11

GUI Components: Part 1

Do you think I can listen all day to such stuff?

— Lewis Carroll

Even a minor event in the life of a child is an event of that child' s world and thus a world event.

— Gaston Bachelard

You pays your money and you takes your choice.

— Punch

Guess if you can, choose if you dare.

— Pierre Corneille

OBJECTIVES

In this chapter you will learn:

- The design principles of graphical user interfaces (GUIs).
- To build GUIs and handle events generated by user interactions with GUIs.
- To understand the packages containing GUI components, event-handling classes and interfaces.
- To create and manipulate buttons, labels, lists, text fields and panels.
- To handle mouse events and keyboard events.
- To use layout managers to arrange GUI components

Outline

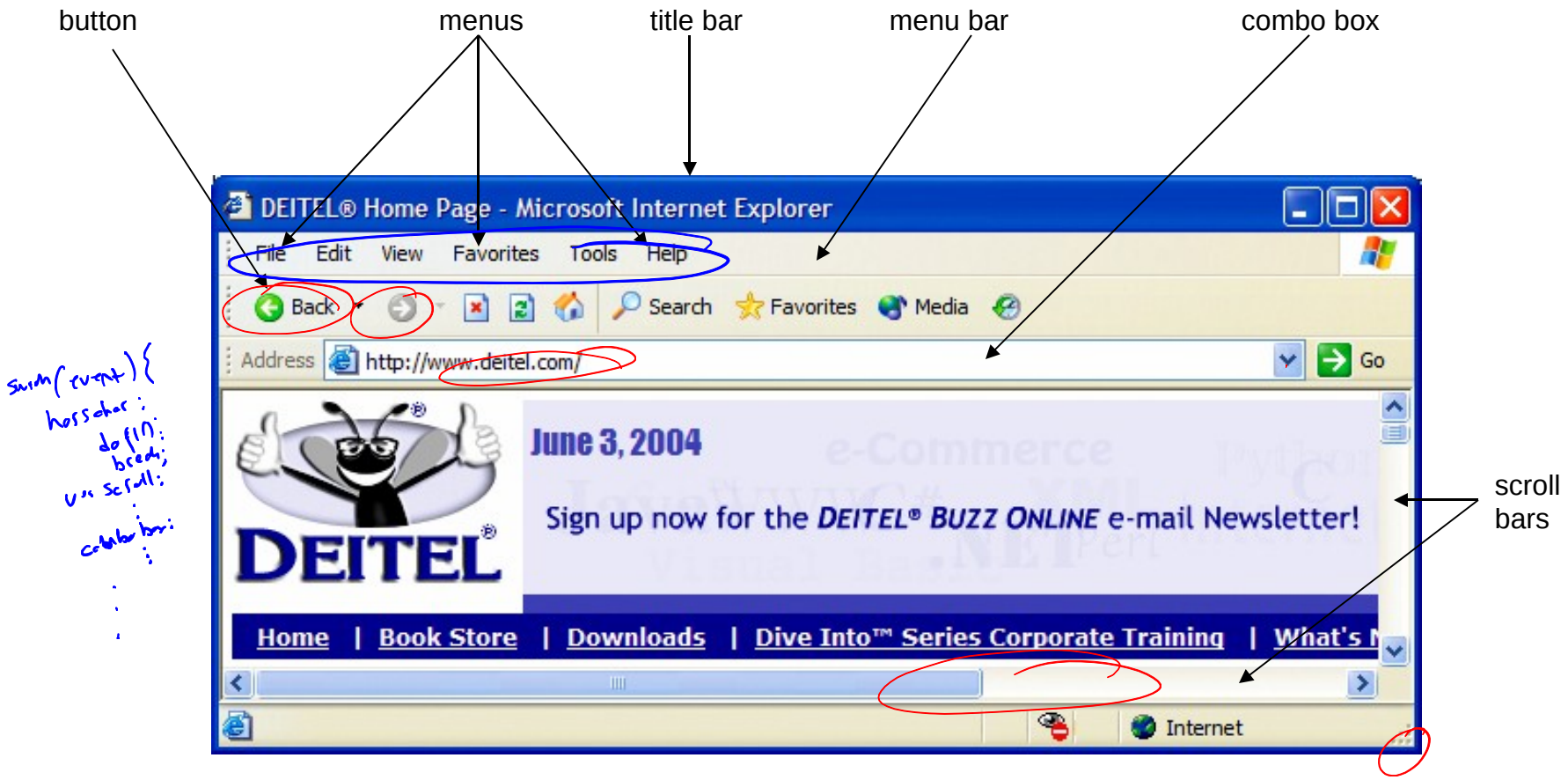
Outline

11.1 Introduction

- **Graphical user interface (GUI)**
  - **Presents a user-friendly mechanism for interacting with an application**
  - **Often contains title bar, menu bar containing menus, buttons and combo boxes**
  - **Built from GUI components**

Look-and-Feel Observation 11.1

Consistent user interfaces enable a user to learn new applications faster.



Event Based Programming

Fig. 11.1 | Internet Explorer window with GUI components.

11.2 Simple GUI-Based Input/Output with `JOptionPane`

- **Dialog boxes**
  - **Used by applications to interact with the user**
  - **Provided by Java's `JOptionPane` class**
    - **Contains input dialogs and message dialogs**

Outline

Addition.java

(1 of 2)

```java
// Fig. 11.2: Addition.java
// Addition program that uses JOptionPane for input and output.
import javax.swing.JOptionPane; // program uses JOptionPane

public class Addition
{
   public static void main( String args[] )
   {
      // obtain user input from JOptionPane input dialogs
      String firstNumber =
         JOptionPane.showInputDialog( "Enter first integer" );
      String secondNumber =
          JOptionPane.showInputDialog( "Enter second integer" );

      // convert String inputs to int values for use in a calculation
      int number1 = Integer.parseInt( firstNumber );
      int number2 = Integer.parseInt( secondNumber );

      int sum = number1 + number2; // add numbers

      // display result in a JOptionPane message dialog
      JOptionPane.showMessageDialog( null, "The sum is " + sum,
         "Sum of Two Integers", JOptionPane.PLAIN_MESSAGE );
   } // end method main
} // end class Addition
```

Boring

int ⇄ Integer

println(o)

<<
>>

Outline

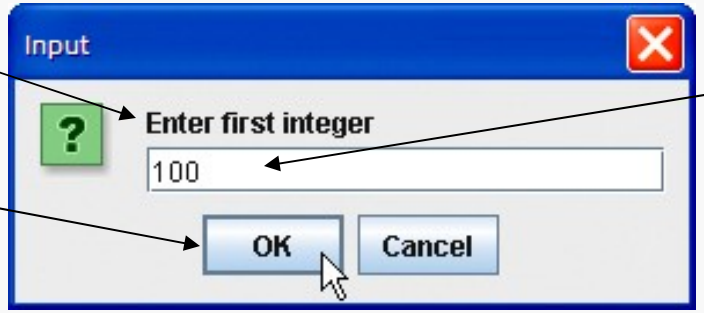
`Addition.java`

(2 of 2)

Input dialog displayed by lines 10–11

Prompt to the user

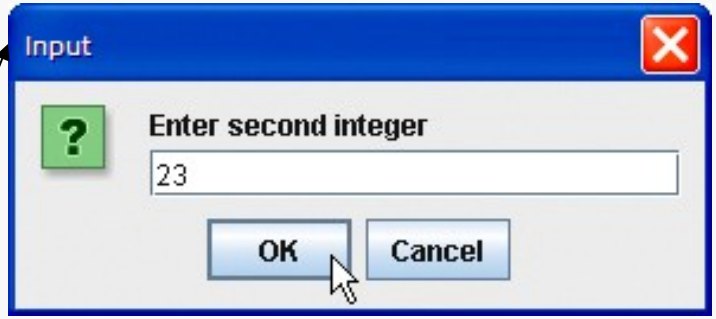
When the user clicks **OK**, **`showInputDialog`** returns to the program the **100** typed by the user as a **`String`**. The program must convert the **`String`** to an **`int`**



Text field in which the user types a value

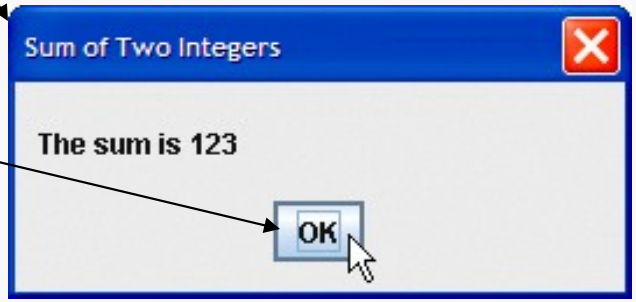
Input dialog displayed by lines 12–13

title bar



Message dialog displayed by lines 22–23

When the user clicks **OK**, the message dialog is dismissed (removed from the screen)



Look-and-Feel Observation 11.2

The prompt in an input dialog typically uses *sentence-style capitalization*—a style that capitalizes only the first letter of the first word in the text unless the word is a proper noun (for example, Deitel).

Look-and-Feel Observation 11.3

The title bar of a window typically uses *book-title capitalization*—a style that capitalizes the first letter of each significant word in the text and does not end with any punctuation (for example, Capitalization in a Book Title).

Message dialog type	Icon	Description
`ERROR_MESSAGE`		A dialog that indicates an error to the user.
`INFORMATION_MESSAGE`		A dialog with an informational message to the user.
`WARNING_MESSAGE`		A dialog warning the user of a potential problem.
`QUESTION_MESSAGE`		A dialog that poses a question to the user. This dialog normally requires a response, such as clicking a `Yes` or a `No` button.
`PLAIN_MESSAGE`	no icon	A dialog that contains a message, but no icon.

Fig. 11.3 | `JOptionPane` `static` constants for message dialogs.

11.3 Overview of Swing Components

- **Swing GUI components**
  - **Declared in package `javax.swing`**
  - **Most are pure Java components**
  - **Part of the Java Foundation Classes (JFC)**

Component	Description
`JLabel`	Displays uneditable text or icons.
`JTextField`	Enables user to enter data from the keyboard. Can also be used to display editable or uneditable text.
`JButton`	Triggers an event when clicked with the mouse.
`JCheckBox`	Specifies an option that can be selected or not selected.
`JComboBox`	Provides a drop-down list of items from which the user can make a selection by clicking an item or possibly by typing into the box.
`JList`	Provides a list of items from which the user can make a selection by clicking on any item in the list. Multiple elements can be selected.
`JPanel`	Provides an area in which components can be placed and organized. Can also be used as a drawing area for graphics.

Frame

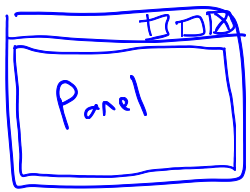


Fig. 11.4 | Some basic GUI components.

Swing vs. AWT

- **Abstract Window Toolkit (AWT)**
  - **Precursor to Swing**
  - **Declared in package `java.awt`**
  - **Does not provide consistent, cross-platform look-and-feel**

Portability Tip 11.1

Swing components are implemented in Java, so they are more portable and flexible than the original Java GUI components from package `java.awt`, which were based on the GUI components of the underlying platform. For this reason, Swing GUI components are generally preferred.

Lightweight vs. Heavyweight GUI Components

- **Lightweight components**
  - **Not tied directly to GUI components supported by underlying platform**
- **Heavyweight components**
  - **Tied directly to the local platform**
  - **AWT components**
  - **Some Swing components**

Look-and-Feel Observation 11.4

The look and feel of a GUI defined with heavyweight GUI components from package `java.awt` may vary across platforms. Because heavyweight components are tied to the local-platform GUI, the look and feel varies from platform to platform.

Superclasses of Swing's Lightweight GUI Components

- **Class** `Component` **(package** `java.awt`**)**
  - **Subclass of** `Object`
  - **Declares many behaviors and attributes common to GUI components**
- **Class** `Container` **(package** `java.awt`**)**
  - **Subclass of** `Component`
  - **Organizes** `Components`
- **Class** `JComponent` **(package** `javax.swing`**)**
  - **Subclass of** `Container`
  - **Superclass of all lightweight Swing components**

Software Engineering Observation 11.1

Study the attributes and behaviors of the classes in the class hierarchy of Fig. 11.5. These classes declare the features that are common to most Swing components.

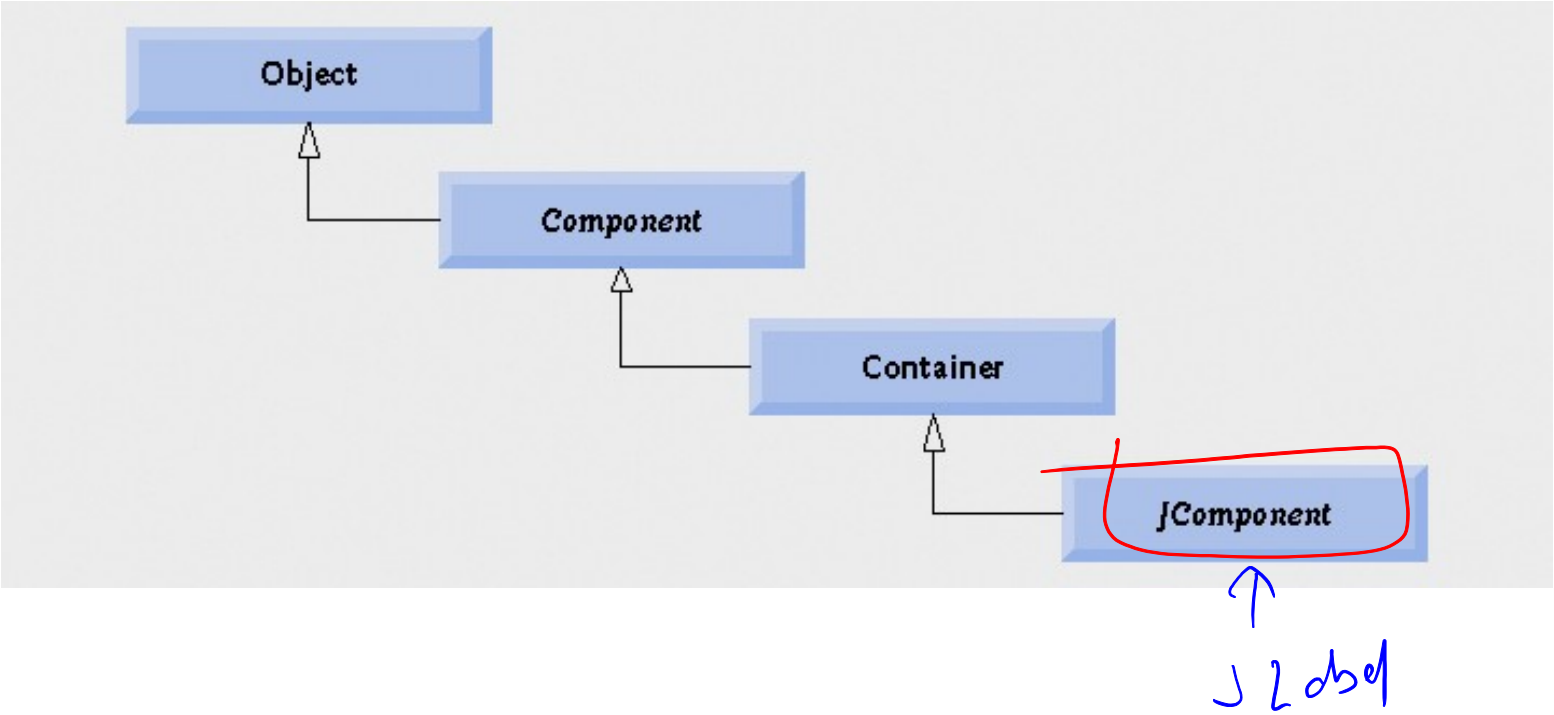


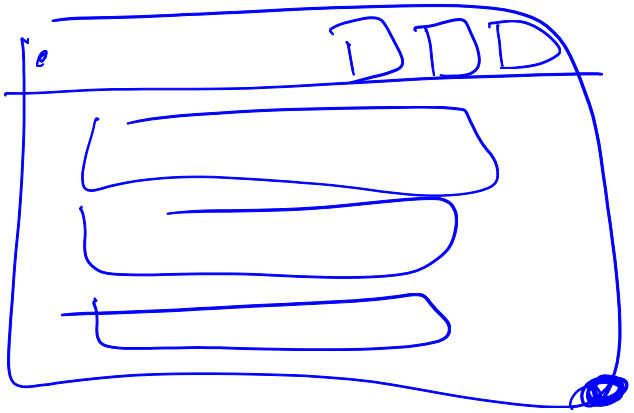
Fig. 11.5 | Common superclasses of many of the Swing components.

Superclasses of Swing's Lightweight GUI Components

- **Common lightweight component features**
  - **Pluggable look-and-feel to customize the appearance of components**
  - **Shortcut keys (called mnemonics)**
  - **Common event-handling capabilities**
  - **Brief description of component's purpose (called tool tips)**
  - **Support for localization**

11.4 Displaying Text and Images in a Window

- **Class `JFrame`**
  - **Most windows are an instance or subclass of this class**
  - **Provides title bar**
  - **Provides buttons to minimize, maximize and close the application**



Labeling GUI Components

- **Label**
  - **Text instructions or information stating the purpose of each component**
  - **Created with class** `JLabel`

Look-and-Feel Observation 11.5

Text in a `JLabel` normally uses sentence-style capitalization.

Specifying the Layout

- **Laying out containers**
  - **Determines where components are placed in the container**
  - **Done in Java with layout managers**
    - **One of which is class `FlowLayout`**
  - **Set with the `setLayout` method of class `JFrame`**

Outline

LabelFrame.java

(1 of 2)

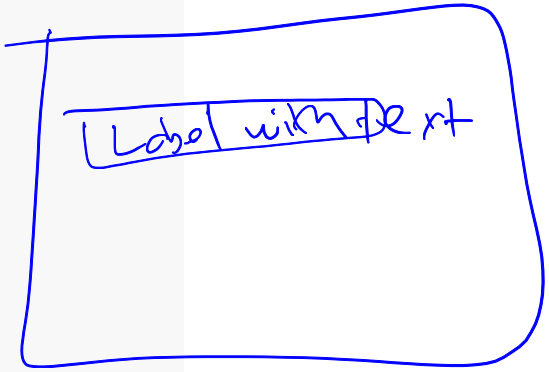
```java
// Fig. 11.6: LabelFrame.java
// Demonstrating the JLabel class.
import java.awt.FlowLayout; // specifies how components are arranged
import javax.swing.JFrame; // provides basic window features
import javax.swing.JLabel; // displays text and images
import javax.swing.SwingConstants; // common constants used with Swing
import javax.swing.Icon; // interface used to manipulate images
import javax.swing.ImageIcon; // loads images

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

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

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

```



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

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

Outline

LabelFrame.java

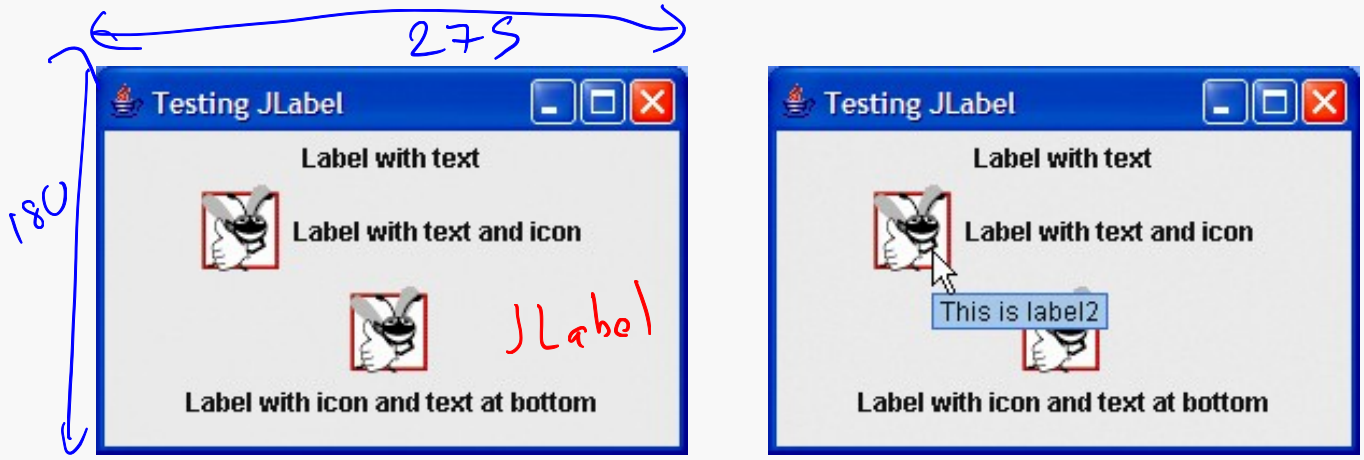
(2 of 2)

Outline

LabelTest.java

```java
// Fig. 11.7: LabelTest.java
// Testing LabelFrame.
import javax.swing.JFrame;

public class LabelTest
{
   public static void main( String args[] )
   {
      LabelFrame labelFrame = new LabelFrame(); // create LabelFrame
      labelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      labelFrame.setSize( 275, 180 ); // set frame size
      labelFrame.setVisible( true ); // display frame
   } // end main
} // end class LabelTest
```



Creating and Attaching `label1`

- **Method** `setToolTipText` **of class** `JComponent`
  - **Specifies the tool tip**
- **Method** `add` **of class** `Container`
  - **Adds a component to a container**

Common Programming Error 11.1

If you do not explicitly add a GUI component to a container, the GUI component will not be displayed when the container appears on the screen.

Look-and-Feel Observation 11.6

Use tool tips to add descriptive text to your GUI components. This text helps the user determine the GUI component's purpose in the user interface.

Creating and Attaching `label2`

- **Interface** `Icon`
  - **Can be added to a** `JLabel` **with the** `setIcon` **method**
  - **Implemented by class** `ImageIcon`
- **Interface** `SwingConstants`
  - **Declares a set of common integer constants such as those used to set the alignment of components**
  - **Can be used with methods** `setHorizontalAlignment` **and** `setVerticalAlignment`

Creating and Attaching `label3`

- **Other `JLabel` methods**
  - `getText` **and** `setText`
    - **For setting and retrieving the text of a label**
  - `getIcon` **and** `setIcon`
    - **For setting and retrieving the icon displayed in the label**
  - `getHorizontalTextPosition` **and** `setHorizontalTextPosition`
    - **For setting and retrieving the horizontal position of the text displayed in the label**

Constant	Description
Horizontal-position constants	
`SwingConstants.LEFT`	Place text on the left.
`SwingConstants.CENTER`	Place text in the center.
`SwingConstants.RIGHT`	Place text on the right.
Vertical-position constants	
`SwingConstants.TOP`	Place text at the top.
`SwingConstants.CENTER`	Place text in the center.
`SwingConstants.BOTTOM`	Place text at the bottom.

Fig. 11.8 | Some basic GUI components.

Creating and Displaying a `LabelFrame` Window

- **Other `JFrame` methods**
  - `setDefaultCloseOperation`
    - **Dictates how the application reacts when the user clicks the close button**
  - `setSize`
    - **Specifies the width and height of the window**
  - `setVisible`
    - **Determines whether the window is displayed (`true`) or not (`false`)**

11.5 Text Fields and an Introduction to Event Handling with Nested Classes

- **GUIs are event-driven**
  - **A user interaction creates an event**
    - **Common events are clicking a button, typing in a text field, selecting an item from a menu, closing and window and moving the mouse**
  - **The event causes a call to a method called an event handler**

11.5 Text Fields and an Introduction to Event Handling with Nested Classes

- **Class** `JTextComponent`
  - **Superclass of** `JTextField`
    - **Superclass of** `JPasswordField`
      - **Adds echo character to hide text input in component**
  - **Allows user to enter text in the component when component has the application's focus**

Outline

TextFieldFrame.java

(1 of 3)

```java
// Fig. 11.9: TextFieldFrame.java
// Demonstrating the JTextField class.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JPasswordField;
import javax.swing.JOptionPane;

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

   // TextFieldFrame constructor adds JTextFields to JFrame
   public TextFieldFrame()
   {
      super( "Testing JTextField and JPasswordField" );
      setLayout( new FlowLayout() ); // set frame layout

      // construct textfield with 10 columns
      textField1 = new JTextField( 10 );
      add( textField1 ); // add textField1 to JFrame

```

```
      // construct textfield with default text
      textField2 = new JTextField( "Enter text here" );
      add( textField2 ); // add textField2 to JFrame

      // construct textfield with default text and 21 columns
      textField3 = new JTextField( "Uneditable text field", 21 );
      textField3.setEditable( false ); // disable editing
      add( textField3 ); // add textField3 to JFrame

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

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

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

```

Outline

TextFieldFrame.java

(2 of 3)

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

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

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

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

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

Outline

TextFieldFrame.java

(3 of 3)

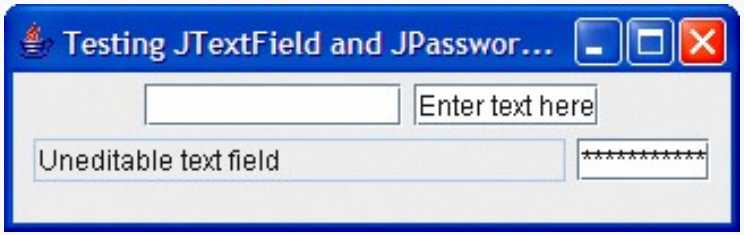
Outline

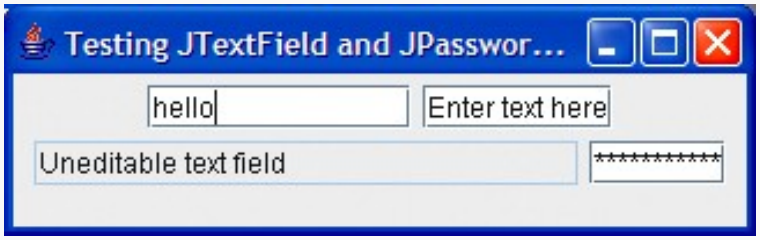
TextFieldTest.java

(1 of 2)

```java
// Fig. 11.10: TextFieldTest.java
// Testing TextFieldFrame.
import javax.swing.JFrame;

public class TextFieldTest
{
   public static void main( String args[] )
   {
      TextFieldFrame textFieldFrame = new TextFieldFrame();
      textFieldFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      textFieldFrame.setSize( 325, 100 ); // set frame size
      textFieldFrame.setVisible( true ); // display frame
   } // end main
} // end class TextFieldTest
```

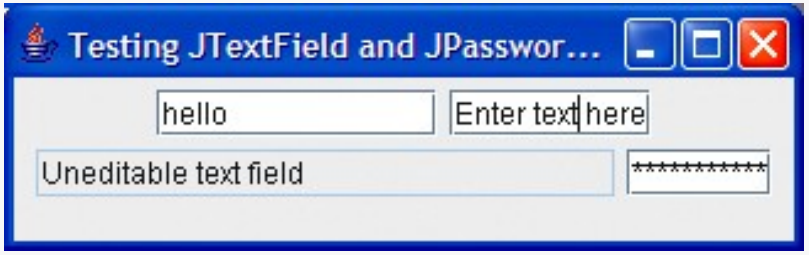


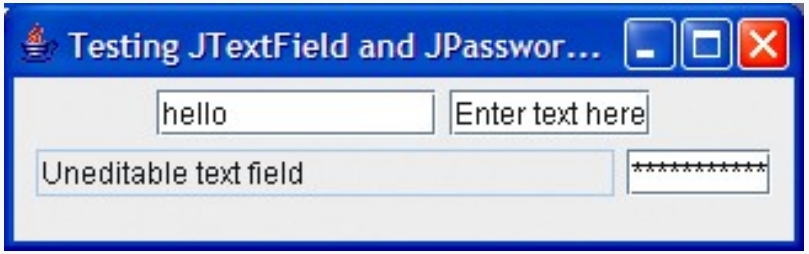


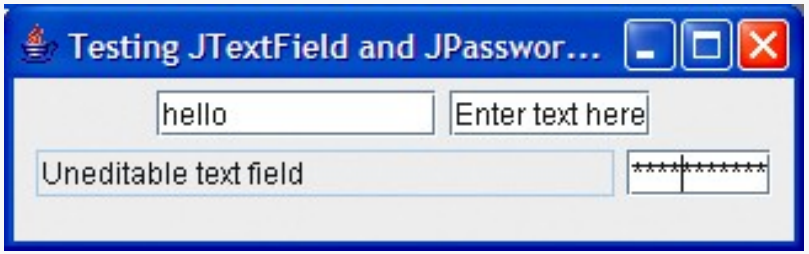
Outline

`TextFieldTest.java`

(2 of 2)







Steps Required to Set Up Event Handling for a GUI Component

- **Several coding steps are required for an application to respond to events**
  - **Create a class for the event handler**
  - **Implement an appropriate event-listener interface**
  - **Register the event handler**

Using a Nested Class to Implement an Event Handler

- **Top-level classes**
  - **Not declared within another class**
- **Nested classes**
  - **Declared within another class**
  - **Non-`static` nested classes are called inner classes**
    - **Frequently used for event handling**

Software Engineering Observation 11.2

An inner class is allowed to directly access its top-level class's variables and methods, even if they are `private`.

Using a Nested Class to Implement an Event Handler

- `JTextFields` **and** `JPasswordFields`
  - **Pressing enter within either of these fields causes an** `ActionEvent`
    - **Processed by objects that implement the** `ActionListener` **interface**

Registering the Event Handler for Each Text Field

- **Registering an event handler**
  - **Call method** `addActionListener` **to register an** `ActionListener` **object**
  - `ActionListener` **listens for events on the object**

Software Engineering Observation 11.3

The event listener for an event must implement the appropriate event-listener interface.

Common Programming Error 11.2

Forgetting to register an event-handler object for a particular GUI component's event type causes events of that type to be ignored.

Details of Class `TextFieldHandler`'s `actionPerformed` Method

- **Event source**
  - **Component from which event originates**
  - **Can be determined using method** `getSource`
  - **Text from a** `JTextField` **can be acquired using** `getActionCommand`
  - **Text from a** `JPasswordField` **can be acquired using** `getPassword`

11.6 Common GUI Event Types and Listener Interfaces

- **Event types**
  - **All are subclasses of** `AWTEvent`
  - **Some declared in package** `java.awt.event`
  - **Those specific to Swing components declared in** `javax.swing.event`

11.6 Common GUI Event Types and Listener Interfaces

- **Delegation event model**
  - **Event source is the component with which user interacts**
  - **Event object is created and contains information about the event that happened**
  - **Event listener is notified when an event happens**

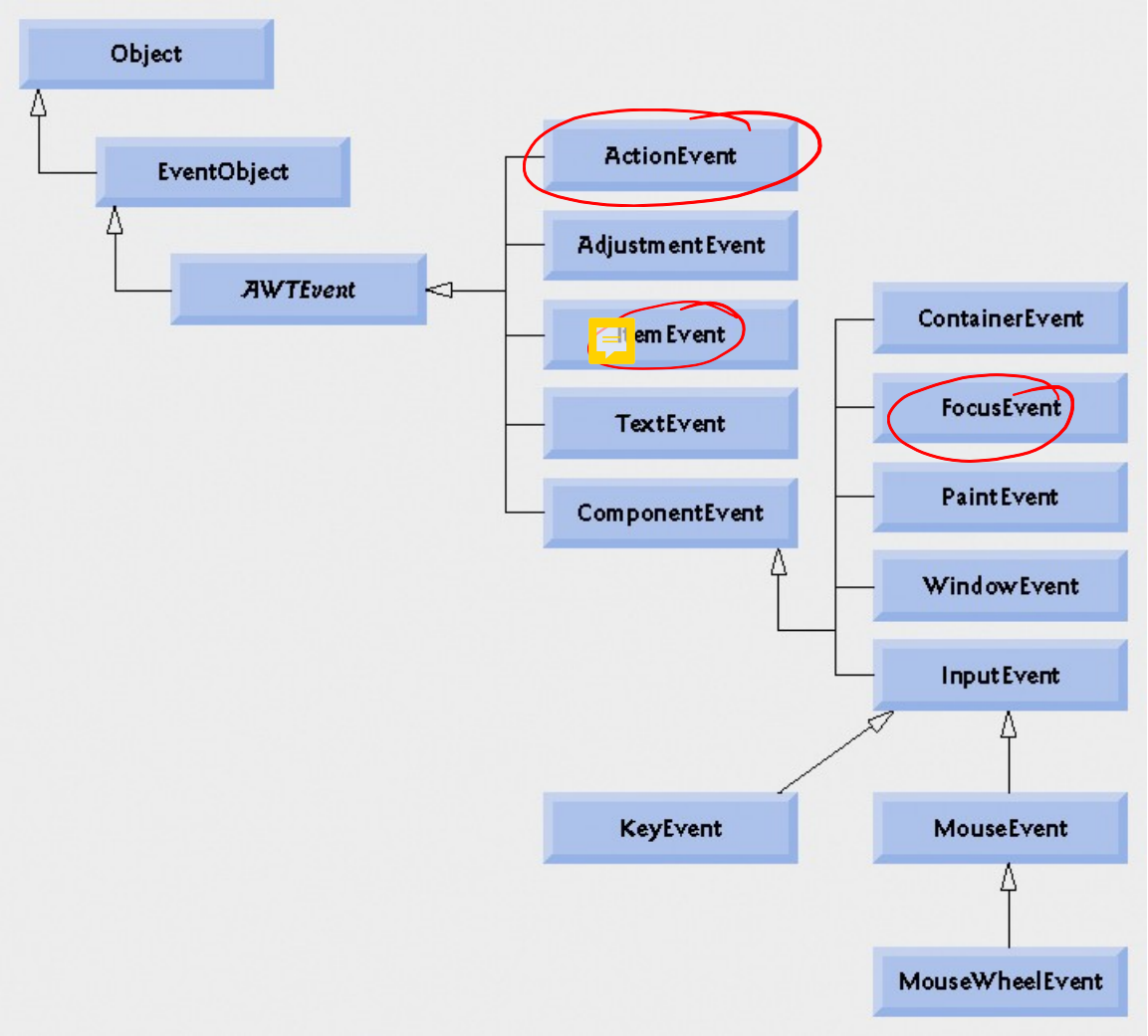


Fig. 11.11 | Some event classes of package `java.awt.event`.

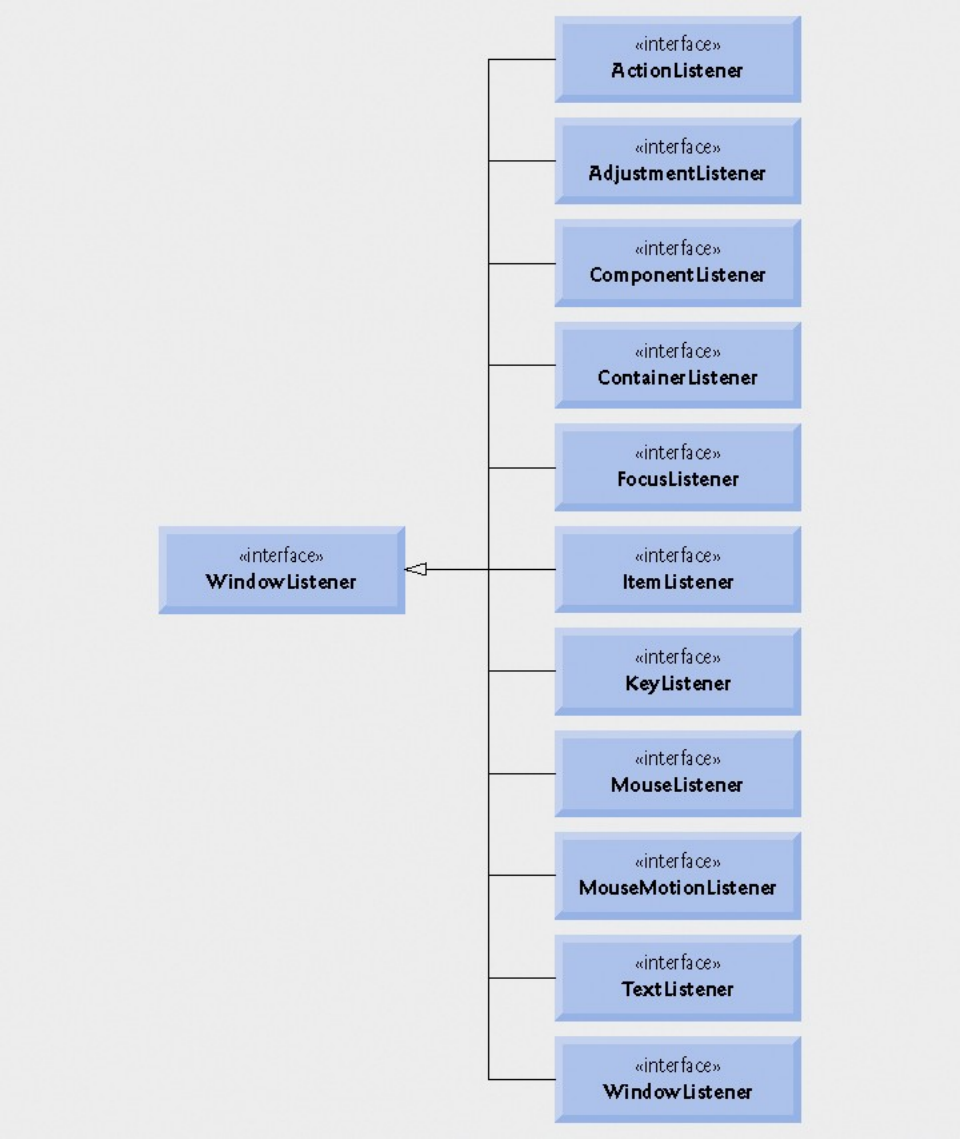


Fig. 11.12 | Some common event-listener interfaces of package `java.awt.event`.

11.7 How Event Handling Works

- **Remaining questions**
  - **How did the event handler get registered?**
  - **How does the GUI component know to call `actionPerformed` rather than some other event-handling method?**

Registering Events

- **Every** `JComponent` **has instance variable** `listenerList`
  - **Object of type** `EventListenerList`
  - **Maintains references to all its registered listeners**

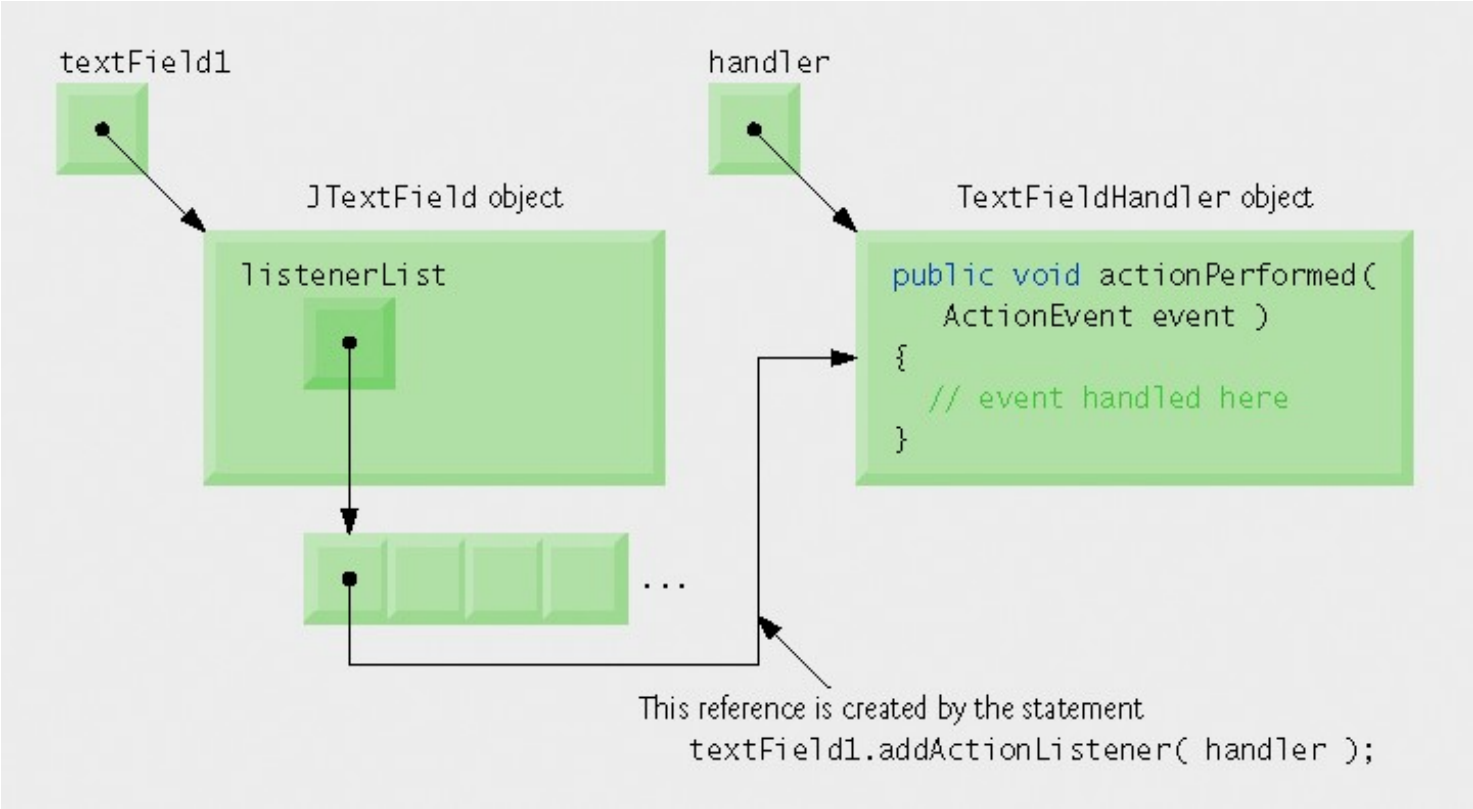


Fig. 11.13 | Event registration for `JTextField textField1`.

Event-Handler Invocation

- **Events are dispatched to only the event listeners that match the event type**
  - **Events have a unique event ID specifying the event type**
- `MouseEvents` **are handled by** `MouseListeners` **and** `MouseMotionsListeners`
- `KeyEvents` **are handled by** `KeyListeners`

11.8 `JButton`

- **Button**
  - **Component user clicks to trigger a specific action**
  - **Can be command button, check box, toggle button or radio button**
  - **Button types are subclasses of class `AbstractButton`**

Look-and-Feel Observation 11.7

Buttons typically use book-title capitalization.

11.8 JButton

- **Command button**
  - **Generates an `ActionEvent` when it is clicked**
  - **Created with class `JButton`**
  - **Text on the face of the button is called button label**

Look-and-Feel Observation 11.8

Having more than one `JButton` with the same label makes the `JButton`s ambiguous to the user. Provide a unique label for each button.

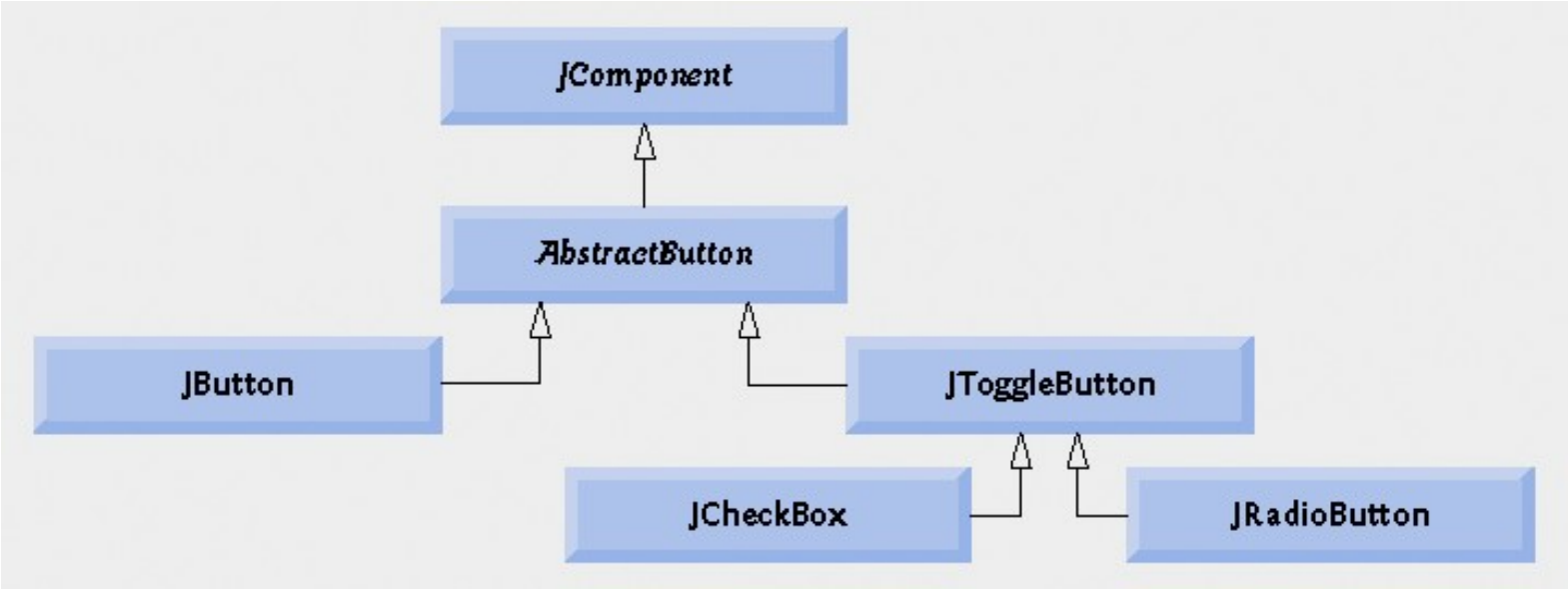


Fig. 11.14 | Swing button hierarchy.

Outline

ButtonFrame.java

(1 of 2)

```java
// Fig. 11.15: ButtonFrame.java
// Creating JButtons.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JOptionPane;

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

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

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

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

Outline

ButtonFrame.java

(2 of 2)

```
31
32       // create new ButtonHandler for button event handling
33       ButtonHandler handler = new ButtonHandler();
34       fancyJButton.addActionListener( handler );
35       plainJButton.addActionListener( handler );
36    } // end ButtonFrame constructor
37
38    // inner class for button event handling
39    private class ButtonHandler implements ActionListener
40    {
41       // handle button event
42       public void actionPerformed( ActionEvent event )
43       {
44          JOptionPane.showMessageDialog( ButtonFrame.this, String.format(
45             "You pressed: %s", event.getActionCommand() ) );
46       } // end method actionPerformed
47    } // end private inner class ButtonHandler
48 } // end class ButtonFrame
```

Name listener

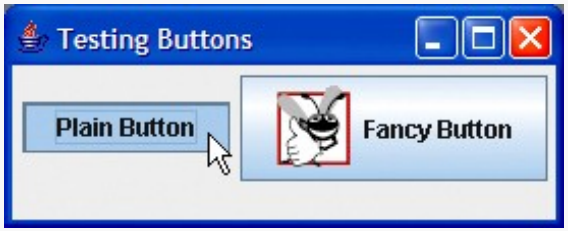
Outline

ButtonTest.java

(1 of 2)

```java
// Fig. 11.16: ButtonTest.java
// Testing ButtonFrame.
import javax.swing.JFrame;

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



Outline

`ButtonTest.java`

(2 of 2)

11.8 JButton

- JButtons can have a rollover icon
  - Appears when mouse is positioned over a button
  - Added to a JButton with method setRolloverIcon

Look-and-Feel Observation 11.9

Because class `AbstractButton` supports displaying text and images on a button, all subclasses of `AbstractButton` also support displaying text and images.

Look-and-Feel Observation 11.10

Using rollover icons for `JButton`s provides users with visual feedback indicating that when they click the mouse while the cursor is positioned over the button, an action will occur.

Software Engineering Observation 11.4

When used in an inner class, keyword `this` refers to the current inner-class object being manipulated. An inner-class method can use its outer-class object's `this` by preceding `this` with the outer-class name and a dot, as in `ButtonFrame.this`.

11.9 Buttons That Maintain State

- **State buttons**
  - **Swing contains three types of state buttons**
  - `JToggleButton`, `JCheckBox` **and** `JRadioButton`
  - `JCheckBox` **and** `JRadioButton` **are subclasses of** `JToggleButton`

11.9.1 JCheckBox

- JCheckBox
  - **Contains a check box label that appears to right of check box by default**
  - **Generates an** `ItemEvent` **when it is clicked**
    - `ItemEvents` **are handled by an** `ItemListener`
    - **Passed to method** `itemStateChanged`
  - **Method** `isSelected` **returns whether check box is selected (**`true`**) or not (**`false`**)**

```java
// Fig. 11.17: CheckBoxFrame.java
// Creating JCheckBox buttons.
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JCheckBox;

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

   // CheckBoxFrame constructor adds JCheckBoxes to JFrame
   public CheckBoxFrame()
   {
      super( "JCheckBox Test" );
      setLayout( new FlowLayout() ); // set frame layout

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

```

Outline

CheckBoxFrame .java

(1 of 3)

Outline

CheckBoxFrame.java

(2 of 3)

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

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

   // private inner class for ItemListener event handling
   private class CheckBoxHandler implements ItemListener
   {
      private int valBold = Font.PLAIN; // controls bold font style
      private int valItalic = Font.PLAIN; // controls italic font style

      // respond to checkbox events
      public void itemStateChanged( ItemEvent event )
      {
         // process bold checkbox events
         if ( event.getSource() == boldJCheckBox )
            valBold =
               boldJCheckBox.isSelected() ? Font.BOLD : Font.PLAIN;

```

PLAIN 0
BOLD 1
PLAIN ITALICS 2
BOLD 1
ITALICS 2

Outline

CheckBoxFrame.java

(3 of 3)

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

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

setEditable

P	0
B	1
I	2
U	4
S	8
O	16

0
1
3
2

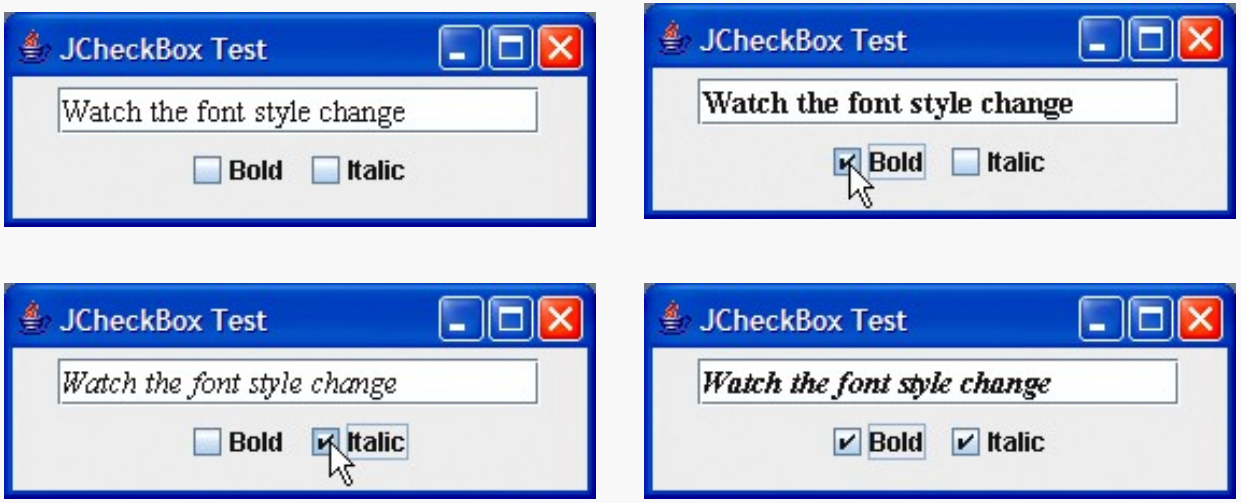
U I B

Outline

CheckBoxTest.java

```java
// Fig. 11.18: CheckBoxTest.java
// Testing CheckBoxFrame.
import javax.swing.JFrame;

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



11.9.2 JRadioButton

- JRadioButton
  - **Has two states – selected and unselected**
  - **Normally appear in a group in which only one radio button can be selected at once**
    - **Group maintained by a** `ButtonGroup` **object**
      - **Declares method** `add` **to add a** `JRadioButton` **to group**
  - **Usually represents mutually exclusive options**

Common Programming Error 11.3

Adding a `ButtonGroup` object (or an object of any other class that does not derive from `Component`) to a container results in a compilation error.

Outline

RadioButtonFrame.java

(1 of 3)

```java
// Fig. 11.19: RadioButtonFrame.java
// Creating radio buttons using ButtonGroup and JRadioButton.
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JRadioButton;
import javax.swing.ButtonGroup;

public class RadioButtonFrame extends JFrame
{
   private JTextField textField; // used to display font changes
   private Font plainFont; // font for plain text
   private Font boldFont; // font for bold text
   private Font italicFont; // font for italic text
   private Font boldItalicFont; // font for bold and italic text
   private JRadioButton plainJRadioButton; // selects plain text
   private JRadioButton boldJRadioButton; // selects bold text
   private JRadioButton italicJRadioButton; // selects italic text
   private JRadioButton boldItalicJRadioButton; // bold and italic
   private ButtonGroup radioGroup; // buttongroup to hold radio buttons

   // RadioButtonFrame constructor adds JRadioButtons to JFrame
   public RadioButtonFrame()
   {
      super( "RadioButton Test" );
      setLayout( new FlowLayout() ); // set frame layout

```

```
      textField = new JTextField( "Watch the font style change", 25 );
      add( textField ); // add textField to JFrame

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

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

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

```

Outline

RadioButtonFrame.java

(2 of 3)

```
      // 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
```

Outline

RadioButtonFrame.java

(3 of 3)

Outline

RadioButtonTest .java

```java
// Fig. 11.20: RadioButtonTest.java
// Testing RadioButtonFrame.
import javax.swing.JFrame;

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

Watch the font style change

● Plain ○ Bold ○ Italic ○ Bold/Italic

Watch the font style change

○ Plain ● Bold ○ Italic ○ Bold/Italic

Watch the font style change

○ Plain ○ Bold ● Italic ○ Bold/Italic

Watch the font style change

○ Plain ○ Bold ○ Italic ● Bold/Italic

11.10 `JComboBox` and Using an Anonymous Inner Class for Event Handling

- **Combo box**
  - **Also called a drop-down list**
  - **Implemented by class `JComboBox`**
  - **Each item in the list has an index**
  - `setMaximumRowCount` **sets the maximum number of rows shown at once**
  - `JComboBox` **provides a scrollbar and up and down arrows to traverse list**

Look-and-Feel Observation 11.11

Set the maximum row count for a `JComboBox` to a number of rows that prevents the list from expanding outside the bounds of the window in which it is used. This configuration will ensure that the list displays correctly when it is expanded by the user.

Using an Anonymous Inner Class for Event Handling

- **Anonymous inner class**
  - **Special form of inner class**
  - **Declared without a name**
  - **Typically appears inside a method call**
  - **Has limited access to local variables**

Outline

ComboBoxFrame.java

(1 of 2)

```java
// Fig. 11.21: ComboBoxFrame.java
// Using a JComboBox to select an image to display.
import java.awt.FlowLayout;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JComboBox;
import javax.swing.Icon;
import javax.swing.ImageIcon;

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

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

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

```

```java
      imagesJComboBox = new JComboBox( names ); // set up JComboBox
      imagesJComboBox.setMaximumRowCount( 3 ); // display three rows

      imagesJComboBox.addItemListener(
         new ItemListener() // anonymous inner class
         {
            // handle JComboBox event
            public void itemStateChanged( ItemEvent event )
            {
               // determine whether check box selected
               if ( event.getStateChange() == ItemEvent.SELECTED )
                  label.setIcon( icons[
                     imagesJComboBox.getSelectedIndex() ] );
            } // end method itemStateChanged
         } // end anonymous inner class
      ); // end call to addItemListener

      add( imagesJComboBox ); // add combobox to JFrame
      label = new JLabel( icons[ 0 ] ); // display first icon
      add( label ); // add label to JFrame
   } // end ComboBoxFrame constructor
} // end class ComboBoxFrame
```

Outline

ComboBoxrame.java

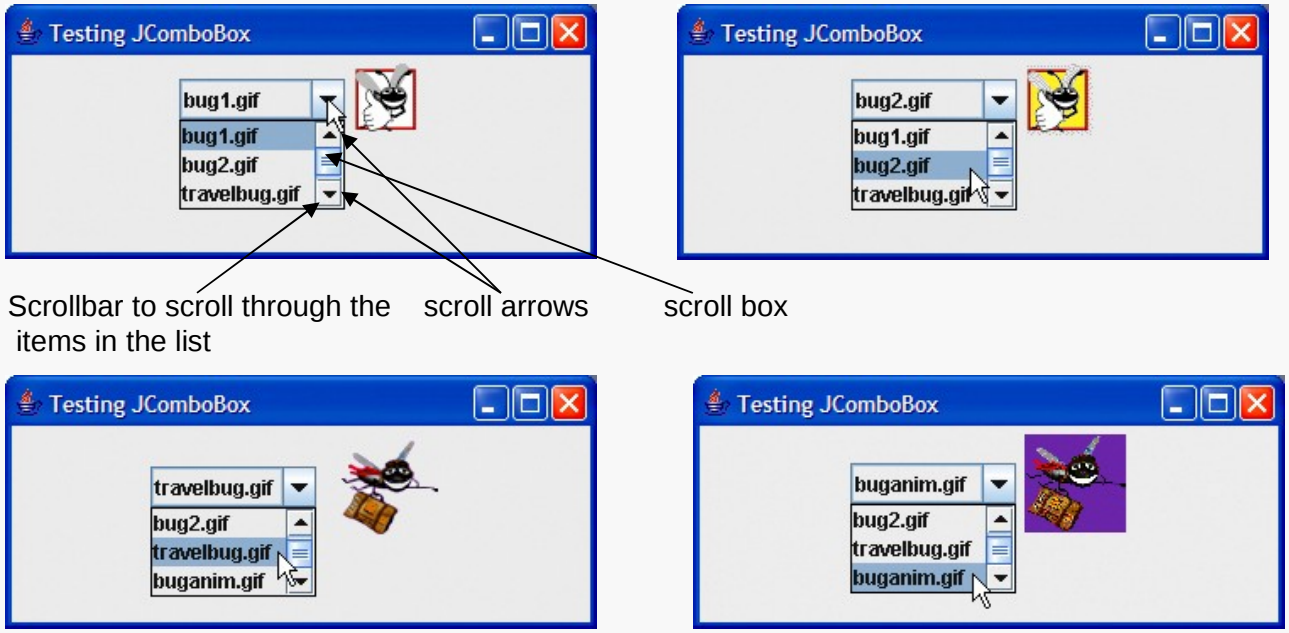
(2 of 2)

Outline

ComboBoxTest.java

```java
// Fig. 11.22: ComboBoxTest.java
// Testing ComboBoxFrame.
import javax.swing.JFrame;

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



Software Engineering Observation 11.5

An anonymous inner class declared in a method can access the instance variables and methods of the top-level class object that declared it, as well as the method's `final` local variables, but cannot access the method's non-`final` variables.

Software Engineering Observation 11.6

Like any other class, when an anonymous inner class implements an interface, the class must implement every method in the interface.

11.11 `JList`

- **List**
  - **Displays a series of items from which the user may select one or more items**
  - **Implemented by class `JList`**
  - **Allows for single-selection lists or multiple-selection lists**
  - **A `ListSelectionEvent` occurs when an item is selected**
    - **Handled by a `ListSelectionListener` and passed to method `valueChanged`**

Outline

ListFrame.java

(1 of 2)

```java
// Fig. 11.23: ListFrame.java
// Selecting colors from a JList.
import java.awt.FlowLayout;
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.event.ListSelectionListener;
import javax.swing.event.ListSelectionEvent;
import javax.swing.ListSelectionModel;

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

   // ListFrame constructor add JScrollPane containing JList to JFrame
   public ListFrame()
   {
      super( "List Test" );
      setLayout( new FlowLayout() ); // set frame layout

```

```java
      colorJList = new JList( colorNames ); // create with colorNames
      colorJList.setVisibleRowCount( 5 ); // display five rows at once

      // do not allow multiple selections
      colorJList.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );

      // add a JScrollPane containing JList to frame
      add( new JScrollPane( colorJList ) );

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

Outline

ListFrame.java

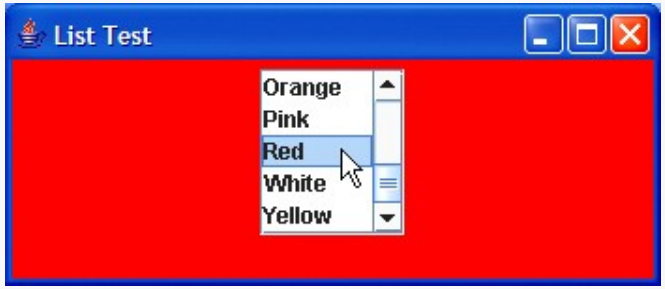
(2 of 2)

Outline

ListTest.java

```java
// Fig. 11.24: ListTest.java
// Selecting colors from a JList.
import javax.swing.JFrame;

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



11.12 Multiple-Selection Lists

- **Multiple-selection list**
  - **Enables users to select many items**
  - **Single interval selection allows only a continuous range of items**
  - **Multiple interval selection allows any set of elements to be selected**

Outline

Multiple SelectionFrame .java

(1 of 3)

```java
// Fig. 11.25: MultipleSelectionFrame.java
// Copying items from one List to another.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JButton;
import javax.swing.JScrollPane;
import javax.swing.ListSelectionModel;

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

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

```

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

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

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

```

Outline

Multiple SelectionFrame .java

(2 of 3)

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

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

Outline

MultipleSelectionFrame.java

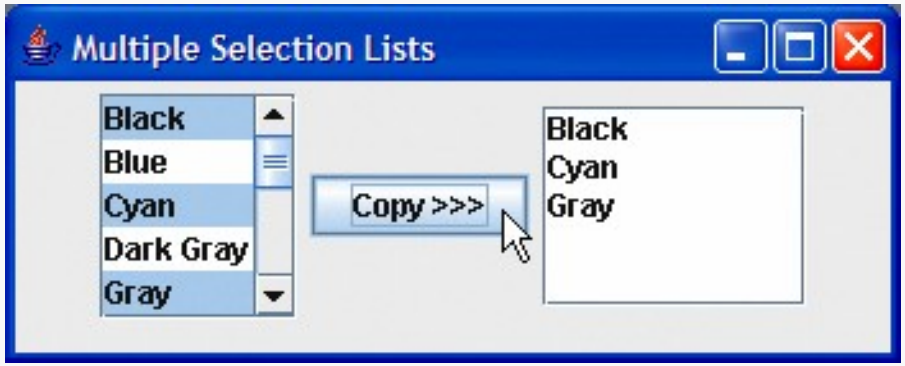
(3 of 3)

Outline

Multiple SelectionTest .java

```java
// Fig. 11.26: MultipleSelectionTest.java
// Testing MultipleSelectionFrame.
import javax.swing.JFrame;

public class MultipleSelectionTest
{
   public static void main( String args[] )
   {
      MultipleSelectionFrame multipleSelectionFrame =
         new MultipleSelectionFrame();
      multipleSelectionFrame.setDefaultCloseOperation(
         JFrame.EXIT_ON_CLOSE );
      multipleSelectionFrame.setSize( 350, 140 ); // set frame size
      multipleSelectionFrame.setVisible( true ); // display frame
   } // end main
} // end class MultipleSelectionTest
```



11.13 Mouse Event Handling

- **Mouse events**
  - **Create a** `MouseEvent` **object**
  - **Handled by** `MouseListener`**s and** `MouseMotionListener`**s**
  - `MouseInputListener` **combines the two interfaces**
  - **Interface** `MouseWheelListener` **declares method** `mouseWheelMoved` **to handle** `MouseWheelEvent`**s**

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.

Fig. 11.27 | `MouseListener` and `MouseMotionListener` interface methods. (Part 1 of 2.)

MouseListener and MouseMotionListener interface methods	
`public void mouseExited( MouseEvent event )`	Called when the mouse cursor leaves the bounds of a component.
Methods of interface `MouseMotionListener`	
`public void mouseDragged( MouseEvent event )`	Called when the mouse button is pressed while the mouse cursor is on a component and the mouse is moved while the mouse button remains pressed. This event is always preceded by a call to `mousePressed`. All drag events are sent to the component on which the user began to drag the mouse.
`public void mouseMoved( MouseEvent event )`	Called when the mouse is moved when the mouse cursor is on a component. All move events are sent to the component over which the mouse is currently positioned.

Fig. 11.27 | `MouseListener` and `MouseMotionListener` interface methods. (Part 2 of 2.)

Look-and-Feel Observation 11.12

Method calls to `mouseDragged` and `mouseReleased` are sent to the `MouseMotionListener` for the `Component` on which a mouse drag operation started. Similarly, the `mouseReleased` method call at the end of a drag operation is sent to the `MouseListener` for the `Component` on which the drag operation started.

Outline

MouseTracker Frame.java

(1 of 4)

```java
// Fig. 11.28: MouseTrackerFrame.java
// Demonstrating mouse events.
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

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

   // MouseTrackerFrame constructor sets up GUI and
   // registers mouse event handlers
   public MouseTrackerFrame()
   {
      super( "Demonstrating Mouse Events" );

      mousePanel = new JPanel(); // create panel
      mousePanel.setBackground( Color.WHITE ); // set background color
      add( mousePanel, BorderLayout.CENTER ); // add panel to JFrame

      statusBar = new JLabel( "Mouse outside JPanel" );
      add( statusBar, BorderLayout.SOUTH ); // add label to JFrame

```

```java
      // create and register listener for mouse and mouse motion events
      MouseHandler handler = new MouseHandler();
      mousePanel.addMouseListener( handler );
      mousePanel.addMouseMotionListener( handler );
   } // end MouseTrackerFrame constructor

   private class MouseHandler implements MouseListener,
      MouseMotionListener
   {
      // MouseListener event handlers
      // handle event when mouse released immediately after press
      public void mouseClicked( MouseEvent event )
      {
         statusBar.setText( String.format( "Clicked at [%d, %d]",
            event.getX(), event.getY() ) );
      } // end method mouseClicked

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

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

Outline

MouseTrackerFrame.java

(2 of 4)

```java

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

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

```

Outline

MouseTrackerFrame.java

(3 of 4)

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

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

Outline

MouseTrackerFrame.java

(4 of 4)

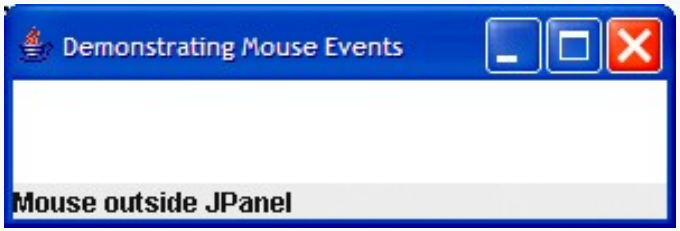
Outline

MouseTracker Frame.java

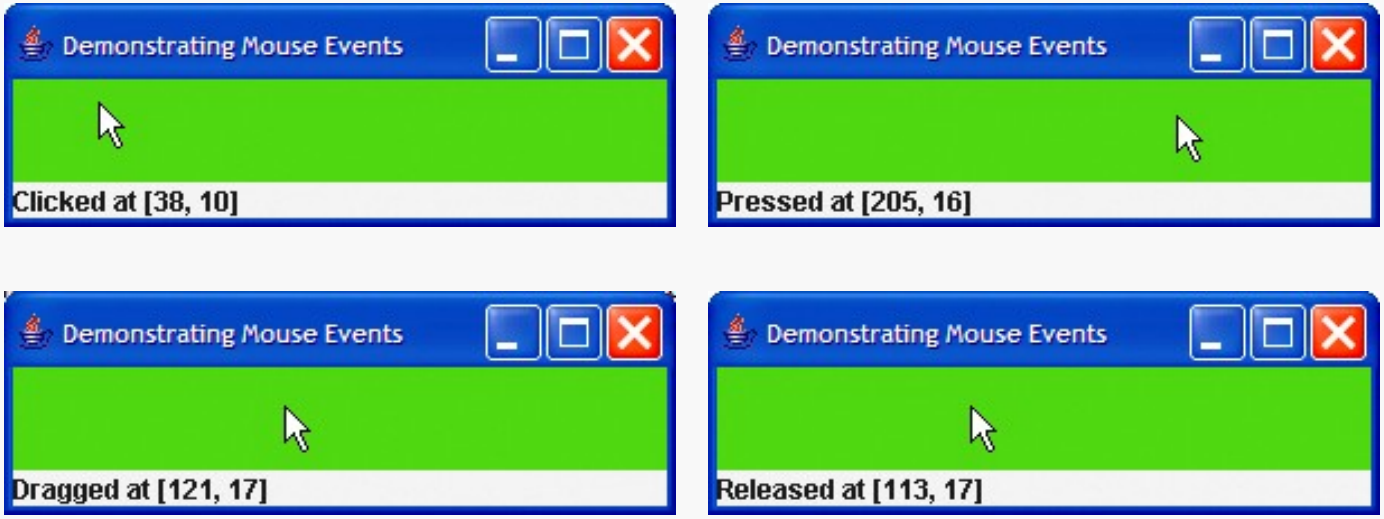
(1 of 2)

```java
// Fig. 11.29: MouseTrackerFrame.java
// Testing MouseTrackerFrame.
import javax.swing.JFrame;

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



Outline



`MouseTrackerFrame.java`

(2 of 2)

11.14 Adapter Classes

- **Adapter class**
  - **Implements event listener interface**
  - **Provides default implementation for all event-handling methods**

Software Engineering Observation 11.7

When a class implements an interface, the class has an "*is a*" relationship with that interface. All direct and indirect subclasses of that class inherit this interface. Thus, an object of a class that extends an event-adapter class *is an* object of the corresponding event-listener type (e.g., an object of a subclass of `MouseAdapter` *is a* `MouseListener`).

Extending `MouseAdapter`

- `MouseAdapter`
  - **Adapter class for** `MouseListener` **and** `MouseMotionListener` **interfaces**
  - **Extending class allows you to override only the methods you wish to use**

Common Programming Error 11.4

If you extend an adapter class and misspell the name of the method you are overriding, your method simply becomes another method in the class. This is a logic error that is difficult to detect, since the program will call the empty version of the method inherited from the adapter class.

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

Fig. 11.30 | Event-adapter classes and the interfaces they implement in package `java.awt.event`.

Outline

MouseDetailsFrame.java

(1 of 2)

```
// Fig. 11.31: MouseDetailsFrame.java
// Demonstrating mouse clicks and distinguishing between mouse buttons.
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; // String representing
   private JLabel statusBar; // JLabel that appears at bottom of window

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

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

```

Register event handler

Outline

MouseDetailsFrame.java

(2 of 2)

```
   // 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
   } // end private inner class MouseClickHandler
} // end class MouseDetailsFrame
```

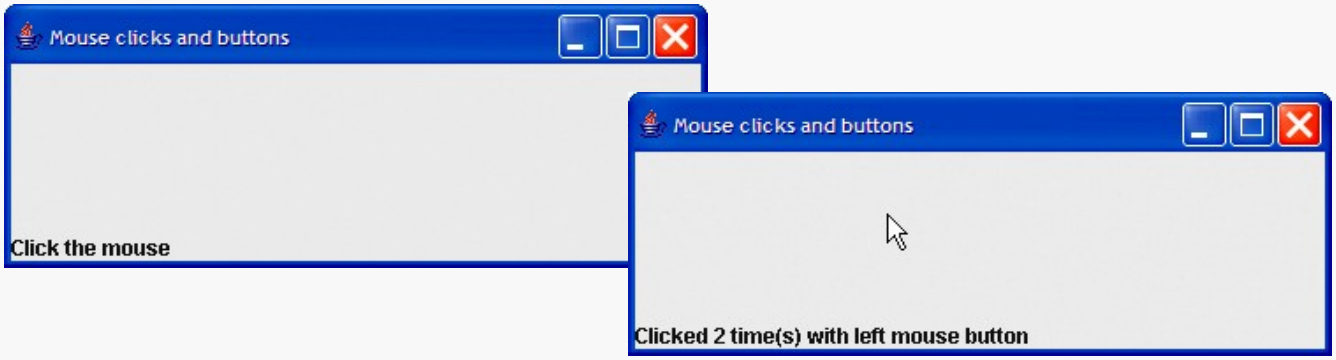
Outline

MouseDetails.java

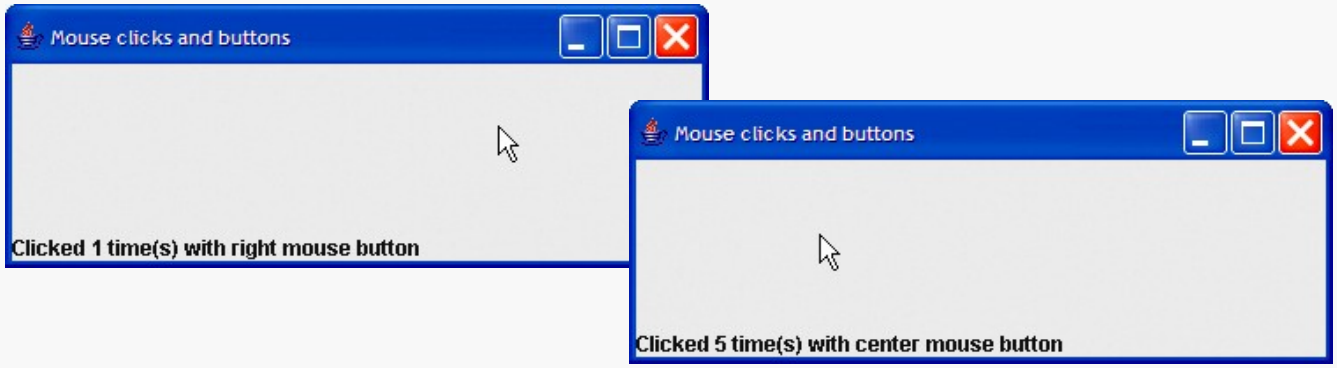
(1 of 2)

```java
// Fig. 11.32: MouseDetails.java
// Testing MouseDetailsFrame.
import javax.swing.JFrame;

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



Outline



`MouseDetails.java`

(2 of 2)

11.15 `JPanel` Subclass for Drawing with the Mouse

- **Overriding class `JPanel`**
  - **Provides a dedicated drawing area**

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

Fig. 11.33 | `InputEvent` methods that help distinguish among left-, center- and right-mouse-button clicks.

Method `paintComponent`

- **Method** `paintComponent`
  - **Draws on a Swing component**
  - **Overriding method allows you to create custom drawings**
  - **Must call superclass method first when overridden**

Look-and-Feel Observation 11.13

Most Swing GUI components can be transparent or opaque. If a Swing GUI component is opaque, its background will be cleared when its `paintComponent` method is called. Only opaque components can display a customized background color. `JPanel` objects are opaque by default.

Error-Prevention Tip 11.1

In a `JComponent` subclass's `paintComponent` method, the first statement should always be a call to the superclass's `paintComponent` method to ensure that an object of the subclass displays correctly.

Common Programming Error 11.5

If an overridden `paintComponent` method does not call the superclass's version, the subclass component may not display properly. If an overridden `paintComponent` method calls the superclass's version after other drawing is performed, the drawing will be erased.

Defining the Custom Drawing Area

- **Customized subclass of `JPanel`**
  - **Provides custom drawing area**
  - **Class `Graphics` is used to draw on Swing components**
  - **Class `Point` represents an *x*-*y* coordinate**

Outline

PaintPanel.java

(1 of 2)

```java
// Fig. 11.34: PaintPanel.java
// Using class MouseMotionAdapter.
import java.awt.Point;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;
import javax.swing.JPanel;

public class PaintPanel extends JPanel
{
   private int pointCount = 0; // count number of points

   // array of 10000 java.awt.Point references
   private Point points[] = new Point[ 10000 ];

   // set up GUI and register mouse event handler
   public PaintPanel()
   {
      // handle frame mouse motion event
      addMouseMotionListener(

```

Create array of `Points`

```
22          new MouseMotionAdapter() // anonymous inner class
23          {
24             // store drag coordinates and repaint
25             public void mouseDragged( MouseEvent event )
26             {
27                if ( pointCount < points.length )
28                {
29                   points[ pointCount ] = event.getPoint(); // find point
30                   pointCount++; // increment number of points in array
31                   repaint(); // repaint JFrame
32                } // end if
33             } // end method mouseDragged
34          } // end anonymous inner class
35       ); // end call to addMouseMotionListener
36    } // end PaintPanel constructor
37
38    // draw oval in a 4-by-4 bounding box at specified location on window
39    public void paintComponent( Graphics g )
40    {
41       super.paintComponent( g ); // clears drawing area
42
43       // draw all points in array
44       for ( int i = 0; i < pointCount; i++ )
45          g.fillOval( points[ i ].x, points[ i ].y, 4, 4 );
46    } // end method paintComponent
47 } // end class PaintPanel
```

Anonymous inner class for event handling

Override `mouseDragged` method

Get location of mouse cursor

Repaint the `JFrame`

Get the *x* and *y*-coordinates of the `Point`

PaintPanel.java

(2 of 2)

Look-and-Feel Observation 11.14

Calling `repaint` for a Swing GUI component indicates that the component should be refreshed on the screen as soon as possible. The background of the GUI component is cleared only if the component is opaque. `JComponent` method `setOpaque` can be passed a `boolean` argument indicating whether the component is opaque (`true`) or transparent (`false`).

Look-and-Feel Observation 11.15

Drawing on any GUI component is performed with coordinates that are measured from the upper-left corner (0, 0) of that GUI component, not the upper-left corner of the screen.

Outline

Painter.java

(1 of 2)

```java
// Fig. 11.35: Painter.java
// Testing PaintPanel.
import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class Painter
{
   public static void main( String args[] )
   {
      // create JFrame
      JFrame application = new JFrame( "A simple paint program" );

      PaintPanel paintPanel = new PaintPanel(); // create paint panel
      application.add( paintPanel, BorderLayout.CENTER ); // in center

      // create a label and place it in SOUTH of Bor
      application.add( new JLabel( "Drag the mouse to draw" ),
         BorderLayout.SOUTH );

      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      application.setSize( 400, 200 ); // set frame size
      application.setVisible( true ); // display frame
   } // end main
} // end class Painter
```

Create instance of custom drawing panel

Outline

`Painter.java`

(2 of 2)

11.16 Key-Event Handling

- `KeyListener` **interface**
  - **For handling** `KeyEvents`
  - **Declares methods** `keyPressed`, `keyReleased` **and** `keyTyped`

Outline

KeyDemoFrame.java

(1 of 3)

```java
// Fig. 11.36: KeyDemoFrame.java
// Demonstrating keystroke events.
import java.awt.Color;
import java.awt.event.KeyListener;
import java.awt.event.KeyEvent;
import javax.swing.JFrame;
import javax.swing.JTextArea;

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

   // KeyDemoFrame constructor
   public KeyDemoFrame()
   {
      super( "Demonstrating Keystroke Events" );

      textArea = new JTextArea( 10, 15 ); // set up JTextArea
      textArea.setText( "Press any key on the keyboard..." );
      textArea.setEnabled( false ); // disable textarea
      textArea.setDisabledTextColor( Color.BLACK ); // set text color
      add( textArea ); // add textarea to JFrame

      addKeyListener( this ); // allow frame to process key events
   } // end KeyDemoFrame constructor

```

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

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

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

```

Outline

KeyDemoFrame.java

(2 of 3)

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

      String temp = event.getKeyModifiersText( event.getModifiers() );

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

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

Outline

KeyDemoFrame.java

(3 of 3)

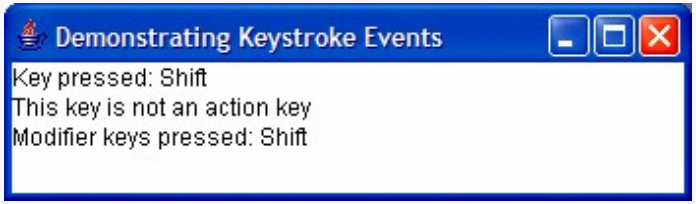
Outline

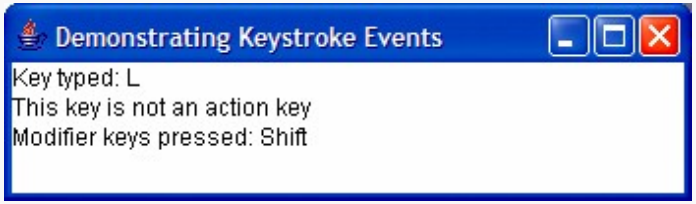
KeyDemo.java

(1 of 2)

```java
// Fig. 11.37: KeyDemo.java
// Testing KeyDemoFrame.
import javax.swing.JFrame;

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

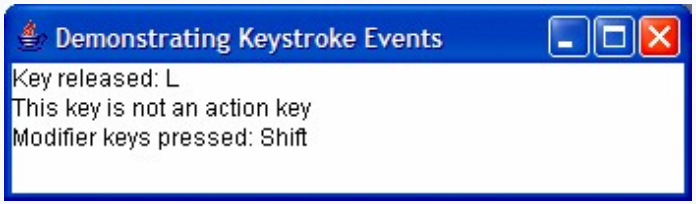




Outline

`KeyDemo.java`

(1 of 2)



11.17 Layout Managers

- **Layout managers**
  - **Provided to arrange GUI components in a container**
  - **Provide basic layout capabilities**
  - **Implement the interface** `LayoutManager`

Look-and-Feel Observation 11.16

Most Java programming environments provide GUI design tools that help a programmer graphically design a GUI; the design tools then write the Java code to create the GUI. Such tools often provide greater control over the size, position and alignment of GUI components than do the built-in layout managers.

Look-and-Feel Observation 11.17

It is possible to set a `Container`'s layout to `null`, which indicates that no layout manager should be used. In a `Container` without a layout manager, the programmer must position and size the components in the given container and take care that, on resize events, all components are repositioned as necessary. A component's resize events can be processed by a `ComponentListener`.

11.17.1 `FlowLayout`

- `FlowLayout`
  - **Simplest layout manager**
  - **Components are placed left to right in order they are added**
  - **Components can be left aligned, centered or right aligned**

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 `JFrame`s (and other windows). Arranges the components into five areas: `NORTH`, `SOUTH`, `EAST`, `WEST` and `CENTER`.
`GridLayout`	Arranges the components into rows and columns.

Fig. 11.38 | Layout managers.

Outline

FlowLayoutFrame.java

(1 of 3)

```
// Fig. 11.39: FlowLayoutFrame.java
// Demonstrating FlowLayout alignments.
import java.awt.FlowLayout;
import java.awt.Container;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;

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

   // set up GUI and register button listeners
   public FlowLayoutFrame()
   {
      super( "FlowLayout Demo" );

      layout = new FlowLayout(); // create FlowLayout
      container = getContentPane(); // get container to layout
      setLayout( layout ); // set frame layout

```

```
      // set up leftJButton and register listener
      leftJButton = new JButton( "Left" ); // create Left button
      add( leftJButton ); // add Left button to frame
      leftJButton.addActionListener(

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

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

      // set up centerJButton and register listener
      centerJButton = new JButton( "Center" ); // create Center button
      add( centerJButton ); // add Center button to frame
      centerJButton.addActionListener(

         new ActionListener() // anonymous inner class
         {
            // process centerJButton event
            public void actionPerformed( ActionEvent event )
            {
               layout.setAlignment( FlowLayout.CENTER );

```

Outline

FlowLayoutFrame.java

(2 of 3)

Outline

FlowLayoutFrame.java

(3 of 3)

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

   // set up rightJButton and register listener
   rightJButton = new JButton( "Right" ); // create Right button
   add( rightJButton ); // add Right button to frame
   rightJButton.addActionListener(

      new ActionListener() // anonymous inner class
      {
         // process rightJButton event
         public void actionPerformed( ActionEvent event )
         {
            layout.setAlignment( FlowLayout.RIGHT );

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

Outline

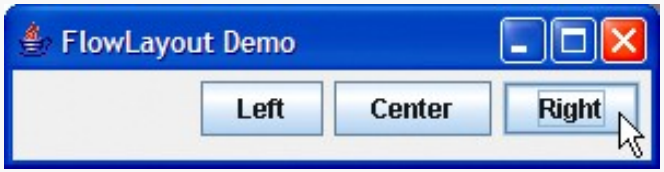
FlowLayoutDemo.java

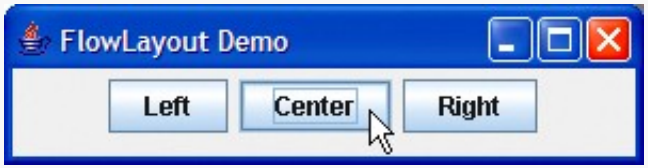
(1 of 2)

```java
// Fig. 11.40: FlowLayoutDemo.java
// Testing FlowLayoutFrame.
import javax.swing.JFrame;

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

Outline





`FlowLayoutDemo.java`

(2 of 2)

11.17.2 `BorderLayout`

- `BorderLayout`
  - **Arranges components into five regions – north, south, east, west and center**
  - **Implements interface** `LayoutManager2`
  - **Provides horizontal gap spacing and vertical gap spacing**

Look-and-Feel Observation 11.18

Each container can have only one layout manager. Separate containers in the same application can use different layout managers.

Look-and-Feel Observation 11.19

If no region is specified when adding a `Component` to a `BorderLayout`, the layout manager assumes that the `Component` should be added to region `BorderLayout.CENTER`.

Common Programming Error 11.6

When more than one component is added to a region in a `BorderLayout`, only the last component added to that region will be displayed. There is no error that indicates this problem.

Outline

BorderLayoutFrame.java

(1 of 2)

```java
// Fig. 11.41: BorderLayoutFrame.java
// Demonstrating BorderLayout.
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;

public class BorderLayoutFrame extends JFrame implements ActionListener
{
   private JButton buttons[]; // array of buttons to hide portions
   private final String names[] = { "Hide North", "Hide South",
      "Hide East", "Hide West", "Hide Center" };
   private BorderLayout layout; // borderlayout object

   // set up GUI and event handling
   public BorderLayoutFrame()
   {
      super( "BorderLayout Demo" );

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

      // create JButtons and register listeners for them
      for ( int count = 0; count < names.length; count++ )
      {
         buttons[ count ] = new JButton( names[ count ] );
         buttons[ count ].addActionListener( this );
      } // end for
```

```java

      add( buttons[ 0 ], BorderLayout.NORTH ); // add button to north
      add( buttons[ 1 ], BorderLayout.SOUTH ); // add button to south
      add( buttons[ 2 ], BorderLayout.EAST ); // add button to east
      add( buttons[ 3 ], BorderLayout.WEST ); // add button to west
      add( buttons[ 4 ], BorderLayout.CENTER ); // add button to center
   } // end BorderLayoutFrame constructor

   // handle button events
   public void actionPerformed( ActionEvent event )
   {
      // check event source and 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
      } // end for

      layout.layoutContainer( getContentPane() ); // layout content pane
   } // end method actionPerformed
} // end class BorderLayoutFrame
```

Outline

BorderLayoutFrame.java

(2 of 2)

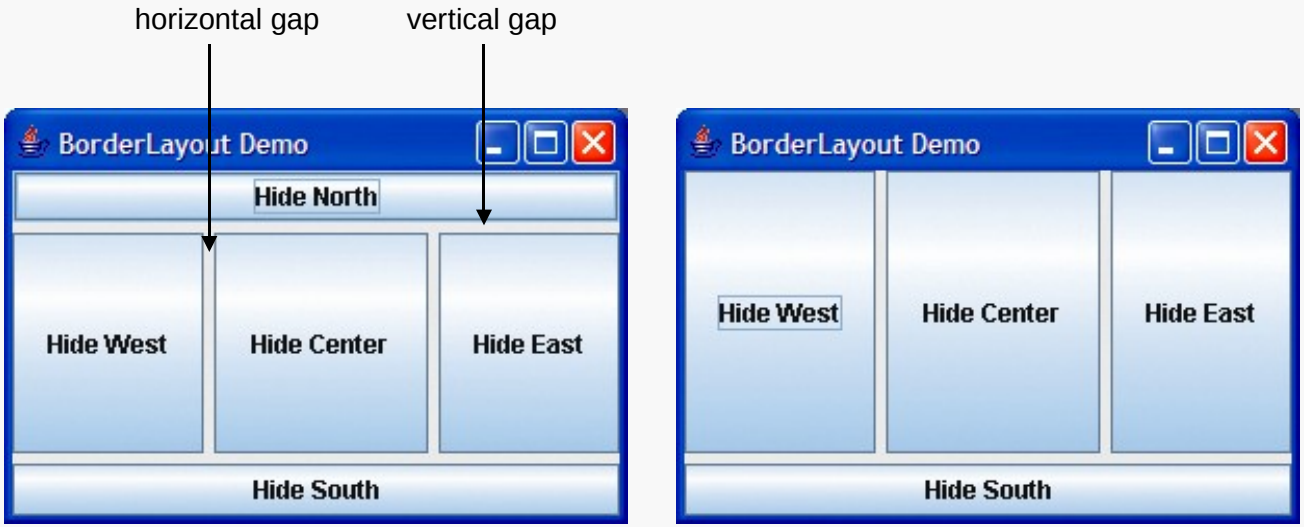
Outline

BorderLayoutDemo.java

(1 of 2)

```java
// Fig. 11.42: BorderLayoutDemo.java
// Testing BorderLayoutFrame.
import javax.swing.JFrame;

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



Outline

`BorderLayout Demo.java`

(2 of 2)

11.17.3 GridLayout

- GridLayout
  - **Divides container into a grid**
  - **Every component has the same width and height**

Outline

GridLayoutFrame.java

(1 of 2)

```java
// Fig. 11.43: GridLayoutFrame.java
// Demonstrating GridLayout.
import java.awt.GridLayout;
import java.awt.Container;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;

public class GridLayoutFrame extends JFrame implements ActionListener
{
   private JButton buttons[]; // array of buttons
   private 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

```

```java
      for ( int count = 0; count < names.length; count++ )
      {
         buttons[ count ] = new JButton( names[ count ] );
         buttons[ count ].addActionListener( this ); // register listener
         add( buttons[ count ] ); // add button to JFrame
      } // end for
   } // end GridLayoutFrame constructor

   // handle button events by toggling between layouts
   public void actionPerformed( ActionEvent event )
   {
      if ( toggle )
         container.setLayout( gridLayout2 ); // set layout to second
      else
         container.setLayout( gridLayout1 ); // set layout to first

      toggle = !toggle; // set toggle to opposite value
      container.validate(); // re-layout container
   } // end method actionPerformed
} // end class GridLayoutFrame
```

Outline

GridLayoutFrame.java

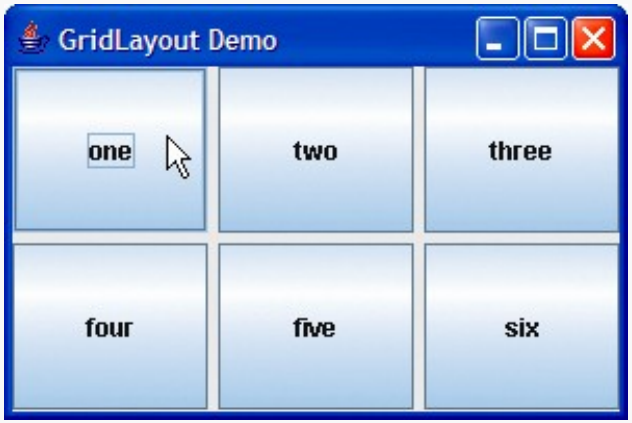
(2 of 2)

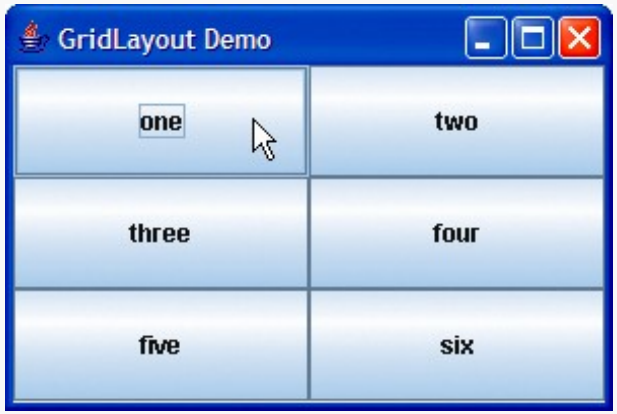
Outline

GridLayoutDemo.java

```
// Fig. 11.44: GridLayoutDemo.java
// Testing GridLayoutFrame.
import javax.swing.JFrame;

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





11.18 Using Panels to Manage More Complex Layouts

- **Complex GUIs often require multiple panels to arrange their components properly**

Outline

PanelFrame.java

(1 of 2)

```
// Fig. 11.45: PanelFrame.java
// Using a JPanel to help lay out components.
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

   // no-argument constructor
   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 ) );

```

Declare a `JPanel` to hold buttons

Create `JPanel`

Set layout

Outline

```
      // 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
      } // end for

      add( buttonJPanel, BorderLayout.SOUTH ); // add panel to JFrame
   } // end PanelFrame constructor
} // end class PanelFrame
```

Add button to panel

Add panel to application

rame.java

(2 of 2)

Outline

PanelDemo.java

```java
// Fig. 11.46: PanelDemo.java
// Testing PanelFrame.
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 ); // set frame size
      panelFrame.setVisible( true ); // display frame
   } // end main
} // end class PanelDemo
```

11.19 JTextArea

- JTextArea
  - **Provides an area for manipulating multiple lines of text**
- Box **container**
  - **Subclass of** Container
  - **Uses a** BoxLayout **layout manager**

Look-and-Feel Observation 11.20

To provide line-wrapping functionality for a `JTextArea`, invoke `JTextArea` method `setLineWrap` with a `true` argument.

Outline

TextAreaFrame.java

(1 of 2)

```java
// Fig. 11.47: TextAreaFrame.java
// Copying selected text from one textarea to another.
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.Box;
import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.JButton;
import javax.swing.JScrollPane;

public class TextAreaFrame extends JFrame
{
   private JTextArea textArea1; // displays demo string
   private JTextArea textArea2; // highlighted text is copied here
   private JButton copyJButton; // initiates copying of text

   // no-argument constructor
   public TextAreaFrame()
   {
      super( "TextArea Demo" );
      Box box = Box.createHorizontalBox(); // create box
      String demo = "This is a demo string to\n" +
         "illustrate copying text\nfrom one textarea
         "another textarea using an\nexternal event\n

      textArea1 = new JTextArea( demo, 10, 15 ); // create textarea1
      box.add( new JScrollPane( textArea1 ) ); // add scrollpane

```

Declare `JTextArea` instance variables

Create a `Box` container

Create text area and add to box

Outline

TextAreaFrame.java

(2 of 2)

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

         new ActionListener() // anonymous inner class
         {
            // set text in textArea2 to selected text from textArea1
            public void actionPerformed( ActionEvent event )
            {
               textArea2.setText( textArea1.getSelectedText() );
            } // end method actionPerformed
         } // end anonymous inner class
      ); // end call to addActionListener

      textArea2 = new JTextArea( 10, 15 ); // create second textarea
      textArea2.setEditable( false ); // disable editing
      box.add( new JScrollPane( textArea2 ) ); // add scrollpane

      add( box ); // add box to frame
   } // end TextAreaFrame constructor
} // end class TextAreaFrame
```

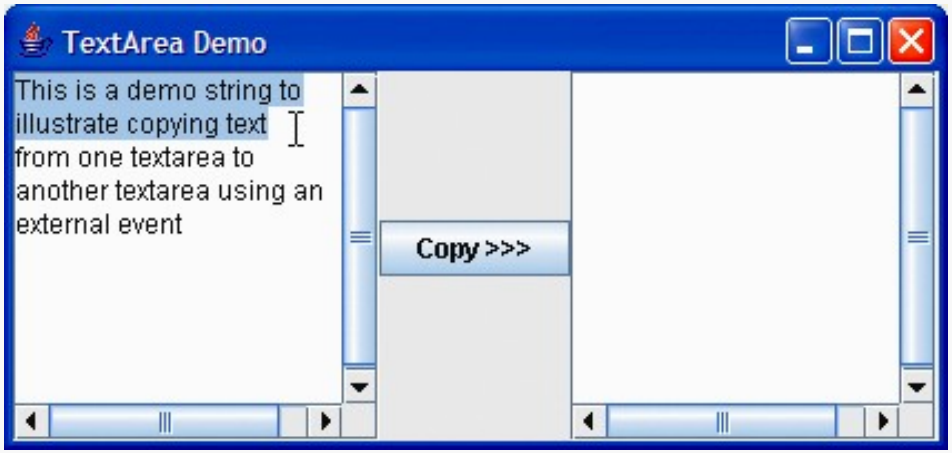
Add button to box

Copy selected text from one text area to the other

Create second text area and add it to box

```java
// Fig. 11.48: TextAreaDemo.java
// Copying selected text from one textarea to another.
import javax.swing.JFrame;

public class TextAreaDemo
{
   public static void main( String args[] )
   {
      TextAreaFrame textAreaFrame = new TextAreaFrame();
      textAreaFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      textAreaFrame.setSize( 425, 200 ); // set frame size
      textAreaFrame.setVisible( true ); // display frame
   } // end main
} // end class TextAreaDemo
```

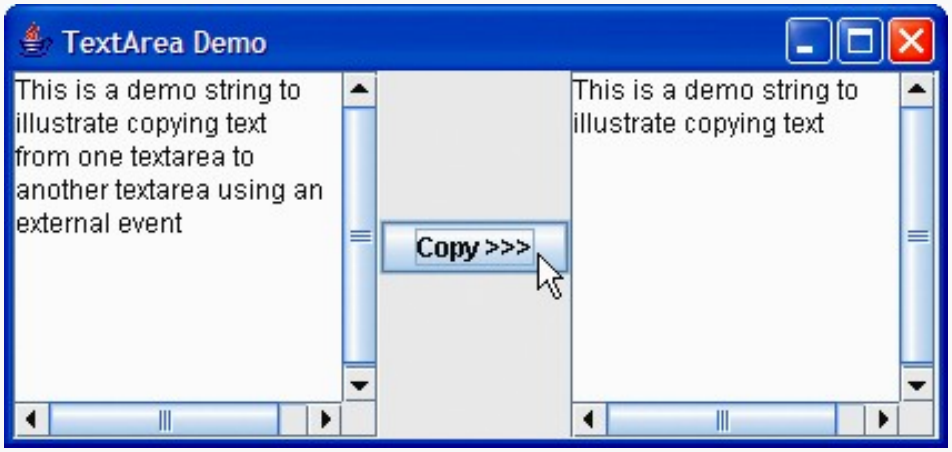


Outline

TextAreaDemo.java

(1 of 2)

Outline



`TextAreaDemo.java`

(2 of 2)

JScrollPane Scrollbar Policies

- JScrollPane has scrollbar policies
  - Horizontal policies
    - Always (HORIZONTAL_SCROLLBAR_ALWAYS)
    - As needed (HORIZONTAL_SCROLLBAR_AS_NEEDED)
    - Never (HORIZONTAL_SCROLLBAR_NEVER)
  - Vertical policies
    - Always (VERTICAL_SCROLLBAR_ALWAYS)
    - As needed (VERTICAL_SCROLLBAR_AS_NEEDED)
    - Never (VERTICAL_SCROLLBAR_NEVER)