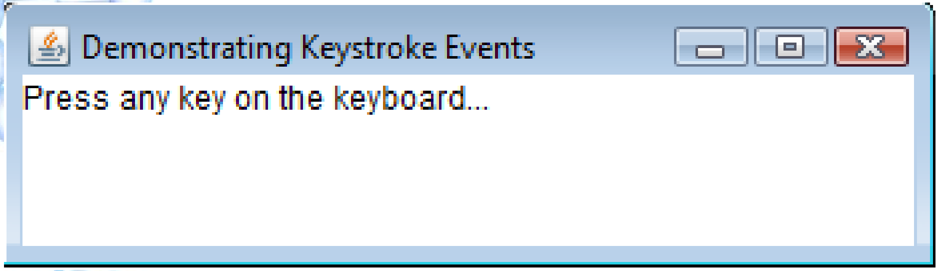
Key-Event Handling

```
// handle press of any key
    public void keyPressed(KeyEvent event) {
      //getKeyText() returns string describing the keyCode, e.g "HOME","F1","A".
       line1 = "Key pressed: " + event.getKeyText(event.getKeyCode());
       setLines2and3(event); // set output lines two and three
    }
    // handle release of any key
    public void keyReleased(KeyEvent event) {
       line1 = "Key released: " + event.getKeyText(event.getKeyCode());
       setLines2and3(event); // set output lines two and three
    }
    // handle press of an action key
    public void keyTyped(KeyEvent event) {
       line1 = "Key typed: " + event.getKeyChar();
       setLines2and3(event); // set output lines two and three
       if (event.getKeyChar() == 'A') {
           System.out.print("A was typed \t");
        }
    }
    //set second and third lines of output
    private void setLines2and3(KeyEvent event) {
       line2 = "This key is " + (event.isActionKey() ? "" : "not ") +
                        "an action key";
       String temp = event.getKeyModifiersText(event.getModifiers());
       line3="Modifier keys pressed: " + (temp.equals("") ? "none" : temp);
       textArea.setText(line1 + "\n" + line2 + "\n" + line3 + "\n");
    }
}
```

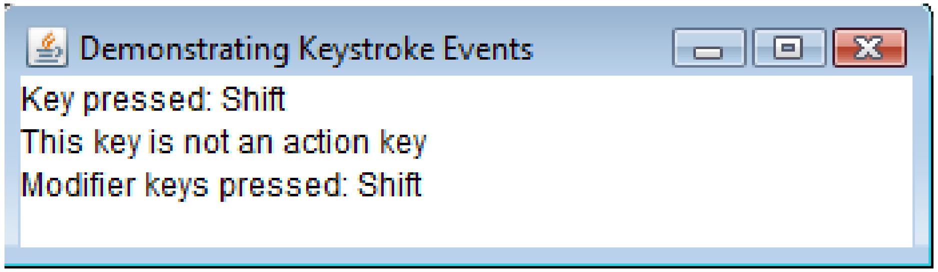
Key-Event Handling

```
package sampleapplications;
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
    }
}
```

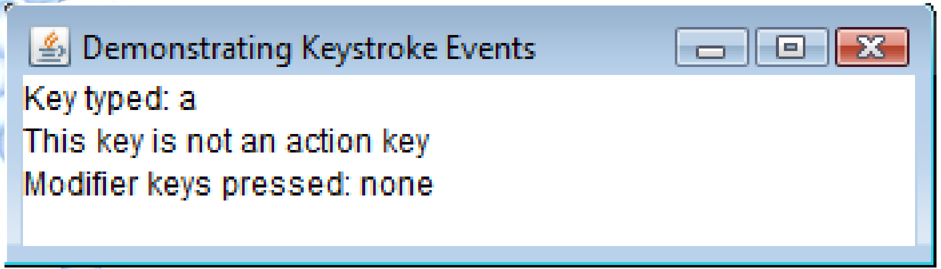
Key-Event Handling


Demonstrating Keystroke Events

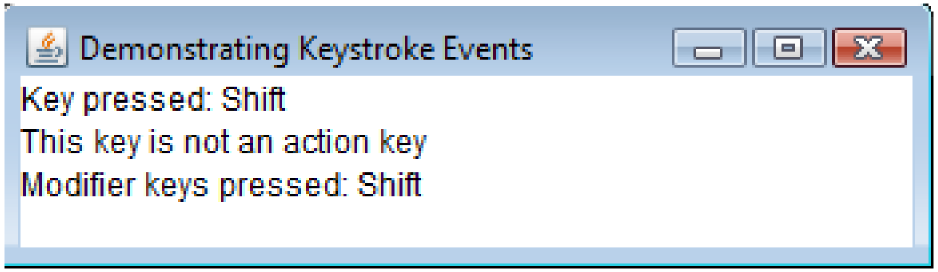
```
Press any key on the keyboard...
```


Demonstrating Keystroke Events

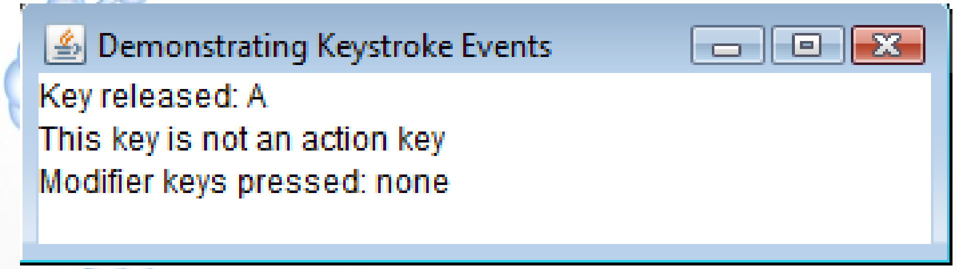
```
Key pressed: Shift
This key is not an action key
Modifier keys pressed: Shift
```


Demonstrating Keystroke Events

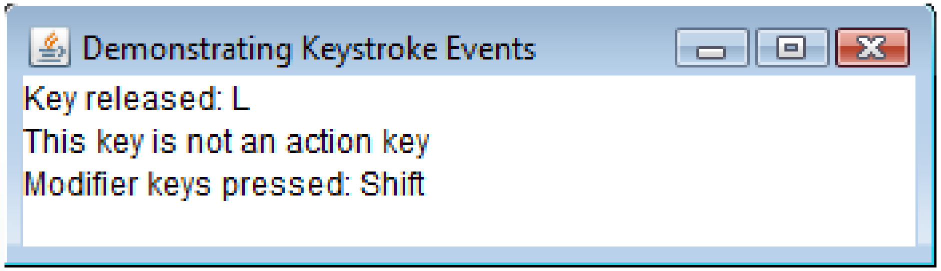
```
Key typed: a
This key is not an action key
Modifier keys pressed: none
```


Demonstrating Keystroke Events

```
Key pressed: Shift
This key is not an action key
Modifier keys pressed: Shift
```


Demonstrating Keystroke Events

```
Key released: A
This key is not an action key
Modifier keys pressed: none
```


Demonstrating Keystroke Events

```
Key released: L
This key is not an action key
Modifier keys pressed: Shift
```

Key-Event Handling

- Program notes
  - The application handles its own key events by using method addKeyListener
  - Output is displayed in a JTextArea (textArea)
    - textArea occupies the entire window because when a single Component is added to a BorderLayout (the default layout), the Component occupies the entire Container
    - textArea is disabled so that its contents cannot be edited by the user
    - Methods keyPressed and keyReleased use KeyEvent method getKeyCode to get the virtual key code of the key that was pressed

Key-Event Handling

- Program notes
  - Methods keyPressed and keyReleased use KeyEvent method getKeyCode to get the virtual key code of the key that was pressed
    - Class KeyEvent maintains a set of constants – the virtual key-code constants – that represents every key on the keyboard
    - These constants can be compared with the return value of getKeyCode to test for individual keys on the keyboard
    - The value returned by getKeyCode is passed to KeyEvent method getKeyText, which returns a string containing the name of the key that was pressed
    - Method keyTyped uses KeyEvent method getKeyChar to get the Unicode value of the character typed
    - KeyEvent method isActionKey is used to determine whether the key in the event was an action key
    - InputEvent method getModifiers is used to determine whether any modifier keys (such as Shift , Alt and Ctrl) were pressed when the key event occurred
    - KeyEvent method getKeyModifiersText is used to produce a string containing the name of pressed modifier keys.

Outline

Layouts

- 3 options to arrange components in a GUI:
  - Absolute positioning
    - Greatest level of control over a GUI’s appearance
    - Programmer sets Container’s layout to null, and then specifies the absolute position of each GUI component
    - Programmer must also specify each GUI component’s size
    - Process is tedious unless an IDE is used to automatically generate the code
  - Use of Layout managers
    - Used to quickly position components on GUI
    - Penalty is loss of some control over the size and the precision of the positioning
  - Visual programming
    - IDEs like NetBeans provide tools that make it easy to create GUIs
    - IDE provides a GUI design tool that allows you to drag and drop GUI components from a tool box onto a design area
    - Components can then be positioned, sized and aligned
    - IDE generates the Java code that creates the GUI
    - Can also add event-handling code for a particular component

Layouts

- Examples of layout managers

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

Layouts

- FlowLayout
  - Simplest layout manager
  - GUI components are placed on a container from left to right in the order in which they are added to the container
  - When the edge of the container is reached, components continue to display on the next line
  - GUI components can be left-aligned, centred, and right aligned

Layouts

- Sample program
  - Creates three JButton objects and adds them to the application, using a FlowLayout layout manager
  - Buttons are centre-aligned by default
  - When the user clicks Left, the alignment for the layout manager is changed to a left-aligned FlowLayout
    - Right for right-alignment and Center for centre-alignment
  - Each button has its own event handler that is declared with an inner class that implements ActionListener

FlowLayout Example

```
package sampleapplications;
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
```

FlowLayout Example

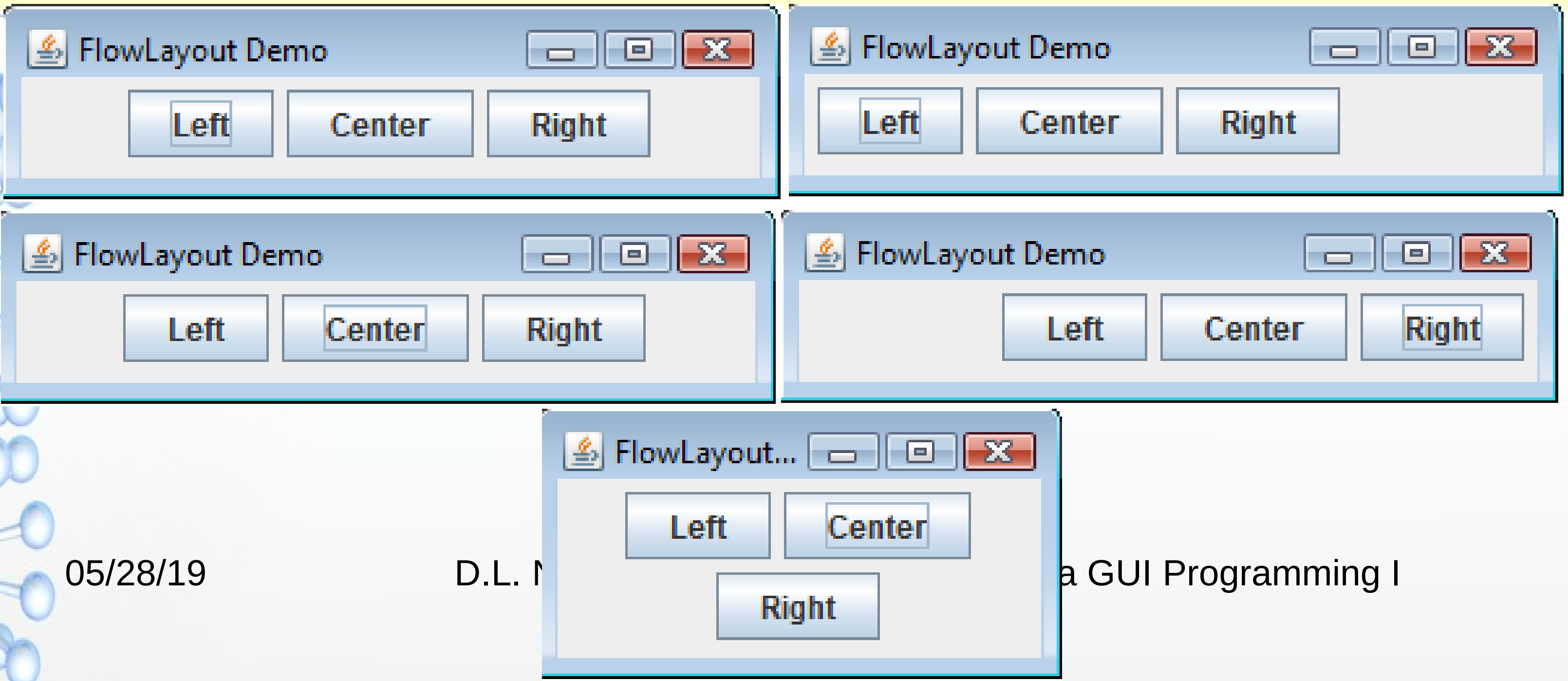
```
leftJButton.addActionListener(
        new ActionListener() {// anonymous inner class
            // process leftJButton event
            public void actionPerformed(ActionEvent event) {
                layout.setAlignment(FlowLayout.LEFT);
                // realign attached components
                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
```

FlowLayout Example

```
        // 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
    }
}
```

FlowLayout Example

```
package sampleapplications;
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
    }
}
```



BorderLayout Example

```
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 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);
        }
```

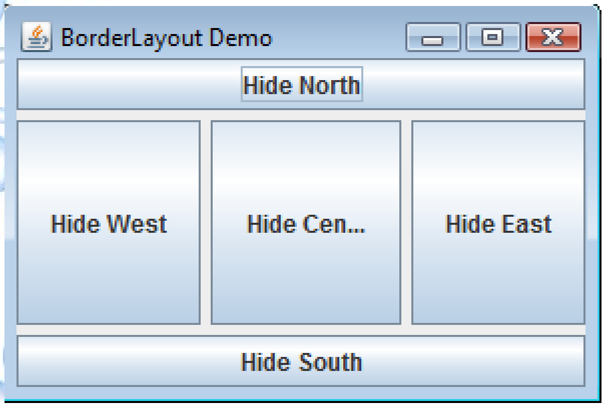
BorderLayout Example

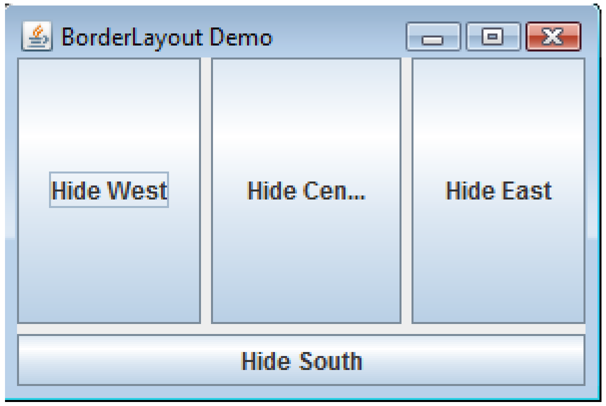
```
        add(buttons[0], BorderLayout.NORTH); // add button to north
        add(buttons[1], BorderLayout.SOUTH); // add button to south
        add(buttons[2], BorderLayout.EAST); // add button to east
        add(buttons[3], BorderLayout.WEST); // add button to west
        add(buttons[4], BorderLayout.CENTER); // add button to center
    }
    // handle button events
    public void actionPerformed(ActionEvent event) {
        // check event source and layout content pane correspondingly
        for (JButton button : buttons) {
            if (event.getSource() == button) {
                button.setVisible(false); // hide button clicked
            } else {
                button.setVisible(true); // show other buttons
            }
        }
        layout.layoutContainer(getContentPane()); // layout content pane
    }
}
```

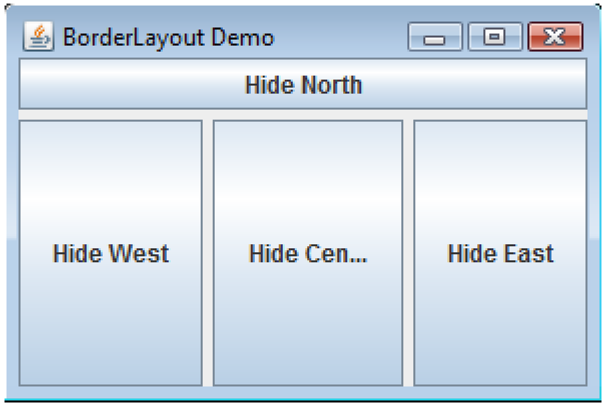
BorderLayout Example

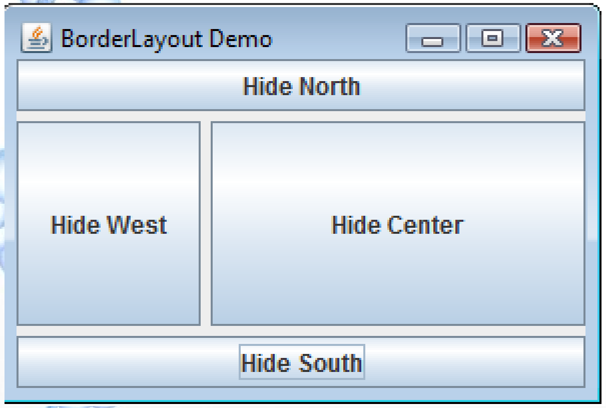
```
package sampleapplications;
import javax.swing.JFrame;
public class BorderLayoutDemo {
    public static void main(String args[]) {
        BorderLayoutFrame borderLayoutFrame = new BorderLayoutFrame();

borderLayoutFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        borderLayoutFrame.setSize(300, 200); // set frame size
        borderLayoutFrame.setVisible(true); // display frame
    }
}
```

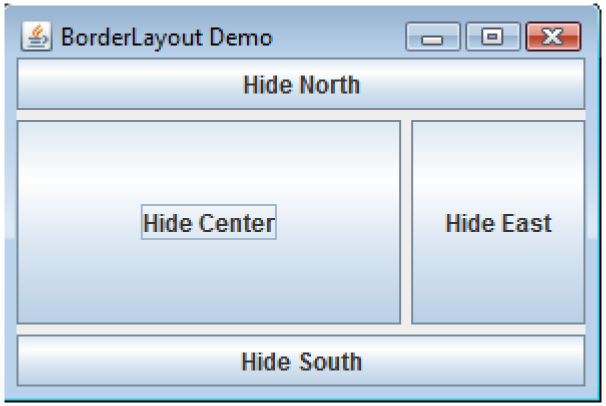
BorderLayout Example

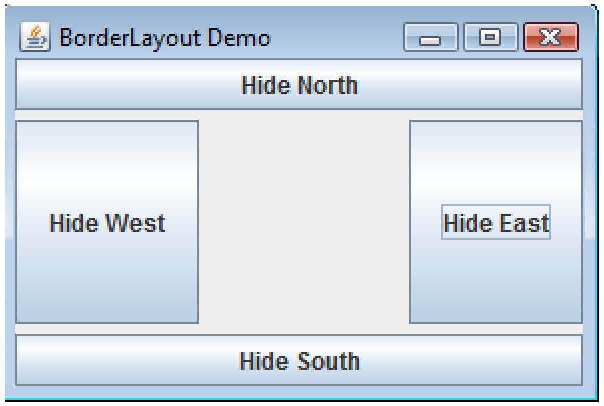












GridLayout Example

```
package sampleapplications;
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 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 gaps
    container = getContentPane(); // get content pane
    setLayout(gridLayout1); // set JFrame layout
    buttons = new JButton[names.length]; // create array of JButtons
```

GridLayout Example

```
        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
        }
    }
    // 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-layout container
    } // end method actionPerformed
}
```

GridLayout Example

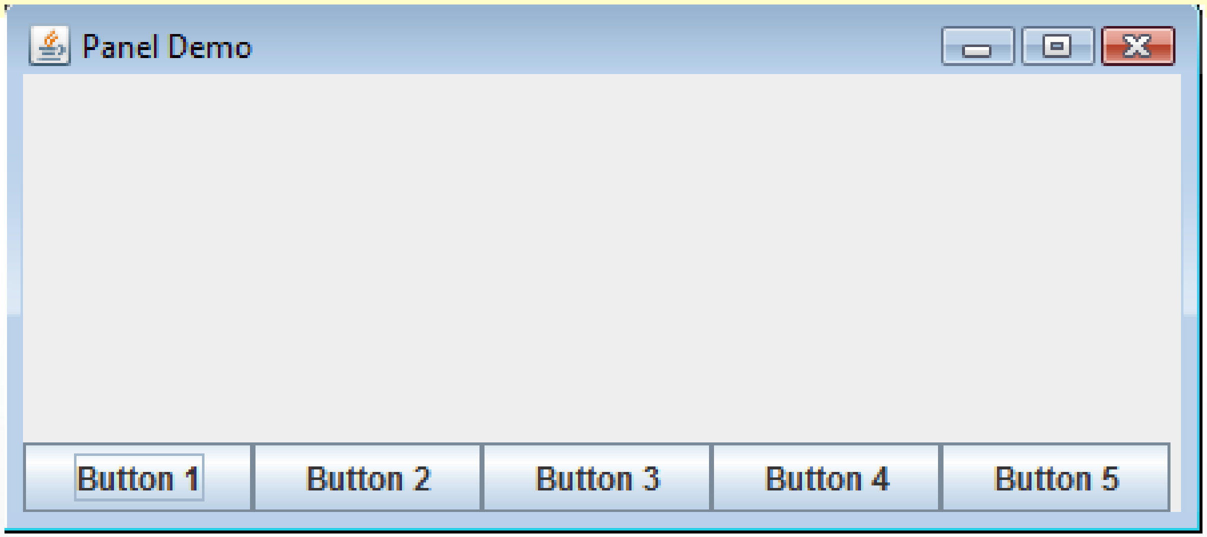
```
package sampleapplications;
import javax.swing.JFrame;
public class GridLayoutDemo {
    public static void main(String args[]) {
        GridLayoutFrame gridLayoutFrame = new GridLayoutFrame();
        gridLayoutFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        gridLayoutFrame.setSize(300, 200); // set frame size
        gridLayoutFrame.setVisible(true); // display frame
    }
}
```

Using panels for more complex layouts

```
package sampleapplications;
import java.awt.GridLayout;
import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;

public class PanelFrame extends JFrame {
    private JPanel buttonJPanel; // panel to hold buttons
    private JButton buttons[]; // array of buttons
    public PanelFrame() {
        super("Panel Demo");
        buttons = new JButton[5]; // create buttons array
        buttonJPanel = new JPanel(); // set up panel
        buttonJPanel.setLayout(new GridLayout(1, buttons.length));
        // create and add buttons
        for (int count = 0; count < buttons.length; count++) {
            buttons[ count] = new JButton("Button " + (count + 1));
            buttonJPanel.add(buttons[ count]); // add button to panel
        }

        add(buttonJPanel, BorderLayout.SOUTH); // add panel to JFrame
    }
}
```

Using panels for more complex layouts

```
package sampleapplications;
import javax.swing.JFrame;
public class PanelDemo extends JFrame {
    public static void main(String args[]) {
        PanelFrame panelFrame = new PanelFrame();
        panelFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        panelFrame.setSize(450, 200);
        panelFrame.setVisible(true);
    }
}
```


Panel Demo
Button 1
Button 2
Button 3
Button 4
Button 5