

Lesson 12

GUI Components: Part 1

Assoc. Prof. Marenglen Biba

In this Chapter you'll learn:

- The design principles of graphical user interfaces (GUIs).
 - ~~■ How to use Java's new, elegant, cross-platform Nimbus look and feel.~~
 - 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

Next class



14.1 Introduction

~~**14.2** Java's New Nimbus Look-and-Feel~~

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

14.4 Overview of Swing Components

14.5 Displaying Text and Images in a Window

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

14.7 Common GUI Event Types and Listener Interfaces

14.8 How Event Handling Works

14.9 `JButton`

14.10 Buttons That Maintain State

14.10.1 `JCheckBox`

14.10.2 `JRadioButton`

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

14.12 JList

14.13 Multiple-Selection Lists

14.14 Mouse Event Handling

14.15 Adapter Classes

~~**14.16** JPanel Subclass for Drawing with the Mouse~~

14.17 Key Event Handling

14.18 Introduction to Layout Managers

14.18.1 FlowLayout

14.18.2 BorderLayout

14.18.3 GridLayout

14.19 Using Panels to Manage More Complex Layouts

14.20 JTextArea

14.21 Wrap-Up

← Next class

14.1 Introduction

- ▶ A **graphical user interface (GUI)** presents a user-friendly mechanism for interacting with an application.
 - Pronounced “GOO-ee”
 - Gives an application a distinctive “look” and “feel.”
 - Consistent, intuitive user-interface components give users a sense of familiarity
 - Learn new applications more quickly and use them more productively.

14.1 Introduction (cont.)

- ▶ Built from GUI components.
 - Sometimes called **controls** or **widgets** — short for **window gadgets**.
- ▶ User interacts via the mouse, the keyboard or another form of input, such as voice recognition.
- ▶ IDEs
 - Provide **GUI design tools** to specify a component's exact size and location in a visual manner by using the mouse.
 - **Generates** the GUI code for you.
 - Greatly simplifies creating GUIs, but each IDE has **different capabilities and generates different code**.

14.3 Simple GUI-Based Input/Output with JOptionPane

- ▶ Most applications use windows or **dialog boxes** (also called **dialogs**) to interact with the user.
- ▶ `JOptionPane` (package `javax.swing`) provides **prebuilt dialog boxes** for input and output
 - Displayed via `static JOptionPane` methods.
- ▶ Figure 14.2 uses two **input dialogs** to obtain integers from the user and a **message dialog** to display the sum of the integers the user enters.

```
1 // Fig. 14.2: Addition.java
2 // Addition program that uses JOptionPane for input and output.
3 import javax.swing.JOptionPane; // program uses JOptionPane
4
5 public class Addition
6 {
7     public static void main( String[] args )
8     {
9         // obtain user input from JOptionPane input dialogs
10        String firstNumber =
11            JOptionPane.showInputDialog( "Enter first integer" );
12        String secondNumber =
13            JOptionPane.showInputDialog( "Enter second integer" );
14
15        // convert String inputs to int values for use in a calculation
16        int number1 = Integer.parseInt( firstNumber );
17        int number2 = Integer.parseInt( secondNumber );
18
19        int sum = number1 + number2; // add numbers
20
```

Displays an input dialog and returns a typed by the user

Fig. 14.2 | Addition program that uses JOptionPane for input and output. (Part I of 3.)


```

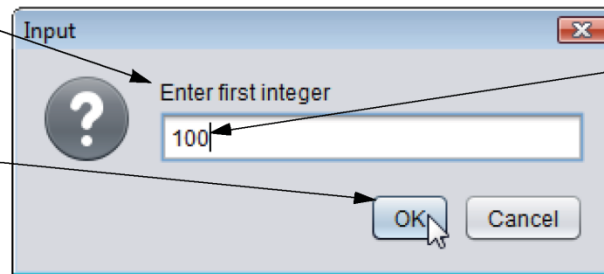
21 // display result in a JOptionPane message dialog
22 JOptionPane.showMessageDialog( null, "The sum is " + sum,
23 "Sum of Two Integers", JOptionPane.PLAIN_MESSAGE );
24 } // end method main
25 } // end class Addition

```

Displays a message dialog centered on the screen with no icon.

(a) Input dialog displayed by lines 10–11

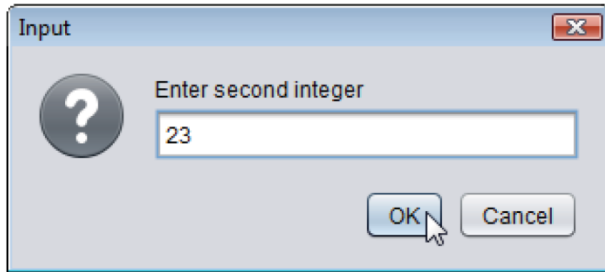
Prompt to the user
 When the user clicks **OK**, `showInputDialog` returns to the program the **100** typed by the user as a **String**. The program must convert the **String** to an **int**



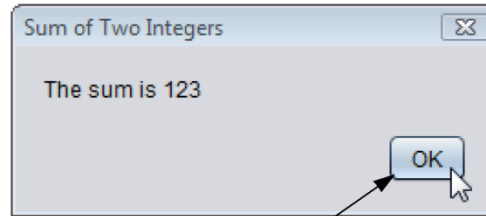
Text field in which the user types a value

Fig. 14.2 | Addition program that uses `JOptionPane` for input and output. (Part 2 of 3.)

(b) Input dialog displayed by lines 12–13



(c) Message dialog displayed by lines 22–23



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

Fig. 14.2 | Addition program that uses `JOptionPane` for input and output. (Part 3 of 3.)





Message dialog type	Icon	Description
ERROR_MESSAGE		Indicates an error to the user.
INFORMATION_MESSAGE		Indicates an informational message to the user.
WARNING_MESSAGE		Warns the user of a potential problem.
QUESTION_MESSAGE		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. 14.3 | JOptionPane static constants for message dialogs.

14.4 Overview of Swing Components

- ▶ Swing GUI components located in package `javax.swing`.
- ▶ Most are pure Java components
 - Written, manipulated and displayed completely in Java.
 - Part of the Java Foundation Classes (JFC) for cross-platform GUI development.
 - JFC and Java desktop technologies

Component	Description
JLabel	Displays uneditable text or icons.
TextField	Enables user to enter data from the keyboard. Can also be used to display editable or uneditable text.
Button	Triggers an event when clicked with the mouse.
CheckBox	Specifies an option that can be selected or not selected.
ComboBox	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.
List	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.
Panel	Provides an area in which components can be placed and organized. Can also be used as a drawing area for graphics.

Fig. 14.4 | Some basic GUI components.

14.4 Overview of Swing Components (cont.)

- ▶ **Abstract Window Toolkit (AWT)** in package `java.awt` is another set of GUI components in Java.
 - When a Java application with an AWT GUI executes on **different Java platforms**, the application's GUI components **display differently** on each platform.
- ▶ Together, the appearance and the way in which the user interacts with the application are known as that application's **look-and-feel**.
- ▶ Swing GUI components allow you to specify a **uniform look-and-feel** for your application **across all platforms** or to use each platform's custom look-and-feel.

14.4 Overview of Swing Components (cont.)

- ▶ Most Swing components **are not tied** to actual GUI components of the underlying platform.
 - Known as **lightweight components**.
- ▶ AWT components are tied to the local platform and are called **heavyweight components**, because they rely on the local platform's **windowing system** to determine their functionality and their look-and-feel.
- ▶ **Several** Swing components are heavyweight components.

14.4 Overview of Swing Components (cont.)

- ▶ 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.
- ▶ Online documentation:
 - <http://docs.oracle.com/javase/8/docs/api/java/awt/Component.html>

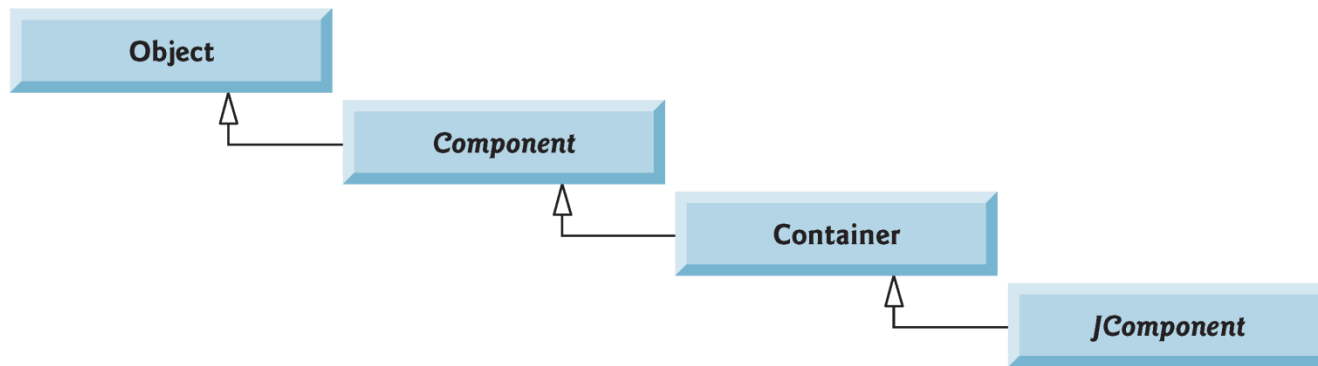


Fig. 14.5 | Common superclasses of many of the Swing components.

14.4 Overview of Swing Components (cont.)

- ▶ Class `Container` (package `java.awt`) is a subclass of `Component`.
- ▶ Components are *attached* to `Containers` so that they can be organized and displayed on the screen.
- ▶ Any object that *is a Container* can be used to *organize* other `Components` in a GUI.
- ▶ Because a `Container` *is a Component*, you can *place Containers in other Containers* to help organize a GUI.
- ▶ Online documentation:
 - <http://docs.oracle.com/javase/8/docs/api/java/awt/Component.html>

14.4 Overview of Swing Components (cont.)

- ▶ Class `JComponent` (package `javax.swing`) is a subclass of `Container`.
- ▶ `JComponent` is the superclass of all lightweight Swing components, all of which are also `Containers`.

14.4 Overview of Swing Components (cont.)

- ▶ Some common lightweight component features supported by `JComponent` include:
 - Pluggable look-and-feel
 - Shortcut keys (called `mnemonics`)
 - Common `event-handling` capabilities for components that initiate the same actions in an application.
 - `tool tips` or `info tip` used in conjunction with a cursor, usually a `pointer`
 - Support for `accessibility`
 - Support for user-interface `localization`
- ▶ Online documentation:
 - <http://docs.oracle.com/javase/8/docs/api/javafx/swing/JComponent.html>

14.5 Displaying Text and Images in a Window (cont.)

- ▶ Most windows that can contain Swing GUI components are **instances** of class `JFrame` or a subclass of `JFrame`.
- ▶ `JFrame` is an indirect subclass of class `java.awt.Window`
- ▶ Provides the basic **attributes and behaviors of a window**
 - a title bar at the top
 - buttons to minimize, maximize and close the window
- ▶ Most of our examples will consist of **two classes**
 - a **subclass of `JFrame`** that demonstrates new GUI concepts
 - an **application class** in which `main` creates and displays the application's primary window.

```
1 // Fig. 14.6: LabelFrame.java
2 // Demonstrating the JLabel class.
3 import java.awt.FlowLayout; // specifies how components are arranged
4 import javax.swing.JFrame; // provides basic window features
5 import javax.swing.JLabel; // displays text and images
6 import javax.swing.SwingConstants; // common constants used with Swing
7 import javax.swing.Icon; // interface used to manipulate images
8 import javax.swing.ImageIcon; // loads images
9
10 public class LabelFrame extends JFrame
11 {
12     private JLabel label1; // JLabel with just text
13     private JLabel label2; // JLabel constructed with text and icon
14     private JLabel label3; // JLabel with added text and icon
15
16     // LabelFrame constructor adds JLabels to JFrame
17     public LabelFrame()
18     {
19         super( "Testing JLabel" );
20         setLayout( new FlowLayout() ); // set frame layout
21     }
22 }
```

Custom GUIs are often built in classes that extend JFrame

Sets the JFrame's title bar text to the specified String

Fig. 14.6 | JLabels with text and icons. (Part I of 2.)

```

22 // JLabel constructor with a string argument
23 label1 = new JLabel( "Label with text" );
24 label1.setToolTipText( "This is label1" );
25 add( label1 ); // add label1 to JFrame
26
27 // JLabel constructor with string, Icon and alignment arguments
28 Icon bug = new ImageIcon( getClass().getResource( "bug1.png" ) );
29 label2 = new JLabel( "Label with text and icon", bug,
30     SwingConstants.LEFT );
31 label2.setToolTipText( "This is label2" );
32 add( label2 ); // add label2 to JFrame
33
34 label3 = new JLabel(); // JLabel constructor no arguments
35 label3.setText( "Label with icon and text at bottom" );
36 label3.setIcon( bug ); // add icon to JLabel
37 label3.setHorizontalTextPosition( SwingConstants.CENTER );
38 label3.setVerticalTextPosition( SwingConstants.BOTTOM );
39 label3.setToolTipText( "This is label3" );
40 add( label3 ); // add label3 to JFrame
41 } // end LabelFrame constructor
42 } // end class LabelFrame

```

← Create a JLabel with the specified text then set its tooltip

← Load in icon from the same location as class LabelFrame, then create a JLabel with text and an icon and set the JLabel's tooltip text.

← Create an empty JLabel then use set methods to change its characteristics.

Fig. 14.6 | JLabels with text and icons. (Part 2 of 2.)

```

1 // Fig. 14.7: LabelTest.java
2 // Testing JLabel.
3 import javax.swing.JFrame;
4
5 public class LabelTest
6 {
7     public static void main( String[] args )
8     {
9         JLabelFrame labelFrame = new JLabelFrame(); // create JLabelFrame
10        labelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        labelFrame.setSize( 260, 180 ); // set frame size
12        labelFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class LabelTest

```

Program should terminate when the user clicks the window's close button.

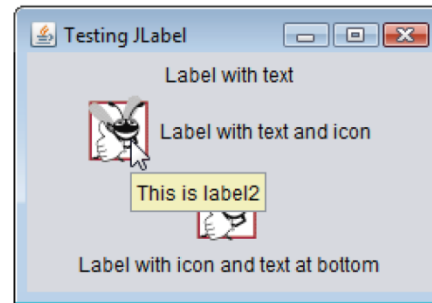
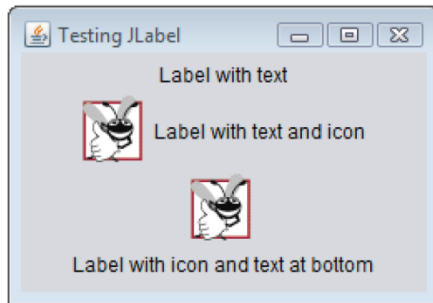


Fig. 14.7 | Test class for JLabelFrame.

14.5 Displaying Text and Images in a Window (cont.)

- ▶ JFrame's constructor uses its `String` argument as the text in the window's title bar.
- ▶ Must **attach** each GUI component **to a container**, such as a JFrame.
- ▶ You typically must decide where to position each GUI component.
 - Known as **specifying the layout** of the GUI components.
 - Java provides several **layout managers** that can help you position components.

14.5 Displaying Text and Images in a Window (cont.)

- ▶ Many IDEs provide GUI design tools in which you can specify the exact size and location of a component
- ▶ IDE generates the GUI code for you
- ▶ Greatly simplifies GUI creation
- ▶ To ensure that the **examples can be used with any IDE**, we will not use an IDE to create the GUI code
 - We will create the code ourselves
 - This way you also learn better each component
- ▶ We will use Java's **layout managers** in our GUI examples

14.5 Displaying Text and Images in a Window (cont.)

- ▶ `FlowLayout`
 - GUI components are placed on a container **from left to right** in the order in which the program attaches them to the container.
 - When there is **no more room to fit** components left to right, components continue to display left to right **on the next line**.
 - If the container is **resized**, a `FlowLayout` **reflows** the components to accommodate the new width of the container, possibly with fewer or more rows of GUI components.
- ▶ Method `setLayout` is inherited from class `Container`.
 - argument must be an object of a class that implements the `LayoutManager` interface (e.g., `FlowLayout`).

14.5 Displaying Text and Images in a Window (cont.)

- ▶ A `JLabel` can display an `Icon`.
- ▶ `JLabel` constructor can receive text and an `Icon`.
 - The last constructor `argument` indicates the `justification` of the label's contents.
 - Interface `SwingConstants` (package `javax.swing`) declares a set of common integer constants (such as `SwingConstants.LEFT`) that are used with many Swing components.
 - By `default`, the text appears to the right of the image when a label contains both text and an image.
 - The horizontal and vertical alignments of a `JLabel` can be set with methods `setHorizontalAlignment` and `setVerticalAlignment`, respectively.

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

Fig. 14.8 | Positioning constants.

14.5 Displaying Text and Images in a Window (cont.)

- ▶ By default, closing a window simply hides the window.
- ▶ Calling method `setDefaultCloseOperation` (inherited from class `JFrame`) with the argument `JFrame.EXIT_ON_CLOSE` indicates that the program should terminate when the window is closed by the user.
- ▶ Method `setSize` specifies the width and height of the window in pixels.
- ▶ Method `setVisible` with the argument `true` displays the window on the screen.

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

- ▶ GUIs are **event driven**.
- ▶ When the user **interacts** with a GUI component, the interaction — known as an **event** — drives the program to perform a task.
- ▶ 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**.

14.6 Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- ▶ `JTextField` and `JPasswordField` (package `javax.swing`).
- ▶ `JTextField` extends class `JTextComponent` (package `javax.swing.text`), which provides many features common to Swing's text-based components.
- ▶ Class `JPasswordField` extends `JTextField` and adds methods that are specific to processing passwords.
- ▶ `JPasswordField` shows that characters are being typed as the user enters them, but hides the actual characters with an echo character.

```
1 // Fig. 14.9: TextFieldFrame.java
2 // Demonstrating the JTextField class.
3 import java.awt.FlowLayout;
4 import java.awt.event.ActionListener;
5 import java.awt.event.ActionEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JTextField;
8 import javax.swing.JPasswordField;
9 import javax.swing.JOptionPane;
10
11 public class TextFieldFrame extends JFrame
12 {
13     private JTextField textField1; // text field with set size
14     private JTextField textField2; // text field constructed with text
15     private JTextField textField3; // text field with text and size
16     private JPasswordField passwordField; // password field with text
17
18     // TextFieldFrame constructor adds JTextFields to JFrame
19     public TextFieldFrame()
20     {
21         super( "Testing JTextField and JPasswordField" );
22         setLayout( new FlowLayout() ); // set frame layout
23
```

Fig. 14.9 | JTextFields and JPasswordField. (Part I of 4.)

```
24 // construct textfield with 10 columns
25 textField1 = new JTextField( 10 );
26 add( textField1 ); // add textField1 to JFrame
27
28 // construct textfield with default text
29 textField2 = new JTextField( "Enter text here" );
30 add( textField2 ); // add textField2 to JFrame
31
32 // construct textfield with default text and 21 columns
33 textField3 = new JTextField( "Uneditable text field", 21 );
34 textField3.setEditable( false ); // disable editing
35 add( textField3 ); // add textField3 to JFrame
36
37 // construct passwordfield with default text
38 passwordField = new JPasswordField( "Hidden text" );
39 add( passwordField ); // add passwordField to JFrame
40
41 // register event handlers
42 TextFieldHandler handler = new TextFieldHandler();
43 textField1.addActionListener( handler );
44 textField2.addActionListener( handler );
45 textField3.addActionListener( handler );
46 passwordField.addActionListener( handler );
47 } // end TextFieldFrame constructor
```

Width of the JTextField is based on the component's current font unless a layout manager overrides that size.

Width of the JTextField is based on the default text unless a layout manager overrides that size.

Width based on second argument unless a layout manager overrides that size.

Text in this component will be hidden by asterisks (*) by default.

TextFieldHandler inner class implements ActionListener interface, so it can respond to JTextField events. Lines 43–46 register the object handler to respond to each component's events.

Fig. 14.9 | JTextField and JPasswordField. (Part 2 of 4.)

```
48
49 // private inner class for event handling
50 private class TextFieldHandler implements ActionListener
51 {
52     // process text field events
53     public void actionPerformed((ActionEvent event) )
54     {
55         String string = ""; // declare string to display
56
57         // user pressed Enter in JTextField textField1
58         if ( event.getSource() == textField1 )
59             string = String.format( "textField1: %s",
60                                     event.getActionCommand() );
61
62         // user pressed Enter in JTextField textField2
63         else if ( event.getSource() == textField2 )
64             string = String.format( "textField2: %s",
65                                     event.getActionCommand() );
66
67         // user pressed Enter in JTextField textField3
68         else if ( event.getSource() == textField3 )
69             string = String.format( "textField3: %s",
70                                     event.getActionCommand() );
71
```

A TextFieldHandler is an ActionListener.

Called when the user presses *Enter* in a JTextField or JPasswordField.

getSource specifies which component the user interacted with

Obtains the text the user typed in the textfield.

Fig. 14.9 | JTextFields and JPasswordField. (Part 3 of 4.)

```
72         // user pressed Enter in JTextField passwordField
73         else if ( event.getSource() == passwordField )
74             string = String.format( "passwordField: %s",
75                                     event.getActionCommand() );
76
77         // display JTextField content
78         JOptionPane.showMessageDialog( null, string );
79     } // end method actionPerformed
80 } // end private inner class TextFieldHandler
81 } // end class TextFieldFrame
```

Fig. 14.9 | JTextFields and JPasswordField. (Part 4 of 4.)

```
1 // Fig. 14.10: TextFieldTest.java
2 // Testing TextFieldFrame.
3 import javax.swing.JFrame;
4
5 public class TextFieldTest
6 {
7     public static void main( String[] args )
8     {
9         TextFieldFrame textFieldFrame = new TextFieldFrame();
10        textFieldFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        textFieldFrame.setSize( 350, 100 ); // set frame size
12        textFieldFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class TextFieldTest
```

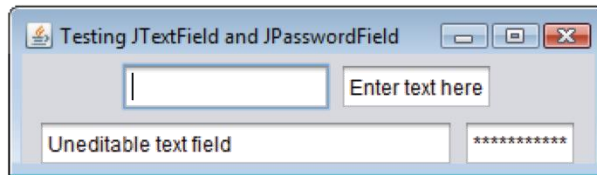


Fig. 14.10 | Test class for TextFieldFrame. (Part 1 of 3.)

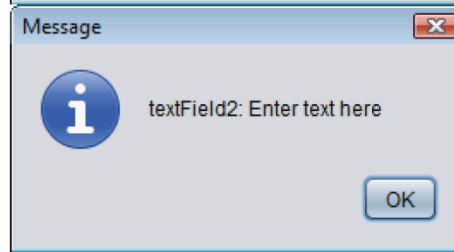
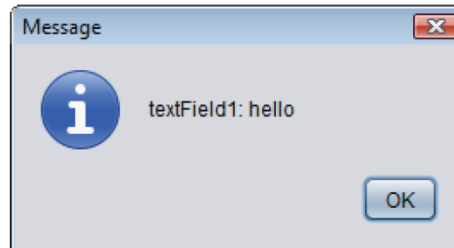
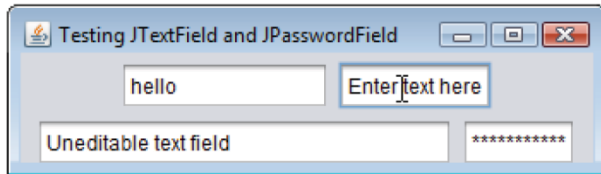
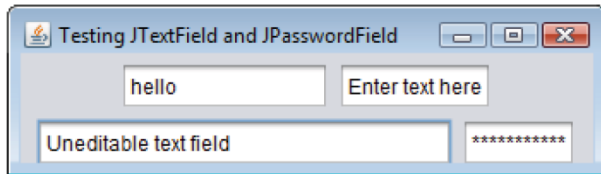


Fig. 14.10 | Test class for TextFieldFrame. (Part 2 of 3.)

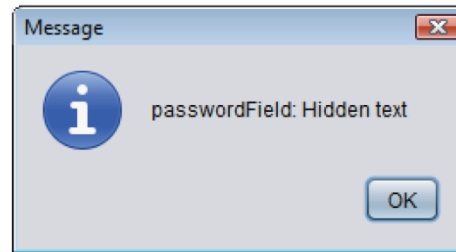
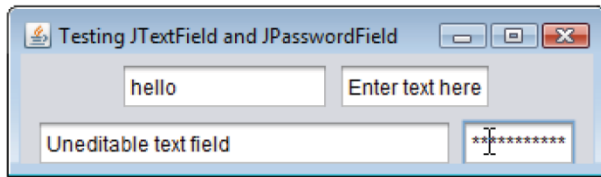
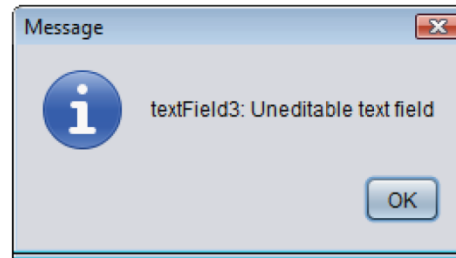
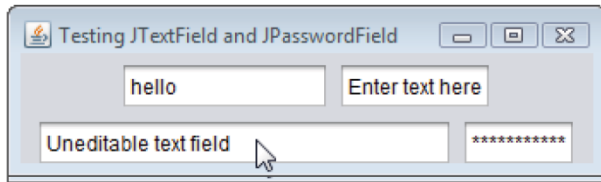


Fig. 14.10 | Test class for TextFieldFrame. (Part 3 of 3.)

14.6 Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- ▶ When the user **types data** into a `JTextField` or a `JPasswordField`, then **presses *Enter***, an event occurs.
- ▶ You can type only in the text field that is “**in focus.**”
- ▶ A component **receives the focus** when the user clicks the component.

14.6 Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- ▶ Before an application can respond to an event for a particular GUI component, you must perform several coding steps:
 - 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**.

14.6 Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- ▶ All the classes discussed so far were so-called **top-level classes** — that is, they were not declared inside another class.
- ▶ Java allows you to declare **classes inside other classes** — these are called **nested classes**.
 - Can be **static** or **non-static**.
 - **Non-static** nested classes are called **inner classes** and are frequently used to implement event handlers.

14.6 Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- ▶ **Before** an object of an inner class can be created, there must **first** be an object of the top-level class that **contains** the inner class.
- ▶ This is required because an inner-class object **implicitly has a reference** to an object of its top-level class.
- ▶ There is also a **special relationship** between these objects — the inner-class object is allowed to **directly access all the variables and methods of the outer class**.
- ▶ A nested class that is **static does not** require an object of its top-level class and **does not** implicitly have a reference to an object of the top-level class.

14.6 Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- ▶ Inner classes can be declared `public`, `protected` or `private`.
- ▶ Since event handlers tend to be `specific to the application` in which they are defined, they are often implemented as `private inner classes`.

14.6 Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- ▶ GUI components can **generate many 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.
- ▶ Normally, a component's **supported events** are **described** in the **Java API** documentation for that component's class and its superclasses.

14.6 Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- ▶ When the user presses *Enter* in a `JTextField` or `JPasswordField`, an `ActionEvent` (package `java.awt.event`) occurs.
- ▶ Processed by an object that implements the interface `ActionListener` (package `java.awt.event`).
- ▶ To handle `ActionEvents`, a class must implement interface `ActionListener` and declare method `actionPerformed`.
 - This method specifies the tasks to perform when an `ActionEvent` occurs.



Software Engineering Observation 14.3

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



Common Programming Error 14.2

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

14.6 Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- ▶ Must register an object as the event handler for each text field.
- ▶ `addActionListener` registers an `ActionListener` object to handle `ActionEvents`.
- ▶ After an event handler is registered the object listens for events.

14.6 Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

- ▶ The GUI component with which the user interacts is the **event source**.
- ▶ `ActionEvent` method `getSource` (inherited from class `EventObject`) **returns a reference to the event source**.
- ▶ `ActionEvent` method `getActionCommand` obtains the text the user typed in the text field that generated the event.
- ▶ `JPasswordField` method `getPassword` returns the password's characters as an array of type `char`.

14.7 Common GUI Event Types and Listener Interfaces

- ▶ Figure 14.11 illustrates a hierarchy containing many event classes from the package `java.awt.event`.
- ▶ Used with both AWT and Swing components.
- ▶ Additional event types that are specific to Swing GUI components are declared in package `javax.swing.event`.

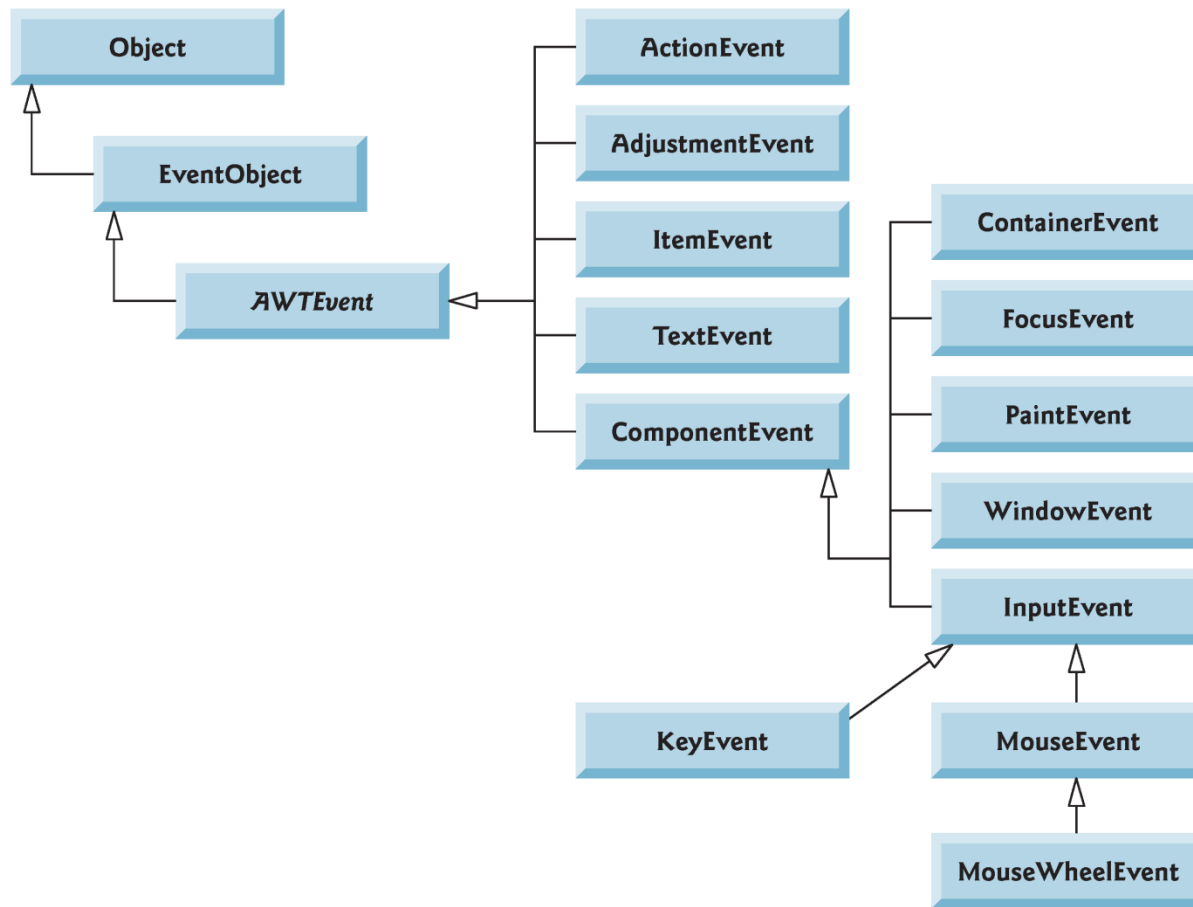


Fig. 14.11 | Some event classes of package `java.awt.event`.

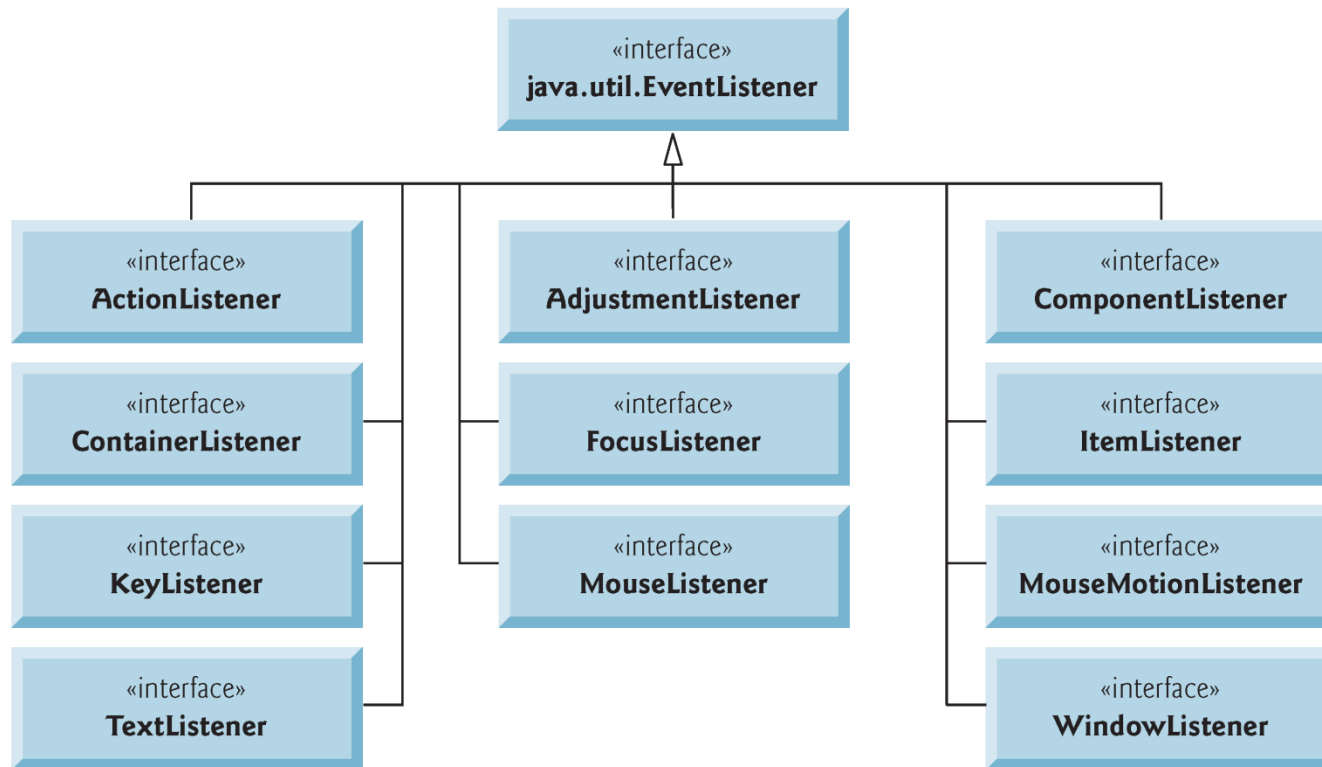


Fig. 14.12 | Some common event-listener interfaces of package `java.awt.event`.

14.7 Common GUI Event Types and Listener Interfaces (cont.)

- ▶ **Delegation event model** — an event's processing is delegated to an object (the **event listener**) in the application.
- ▶ For each event-object type, there is typically a **corresponding event-listener interface**.
- ▶ Many event-listener types are **common** to both Swing and AWT components.
 - Such types are declared in package `java.awt.event`, and some of them are shown in Fig. 14.12.
- ▶ Additional event-listener types that are **specific to Swing** components are declared in package `javax.swing.event`.

14.7 Common GUI Event Types and Listener Interfaces (cont.)

- ▶ Each event-listener interface specifies one or more **event-handling methods** that **must be** declared in the class that implements the interface.
- ▶ When an event occurs, the GUI component with which the user interacted **notifies** its registered listeners by calling each listener's appropriate event-handling method.

14.8 How Event Handling Works

- ▶ How the event-handling mechanism works:
- ▶ Every `JComponent` has a variable `listenerList` that refers to an `EventListenerList` (package `javax.swing.event`).
- ▶ Maintains `references to registered listeners` in the `listenerList`.
- ▶ When a listener is registered, a new entry is placed in the component's `listenerList`.
- ▶ Every entry also includes the listener's type.

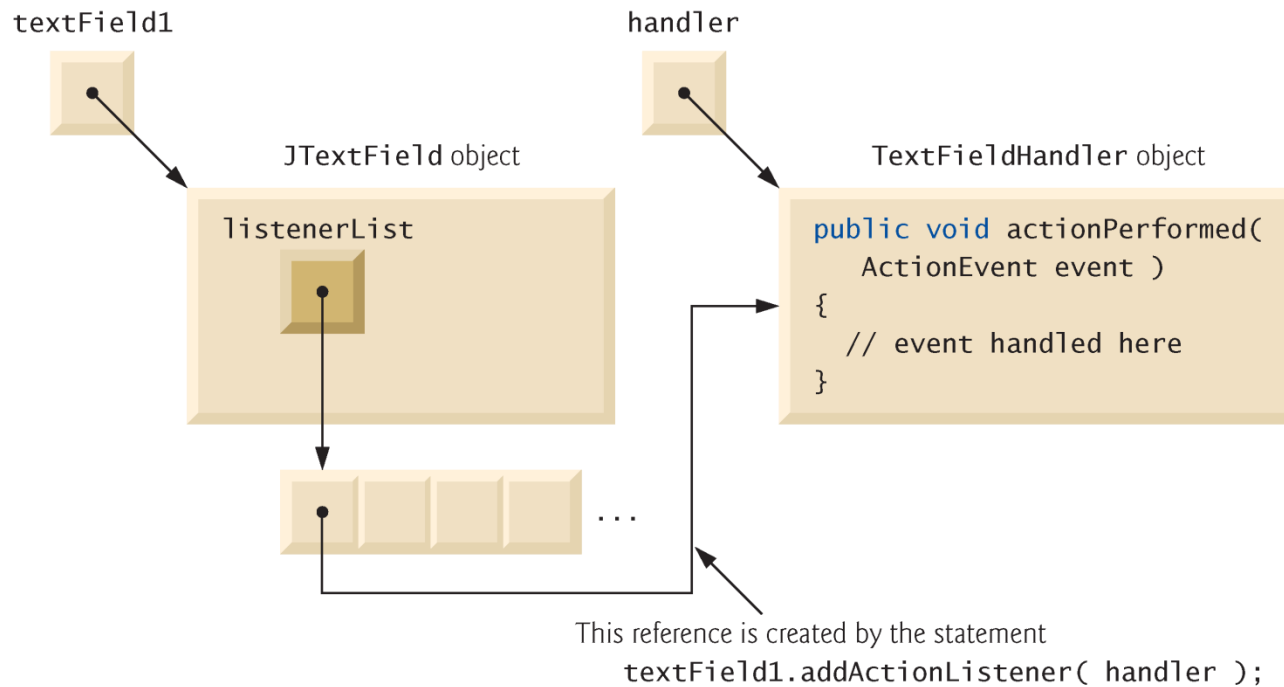


Fig. 14.13 | Event registration for `JTextField` `textField1`.

14.8 How Event Handling Works (cont.)

- ▶ How does the GUI component know to call `actionPerformed` rather than another method?
 - Every GUI component supports several event types, including `mouse events`, `key events` and others.
 - When an event occurs, the event is `dispatched` only to the event listeners of the `appropriate type`.
 - `Dispatching` is simply the process by which the GUI component calls an event-handling method on each of its listeners that are registered for the event type that occurred.

14.8 How Event Handling Works (cont.)

- ▶ Each event type has **one or more** corresponding event-listener interfaces.
 - `ActionEvents` are handled by `ActionListeners`
 - `MouseEvent`s are handled by `MouseListeners` and `MouseMotionListeners`
 - `KeyEvent`s are handled by `KeyListener`s
- ▶ When an event occurs, the GUI component receives (from the JVM) a unique **event ID** specifying the event type.
 - The component uses the **event ID** to decide the **listener type** to which the event should be dispatched and to decide **which method** to call on each listener object.

14.8 How Event Handling Works (cont.)

- ▶ For an `ActionEvent`, the event is dispatched to every registered `ActionListener`'s `actionPerformed` method.
- ▶ For a `MouseEvent`, the event is dispatched to every registered `MouseListener` or `MouseMotionListener`, depending on the mouse event that occurs.
 - The `MouseEvent`'s `event ID` determines which of the several mouse event-handling methods are called.

14.9 JButton

- ▶ A **button** is a component the user clicks to trigger a specific action.
- ▶ Several types of buttons
 - **command buttons**
 - **checkboxes**
 - **toggle buttons**
 - **radio buttons**
- ▶ Button types are subclasses of **AbstractButton** (package `javax.swing`), which declares the common features of Swing buttons.

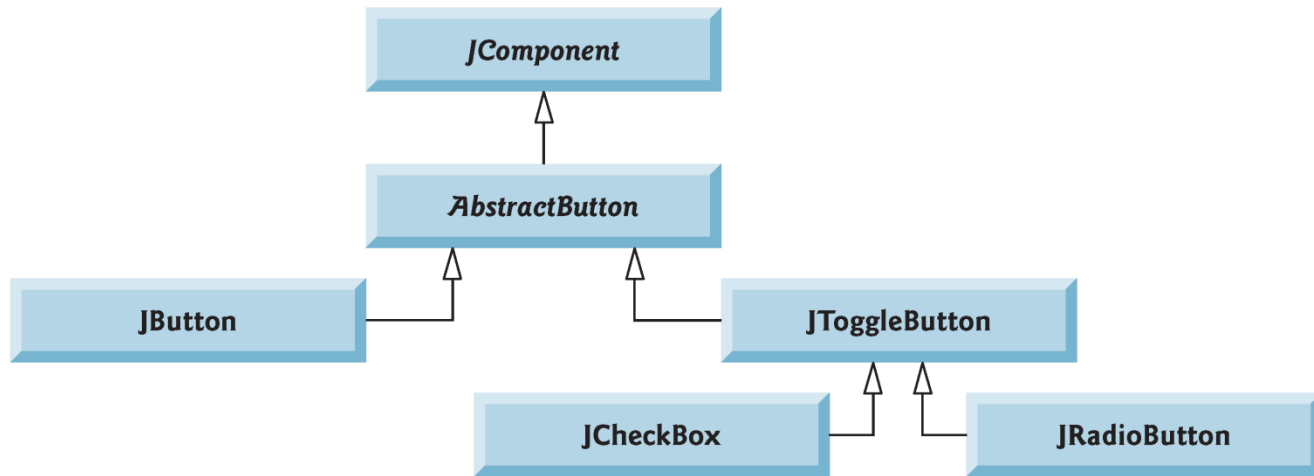


Fig. 14.14 | Swing button hierarchy.

14.9 JButton (cont.)

- ▶ A command button generates an `ActionEvent` when the user clicks it.
- ▶ Command buttons are created with class `JButton`.
- ▶ The text on the face of a `JButton` is called a `button label`.

```
1 // Fig. 14.15: ButtonFrame.java
2 // Creating JButtons.
3 import java.awt.FlowLayout;
4 import java.awt.event.ActionListener;
5 import java.awt.event.ActionEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JButton;
8 import javax.swing.Icon;
9 import javax.swing.ImageIcon;
10 import javax.swing.JOptionPane;
11
12 public class ButtonFrame extends JFrame
13 {
14     private JButton plainJButton; // button with just text
15     private JButton fancyJButton; // button with icons
16
17     // ButtonFrame adds JButtons to JFrame
18     public ButtonFrame()
19     {
20         super( "Testing Buttons" );
21         setLayout( new FlowLayout() ); // set frame layout
22
23         plainJButton = new JButton( "Plain Button" ); // button with text
24         add( plainJButton ); // add plainJButton to JFrame
```

Creates a JButton
with the specified text
as its label.

Fig. 14.15 | Command buttons and action events. (Part 1 of 2.)


```

25
26 Icon bug1 = new ImageIcon( getClass().getResource( "bug1.gif" ) );
27 Icon bug2 = new ImageIcon( getClass().getResource( "bug2.gif" ) );
28 fancyJButton = new JButton( "Fancy Button", bug1 ); // set image
29 fancyJButton.setRolloverIcon( bug2 ); // set rollover image
30 add( fancyJButton ); // add fancyJButton to JFrame
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

```

Load two images from the same location as class `ButtonFrame`, then use the first as the default icon on the `JButton` and the second as the rollover icon.

Create object of inner class `ButtonHandler` and register it to handle the `ActionEvents` for both `JButtons`.

Objects of this class can respond to `ActionEvents`.

`ButtonFrame.this` is special notation that enables the inner class to access the `this` reference from the top-level class `ButtonFrame`.

Fig. 14.15 | Command buttons and action events. (Part 2 of 2.)

```
1 // Fig. 14.16: ButtonTest.java
2 // Testing ButtonFrame.
3 import javax.swing.JFrame;
4
5 public class ButtonTest
6 {
7     public static void main( String[] args )
8     {
9         ButtonFrame buttonFrame = new ButtonFrame(); // create ButtonFrame
10        buttonFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        buttonFrame.setSize( 275, 110 ); // set frame size
12        buttonFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class ButtonTest
```

Fig. 14.16 | Test class for ButtonFrame. (Part 1 of 2.)

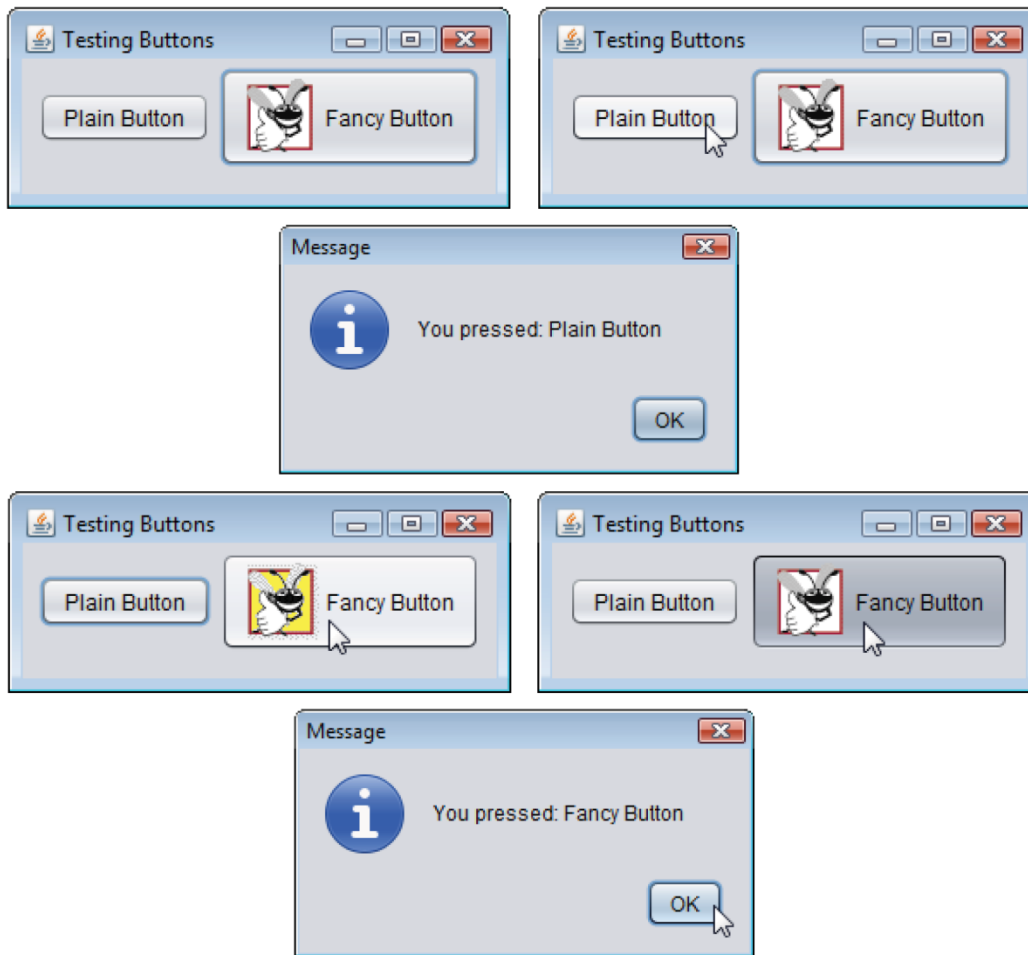


Fig. 14.16 | Test class for ButtonFrame. (Part 2 of 2.)

14.10 Buttons That Maintain State

- ▶ Three types of **state buttons** — `JToggleButton`, `JCheckBox` and `JRadioButton` — that have on/off or true/false values.
- ▶ Classes `JCheckBox` and `JRadioButton` are subclasses of `JToggleButton`.
- ▶ `JRadioButtons` are grouped together and are **mutually exclusive** — **only one** in the group can be selected at any time

14.10.1 JCheckBox

- ▶ `JTextField` method `setFont` (inherited by `JTextField` indirectly from class `Component`) sets the font of the `JTextField` to a new `Font` (package `java.awt`).
- ▶ `String` passed to the `JCheckBox` constructor is the `checkbox label` that appears to the right of the `JCheckBox` by default.
- ▶ When the user clicks a `JCheckBox`, an `ItemEvent` occurs.
 - Handled by an `ItemListener` object, which must implement method `itemStateChanged`.
- ▶ An `ItemListener` is registered with method `addItemListener`.
- ▶ `JCheckBox` method `isSelected` returns `true` if a `JCheckBox` is selected.

```
1 // Fig. 14.17: CheckBoxFrame.java
2 // Creating JCheckBox buttons.
3 import java.awt.FlowLayout;
4 import java.awt.Font;
5 import java.awt.event.ItemListener;
6 import java.awt.event.ItemEvent;
7 import javax.swing.JFrame;
8 import javax.swing.JTextField;
9 import javax.swing.JCheckBox;
10
11 public class CheckBoxFrame extends JFrame
12 {
13     private JTextField textField; // displays text in changing fonts
14     private JCheckBox boldJCheckBox; // to select/deselect bold
15     private JCheckBox italicJCheckBox; // to select/deselect italic
16
17     // CheckBoxFrame constructor adds JCheckBoxes to JFrame
18     public CheckBoxFrame()
19     {
20         super( "JCheckBox Test" );
21         setLayout( new FlowLayout() ); // set frame layout
22
23         // set up JTextField and set its font
24         textField = new JTextField( "Watch the font style change", 20 );
```

Fig. 14.17 | JCheckBox buttons and item events. (Part I of 3.)

```
25  textField.setFont( new Font( "Serif", Font.PLAIN, 14 ) );
26  add( textField ); // add textField to JFrame
27
28  boldJCheckBox = new JCheckBox( "Bold" ); // create bold checkbox
29  italicJCheckBox = new JCheckBox( "Italic" ); // create italic
30  add( boldJCheckBox ); // add bold checkbox to JFrame
31  add( italicJCheckBox ); // add italic checkbox to JFrame
32
33  // register listeners for JCheckBoxes
34  CheckBoxHandler handler = new CheckBoxHandler();
35  boldJCheckBox.addItemListener( handler );
36  italicJCheckBox.addItemListener( handler );
37 } // end CheckBoxFrame constructor
38
39 // private inner class for ItemListener event handling
40 private class CheckBoxHandler implements ItemListener
41 {
42     // respond to checkbox events
43     public void itemStateChanged( ItemEvent event )
44     {
45         Font font = null; // stores the new Font
46
```

setFont can be used to change the font for any component.

Create and register the event handler for both JCheckBoxes.

An object of this class can respond to ItemEvents.

Fig. 14.17 | JCheckBox buttons and item events. (Part 2 of 3.)

```
47 // determine which CheckBoxes are checked and create Font
48 if ( boldJCheckBox.isSelected() && italicJCheckBox.isSelected() )
49     font = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
50 else if ( boldJCheckBox.isSelected() )
51     font = new Font( "Serif", Font.BOLD, 14 );
52 else if ( italicJCheckBox.isSelected() )
53     font = new Font( "Serif", Font.ITALIC, 14 );
54 else
55     font = new Font( "Serif", Font.PLAIN, 14 );
56
57     textField.setFont( font ); // set textField's font
58 } // end method itemStateChanged
59 } // end private inner class CheckBoxHandler
60 } // end class CheckBoxFrame
```

JCheckBox method
isSelected returns
true if the JCheckBox
on which it's called is
checked.

Fig. 14.17 | JCheckBox buttons and item events. (Part 3 of 3.)

```
1 // Fig. 14.18: CheckBoxTest.java
2 // Testing CheckBoxFrame.
3 import javax.swing.JFrame;
4
5 public class CheckBoxTest
6 {
7     public static void main( String[] args )
8     {
9         CheckBoxFrame checkBoxFrame = new CheckBoxFrame();
10        checkBoxFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        checkBoxFrame.setSize( 275, 100 ); // set frame size
12        checkBoxFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class CheckBoxTest
```

Fig. 14.18 | Test class for CheckBoxFrame. (Part 1 of 2.)

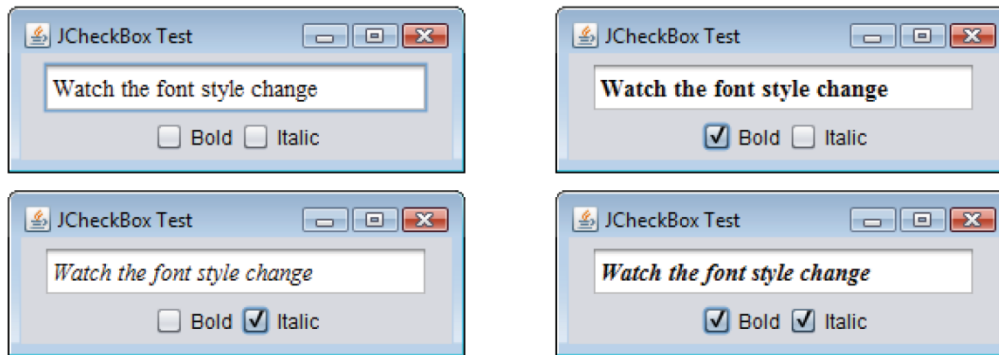


Fig. 14.18 | Test class for CheckBoxFrame. (Part 2 of 2.)

14.10.2 JRadioButton

- ▶ **Radio buttons** (declared with class `JRadioButton`) are similar to checkboxes in that they have **two states** — selected and not selected (also called deselected).
- ▶ 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.
- ▶ Used to represent **mutually exclusive options**.
- ▶ The logical relationship between radio buttons is maintained by a `ButtonGroup` object (package `javax.swing`), which organizes a group of buttons and is not itself displayed in a user interface.

```
1 // Fig. 14.19: RadioButtonFrame.java
2 // Creating radio buttons using ButtonGroup and JRadioButton.
3 import java.awt.FlowLayout;
4 import java.awt.Font;
5 import java.awt.event.ItemListener;
6 import java.awt.event.ItemEvent;
7 import javax.swing.JFrame;
8 import javax.swing.JTextField;
9 import javax.swing.JRadioButton;
10 import javax.swing.ButtonGroup;
11
12 public class RadioButtonFrame extends JFrame
13 {
14     private JTextField textField; // used to display font changes
15     private Font plainFont; // font for plain text
16     private Font boldFont; // font for bold text
17     private Font italicFont; // font for italic text
18     private Font boldItalicFont; // font for bold and italic text
19     private JRadioButton plainJRadioButton; // selects plain text
20     private JRadioButton boldJRadioButton; // selects bold text
21     private JRadioButton italicJRadioButton; // selects italic text
22     private JRadioButton boldItalicJRadioButton; // bold and italic
23     private ButtonGroup radioGroup; // buttongroup to hold radio buttons
24
```

Manages the relationship between radio buttons.

Fig. 14.19 | JRadioButtons and ButtonGroups. (Part I of 4.)

```

25 // RadioButtonFrame constructor adds JRadioButtons to JFrame
26 public RadioButtonFrame()
27 {
28     super( "RadioButton Test" );
29     setLayout( new FlowLayout() ); // set frame layout
30
31     textField = new JTextField( "Watch the font style change", 25 );
32     add( textField ); // add textField to JFrame
33
34     // create radio buttons
35     plainJRadioButton = new JRadioButton( "Plain", true );
36     boldJRadioButton = new JRadioButton( "Bold", false );
37     italicJRadioButton = new JRadioButton( "Italic", false );
38     boldItalicJRadioButton = new JRadioButton( "Bold/Italic", false );
39     add( plainJRadioButton ); // add plain button to JFrame
40     add( boldJRadioButton ); // add bold button to JFrame
41     add( italicJRadioButton ); // add italic button to JFrame
42     add( boldItalicJRadioButton ); // add bold and italic button
43
44     // create logical relationship between JRadioButtons
45     radioButtonGroup = new ButtonGroup(); // create ButtonGroup
46     radioButtonGroup.add( plainJRadioButton ); // add plain to group
47     radioButtonGroup.add( boldJRadioButton ); // add bold to group

```

← This one will be selected initially.

← Manages the relationship between all radio buttons that are added to the group.

Fig. 14.19 | JRadioButtons and ButtonGroups. (Part 2 of 4.)

```
48     radioGroup.add( italicJRadioButton ); // add italic to group
49     radioGroup.add( boldItalicJRadioButton ); // add bold and italic
50
51     // create font objects
52     plainFont = new Font( "Serif", Font.PLAIN, 14 );
53     boldFont = new Font( "Serif", Font.BOLD, 14 );
54     italicFont = new Font( "Serif", Font.ITALIC, 14 );
55     boldItalicFont = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
56     textField.setFont( plainFont ); // set initial font to plain
57
58     // register events for JRadioButtons
59     plainJRadioButton.addItemListener(
60         new RadioButtonHandler( plainFont ) );
61     boldJRadioButton.addItemListener(
62         new RadioButtonHandler( boldFont ) );
63     italicJRadioButton.addItemListener(
64         new RadioButtonHandler( italicFont ) );
65     boldItalicJRadioButton.addItemListener(
66         new RadioButtonHandler( boldItalicFont ) );
67 } // end RadioButtonFrame constructor
68
```

← Notice that we are creating a separate event-handling object for each `JRadioButton`. This enables us to specify the exact `Font` will be used when a particular one is selected.

Fig. 14.19 | `JRadioButtons` and `ButtonGroups`. (Part 3 of 4.)

```
69 // private inner class to handle radio button events
70 private class RadioButtonHandler implements ItemListener
71 {
72     private Font font; // font associated with this listener
73
74     public RadioButtonHandler( Font f )
75     {
76         font = f; // set the font of this listener
77     } // end constructor RadioButtonHandler
78
79     // handle radio button events
80     public void itemStateChanged( ItemEvent event )
81     {
82         textField.setFont( font ); // set font of textField
83     } // end method itemStateChanged
84 } // end private inner class RadioButtonHandler
85 } // end class RadioButtonFrame
```

Objects of this class can respond to ItemEvents,

Stores the Font that is specific to a particular radio button.

Fig. 14.19 | JRadioButtons and ButtonGroups. (Part 4 of 4.)

```
1 // Fig. 14.20: RadioButtonTest.java
2 // Testing RadioButtonFrame.
3 import javax.swing.JFrame;
4
5 public class RadioButtonTest
6 {
7     public static void main( String[] args )
8     {
9         RadioButtonFrame radioButtonFrame = new RadioButtonFrame();
10        radioButtonFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        radioButtonFrame.setSize( 300, 100 ); // set frame size
12        radioButtonFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class RadioButtonTest
```

Fig. 14.20 | Test class for RadioButtonFrame. (Part 1 of 2.)

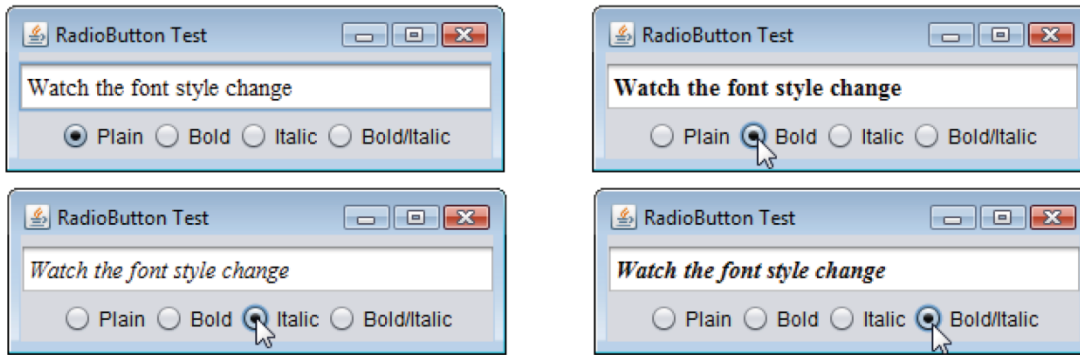


Fig. 14.20 | Test class for RadioButtonFrame. (Part 2 of 2.)

14.11 JComboBox and Using an Anonymous Inner Class for Event Handling

- ▶ A combo box (or **drop-down list**) enables the user to select one item from a list.
- ▶ Combo boxes are implemented with class `JComboBox`, which extends class `JComponent`.
- ▶ `JComboBoxes` generate `ItemEvents`.

```
1 // Fig. 14.21: ComboBoxFrame.java
2 // JComboBox that displays a list of image names.
3 import java.awt.FlowLayout;
4 import java.awt.event.ItemListener;
5 import java.awt.event.ItemEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JLabel;
8 import javax.swing.JComboBox;
9 import javax.swing.Icon;
10 import javax.swing.ImageIcon;
11
12 public class ComboBoxFrame extends JFrame
13 {
14     private JComboBox imagesJComboBox; // combobox to hold names of icons
15     private JLabel label; // label to display selected icon
16
17     private static final String[] names =
18         { "bug1.gif", "bug2.gif", "travelbug.gif", "buganim.gif" };
19     private Icon[] icons = {
20         new ImageIcon( getClass().getResource( names[ 0 ] ) ),
21         new ImageIcon( getClass().getResource( names[ 1 ] ) ),
22         new ImageIcon( getClass().getResource( names[ 2 ] ) ),
23         new ImageIcon( getClass().getResource( names[ 3 ] ) ) };
24
```

Fig. 14.21 | JComboBox that displays a list of image names. (Part I of 3.)

```

25 // JComboBoxFrame constructor adds JComboBox to JFrame
26 public JComboBoxFrame()
27 {
28     super( "Testing JComboBox" );
29     setLayout( new FlowLayout() ); // set frame layout
30
31     imagesJComboBox = new JComboBox( names ); // set up JComboBox
32     imagesJComboBox.setMaximumRowCount( 3 ); // display three rows
33
34     imagesJComboBox.addItemListener(
35         new ItemListener() // anonymous inner class
36         {
37             // handle JComboBox event
38             public void itemStateChanged( ItemEvent event )
39             {
40                 // determine whether item selected
41                 if ( event.getStateChange() == ItemEvent.SELECTED )
42                     label.setIcon( icons[
43                         imagesJComboBox.getSelectedIndex() ] );
44             } // end method itemStateChanged
45         } // end anonymous inner class
46     ); // end call to addItemListener
47

```

← Uses the Strings in array names as the options in the JComboBox.

← Lines 34–46 create an object of an anonymous inner class that implements interface ItemListener and register that object to handle the JComboBox's ItemEvents.

Fig. 14.21 | JComboBox that displays a list of image names. (Part 2 of 3.)

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

Fig. 14.21 | JComboBox that displays a list of image names. (Part 3 of 3.)

```
1 // Fig. 14.22: ComboBoxTest.java
2 // Testing ComboBoxFrame.
3 import javax.swing.JFrame;
4
5 public class ComboBoxTest
6 {
7     public static void main( String[] args )
8     {
9         ComboBoxFrame comboBoxFrame = new ComboBoxFrame();
10        comboBoxFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        comboBoxFrame.setSize( 350, 150 ); // set frame size
12        comboBoxFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class ComboBoxTest
```

Fig. 14.22 | Testing ComboBoxFrame. (Part I of 2.)

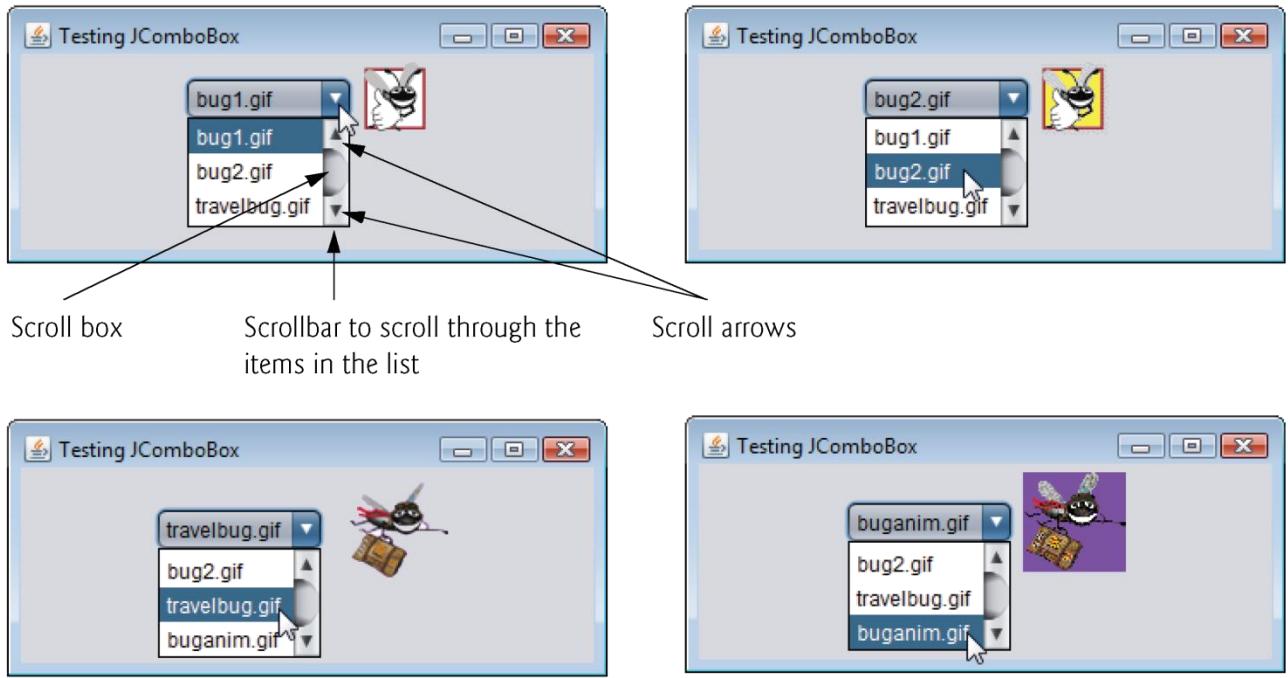


Fig. 14.22 | Testing JComboBoxFrame. (Part 2 of 2.)

14.11 JComboBox and Using an Anonymous Inner Class for Event Handling (cont.)

- ▶ An **anonymous inner class** is an inner class that is declared without a name and typically appears inside a method declaration.
- ▶ As with other inner classes, an anonymous inner class can **access its top-level class's members**.
- ▶ Since an anonymous inner class has no name, one object of the anonymous inner class **must be created** at the point where the class is declared.

14.12 JList

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

```
1 // Fig. 14.23: ListFrame.java
2 // JList that displays a list of colors.
3 import java.awt.FlowLayout;
4 import java.awt.Color;
5 import javax.swing.JFrame;
6 import javax.swing.JList;
7 import javax.swing.JScrollPane;
8 import javax.swing.event.ListSelectionListener;
9 import javax.swing.event.ListSelectionEvent;
10 import javax.swing.ListSelectionModel;
11
12 public class ListFrame extends JFrame
13 {
14     private JList colorJList; // list to display colors
15     private static final String[] colorNames = { "Black", "Blue", "Cyan",
16         "Dark Gray", "Gray", "Green", "Light Gray", "Magenta",
17         "Orange", "Pink", "Red", "White", "Yellow" };
18     private static final Color[] colors = { Color.BLACK, Color.BLUE,
19         Color.CYAN, Color.DARK_GRAY, Color.GRAY, Color.GREEN,
20         Color.LIGHT_GRAY, Color.MAGENTA, Color.ORANGE, Color.PINK,
21         Color.RED, Color.WHITE, Color.YELLOW };
22
```

Fig. 14.23 | JList that displays a list of colors. (Part 1 of 3.)

```
23 // ListFrame constructor add JScrollPane containing JList to JFrame
24 public ListFrame()
25 {
26     super( "List Test" );
27     setLayout( new FlowLayout() ); // set frame layout
28
29     colorJList = new JList( colorNames ); // create with colorNames
30     colorJList.setVisibleRowCount( 5 ); // display five rows at once
31
32     // do not allow multiple selections
33     colorJList.setSelectionMode( ListSelectionMode.SINGLE_SELECTION );
34
35     // add a JScrollPane containing JList to frame
36     add( new JScrollPane( colorJList ) );
37
```

Populate the JList with the Strings in array colorNames.

Allow only single selections.

Provide scrollbars for the JList if necessary.

Fig. 14.23 | JList that displays a list of colors. (Part 2 of 3.)

```
38     colorJList.addListSelectionListener(  
39         new ListSelectionListener() // anonymous inner class  
40     {  
41         // handle list selection events  
42         public void valueChanged( ListSelectionEvent event )  
43     {  
44         getContentPane().setBackground(  
45             colors[ colorJList.getSelectedIndex() ] );  
46         } // end method valueChanged  
47     } // end anonymous inner class  
48 ); // end call to addListSelectionListener  
49 } // end ListFrame constructor  
50 } // end class ListFrame
```

Choose the appropriate Color to change the window's background color.

Fig. 14.23 | JList that displays a list of colors. (Part 3 of 3.)

```
1 // Fig. 14.24: ListTest.java
2 // Selecting colors from a JList.
3 import javax.swing.JFrame;
4
5 public class ListTest
6 {
7     public static void main( String[] args )
8     {
9         ListFrame listFrame = new ListFrame(); // create ListFrame
10        listFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        listFrame.setSize( 350, 150 ); // set frame size
12        listFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class ListTest
```

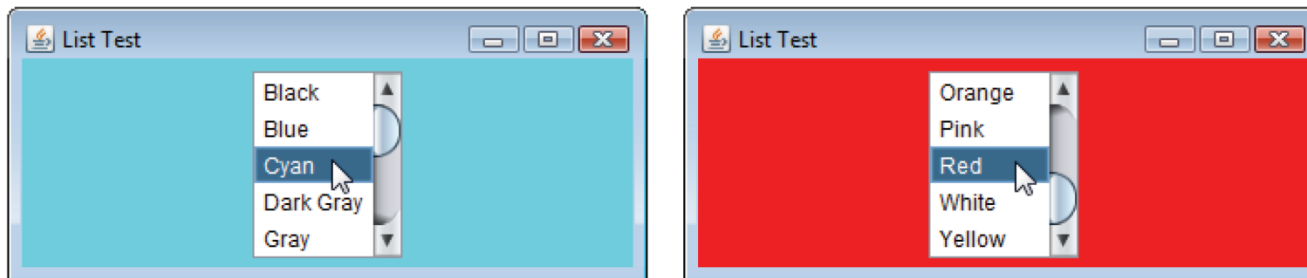


Fig. 14.24 | Test class for ListFrame.

14.12 JList (cont.)

- ▶ `setVisibleRowCount` specifies the number of items visible in the list.
- ▶ `setSelectionMode` specifies the list's **selection mode**.
- ▶ Class `ListSelectionModel` (of package `javax.swing`) declares selection-mode constants
 - `SINGLE_SELECTION` (only one item to be selected at a time)
 - `SINGLE_INTERVAL_SELECTION` (allows selection of several contiguous items)
 - `MULTIPLE_INTERVAL_SELECTION` (does not restrict the items that can be selected).

14.12 JList (cont.)

- ▶ Unlike a JComboBox, a JList *does not* provide a scrollbar if there are more items in the list than the number of visible rows.
 - A JScrollPane object is used to provide the scrolling capability.
- ▶ `addListSelectionListener` registers a `ListSelectionListener` (package `javax.swing.event`) as the listener for a JList's selection events.

14.12 JList (cont.)

- ▶ 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** displays 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 color behind the GUI components, you must change the content pane's background color.
- ▶ Method **getContentPane** returns a reference to the **JFrame**'s content pane (an object of class **Container**).
- ▶ **List** method **getSelectedIndex** returns the selected item's index.

14.13 Multiple-Selection Lists

- ▶ A **multiple-selection list** enables the user to select many items from a `JList`.
- ▶ A `SINGLE_INTERVAL_SELECTION` list allows selecting a **contiguous range of items**.
 - To do so, click the first item, then press and **hold the *Shift* key** while clicking the last item in the range.
- ▶ A `MULTIPLE_INTERVAL_SELECTION` list (the default) allows continuous range selection as described for a `SINGLE_INTERVAL_SELECTION` list and 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.

```
1 // Fig. 14.25: MultipleSelectionFrame.java
2 // Copying items from one List to another.
3 import java.awt.FlowLayout;
4 import java.awt.event.ActionListener;
5 import java.awt.event.ActionEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JList;
8 import javax.swing.JButton;
9 import javax.swing.JScrollPane;
10 import javax.swing.ListSelectionModel;
11
12 public class MultipleSelectionFrame extends JFrame
13 {
14     private JList colorJList; // list to hold color names
15     private JList copyJList; // list to copy color names into
16     private JButton copyJButton; // button to copy selected names
17     private static final String[] colorNames = { "Black", "Blue", "Cyan",
18         "Dark Gray", "Gray", "Green", "Light Gray", "Magenta", "Orange",
19         "Pink", "Red", "White", "Yellow" };
20
21     // MultipleSelectionFrame constructor
22     public MultipleSelectionFrame()
23     {
```

Fig. 14.25 | JList that allows multiple selections. (Part 1 of 3.)

```
24 super( "Multiple Selection Lists" );
25 setLayout( new FlowLayout() ); // set frame layout
26
27 colorJList = new JList( colorNames ); // holds names of all colors
28 colorJList.setVisibleRowCount( 5 ); // show five rows
29 colorJList.setSelectionMode(
30     ListSelectionMode.MultipleIntervalSelection );
31 add( new JScrollPane( colorJList ) ); // add list with scrollpane
32
33 copyJButton = new JButton( "Copy >>>" ); // create copy button
34 copyJButton.addActionListener(
35
36     new ActionListener() // anonymous inner class
37     {
38         // handle button event
39         public void actionPerformed( ActionEvent event )
40         {
41             // place selected values in copyJList
42             copyJList.setListData( colorJList.getSelectedValues() );
43         } // end method actionPerformed
44     } // end anonymous inner class
45 ); // end call to addActionListener
46
47 add( copyJButton ); // add copy button to JFrame
```

Allow multiple selections or intervals in the JList.

Fig. 14.25 | JList that allows multiple selections. (Part 2 of 3.)

```
48
49     copyJList = new JList(); // create list to hold copied color names
50     copyJList.setVisibleRowCount( 5 ); // show 5 rows
51     copyJList.setFixedCellWidth( 100 ); // set width
52     copyJList.setFixedCellHeight( 15 ); // set height
53     copyJList.setSelectionMode(
54         ListSelectionMode.SINGLE_INTERVAL_SELECTION );
55     add( new JScrollPane( copyJList ) ); // add list with scrollpane
56 } // end MultipleSelectionFrame constructor
57 } // end class MultipleSelectionFrame
```

Allow only single intervals (ranges) to be selected.

Fig. 14.25 | JList that allows multiple selections. (Part 3 of 3.)

```
1 // Fig. 14.26: MultipleSelectionTest.java
2 // Testing MultipleSelectionFrame.
3 import javax.swing.JFrame;
4
5 public class MultipleSelectionTest
6 {
7     public static void main( String[] args )
8     {
9         MultipleSelectionFrame multipleSelectionFrame =
10             new MultipleSelectionFrame();
11         multipleSelectionFrame.setDefaultCloseOperation(
12             JFrame.EXIT_ON_CLOSE );
13         multipleSelectionFrame.setSize( 350, 150 ); // set frame size
14         multipleSelectionFrame.setVisible( true ); // display frame
15     } // end main
16 } // end class MultipleSelectionTest
```

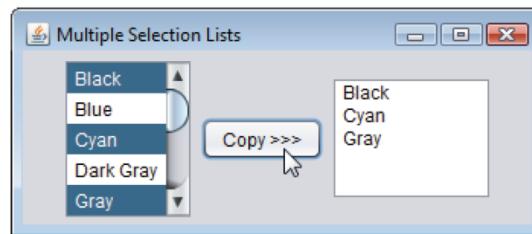


Fig. 14.26 | Test class for MultipleSelectionFrame.

14.13 Multiple-Selection Lists (cont.)

- ▶ If a `JList` does not contain items it will not display in a `FlowLayout`.
 - use `JList` methods `setFixedCellWidth` and `setFixedCellHeight` to set the item width and height
- ▶ There are no events to indicate that a user has made multiple selections in a multiple-selection list.
 - An event generated by another GUI component (known as an `external event`) specifies when the multiple selections in a `JList` should be processed.
- ▶ Method `setListData` sets the items displayed in a `JList`.
- ▶ Method `getSelectedValues` returns an array of `Objects` representing the selected items in a `JList`.

14.14 Mouse Event Handling

- ▶ `MouseListener` and `MouseMotionListener` event-listener interfaces for handling mouse events.
 - Any GUI component
- ▶ Package `javax.swing.event` contains interface `MouseListener`, which extends interfaces `MouseListener` and `MouseMotionListener` to create a single interface containing all the methods.
- ▶ `MouseListener` and `MouseMotionListener` methods are called when the mouse interacts with a `Component` if appropriate event-listener objects are registered for that `Component`.

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.

Fig. 14.27 | MouseListener and MouseMotionListener interface methods.

(Part 1 of 2.)

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

Fig. 14.27 | MouseListener and MouseMotionListener interface methods.
(Part 2 of 2.)

14.14 Mouse Event Handling (cont.)

- ▶ Each mouse event-handling method receives a `MouseEvent` object that contains **information about the mouse event** that occurred, including the x - and y -coordinates of the location where the event occurred.
- ▶ Coordinates are measured from the **upper-left corner** of the GUI component on which the event occurred.
- ▶ The x -coordinates start at 0 and increase from left to right. The y -coordinates start at 0 and increase from top to bottom.
- ▶ The methods and constants of class `InputEvent` (`MouseEvent`'s superclass) enable you to determine which mouse button the user clicked.

14.14 Mouse Event Handling (cont.)

- ▶ Interface `MouseWheelListener` enables applications to respond to the rotation of a mouse wheel.
- ▶ Method `mouseWheelMoved` 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.

```
1 // Fig. 14.28: MouseTrackerFrame.java
2 // Demonstrating mouse events.
3 import java.awt.Color;
4 import java.awt.BorderLayout;
5 import java.awt.event.MouseListener;
6 import java.awt.event.MouseMotionListener;
7 import java.awt.event.MouseEvent;
8 import javax.swing.JFrame;
9 import javax.swing.JLabel;
10 import javax.swing.JPanel;
11
12 public class MouseTrackerFrame extends JFrame
13 {
14     private JPanel mousePanel; // panel in which mouse events will occur
15     private JLabel statusBar; // label that displays event information
16
17     // MouseTrackerFrame constructor sets up GUI and
18     // registers mouse event handlers
19     public MouseTrackerFrame()
20     {
21         super( "Demonstrating Mouse Events" );
22     }
23 }
```

Fig. 14.28 | Mouse event handling. (Part I of 4.)

```

23 mousePanel = new JPanel(); // create panel
24 mousePanel.setBackground( Color.WHITE ); // set background color
25 add( mousePanel, BorderLayout.CENTER ); // add panel to JFrame
26
27 statusBar = new JLabel( "Mouse outside JPanel" );
28 add( statusBar, BorderLayout.SOUTH ); // add label to JFrame
29
30 // create and register listener for mouse and mouse motion events
31 MouseHandler handler = new MouseHandler();
32 mousePanel.addMouseListener( handler );
33 mousePanel.addMouseMotionListener( handler );
34 } // end MouseTrackerFrame constructor
35
36 private class MouseHandler implements MouseListener,
37     MouseMotionListener
38 {
39     // MouseListener event handlers
40     // handle event when mouse released immediately after press
41     public void mouseClicked( MouseEvent event )
42     {
43         statusBar.setText( String.format( "Clicked at [%d, %d]",
44             event.getX(), event.getY() ) );
45     } // end method mouseClicked
46

```

Object that handles both mouse events and mouse motion events.

An object of this class is a `MouseListener` and is a `MouseMotionListener`

Get the mouse coordinates at the time the click event occurred.

Fig. 14.28 | Mouse event handling. (Part 2 of 4.)

```
47 // handle event when mouse pressed
48 public void mousePressed( MouseEvent event )
49 {
50     statusBar.setText( String.format( "Pressed at [%d, %d]",
51         event.getX(), event.getY() ) );
52 } // end method mousePressed
53
54 // handle event when mouse released
55 public void mouseReleased( MouseEvent event )
56 {
57     statusBar.setText( String.format( "Released at [%d, %d]",
58         event.getX(), event.getY() ) );
59 } // end method mouseReleased
60
61 // handle event when mouse enters area
62 public void mouseEntered( MouseEvent event )
63 {
64     statusBar.setText( String.format( "Mouse entered at [%d, %d]",
65         event.getX(), event.getY() ) );
66     mousePanel.setBackground( Color.GREEN );
67 } // end method mouseEntered
68
```

Get the mouse coordinates at the time the pressed event occurred.

Get the mouse coordinates at the time the released event occurred.

Get the mouse coordinates at the time the entered event occurred then change the background to green.

Fig. 14.28 | Mouse event handling. (Part 3 of 4.)

```

69 // handle event when mouse exits area
70 public void mouseExited( MouseEvent event )
71 {
72     statusBar.setText( "Mouse outside JPanel" );
73     mousePanel.setBackground( Color.WHITE );
74 } // end method mouseExited
75
76 // MouseMotionListener event handlers
77 // handle event when user drags mouse with button pressed
78 public void mouseDragged( MouseEvent event )
79 {
80     statusBar.setText( String.format( "Dragged at [%d, %d]",
81         event.getX(), event.getY() ) );
82 } // end method mouseDragged
83
84 // handle event when user moves mouse
85 public void mouseMoved( MouseEvent event )
86 {
87     statusBar.setText( String.format( "Moved at [%d, %d]",
88         event.getX(), event.getY() ) );
89 } // end method mouseMoved
90 } // end inner class MouseHandler
91 } // end class MouseTrackerFrame

```

Change the background to white when the mouse exits the area.

Get the mouse coordinates at the time the dragged event occurred.

Get the mouse coordinates at the time the moved event occurred.

Fig. 14.28 | Mouse event handling. (Part 4 of 4.)

```
1 // Fig. 14.29: MouseTrackerFrame.java
2 // Testing MouseTrackerFrame.
3 import javax.swing.JFrame;
4
5 public class MouseTracker
6 {
7     public static void main( String[] args )
8     {
9         MouseTrackerFrame mouseTrackerFrame = new MouseTrackerFrame();
10        mouseTrackerFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        mouseTrackerFrame.setSize( 300, 100 ); // set frame size
12        mouseTrackerFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class MouseTracker
```

Fig. 14.29 | Test class for MouseTrackerFrame. (Part 1 of 2.)

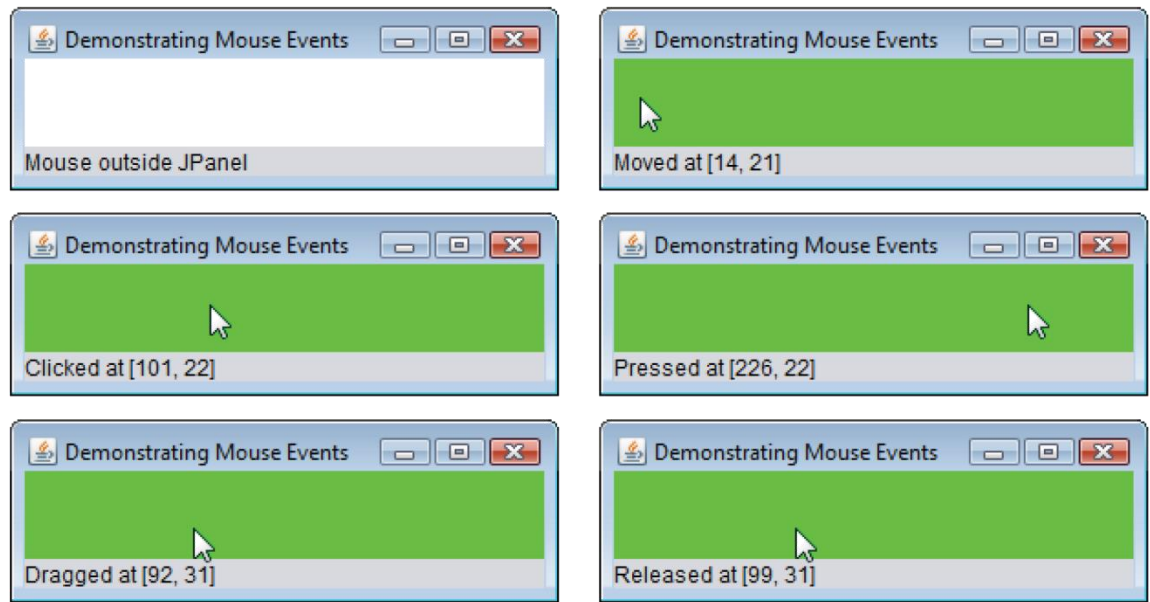


Fig. 14.29 | Test class for MouseTrackerFrame. (Part 2 of 2.)

14.14 Mouse Event Handling (cont.)

- ▶ `BorderLayout` arranges components into five regions: `NORTH`, `SOUTH`, `EAST`, `WEST` and `CENTER`.
- ▶ `BorderLayout` sizes the component in the `CENTER` to use all available space that is not occupied
- ▶ Methods `addMouseListener` and `addMouseMotionListener` register `MouseListener`s and `MouseMotionListener`s, respectively.
- ▶ `MouseEvent` methods `getX` and `getY` return the x - and y -coordinates of the mouse at the time the event occurred.

14.15 Adapter Classes

- ▶ Many event-listener interfaces contain multiple methods.
- ▶ An **adapter class** implements an interface and provides a **default implementation** (with an **empty method body**) of each method in the interface.
- ▶ You **extend an adapter class** to inherit the default implementation of every method and **override only the method(s) you need** for event handling.



Software Engineering Observation 14.8

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`).

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

Fig. 14.30 | Event-adapter classes and the interfaces they implement in package `java.awt.event`.

```
1 // Fig. 14.31: MouseDetailsFrame.java
2 // Demonstrating mouse clicks and distinguishing between mouse buttons.
3 import java.awt.BorderLayout;
4 import java.awt.event.MouseAdapter;
5 import java.awt.event.MouseEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JLabel;
8
9 public class MouseDetailsFrame extends JFrame
10 {
11     private String details; // String that is displayed in the statusBar
12     private JLabel statusBar; // JLabel that appears at bottom of window
13
14     // constructor sets title bar String and register mouse listener
15     public MouseDetailsFrame()
16     {
17         super( "Mouse clicks and buttons" );
18
19         statusBar = new JLabel( "Click the mouse" );
20         add( statusBar, BorderLayout.SOUTH );
21         addMouseListener( new MouseClickListener() ); // add handler
22     } // end MouseDetailsFrame constructor
23
```

Fig. 14.31 | Left, center and right mouse-button clicks. (Part 1 of 2.)

```

24 // inner class to handle mouse events
25 private class MouseClickHandler extends MouseAdapter
26 {
27     // handle mouse-click event and determine which button was pressed
28     public void mouseClicked( MouseEvent event )
29     {
30         int xPos = event.getX(); // get x-position of mouse
31         int yPos = event.getY(); // get y-position of mouse
32
33         details = String.format( "Clicked %d time(s)",
34             event.getClickCount() );
35
36         if ( event.isMetaDown() ) // right mouse button
37             details += " with right mouse button";
38         else if ( event.isAltDown() ) // middle mouse button
39             details += " with center mouse button";
40         else // left mouse button
41             details += " with left mouse button";
42
43         statusBar.setText( details ); // display message in statusBar
44     } // end method mouseClicked
45 } // end private inner class MouseClickHandler
46 } // end class MouseDetailsFrame

```

Adapter enables us to override the one method we use in this example.

Returns the number of mouse clicks. If you wait long enough between clicks, the count resets to 0.

Help determine which button the user pressed on the mouse.

Fig. 14.31 | Left, center and right mouse-button clicks. (Part 2 of 2.)

```
1 // Fig. 14.32: MouseDetails.java
2 // Testing MouseDetailsFrame.
3 import javax.swing.JFrame;
4
5 public class MouseDetails
6 {
7     public static void main( String[] args )
8     {
9         MouseDetailsFrame mouseDetailsFrame = new MouseDetailsFrame();
10        mouseDetailsFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        mouseDetailsFrame.setSize( 400, 150 ); // set frame size
12        mouseDetailsFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class MouseDetails
```

Fig. 14.32 | Test class for MouseDetailsFrame. (Part 1 of 2.)

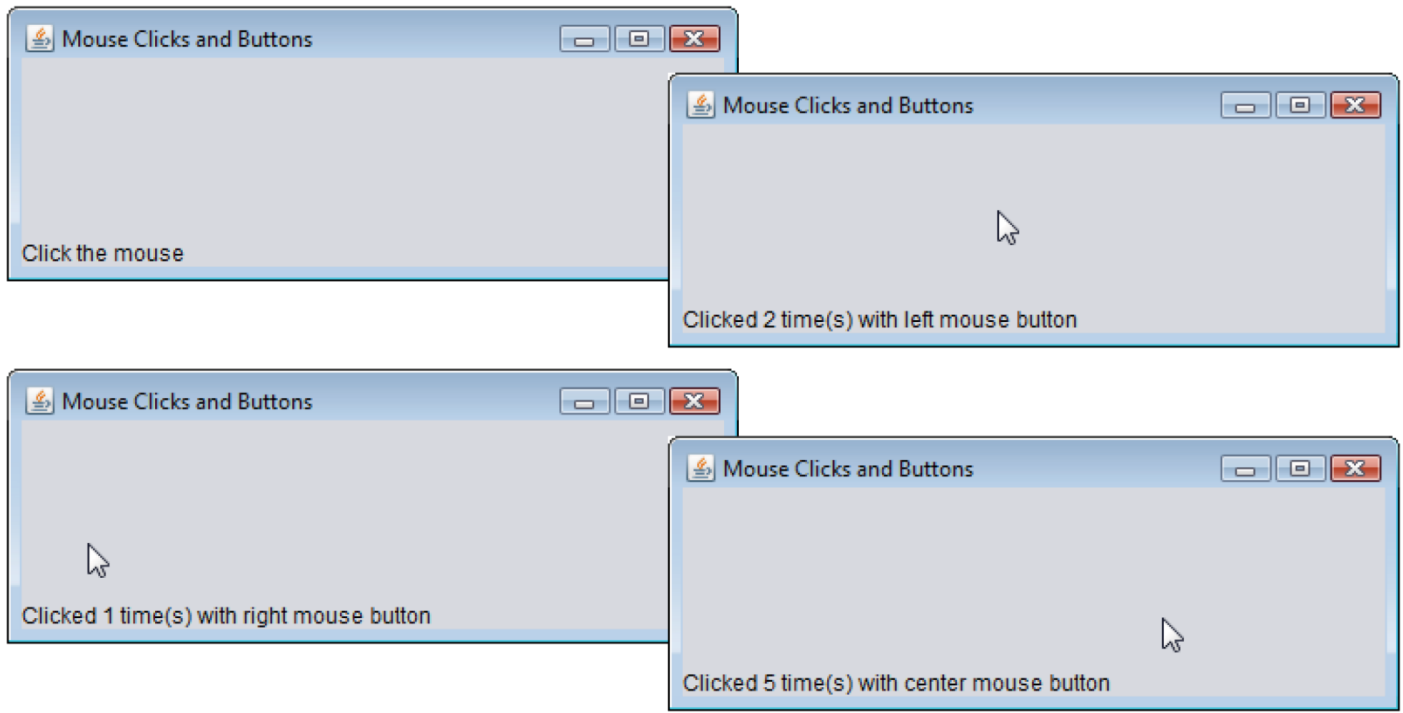


Fig. 14.32 | Test class for MouseDetailsFrame. (Part 2 of 2.)

14.17 Key Event Handling

- ▶ `KeyListener` interface for handling `key events`.
- ▶ Key events are generated when keys on the keyboard are pressed and released.
- ▶ A `KeyListener` must define methods `keyPressed`, `keyReleased` and `keyTyped`
 - each receives a `KeyEvent` as its argument
- ▶ Class `KeyEvent` is a subclass of `InputEvent`.
- ▶ Method `keyPressed` is called in response to pressing any key.
- ▶ Method `keyTyped` is called in response to pressing any key that is not an `action key` (ex. `copy`, `paste`, or `F1`, `F2`, etc)
- ▶ Method `keyReleased` is called when the key is released after any `keyPressed` or `keyTyped` event.

```
1 // Fig. 14.36: KeyDemoFrame.java
2 // Demonstrating keystroke events.
3 import java.awt.Color;
4 import java.awt.event.KeyListener;
5 import java.awt.event.KeyEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JTextArea;
8
9 public class KeyDemoFrame extends JFrame implements KeyListener
10 {
11     private String line1 = ""; // first line of textarea
12     private String line2 = ""; // second line of textarea
13     private String line3 = ""; // third line of textarea
14     private JTextArea textArea; // textarea to display output
15
16     // KeyDemoFrame constructor
17     public KeyDemoFrame()
18     {
19         super( "Demonstrating Keystroke Events" );
20
21         textArea = new JTextArea( 10, 15 ); // set up JTextArea
22         textArea.setText( "Press any key on the keyboard..." );
23         textArea.setEnabled( false ); // disable textarea
```

This class can handle its own KeyEvents.

Fig. 14.36 | Key event handling. (Part I of 3.)

```

24     textArea.setDisabledTextColor( Color.BLACK ); // set text color
25     add( textArea ); // add textarea to JFrame
26
27     addKeyListener( this ); // allow frame to process key events
28 } // end KeyDemoFrame constructor
29
30 // handle press of any key
31 public void keyPressed( KeyEvent event )
32 {
33     line1 = String.format( "Key pressed: %s",
34         KeyEvent.getKeyText( event.getKeyCode() ) ); // show pressed key
35     setLines2and3( event ); // set output lines two and three
36 } // end method keyPressed
37
38 // handle release of any key
39 public void keyReleased( KeyEvent event )
40 {
41     line1 = String.format( "Key released: %s",
42         KeyEvent.getKeyText( event.getKeyCode() ) ); // show released key
43     setLines2and3( event ); // set output lines two and three
44 } // end method keyReleased
45

```

Registers the object of this class as the event handler.

Gets text of pressed key.

Gets text of pressed key.

Fig. 14.36 | Key event handling. (Part 2 of 3.)

```

46 // handle press of an action key
47 public void keyTyped( KeyEvent event )
48 {
49     line1 = String.format( "Key typed: %s", event.getKeyChar() );
50     setLines2and3( event ); // set output lines two and three
51 } // end method keyTyped
52
53 // set second and third lines of output
54 private void setLines2and3( KeyEvent event )
55 {
56     line2 = String.format( "This key is %san action key",
57         ( event.isActionKey() ? "" : "not " ) );
58
59     String temp = KeyEvent.getKeyModifiersText( event.getModifiers() );
60
61     line3 = String.format( "Modifier keys pressed: %s",
62         ( temp.equals( "" ) ? "none" : temp ) ); // output modifiers
63
64     textArea.setText( String.format( "%s\n%s\n%s\n",
65         line1, line2, line3 ) ); // output three lines of text
66 } // end method setLines2and3
67 } // end class KeyDemoFrame

```

← Gets text of pressed modifier keys.

Fig. 14.36 | Key event handling. (Part 3 of 3.)

```
1 // Fig. 14.37: KeyDemo.java
2 // Testing KeyDemoFrame.
3 import javax.swing.JFrame;
4
5 public class KeyDemo
6 {
7     public static void main( String[] args )
8     {
9         KeyDemoFrame keyDemoFrame = new KeyDemoFrame();
10        keyDemoFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        keyDemoFrame.setSize( 350, 100 ); // set frame size
12        keyDemoFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class KeyDemo
```

Fig. 14.37 | Test class for KeyDemoFrame. (Part 1 of 2.)

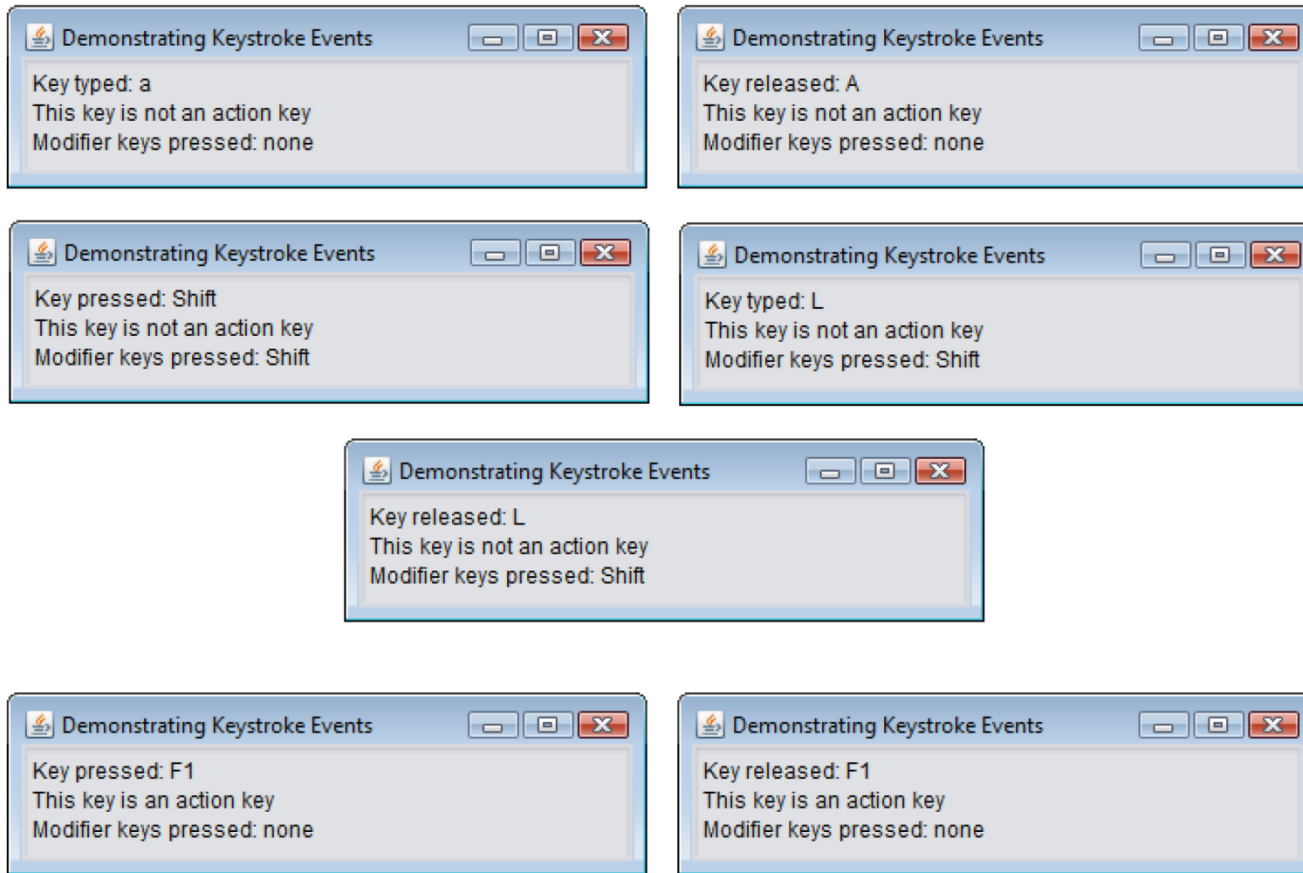


Fig. 14.37 | Test class for KeyDemoFrame. (Part 2 of 2.)

14.17 Key Event Handling (cont.)

- ▶ Registers key event handlers with method `addKeyListener` from class `Component`.
- ▶ `KeyEvent` method `getKeyCode` gets the **virtual key code** of the pressed key.
- ▶ `KeyEvent` contains virtual key-code constants that represents every key on the keyboard.
- ▶ Value returned by `getKeyCode` can be passed to `static KeyEvent` method `getKeyText` to get a string containing the name of the key that was pressed.
- ▶ `KeyEvent` method `getKeyChar` (which returns a `char`) gets the Unicode value of the character typed.
- ▶ `KeyEvent` method `isActionKey` determines whether the key in the event was an action key.

14.17 Key Event Handling (cont.)

- ▶ Method `getModifiers` determines whether any modifier keys (such as *Shift*, *Alt* and *Ctrl*) were pressed when the key event occurred.
 - Result can be passed to `static KeyEvent` method `getKeyModifiersText` to get a string containing the names of the pressed modifier keys.
- ▶ `InputEvent` methods `isAltDown`, `isControlDown`, `isMetaDown` and `isShiftDown` each return a `boolean` indicating whether the particular key was pressed during the key event.

Lab Session

- ▶ Ex. 1. *Temperature Conversion*). Write a temperature–conversion application that converts from Fahrenheit to Celsius.
- ▶ The Fahrenheit temperature should be entered from the keyboard (via a JTextField).
- ▶ A JLabel should be used to display the converted temperature. Use the following formula for the conversion:
- ▶ $Celsius = 5/9 \times (Fahrenheit - 32)$

Ex. 2

- ▶ ***Temperature–Conversion Modification.***
Enhance the temperature–conversion application of Exercise 1 by adding the Kelvin temperature scale.
- ▶ The application should also allow the user to make conversions between any two scales.
- ▶ Use the following formula for the conversion between Kelvin and Celsius (in addition to the formula in Exercise 1):
- ▶ $Kelvin = Celsius + 273.15$

Ex. 3

- ▶ *Guess-the-Number Game*.
- ▶ Write an application that plays “guess the number” as follows:
- ▶ Your application chooses the number to be guessed by selecting an integer at random in the range 1–1000. The application then displays the following in a label:
 - ▶ I have a number between 1 and 1000. Can you guess my number?
 - ▶ Please enter your first guess.
 - ▶ A JTextField should be used to input the guess.
 - ▶ As each guess is input, the background color should change to either red or blue. Red indicates that the user is getting “warmer,” and blue, “colder.” A JLabel should display either "Too High" or "Too Low" to help the user zero in.
 - ▶ When the user gets the correct answer, "Correct!" should be displayed, and the JTextField used for input should be changed to be uneditable.
 - ▶ A JButton should be provided to allow the user to play the game again.
 - ▶ When the JButton is clicked, a new random number should be generated and the input JTextField changed to be editable.

End of class – GUI PART I