



# Chapter 12

# GUI Components: Part 1

Java How to Program, 10/e



## OBJECTIVES

In this chapter you'll:

- Learn how to use the Nimbus look-and-feel.
- Build GUIs and handle events generated by user interactions with GUIs.
- Understand the packages containing GUI components, event-handling classes and interfaces.
- Create and manipulate buttons, labels, lists, text fields and panels.
- Handle mouse events and keyboard events.
- Use layout managers to arrange GUI components.



- 
- 12.1** Introduction
  - 12.2** Java's Nimbus Look-and-Feel
  - 12.3** Simple GUI-Based Input/Output with `JOptionPane`
  - 12.4** Overview of Swing Components
  - 12.5** Displaying Text and Images in a Window
  - 12.6** Text Fields and an Introduction to Event Handling with Nested Classes
  - 12.7** Common GUI Event Types and Listener Interfaces
  - 12.8** How Event Handling Works
  - 12.9** `JButton`
  - 12.10** Buttons That Maintain State
    - 12.10.1 `JCheckBox`
    - 12.10.2 `JRadioButton`
  - 12.11** `JComboBox`; Using an Anonymous Inner Class for Event Handling
  - 12.12** `JList`
  - 12.13** Multiple-Selection Lists
  - 12.14** Mouse Event Handling
-



---

**12.15 Adapter Classes**

**12.16 JPanel Subclass for Drawing with the Mouse**

**12.17 Key Event Handling**

**12.18 Introduction to Layout Managers**

12.18.1 **FlowLayout**

12.18.2 **BorderLayout**

12.18.3 **GridLayout**

**12.19 Using Panels to Manage More Complex Layouts**

**12.20 JTextArea**

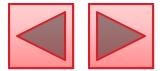
**12.21 Wrap-Up**

---



## 12.1 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.



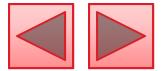
## Look-and-Feel Observation 12.1

*Providing different applications with consistent, intuitive user-interface components gives users a sense of familiarity with a new application, so that they can learn it more quickly and use it more productively.*



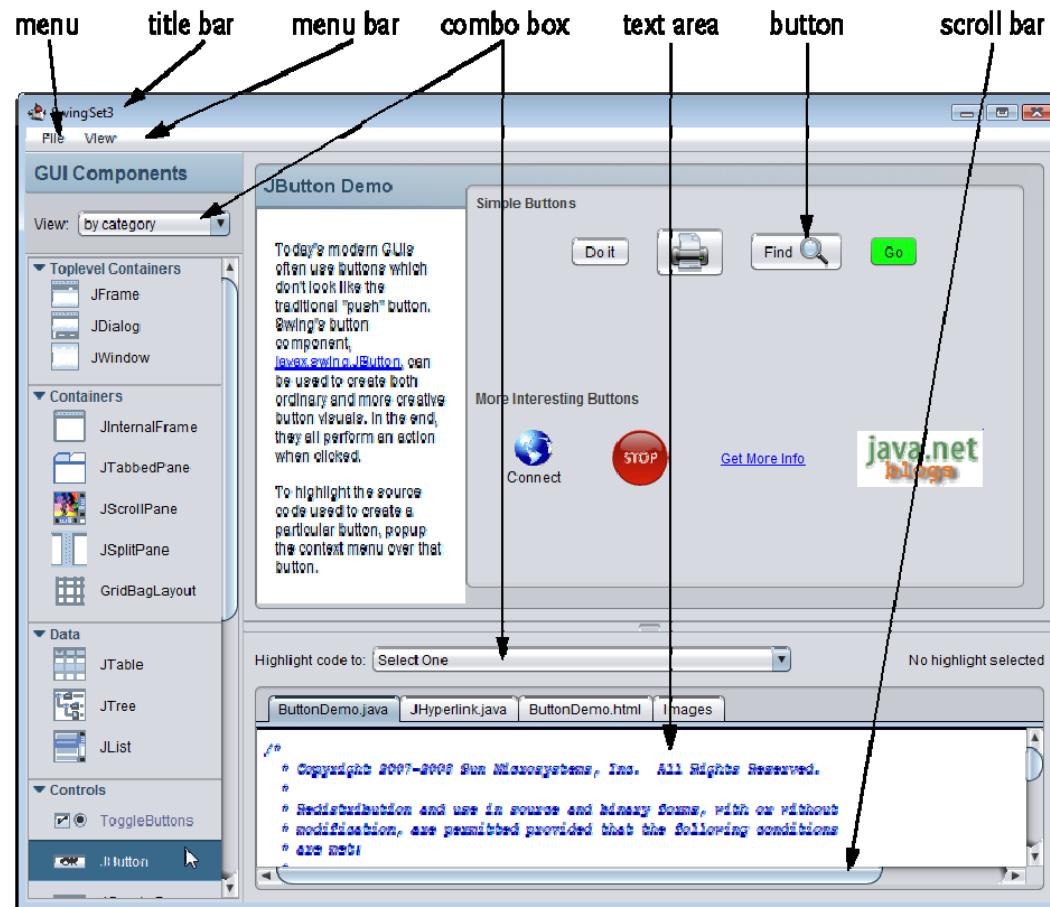
## 12.1 Introduction (cont.)

- ▶ 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 *size*, *location* and other attributes in a visual manner by using the mouse, keyboard and drag-and-drop.
  - Generate the GUI code for you.
  - Greatly simplify creating GUIs, but each IDE has different capabilities and generates different code.



## 12.1 Introduction (cont.)

- ▶ Example of a GUI: SwingSet3 application (Fig. 12.1)  
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- ▶ **title bar** at top contains the window's title.
- ▶ **menu bar** contains **menus** (**File** and **View**).
- ▶ In the top-right region of the window is a set of **buttons**
  - Typically, users press buttons to perform tasks.
- ▶ In the **GUI Components** area of the window is a **combo box**;
  - User can click the down arrow at the right side of the box to select from a list of items.

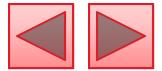


**Fig. 12.1 |** **SwingSet3** application demonstrates many of Java's Swing GUI components.



## 12.2 Java's Nimbus Look-and-Feel

- ▶ Swing has a cross-platform look-and-feel known as [Nimbus](#).
- ▶ We've configured our systems to use Nimbus as the default look-and-feel.



## 12.2 Java's Nimbus Look-and-Feel (cont.)

- ▶ Three ways to use Nimbus:
  - Set it as the default for all Java applications that run on your computer.
  - Set it as the look-and-feel when you launch an application by passing a command-line argument to the `java` command.
  - Set it as the look-and-feel programmatically in your application (Section 22.6).



## 12.2 Java's Nimbus Look-and-Feel (cont.)

- ▶ To set Nimbus as the default for all Java applications:
  - Create a text file named `swing.properties` in the `lib` folder of both your JDK installation folder and your JRE installation folder.
  - Place the following line of code in the file:  
`swing.defaultlaf=`  
`com.sun.java.swing.plaf.nimbus.`  
`NimbusLookAndFeel`
- ▶ In addition to the standalone JRE, there is a JRE nested in your JDK's installation folder. If you are using an IDE that depends on the JDK (e.g., NetBeans), you may also need to place the `swing.properties` file in the nested `jre` folder's `lib` folder.



## 12.2 Java's Nimbus Look-and-Feel (cont.)

- ▶ To select Nimbus on an application-by-application basis:
  - Place the following command-line argument after the `java` command and before the application's name when you run the application:  
`-Dswing.defaultLookAndFeel=`  
`com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel`



## 12.3 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 12.2 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. 12.2: Addition.java
2 // Addition program that uses JOptionPane for input and output.
3 import javax.swing.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;
20    }
}
```

---

**Fig. 12.2** | Addition program that uses JOptionPane for input and output. (Part I of 3.)

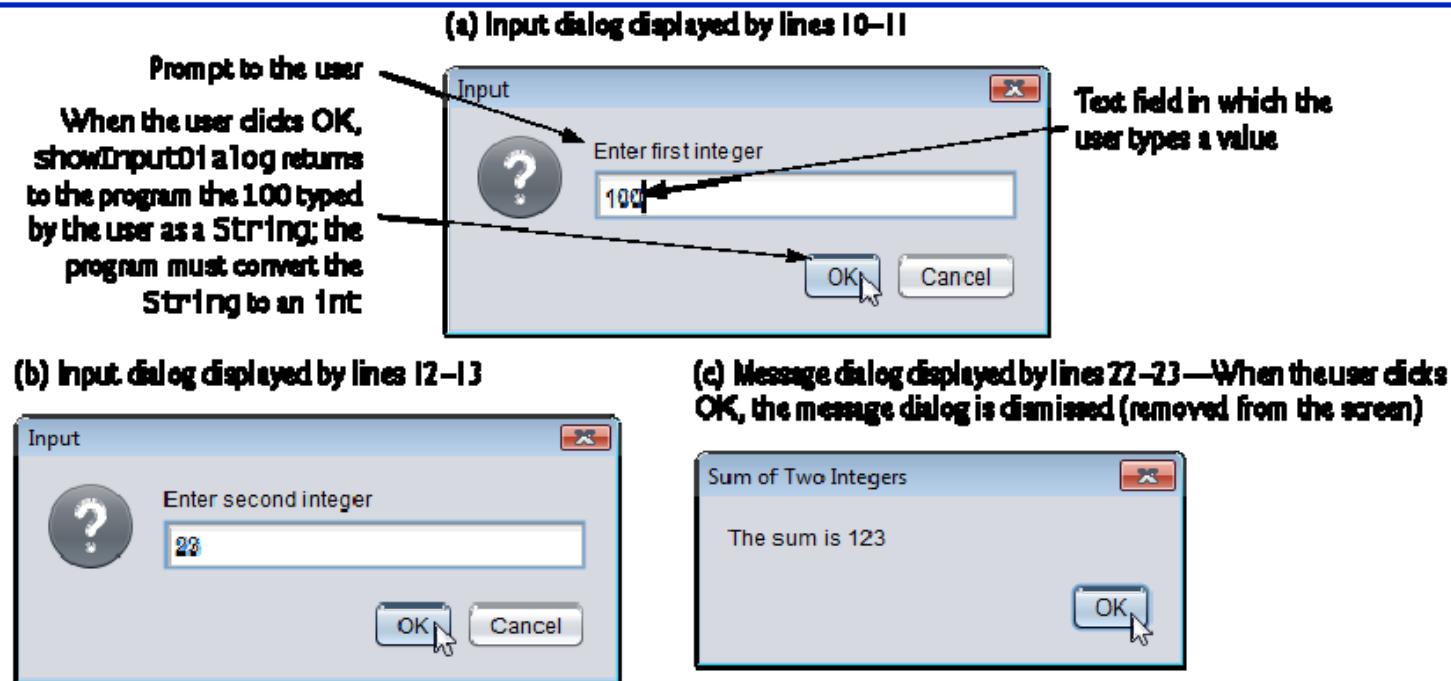


---

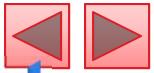
```
21     // display result in a JOptionPane message dialog
22     JOptionPane.showMessageDialog(null, "The sum is " + sum,
23         "Sum of Two Integers", JOptionPane.PLAIN_MESSAGE);
24 }
25 } // end class Addition
```

---

**Fig. 12.2** | Addition program that uses `JOptionPane` for input and output. (Part 2 of 3.)

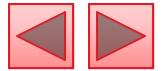


**Fig. 12.2** | Addition program that uses JOptionPane for input and output. (Part 3 of 3.)



## 12.3 Simple GUI-Based Input/Output with JOptionPane (cont.)

- ▶ `JOptionPane static method showInputDialog` displays an input dialog, using the method's `String` argument as a prompt.
  - The user types characters in the text field, then clicks `OK` or presses the *Enter* key to submit the `String` to the program.
  - Clicking `OK` `dismisses (hides)` the dialog.
  - Can input only `Strings`. Typical of most GUI components.
  - If the user clicks `Cancel`, returns `null`.
  - `JOptionPane` dialog are `dialog`—the user cannot interact with the rest of the application while dialog is displayed.



## Look-and-Feel Observation 12.2

*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, Jones).*



## Look-and-Feel Observation 12.3

*Do not overuse modal dialogs, as they can reduce the usability of your applications. Use a modal dialog only when it's necessary to prevent users from interacting with the rest of an application until they dismiss the dialog.*

## 12.3 Simple GUI-Based Input/Output with JOptionPane (cont.)



- ▶ Converting Strings to int Values
  - Integer class's static method parseInt converts its String argument to an int value and might throw a NumberFormatException.
- ▶ Message Dialogs
  - JOptionPane static method showMessageDialog displays a message dialog.
  - The first argument helps determine where to position the dialog.
    - If null, the dialog box is displayed at the center of your screen.
  - The second argument is the message to display.
  - The third argument is the String that should appear in the title bar at the top of the dialog.
  - The fourth argument is the type of message dialog to display.



## 12.3 Simple GUI-Based Input/Output with JOptionPane (cont.)

- ▶ Message Dialogs
  - A `JOptionPane.PLAIN_MESSAGE` dialog does not display an icon to the left of the message.
- ▶ `JOptionPane` online documentation:
  - <http://docs.oracle.com/javase/7/docs/api/java/swing/JOptionPane.html>



## Look-and-Feel Observation 12.4

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		Indicates an error.
INFORMATION_MESSAGE		Indicates an informational message.
WARNING_MESSAGE		Warns of a potential problem.
QUESTION_MESSAGE		Poses a question. 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. 12.3** | JOptionPane static constants for message dialogs.



## 12.4 Overview of Swing Components

- ▶ Swing GUI components located in package `javax.swing`.
- ▶ 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.



Component	Description
JLabel	Displays <i>uneditable text</i> and/or icons.
JTextField	Typically <i>receives input</i> from the user.
JButton	Triggers an event when <i>clicked with the mouse</i> .
JCheckBox	Specifies an option that can be <i>selected or not selected</i> .
JComboBox	A <i>drop-down list of items</i> from which the <i>user</i> can make a <i>selection</i> .
JList	A <i>list of items</i> from which the <i>user</i> can make a <i>selection</i> by <i>clicking on any one</i> of them. <i>Multiple elements</i> can be selected.
JPanel	An area in which <i>components</i> can be <i>placed and organized</i> .

**Fig. 12.4** | Some basic Swing GUI components.



## 12.4 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.



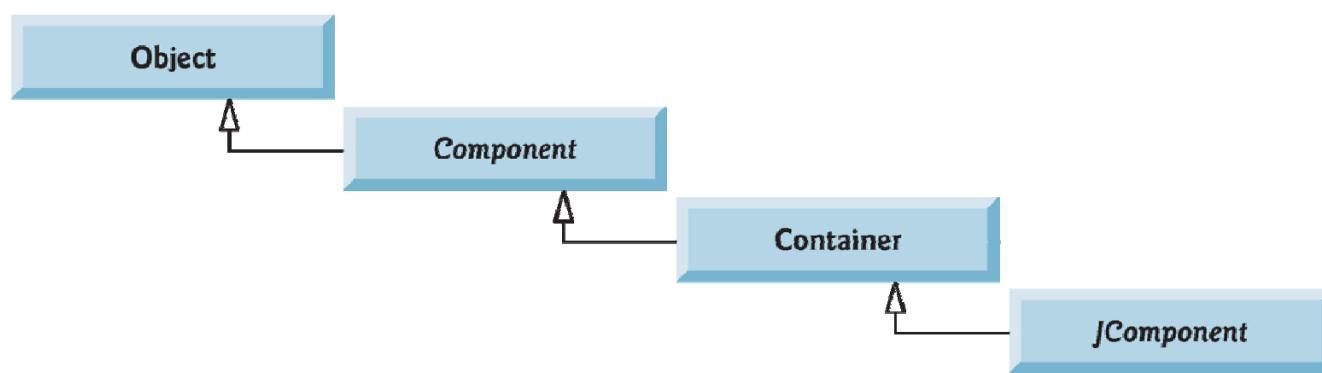
## 12.4 Overview of Swing Components (cont.)

- ▶ 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.



## Look-and-Feel Observation 12.5

*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. An application can even change the look-and-feel during execution to enable users to choose their own preferred look-and-feel.*



**Fig. 12.5** | Common superclasses of the lightweight Swing components.



## 12.4 Overview of Swing Components (cont.)

- ▶ Class **Container** (package `j ava. 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.



## 12.4 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`.



## 12.4 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`
  - Support for accessibility
  - Support for user-interface `localization`



## 12.5 Displaying Text and Images in a Window

- ▶ 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.



## 12.5 Displaying Text and Images in a Window (cont.)

- ▶ In a large GUI
  - Difficult to identify the purpose of every component.
  - Provide text stating each component's purpose.
- ▶ Such text is known as a **label** and is created with class **JLabel**—a subclass of **JComponent**.
  - Displays read-only text, an image, or both text and an image.



## Look-and-Feel Observation 12.6

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



```
1 // Fig. 12.6: LabelFrame.java
2 // JLabels with text and icons.
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 final JLabel label1; // JLabel with just text
13     private final JLabel label2; // JLabel constructed with text and icon
14     private final 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         // 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
```

**Fig. 12.6 |** JLabels with text and icons. (Part 1 of 2.)



---

```
26
27 // JLabel constructor with string, Icon and alignment arguments
28 Icon bug = new ImageIcon(getClass().getResource( "bug1.png"));
29 label12 = new JLabel("Label with text and icon", bug,
30     SwingConstants.LEFT);
31 label12.setToolTipText("This is label12");
32 add(label12); // add label12 to JFrame
33
34 label13 = new JLabel(); // JLabel constructor no arguments
35 label13.setText("Label with icon and text at bottom");
36 label13.setIcon(bug); // add icon to JLabel
37 label13.setHorizontalTextPosition(SwingConstants.CENTER);
38 label13.setVerticalTextPosition(SwingConstants.BOTTOM);
39 label13.setToolTipText("This is label13");
40 add(label13); // add label13 to JFrame
41 }
42 } // end class LabelFrame
```

---

**Fig. 12.6** | **JLabels with text and icons.** (Part 2 of 2.)



---

```
1 // Fig. 12.7: LabelTest.java
2 // Testing LabelFrame.
3 import javax.swing.JFrame;
4
5 public class LabelTest
6 {
7     public static void main(String[] args)
8     {
9         LabelFrame labelFrame = new LabelFrame();
10        labelFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        labelFrame.setSize(260, 180);
12        labelFrame.setVisible(true);
13    }
14 } // end class LabelTest
```

---

**Fig. 12.7** | Testing LabelFrame. (Part 1 of 2.)



**Fig. 12.7 | Testing LabelFrame. (Part 2 of 2.)**



## 12.5 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.



## 12.5 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 this book's examples can be used with *any* IDE, we did *not* use an IDE to create the GUI code
- ▶ We use Java's layout managers in our GUI examples



## 12.5 Displaying Text and Images in a Window (cont.)

### ► `FlowLayout`

- GUI components are placed in 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`).



## 12.5 Displaying Text and Images in a Window (cont.)

- ▶ `JLabel` constructor can receive a `String` specifying the label's text.
- ▶ Method `setToolTipText` (inherited by `JLabel` from `JComponent`) specifies the tool tip that is displayed when the user positions the mouse cursor over a `JComponent` (such as a `JLabel` ).
- ▶ You attach a component to a container using the `add` method, which is inherited indirectly from class `Container`.



## Common Programming Error 12.1

*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 12.7

*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.*



## 12.5 Displaying Text and Images in a Window (cont.)

- ▶ Icons enhance the look-and-feel of an application and are also commonly used to indicate functionality.
- ▶ An icon is normally specified with an `Icon` (package `javax.swing`) argument to a constructor or to the component's `setIcon` method.
- ▶ `ImageIcon` (package `javax.swing`) supports several image formats, including **Graphics Interchange Format (GIF)**, **Portable Network Graphics (PNG)** and **Joint Photographic Experts Group (JPEG)**.



## 12.5 Displaying Text and Images in a Window (cont.)

- ▶ `getClass().getResource("bug1.png")`
  - Invokes method `getClass` (inherited indirectly from class `Object`) to retrieve a reference to the `Class` object that represents the `LabelFrame` class declaration.
  - Next, invokes `Class` method `getResource`, which returns the location of the image as a URL.
  - The `ImageIcon` constructor uses the URL to locate the image, then loads it into memory.
  - The class loader knows where each class it loads is located on disk. Method `getResource` uses the `Class` object's class loader to determine the *location* of a resource, such as an image file.



## 12.5 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`, `SwingConstants.CENTER` and `SwingConstants.RIGHT`) 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.



## 12.5 Displaying Text and Images in a Window (cont.)

- ▶ Class `JLabel` provides methods to change a `JLabel` 's appearance after it has been instantiated.
- ▶ Method `setText` sets the text displayed on the label.
- ▶ Method `getText` retrieves the `JLabel` 's current text.
- ▶ Method `setIcon` specifies the icon to display.
- ▶ Method `getIcon` retrieves the current icon displayed on a label.
- ▶ Methods `setHorizontalTextPosition` and `setVerticalTextPosition` specify the text position in the label.



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

**Fig. 12.8** | Positioning constants (static members of interface `SwingConstants`).



## 12.5 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.



## 12.6 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 process of responding to events is known as **event handling**.



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

- ▶ `JTextFields` and `JPasswordField`s (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. 12.9: TextFieldFrame.java
2 // JTextFields and JPasswordField.
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 final JTextField textField1; // text field with set size
14     private final JTextField textField2; // text field with text
15     private final JTextField textField3; // text field with text and size
16     private final 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());
23     }

```

---

**Fig. 12.9** | JTextFields and JPasswordField. (Part 1 of 4.)



```
24 // construct text field with 10 columns
25 textField1 = new JTextField(10);
26 add(textField1); // add textField1 to JFrame
27
28 // construct text field with default text
29 textField2 = new JTextField("Enter text here");
30 add(textField2); // add textField2 to JFrame
31
32 // construct text field 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 password field 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 }
```

**Fig. 12.9** | **JTextFields 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     @Override
54     public void actionPerformed(ActionEvent event)
55     {
56         String string = "";
57
58         // user pressed Enter in JTextField textField1
59         if (event.getSource() == textField1)
60             string = String.format("textField1: %s",
61                 event.getActionCommand());
62
63         // user pressed Enter in JTextField textField2
64         else if (event.getSource() == textField2)
65             string = String.format("textField2: %s",
66                 event.getActionCommand());
67
68         // user pressed Enter in JTextField textField3
69         else if (event.getSource() == textField3)
70             string = String.format("textField3: %s",
71                 event.getActionCommand());
```

**Fig. 12.9** | JTextFields and JPasswordField. (Part 3 of 4.)



---

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

---

**Fig. 12.9** | JTextFields and JPasswordFields. (Part 4 of 4.)



## Software Engineering Observation 12.1

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



## Common Programming Error 12.2

*If you forget to register an event-handler object for a particular GUI component's event type, events of that type will be ignored.*



## 12.6 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.



## 12.6 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](#).



## 12.6 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.



## 12.6 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.



## 12.6 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.



## 12.6 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.



## 12.6 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.



## 12.6 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`.

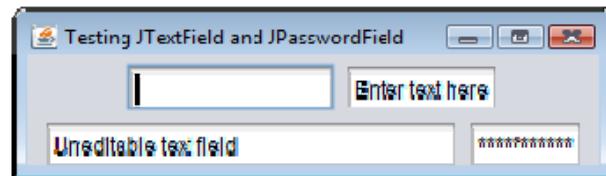


## 12.6 Text Fields and an Introduction to Event Handling with Nested Classes (cont.)

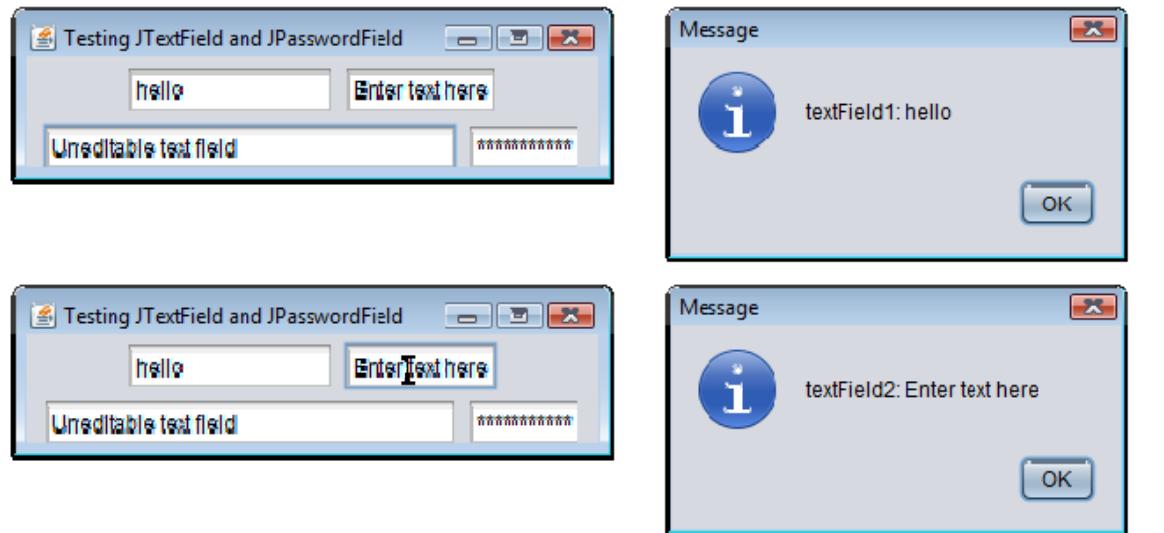
- ▶ The component with which the user interacts is the `event source`.
- ▶ `ActionEvent` method `getSource` (inherited from class `EventObject`) returns a reference to the event source.
- ▶ `ActionEvent` method `getActionCommand` obtains the text the user typed in the text field that generated the event.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



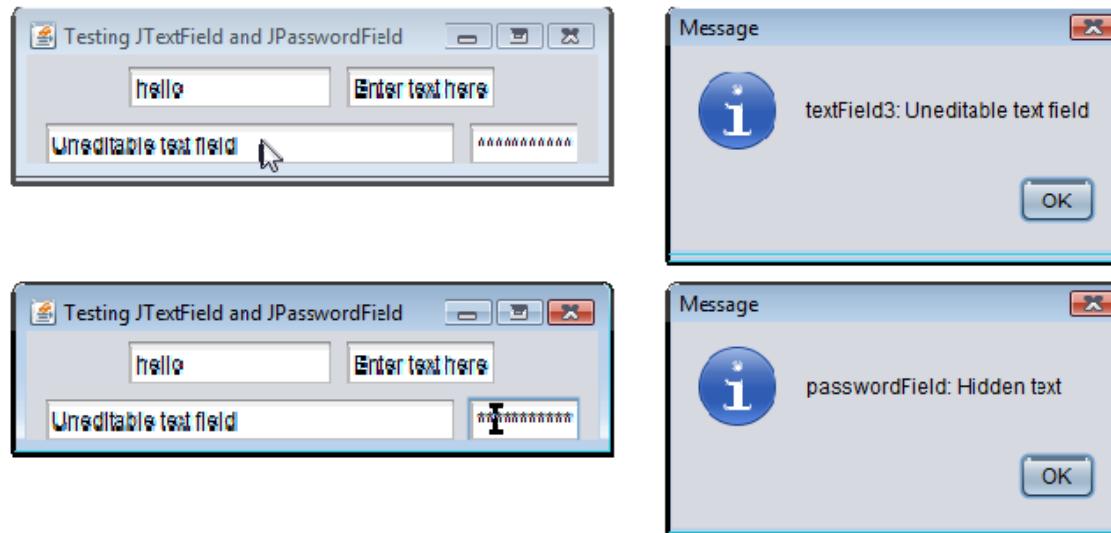
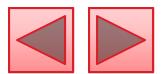
```
1 // Fig. 12.10: TextFieldTest.java
2 // Testing JTextFieldFrame.
3 import javax.swing.JFrame;
4
5 public class TextFieldTest
6 {
7     public static void main(String[] args)
8     {
9         JTextFieldFrame textFieldFrame = new JTextFieldFrame();
10        textFieldFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        textFieldFrame.setSize(350, 100);
12        textFieldFrame.setVisible(true);
13    }
14 } // end class TextFieldTest
```



**Fig. 12.10 |** Testing JTextFieldFrame. (Part 1 of 3.)



**Fig. 12.10 | Testing TextFieldFrame (Part 2 of 3.)**

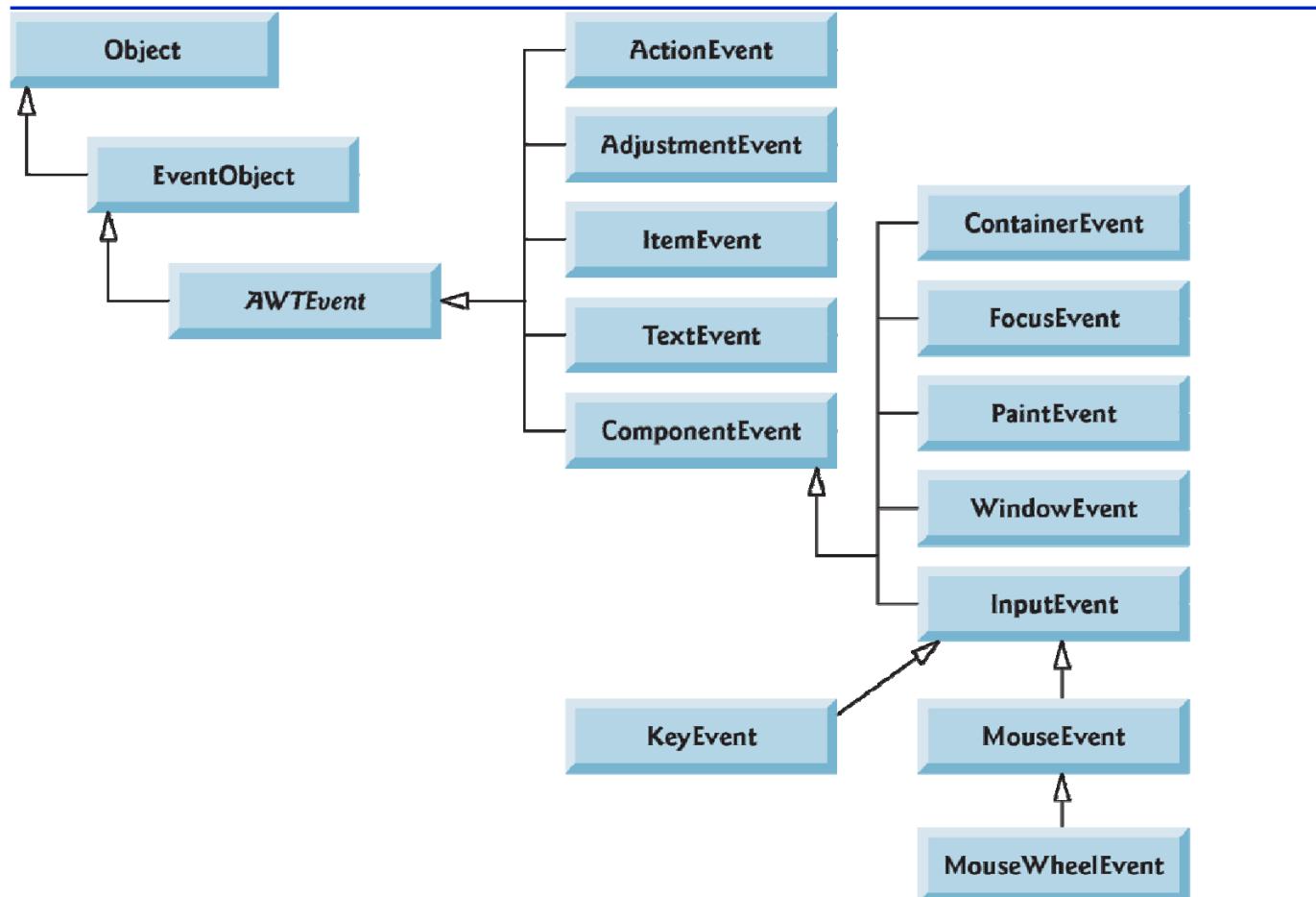


**Fig. 12.10 | Testing TextFieldFrame (Part 3 of 3.)**



## 12.7 Common GUI Event Types and Listener Interfaces

- ▶ Figure 12.11 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`.

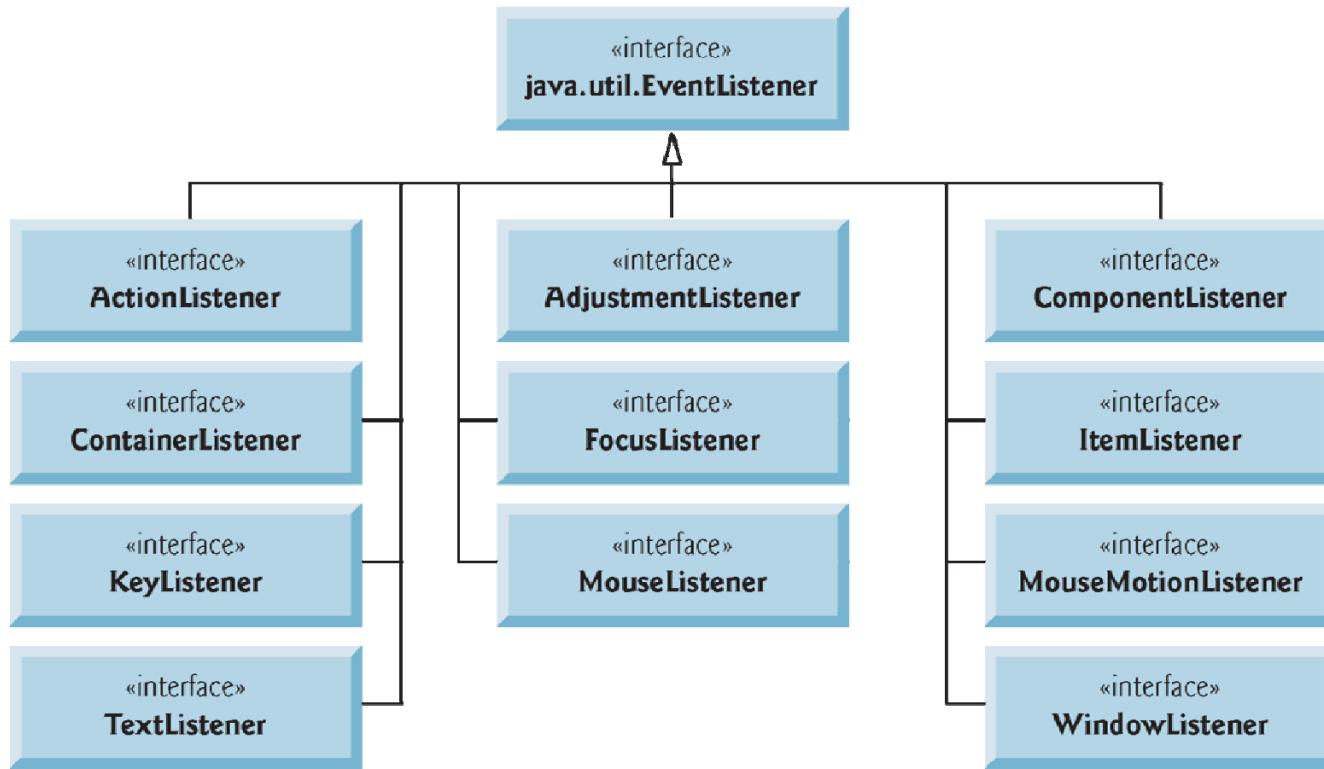


**Fig. 12.11** | Some event classes of package `java.awt.event`.



## 12.7 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 `j ava. awt. event`, and some of them are shown in Fig. 12.12.
- ▶ Additional event-listener types that are specific to Swing components are declared in package `j avax. swi ng. event`.



**Fig. 12.12** | Some common event-listener interfaces of package `java.awt.event`.



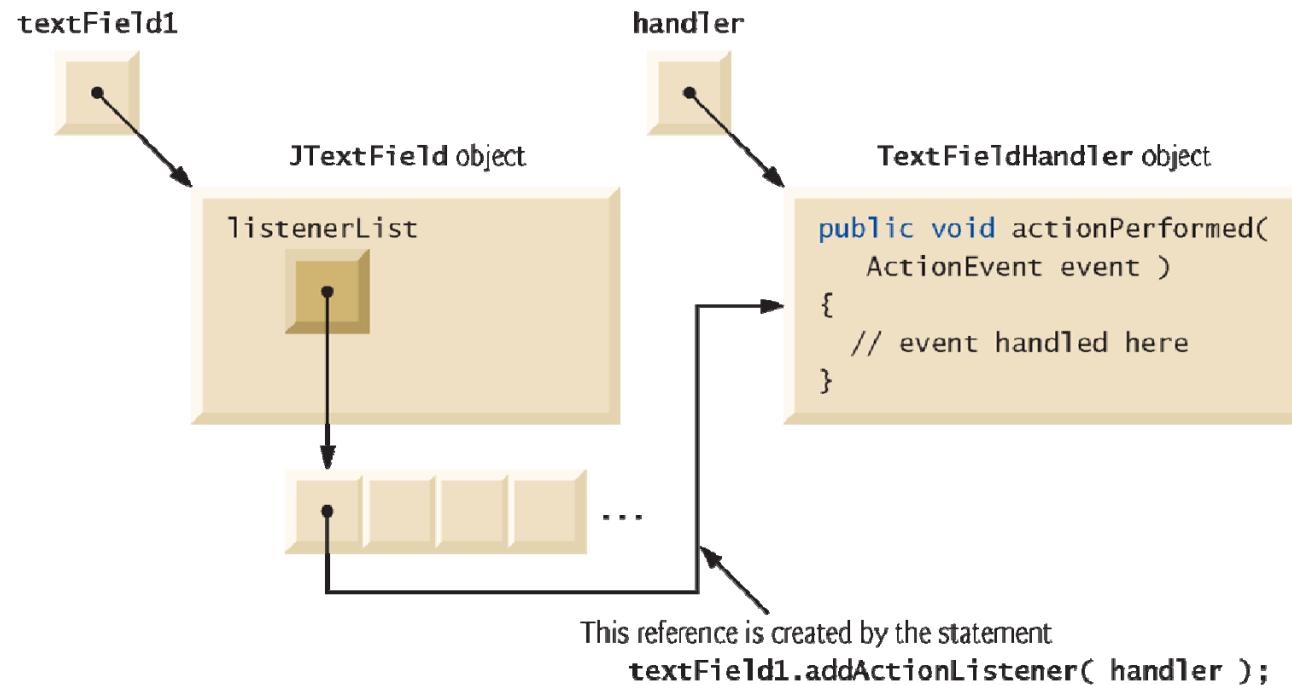
## 12.7 Common GUI Event Types and Listener Interfaces (cont.)

- ▶ 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.



## 12.8 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. 12.13** | Event registration for `JTextField` `textField1`.



## 12.8 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.



## 12.8 How Event Handling Works (cont.)

- ▶ Each *event type* has one or more corresponding event-listener interfaces.
  - **ActionEvents** are handled by **ActionListeners**
  - **MouseEvents** are handled by **MouseListeners** and **MouseMotionListeners**
  - **KeyEvents** are handled by **KeyListeners**
- ▶ When an event occurs, the GUI component receives (from the JVM) a unique **event ID** specifying the event type.
  - The 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.



## 12.8 How Event Handling Works (cont.)

- ▶ For an `ActionEvent`, the event is dispatched to every registered `ActionListener`'s `actionPerformed` method.
- ▶ For a `Mouse-Event`, 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.



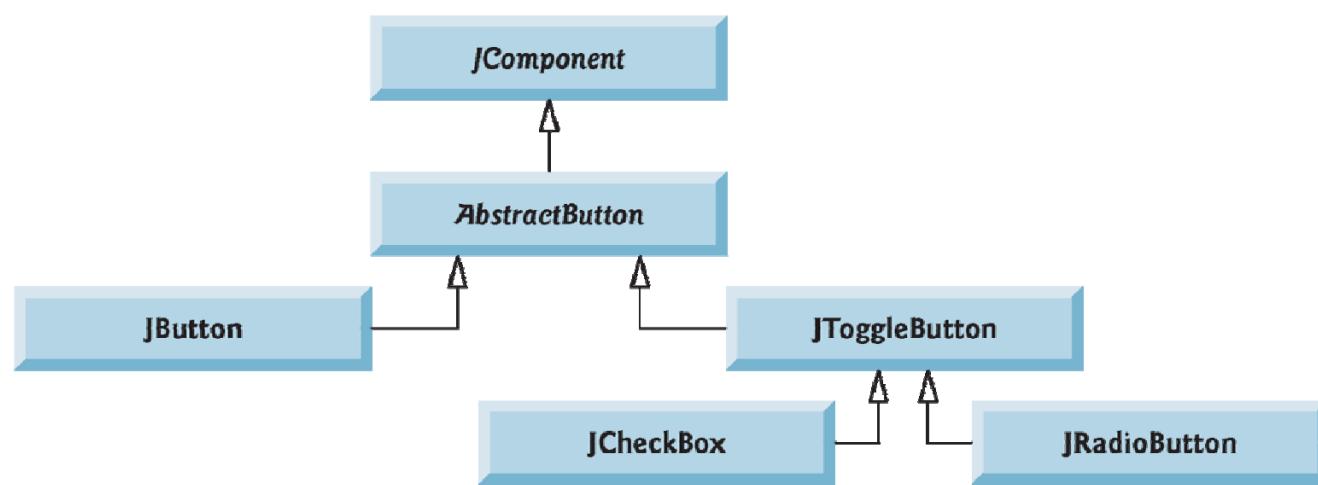
## Performance Tip 12.1

*GUIs should always remain responsive to the user. Performing a long-running task in an event handler prevents the user from interacting with the GUI until that task completes. Section 23.11 demonstrates techniques prevent such problems.*



## 12.9 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 `j avax. swi ng`), which declares the common features of Swing buttons.



**Fig. 12.14** | Swing button hierarchy.



## 12.9 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`.



## Look-and-Feel Observation 12.8

*The text on buttons typically uses book-title capitalization.*



## Look-and-Feel Observation 12.9

*A GUI can have many JButtons, but each button label should be unique in the portion of the GUI that's currently displayed. Having more than one JButton with the same label makes the JButtons ambiguous to the user.*



---

```
1 // Fig. 12.15: ButtonFrame.java
2 // Command buttons and action events.
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 final JButton plainJButton; // button with just text
15     private final JButton fancyJButton; // button with icons
16
17     // ButtonFrame adds JButtons to JFrame
18     public ButtonFrame()
19     {
20         super("Testing Buttons");
21         setLayout(new FlowLayout());
22
23         plainJButton = new JButton("Plain Button"); // button with text
24         add(plainJButton); // add plainJButton to JFrame
```

---

**Fig. 12.15** | 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 }
37
38 // inner class for button event handling
39 private class ButtonHandler implements ActionListener
40 {
41     // handle button event
42     @Override
43     public void actionPerformed(ActionEvent event)
44     {
45         JOptionPane.showMessageDialog(ButtonFrame.this, String.format(
46             "You pressed: %s", event.getActionCommand()));
47     }
48 }
49 } // end class ButtonFrame
```

**Fig. 12.15** | Command buttons and action events. (Part 2 of 2.)

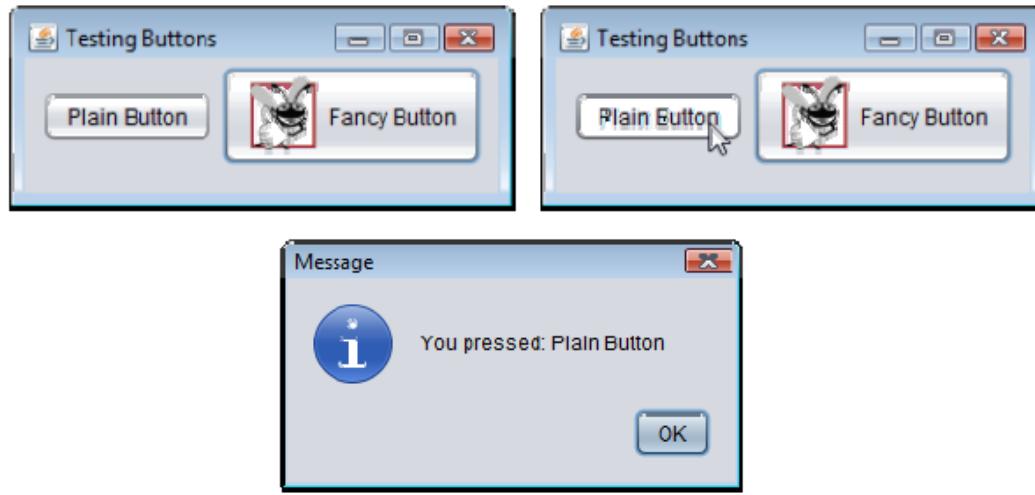


---

```
1 // Fig. 12.16: 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();
10        buttonFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        buttonFrame.setSize(275, 110);
12        buttonFrame.setVisible(true);
13    }
14 } // end class ButtonTest
```

---

**Fig. 12.16** | Testing ButtonFrame. (Part 1 of 3.)



**Fig. 12.16** | Testing ButtonFrame. (Part 2 of 3.)



---

**Fig. 12.16** | Testing ButtonFrame. (Part 3 of 3.)



## 12.9 JButton (cont.)

- ▶ A JButton can display an Icon.
- ▶ A JButton can also have a **rollover Icon**
  - displayed when the user positions the mouse over the JButton.
  - The icon on the JButton changes as the mouse moves in and out of the JButton's area on the screen.
- ▶ AbstractButton method **setRolloverIcon** specifies the image displayed on the JButton when the user positions the mouse over it.



## Look-and-Feel Observation 12.10

*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 12.11

*Rollover icons provide visual feedback indicating that an action will occur when a JButton is clicked.*



## 12.9 JButton (cont.)

- ▶ JButtons, like JTextFields, generate ActionEvents that can be processed by any ActionListener object.



## Software Engineering Observation 12.2

*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 (.) separator, as in `ButtonFrame.this`.*



## 12.10 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



## 12.10.1 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. 12.17: CheckBoxFrame.java
2 // JCheckboxes and item events.
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 final JTextField textField; // displays text in changing fonts
14     private final JCheckBox boldJCheckBox; // to select/deselect bold
15     private final 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());
```

---

**Fig. 12.17** | JCheckboxes and item events. (Part 1 of 3.)



```
23 // set up JTextField and set its font
24 textField = new JTextField("Watch the font style change", 20);
25 textField.setFont(new Font("Serif", Font.PLAIN, 14));
26 add(textField); // add textField to JFrame
27
28 boldJCheckBox = new JCheckBox("Bold");
29 italicJCheckBox = new JCheckBox("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 }
38 }
```

**Fig. 12.17** | JCheckboxes and item events. (Part 2 of 3.)

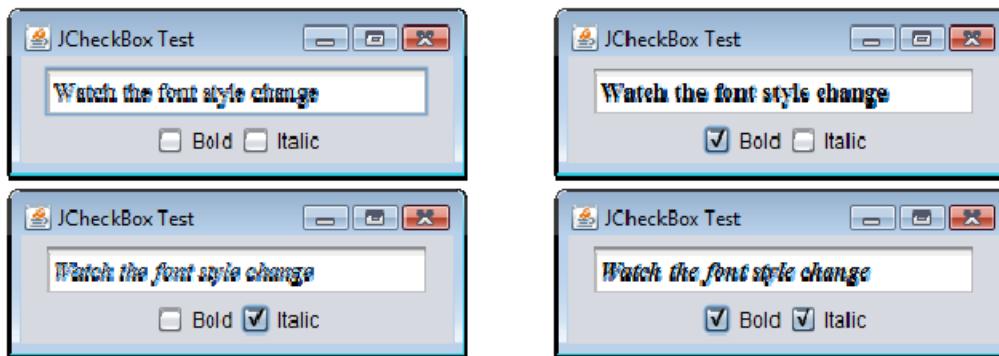


```
39 // private inner class for ItemListener event handling
40 private class CheckBoxHandler implements ItemListener
41 {
42     // respond to checkbox events
43     @Override
44     public void itemStateChanged(ItemEvent event)
45     {
46         Font font = null; // stores the new Font
47
48         // determine which CheckBoxes are checked and create Font
49         if (boldJCheckBox.isSelected() && italicJCheckBox.isSelected())
50             font = new Font("Serif", Font.BOLD + Font.ITALIC, 14);
51         else if (boldJCheckBox.isSelected())
52             font = new Font("Serif", Font.BOLD, 14);
53         else if (italicJCheckBox.isSelected())
54             font = new Font("Serif", Font.ITALIC, 14);
55         else
56             font = new Font("Serif", Font.PLAIN, 14);
57
58         textField.setFont(font);
59     }
60 }
61 } // end class CheckBoxFrame
```

**Fig. 12.17** | JCheck Boxes and item events. (Part 3 of 3.)



```
1 // Fig. 12.18: 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);
12        checkBoxFrame.setVisible(true);
13    }
14 } // end class CheckBoxTest
```



**Fig. 12.18 |** Testing CheckBoxFrame.



## 12.10.2 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.
- ▶ 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. 12.19: 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 final JTextField textField; // used to display font changes
15     private final Font plainFont; // font for plain text
16     private final Font boldFont; // font for bold text
17     private final Font italicFont; // font for italic text
18     private final Font boldItalicFont; // font for bold and italic text
19     private final JRadioButton plainJRadioButton; // selects plain text
20     private final JRadioButton boldJRadioButton; // selects bold text
21     private final JRadioButton italicJRadioButton; // selects italic text
22     private final JRadioButton boldItalicJRadioButton; // bold and italic
23     private final ButtonGroup radioGroup; // holds radio buttons
```

**Fig. 12.19** | Creating radio buttons using `ButtonGroup` and `JRadioButton`. (Part 1 of 4.)



---

```
24
25 // RadioButtonFrame constructor adds JRadioButtons to JFrame
26 public RadioButtonFrame()
27 {
28     super("RadioButton Test");
29     setLayout(new FlowLayout());
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
```

---

**Fig. 12.19** | Creating radio buttons using `ButtonGroup` and `JRadioButton`. (Part 2 of 4.)



```
44 // create logical relationship between JRadioButtons
45 radioGroup = new ButtonGroup(); // create ButtonGroup
46 radioGroup.add(plainJRadioButton); // add plain to group
47 radioGroup.add(boldJRadioButton); // add bold to group
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);
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 }
```

**Fig. 12.19** | Creating radio buttons using `ButtonGroup` and `JRadioButton`. (Part 3 of 4.)



---

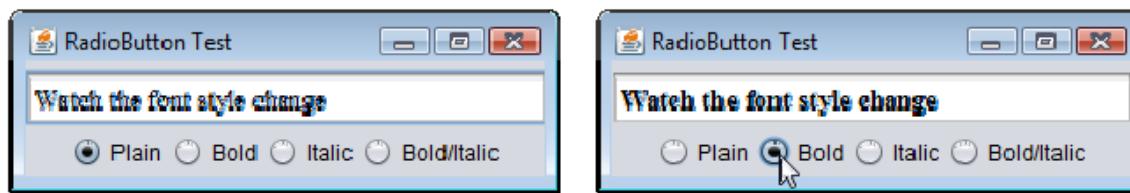
```
68
69     // private inner class to handle radio button events
70     private class RadioButtonHandler implements ItemListener
71     {
72         private Font font; // font associated with this listener
73
74         public RadioButtonHandler(Font f)
75         {
76             font = f;
77         }
78
79         // handle radio button events
80         @Override
81         public void itemStateChanged(ItemEvent event)
82         {
83             textField.setFont(font);
84         }
85     }
86 } // end class RadioButtonFrame
```

---

**Fig. 12.19** | Creating radio buttons using `ButtonGroup` and `JRadioButton`. (Part 4 of 4.)



```
1 // Fig. 12.20: 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);
12        radioButtonFrame.setVisible(true);
13    }
14 } // end class RadioButtonTest
```



**Fig. 12.20 |** Testing RadioButtonFrame. (Part 1 of 2.)



**Fig. 12.20 |** Testing RadioButtonFrame. (Part 2 of 2.)



## 12.10.2 JRadioButton (cont.)

- ▶ ButtonGroup method `add` associates a JRadioButton with the group.
- ▶ If more than one selected JRadioButton object is added to the group, the selected one that was added first will be selected when the GUI is displayed.
- ▶ JRadioButtons, like JCheckboxes, generate ItemEvents when they are clicked.



## 12.11 JComboBox; Using an Anonymous Inner Class for Event Handling

- ▶ A combo box (sometimes called a [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. 12.21: 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 final JComboBox<String> imagesJComboBox; // holds icon names
15     private final JLabel label; // displays selected icon
16
17     private static final String[] names =
18         {"bug1.gif", "bug2.gif", "travelbug.gif", "buganim.gif"};
19     private final 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. 12.21** | JComboBox that displays a list of image names. (Part 1 of 3.)



```
25 // ComboBoxFrame constructor adds JComboBox to JFrame
26 public ComboBoxFrame()
27 {
28     super("Testing JComboBox");
29     setLayout(new FlowLayout()); // set frame layout
30
31     imagesJComboBox = new JComboBox<String>(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         @Override
39         public void itemStateChanged(ItemEvent event)
40         {
41             // determine whether item selected
42             if (event.getStateChange() == ItemEvent.SELECTED)
43                 label.setIcon(Icons[
44                     imagesJComboBox.getSelectedIndex()]);
45         }
46     } // end anonymous inner class
47 ); // end call to addItemListener
48
```

**Fig. 12.21** | JComboBox that displays a list of image names. (Part 2 of 3.)



---

```
49     add(imagesJComboBox); // add combo box to JFrame
50     label = new JLabel(Icons[0]); // display first icon
51     add(label); // add label to JFrame
52 }
53 } // end class ComboBoxFrame
```

---

**Fig. 12.21** | JComboBox that displays a list of image names. (Part 3 of 3.)

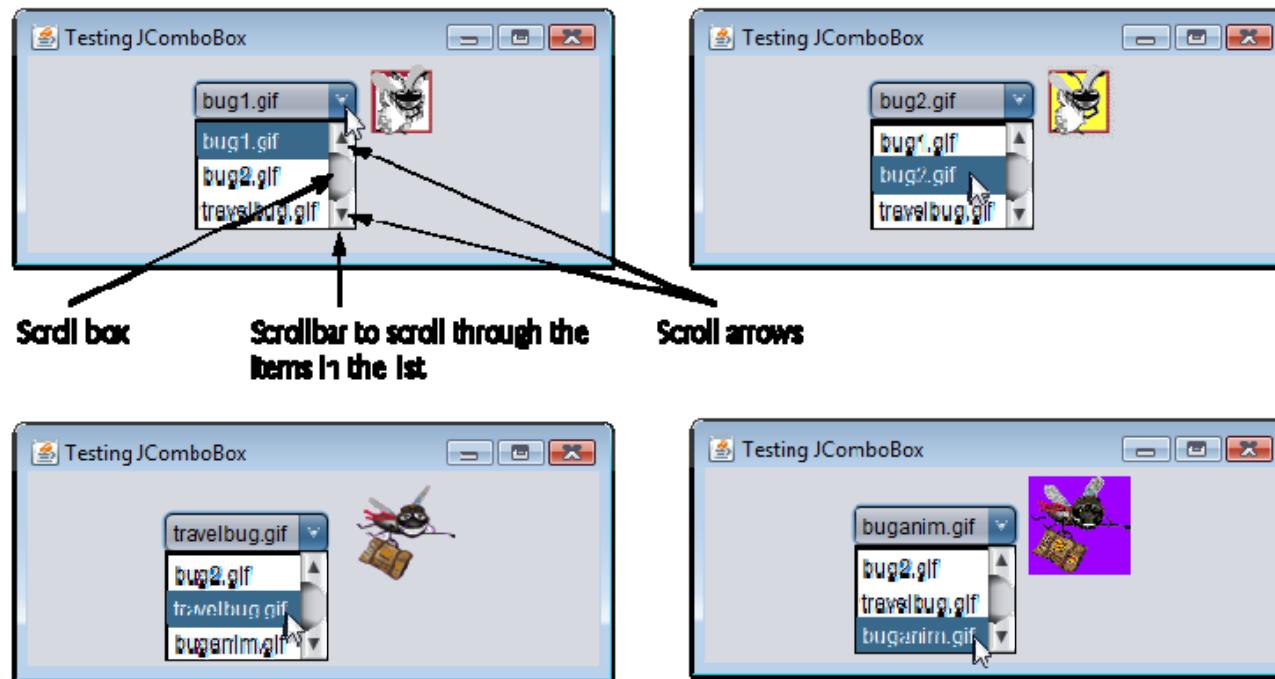


---

```
1 // Fig. 12.22: 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);
12        comboBoxFrame.setVisible(true);
13    }
14 } // end class ComboBoxTest
```

---

**Fig. 12.22** | Testing ComboBoxFrame. (Part 1 of 2.)



**Fig. 12.22 | Testing ComboBoxFrame (Part 2 of 2.)**



## 12.11 JComboBox; Using an Anonymous Inner Class for Event Handling (cont.)

- ▶ The first item added to a JComboBox appears as the currently selected item when the JComboBox is displayed.
- ▶ Other items are selected by clicking the JComboBox, then selecting an item from the list that appears.
- ▶ JComboBox method `setMaximumRowCount` sets the maximum number of elements that are displayed when the user clicks the JComboBox.
- ▶ If there are additional items, the JComboBox provides a `scrollbar` that allows the user to scroll through all the elements in the list.



## Look-and-Feel Observation 12.12

*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's used.*



## 12.11 JComboBox; 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.*
- ▶ An anonymous inner class has limited access to the local variables of the method in which it's declared.
- ▶ 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.



## Software Engineering Observation 12.3

*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 local variables. As of Java SE 8, anonymous inner classes may also access a methods "effectively final" local variables—see Chapter 17 for more information.*



## 12.11 JComboBox; Using an Anonymous Inner Class for Event Handling (cont.)

- ▶ JComboBox method `getSelectedIndex` returns the index of the selected item.
- ▶ For each item selected from a JComboBox, another item is first deselected—so two ItemEvents occur when an item is selected.
- ▶ ItemEvent method `getStateChange` returns the type of state change. ItemEvent.SELECTED indicates that an item was selected.
- ▶ In Section 17.9, we show how to use Java SE 8 lambdas to create event handlers.
  - The compiler translates a lambda into an object of an anonymous inner class.



## Software Engineering Observation 12.4

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



## 12.12 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`.
- ▶ Class `JList`—which like `JComboBox` is a generic class—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. 12.23: 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 final JList<String> 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. 12.23** | JList that displays a list of colors. (Part I of 3.)



```
23 // ListFrame constructor add JScrollPane containing JList to JFrame
24 public ListFrame()
25 {
26     super("List Test");
27     setLayout(new FlowLayout());
28
29     colorJList = new JList<String>(colorNames); // list of 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
```

**Fig. 12.23** | JList that displays a list of colors. (Part 2 of 3.)



---

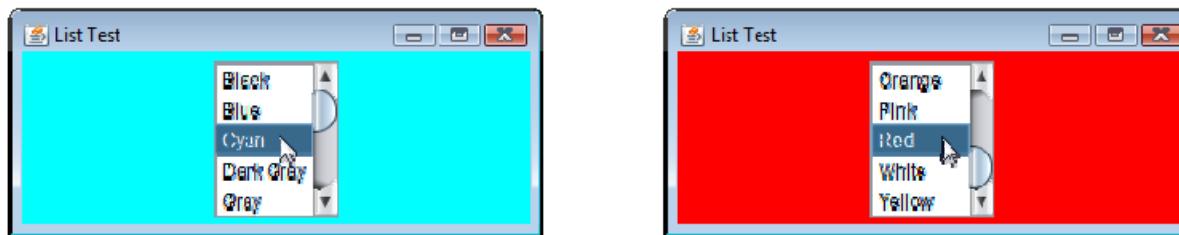
```
38     colorJList.addListSelectionListener(
39         new ListSelectionListener() // anonymous inner class
40     {
41         // handle list selection events
42         @Override
43         public void valueChanged(ListSelectionEvent event)
44         {
45             getContentPane().setBackground(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

---

**Fig. 12.23** | JList that displays a list of colors. (Part 3 of 3.)



```
1 // Fig. 12.24: 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);
12        listFrame.setVisible(true);
13    }
14 } // end class ListTest
```



**Fig. 12.24** | Selecting colors from a JList



## 12.12 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).



## 12.12 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 aJList's selection events.



## 12.12 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 is 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.



## 12.13 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 **MULTI\_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. 12.25: MultipleSelectionFrame.java
2 // JList that allows multiple selections.
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 final JList<String> colorJList; // List to hold color names
15     private final JList<String> copyJList; // List to hold copied names
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     {
24         super("Multiple Selection Lists");
25         setLayout(new FlowLayout());
```

**Fig. 12.25** | JList that allows multiple selections. (Part 1 of 3.)



---

```
26
27     colorJList = new JList<String>(colorNames); // list of color names
28     colorJList.setVisibleRowCount(5); // show five rows
29     colorJList.setSelectionMode(
30         ListSelectionMode.MULTIPLE_INTERVAL_SELECTION);
31     add(new JScrollPane(colorJList)); // add list with scrollpane
32
33     copyJButton = new JButton("Copy >>>");
34     copyJButton.addActionListener(
35         new ActionListener() // anonymous inner class
36         {
37             // handle button event
38             @Override
39             public void actionPerformed(ActionEvent event)
40             {
41                 // place selected values in copyJList
42                 copyJList.setListData(
43                     colorJList.getSelectedValuesList().toArray(
44                         new String[0]));
45             }
46         }
47     );
48 
```

---

**Fig. 12.25** | `JList` that allows multiple selections. (Part 2 of 3.)



---

```
49     add(copyJButton); // add copy button to JFrame
50
51     copyJList = new JList<String>(); // list to hold copied color names
52     copyJList.setVisibleRowCount(5); // show 5 rows
53     copyJList.setFixedCellWidth(100); // set width
54     copyJList.setFixedCellHeight(15); // set height
55     copyJList.setSelectionMode(
56         ListSelectionMode.SINGLE_INTERVAL_SELECTION);
57     add(new JScrollPane(copyJList)); // add List with scrollpane
58 }
59 } // end class MultipleSelectionFrame
```

---

**Fig. 12.25** | `JList` that allows multiple selections. (Part 3 of 3.)

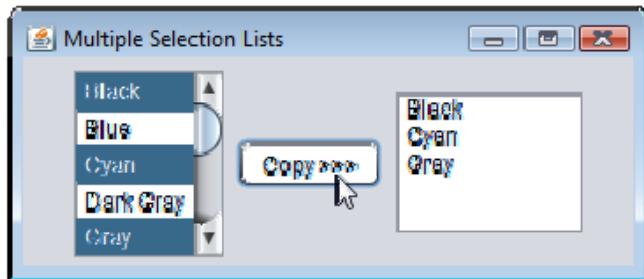


---

```
1 // Fig. 12.26: 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);
14         multipleSelectionFrame.setVisible(true);
15     }
16 } // end class MultipleSelectionTest
```

---

**Fig. 12.26** | Testing MultipleSelectionFrame. (Part 1 of 2.)



**Fig. 12.26** | Testing `MultipleSelectionFrame`. (Part 2 of 2.)



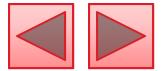
## 12.13 Multiple-Selection Lists (cont.)

- ▶ 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`.



## 12.14 Mouse Event Handling

- ▶ `MouseListener` and `MouseMotionListener` event-listener interfaces for handling `mouse` events.
  - Any GUI component
- ▶ Package `javax.swing.event` contains interface `MouseInputListener`, 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. Always preceded by a call to `mousePressed` and `mouseReleased`.

**public void mouseReleased(MouseEvent event)**

Called when a mouse button is *released after being pressed*. Always preceded by a call to `mousePressed` and one or more calls to `mouseDragged`.

**public void mouseEntered(MouseEvent event)**

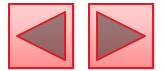
Called when the mouse cursor *enters* the bounds of a component.

**public void mouseExited(MouseEvent event)**

Called when the mouse cursor *leaves* the bounds of a component.

**Fig. 12.27** | MouseListener and MouseMotionListener interface methods.

(Part 1 of 2.)



## MouseListener and MouseMotionListener interface methods

### *Methods of interface MouseMotionListener*

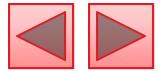
```
public void mouseDragged(MouseEvent event)
```

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*. 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. 12.27** | MouseListener and MouseMotionListener interface methods.  
(Part 2 of 2.)



## 12.14 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` (`Mouse-Event`'s superclass) enable you to determine which mouse button the user clicked.



## Software Engineering Observation 12.5

*Calls to mouseDragged are sent to the MouseMotionListener for the Component on which the drag started. Similarly, the mouseReleased call at the end of a drag operation is sent to the MouseListener for the Component on which the drag operation started.*



## 12.14 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. 12.28: MouseTrackerFrame.java
2 // Mouse event handling.
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 final JPanel mousePanel; // panel in which mouse events occur
15     private final JLabel statusBar; // displays event information
16 }
```

---

**Fig. 12.28** | Mouse event handling. (Part 1 of 5.)



```
17 // MouseTrackerFrame constructor sets up GUI and
18 // registers mouse event handlers
19 public MouseTrackerFrame()
20 {
21     super("Demonstrating Mouse Events");
22
23     mousePanel = new JPanel();
24     mousePanel.setBackground(Color.WHITE);
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 }
35
```

**Fig. 12.28** | Mouse event handling. (Part 2 of 5.)



```
36     private class MouseHandler implements MouseListener,  
37         MouseMotionListener  
38     {  
39         // MouseListener event handlers  
40         // handle event when mouse released immediately after press  
41         @Override  
42         public void mouseClicked(MouseEvent event)  
43         {  
44             statusBar.setText(String.format("Clicked at [%d, %d]",  
45                 event.getX(), event.getY()));  
46         }  
47  
48         // handle event when mouse pressed  
49         @Override  
50         public void mousePressed(MouseEvent event)  
51         {  
52             statusBar.setText(String.format("Pressed at [%d, %d]",  
53                 event.getX(), event.getY()));  
54         }  
55     }
```

**Fig. 12.28** | Mouse event handling. (Part 3 of 5.)



```
56 // handle event when mouse released
57 @Override
58 public void mouseReleased(MouseEvent event)
59 {
60     statusBar.setText(String.format("Released at [%d, %d]",
61         event.getX(), event.getY()));
62 }
63
64 // handle event when mouse enters area
65 @Override
66 public void mouseEntered(MouseEvent event)
67 {
68     statusBar.setText(String.format("Mouse entered at [%d, %d]",
69         event.getX(), event.getY()));
70     mousePanel.setBackground(Color.GREEN);
71 }
72
73 // handle event when mouse exits area
74 @Override
75 public void mouseExited(MouseEvent event)
76 {
77     statusBar.setText("Mouse outside JPanel1");
78     mousePanel.setBackground(Color.WHITE);
79 }
```

**Fig. 12.28** | Mouse event handling. (Part 4 of 5.)



---

```
80
81     // MouseMotionListener event handlers
82     // handle event when user drags mouse with button pressed
83     @Override
84     public void mouseDragged(MouseEvent event)
85     {
86         statusBar.setText(String.format("Dragged at [%d, %d]",
87             event.getX(), event.getY()));
88     }
89
90     // handle event when user moves mouse
91     @Override
92     public void mouseMoved(MouseEvent event)
93     {
94         statusBar.setText(String.format("Moved at [%d, %d]",
95             event.getX(), event.getY()));
96     }
97 } // end inner class MouseHandler
98 } // end class MouseTrackerFrame
```

---

**Fig. 12.28** | Mouse event handling. (Part 5 of 5.)

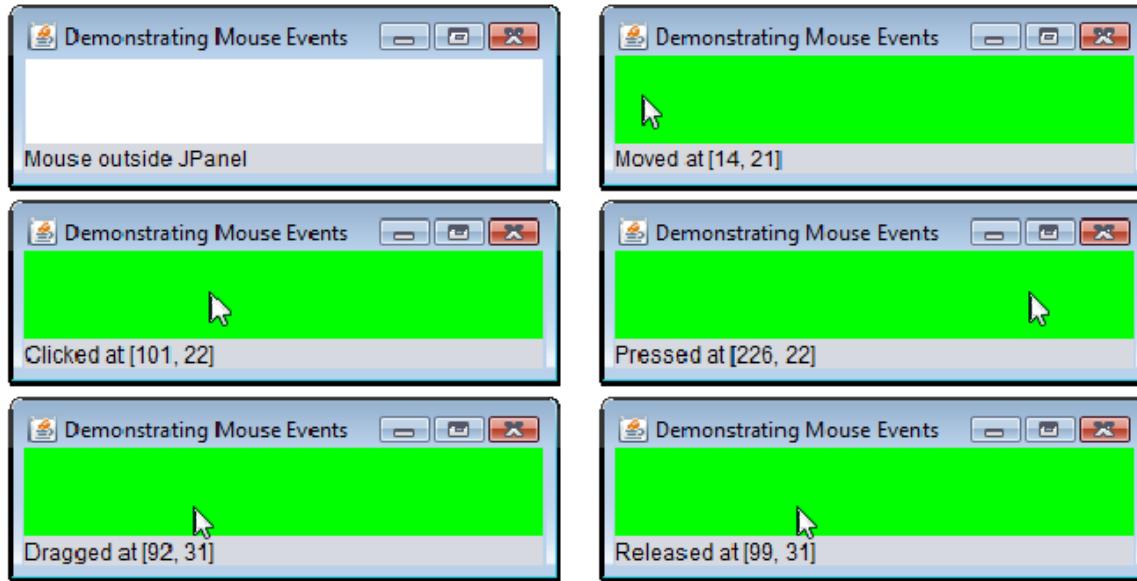
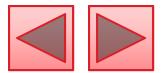


---

```
1 // Fig. 12.29: 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);
12        mouseTrackerFrame.setVisible(true);
13    }
14 } // end class MouseTracker
```

---

**Fig. 12.29** | Testing MouseTrackerFrame. (Part I of 2.)

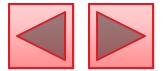


**Fig. 12.29 | Testing MouseTrackerFrame. (Part 2 of 2.)**



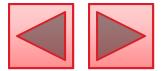
## 12.14 Mouse Event Handling (cont.)

- ▶ `BorderLayout` arranges component NORTH, SOUTH, EAST, WEST and CENTER regions.
- ▶ `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.



## 12.15 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 12.6

*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 `java.awt.event`      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. 12.30** | Event-adapter classes and the interfaces they implement.



```
1 // Fig. 12.31: 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 displayed in the statusBar
12     private final JLabel statusBar; // JLabel 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 MouseClickHandler()); // add handler
22     }
23 }
```

---

**Fig. 12.31** | Demonstrating mouse clicks and distinguishing between mouse buttons. (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     @Override
29     public void mouseClicked(MouseEvent event)
30     {
31         int xPos = event.getX(); // get x-position of mouse
32         int yPos = event.getY(); // get y-position of mouse
33
34         details = String.format("Clicked %d time(s)",
35             event.getClickCount());
36
37         if (event.isMetaDown()) // right mouse button
38             details += " with right mouse button";
39         else if (event.isAltDown()) // middle mouse button
40             details += " with center mouse button";
41         else // left mouse button
42             details += " with left mouse button";
43
44         statusBar.setText(details); // display message in statusBar
45     }
46 }
47 } // end class MouseDetailsFrame
```

**Fig. 12.31** | Demonstrating mouse clicks and distinguishing between mouse buttons. (Part 2 of 2.)

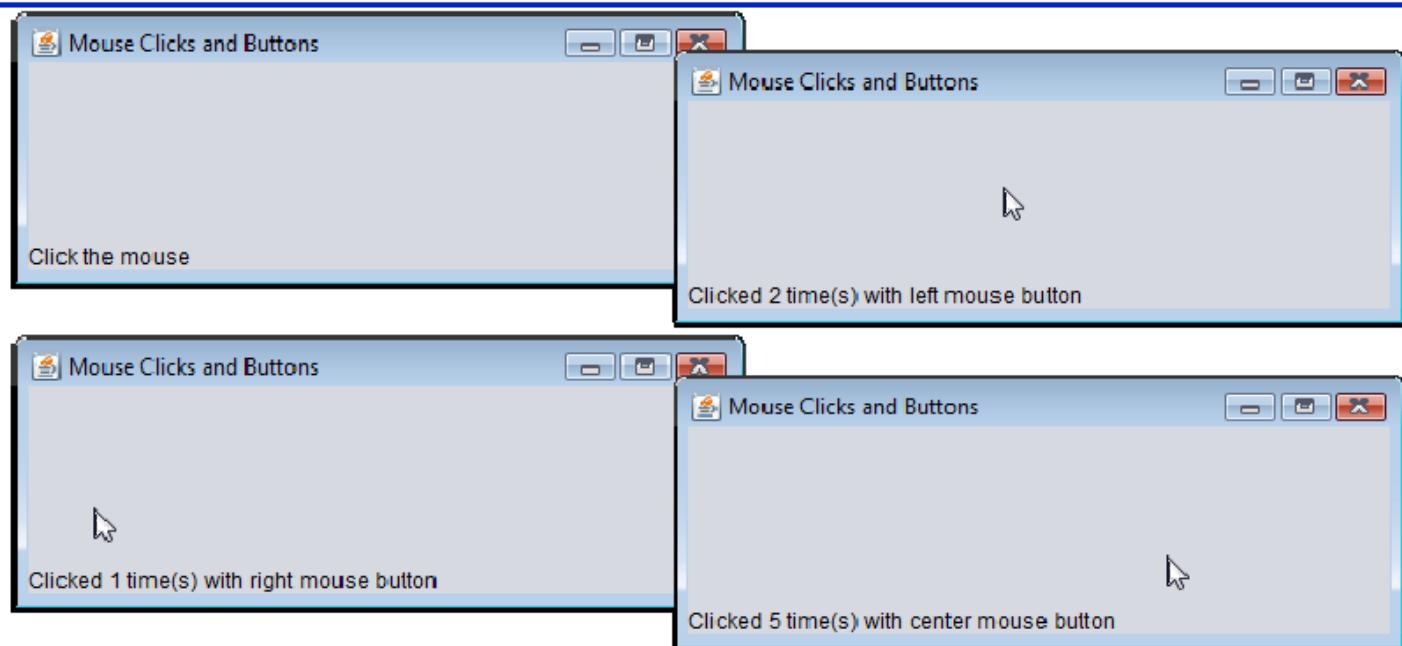


---

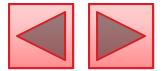
```
1 // Fig. 12.32: MouseDetails.java
2 // Testing MouseDetailsFrame.
3 import javax.swing.JFrame;
4
5 public class MouseDetails
6 {
7     public static void main(String[] args)
8     {
9         MouseDetailsFrame mouseDetailsFrame = new MouseDetailsFrame();
10        mouseDetailsFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        mouseDetailsFrame.setSize(400, 150);
12        mouseDetailsFrame.setVisible(true);
13    }
14 } // end class MouseDetails
```

---

**Fig. 12.32** | Testing `MouseDetailsFrame`. (Part 1 of 2.)



**Fig. 12.32 | Testing MouseDetailsFrame. (Part 2 of 2.)**



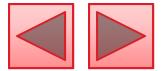
## Common Programming Error 12.3

*If you extend an adapter class and misspell the name of the method you're overriding, and you do not declare the method with @Override, 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.*



## 12.15 Adapter Classes (cont.)

- ▶ A mouse can have one, two or three buttons.
- ▶ Class `MouseEvent` inherits several methods from `InputEvent` that can distinguish among mouse buttons or mimic a multibutton mouse with a combined keystroke and mouse-button click.
- ▶ Java assumes that every mouse contains a left mouse button.



## 12.15 Adapter Classes (cont.)

- ▶ In the case of a one- or two-button mouse, a Java application assumes that the center mouse button is clicked if the user holds down the *Alt* key and clicks the left mouse button on a two-button mouse or the only mouse button on a one-button mouse.
- ▶ In the case of a one-button mouse, a Java application assumes that the right mouse button is clicked if the user holds down the *Meta* key (sometimes called the Command key or the “Apple” key on a Mac) and clicks the mouse button.



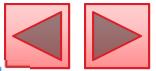
InputEvent method	Description
<code>isMetaDown()</code>	Returns <code>true</code> when the user clicks the <i>right mouse button</i> on a mouse with two or three buttons. To simulate a right-mouse-button click on a one-button mouse, the user can hold down the <i>Meta</i> key on the keyboard and click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

**Fig. 12.33** | `InputEvent` methods that help determine whether the right or center mouse button was clicked.



## 12.15 Adapter Classes (cont.)

- ▶ The number of consecutive mouse clicks is returned by `MouseEvent` method `getClickCount`.
- ▶ Methods `isMetaDown` and `isAltDown` determine which mouse button the user clicked.



## 12.16 JPanel Subclass for Drawing with the Mouse

- ▶ Use a JPanel as a **dedicated drawing area** in which the user can draw by dragging the mouse.
- ▶ Lightweight Swing components that extend class JComponent (such as JPanel) contain method **paintComponent**
  - called when a lightweight Swing component is displayed
- ▶ Override this method to specify how to draw.
  - Call the superclass version of paintComponent as the first statement in the body of the overridden method to ensure that the component displays correctly.



## 12.16 JPanel Subclass for Drawing with the Mouse (cont.)

- ▶ JComponent support transparency.
  - To display a component correctly, the program must determine whether the component is transparent.
  - The code that determines this is in superclass JComponent's `paintComponent` implementation.
  - When a component is transparent, `paintComponent` will not clear its background
  - When a component is opaque, `paintComponent` clears the component's background
  - The transparency of a Swing lightweight component can be set with method `setOpaque` (a false argument indicates that the component is transparent).



## Error-Prevention Tip 12.1

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



## Common Programming Error 12.4

*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.*



---

```
1 // Fig. 12.34: PaintPanel.java
2 // Adapter class used to implement event handlers.
3 import java.awt.Point;
4 import java.awt.Graphics;
5 import java.awt.event.MouseEvent;
6 import java.awt.event.MouseMotionAdapter;
7 import java.util.ArrayList;
8 import javax.swing.JPanel;
9
10 public class PaintPanel extends JPanel
11 {
12     // list of Point references
13     private final ArrayList<Point> points = new ArrayList<>();
14 }
```

---

**Fig. 12.34** | Adapter class used to implement event handlers. (Part 1 of 3.)



```
15 // set up GUI and register mouse event handler
16 public PaintPanel()
17 {
18     // handle frame mouse motion event
19     addMouseMotionListener(
20         new MouseMotionAdapter() // anonymous inner class
21         {
22             // store drag coordinates and repaint
23             @Override
24             public void mouseDragged(MouseEvent event)
25             {
26                 points.add(event.getPoint());
27                 repaint(); // repaint JFrame
28             }
29         }
30     );
31 }
32
33 // draw ovals in a 4-by-4 bounding box at specified locations on window
34 @Override
35 public void paintComponent(Graphics g)
36 {
37     super.paintComponent(g); // clears drawing area
38 }
```

**Fig. 12.34** | Adapter class used to implement event handlers. (Part 2 of 3.)



---

```
39     // draw all points
40     for (Point point : points)
41         g.fillOval(point.x, point.y, 4, 4);
42     }
43 } // end class PaintPanel
```

**Fig. 12.34** | Adapter class used to implement event handlers. (Part 3 of 3.)



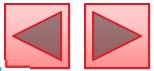
## 12.16 JPanel Subclass for Drawing with the Mouse (cont.)

- ▶ Class `Point` (package `java.awt`) represents an *x-y* coordinate.
  - We use objects of this class to store the coordinates of each mouse drag event.
- ▶ Class `Graphics` is used to draw.
- ▶ `MouseEvent` method `getPoint` obtains the `Point` where the event occurred.
- ▶ Method `repaint` (inherited from `Component`) indicates that a `Component` should be refreshed on the screen as soon as possible.



## Look-and-Feel Observation 12.13

*Calling `repaint` for a Swing GUI component indicates that the component should be refreshed on the screen as soon as possible. The component's background 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`).*



## 12.16 JPanel Subclass for Drawing with the Mouse (cont.)

- ▶ Graphics method `fillOval` draws a solid oval.
  - Four parameters represent a rectangular area (called the bounding box) in which the oval is displayed.
  - The first two are the upper-left x-coordinate and the upper-left y-coordinate of the rectangular area.
  - The last two represent the rectangular area's width and height.
- ▶ Method `fillOval` draws the oval so it touches the middle of each side of the rectangular area.



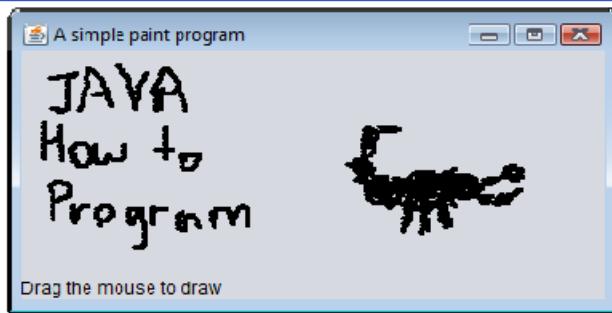
## Look-and-Feel Observation 12.14

*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.*

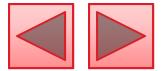


```
1 // Fig. 12.35: Painter.java
2 // Testing PaintPanel.
3 import java.awt.BorderLayout;
4 import javax.swing.JFrame;
5 import javax.swing.JLabel;
6
7 public class Painter
8 {
9     public static void main(String[] args)
10    {
11        // create JFrame
12        JFrame application = new JFrame("A simple paint program");
13
14        PaintPanel paintPanel = new PaintPanel();
15        application.add(paintPanel, BorderLayout.CENTER);
16
17        // create a label and place it in SOUTH of BorderLayout
18        application.add(new JLabel("Drag the mouse to draw"),
19                      BorderLayout.SOUTH);
20
21        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22        application.setSize(400, 200);
23        application.setVisible(true);
24    }
25 } // end class Painter
```

**Fig. 12.35** | Testing PaintPanel. (Part 1 of 2.)



**Fig. 12.35 |** Testing PaintPanel1. (Part 2 of 2.)



## 12.17 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**.
- ▶ Method **keyReleased** is called when the key is released after any **keyPressed** or **keyTyped** event.



---

```
1 // Fig. 12.36: KeyDemoFrame.java
2 // Key event handling.
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 final String line1 = ""; // first line of text area
12     private final String line2 = ""; // second line of text area
13     private final String line3 = ""; // third line of text area
14     private final JTextArea textArea; // text area to display output
15 }
```

---

**Fig. 12.36** | Key event handling. (Part 1 of 4.)



---

```
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);
24     textArea.setDisabledTextColor(Color.BLACK);
25     add(textArea); // add text area to JFrame
26
27     addKeyListener(this); // allow frame to process key events
28 }
29
30 // handle press of any key
31 @Override
32 public void keyPressed(KeyEvent event)
33 {
34     line1 = String.format("Key pressed: %s",
35         KeyEvent.getKeyText(event.getKeyCode())); // show pressed key
36     setLines2and3(event); // set output lines two and three
37 }
38
```

---

**Fig. 12.36** | Key event handling. (Part 2 of 4.)



---

```
39 // handle release of any key
40 @Override
41 public void keyReleased(KeyEvent event)
42 {
43     line1 = String.format("Key released: %s",
44         KeyEvent.getKeyText(event.getKeyCode())); // show released key
45     setLines2and3(event); // set output lines two and three
46 }
47
48 // handle press of an action key
49 @Override
50 public void keyTyped(KeyEvent event)
51 {
52     line1 = String.format("Key typed: %s", event.getKeyChar());
53     setLines2and3(event); // set output lines two and three
54 }
55
```

---

**Fig. 12.36** | Key event handling. (Part 3 of 4.)



---

```
56     // set second and third lines of output
57     private void setLines2and3(KeyEvent event)
58     {
59         line2 = String.format("This key is %san action key",
60             (event.isActionKey() ? "" : "not "));
61
62         String temp = KeyEvent.getKeyModifiersText(event.getModifiers());
63
64         line3 = String.format("Modifier keys pressed: %s",
65             (temp.equals("") ? "none" : temp)); // output modifiers
66
67         textArea.setText(String.format("%s\n%s\n%s\n",
68             line1, line2, line3)); // output three lines of text
69     }
70 } // end class KeyDemoFrame
```

---

**Fig. 12.36** | Key event handling. (Part 4 of 4.)



---

```
1 // Fig. 12.37: 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);
12        keyDemoFrame.setVisible(true);
13    }
14 } // end class KeyDemo
```

---

**Fig. 12.37** | Testing KeyDemoFrame. (Part 1 of 2.)

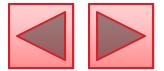


**Fig. 12.37 | Testing KeyDemoFrame. (Part 2 of 2.)**



## 12.17 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.



## 12.17 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.



## 12.18 Introduction to Layout Managers

- ▶ **Layout managers** arrange GUI components in a container for presentation purposes
- ▶ Can use for basic layout capabilities
- ▶ Enable you to concentrate on the basic look-and-feel—the layout manager handles the layout details.
- ▶ Layout managers implement interface `LayoutManager` (in package `j ava. awt`).
- ▶ Container's `setLayout` method takes an object that implements the `LayoutManager` interface as an argument.



## 12.18 Introduction to Layout Managers (cont.)

- ▶ There are three ways for you to arrange components in a GUI:
  - Absolute positioning
    - Greatest level of control.
    - Set Container's layout to null.
    - Specify the absolute position of each GUI component with respect to the upper-left corner of the Container by using Component methods `setSize` and `setLocation` or `setBounds`.
    - Must specify each GUI component's size.



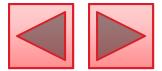
## 12.18 Introduction to Layout Managers (cont.)

- Layout managers
  - Simpler and faster than absolute positioning.
  - Makes your GUIs more resizable.
  - Lose some control over the size and the precise positioning of each component.
- Visual programming in an IDE
  - Use tools that make it easy to create GUIs.
  - Allows you to drag and drop GUI components from a tool box onto a design area.
  - You can then position, size and align GUI components as you like.



## Look-and-Feel Observation 12.15

*Most Java IDEs provide GUI design tools for visually designing a GUI; the tools then write Java code that creates the GUI. Such tools often provide greater control over the size, position and alignment of GUI components than do the built-in layout managers.*



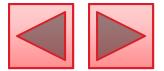
## Look-and-Feel Observation 12.16

*It's 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, you must position and size the components and take care that, on resize events, all components are repositioned as necessary. A component's resize events can be processed by a ComponentListener.*



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

**Fig. 12.38** | Layout managers.



## 12.18.1 FlowLayout

- ▶ FlowLayout is the *simplest* layout manager.
- ▶ GUI components placed from left to right in the order in which they are added to the container.
- ▶ When the edge of the container is reached, components continue to display on the next line.
- ▶ FlowLayout allows GUI components to be *left aligned*, *centered* (the default) and *right aligned*.



## Look-and-Feel Observation 12.17

*Each individual container can have only one layout manager, but multiple containers in the same application can each use different layout managers.*



---

```
1 // Fig. 12.39: FlowLayoutFrame.java
2 // FlowLayout allows components to flow over multiple lines.
3 import java.awt.FlowLayout;
4 import java.awt.Container;
5 import java.awt.event.ActionListener;
6 import java.awt.event.ActionEvent;
7 import javax.swing.JFrame;
8 import javax.swing.JButton;
9
10 public class FlowLayoutFrame extends JFrame
11 {
12     private final JButton leftJButton; // button to set alignment left
13     private final JButton centerJButton; // button to set alignment center
14     private final JButton rightJButton; // button to set alignment right
15     private final FlowLayout layout; // layout object
16     private final Container container; // container to set layout
17 }
```

---

**Fig. 12.39** | FlowLayout allows components to flow over multiple lines. (Part 1 of 4.)



```
18 // set up GUI and register button listeners
19 public FlowLayoutFrame()
20 {
21     super("FlowLayout Demo");
22
23     layout = new FlowLayout();
24     container = getContentPane(); // get container to layout
25     setLayout(layout);
26
27     // set up leftJButton and register listener
28     leftJButton = new JButton("Left");
29     add(leftJButton); // add Left button to frame
30     leftJButton.addActionListener(
31         new ActionListener() // anonymous inner class
32     {
33         // process leftJButton event
34         @Override
35         public void actionPerformed(ActionEvent event)
36     {
37             layout.setAlignment(FlowLayout.LEFT);
38     }
}
```

**Fig. 12.39** | FlowLayout allows components to flow over multiple lines. (Part 2 of 4.)



```
39             // realign attached components
40         layout.layoutContainer(container);
41     }
42 }
43 );
44
45 // set up centerJButton and register listener
46 centerJButton = new JButton("Center");
47 add(centerJButton); // add Center button to frame
48 centerJButton.addActionListener(
49     new ActionListener() // anonymous inner class
50     {
51         // process centerJButton event
52         @Override
53         public void actionPerformed(ActionEvent event)
54         {
55             layout.setAlignment(FlowLayout.CENTER);
56
57             // realign attached components
58             layout.layoutContainer(container);
59         }
60     }
61 );
```

**Fig. 12.39** | FlowLayout allows components to flow over multiple lines. (Part 3 of 4.)



---

```
62
63     // set up rightJButton and register listener
64     rightJButton = new JButton("Right");
65     add(rightJButton); // add Right button to frame
66     rightJButton.addActionListener(
67         new ActionListener() // anonymous inner class
68     {
69         // process rightJButton event
70         @Override
71         public void actionPerformed(ActionEvent event)
72         {
73             layout.setAlignment(FlowLayout.RIGHT);
74
75             // realign attached components
76             layout.layoutContainer(container);
77         }
78     });
79 }
80 } // end FlowLayoutFrame constructor
81 } // end class FlowLayoutFrame
```

---

**Fig. 12.39** | FlowLayout allows components to flow over multiple lines. (Part 4 of 4.)

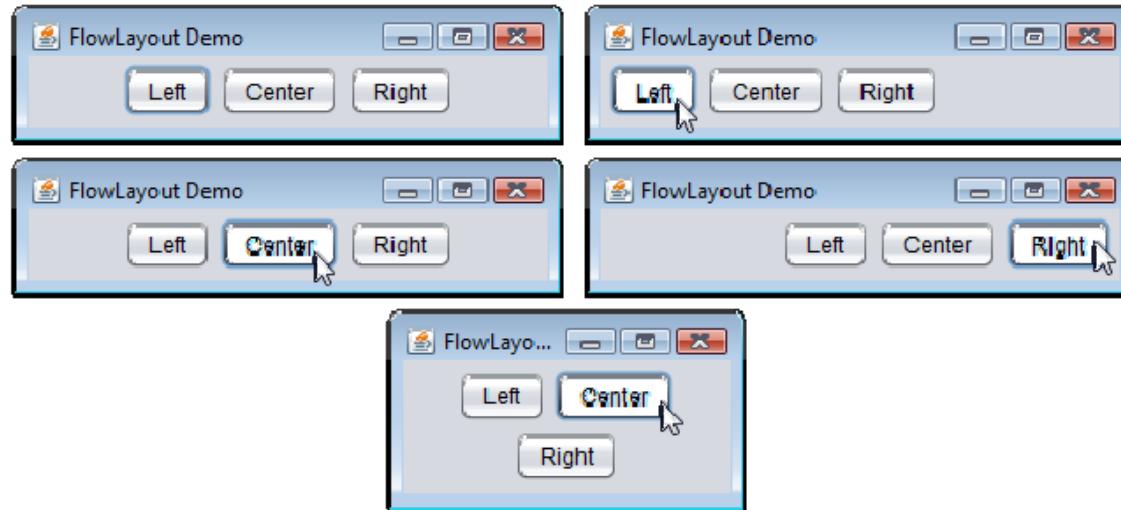


---

```
1 // Fig. 12.40: FlowLayoutDemo.java
2 // Testing FlowLayoutFrame.
3 import javax.swing.JFrame;
4
5 public class FlowLayoutDemo
6 {
7     public static void main(String[] args)
8     {
9         FlowLayoutFrame flowLayoutFrame = new FlowLayoutFrame();
10        flowLayoutFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        flowLayoutFrame.setSize(300, 75);
12        flowLayoutFrame.setVisible(true);
13    }
14 } // end class FlowLayoutDemo
```

---

**Fig. 12.40** | Testing FlowLayoutFrame. (Part 1 of 2.)



**Fig. 12.40** | Testing `FlowLayoutFrame`. (Part 2 of 2.)



## 12.18.1 FlowLayout (cont.)

- ▶ FlowLayout method `setAlignment` changes the alignment for the FlowLayout.
  - `FlowLayout.LEFT`
  - `FlowLayout.CENTER`
  - `FlowLayout.RIGHT`
- ▶ LayoutManager interface method `layoutContainer` (which is inherited by all layout managers) specifies that a container should be rearranged based on the adjusted layout.



## 12.18.2 BorderLayout

- ▶ **BorderLayout**
  - the default layout manager for a `Jframe`
  - arranges components into five regions: NORTH, SOUTH, EAST, WEST and CENTER.
  - NORTH corresponds to the top of the container.
- ▶ **BorderLayout** implements interface `LayoutManager2` (a subinterface of `LayoutManager` that adds several methods for enhanced layout processing).
- ▶ **BorderLayout** limits a `Container` to at most five components—one in each region.
  - The component placed in each region can be a container to which other components are attached.



```
1 // Fig. 12.41: BorderLayoutFrame.java
2 // BorderLayout containing five buttons.
3 import java.awt.BorderLayout;
4 import java.awt.event.ActionListener;
5 import java.awt.event.ActionEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JButton;
8
9 public class BorderLayoutFrame extends JFrame implements ActionListener
10 {
11     private final JButton[] buttons; // array of buttons to hide portions
12     private static final String[] names = {"Hide North", "Hide South",
13         "Hide East", "Hide West", "Hide Center"};
14     private final BorderLayout layout;
15
16     // set up GUI and event handling
17     public BorderLayoutFrame()
18     {
19         super("BorderLayout Demo");
20
21         layout = new BorderLayout(5, 5); // 5 pixel gaps
22        setLayout(layout);
23         buttons = new JButton[names.length];
24 }
```

**Fig. 12.41** | BorderLayout containing five buttons. (Part 1 of 3.)



---

```
25     // create JButtons and register listeners for them
26     for (int count = 0; count < names.length; count++)
27     {
28         buttons[count] = new JButton(names[count]);
29         buttons[count].addActionListener(this);
30     }
31
32     add(buttons[0], BorderLayout.NORTH);
33     add(buttons[1], BorderLayout.SOUTH);
34     add(buttons[2], BorderLayout.EAST);
35     add(buttons[3], BorderLayout.WEST);
36     add(buttons[4], BorderLayout.CENTER);
37 }
38
```

---

**Fig. 12.41** | BorderLayout containing five buttons. (Part 2 of 3.)



---

```
39     // handle button events
40     @Override
41     public void actionPerformed(ActionEvent event)
42     {
43         // check event source and lay out content pane correspondingly
44         for (JButton button : buttons)
45         {
46             if (event.getSource() == button)
47                 button.setVisible(false); // hide the button that was clicked
48             else
49                 button.setVisible(true); // show other buttons
50         }
51         layout.layoutContainer(getContentPane()); // lay out content pane
52     }
53 } // end class BorderLayoutFrame
```

---

**Fig. 12.41** | BorderLayout containing five buttons. (Part 3 of 3.)



## 12.18.2 BorderLayout (cont.)

- ▶ BorderLayout constructor arguments specify the number of pixels between components that are arranged horizontally (**horizontal gap space**) and between components that are arranged vertically (**vertical gap space**), respectively.
  - The default is one pixel of gap space horizontally and vertically.



## Look-and-Feel Observation 12.18

*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 12.5

*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's no error that indicates this problem.*

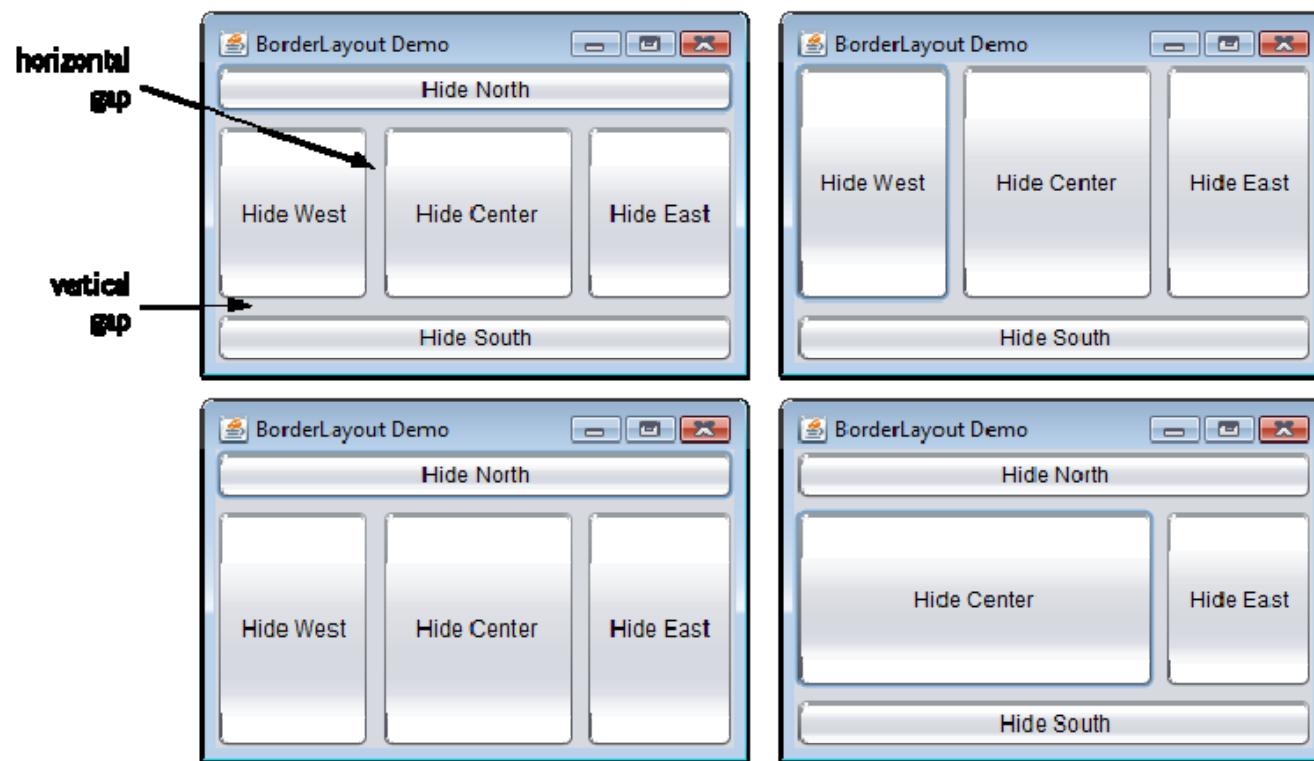


---

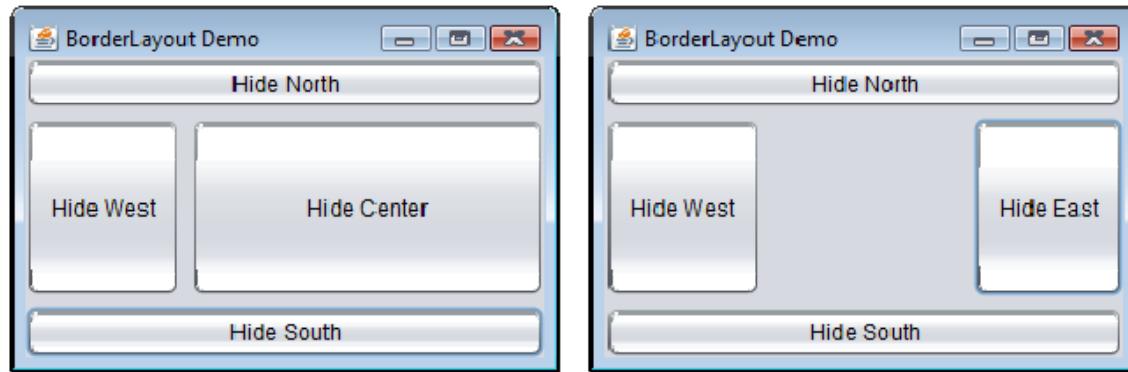
```
1 // Fig. 12.42: BorderLayoutDemo.java
2 // Testing BorderLayoutFrame.
3 import javax.swing.JFrame;
4
5 public class BorderLayoutDemo
6 {
7     public static void main(String[] args)
8     {
9         BorderLayoutFrame borderLayoutFrame = new BorderLayoutFrame();
10        borderLayoutFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        borderLayoutFrame.setSize(300, 200);
12        borderLayoutFrame.setVisible(true);
13    }
14 } // end class BorderLayoutDemo
```

---

**Fig. 12.42** | Testing BorderLayoutFrame. (Part 1 of 3.)



**Fig. 12.42 |** Testing BorderLayoutFrame. (Part 2 of 3.)



**Fig. 12.42 | Testing BorderLayoutFrame. (Part 3 of 3.)**



## 12.18.3 GridLayout

- ▶ `GridLayout` divides the container into a *grid* of *rows* and *columns*.
  - Implements interface `LayoutManager`.
  - Every `Component` has the same width and height.
  - Components are added starting at the top-left cell of the grid and proceeding left to right until the row is full. Then the process continues left to right on the next row of the grid, and so on.
- ▶ Container method `validate` recomputes the container's layout based on the current layout manager and the current set of displayed GUI components.



---

```
1 // Fig. 12.43: GridLayoutFrame.java
2 // GridLayout containing six buttons.
3 import java.awt.GridLayout;
4 import java.awt.Container;
5 import java.awt.event.ActionListener;
6 import java.awt.event.ActionEvent;
7 import javax.swing.JFrame;
8 import javax.swing.JButton;
9
10 public class GridLayoutFrame extends JFrame implements ActionListener
11 {
12     private final JButton[] buttons; // array of buttons
13     private static final String[] names =
14         { "one", "two", "three", "four", "five", "six" };
15     private boolean toggle = true; // toggle between two layouts
16     private final Container container; // frame container
17     private final GridLayout gridLayout1; // first GridLayout
18     private final GridLayout gridLayout2; // second GridLayout
19 }
```

---

**Fig. 12.43** | GridLayout containing six buttons. (Part 1 of 3.)



```
20 // no-argument constructor
21 public GridLayoutFrame()
22 {
23     super("GridLayout Demo");
24     gridLayout1 = new GridLayout(2, 3, 5, 5); // 2 by 3; gaps of 5
25     gridLayout2 = new GridLayout(3, 2); // 3 by 2; no gaps
26     container = getContentPane();
27     setLayout(gridLayout1);
28     buttons = new JButton[names.length];
29
30     for (int count = 0; count < names.length; count++)
31     {
32         buttons[count] = new JButton(names[count]);
33         buttons[count].addActionListener(this); // register listener
34         add(buttons[count]); // add button to JFrame
35     }
36 }
37
```

**Fig. 12.43** | GridLayout containing six buttons. (Part 2 of 3.)



---

```
38     // handle button events by toggling between layouts
39     @Override
40     public void actionPerformed(ActionEvent event)
41     {
42         if (toggle) // set layout based on toggle
43             container.setLayout(gridLayout2);
44         else
45             container.setLayout(gridLayout1);
46
47         toggle = !toggle;
48         container.validate(); // re-lay out container
49     }
50 } // end class GridLayoutFrame
```

---

**Fig. 12.43** | GridLayout containing six buttons. (Part 3 of 3.)



```
1 // Fig. 12.44: GridLayoutDemo.java
2 // Testing GridLayoutFrame.
3 import javax.swing.JFrame;
4
5 public class GridLayoutDemo
6 {
7     public static void main(String[] args)
8     {
9         GridLayoutFrame gridLayoutFrame = new GridLayoutFrame();
10        gridLayoutFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        gridLayoutFrame.setSize(300, 200);
12        gridLayoutFrame.setVisible(true);
13    }
14 } // end class GridLayoutDemo
```



**Fig. 12.44** | Testing GridLayoutFrame.



## 12.19 Using Panels to Manage More Complex Layouts

- ▶ Complex GUIs often require that each component be placed in an exact location.
  - Often consist of multiple panels, with each panel's components arranged in a specific layout.
- ▶ Class `JPanel` extends `JComponent` and `JComponent` extends class `Container`, so every `JPanel` is a `Container`.
- ▶ Every `JPanel` may have components, including other panels, attached to it with `Container` method `add`.
- ▶ `JPanel` can be used to create a more complex layout in which several components are in a specific area of another container.



---

```
1 // Fig. 12.45: PanelFrame.java
2 // Using a JPanel to help lay out components.
3 import java.awt.GridLayout;
4 import java.awt.BorderLayout;
5 import javax.swing.JFrame;
6 import javax.swing.JPanel;
7 import javax.swing.JButton;
8
9 public class PanelFrame extends JFrame
10 {
11     private final JPanel buttonJPanel; // panel to hold buttons
12     private final JButton[] buttons;
13
14     // no-argument constructor
15     public PanelFrame()
16     {
17         super("Panel Demo");
18         buttons = new JButton[5];
19         buttonJPanel = new JPanel();
20         buttonJPanel.setLayout(new GridLayout(1, buttons.length));
21     }
```

---

**Fig. 12.45** | JPanel with five JButtons in a GridLayout attached to the SOUTH region of a BorderLayout. (Part I of 2.)



---

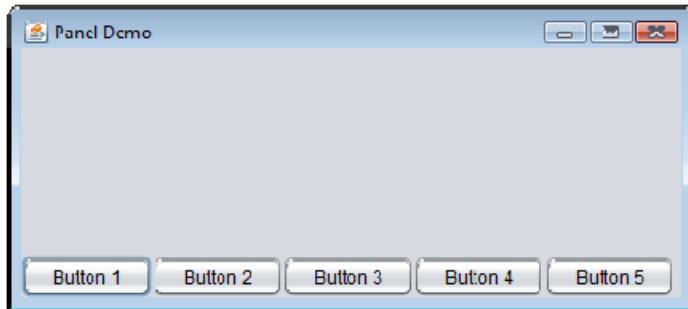
```
22 // create and add buttons
23 for (int count = 0; count < buttons.length; count++)
24 {
25     buttons[count] = new JButton("Button " + (count + 1));
26     buttonJPanel.add(buttons[count]); // add button to panel
27 }
28
29         add(buttonJPanel, BorderLayout.SOUTH); // add panel to JFrame
30     }
31 } // end class PanelFrame
```

---

**Fig. 12.45** | JPanel with five JButtons in a GridLayout attached to the SOUTH region of a BorderLayout. (Part 2 of 2.)



```
1 // Fig. 12.46: PanelDemo.java
2 // Testing PanelFrame.
3 import javax.swing.JFrame;
4
5 public class PanelDemo extends JFrame
6 {
7     public static void main(String[] args)
8     {
9         PanelFrame panelFrame = new PanelFrame();
10        panelFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        panelFrame.setSize(450, 200);
12        panelFrame.setVisible(true);
13    }
14 } // end class PanelDemo
```



**Fig. 12.46** | Testing PanelFrame.



## 12.20 JTextArea

- ▶ A `JTextArea` provides an area for manipulating multiple lines of text.
- ▶ `JTextArea` is a subclass of `JTextComponent`, which declares common methods for `JTextField`s, `JTextArea`s and several other text-based GUI components.



```
1 // Fig. 12.47: TextAreaFrame.java
2 // Copying selected text from one JTextArea to another.
3 import java.awt.event.ActionListener;
4 import java.awt.event.ActionEvent;
5 import javax.swing.Box;
6 import javax.swing.JFrame;
7 import javax.swing.JTextArea;
8 import javax.swing.JButton;
9 import javax.swing.JScrollPane;
10
11 public class TextAreaFrame extends JFrame
12 {
13     private final JTextArea textArea1; // displays demo string
14     private final JTextArea textArea2; // highlighted text is copied here
15     private final JButton copyJButton; // initiates copying of text
16
17     // no-argument constructor
18     public TextAreaFrame()
19     {
20         super("TextArea Demo");
21         Box box = Box.createHorizontalBox(); // create box
22         String demo = "This is a demo string to\n" +
23             "illustrate copying text\nfrom one textarea to \n" +
24             "another textarea using an\nexternal event\n";
```

**Fig. 12.47** | Copying selected text from one JTextArea to another. (Part I of 2.)

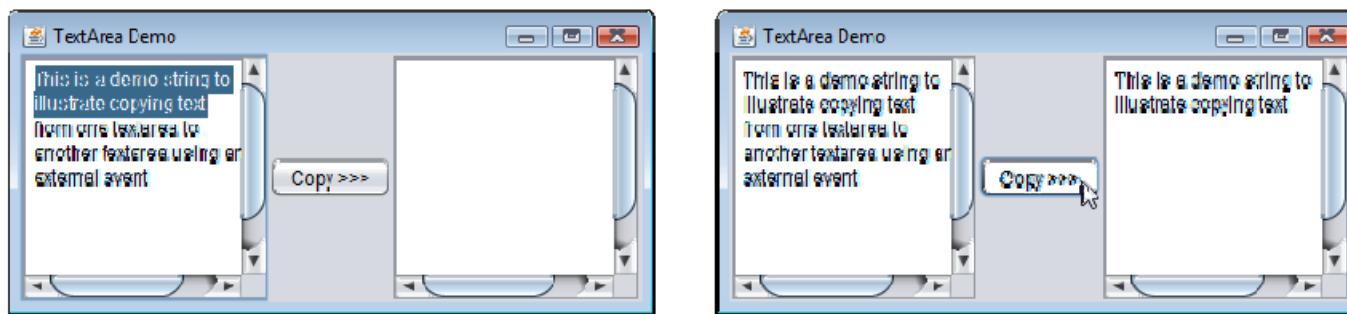


```
25
26     textArea1 = new JTextArea(demo, 10, 15);
27     box.add(new JScrollPane(textArea1)); // add scrollpane
28
29     copyJButton = new JButton("Copy >>>"); // create copy button
30     box.add(copyJButton); // add copy button to box
31     copyJButton.addActionListener(
32         new ActionListener() // anonymous inner class
33     {
34         // set text in textArea2 to selected text from textArea1
35         @Override
36         public void actionPerformed(ActionEvent event)
37         {
38             textArea2.setText(textArea1.getSelectedText());
39         }
40     }
41 );
42
43     textArea2 = new JTextArea(10, 15);
44     textArea2.setEditable(false);
45     box.add(new JScrollPane(textArea2)); // add scrollpane
46
47     add(box); // add box to frame
48 }
49 } // end class TextAreaFrame
```

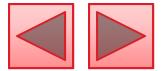
**Fig. 12.47** | Copying selected text from one JTextArea to another. (Part 2 of 2.)



```
1 // Fig. 12.48: TextAreaDemo.java
2 // Testing TextAreaFrame.
3 import javax.swing.JFrame;
4
5 public class TextAreaDemo
6 {
7     public static void main(String[] args)
8     {
9         TextAreaFrame textAreaFrame = new TextAreaFrame();
10        textAreaFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        textAreaFrame.setSize(425, 200);
12        textAreaFrame.setVisible(true);
13    }
14 } // end class TextAreaDemo
```



**Fig. 12.48** | Testing TextAreaFrame.



## 12.20 JTextArea

- ▶ A `JTextArea` provides an area for manipulating multiple lines of text.
- ▶ `JTextArea` is a subclass of `JTextComponent`.



## Look-and-Feel Observation 12.19

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



## 12.20 JTextArea (cont.)

- ▶ `Box` is a subclass of `Container` that uses a `BoxLayout` to arrange the GUI components horizontally or vertically.
- ▶ `Box static` method `createHorizontalBox` creates a `Box` that arranges components left to right in the order that they are attached.
- ▶ `JTextArea` method `getSelectedText` (inherited from `JTextComponent`) returns the selected text from a `JTextArea`.
- ▶ `JTextArea` method `setText` changes the text in a `JTextArea`.
- ▶ When text reaches the right edge of a `JTextArea` the text can wrap to the next line.
  - Referred to as `line wrapping`.
  - By default, `JTextArea` does not wrap lines.



## 12.20 JTextArea (cont.)

- ▶ You can set the horizontal and vertical `scrollbar policies` of a `JScrollPane` when it's constructed.
- ▶ You can also use `JScrollPane` methods `setHorizontalScrollBarPolicy` and `setVerticalScrollBarPolicy` to change the scrollbar policies.



## 12.20 JTextArea (cont.)

- ▶ Class JScrollPane declares the constants
  - `JScrollPane.VERTICAL_SCROLLBAR_ALWAYS`  
`JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS`
  - to indicate that a scrollbar should always appear, constants
    - `JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED`  
`JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED`
    - to indicate that a scrollbar should appear only if necessary (the defaults) and constants
      - `JScrollPane.VERTICAL_SCROLLBAR_NEVER`  
`JScrollPane.HORIZONTAL_SCROLLBAR_NEVER`
      - to indicate that a scrollbar should never appear.
  - ▶ If policy is set to `HORIZONTAL_SCROLLBAR_NEVER`, a `JTextArea` attached to the `JScrollPane` will automatically wrap lines.