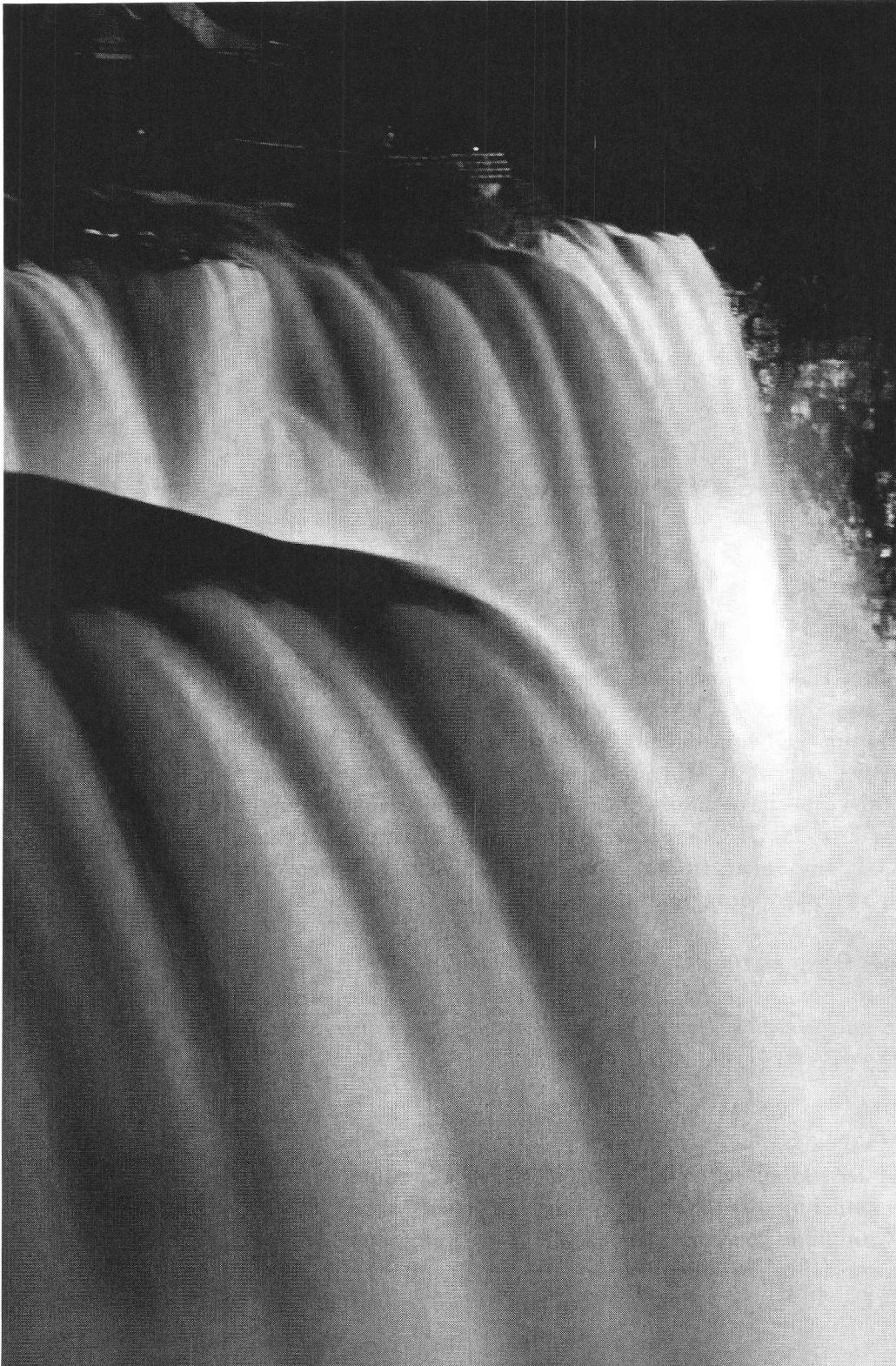


14

GUI Components: Part I



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

—Lewis Carroll

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

—Gaston Bachelard

*You pays your money and you
takes your choice.*

—Punch

Objectives

In this chapter you'll learn:

- How to use Java's elegant, cross-platform Nimbus look-and-feel.
- To build GUIs and handle events generated by user interactions with GUIs.
- To understand the packages containing GUI components, event-handling classes and interfaces.
- To create and manipulate buttons, labels, lists, text fields and panels.
- To handle mouse events and keyboard events.
- To use layout managers to arrange GUI components.

Outline

-
- | | |
|--|---|
| <p>14.1 Introduction</p> <p>14.2 Java's New Nimbus Look-and-Feel</p> <p>14.3 Simple GUI-Based Input/Output with <code>JOptionPane</code></p> <p>14.4 Overview of Swing Components</p> <p>14.5 Displaying Text and Images in a Window</p> <p>14.6 Text Fields and an Introduction to Event Handling with Nested Classes</p> <p>14.7 Common GUI Event Types and Listener Interfaces</p> <p>14.8 How Event Handling Works</p> <p>14.9 <code>JButton</code></p> <p>14.10 Buttons That Maintain State</p> <ul style="list-style-type: none"> 14.10.1 <code>JCheckBox</code> 14.10.2 <code>JRadioButton</code> <p>14.11 <code>JComboBox</code>; Using an Anonymous Inner Class for Event Handling</p> | <p>14.12 <code>JList</code></p> <p>14.13 Multiple-Selection Lists</p> <p>14.14 Mouse Event Handling</p> <p>14.15 Adapter Classes</p> <p>14.16 <code>JPanel</code> Subclass for Drawing with the Mouse</p> <p>14.17 Key Event Handling</p> <p>14.18 Introduction to Layout Managers</p> <ul style="list-style-type: none"> 14.18.1 <code>FlowLayout</code> 14.18.2 <code>BorderLayout</code> 14.18.3 <code>GridLayout</code> <p>14.19 Using Panels to Manage More Complex Layouts</p> <p>14.20 <code>JTextArea</code></p> <p>14.21 Wrap-Up</p> |
|--|---|
-

[Summary](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#) | [Making a Difference](#)

14.1 Introduction

A **graphical user interface (GUI)** presents a user-friendly mechanism for interacting with an application. A GUI (pronounced “GOO-ee”) gives an application a distinctive “look and feel.” GUIs are built from **GUI components**. These are sometimes called controls or widgets—short for window gadgets. A GUI component is an object with which the user interacts via the mouse, the keyboard or another form of input, such as voice recognition. In this chapter and Chapter 25, GUI Components: Part 2, you’ll learn about many of Java’s so-called **Swing GUI components** from the `javax.swing` package. We cover other GUI components as they’re needed throughout the rest of the book.



Look-and-Feel Observation 14.1

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.

IDE Support for GUI Design

Many IDEs provide GUI design tools with which you can specify a component’s exact size and location in a visual manner by using the mouse. The IDE generates the GUI code for you. Though this greatly simplifies creating GUIs, each IDE generates this code differently. For this reason, we wrote the GUI code by hand.

Sample GUI: The SwingSet3 Demo Application

As an example of a GUI, consider Fig. 14.1, which shows the SwingSet3 application that's available at download.java.net/javadesktop/swingset3/SwingSet3.jnlp. This application is a nice way for you to browse through the various GUI components provided by Java's Swing GUI APIs. Simply click a component name (e.g., JFrame, JTabbedPane, etc.) in the **GUI Components** area at the left of the window to see a demonstration of the GUI component in the right side of the window. The source code for each demo is shown in the text area at the bottom of the window. We've labeled a few of the GUI components in the application. At the top of the window is a **title bar** that contains the window's title. Below that is a **menu bar** containing **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**; the user can click the down arrow at the right side of the box to select from a list of items. The menus, buttons and combo box are part of the application's GUI. They enable you to interact with the application.

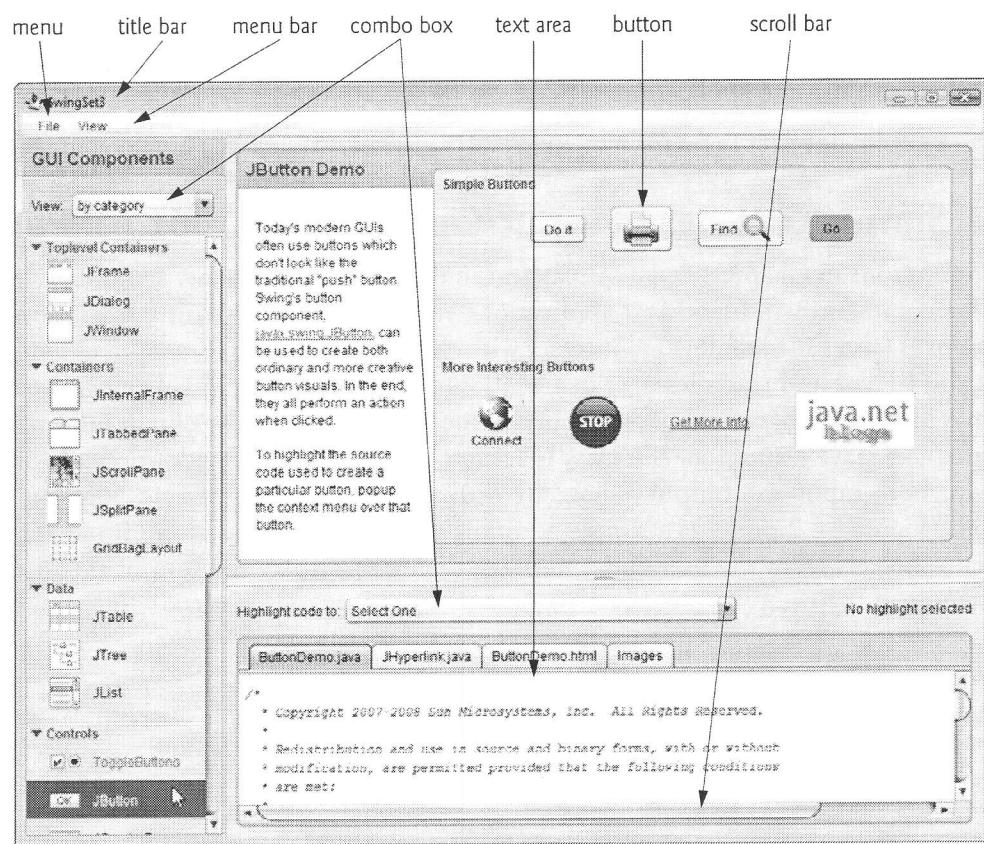


Fig. 14.1 | SwingSet3 application demonstrates many of Java's Swing GUI components.

14.2 Java's New Nimbus Look-and-Feel

In Java SE 6 update 10, Java's elegant, cross-platform look-and-feel known as **Nimbus** was introduced. For GUI screen captures like Fig. 14.1, we've configured our systems to use Nimbus as the default look-and-feel. There are three ways that you can use Nimbus:

1. Set it as the default for all Java applications that run on your computer.
2. Set it as the look-and-feel at the time that you launch an application by passing a command-line argument to the `java` command.
3. Set it as the look-and-feel programmatically in your application (see Section 25.6).

To set Nimbus as the default for all Java applications, you must 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
```

For more information on locating these installation folders visit

bit.ly/JavaInstallationInstructions

In addition to the standalone JRE, there is a JRE nested in your JDK's installation folder. If you're using an IDE that depends on the JDK, you may also need to place the `swing.properties` file in the nested `jre` folder's `lib` folder.

If you prefer 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.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

14.3 Simple GUI-Based Input/Output with JOptionPane

The applications in Chapters 2–10 display text in the command window and obtain input from the command window. Most applications you use on a daily basis use windows or **dialog boxes** (also called **dialogs**) to interact with the user. For example, an e-mail program allows you to type and read messages in a window the program provides. Dialog boxes are windows in which programs display important messages to the user or obtain information from the user. Java's `JOptionPane` class (package `javax.swing`) provides prebuilt dialog boxes for both input and output. These are displayed by invoking static `JOptionPane` methods. Figure 14.2 presents a simple addition application that uses two **input dialogs** to obtain integers from the user and a **message dialog** to display the sum of the integers the user enters.

```

1 // Fig. 14.2: Addition.java
2 // Addition program that uses JOptionPane for input and output.
3 import javax.swing.JOptionPane; // program uses JOptionPane
4
5 public class Addition
6 {
7     public static void main( String[] args )
8     {
9         // obtain user input from JOptionPane input dialogs
10        String firstNumber =
11            JOptionPane.showInputDialog( "Enter first integer" );
12        String secondNumber =
13            JOptionPane.showInputDialog( "Enter second integer" );

```

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

```

14
15     // convert String inputs to int values for use in a calculation
16     int number1 = Integer.parseInt( firstNumber );
17     int number2 = Integer.parseInt( secondNumber );
18
19     int sum = number1 + number2; // add numbers
20
21     // display result in a JOptionPane message dialog
22     JOptionPane.showMessageDialog( null, "The sum is " + sum,
23         "Sum of Two Integers", JOptionPane.PLAIN_MESSAGE );
24 } // end method main
25 } // end class Addition

```

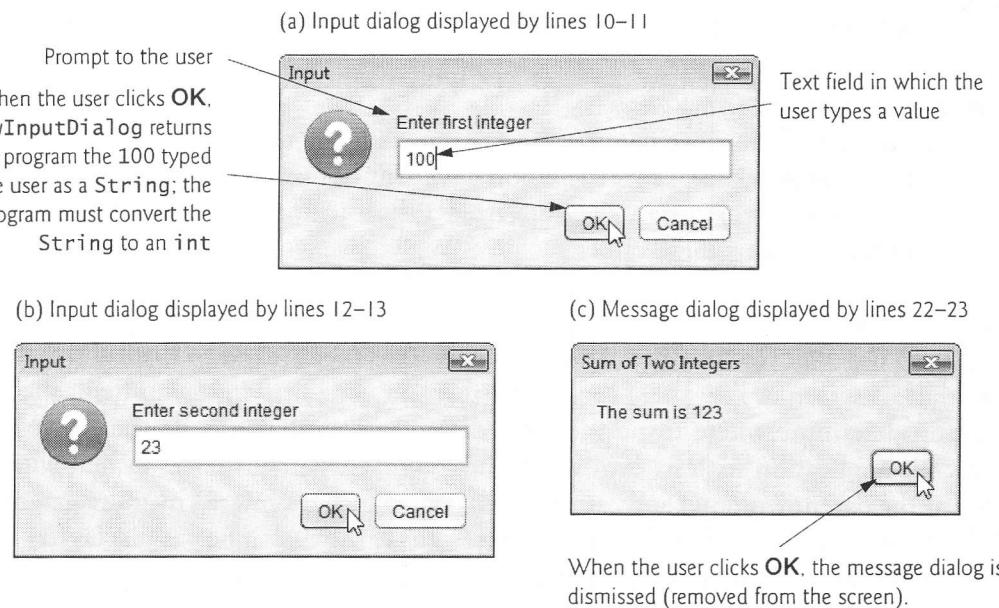


Fig. 14.2 | Addition program that uses JOptionPane for input and output. (Part 2 of 2.)

Input Dialogs

Line 3 imports class JOptionPane. Lines 10–11 declare the local String variable `firstNumber` and assign it the result of the call to JOptionPane static method `showInputDialog`. This method displays an input dialog (see the first screen capture in Fig. 14.2), using the method's String argument ("Enter first integer") as a prompt.



Look-and-Feel Observation 14.2

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).

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** also **dismisses** (hides) the dialog. [Note: If you type in the text field and nothing appears, activate the text field by clicking it with the mouse.] Unlike Scanner, which can be used to input values of *several* types from the user at the keyboard, an input dialog can input only **Strings**. This is typical of most GUI

components. The user can type any characters in the input dialog's text field. Our program assumes that the user enters a valid integer. If the user clicks `Cancel`, `showInputDialog` returns `null`. If the user either types a noninteger value or clicks the `Cancel` button in the input dialog, an exception will occur and the program will not operate correctly. Chapter 11 discussed how to handle such errors. Lines 12–13 display another input dialog that prompts the user to enter the second integer. Each `JOptionPane` dialog that you display is a so called **modal dialog**—while the dialog is on the screen, the user *cannot* interact with the rest of the application.



Look-and-Feel Observation 14.3

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.

Converting Strings to `int` Values

To perform the calculation, we convert the `Strings` that the user entered to `int` values. Recall that the `Integer` class's static method `parseInt` converts its `String` argument to an `int` value. Lines 16–17 assign the converted values to local variables `number1` and `number2`, and line 19 sums these values.

Message Dialogs

Lines 22–23 use `JOptionPane` static method `showMessageDialog` to display a message dialog (the last screen of Fig. 14.2) containing the sum. The first argument helps the Java application determine where to position the dialog box. A dialog is typically displayed from a GUI application with its own window. The first argument refers to that window (known as the parent window) and causes the dialog to appear centered over the parent (as we'll do in Section 14.9). If the first argument is `null`, the dialog box is displayed at the center of your screen. The second argument is the message to display—in this case, the result of concatenating the `String` "The sum is " and the value of `sum`. The third argument—"Sum of Two Integers"—is the `String` that should appear in the *title bar* at the top of the dialog. The fourth argument—`JOptionPane.PLAIN_MESSAGE`—is the type of message dialog to display. A `PLAIN_MESSAGE` dialog does not display an icon to the left of the message. Class `JOptionPane` provides several overloaded versions of methods `showInputDialog` and `showMessageDialog`, as well as methods that display other dialog types. For complete information on class `JOptionPane`, visit download.oracle.com/javase/6/docs/api/javax/swing/JOptionPane.html.



Look-and-Feel Observation 14.4

*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).*

`JOptionPane` Message Dialog Constants

The constants that represent the message dialog types are shown in Fig. 14.3. All message dialog types except `PLAIN_MESSAGE` display an icon to the left of the message. These icons provide a visual indication of the message's importance to the user. A `QUESTION_MESSAGE` icon is the *default icon* for an input dialog box (see Fig. 14.2).

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. 14.3 | JOptionPane static constants for message dialogs.

14.4 Overview of Swing Components

Though it's possible to perform input and output using the JOptionPane dialogs, most GUI applications require more elaborate user interfaces. The remainder of this chapter discusses many GUI components that enable application developers to create robust GUIs. Figure 14.4 lists several basic Swing GUI components that we discuss.

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

Fig. 14.4 | Some basic GUI components.

Swing vs. AWT

There are actually two sets of Java GUI components. In Java's early days, GUIs were built with components from the Abstract Window Toolkit (AWT) in package `java.awt`. These look like the native GUI components of the platform on which a Java program executes. For example, a `Button` object displayed in a Java program running on Microsoft Windows looks like those in other *Windows* applications. On Apple Mac OS X, the `Button` looks like those in other *Mac* applications. Sometimes, even the manner in which a user can interact with an AWT component *differs between platforms*. The component's appearance and the way in which the user interacts with it are known as its *look-and-feel*.



Look-and-Feel Observation 14.5

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.

Lightweight vs. Heavyweight GUI Components

Most Swing components are **lightweight components**—they're written, manipulated and displayed completely in Java. AWT components are **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.

Superclasses of Swing's Lightweight GUI Components

The UML class diagram of Fig. 14.5 shows an inheritance hierarchy of classes from which lightweight Swing components inherit their common attributes and behaviors.

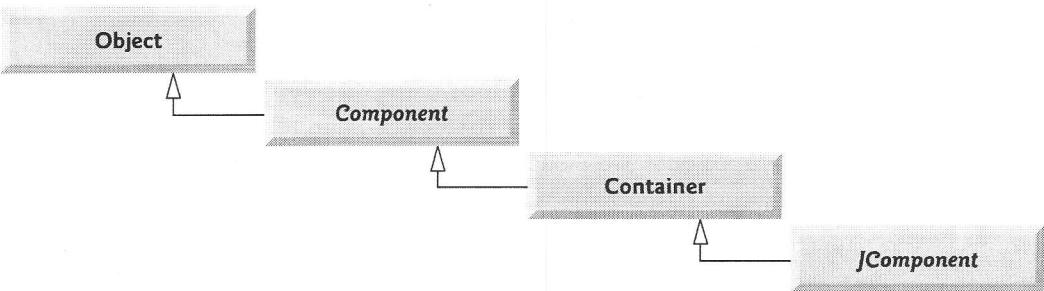


Fig. 14.5 | Common superclasses of the lightweight Swing components.

Class **Component** (package `java.awt`) is a superclass that declares the common features of GUI components in packages `java.awt` and `javax.swing`. Any object that is a **Container** (package `java.awt`) can be used to organize **Components** by attaching the **Components** to the **Container**. **Containers** can be placed in other **Containers** to organize a GUI.

Class **JComponent** (package `javax.swing`) is a subclass of **Container**. **JComponent** is the superclass of all lightweight Swing components and declares their common attributes and behaviors. Because **JComponent** is a subclass of **Container**, all lightweight Swing components are also **Containers**. Some common features supported by **JComponent** include:

1. A pluggable look-and-feel for customizing the appearance of components (e.g., for use on particular platforms). You'll see an example of this in Section 25.6.
2. Shortcut keys (called **mnemonics**) for direct access to GUI components through the keyboard. You'll see an example of this in Section 25.4.
3. Brief descriptions of a GUI component's purpose (called **tool tips**) that are displayed when the mouse cursor is positioned over the component for a short time. You'll see an example of this in the next section.
4. Support for accessibility, such as braille screen readers for the visually impaired.
5. Support for user-interface localization—that is, customizing the user interface to display in different languages and use local cultural conventions.

14.5 Displaying Text and Images in a Window

Our next example introduces a framework for building GUI applications. Several concepts in this framework will appear in many of our GUI applications. This is our first example in which the application appears in its own window. Most windows you'll create 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` that provides the basic attributes and behaviors of a window—a title bar at the top, and buttons to minimize, maximize and close the window. Since an application's GUI is typically specific to the application, most of our examples will consist of two classes—a subclass of `JFrame` that helps us demonstrate new GUI concepts and an application class in which `main` creates and displays the application's primary window.

Labeling GUI Components

A typical GUI consists of many components. GUI designers often provide text stating the purpose of each. Such text is known as a **label** and is created with a `JLabel`—a subclass of `JComponent`. A `JLabel` displays read-only text, an image, or both text and an image. Applications rarely change a label's contents after creating it.

Look-and-Feel Observation 14.6



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

The application of Figs. 14.6–14.7 demonstrates several `JLabel` features and presents the framework we use in most of our GUI examples. We did not highlight the code in this example, since most of it is new. [Note: There are many more features for each GUI component than we can cover in our examples. To learn the complete details of each GUI component, visit its page in the online documentation. For class `JLabel`, visit download.oracle.com/javase/6/docs/api/javax/swing/JLabel.html.]

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

Fig. 14.6 | `JLabels` with text and icons. (Part 1 of 2.)

```

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

```

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

```

1 // Fig. 14.7: 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(); // create LabelFrame
10        labelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        labelFrame.setSize( 260, 180 ); // set frame size
12        labelFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class LabelTest

```

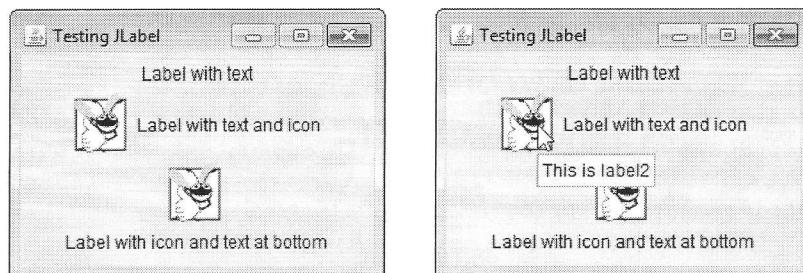


Fig. 14.7 | Test class for LabelFrame.

Class `LabelFrame` (Fig. 14.6) is a subclass of `JFrame`. We'll use an instance of class `LabelFrame` to display a window containing three `JLabel`s. Lines 3–8 import the classes used in class `LabelFrame`. The class extends `JFrame` to inherit the features of a window. Lines 12–14 declare the three `JLabel` instance variables that are instantiated in the `LabelFrame` constructor (lines 17–41). Typically, the `JFrame` subclass's constructor builds the GUI that's displayed in the window when the application executes. Line 19 invokes super-class `JFrame`'s constructor with the argument "Testing `JLabel`". `JFrame`'s constructor uses this `String` as the text in the window's title bar.

Specifying the Layout

When building a GUI, you must attach each GUI component to a container, such as a window created with a `JFrame`. Also, you typically must decide *where* to position each GUI component—known as specifying the layout. Java provides several **layout managers** that can help you position components, as you'll learn at the end of this chapter and in Chapter 25.

Many IDEs provide GUI design tools in which you can specify components' exact sizes and locations in a visual manner by using the mouse; then the IDE will generate the GUI code for you. Such IDEs can greatly simplify GUI creation.

To ensure that our GUIs can be used with *any* IDE, we did *not* use an IDE to create the GUI code. We use Java's layout managers to size and position components. With the `FlowLayout` layout manager, components are placed on a container from left to right in the order in which they're added. When no more components can fit on the current line, they continue to display left to right on the next line. If the container is resized, a `FlowLayout` *reflows* the components, possibly with fewer or more rows based on the new container width. Every container has a default layout, which we're changing for `LabelFrame` to a `FlowLayout` (line 20). Method `setLayout` is inherited into class `LabelFrame` indirectly from class `Container`. The argument to the method must be an object of a class that implements the `LayoutManager` interface (e.g., `FlowLayout`). Line 20 creates a new `FlowLayout` object and passes its reference as the argument to `setLayout`.

Creating and Attaching `label11`

Now that we've specified the window's layout, we can begin creating and attaching GUI components to the window. Line 23 creates a `JLabel` object and passes "Label with text" to the constructor. The `JLabel` displays this text on the screen as part of the application's GUI. Line 24 uses method `setToolTipText` (inherited by `JLabel` from `JComponent`) to specify the tool tip that's displayed when the user positions the mouse cursor over the `JLabel` in the GUI. You can see a sample tool tip in the second screen capture of Fig. 14.7. When you execute this application, try positioning the mouse over each `JLabel` to see its tool tip. Line 25 attaches `label11` to the `LabelFrame` by passing `label11` to the `add` method, which is inherited indirectly from class `Container`.



Common Programming Error 14.1

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 14.7

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.

The Icon Interface and Class ImageIcon

Icons are a popular way to enhance the look-and-feel of an application and are also commonly used to indicate functionality. For example, the same icon is used to play most of today's media on devices like DVD players and MP3 players. Several Swing components can display images. An icon is normally specified with an **Icon** argument to a constructor or to the component's **setIcon** method. An **Icon** is an object of any class that implements interface **Icon** (package `javax.swing`). Class **ImageIcon** supports several image formats, including Graphics Interchange Format (GIF), Portable Network Graphics (PNG) and Joint Photographic Experts Group (JPEG).

Line 28 declares an **ImageIcon**. The file `bug1.png` contains the image to load and store in the **ImageIcon** object. This image is included in the directory for this example. The **ImageIcon** object is assigned to **Icon** reference `bug`.

Loading an Image Resource

In line 28, the expression `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. That reference is then used to invoke `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. As we discussed in Chapter 1, the JVM loads class declarations into memory, using a class loader. 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. In this example, the image file is stored in the same location as the `LabelFrame.class` file. The techniques described here enable an application to load image files from locations that are relative to the class file's location.

Creating and Attaching label2

Lines 29–30 use another `JLabel` constructor to create a `JLabel` that displays the text "Label with text and icon" and the `Icon` `bug` created in line 28. The last constructor argument indicates that the label's contents are left justified, or left aligned (i.e., the icon and text are at the left side of the label's area on the screen). Interface `SwingConstants` (package `javax.swing`) declares a set of common integer constants (such as `SwingConstants.LEFT`) that are used with many Swing components. By default, the text appears to the right of the image when a label contains both text and an image. The horizontal and vertical alignments of a `JLabel` can be set with methods **setHorizontalAlignment** and **setVerticalAlignment**, respectively. Line 31 specifies the tool-tip text for `label2`, and line 32 adds `label2` to the `JFrame`.

Creating and Attaching label3

Class `JLabel` provides methods to change a label's appearance after it's been instantiated. Line 34 creates an empty `JLabel` with the no-argument constructor. Line 35 uses `JLabel` method **setText** to set the text displayed on the label. Method **getText** can be used to retrieve the current text displayed on a label. Line 36 uses `JLabel` method **setIcon** to specify the `Icon` to display on the label. Method **getIcon** can be used to retrieve the current `Icon` displayed on a label. Lines 37–38 use `JLabel` methods **setHorizontalTextPosition** and **setVerticalTextPosition** to specify the text position in the label. In this case, the text will be centered horizontally and will appear at the bottom of the label. Thus, the `Icon`

will appear above the text. The horizontal-position constants in `SwingConstants` are `LEFT`, `CENTER` and `RIGHT` (Fig. 14.8). The vertical-position constants in `SwingConstants` are `TOP`, `CENTER` and `BOTTOM` (Fig. 14.8). Line 39 sets the tool-tip text for `label3`. Line 40 adds `label3` to the `JFrame`.

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

Fig. 14.8 | Positioning constants (`static` members of interface `SwingConstants`).

Creating and Displaying a `LabelFrame` Window

Class `LabelTest` (Fig. 14.7) creates an object of class `LabelFrame` (line 9), then specifies the default close operation for the window. By default, closing a window simply hides the window. However, when the user closes the `LabelFrame` window, we would like the application to terminate. Line 10 invokes `LabelFrame`'s `setDefaultCloseOperation` method (inherited from class `JFrame`) with constant `JFrame.EXIT_ON_CLOSE` as the argument to indicate that the program should terminate when the window is closed by the user. This line is important. Without it the application will not terminate when the user closes the window. Next, line 11 invokes `LabelFrame`'s `setSize` method to specify the width and height of the window in pixels. Finally, line 12 invokes `LabelFrame`'s `setVisible` method with the argument `true` to display the window on the screen. Try resizing the window to see how the `FlowLayout` changes the `JLabel` positions as the window width changes.

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

Normally, a user interacts with an application's GUI to indicate the tasks that the application should perform. For example, when you write an e-mail in an e-mail application, clicking the `Send` button tells the application to send the e-mail to the specified e-mail addresses. 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. Some common user interactions that cause an application to perform a task include clicking a button, typing in a text field, selecting an item from a menu, closing a window and moving the mouse. The code that performs a task in response to an event is called an **event handler**, and the overall process of responding to events is known as **event handling**.

Let's consider two other GUI components that can generate events—`JTextFields` and `JPasswordFields` (package `javax.swing`). Class `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. Each of these components is a single-line area in which the user can enter text via the keyboard. Applications can also display text in a `JTextField` (see the output of Fig. 14.10). A `JPasswordField` shows that characters are

being typed as the user enters them, but hides the actual characters with an **echo character**, assuming that they represent a password that should remain known only to the user.

When the user types in a `JTextField` or a `JPasswordField`, then presses *Enter*, an event occurs. Our next example demonstrates how a program can perform a task in response to that event. The techniques shown here are applicable to all GUI components that generate events.

The application of Figs. 14.9–14.10 uses classes `JTextField` and `JPasswordField` to create and manipulate four text fields. When the user types in one of the text fields, then presses *Enter*, the application displays a message dialog box containing the text the user typed. You can type only in the text field that's “in focus.” When you click a component, it *receives the focus*. This is important, because the text field with the focus is the one that generates an event when you press *Enter*. In this example, you press *Enter* in the `JPasswordField`, the password is revealed. We begin by discussing the setup of the GUI, then discuss the event-handling code.

Lines 3–9 import the classes and interfaces we use in this example. Class `TextFieldFrame` extends `JFrame` and declares three `JTextField` variables and a `JPasswordField` variable (lines 13–16). Each of the corresponding text fields is instantiated and attached to the `TextFieldFrame` in the constructor (lines 19–47).

```

1 // Fig. 14.9: TextFieldFrame.java
2 // Demonstrating the JTextField class.
3 import java.awt.FlowLayout;
4 import java.awt.event.ActionListener;
5 import java.awt.event.ActionEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JTextField;
8 import javax.swing.JPasswordField;
9 import javax.swing.JOptionPane;
10
11 public class TextFieldFrame extends JFrame
12 {
13     private JTextField textField1; // text field with set size
14     private JTextField textField2; // text field constructed with text
15     private JTextField textField3; // text field with text and size
16     private JPasswordField passwordField; // password field with text
17
18     // TextFieldFrame constructor adds JTextFields to JFrame
19     public TextFieldFrame()
20     {
21         super( "Testing JTextField and JPasswordField" );
22         setLayout( new FlowLayout() ); // set frame layout
23
24         // construct textfield with 10 columns
25         textField1 = new JTextField( 10 );
26         add( textField1 ); // add textField1 to JFrame
27
28         // construct textfield with default text
29         textField2 = new JTextField( "Enter text here" );
30         add( textField2 ); // add textField2 to JFrame

```

Fig. 14.9 | `JTextFields` and `JPasswordFields`. (Part I of 2.)

```
31      // construct textfield with default text and 21 columns
32      textField3 = new JTextField( "Uneditable text field", 21 );
33      textField3.setEditable( false ); // disable editing
34      add( textField3 ); // add textField3 to JFrame
35
36      // construct passwordfield with default text
37      passwordField = new JPasswordField( "Hidden text" );
38      add( passwordField ); // add passwordField to JFrame
39
40      // register event handlers
41      TextFieldHandler handler = new TextFieldHandler();
42      textField1.addActionListener( handler );
43      textField2.addActionListener( handler );
44      textField3.addActionListener( handler );
45      passwordField.addActionListener( handler );
46
47  } // end TextFieldFrame constructor
48
49  // private inner class for event handling
50  private class TextFieldHandler implements ActionListener
51  {
52      // process text field events
53      public void actionPerformed( ActionEvent event )
54      {
55          String string = ""; // declare string to display
56
57          // user pressed Enter in JTextField textField1
58          if ( event.getSource() == textField1 )
59              string = String.format( "textField1: %s",
60                                     event.getActionCommand() );
61
62          // user pressed Enter in JTextField textField2
63          else if ( event.getSource() == textField2 )
64              string = String.format( "textField2: %s",
65                                     event.getActionCommand() );
66
67          // user pressed Enter in JTextField textField3
68          else if ( event.getSource() == textField3 )
69              string = String.format( "textField3: %s",
70                                     event.getActionCommand() );
71
72          // user pressed Enter in JPasswordField passwordField
73          else if ( event.getSource() == passwordField )
74              string = String.format( "passwordField: %s",
75                                     event.getActionCommand() );
76
77          // display JTextField content
78          JOptionPane.showMessageDialog( null, string );
79      } // end method actionPerformed
80  } // end private inner class TextFieldHandler
81 } // end class TextFieldFrame
```

Fig. 14.9 | JTextFields and JPasswordFields. (Part 2 of 2.)

Creating the GUI

Line 22 sets the `TextFieldFrame`'s layout to `FlowLayout`. Line 25 creates `textField1` with 10 columns of text. A text column's width in *pixels* is determined by the average width of a character in the text field's current font. When text is displayed in a text field and the text is wider than the field itself, a portion of the text at the right side is not visible. If you're typing in a text field and the cursor reaches the right edge, the text at the left edge is pushed off the left side of the field and is no longer visible. Users can use the left and right arrow keys to move through the complete text. Line 26 adds `textField1` to the `JFrame`.

Line 29 creates `textField2` with the initial text "Enter text here" to display in the text field. The width of the field is determined by the width of the default text specified in the constructor. Line 30 adds `textField2` to the `JFrame`.

Line 33 creates `textField3` and calls the `JTextField` constructor with two arguments—the default text "Uneditable text field" to display and the text field's width in columns (21). Line 34 uses method `setEditable` (inherited by `JTextField` from class `JTextComponent`) to make the text field *uneditable*—i.e., the user cannot modify the text in the field. Line 35 adds `textField3` to the `JFrame`.

Line 38 creates `passwordField` with the text "Hidden text" to display in the text field. The width of the field is determined by the width of the default text. When you execute the application, notice that the text is displayed as a string of asterisks. Line 39 adds `passwordField` to the `JFrame`.

Steps Required to Set Up Event Handling for a GUI Component

This example should display a message dialog containing the text from a text field when the user presses *Enter* in that text field. Before an application can respond to an event for a particular GUI component, you must:

1. Create a class that represents the event handler and implements an appropriate interface—known as an **event-listener interface**.
2. Indicate that an object of the class from *Step 1* should be notified when the event occurs—known as **registering the event handler**.

Using a Nested Class to Implement an Event Handler

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**. Nested classes can be **static** or **non-static**. Non-static nested classes are called **inner classes** and are frequently used to implement *event handlers*.

An inner-class object must be created by an object of the top-level class that contains the inner class. Each inner-class object *implicitly* has a reference to an object of its top-level class. The inner-class object is allowed to use this implicit reference to directly access all the variables and methods of the top-level class. A nested class that's **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. As you'll see in Chapter 15, Graphics and Java 2D, the Java 2D graphics API uses static nested classes extensively.

Inner Class `TextFieldHandler`

The event handling in this example is performed by an object of the private inner class `TextFieldHandler` (lines 50–80). This class is **private** because it will be used only to cre-

ate event handlers for the text fields in top-level class `TextFieldFrame`. As with other class members, *inner classes* can be declared `public`, `protected` or `private`. Since event handlers tend to be specific to the application in which they're defined, they're often implemented as `private` inner classes or as *anonymous inner classes* (Section 14.11).

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. When the user presses *Enter* in a `JTextField` or `JPasswordField`, an `ActionEvent` (package `java.awt.event`) occurs. Such an event is processed by an object that implements the interface `ActionListener` (package `java.awt.event`). The information discussed here is available in the Java API documentation for classes `JTextField` and `ActionEvent`. Since `JPasswordField` is a subclass of `JTextField`, `JPasswordField` supports the same events.

To prepare to handle the events in this example, inner class `TextFieldHandler` implements interface `ActionListener` and declares the only method in that interface—`actionPerformed` (lines 53–79). This method specifies the tasks to perform when an `ActionEvent` occurs. So, inner class `TextFieldHandler` satisfies *Step 1* listed earlier in this section. We'll discuss the details of method `actionPerformed` shortly.

Registering the Event Handler for Each Text Field

In the `TextFieldFrame` constructor, line 42 creates a `TextFieldHandler` object and assigns it to variable `handler`. This object's `actionPerformed` method will be called automatically when the user presses *Enter* in any of the GUI's text fields. However, before this can occur, the program must register this object as the event handler for each text field. Lines 43–46 are the event-registration statements that specify `handler` as the event handler for the three `JTextFields` and the `JPasswordField`. The application calls `JTextField` method `addActionListener` to register the event handler for each component. This method receives as its argument an `ActionListener` object, which can be an object of any class that implements `ActionListener`. The object `handler` is an `ActionListener`, because class `TextFieldHandler` implements `ActionListener`. After lines 43–46 execute, the object `handler` **listens for events**. Now, when the user presses *Enter* in any of these four text fields, method `actionPerformed` (line 53–79) in class `TextFieldHandler` is called to handle the event. If an event handler is not registered for a particular text field, the event that occurs when the user presses *Enter* in that text field is **consumed**—i.e., it's simply ignored by the application.



Software Engineering Observation 14.1

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



Common Programming Error 14.2

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

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

In this example, we're using one event-handling object's `actionPerformed` method (lines 53–79) to handle the events generated by four text fields. Since we'd like to output the

name of each text field's instance variable for demonstration purposes, we must determine which text field generated the event each time `actionPerformed` is called. The **event source** is the GUI component with which the user interacted. When the user presses *Enter* while one of the text fields or the password field *has the focus*, the system creates a unique `ActionEvent` object that contains information about the event that just occurred, such as the event source and the text in the text field. The system passes this `ActionEvent` object to the event listener's `actionPerformed` method. Line 55 declares the `String` that will be displayed. The variable is initialized with the **empty string**—a `String` containing no characters. The compiler requires the variable to be initialized in case none of the branches of the nested `if` in lines 58–75 executes.

`ActionEvent` method `getSource` (called in lines 58, 63, 68 and 73) returns a reference to the event source. The condition in line 58 asks, “Is the event source `textField1`?” This condition compares references with the `==` operator to determine if they refer to the same object. If they *both* refer to `textField1`, the user pressed *Enter* in `textField1`. Then, lines 59–60 create a `String` containing the message that line 78 displays in a message dialog. Line 60 uses `ActionEvent` method `getActionCommand` to obtain the text the user typed in the text field that generated the event.

In this example, we display the text of the password in the `JPasswordField` when the user presses *Enter* in that field. Sometimes it's necessary to programmatically process the characters in a password. Class `JPasswordField` method `getPassword` returns the password's characters as an array of type `char`.

Class TextFieldTest

Class `TextFieldTest` (Fig. 14.10) contains the `main` method that executes this application and displays an object of class `TextFieldFrame`. When you execute the application, even the uneditable `JTextField` (`textField3`) can generate an `ActionEvent`. To test this, click the text field to give it the focus, then press *Enter*. Also, the actual text of the password is displayed when you press *Enter* in the `JPasswordField`. Of course, you would normally not display the password!

This application used a single object of class `TextFieldHandler` as the event listener for four text fields. Starting in Section 14.10, you'll see that it's possible to declare several event-listener objects of the same type and register each object for a separate GUI component's event. This technique enables us to eliminate the `if...else` logic used in this example's event handler by providing separate event handlers for each component's events.

```

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

```

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

```
I2     textFieldFrame.setVisible( true ); // display frame
I3 } // end main
I4 } // end class TextFieldTest
```

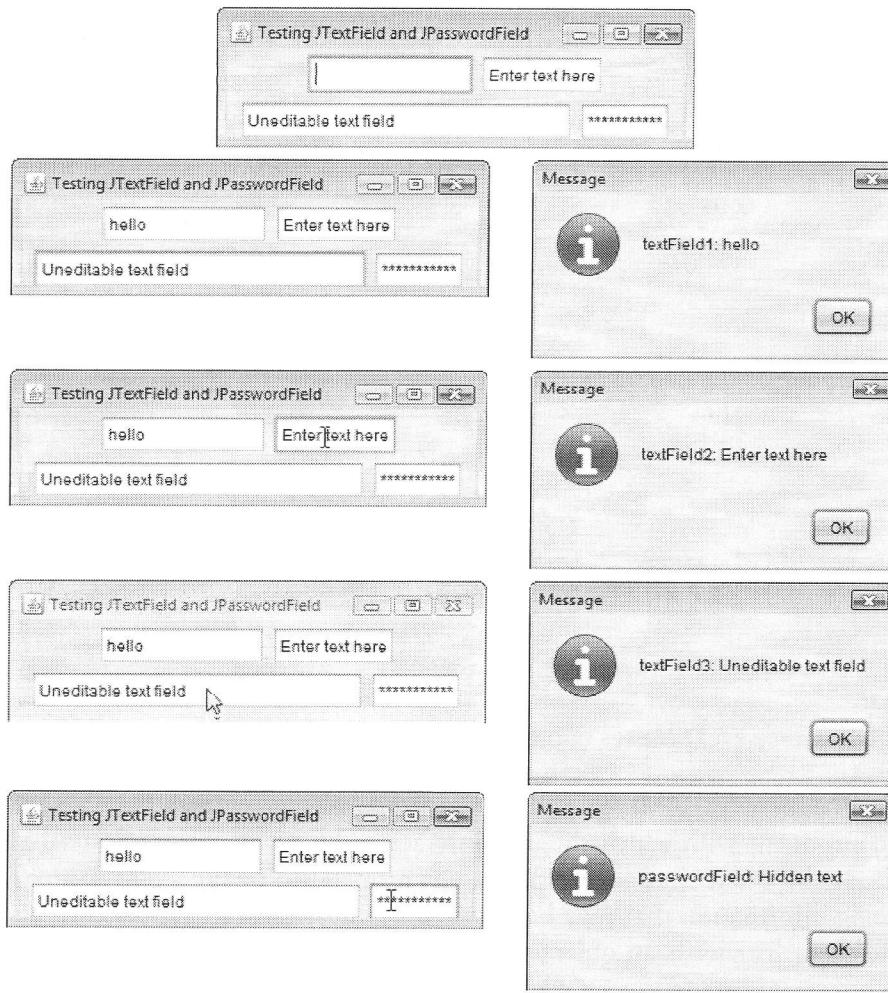


Fig. 14.10 | Test class for TextFieldFrame. (Part 2 of 2.)

14.7 Common GUI Event Types and Listener Interfaces

In Section 14.6, you learned that information about the event that occurs when the user presses *Enter* in a text field is stored in an `ActionEvent` object. Many different types of events can occur when the user interacts with a GUI. The event information is stored in an object of a class that extends `AWTEvent` (from package `java.awt`). Figure 14.11 illustrates a hierarchy containing many event classes from the package `java.awt.event`. Some of these are discussed in this chapter and Chapter 25. These event types are used with both AWT and Swing components. Additional event types that are specific to Swing GUI components are declared in package `javax.swing.event`.

Let's summarize the three parts to the event-handling mechanism that you saw in Section 14.6—the *event source*, the *event object* and the *event listener*. The event source is

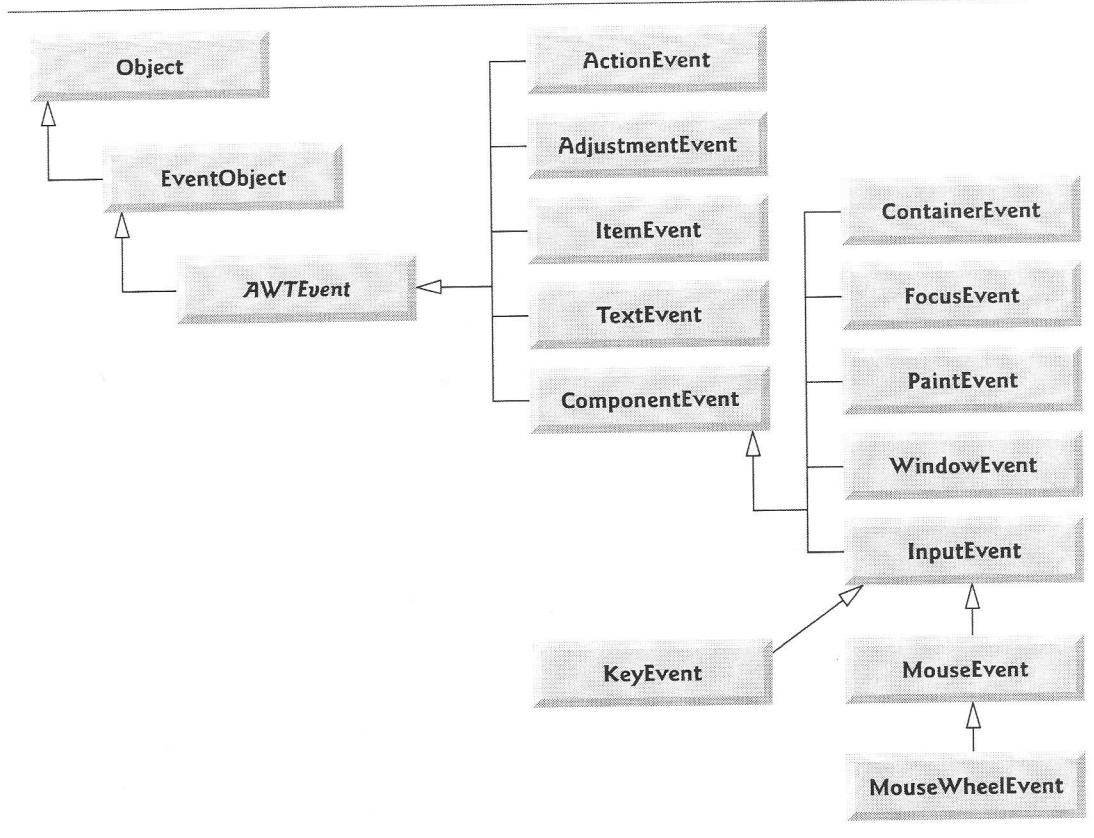


Fig. 14.11 | Some event classes of package `java.awt.event`.

the GUI component with which the user interacts. The event object encapsulates information about the event that occurred, such as a reference to the event source and any event-specific information that may be required by the event listener for it to handle the event. The event listener is an object that's notified by the event source when an event occurs; in effect, it “listens” for an event, and one of its methods executes in response to the event. A method of the event listener receives an event object when the event listener is notified of the event. The event listener then uses the event object to respond to the event. This event-handling model is known as the **delegation event model**—an event's processing is delegated to an object (the event listener) in the application.

For each event-object type, there's typically a corresponding event-listener interface. An event listener for a GUI event is an object of a class that implements one or more of the event-listener interfaces from packages `java.awt.event` and `javax.swing.event`. Many of the event-listener types are common to both Swing and AWT components. Such types are declared in package `java.awt.event`, and some of them are shown in Fig. 14.12. Additional event-listener types that are specific to Swing components are declared in package `javax.swing.event`.

Each event-listener interface specifies one or more event-handling methods that *must* be declared in the class that implements the interface. Recall from Section 10.7 that any class which implements an interface must declare *all* the abstract methods of that interface; otherwise, the class is an abstract class and cannot be used to create objects.

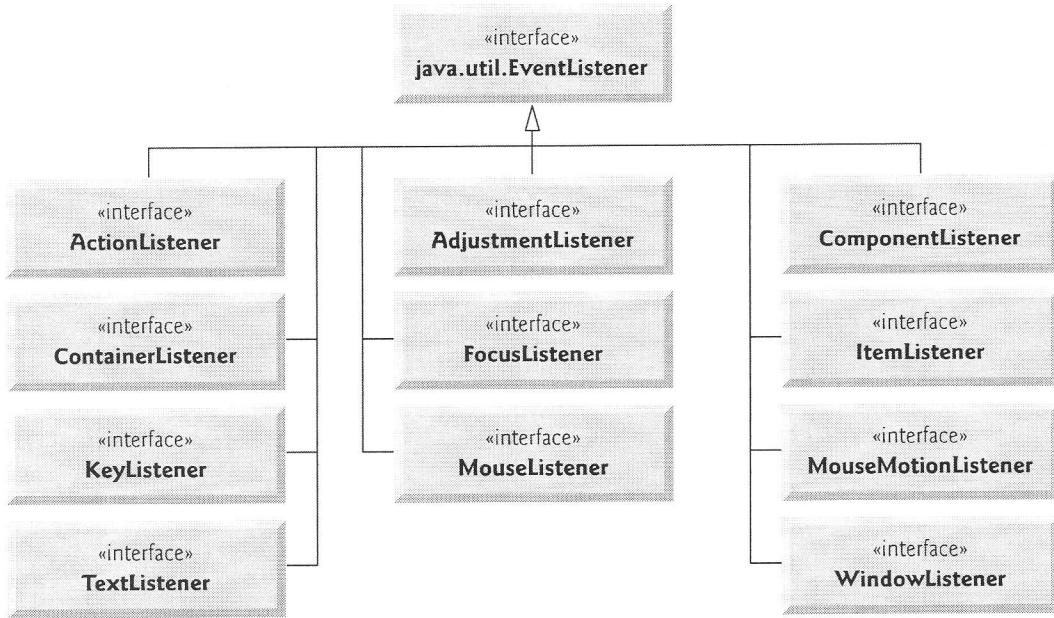


Fig. 14.12 | Some common event-listener interfaces of package `java.awt.event`.

When an event occurs, the GUI component with which the user interacted notifies its *registered listeners* by calling each listener's appropriate *event-handling method*. For example, when the user presses the *Enter* key in a `JTextField`, the registered listener's `actionPerformed` method is called. How did the event handler get registered? How does the GUI component know to call `actionPerformed` rather than another event-handling method? We answer these questions and diagram the interaction in the next section.

14.8 How Event Handling Works

Let's illustrate how the event-handling mechanism works, using `textField1` from the example of Fig. 14.9. We have two remaining open questions from Section 14.7:

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

The first question is answered by the event registration performed in lines 43–46 of Fig. 14.9. Figure 14.13 diagrams `JTextField` variable `textField1`, `TextFieldHandler` variable `handler` and the objects to which they refer.

Registering Events

Every `JComponent` has an instance variable called `listenerList` that refers to an object of class `EventListenerList` (package `javax.swing.event`). Each object of a `JComponent` subclass maintains references to its registered listeners in the `listenerList`. For simplicity, we've diagrammed `listenerList` as an array below the `JTextField` object in Fig. 14.13.

When line 43 of Fig. 14.9

```
textField1.addActionListener( handler );
```

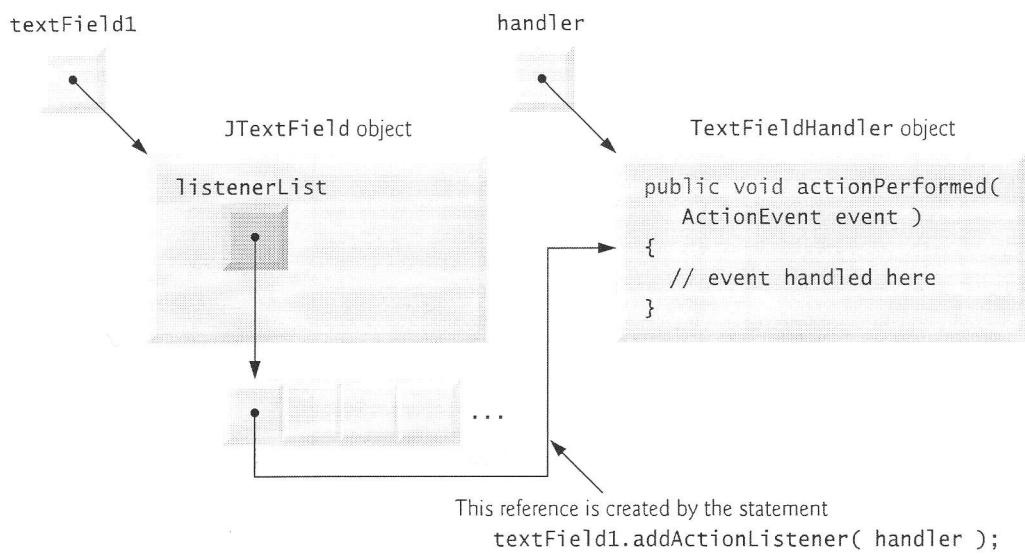


Fig. 14.13 | Event registration for JTextField textField1.

executes, a new entry containing a reference to the TextFieldHandler object is placed in textField1's ListenerList. Although not shown in the diagram, this new entry also includes the listener's type (in this case, ActionListener). Using this mechanism, each lightweight Swing GUI component maintains its own list of *listeners* that were *registered* to handle the component's *events*.

Event-Handler Invocation

The event-listener type is important in answering the second question: 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.

Each *event type* has one or more corresponding *event-listener interfaces*. For example, ActionEvents are handled by ActionListener, MouseEvents by MouseListeners and MouseMotionListeners, and KeyEvents by KeyListeners. When an event occurs, the GUI component receives (from the JVM) a unique *event ID* specifying the event type. The GUI component uses the event ID to decide the listener type to which the event should be dispatched and to decide which method to call on each listener object. For an ActionEvent, the event is dispatched to *every* registered ActionListener's actionPerformed method (the only method in interface ActionListener). For a MouseEvent, the event is dispatched to *every* registered MouseListener or MouseMotionListener, depending on the mouse event that occurs. The MouseEvent's event ID determines which of the several mouse event-handling methods are called. All these decisions are handled for you by the GUI components. All you need to do is register an event handler for the particular event type that your application requires, and the GUI component will ensure that the event handler's appropriate method gets called when the event occurs. We discuss

other event types and event-listener interfaces as they're needed with each new component we introduce.

14.9 JButton

A button is a component the user clicks to trigger a specific action. A Java application can use several types of buttons, including **command buttons**, **checkboxes**, **toggle buttons** and **radio buttons**. Figure 14.14 shows the inheritance hierarchy of the Swing buttons we cover in this chapter. As you can see, all the button types are subclasses of **AbstractButton** (package `javax.swing`), which declares the common features of Swing buttons. In this section, we concentrate on buttons that are typically used to initiate a command.

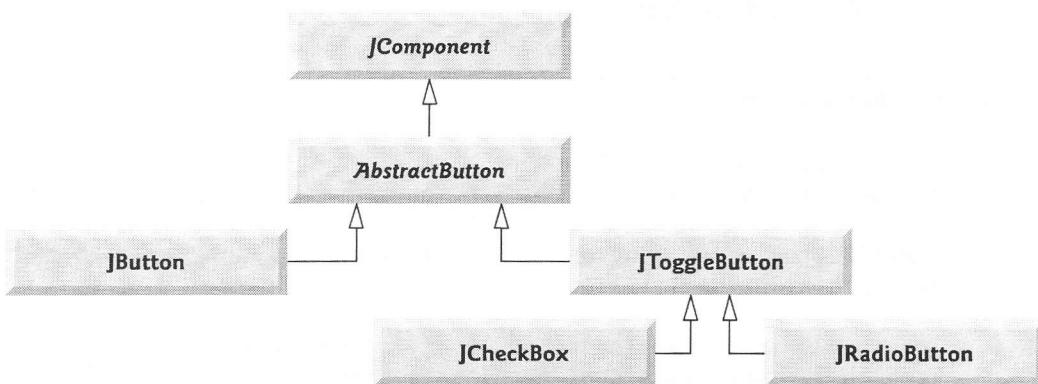


Fig. 14.14 | Swing button hierarchy.

A command button (see Fig. 14.16's output) 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**. A GUI can have many `JButtons`, but each button label should be unique in the portion of the GUI that's currently displayed.



Look-and-Feel Observation 14.8

The text on buttons typically uses book-title capitalization.



Look-and-Feel Observation 14.9

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

The application of Figs. 14.15 and 14.16 creates two `JButtons` and demonstrates that `JButtons` support the display of `Icons`. Event handling for the buttons is performed by a single instance of *inner class* `ButtonHandler` (lines 39–47).

Lines 14–15 declare `JButton` variables `plainJButton` and `fancyJButton`. The corresponding objects are instantiated in the constructor. Line 23 creates `plainJButton` with the button label "Plain Button". Line 24 adds the `JButton` to the `JFrame`.

A `JButton` can display an `Icon`. To provide the user with an extra level of visual interaction with the GUI, a `JButton` can also have a **rollover Icon**—an `Icon` that's displayed

```

1 // Fig. 14.15: ButtonFrame.java
2 // Creating JButtons.
3 import java.awt.FlowLayout;
4 import java.awt.event.ActionListener;
5 import java.awt.event.ActionEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JButton;
8 import javax.swing.Icon;
9 import javax.swing.ImageIcon;
10 import javax.swing.JOptionPane;
11
12 public class ButtonFrame extends JFrame
13 {
14     private JButton plainJButton; // button with just text
15     private JButton fancyJButton; // button with icons
16
17     // ButtonFrame adds JButtons to JFrame
18     public ButtonFrame()
19     {
20         super( "Testing Buttons" );
21         setLayout( new FlowLayout() ); // set frame layout
22
23         plainJButton = new JButton( "Plain Button" ); // button with text
24         add( plainJButton ); // add plainJButton to JFrame
25
26         Icon bug1 = new ImageIcon( getClass().getResource( "bug1.gif" ) );
27         Icon bug2 = new ImageIcon( getClass().getResource( "bug2.gif" ) );
28         fancyJButton = new JButton( "Fancy Button", bug1 ); // set image
29         fancyJButton.setRolloverIcon( bug2 ); // set rollover image
30         add( fancyJButton ); // add fancyJButton to JFrame
31
32         // create new ButtonHandler for button event handling
33         ButtonHandler handler = new ButtonHandler();
34         fancyJButton.addActionListener( handler );
35         plainJButton.addActionListener( handler );
36     } // end ButtonFrame constructor
37
38     // inner class for button event handling
39     private class ButtonHandler implements ActionListener
40     {
41         // handle button event
42         public void actionPerformed( ActionEvent event )
43         {
44             JOptionPane.showMessageDialog( ButtonFrame.this, String.format(
45                 "You pressed: %s", event.getActionCommand() ) );
46         } // end method actionPerformed
47     } // end private inner class ButtonHandler
48 } // end class ButtonFrame

```

Fig. 14.15 | Command buttons and action events.

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. Lines 26–27 (Fig. 14.15) create two ImageIcon objects that represent the default Icon and rollover Icon for the

```

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

```

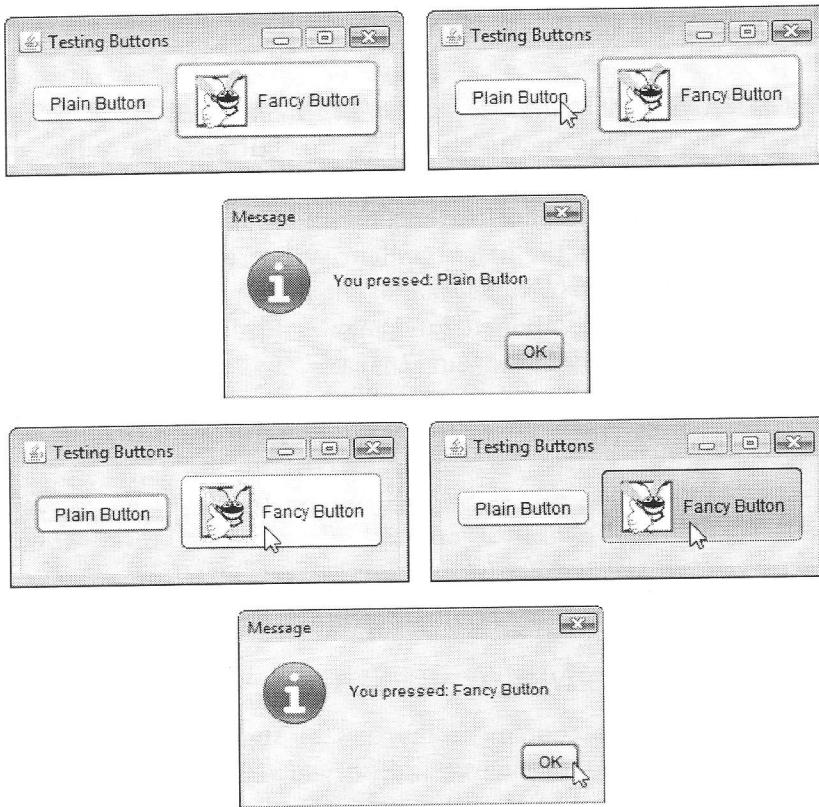


Fig. 14.16 | Test class for ButtonFrame.

JButton created at line 28. Both statements assume that the image files are stored in the same directory as the application. Images are commonly placed in the same directory as the application or a subdirectory like `images`). These image files have been provided for you with the example.

Line 28 creates `fancyButton` with the text "Fancy Button" and the icon `bug1`. By default, the text is displayed to the right of the icon. Line 29 uses `setRolloverIcon` (inherited from class `AbstractButton`) to specify the image displayed on the JButton when the user positions the mouse over it. Line 30 adds the JButton to the JFrame.

**Look-and-Feel Observation 14.10**

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 14.11**

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

JButtons, like JTextFields, generate ActionEvents that can be processed by any ActionListener object. Lines 33–35 create an object of private inner class ButtonHandler and use addActionListener to register it as the event handler for each JButton. Class ButtonHandler (lines 39–47) declares actionPerformed to display a message dialog box containing the label for the button the user pressed. For a JButton event, ActionEvent method getActionCommand returns the label on the JButton.

Accessing the this Reference in an Object of a Top-Level Class From an Inner Class
 When you execute this application and click one of its buttons, notice that the message dialog that appears is centered over the application's window. This occurs because the call to JOptionPane method showMessageDialog (lines 44–45 of Fig. 14.15) uses ButtonFrame.this rather than null as the first argument. When this argument is not null, it represents the so-called *parent GUI component* of the message dialog (in this case the application window is the parent component) and enables the dialog to be centered over that component when the dialog is displayed. ButtonFrame.this represents the this reference of the object of top-level class ButtonFrame.

**Software Engineering Observation 14.2**

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, as in ButtonFrame.this.

14.10 Buttons That Maintain State

The Swing GUI components contain three types of state buttons—**JToggleButton**, **JCheckBox** and **JRadioButton**—that have on/off or true/false values. Classes **JCheckBox** and **JRadioButton** are subclasses of **JToggleButton** (Fig. 14.14). A **JRadioButton** is different from a **JCheckBox** in that normally several **JRadioButtons** are grouped together and are mutually exclusive—only one in the group can be selected at any time, just like the buttons on a car radio. We first discuss class **JCheckBox**.

14.10.1 JCheckBox

The application of Figs. 14.17–14.18 uses two **JCheckBoxes** to select the desired font style of the text displayed in a **JTextField**. When selected, one applies a bold style and the other an italic style. If both are selected, the style is bold and italic. When the application initially executes, neither **JCheckBox** is checked (i.e., they're both false), so the font is plain. Class **CheckBoxTest** (Fig. 14.18) contains the **main** method that executes this application.

```
1 // Fig. 14.17: CheckBoxFrame.java
2 // Creating JCheckBox buttons.
3 import java.awt.FlowLayout;
4 import java.awt.Font;
5 import java.awt.event.ItemListener;
6 import java.awt.event.ItemEvent;
7 import javax.swing.JFrame;
8 import javax.swing.JTextField;
9 import javax.swing.JCheckBox;
10
11 public class CheckBoxFrame extends JFrame
12 {
13     private JTextField textField; // displays text in changing fonts
14     private JCheckBox boldJCheckBox; // to select/deselect bold
15     private JCheckBox italicJCheckBox; // to select/deselect italic
16
17     // CheckBoxFrame constructor adds JCheckboxes to JFrame
18     public CheckBoxFrame()
19     {
20         super( "JCheckBox Test" );
21         setLayout( new FlowLayout() ); // set frame layout
22
23         // set up JTextField and set its font
24         textField = new JTextField( "Watch the font style change", 20 );
25         textField.setFont( new Font( "Serif", Font.PLAIN, 14 ) );
26         add( textField ); // add textField to JFrame
27
28         boldJCheckBox = new JCheckBox( "Bold" ); // create bold checkbox
29         italicJCheckBox = new JCheckBox( "Italic" ); // create italic
30         add( boldJCheckBox ); // add bold checkbox to JFrame
31         add( italicJCheckBox ); // add italic checkbox to JFrame
32
33         // register listeners for JCheckboxes
34         CheckBoxHandler handler = new CheckBoxHandler();
35         boldJCheckBox.addItemListener( handler );
36         italicJCheckBox.addItemListener( handler );
37     } // end CheckBoxFrame constructor
38
39     // private inner class for ItemListener event handling
40     private class CheckBoxHandler implements ItemListener
41     {
42         // respond to checkbox events
43         public void itemStateChanged( ItemEvent event )
44         {
45             Font font = null; // stores the new Font
46
47             // determine which Checkboxes are checked and create Font
48             if ( boldJCheckBox.isSelected() && italicJCheckBox.isSelected() )
49                 font = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
50             else if ( boldJCheckBox.isSelected() )
51                 font = new Font( "Serif", Font.BOLD, 14 );
52             else if ( italicJCheckBox.isSelected() )
53                 font = new Font( "Serif", Font.ITALIC, 14 );
54         }
55     }
56 }
```

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

```

54         else
55             font = new Font( "Serif", Font.PLAIN, 14 );
56
57             textField.setFont( font ); // set textField's font
58         } // end method itemStateChanged
59     } // end private inner class CheckBoxHandler
60 } // end class CheckBoxFrame

```

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

```

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

```

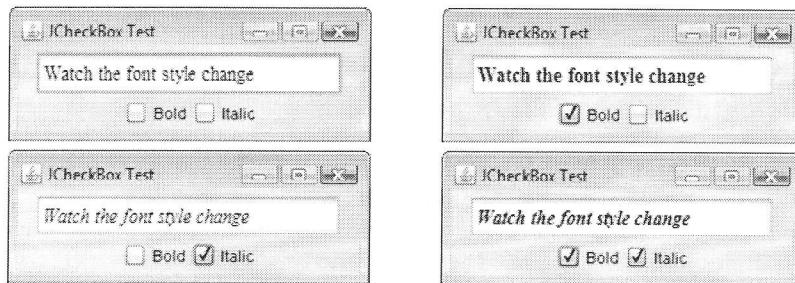


Fig. 14.18 | Test class for CheckBoxFrame.

After the JTextField is created and initialized (Fig. 14.17, line 24), line 25 uses method **setFont** (inherited by JTextField indirectly from class Component) to set the font of the JTextField to a new object of class **Font** (package java.awt). The new Font is initialized with "Serif" (a generic font name that represents a font such as Times and is supported on all Java platforms), Font.PLAIN style and 14-point size. Next, lines 28–29 create two JCheckBox objects. The 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. This event can be handled by an **ItemListener** object, which *must* implement method **itemStateChanged**. In this example, the event handling is performed by an instance of *private inner class CheckBoxHandler* (lines 40–59). Lines 34–36 create an instance of class **CheckBoxHandler** and register it with method **addItemListener** as the listener for both the JCheckBox objects.

CheckBoxHandler method **itemStateChanged** (lines 43–58) is called when the user clicks the **boldJCheckBox** or the **italicJCheckBox**. In this example, we don't need to

know which of the two `JCheckBoxes` was clicked, just whether or not each one is checked. Line 48 uses `JCheckBox` method `isSelected` to determine if both `JCheckBoxes` are selected. If so, line 49 creates a bold italic font by adding the `Font` constants `Font.BOLD` and `Font.ITALIC` for the font-style argument of the `Font` constructor. Line 50 determines whether the `boldJCheckBox` is selected, and if so line 51 creates a bold font. Line 52 determines whether the `italicJCheckBox` is selected, and if so line 53 creates an italic font. If none of the preceding conditions are true, line 55 creates a plain font using the `Font` constant `Font.PLAIN`. Finally, line 57 sets `textField`'s new font, which changes the font in the `JTextField` on the screen.

Relationship Between an Inner Class and Its Top-Level Class

Class `CheckBoxHandler` used variables `boldJCheckBox` (Fig. 14.17, lines 48 and 50), `italicJCheckBox` (lines 48 and 52) and `textField` (line 57) even though they are *not* declared in the inner class. Recall that an inner class has a special relationship with its top-level class—it's allowed to access *all* the variables and methods of the top-level class. `CheckBoxHandler` method `itemStateChanged` (line 43–58) uses this relationship to determine which `JCheckBoxes` are checked and to set the font on the `JTextField`. Notice that none of the code in inner class `CheckBoxHandler` requires an explicit reference to the top-level class object.

14.10.2 `JRadioButton`

Radio buttons (declared with class `JRadioButton`) are similar to checkboxes in that they have two states—selected and not selected (also called deselected). However, radio buttons normally appear as a **group** in which only one button can be selected at a time (see the output of Fig. 14.20). Selecting a different radio button forces all others to be deselected. Radio buttons are used to represent **mutually exclusive options** (i.e., multiple options in the group *cannot* be selected at the same time). The logical relationship between radio buttons is maintained by a `ButtonGroup` object (package `javax.swing`), which itself is not a GUI component. A `ButtonGroup` object organizes a group of buttons and is not itself displayed in a user interface. Rather, the individual `JRadioButton` objects from the group are displayed in the GUI.



Common Programming Error 14.3

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

The application of Figs. 14.19–14.20 is similar to that of Figs. 14.17–14.18. The user can alter the font style of a `JTextField`'s text. The application uses radio buttons that permit only a single font style in the group to be selected at a time. Class `RadioButtonTest` (Fig. 14.20) contains the `main` method that executes this application.

```

1 // Fig. 14.19: RadioButtonFrame.java
2 // Creating radio buttons using ButtonGroup and JRadioButton.
3 import java.awt.FlowLayout;
4 import java.awt.Font;
```

Fig. 14.19 | `JRadioButton`s and `ButtonGroups`. (Part I of 3.)

```
5 import java.awt.event.ItemListener;
6 import java.awt.event.ItemEvent;
7 import javax.swing.JFrame;
8 import javax.swing.JTextField;
9 import javax.swing.JRadioButton;
10 import javax.swing.ButtonGroup;
11
12 public class RadioButtonFrame extends JFrame
13 {
14     private JTextField textField; // used to display font changes
15     private Font plainFont; // font for plain text
16     private Font boldFont; // font for bold text
17     private Font italicFont; // font for italic text
18     private Font boldItalicFont; // font for bold and italic text
19     private JRadioButton plainJRadioButton; // selects plain text
20     private JRadioButton boldJRadioButton; // selects bold text
21     private JRadioButton italicJRadioButton; // selects italic text
22     private JRadioButton boldItalicJRadioButton; // bold and italic
23     private ButtonGroup radioGroup; // button group to hold radio buttons
24
25     // RadioButtonFrame constructor adds JRadioButtons to JFrame
26     public RadioButtonFrame()
27     {
28         super( "RadioButton Test" );
29         setLayout( new FlowLayout() ); // set frame layout
30
31         textField = new JTextField( "Watch the font style change", 25 );
32         add( textField ); // add textField to JFrame
33
34         // create radio buttons
35         plainJRadioButton = new JRadioButton( "Plain", true );
36         boldJRadioButton = new JRadioButton( "Bold", false );
37         italicJRadioButton = new JRadioButton( "Italic", false );
38         boldItalicJRadioButton = new JRadioButton( "Bold/Italic", false );
39         add( plainJRadioButton ); // add plain button to JFrame
40         add( boldJRadioButton ); // add bold button to JFrame
41         add( italicJRadioButton ); // add italic button to JFrame
42         add( boldItalicJRadioButton ); // add bold and italic button
43
44         // create logical relationship between JRadioButtons
45         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 ); // set initial font to plain
57 }
```

Fig. 14.19 | JRadioButtons and ButtonGroups. (Part 2 of 3.)

```

58     // register events for JRadioButtons
59     plainJRadioButton.addItemListener(
60         new RadioButtonHandler( plainFont ) );
61     boldJRadioButton.addItemListener(
62         new RadioButtonHandler( boldFont ) );
63     italicJRadioButton.addItemListener(
64         new RadioButtonHandler( italicFont ) );
65     boldItalicJRadioButton.addItemListener(
66         new RadioButtonHandler( boldItalicFont ) );
67 } // end RadioButtonFrame constructor
68
69 // private inner class to handle radio button events
70 private class RadioButtonHandler implements ItemListener
71 {
72     private Font font; // font associated with this listener
73
74     public RadioButtonHandler( Font f )
75     {
76         font = f; // set the font of this listener
77     } // end constructor RadioButtonHandler
78
79     // handle radio button events
80     public void itemStateChanged( ItemEvent event )
81     {
82         textField.setFont( font ); // set font of textField
83     } // end method itemStateChanged
84 } // end private inner class RadioButtonHandler
85 } // end class RadioButtonFrame

```

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

```

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

```

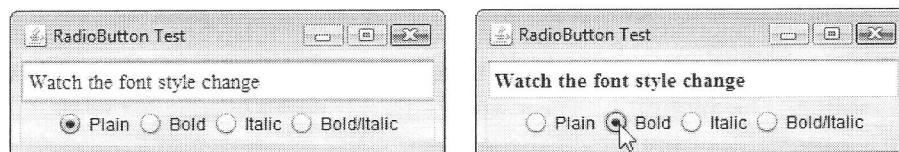


Fig. 14.20 | Test class for RadioButtonFrame. (Part 1 of 2.)

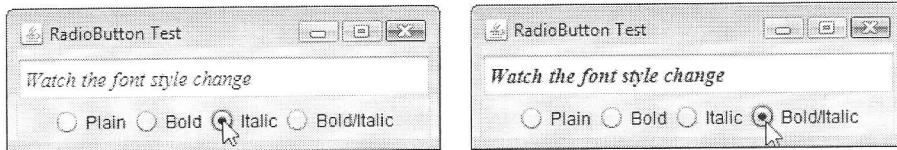


Fig. 14.20 | Test class for RadioButtonFrame. (Part 2 of 2.)

Lines 35–42 in the constructor (Fig. 14.19) create four `JRadioButton` objects and add them to the `JFrame`. Each `JRadioButton` is created with a constructor call like that in line 35. This constructor specifies the label that appears to the right of the `JRadioButton` by default and the initial state of the `JRadioButton`. A true second argument indicates that the `JRadioButton` should appear selected when it's displayed.

Line 45 instantiates `ButtonGroup` object `radioGroup`. This object is the “glue” that forms the logical relationship between the four `JRadioButton` objects and allows only one of the four to be selected at a time. It's possible that no `JRadioButtons` in a `ButtonGroup` are selected, but this can occur *only* if no preselected `JRadioButtons` are added to the `ButtonGroup` and the user has not selected a `JRadioButton` yet. Lines 46–49 use `ButtonGroup` method `add` to associate each of the `JRadioButtons` with `radioGroup`. 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're clicked. Lines 59–66 create four instances of inner class `RadioButtonHandler` (declared at lines 70–84). In this example, each event-listener object is registered to handle the `ItemEvent` generated when the user clicks a particular `JRadioButton`. Notice that each `RadioButtonHandler` object is initialized with a particular `Font` object (created in lines 52–55).

Class `RadioButtonHandler` (line 70–84) implements interface `ItemListener` so it can handle `ItemEvents` generated by the `JRadioButtons`. The constructor stores the `Font` object it receives as an argument in the event-listener object's instance variable `font` (declared at line 72). When the user clicks a `JRadioButton`, `radioGroup` turns off the previously selected `JRadioButton`, and method `itemStateChanged` (line 80–83) sets the font in the `JTextField` to the `Font` stored in the `JRadioButton`'s corresponding event-listener object. Notice that line 82 of inner class `RadioButtonHandler` uses the top-level class's `textField` instance variable to set the font.

14.11 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 (Fig. 14.22). Combo boxes are implemented with class `JComboBox`, which extends class `JComponent`. `JComboBoxes` generate `ItemEvents` just as `JCheckboxes` and `JRadioButtons` do. This example also demonstrates a special form of inner class that's used frequently in event handling. The application (Figs. 14.21–14.22) uses a `JComboBox` to provide a list of four image-file names from which the user can select one image to display. When the user selects a name, the application displays the corresponding image as an `Icon` on a `JLabel`. Class `ComboBoxTest` (Fig. 14.22) contains the `main` method that executes this appli-

cation. The screen captures for this application show the JComboBox list after the selection was made to illustrate which image-file name was selected.

Lines 19–23 (Fig. 14.21) declare and initialize array icons with four new ImageIcon objects. String array names (lines 17–18) contains the names of the four image files that are stored in the same directory as the application.

```
1 // Fig. 14.21: ComboBoxFrame.java
2 // JComboBox that displays a list of image names.
3 import java.awt.FlowLayout;
4 import java.awt.event.ItemListener;
5 import java.awt.event.ItemEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JLabel;
8 import javax.swing.JComboBox;
9 import javax.swing.Icon;
10 import javax.swing.ImageIcon;
11
12 public class ComboBoxFrame extends JFrame
13 {
14     private JComboBox imagesJComboBox; // combobox to hold names of icons
15     private JLabel label; // label to display selected icon
16
17     private static final String[] names =
18         { "bug1.gif", "bug2.gif", "travelbug.gif", "buganim.gif" };
19     private Icon[] icons = {
20         new ImageIcon( getClass().getResource( names[ 0 ] ) ),
21         new ImageIcon( getClass().getResource( names[ 1 ] ) ),
22         new ImageIcon( getClass().getResource( names[ 2 ] ) ),
23         new ImageIcon( getClass().getResource( names[ 3 ] ) ) };
24
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( names ); // set up JComboBox
32         imagesJComboBox.setMaximumRowCount( 3 ); // display three rows
33
34         imagesJComboBox.addItemListener(
35             new ItemListener() // anonymous inner class
36             {
37                 // handle JComboBox event
38                 public void itemStateChanged( ItemEvent event )
39                 {
40                     // determine whether item selected
41                     if ( event.getStateChange() == ItemEvent.SELECTED )
42                         label.setIcon( icons[
43                             imagesJComboBox.getSelectedIndex() ] );
44                 } // end method itemStateChanged
45             } // end anonymous inner class
46         ); // end call to addItemListener
```

Fig. 14.21 | JComboBox that displays a list of image names. (Part 1 of 2.)

```

47
48     add( imagesJComboBox ); // add combobox to JFrame
49     label = new JLabel( icons[ 0 ] ); // display first icon
50     add( label ); // add label to JFrame
51 } // end ComboBoxFrame constructor
52 } // end class ComboBoxFrame

```

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

```

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

```

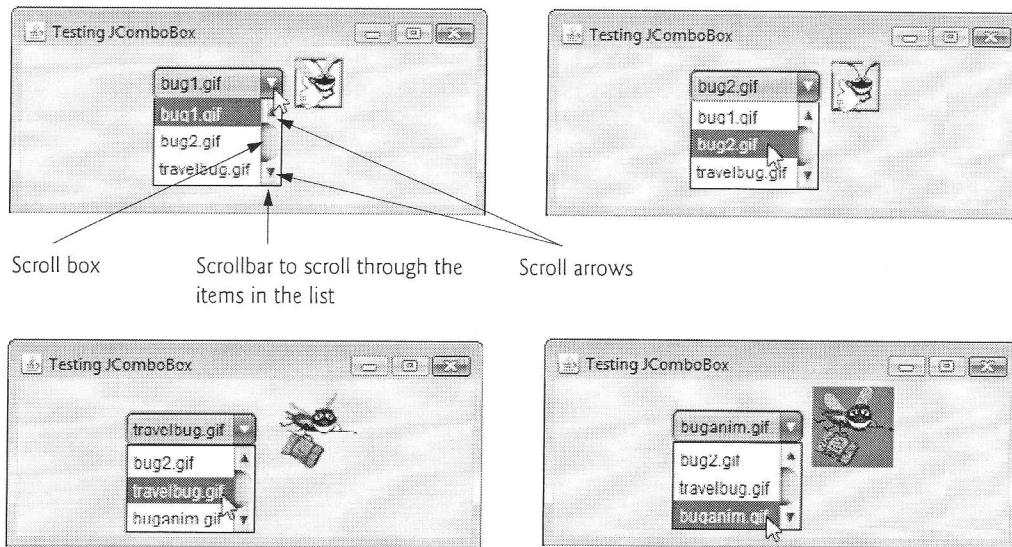


Fig. 14.22 | Testing ComboBoxFrame.

At line 31, the constructor initializes a JComboBox object with the Strings in array names as the elements in the list. Each item in the list has an index. The first item is added at index 0, the next at index 1 and so forth. 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.

Line 32 uses JComboBox method `setMaximumRowCount` to set the maximum number of elements that are displayed when the user clicks the JComboBox. If there are additional items, the JComboBox provides a scrollbar (see the first screen) that allows the user to scroll through all the elements in the list. The user can click the scroll arrows at the top and bottom of the scrollbar to move up and down through the list one element at a time, or else drag the scroll box in the middle of the scrollbar up and down. To drag the scroll box, position the mouse cursor on it, hold the mouse button down and move the mouse. In this example, the drop-down list is too short to drag the scroll box, so you can click the up and down arrows or use your mouse's wheel to scroll through the four items in the list.



Look-and-Feel Observation 14.12

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.

Line 48 attaches the JComboBox to the ComboBoxFrame's FlowLayout (set in line 29). Line 49 creates the JLabel that displays ImageIcons and initializes it with the first ImageIcon in array icons. Line 50 attaches the JLabel to the ComboBoxFrame's FlowLayout.

Using an Anonymous Inner Class for Event Handling

Lines 34–46 are one statement that declares the event listener's class, creates an object of that class and registers it as the listener for imagesJComboBox's ItemEvents. This event-listener object is an instance of an **anonymous inner class**—an inner class that's declared without a name and typically appears inside a method declaration. *As with other inner classes, an anonymous inner class can access its top-level class's members.* However, an anonymous inner class has limited access to the local variables of the method in which it's declared. Since an anonymous inner class has no name, one object of the class must be created at the point where the class is declared (starting at line 35).



Software Engineering Observation 14.3

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.

Lines 34–46 are a call to imagesJComboBox's addItemListener method. The argument to this method must be an object that is an ItemListener (i.e., any object of a class that implements ItemListener). Lines 35–45 are a class-instance creation expression that declares an anonymous inner class and creates one object of that class. A reference to that object is then passed as the argument to addItemListener. The syntax ItemListener() after new begins the declaration of an anonymous inner class that implements interface ItemListener. This is similar to beginning a class declaration with

```
public class MyHandler implements ItemListener
```

The opening left brace at 36 and the closing right brace at line 45 delimit the body of the anonymous inner class. Lines 38–44 declare the ItemListener's itemStateChanged method. When the user makes a selection from imagesJComboBox, this method sets label's Icon. The Icon is selected from array icons by determining the index of the selected item in the JComboBox with method `getSelectedIndex` in line 43. For each item selected from a JComboBox, another item is first deselected—so two ItemEvents occur

when an item is selected. We wish to display only the icon for the item the user just selected. For this reason, line 41 determines whether `ItemEvent` method `getStateChange` returns `ItemEvent.SELECTED`. If so, lines 42–43 set `label`'s icon.



Software Engineering Observation 14.4

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

The syntax shown in lines 35–45 for creating an event handler with an anonymous inner class is similar to the code that would be generated by a Java integrated development environment (IDE). Typically, an IDE enables you to design a GUI visually, then it generates code that implements the GUI. You simply insert statements in the event-handling methods that declare how to handle each event.

14.12 JList

A list displays a series of items from which the user may *select one or more items* (see the output of Fig. 14.24). Lists are created with class `JList`, which directly extends class `JComponent`. Class `JList` supports **single-selection lists** (which allow only one item to be selected at a time) and **multiple-selection lists** (which allow any number of items to be selected). In this section, we discuss single-selection lists.

The application of Figs. 14.23–14.24 creates a `JList` containing 13 color names. When a color name is clicked in the `JList`, a `ListSelectionEvent` occurs and the application changes the background color of the application window to the selected color. Class `ListTest` (Fig. 14.24) contains the `main` method that executes this application.

```

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

```

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

```

23 // ListFrame constructor add JScrollPane containing JList to JFrame
24 public ListFrame()
25 {
26     super( "List Test" );
27     setLayout( new FlowLayout() ); // set frame layout
28
29     colorJList = new JList( colorNames ); // create with colorNames
30     colorJList.setVisibleRowCount( 5 ); // display five rows at once
31
32     // do not allow multiple selections
33     colorJList.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
34
35     // add a JScrollPane containing JList to frame
36     add( new JScrollPane( colorJList ) );
37
38     colorJList.addListSelectionListener(
39         new ListSelectionListener() // anonymous inner class
40     {
41         // handle list selection events
42         public void valueChanged( ListSelectionEvent event )
43         {
44             getContentPane().setBackground(
45                 colors[ colorJList.getSelectedIndex() ] );
46         } // end method valueChanged
47     } // end anonymous inner class
48 ); // end call to addListSelectionListener
49 } // end ListFrame constructor
50 } // end class ListFrame

```

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

```

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

```

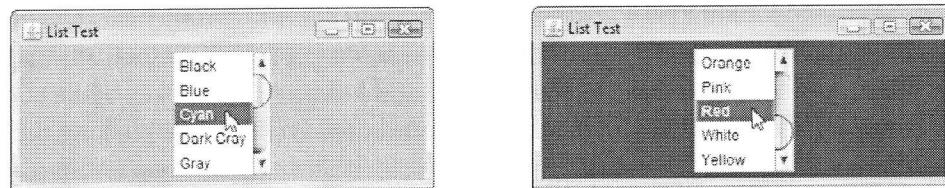


Fig. 14.24 | Test class for ListFrame.

Line 29 (Fig. 14.23) creates `JList` object `colorJList`. The argument to the `JList` constructor is the array of `Objects` (in this case `Strings`) to display in the list. Line 30 uses `JList` method `setVisibleRowCount` to determine the number of items visible in the list.

Line 33 uses `JList` method `setSelectionMode` to specify the list's selection mode. Class `ListSelectionMode` (of package `javax.swing`) declares three constants that specify a `JList`'s selection mode—`SINGLE_SELECTION` (which allows only one item to be selected at a time), `SINGLE_INTERVAL_SELECTION` (for a multiple-selection list that allows selection of several contiguous items) and `MULTIPLE_INTERVAL_SELECTION` (for a multiple-selection list that does not restrict the items that can be selected).

Unlike a `JComboBox`, a `JList` *does not provide a scrollbar* if there are more items in the list than the number of visible rows. In this case, a `JScrollPane` object is used to provide the scrolling capability. Line 36 adds a new instance of class `JScrollPane` to the `JFrame`. The `JScrollPane` constructor receives as its argument the `JComponent` that needs scrolling functionality (in this case, `colorJList`). Notice in the screen captures that a scrollbar created by the `JScrollPane` appears at the right side of the `JList`. By default, the scrollbar appears only when the number of items in the `JList` exceeds the number of visible items.

Lines 38–48 use `JList` method `addListSelectionListener` to register an object that implements `ListSelectionListener` (package `javax.swing.event`) as the listener for the `JList`'s selection events. Once again, we use an instance of an anonymous inner class (lines 39–47) as the listener. In this example, when the user makes a selection from `colorJList`, method `valueChanged` (line 42–46) should change the background color of the `ListFrame` to the selected color. This is accomplished in lines 44–45. Note the use of `JFrame` method `getContentPane` in line 44. Each `JFrame` actually consists of *three layers*—the *background*, the *content pane* and the *glass pane*. The content pane appears in front of the background and is where the GUI components in the `JFrame` are displayed. The glass pane is used to display tool tips and other items that should appear in front of the GUI components on the screen. The content pane completely hides the background of the `JFrame`; thus, 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`). In line 44, we then use that reference to call method `setBackground`, which sets the content pane's background color to an element in the `colors` array. The color is selected from the array by using the selected item's index. `JList` method `getSelectedIndex` returns the selected item's index. As with arrays and `JComboBoxes`, `JList` indexing is zero based.

14.13 Multiple-Selection Lists

A multiple-selection list enables the user to select many items from a `JList` (see the output of Fig. 14.26). A `SINGLE_INTERVAL_SELECTION` list allows selecting a contiguous range of items. To do so, click the first item, then press and hold the *Shift* key while clicking the last item in the range. A `MULTIPLE_INTERVAL_SELECTION` list (the default) allows continuous range selection as described for a `SINGLE_INTERVAL_SELECTION` list. Such a list also 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.

The application of Figs. 14.25–14.26 uses multiple-selection lists to copy items from one `JList` to another. One list is a `MULTIPLE_INTERVAL_SELECTION` list and the other is a

SINGLE_INTERVAL_SELECTION list. When you execute the application, try using the selection techniques described previously to select items in both lists.

```
1 // Fig. 14.25: MultipleSelectionFrame.java
2 // Copying items from one List to another.
3 import java.awt.FlowLayout;
4 import java.awt.event.ActionListener;
5 import java.awt.event.ActionEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JList;
8 import javax.swing.JButton;
9 import javax.swing.JScrollPane;
10 import javax.swing.ListSelectionModel;
11
12 public class MultipleSelectionFrame extends JFrame
13 {
14     private JList colorJList; // list to hold color names
15     private JList copyJList; // list to copy color names into
16     private JButton copyJButton; // button to copy selected names
17     private static final String[] colorNames = { "Black", "Blue", "Cyan",
18         "Dark Gray", "Gray", "Green", "Light Gray", "Magenta", "Orange",
19         "Pink", "Red", "White", "Yellow" };
20
21     // MultipleSelectionFrame constructor
22     public MultipleSelectionFrame()
23     {
24         super( "Multiple Selection Lists" );
25         setLayout( new FlowLayout() ); // set frame layout
26
27         colorJList = new JList( colorNames ); // holds names of all colors
28         colorJList.setVisibleRowCount( 5 ); // show five rows
29         colorJList.setSelectionMode(
30             ListSelectionModel.MULTIPLE_INTERVAL_SELECTION );
31         add( new JScrollPane( colorJList ) ); // add list with scrollpane
32
33         copyJButton = new JButton( "Copy >>>" ); // create copy button
34         copyJButton.addActionListener(
35
36             new ActionListener() // anonymous inner class
37             {
38                 // handle button event
39                 public void actionPerformed( ActionEvent event )
40                 {
41                     // place selected values in copyJList
42                     copyJList.setListData( colorJList.getSelectedValues() );
43                 } // end method actionPerformed
44             } // end anonymous inner class
45         ); // end call to addActionListener
46
47         add( copyJButton ); // add copy button to JFrame
48
49         copyJList = new JList(); // create list to hold copied color names
```

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

```

50      copyJList.setVisibleRowCount( 5 ); // show 5 rows
51      copyJList.setFixedCellWidth( 100 ); // set width
52      copyJList.setFixedCellHeight( 15 ); // set height
53      copyJList.setSelectionMode(
54          ListSelectionModel.SINGLE_INTERVAL_SELECTION );
55      add( new JScrollPane( copyJList ) ); // add list with scrollpane
56  } // end MultipleSelectionFrame constructor
57 } // end class MultipleSelectionFrame

```

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

```

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

```

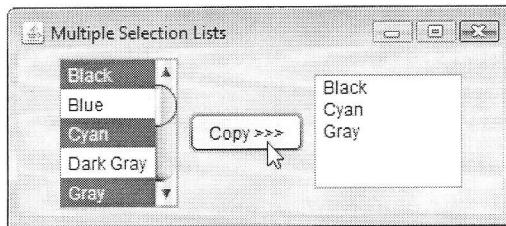


Fig. 14.26 | Test class for MultipleSelectionFrame.

Line 27 of Fig. 14.25 creates JList colorJList and initializes it with the Strings in the array colorNames. Line 28 sets the number of visible rows in colorJList to 5. Lines 29–30 specify that colorJList is a MULTIPLE_INTERVAL_SELECTION list. Line 31 adds a new JScrollPane containing colorJList to the JFrame. Lines 49–55 perform similar tasks for copyJList, which is declared as a SINGLE_INTERVAL_SELECTION list. If a JList does not contain items, it will not display in a FlowLayout. For this reason, lines 51–52 use JList methods `setFixedCellWidth` and `setFixedCellHeight` to set copyJList's width to 100 pixels and the height of each item in the JList to 15 pixels, respectively.

Normally, an event generated by another GUI component (known as an **external event**) specifies when the multiple selections in a JList should be processed. In this example, the user clicks the JButton called copyJButton to trigger the event that copies the selected items in colorJList to copyJList.

Lines 34–45 declare, create and register an `ActionListener` for the `copyJButton`. When the user clicks `copyJButton`, method `actionPerformed` (lines 39–43) uses `JList` method `setListData` to set the items displayed in `copyJList`. Line 42 calls `colorJList`'s method `getSelectedValues`, which returns an array of `Objects` representing the selected items in `colorJList`. In this example, the returned array is passed as the argument to `copyJList`'s `setListData` method.

You might be wondering why `copyJList` can be used in line 42 even though the application does not create the object to which it refers until line 49. Remember that method `actionPerformed` (lines 39–43) does not execute until the user presses the `copyJButton`, which cannot occur until after the constructor completes execution and the application displays the GUI. At that point in the application's execution, `copyJList` is already initialized with a new `JList` object.

14.14 Mouse Event Handling

This section presents the `MouseListener` and `MouseMotionListener` event-listener interfaces for handling mouse events. Mouse events can be processed for any GUI component that derives from `java.awt.Component`. The methods of interfaces `MouseListener` and `MouseMotionListener` are summarized in Figure 14.27. Package `javax.swing.event` contains interface `MouseInputListener`, which extends interfaces `MouseListener` and `MouseMotionListener` to create a single interface containing all the `MouseListener` and `MouseMotionListener` methods. The `MouseListener` and `MouseMotionListener` methods are called when the mouse interacts with a Component if appropriate event-listener objects are registered for that Component.

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

MouseListener and MouseMotionListener interface methods

Methods of interface MouseListener

```
public void mousePressed( MouseEvent event )
```

Called when a mouse button is *pressed* while the mouse cursor is on a component.

```
public void mouseClicked( MouseEvent event )
```

Called when a mouse button is *pressed and released* while the mouse cursor remains stationary on a component. This event is always preceded by a call to `mousePressed`.

```
public void mouseReleased( MouseEvent event )
```

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

Fig. 14.27 | `MouseListener` and `MouseMotionListener` interface methods. (Part 1 of 2.)

MouseListener and MouseMotionListener interface methods

```

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.

Methods of interface MouseMotionListener

public void mouseDragged( MouseEvent event )
    Called when the mouse button is pressed while the mouse cursor is on a component and
    the mouse is moved while the mouse button remains pressed. This event is always preceded
    by a call to mousePressed. All drag events are sent to the component on which the user
    began to drag the mouse.

public void mouseMoved( MouseEvent event )
    Called when the mouse is moved (with no mouse buttons pressed) when the mouse cursor
    is on a component. All move events are sent to the component over which the mouse is
    currently positioned.

```

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



Software Engineering Observation 14.5

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.

Java also provides interface **MouseWheelListener** to enable applications to respond to the *rotation of a mouse wheel*. This interface declares method **mouseWheelMoved**, which receives a **MouseWheelEvent** as its argument. Class **MouseWheelEvent** (a subclass of **MouseEvent**) contains methods that enable the event handler to obtain information about the amount of wheel rotation.

Tracking Mouse Events on a JPanel

The **MouseTracker** application (Figs. 14.28–14.29) demonstrates the **MouseListener** and **MouseMotionListener** interface methods. The event-handler class (lines 36–90) implements both interfaces. You *must* declare all seven methods from these two interfaces when your class implements them both. Each mouse event in this example displays a **String** in the **JLabel** called **statusBar** that is attached to the bottom of the window.

```

1 // Fig. 14.28: MouseTrackerFrame.java
2 // Demonstrating mouse events.
3 import java.awt.Color;
4 import java.awt.BorderLayout;
5 import java.awt.event.MouseListener;
6 import java.awt.event.MouseMotionListener;
7 import java.awt.event.MouseEvent;

```

Fig. 14.28 | Mouse event handling. (Part 1 of 3.)

```
8 import javax.swing.JFrame;
9 import javax.swing.JLabel;
10 import javax.swing.JPanel;
11
12 public class MouseTrackerFrame extends JFrame
13 {
14     private JPanel mousePanel; // panel in which mouse events will occur
15     private JLabel statusBar; // label that displays event information
16
17     // MouseTrackerFrame constructor sets up GUI and
18     // registers mouse event handlers
19     public MouseTrackerFrame()
20     {
21         super( "Demonstrating Mouse Events" );
22
23         mousePanel = new JPanel(); // create panel
24         mousePanel.setBackground( Color.WHITE ); // set background color
25         add( mousePanel, BorderLayout.CENTER ); // add panel to JFrame
26
27         statusBar = new JLabel( "Mouse outside JPanel" );
28         add( statusBar, BorderLayout.SOUTH ); // add label to JFrame
29
30         // create and register listener for mouse and mouse motion events
31         MouseHandler handler = new MouseHandler();
32         mousePanel.addMouseListener( handler );
33         mousePanel.addMouseMotionListener( handler );
34     } // end MouseTrackerFrame constructor
35
36     private class MouseHandler implements MouseListener,
37             MouseMotionListener
38     {
39         // MouseListener event handlers
40         // handle event when mouse released immediately after press
41         public void mouseClicked( MouseEvent event )
42         {
43             statusBar.setText( String.format( "Clicked at [%d, %d]", 
44                 event.getX(), event.getY() ) );
45         } // end method mouseClicked
46
47         // handle event when mouse pressed
48         public void mousePressed( MouseEvent event )
49         {
50             statusBar.setText( String.format( "Pressed at [%d, %d]", 
51                 event.getX(), event.getY() ) );
52         } // end method mousePressed
53
54         // handle event when mouse released
55         public void mouseReleased( MouseEvent event )
56         {
57             statusBar.setText( String.format( "Released at [%d, %d]", 
58                 event.getX(), event.getY() ) );
59         } // end method mouseReleased
60     }
```

Fig. 14.28 | Mouse event handling. (Part 2 of 3.)

```

61      // handle event when mouse enters area
62      public void mouseEntered( MouseEvent event )
63      {
64          statusBar.setText( String.format( "Mouse entered at [%d, %d]", 
65              event.getX(), event.getY() ) );
66          mousePanel.setBackground( Color.GREEN );
67      } // end method mouseEntered
68
69      // handle event when mouse exits area
70      public void mouseExited( MouseEvent event )
71      {
72          statusBar.setText( "Mouse outside JPanel" );
73          mousePanel.setBackground( Color.WHITE );
74      } // end method mouseExited
75
76      // MouseMotionListener event handlers
77      // handle event when user drags mouse with button pressed
78      public void mouseDragged( MouseEvent event )
79      {
80          statusBar.setText( String.format( "Dragged at [%d, %d]", 
81              event.getX(), event.getY() ) );
82      } // end method mouseDragged
83
84      // handle event when user moves mouse
85      public void mouseMoved( MouseEvent event )
86      {
87          statusBar.setText( String.format( "Moved at [%d, %d]", 
88              event.getX(), event.getY() ) );
89      } // end method mouseMoved
90  } // end inner class MouseHandler
91 } // end class MouseTrackerFrame

```

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

Line 23 in Fig. 14.28 creates JPanel mousePanel. This JPanel's mouse events will be tracked by the application. Line 24 sets mousePanel's background color to white. When the user moves the mouse into the mousePanel, the application will change mousePanel's background color to green. When the user moves the mouse out of the mousePanel, the application will change the background color back to white. Line 25 attaches mousePanel to the JFrame. As you learned in Section 14.5, you typically must specify the layout of the GUI components in a JFrame. In that section, we introduced the layout manager FlowLayout. Here we use the default layout of a JFrame's content pane—**BorderLayout**. This layout manager arranges components into five regions: **NORTH**, **SOUTH**, **EAST**, **WEST** and **CENTER**. **NORTH** corresponds to the top of the container. This example uses the **CENTER** and **SOUTH** regions. Line 25 uses a two-argument version of method add to place mousePanel in the **CENTER** region. The BorderLayout automatically sizes the component in the **CENTER** to use all the space in the JFrame that is not occupied by components in the other regions. Section 14.18.2 discusses BorderLayout in more detail.

Lines 27–28 in the constructor declare JLabel statusBar and attach it to the JFrame's **SOUTH** region. This JLabel occupies the width of the JFrame. The region's height is determined by the JLabel.

```

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

```

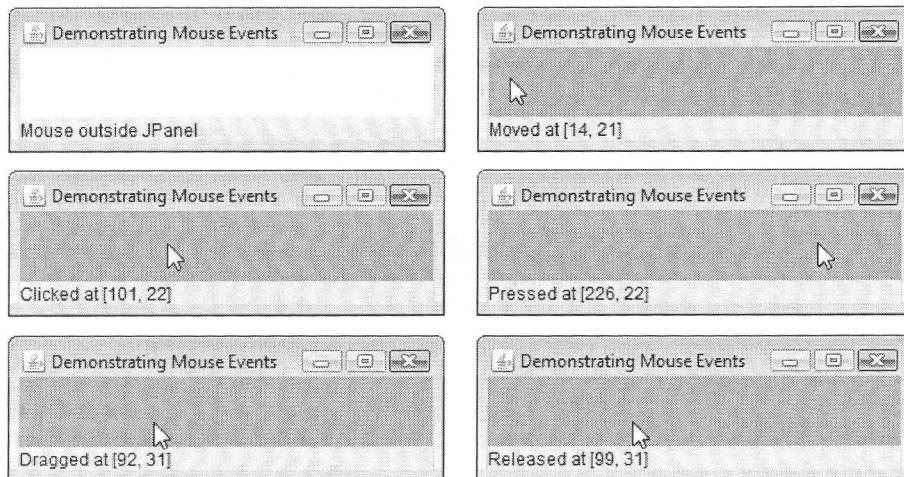


Fig. 14.29 | Test class for MouseTrackerFrame.

Line 31 creates an instance of inner class `MouseHandler` (lines 36–90) called `handler` that responds to mouse events. Lines 32–33 register `handler` as the listener for `mousePanel`'s mouse events. Methods `addMouseListener` and `addMouseMotionListener` are inherited indirectly from class `Component` and can be used to register `MouseListener`s and `MouseMotionListener`s, respectively. A `MouseHandler` object *is a* `MouseListener` and *is a* `MouseMotionListener` because the class implements *both* interfaces. We chose to implement both interfaces here to demonstrate a class that implements more than one interface, but we could have implemented interface `MouseInputListener` instead.]

When the mouse enters and exits `mousePanel`'s area, methods `mouseEntered` (lines 62–67) and `mouseExited` (lines 70–74) are called, respectively. Method `mouseEntered` displays a message in the `statusBar` indicating that the mouse entered the `JPanel` and changes the background color to green. Method `mouseExited` displays a message in the `statusBar` indicating that the mouse is outside the `JPanel` (see the first sample output window) and changes the background color to white.

The other five events display a string in the `statusBar` that includes the event and the coordinates at which it occurred. `MouseEvent` methods `getX` and `getY` return the *x*- and *y*-coordinates, respectively, of the mouse at the time the event occurred.

14.15 Adapter Classes

Many event-listener interfaces, such as `MouseListener` and `MouseMotionListener`, contain multiple methods. It's not always desirable to declare every method in an event-listener interface. For instance, an application may need only the `mouseClicked` handler from `MouseListener` or the `mouseDragged` handler from `MouseMotionListener`. Interface `WindowListener` specifies seven window event-handling methods. For many of the listener interfaces that have multiple methods, packages `java.awt.event` and `javax.swing.event` provide event-listener adapter classes. An adapter class implements an interface and provides a default implementation (with an empty method body) of each method in the interface. Figure 14.30 shows several `java.awt.event` adapter classes and the interfaces they implement. You can extend an adapter class to inherit the default implementation of every method and subsequently override only the method(s) you need for event handling.



Software Engineering Observation 14.6

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

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

Extending `MouseAdapter`

The application of Figs. 14.31–14.32 demonstrates how to determine the number of mouse clicks (i.e., the click count) and how to distinguish between the different mouse buttons. The event listener in this application is an object of inner class `MouseClickHandler` (lines 25–45) that extends `MouseAdapter`, so we can declare just the `mouseClicked` method we need in this example.

```

1 // Fig. 14.31: MouseDetailsFrame.java
2 // Demonstrating mouse clicks and distinguishing between mouse buttons.
3 import java.awt.BorderLayout;
4 import java.awt.event.MouseAdapter;
```

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

```

5 import java.awt.event.MouseEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JLabel;
8
9 public class MouseDetailsFrame extends JFrame
10 {
11     private String details; // String that is displayed in the statusBar
12     private JLabel statusBar; // JLabel that appears at bottom of window
13
14     // constructor sets title bar String and register mouse listener
15     public MouseDetailsFrame()
16     {
17         super( "Mouse clicks and buttons" );
18
19         statusBar = new JLabel( "Click the mouse" );
20         add( statusBar, BorderLayout.SOUTH );
21         addMouseListener( new MouseClickHandler() ); // add handler
22     } // end MouseDetailsFrame constructor
23
24     // inner class to handle mouse events
25     private class MouseClickHandler extends MouseAdapter
26     {
27         // handle mouse-click event and determine which button was pressed
28         public void mouseClicked( MouseEvent event )
29         {
30             int xPos = event.getX(); // get x-position of mouse
31             int yPos = event.getY(); // get y-position of mouse
32
33             details = String.format( "Clicked %d time(s)",
34                 event.getClickCount() );
35
36             if ( event.isMetaDown() ) // right mouse button
37                 details += " with right mouse button";
38             else if ( event.isAltDown() ) // middle mouse button
39                 details += " with center mouse button";
40             else // left mouse button
41                 details += " with left mouse button";
42
43             statusBar.setText( details ); // display message in statusBar
44         } // end method mouseClicked
45     } // end private inner class MouseClickHandler
46 } // end class MouseDetailsFrame

```

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

```

1 // Fig. 14.32: MouseDetails.java
2 // Testing MouseDetailsFrame.
3 import javax.swing.JFrame;
4
5 public class MouseDetails
6 {

```

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

```

7  public static void main( String[] args )
8  {
9      MouseDetailsFrame mouseDetailsFrame = new MouseDetailsFrame();
10     mouseDetailsFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11     mouseDetailsFrame.setSize( 400, 150 ); // set frame size
12     mouseDetailsFrame.setVisible( true ); // display frame
13 } // end main
14 } // end class MouseDetails

```

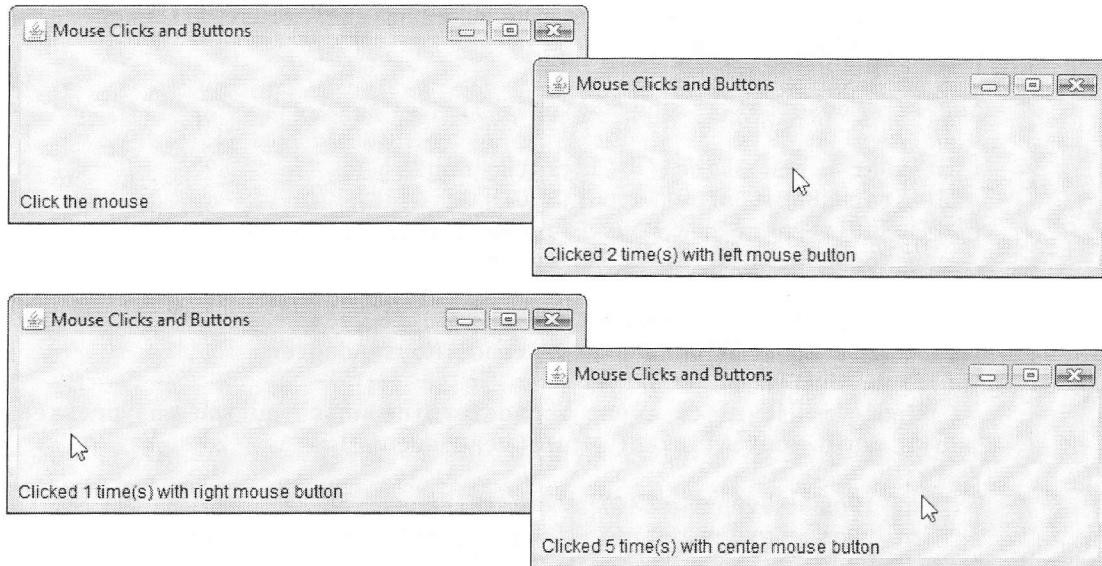


Fig. 14.32 | Test class for `MouseDetailsFrame`. (Part 2 of 2.)



Common Programming Error 14.4

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

A user of a Java application may be on a system with a one-, two- or three-button mouse. Java provides a mechanism to distinguish among mouse buttons. Class `MouseEvent` inherits several methods from class `InputEvent` that can distinguish among mouse buttons on a multibutton mouse or can mimic a multibutton mouse with a combined keystroke and mouse-button click. Figure 14.33 shows the `InputEvent` methods used to distinguish among mouse-button clicks. Java assumes that every mouse contains a left mouse button. Thus, it's simple to test for a left-mouse-button click. However, users with a one- or two-button mouse must use a combination of keystrokes and mouse-button clicks at the same time to simulate the missing buttons on the mouse. 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 true 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 true 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. 14.33 | InputEvent methods that help determine whether the right or center mouse button was clicked.

Line 21 of Fig. 14.31 registers a `MouseListener` for the `MouseDetailsFrame`. The event listener is an object of class `MouseClickHandler`, which extends `MouseAdapter`. This enables us to declare only method `mouseClicked` (lines 28–44). This method first captures the coordinates where the event occurred and stores them in local variables `xPos` and `yPos` (lines 30–31). Lines 33–34 create a `String` called `details` containing the number of consecutive mouse clicks, which is returned by `MouseEvent` method `getClickCount` at line 34. Lines 36–41 use methods `isMetaDown` and `isAltDown` to determine which mouse button the user clicked and append an appropriate `String` to `details` in each case. The resulting `String` is displayed in the `statusBar`. Class `MouseDetails` (Fig. 14.32) contains the `main` method that executes the application. Try clicking with each of your mouse's buttons repeatedly to see the click count increment.

14.16 JPanel Subclass for Drawing with the Mouse

Section 14.14 showed how to track mouse events in a `JPanel`. In this section, we use a `JPanel` as a dedicated drawing area in which the user can draw by dragging the mouse. In addition, this section demonstrates an event listener that extends an adapter class.

Method `paintComponent`

Lightweight Swing components that extend class `JComponent` (such as `JPanel`) contain method `paintComponent`, which is called when a lightweight Swing component is displayed. By overriding this method, you can specify how to draw shapes using Java's graphics capabilities. When customizing a `JPanel` for use as a dedicated drawing area, the subclass should override method `paintComponent` and call the superclass version of `paintComponent` as the first statement in the body of the overridden method to ensure that the component displays correctly. The reason is that subclasses of `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 the program displays the component. When a component is opaque, `paintComponent` clears the component's background before the component is displayed. 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 14.1

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 14.5

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

Defining the Custom Drawing Area

The Painter application of Figs. 14.34–14.35 demonstrates a customized subclass of JPanel that is used to create a dedicated drawing area. The application uses the mouseDragged event handler to create a simple drawing application. The user can draw pictures by dragging the mouse on the JPanel. This example does not use method mouseMoved, so our *event-listener class* (the *anonymous inner class* at lines 22–34) extends MouseMotionAdapter. Since this class already declares both mouseMoved and mouseDragged, we can simply override mouseDragged to provide the event handling this application requires.

```

1 // Fig. 14.34: PaintPanel.java
2 // Using class MouseMotionAdapter.
3 import java.awt.Point;
4 import java.awt.Graphics;
5 import java.awt.event.MouseEvent;
6 import java.awt.event.MouseMotionAdapter;
7 import javax.swing.JPanel;
8
9 public class PaintPanel extends JPanel
10 {
11     private int pointCount = 0; // count number of points
12
13     // array of 10000 java.awt.Point references
14     private Point[] points = new Point[ 1000 ];
15
16     // set up GUI and register mouse event handler
17     public PaintPanel()
18     {
19         // handle frame mouse motion event
20         addMouseMotionListener(
21
22             new MouseMotionAdapter() // anonymous inner class
23             {
24                 // store drag coordinates and repaint
25                 public void mouseDragged( MouseEvent event )
26                 {
27                     if ( pointCount < points.length )
28                 }

```

Fig. 14.34 | Adapter class used to implement event handlers. (Part I of 2.)

```

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

```

Fig. 14.34 | Adapter class used to implement event handlers. (Part 2 of 2.)

Class **PaintPanel** (Fig. 14.34) extends **JPanel** to create the dedicated drawing area. Lines 3–7 import the classes used in class **PaintPanel**. 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.

In this example, we use an array of 10,000 **Points** (line 14) to store the location at which each mouse drag event occurs. As you'll see, method **paintComponent** uses these **Points** to draw. Instance variable **pointCount** (line 11) maintains the total number of **Points** captured from mouse drag events so far.

Lines 20–35 register a **MouseMotionListener** to listen for the **PaintPanel**'s mouse motion events. Lines 22–34 create an object of an anonymous inner class that extends the adapter class **MouseMotionAdapter**. Recall that **MouseMotionAdapter** implements **MouseMotionListener**, so the *anonymous inner class* object is a **MouseMotionListener**. The anonymous inner class inherits default **mouseMoved** and **mouseDragged** implementations, so it already implements all the interface's methods. However, the default methods do nothing when they're called. So, we override method **mouseDragged** at lines 25–33 to capture the coordinates of a mouse drag event and store them as a **Point** object. Line 27 ensures that we store the event's coordinates *only* if there are still empty elements in the array. If so, line 29 invokes the **MouseEvent**'s **getPoint** method to obtain the **Point** where the event occurred and stores it in the array at index **pointCount**. Line 30 increments the **pointCount**, and line 31 calls method **repaint** (inherited indirectly from class **Component**) to indicate that the **PaintPanel** should be refreshed on the screen as soon as possible with a call to the **PaintPanel**'s **paintComponent** method.

Method **paintComponent** (lines 39–46), which receives a **Graphics** parameter, is called automatically any time the **PaintPanel** needs to be displayed on the screen—such as when the GUI is first displayed—or refreshed on the screen—such as when method **repaint** is called or when the GUI component has been hidden by another window on the screen and subsequently becomes visible again.



Look-and-Feel Observation 14.13

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).

Line 41 invokes the superclass version of paintComponent to clear the PaintPanel's background (JPanels are opaque by default). Lines 44–45 draw an oval at the location specified by each Point in the array (up to the pointCount). Graphics method fillOval draws a solid oval. The method's four parameters represent a rectangular area (called the bounding box) in which the oval is displayed. The first two parameters are the upper-left x-coordinate and the upper-left y-coordinate of the rectangular area. The last two coordinates 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. In line 45, the first two arguments are specified by using class Point's two public instance variables—x and y. The loop terminates when pointCount points have been displayed. You'll learn more Graphics features in Chapter 15.



Look-and-Feel Observation 14.14

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.

Using the Custom JPanel in an Application

Class Painter (Fig. 14.35) contains the main method that executes this application. Line 14 creates a PaintPanel object on which the user can drag the mouse to draw. Line 15 attaches the PaintPanel to the JFrame.

```

1 // Fig. 14.35: Painter.java
2 // Testing PaintPanel.
3 import java.awt.BorderLayout;
4 import javax.swing.JFrame;
5 import javax.swing.JPanel;
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(); // create paint panel
15        application.add( paintPanel, BorderLayout.CENTER ); // in 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 );

```

Fig. 14.35 | Test class for PaintPanel. (Part I of 2.)

```

22     application.setSize( 400, 200 ); // set frame size
23     application.setVisible( true ); // display frame
24 } // end main
25 } // end class Painter

```

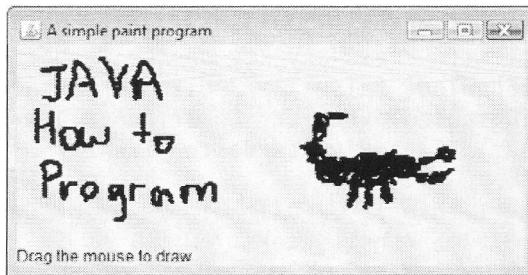


Fig. 14.35 | Test class for PaintPanel. (Part 2 of 2.)

14.17 Key Event Handling

This section presents the `KeyListener` interface for handling key events. Key events are generated when keys on the keyboard are pressed and released. A class that implements `KeyListener` must provide declarations for methods `keyPressed`, `keyReleased` and `keyTyped`, each of which receives a `KeyEvent` as its argument. 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**. (The action keys are any arrow key, *Home*, *End*, *Page Up*, *Page Down*, any function key, etc.) Method `keyReleased` is called when the key is released after any `keyPressed` or `keyTyped` event.

The application of Figs. 14.36–14.37 demonstrates the `KeyListener` methods. Class `KeyDemoFrame` implements the `KeyListener` interface, so all three methods are declared in the application. The constructor (Fig. 14.36, lines 17–28) registers the application to handle its own key events by using method `addKeyListener` at line 27. Method `addKeyListener` is declared in class `Component`, so every subclass of `Component` can notify `KeyListener` objects of key events for that `Component`.

```

1 // Fig. 14.36: KeyDemoFrame.java
2 // Demonstrating keystroke events.
3 import java.awt.Color;
4 import java.awt.event.KeyListener;
5 import java.awt.event.KeyEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JTextArea;
8
9 public class KeyDemoFrame extends JFrame implements KeyListener
10 {
11     private String line1 = ""; // first line of textarea
12     private String line2 = ""; // second line of textarea
13     private String line3 = ""; // third line of textarea
14     private JTextArea textArea; // textarea to display output
15

```

Fig. 14.36 | Key event handling. (Part 1 of 2.)

```
16 // KeyDemoFrame constructor
17 public KeyDemoFrame()
18 {
19     super( "Demonstrating Keystroke Events" );
20
21     textArea = new JTextArea( 10, 15 ); // set up JTextArea
22     textArea.setText( "Press any key on the keyboard..." );
23     textArea.setEnabled( false ); // disable text area
24     textArea.setDisabledTextColor( Color.BLACK ); // set text color
25     add( textArea ); // add text area to JFrame
26
27     addKeyListener( this ); // allow frame to process key events
28 } // end KeyDemoFrame constructor
29
30 // handle press of any key
31 public void keyPressed( KeyEvent event )
32 {
33     line1 = String.format( "Key pressed: %s",
34         KeyEvent.getKeyText( event.getKeyCode() ) ); // show pressed key
35     setLines2and3( event ); // set output lines two and three
36 } // end method keyPressed
37
38 // handle release of any key
39 public void keyReleased( KeyEvent event )
40 {
41     line1 = String.format( "Key released: %s",
42         KeyEvent.getKeyText( event.getKeyCode() ) ); // show released key
43     setLines2and3( event ); // set output lines two and three
44 } // end method keyReleased
45
46 // handle press of an action key
47 public void keyTyped( KeyEvent event )
48 {
49     line1 = String.format( "Key typed: %s", event.getKeyChar() );
50     setLines2and3( event ); // set output lines two and three
51 } // end method keyTyped
52
53 // set second and third lines of output
54 private void setLines2and3( KeyEvent event )
55 {
56     line2 = String.format( "This key is %san action key",
57         ( event.isActionKey() ? "" : "not " ) );
58
59     String temp = KeyEvent.getKeyModifiersText( event.getModifiers() );
60
61     line3 = String.format( "Modifier keys pressed: %s",
62         ( temp.equals( "" ) ? "none" : temp ) ); // output modifiers
63
64     textArea.setText( String.format( "%s\n%s\n%s\n",
65         line1, line2, line3 ) ); // output three lines of text
66 } // end method setLines2and3
67 } // end class KeyDemoFrame
```

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

```

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

```

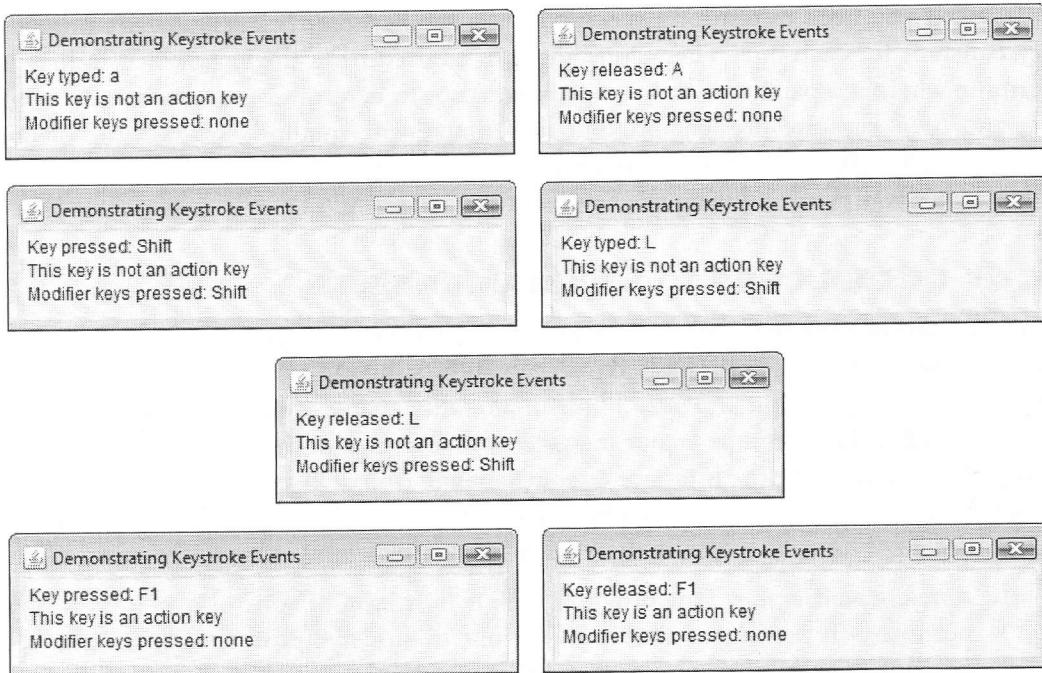


Fig. 14.37 | Test class for KeyDemoFrame.

At line 25, the constructor adds the `JTextArea` `textArea` (where the application's output is displayed) to the `JFrame`. A `JTextArea` is a *multiline area* in which you can display text. (We discuss `JTextAreas` in more detail in Section 14.20.) Notice in the screen captures that `textArea` occupies the *entire window*. This is due to the `JFrame`'s default `BorderLayout` (discussed in Section 14.18.2 and demonstrated in Fig. 14.41). When a single Component is added to a `BorderLayout`, the Component occupies the entire Container. Line 23 disables the `JTextArea` so the user cannot type in it. This causes the text in the `JTextArea` to become gray. Line 24 uses method `setDisabledTextColor` to change the text color in the `JTextArea` to black for readability.

Methods `keyPressed` (lines 31–36) and `keyReleased` (lines 39–44) use `KeyEvent` method `getKeyCode` to get the *virtual key code* of the pressed key. Class `KeyEvent` contains virtual key-code constants that represent every key on the keyboard. These constants

can be compared with `getKeyCode`'s return value to test for individual keys on the keyboard. The value returned by `getKeyCode` is passed to static `KeyEvent` method `getKeyText`, which returns a string containing the name of the key that was pressed. For a complete list of virtual key constants, see the on-line documentation for class `KeyEvent` (package `java.awt.event`). Method `keyTyped` (lines 47–51) uses `KeyEvent` method `getKeyChar` (which returns a `char`) to get the Unicode value of the character typed.

All three event-handling methods finish by calling method `setLines2and3` (lines 54–66) and passing it the `KeyEvent` object. This method uses `KeyEvent` method `isActionKey` (line 57) to determine whether the key in the event was an action key. Also, `InputEvent` method `getModifiers` is called (line 59) to determine whether any modifier keys (such as *Shift*, *Alt* and *Ctrl*) were pressed when the key event occurred. The result of this method is passed to static `KeyEvent` method `getKeyModifiersText`, which produces a string containing the names of the pressed modifier keys.

[*Note:* If you need to test for a specific key on the keyboard, class `KeyEvent` provides a **key constant** for each one. These constants can be used from the key event handlers to determine whether a particular key was pressed. Also, to determine whether the *Alt*, *Ctrl*, *Meta* and *Shift* keys are pressed individually, `InputEvent` methods `isAltDown`, `isControlDown`, `isMetaDown` and `isShiftDown` each return a boolean indicating whether the particular key was pressed during the key event.]

14.18 Introduction to Layout Managers

Layout managers arrange GUI components in a container for presentation purposes. You can use the layout managers for basic layout capabilities instead of determining every GUI component's exact position and size. This functionality enables you to concentrate on the basic look-and-feel and lets the layout managers process most of the layout details. All layout managers implement the interface `LayoutManager` (in package `java.awt`). Class `Container`'s `setLayout` method takes an object that implements the `LayoutManager` interface as an argument. There are basically three ways for you to arrange components in a GUI:

1. *Absolute positioning:* This provides the greatest level of control over a GUI's appearance. By setting a `Container`'s layout to `null`, you can 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`. If you do this, you also must specify each GUI component's size. Programming a GUI with absolute positioning can be tedious, unless you have an integrated development environment (IDE) that can generate the code for you.
2. *Layout managers:* Using layout managers to position elements can be simpler and faster than creating a GUI with absolute positioning, but you lose some control over the size and the precise positioning of GUI components.
3. *Visual programming in an IDE:* IDEs provide tools that make it easy to create GUIs. Each IDE typically provides a GUI design tool that 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. The IDE generates the Java code that creates the GUI. In addition, you can typically add event-handling code for a particular component by double-clicking the component. Some design tools also allow you to use the layout managers described in this chapter and in Chapter 25.

**Look-and-Feel Observation 14.15**

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 14.16**

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 in the given container and take care that, on resize events, all components are repositioned as necessary. A component's resize events can be processed by a Component-Listener.

Figure 14.38 summarizes the layout managers presented in this chapter. Others are discussed in Chapter 25, and the powerful GroupLayout layout manager is discussed in Appendix I.

Layout manager	Description
FlowLayout	Default for <code>javax.swing.JPanel</code> . Places components sequentially (left to right) in the order they were added. It's also possible to specify the order of the components by using the <code>Container</code> method <code>add</code> , which takes a <code>Component</code> and an integer index <code>position</code> 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. 14.38 | Layout managers.

14.18.1 FlowLayout

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

The application of Figs. 14.39–14.40 creates three JButton objects and adds them to the application, using a FlowLayout layout manager. The components are center aligned by default. When the user clicks **Left**, the alignment for the layout manager is changed to a left-aligned FlowLayout. When the user clicks **Right**, the alignment for the layout manager is changed to a right-aligned FlowLayout. When the user clicks **Center**, the alignment for the layout manager is changed to a center-aligned FlowLayout. Each button has its own event handler that's declared with an anonymous inner class that implements ActionListener. The sample output windows show each of the FlowLayout alignments. Also, the last sample output window shows the centered alignment after the window has been resized to a smaller width. Notice that the button **Right** flows onto a new line.

As seen previously, a container's layout is set with method `setLayout` of class `Container`. Line 25 sets the layout manager to the `FlowLayout` declared at line 23. Normally, the layout is set before any GUI components are added to a container.

**Look-and-Feel Observation 14.17**

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

```

1 // Fig. 14.39: FlowLayoutFrame.java
2 // Demonstrating FlowLayout alignments.
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 JButton leftJButton; // button to set alignment left
13     private JButton centerJButton; // button to set alignment center
14     private JButton rightJButton; // button to set alignment right
15     private FlowLayout layout; // layout object
16     private Container container; // container to set layout
17
18     // set up GUI and register button listeners
19     public FlowLayoutFrame()
20     {
21         super( "FlowLayout Demo" );
22
23         layout = new FlowLayout(); // create FlowLayout
24         container = getContentPane(); // get container to layout
25         setLayout( layout ); // set frame layout
26
27         // set up leftJButton and register listener
28         leftJButton = new JButton( "Left" ); // create Left button
29         add( leftJButton ); // add Left button to frame
30         leftJButton.addActionListener(
31
32             new ActionListener() // anonymous inner class
33             {
34                 // process leftJButton event
35                 public void actionPerformed( ActionEvent event )
36                 {
37                     layout.setAlignment( FlowLayout.LEFT );
38
39                     // realign attached components
40                     layout.layoutContainer( container );
41                 } // end method actionPerformed
42             } // end anonymous inner class
43         ); // end call to addActionListener
44
45         // set up centerJButton and register listener
46         centerJButton = new JButton( "Center" ); // create Center button
47         add( centerJButton ); // add Center button to frame

```

Fig. 14.39 | FlowLayout allows components to flow over multiple lines. (Part I of 2.)

```

48     centerJButton.addActionListener(
49
50         new ActionListener() // anonymous inner class
51     {
52         // process centerJButton event
53         public void actionPerformed( ActionEvent event )
54         {
55             layout.setAlignment( FlowLayout.CENTER );
56
57             // realign attached components
58             layout.layoutContainer( container );
59         } // end method actionPerformed
60     } // end anonymous inner class
61 ); // end call to addActionListener
62
63 // set up rightJButton and register listener
64 rightJButton = new JButton( "Right" ); // create Right button
65 add( rightJButton ); // add Right button to frame
66 rightJButton.addActionListener(
67
68     new ActionListener() // anonymous inner class
69     {
70         // process rightJButton event
71         public void actionPerformed( ActionEvent event )
72         {
73             layout.setAlignment( FlowLayout.RIGHT );
74
75             // realign attached components
76             layout.layoutContainer( container );
77         } // end method actionPerformed
78     } // end anonymous inner class
79 ); // end call to addActionListener
80 } // end FlowLayoutFrame constructor
81 } // end class FlowLayoutFrame

```

Fig. 14.39 | *FlowLayout allows components to flow over multiple lines. (Part 2 of 2.)*

```

1 // Fig. 14.40: 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 ); // set frame size
12        flowLayoutFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class FlowLayoutDemo

```

Fig. 14.40 | *Test class for FlowLayoutFrame. (Part 1 of 2.)*

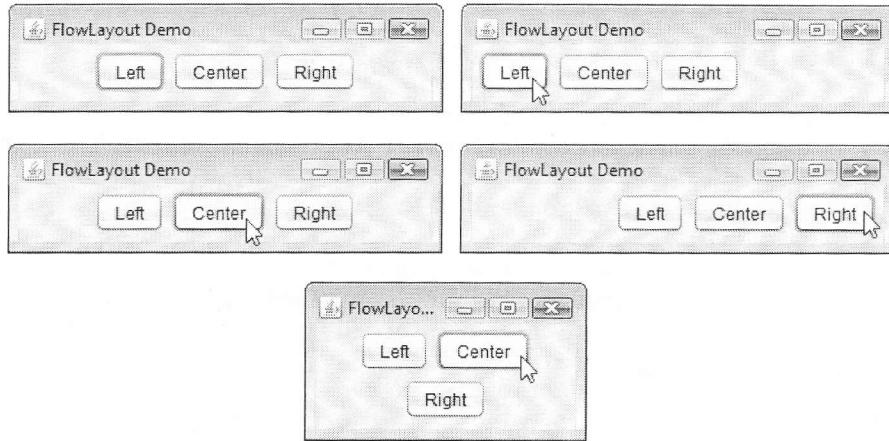


Fig. 14.40 | Test class for `FlowLayoutFrame`. (Part 2 of 2.)

Each button's event handler is specified with a separate anonymous inner-class object (Fig. 14.39, lines 30–43, 48–61 and 66–79, respectively), and method `actionPerformed` in each case executes two statements. For example, line 37 in the event handler for `leftJButton` uses `FlowLayout` method `setAlignment` to change the alignment for the `FlowLayout` to a left-aligned (`FlowLayout.LEFT`) `FlowLayout`. Line 40 uses `LayoutManager` interface method `layoutContainer` (which is inherited by all layout managers) to specify that the `JFrame` should be rearranged based on the adjusted layout. According to which button was clicked, the `actionPerformed` method for each button sets the `FlowLayout`'s alignment to `FlowLayout.LEFT` (line 37), `FlowLayout.CENTER` (line 55) or `FlowLayout.RIGHT` (line 73).

14.18.2 BorderLayout

The `BorderLayout` layout manager (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. Class `BorderLayout` extends `Object` and implements interface `LayoutManager2` (a subinterface of `LayoutManager` that adds several methods for enhanced layout processing).

A `BorderLayout` limits a `Container` to containing at most five components—one in each region. The component placed in each region can be a container to which other components are attached. The components placed in the NORTH and SOUTH regions extend horizontally to the sides of the container and are as tall as the components placed in those regions. The EAST and WEST regions expand vertically between the NORTH and SOUTH regions and are as wide as the components placed in those regions. The component placed in the CENTER region expands to fill all remaining space in the layout (which is the reason the `JTextArea` in Fig. 14.37 occupies the entire window). If all five regions are occupied, the entire container's space is covered by GUI components. If the NORTH or SOUTH region is not occupied, the GUI components in the EAST, CENTER and WEST regions expand vertically to fill the remaining space. If the EAST or WEST region is not occupied, the GUI component in the CENTER region expands horizontally to fill the remaining space. If the CENTER region is not occupied, the area is left empty—the other GUI components do *not* expand

to fill the remaining space. The application of Figs. 14.41–14.42 demonstrates the BorderLayout layout manager by using five JButtons.

```
1 // Fig. 14.41: BorderLayoutFrame.java
2 // Demonstrating BorderLayout.
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 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 BorderLayout layout; // borderlayout object
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 ); // set frame layout
23         buttons = new JButton[ names.length ]; // set size of array
24
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         } // end for
31
32         add( buttons[ 0 ], BorderLayout.NORTH ); // add button to north
33         add( buttons[ 1 ], BorderLayout.SOUTH ); // add button to south
34         add( buttons[ 2 ], BorderLayout.EAST ); // add button to east
35         add( buttons[ 3 ], BorderLayout.WEST ); // add button to west
36         add( buttons[ 4 ], BorderLayout.CENTER ); // add button to center
37     } // end BorderLayoutFrame constructor
38
39     // handle button events
40     public void actionPerformed( ActionEvent event )
41     {
42         // check event source and lay out content pane correspondingly
43         for ( JButton button : buttons )
44         {
45             if ( event.getSource() == button )
46                 button.setVisible( false ); // hide button clicked
47             else
48                 button.setVisible( true ); // show other buttons
49         } // end for
```

Fig. 14.41 | BorderLayout containing five buttons. (Part I of 2.)

```

50
51     layout.setLayoutContainer( getContentPane() ); // lay out content pane
52 } // end method actionPerformed
53 } // end class BorderLayoutFrame

```

Fig. 14.41 | BorderLayout containing five buttons. (Part 2 of 2.)

```

1 // Fig. 14.42: 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 ); // set frame size
12        borderLayoutFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class BorderLayoutDemo

```

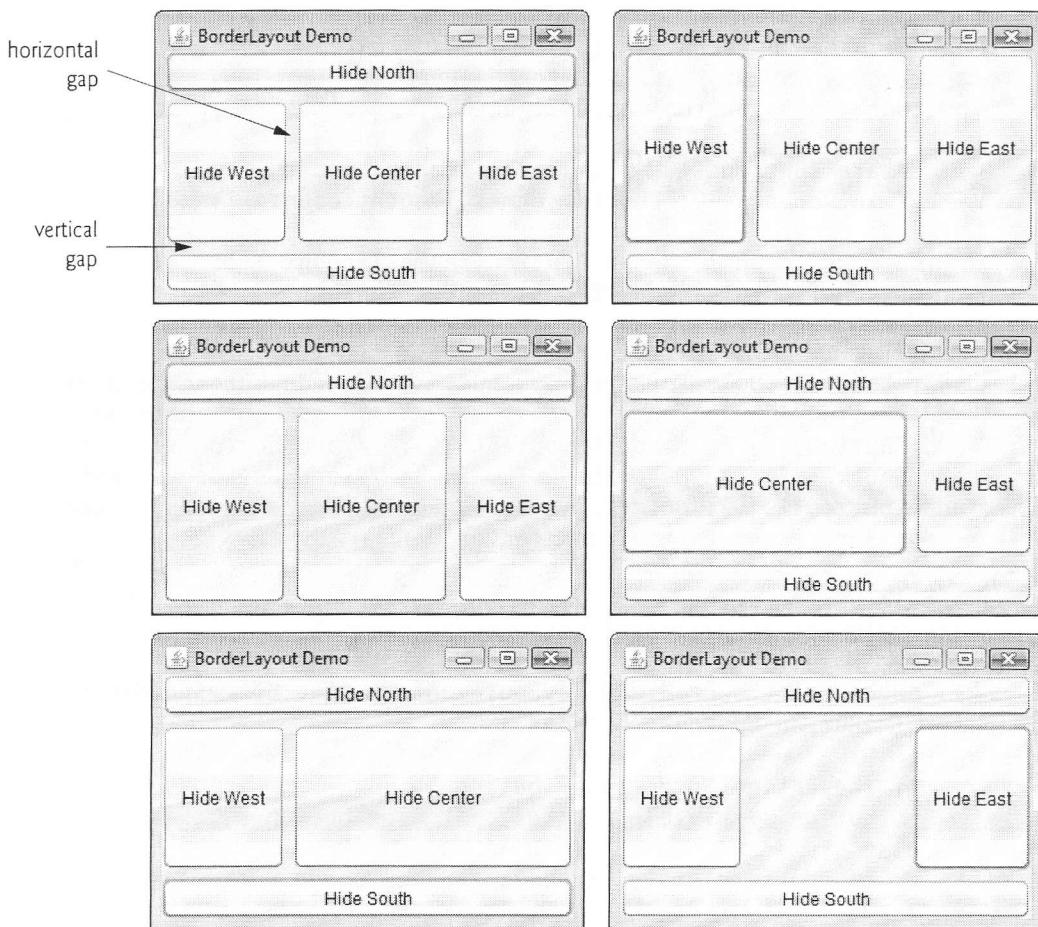


Fig. 14.42 | Test class for BorderLayoutFrame.

Line 21 of Fig. 14.41 creates a `BorderLayout`. The 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. Line 22 uses method `setLayout` to set the content pane's layout to `layout`.

We add Components to a `BorderLayout` with another version of `Container` method `add` that takes two arguments—the Component to add and the region in which the Component should appear. For example, line 32 specifies that `buttons[0]` should appear in the `NORTH` region. The components can be added in any order, but only one component should be added to each region.



Look-and-Feel Observation 14.18

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 14.6

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

Class `BorderLayoutFrame` implements `ActionListener` directly in this example, so the `BorderLayoutFrame` will handle the events of the `JButtons`. For this reason, line 29 passes the `this` reference to the `addActionListener` method of each `JButton`. When the user clicks a particular `JButton` in the layout, method `actionPerformed` (lines 40–52) executes. The enhanced for statement at lines 43–49 uses an `if...else` to hide the particular `JButton` that generated the event. Method `setVisible` (inherited into `JButton` from class `Component`) is called with a `false` argument (line 46) to hide the `JButton`. If the current `JButton` in the array is not the one that generated the event, method `setVisible` is called with a `true` argument (line 48) to ensure that the `JButton` is displayed on the screen. Line 51 uses Layout-Manager method `layoutContainer` to recalculate the layout of the content pane. Notice in the screen captures of Fig. 14.42 that certain regions in the `BorderLayout` change shape as `JButtons` are hidden and displayed in other regions. Try resizing the application window to see how the various regions resize based on the window's width and height. *For more complex layouts, group components in `JPanels`, each with a separate layout manager.* Place the `JPanels` on the `JFrame` using either the default `BorderLayout` or some other layout.

14.18.3 GridLayout

The **GridLayout** layout manager divides the container into a *grid* so that components can be placed in *rows* and *columns*. Class `GridLayout` inherits directly from class `Object` and implements interface `LayoutManager`. Every Component in a `GridLayout` has the *same* width and height. Components are added to a `GridLayout` 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. The application of Figs. 14.43–14.44 demonstrates the `GridLayout` layout manager by using six `JButtons`.

Lines 24–25 create two `GridLayout` objects. The `GridLayout` constructor used at line 24 specifies a `GridLayout` with 2 rows, 3 columns, 5 pixels of horizontal-gap space between Components in the grid and 5 pixels of vertical-gap space between Components in the grid.

```
1 // Fig. 14.43: GridLayoutFrame.java
2 // Demonstrating GridLayout.
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 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 Container container; // frame container
17     private GridLayout gridLayout1; // first gridlayout
18     private GridLayout gridLayout2; // second gridlayout
19
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(); // get content pane
27         setLayout( gridLayout1 ); // set JFrame layout
28         buttons = new JButton[ names.length ]; // create array of JButtons
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         } // end for
36     } // end GridLayoutFrame constructor
37
38     // handle button events by toggling between layouts
39     public void actionPerformed( ActionEvent event )
40     {
41         if ( toggle )
42             container.setLayout( gridLayout2 ); // set layout to second
43         else
44             container.setLayout( gridLayout1 ); // set layout to first
45
46         toggle = !toggle; // set toggle to opposite value
47         container.validate(); // re-lay out container
48     } // end method actionPerformed
49 } // end class GridLayoutFrame
```

Fig. 14.43 | GridLayout containing six buttons.

The GridLayout constructor used at line 25 specifies a GridLayout with 3 rows and 2 columns that uses the default gap space (1 pixel).

```

1 // Fig. 14.44: 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 ); // set frame size
12        gridLayoutFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class GridLayoutDemo

```

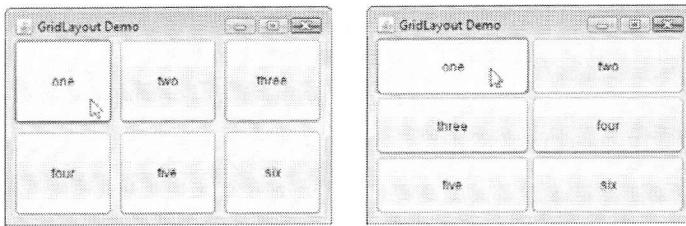


Fig. 14.44 | Test class for GridLayoutFrame.

The JButton objects in this example initially are arranged using gridLayout1 (set for the content pane at line 27 with method setLayout). The first component is added to the first column of the first row. The next component is added to the second column of the first row, and so on. When a JButton is pressed, method actionPerformed (lines 39–48) is called. Every call to actionPerformed toggles the layout between gridLayout2 and gridLayout1, using boolean variable toggle to determine the next layout to set.

Line 47 shows another way to reformat a container for which the layout has changed. Container method validate recomputes the container's layout based on the current layout manager for the Container and the current set of displayed GUI components.

14.19 Using Panels to Manage More Complex Layouts

Complex GUIs (like Fig. 14.1) require that each component be placed in an exact location. They 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. Thus, every JPanel may have components, including other panels, attached to it with Container method add. The application of Figs. 14.45–14.46 demonstrates how a JPanel can be used to create a more complex layout in which several JButton objects are placed in the SOUTH region of a BorderLayout.

After JPanel buttonJPanel is declared (line 11) and created (line 19), line 20 sets buttonJPanel's layout to a GridLayout of one row and five columns (there are five JButton objects in array buttons). Lines 23–27 add the JButton objects in the array to the JPanel. Line 26 adds the buttons directly to the JPanel—class JPanel does not have a content pane, unlike a JFrame. Line 29 uses the JFrame's default BorderLayout to add buttonJPanel to the SOUTH region. The SOUTH region is as tall as the buttons on buttonJPanel. A JPanel is sized

to the components it contains. As more components are added, the JPanel grows (according to the restrictions of its layout manager) to accommodate the components. Resize the window to see how the layout manager affects the size of the JButtons.

```

1 // Fig. 14.45: 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 JPanel buttonJPanel; // panel to hold buttons
12     private JButton[] buttons; // array of buttons
13
14     // no-argument constructor
15     public PanelFrame()
16     {
17         super( "Panel Demo" );
18         buttons = new JButton[ 5 ]; // create buttons array
19         buttonJPanel = new JPanel(); // set up panel
20         buttonJPanel.setLayout( new GridLayout( 1, buttons.length ) );
21
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         } // end for
28
29         add( buttonJPanel, BorderLayout.SOUTH ); // add panel to JFrame
30     } // end PanelFrame constructor
31 } // end class PanelFrame

```

Fig. 14.45 | JPanel with five JButtons in a GridLayout attached to the SOUTH region of a BorderLayout.

```

1 // Fig. 14.46: 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 ); // set frame size

```

Fig. 14.46 | Test class for PanelFrame. (Part I of 2.)

```

12     panelFrame.setVisible( true ); // display frame
13 } // end main
14 } // end class PanelDemo

```

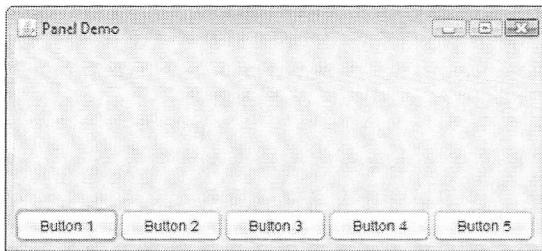


Fig. 14.46 | Test class for PanelFrame. (Part 2 of 2.)

14.20 JTextArea

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

The application in Figs. 14.47–14.48 demonstrates `JTextAreas`. One `JTextArea` displays text that the user can select. The other is uneditable and is used to display the text the user selected in the first `JTextArea`. Unlike `JTextFields`, `JTextAreas` do not have action events—when you press *Enter* while typing in a `JTextArea`, the cursor simply moves to the next line. As with multiple-selection `JLists` (Section 14.13), an external event from another GUI component indicates when to process the text in a `JTextArea`. For example, when typing an e-mail message, you normally click a `Send` button to send the text of the message to the recipient. Similarly, when editing a document in a word processor, you normally save the file by selecting a `Save` or `Save As...` menu item. In this program, the button `Copy >>>` generates the external event that copies the selected text in the left `JTextArea` and displays it in the right `JTextArea`.

```

1 // Fig. 14.47: TextAreaFrame.java
2 // Copying selected text from one textarea 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 JTextArea textArea1; // displays demo string
14     private JTextArea textArea2; // highlighted text is copied here
15     private JButton copyJButton; // initiates copying of text
16

```

Fig. 14.47 | Copying selected text from one `JTextArea` to another. (Part I of 2.)

```

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";
25
26     textArea1 = new JTextArea( demo, 10, 15 ); // create textArea1
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
33         new ActionListener() // anonymous inner class
34     {
35         // set text in textArea2 to selected text from textArea1
36         public void actionPerformed( ActionEvent event )
37         {
38             textArea2.setText( textArea1.getSelectedText() );
39         } // end method actionPerformed
40     } // end anonymous inner class
41 ); // end call to addActionListener
42
43     textArea2 = new JTextArea( 10, 15 ); // create second textarea
44     textArea2.setEditable( false ); // disable editing
45     box.add( new JScrollPane( textArea2 ) ); // add scrollpane
46
47     add( box ); // add box to frame
48 } // end TextAreaFrame constructor
49 } // end class TextAreaFrame

```

Fig. 14.47 | Copying selected text from one JTextArea to another. (Part 2 of 2.)

```

1 // Fig. 14.48: TextAreaDemo.java
2 // Copying selected text from one textarea to another.
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 ); // set frame size
12        textAreaFrame.setVisible( true ); // display frame
13    } // end main
14 } // end class TextAreaDemo

```

Fig. 14.48 | Test class for TextAreaFrame. (Part 1 of 2.)

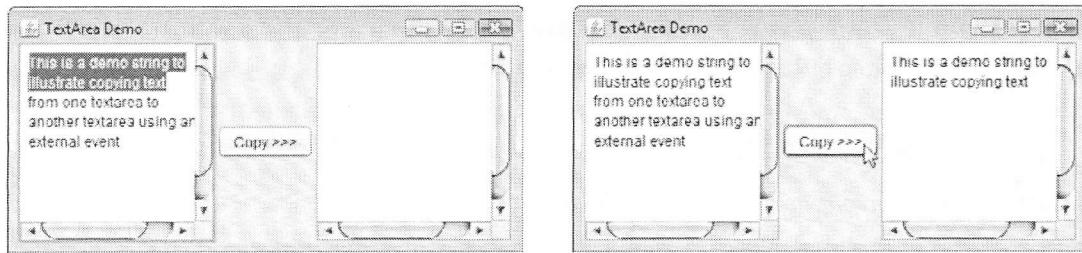


Fig. 14.48 | Test class for TextAreaFrame. (Part 2 of 2.)

In the constructor (lines 18–48), line 21 creates a `Box` container (package `javax.swing`) to organize the GUI components. `Box` is a subclass of `Container` that uses a `BoxLayout` layout manager (discussed in detail in Section 25.9) to arrange the GUI components either horizontally or vertically. `Box`'s static method `createHorizontalBox` creates a `Box` that arranges components from left to right in the order that they're attached.

Lines 26 and 43 create `JTextAreas` `textArea1` and `textArea2`. Line 26 uses `JTextArea`'s three-argument constructor, which takes a `String` representing the initial text and two `ints` specifying that the `JTextArea` has 10 rows and 15 columns. Line 43 uses `JTextArea`'s two-argument constructor, specifying that the `JTextArea` has 10 rows and 15 columns. Line 26 specifies that `demo` should be displayed as the default `JTextArea` content. A `JTextArea` does not provide scrollbars if it cannot display its complete contents. So, line 27 creates a `JScrollPane` object, initializes it with `textArea1` and attaches it to container `box`. By default, horizontal and vertical scrollbars appear as necessary in a `JScrollPane`.

Lines 29–41 create `JButton` object `copyJButton` with the label "Copy >>", add `copyJButton` to container `box` and register the event handler for `copyJButton`'s `ActionEvent`. This button provides the external event that determines when the program should copy the selected text in `textArea1` to `textArea2`. When the user clicks `copyJButton`, line 38 in `actionPerformed` indicates that method `getSelectedText` (inherited into `JTextArea` from `JTextComponent`) should return the selected text from `textArea1`. The user selects text by dragging the mouse over the desired text to highlight it. Method `setText` changes the text in `textArea2` to the string returned by `getSelectedText`.

Lines 43–45 create `textArea2`, set its `editable` property to `false` and add it to container `box`. Line 47 adds `box` to the `JFrame`. Recall from Section 14.18 that the default layout of a `JFrame` is a `BorderLayout` and that the `add` method by default attaches its argument to the `CENTER` of the `BorderLayout`.

When text reaches the right edge of a `JTextArea` the text can wrap to the next line. This is referred to as **line wrapping**. By default, `JTextArea` does *not* wrap lines.



Look-and-Feel Observation 14.19

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

JScrollPane Scrollbar Policies

This example uses a `JScrollPane` to provide scrolling for a `JTextArea`. By default, `JScrollPane` displays scrollbars only if they're required. You can set the horizontal and vertical scrollbar policies of a `JScrollPane` when it's constructed. If a program has a ref-

erence to a `JScrollPane`, the program can use `JScrollPane` methods `setHorizontalScrollBarPolicy` and `setVerticalScrollBarPolicy` to change the scrollbar policies at any time. 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 the horizontal scrollbar policy is set to `JScrollPane.HORIZONTAL_SCROLLBAR_NEVER`, a `JTextArea` attached to the `JScrollPane` will automatically wrap lines.

14.21 Wrap-Up

In this chapter, you learned many GUI components and how to implement event handling. You also learned about nested classes, inner classes and anonymous inner classes. You saw the special relationship between an inner-class object and an object of its top-level class. You learned how to use `JOptionPane` dialogs to obtain text input from the user and how to display messages to the user. You also learned how to create applications that execute in their own windows. We discussed class `JFrame` and components that enable a user to interact with an application. We also showed you how to display text and images to the user. You learned how to customize `JPanels` to create custom drawing areas, which you'll use extensively in the next chapter. You saw how to organize components on a window using layout managers and how to creating more complex GUIs by using `JPanels` to organize components. Finally, you learned about the `JTextArea` component in which a user can enter text and an application can display text. In Chapter 25, you'll learn about more advanced GUI components, such as sliders, menus and more complex layout managers. In the next chapter, you'll learn how to add graphics to your GUI application. Graphics allow you to draw shapes and text with colors and styles.

Summary

Section 14.1 Introduction

- A graphical user interface (GUI; p. 550) presents a user-friendly mechanism for interacting with an application. A GUI gives an application a distinctive look and feel (p. 555).
- 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.
- GUIs are built from GUI components (p. 550)—sometimes called controls or widgets.

Section 14.2 Java's New Nimbus Look-and-Feel

- As of Java SE 6 update 10, Java comes bundled with a new, elegant, cross-platform look-and-feel known as Nimbus (p. 551).

- To set Nimbus as the default for all Java applications, create a `swing.properties` text file in the `lib` folder of your JDK and JRE installation folders. Place the following line of code in the file:

```
swing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

- 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.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

Section 14.3 Simple GUI-Based Input/Output with JOptionPane

- Most applications use windows or dialog boxes (p. 552) to interact with the user.
- Class `JOptionPane` (p. 552) of package `javax.swing` (p. 550) provides prebuilt dialog boxes for both input and output. `JOptionPane` static method `showInputDialog` (p. 553) displays an input dialog (p. 552).
- A prompt typically uses sentence-style capitalization—capitalizing only the first letter of the first word in the text unless the word is a proper noun.
- An input dialog can input only `String`s. This is typical of most GUI components.
- `JOptionPane` static method `showMessageDialog` (p. 554) displays a message dialog (p. 552).

Section 14.4 Overview of Swing Components

- Most Swing GUI components (p. 550) are located in package `javax.swing`.
- 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.
- Lightweight Swing components are not tied to actual GUI components supported by the underlying platform on which an application executes.
- Several Swing components are heavyweight components (p. 556) that require direct interaction with the local windowing system (p. 556), which may restrict their appearance and functionality.
- Class `Component` (p. 556; package `java.awt`) declares many of the attributes and behaviors common to the GUI components in packages `java.awt` (p. 555) and `javax.swing`.
- Class `Container` (p. 556; package `java.awt`) is a subclass of `Component`. Components are attached to Containers so the Components can be organized and displayed on the screen.
- Class `JComponent` (p. 556) of package `javax.swing` is a subclass of `Container`. `JComponent` is the superclass of all lightweight Swing components and declares their common attributes and behaviors.
- Some common `JComponent` features include a pluggable look-and-feel (p. 556), shortcut keys called mnemonics (p. 556), tool tips (p. 556), support for assistive technologies and support for user-interface localization (p. 556).

Section 14.5 Displaying Text and Images in a Window

- Class `JFrame` provides the basic attributes and behaviors of a window.
- A `JLabel` (p. 557) displays read-only text, an image, or both text and an image. Text in a `JLabel` normally uses sentence-style capitalization.
- Each GUI component must be attached to a container, such as a window created with a `JFrame` (p. 559).
- Many IDEs provide GUI design tools (p. 604) in which you can specify the exact size and location of a component by using the mouse; then the IDE will generate the GUI code for you.
- `JComponent` method `setToolTipText` (p. 559) specifies the tool tip that's displayed when the user positions the mouse cursor over a lightweight component (p. 556).

- Container method `add` attaches a GUI component to a Container.
- Class `ImageIcon` (p. 560) supports several image formats, including GIF, PNG and JPEG.
- Method `getClass` of class `Object` (p. 560) retrieves a reference to the `Class` object that represents the class declaration for the object on which the method is called.
- `Class` method `getResource` (p. 560) returns the location of its argument as a URL. The method `getResource` uses the `Class` object's class loader to determine the location of the resource.
- The horizontal and vertical alignments of a `JLabel` can be set with methods `setHorizontalAlignment` (p. 560) and `setVerticalAlignment` (p. 560), respectively.
- `JLabel` methods `setText` (p. 560) and `getText` (p. 560) set and get the text displayed on a label.
- `JLabel` methods `setIcon` (p. 560) and `getIcon` (p. 560) set and get the `Icon` (p. 560) on a label.
- `JLabel` methods `setHorizontalTextPosition` (p. 560) and `setVerticalTextPosition` (p. 560) specify the text position in the label.
- `JFrame` method `setDefaultCloseOperation` (p. 561) with constant `JFrame.EXIT_ON_CLOSE` as the argument indicates that the program should terminate when the window is closed by the user.
- Component method `setSize` (p. 561) specifies the width and height of a component.
- Component method `setVisible` (p. 561) with the argument `true` displays a `JFrame` on the screen.

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

- GUIs are event driven—when the user interacts with a GUI component, events (p. 561) drive the program to perform tasks.
- An event handler (p. 561) performs a task in response to an event.
- Class `JTextField` (p. 561) extends `JTextComponent` (p. 561) of package `javax.swing.text`, which provides common text-based component features. Class `JPasswordField` (p. 561) extends `JTextField` and adds several methods that are specific to processing passwords.
- A `JPasswordField` (p. 561) shows that characters are being typed as the user enters them, but hides the actual characters with echo characters (p. 562).
- A component receives the focus (p. 562) when the user clicks the component.
- `JTextComponent` method `setEditable` (p. 564) can be used to make a text field uneditable.
- To respond to an event for a particular GUI component, you must create a class that represents the event handler and implements an appropriate event-listener interface (p. 564), then register an object of the event-handling class as the event handler (p. 564).
- Non-static nested classes (p. 564) are called inner classes and are frequently used for event handling.
- An object of a non-static inner class (p. 564) must be created by an object of the top-level class (p. 564) that contains the inner class.
- An inner-class object can directly access the instance variables and methods of its top-level class.
- A nested class that's 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.
- Pressing *Enter* in a `JTextField` (p. 561) or `JPasswordField` generates an `ActionEvent` (p. 565) from package `java.awt.event` (p. 567) that can be handled by an `ActionListener` (p. 565; package `java.awt.event`).
- `JTextField` method `addActionListener` (p. 565) registers an event handler for a text field's `ActionEvent`.
- The GUI component with which the user interacts is the event source (p. 566).

- An `ActionEvent` object contains information about the event that just occurred, such as the event source and the text in the text field.
- `ActionEvent` method `getSource` returns a reference to the event source. `ActionEvent` method `getActionCommand` (p. 566) returns the text the user typed in a text field or the label on a `JButton`.
- `JPasswordField` method `getPassword` (p. 566) returns the password the user typed.

Section 14.7 Common GUI Event Types and Listener Interfaces

- Each event-object type typically has a corresponding event-listener interface that specifies one or more event-handling methods which must be declared in the class that implements the interface.

Section 14.8 How Event Handling Works

- 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.
- Every GUI component supports several event types. When an event occurs, the event is dispatched (p. 570) only to the event listeners of the appropriate type.

Section 14.9 JButton

- A button is a component the user clicks to trigger an action. All the button types are subclasses of `AbstractButton` (p. 571; package `javax.swing`). Button labels (p. 571) typically use book-title capitalization (p. 554).
- Command buttons (p. 571) are created with class `JButton`.
- A `JButton` can display an `Icon`. A `JButton` can also have a rollover `Icon` (p. 571)—an `Icon` that's displayed when the user positions the mouse over the button.
- Method `setRolloverIcon` (p. 573) of class `AbstractButton` specifies the image displayed on a button when the user positions the mouse over it.

Section 14.10 Buttons That Maintain State

- There are three Swing state button types—`JToggleButton` (p. 574), `JCheckBox` (p. 574) and `JRadioButton` (p. 574).
- Classes `JCheckBox` and `JRadioButton` are subclasses of `JToggleButton`.
- Component method `setFont` (p. 576) sets the component's font to a new `Font` object (p. 576; package `java.awt`).
- Clicking a `JCheckBox` causes an `ItemEvent` (p. 576) that can be handled by an `ItemListener` (p. 576) which defines method `itemStateChanged` (p. 576). Method `addItemListener` registers the listener for the `ItemEvent` of a `JCheckBox` or `JRadioButton` object.
- `JCheckBox` method `isSelected` determines whether a `JCheckBox` is selected.
- `JRadioButtons` have two states—selected and not selected. Radio buttons (p. 571) normally appear as a group (p. 577) in which only one button can be selected at a time.
- `JRadioButtons` are used to represent mutually exclusive options (p. 577).
- The logical relationship between `JRadioButtons` is maintained by a `ButtonGroup` object (p. 577).
- `ButtonGroup` method `add` (p. 580) associates each `JRadioButton` with a `ButtonGroup`. If more than one selected `JRadioButton` object is added to a group, the selected one that was added first will be selected when the GUI is displayed.
- `JRadioButtons` generate `ItemEvents` when they're clicked.

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

- A `JComboBox` (p. 580) provides a list of items from which the user can make a single selection. `JComboBoxes` generate `ItemEvents`.

- Each item in a JComboBox has an index (p. 582). The first item added to a JComboBox appears as the currently selected item when the JComboBox is displayed.
- JComboBox method setMaximumRowCount (p. 583) sets the maximum number of elements that are displayed when the user clicks the JComboBox.
- An anonymous inner class (p. 583) is an inner class without a name and typically appears inside a method declaration. One object of the anonymous inner class must be created when the class is declared.
- JComboBox method getSelectedIndex (p. 583) returns the index of the selected item.

Section 14.12 JList

- A JList displays a series of items from which the user may select one or more items. Class JList supports single-selection lists (p. 584) and multiple-selection lists.
- When the user clicks an item in a JList, a ListSelectionEvent (p. 584) occurs. JList method addListSelectionListener (p. 586) registers a ListSelectionListener (p. 586) for a JList's selection events. A ListSelectionListener of package javax.swing.event (p. 567) must implement method valueChanged.
- JList method setVisibleRowCount (p. 586) specifies the number of visible items in the list.
- JList method setSelectionMode (p. 586) specifies a list's selection mode (p. 586).
- A JList can be attached to a JScrollPane (p. 586) to provide a scrollbar (p. 583) for the JList.
- JFrame method getContentPane (p. 586) returns a reference to the JFrame's content pane where GUI components are displayed.
- JList method getSelectedIndex (p. 586) returns the selected item's index.

Section 14.13 Multiple-Selection Lists

- A multiple-selection list (p. 584) enables the user to select many items from a JList.
- JList method setFixedCellWidth (p. 588) sets a JList's width. Method setFixedCellHeight (p. 588) sets the height of each item in a JList.
- Normally, an external event (p. 588) generated by another GUI component (such as a JButton) specifies when the multiple selections in a JList should be processed.
- JList method setListData (p. 589) sets the items displayed in a JList. JList method getSelectedValues (p. 589) returns an array of Objects representing the selected items in a JList.

Section 14.14 Mouse Event Handling

- The MouseListener (p. 570) and MouseMotionListener (p. 589) event-listener interfaces are used to handle mouse events (p. 570). Mouse events can be trapped for any GUI component that extends Component.
- Interface MouseInputListener (p. 589) of package javax.swing.event extends interfaces MouseListener and MouseMotionListener to create a single interface containing all their methods.
- Each mouse event-handling method receives a MouseEvent object (p. 570) that contains information about the event, including the x- and y-coordinates where the event occurred. Coordinates are measured from the upper-left corner of the GUI component on which the event occurred.
- The methods and constants of class InputEvent (p. 589; MouseEvent's superclass) enable an application to determine which mouse button the user clicked.
- Interface MouseWheelListener (p. 590) enables applications to respond to the rotation of a mouse wheel.

Section 14.15 Adapter Classes

- An adapter class (p. 594) implements an interface and provides default implementations of its methods. When you extend an adapter class, you can override just the method(s) you need.
- MouseEvent method `getClickCount` (p. 597) returns the number of consecutive mouse-button clicks. Methods `isMetaDown` (p. 604) and `isAltDown` (p. 597) determine which mouse button the user clicked.

Section 14.16 JPanel Subclass for Drawing with the Mouse

- JComponents method `paintComponent` (p. 597) is called when a lightweight Swing component is displayed. Override this method to specify how to draw shapes using Java's graphics capabilities.
- When overriding `paintComponent`, call the superclass version as the first statement in the body.
- Subclasses of JComponent support transparency. When a component is opaque (p. 597), `paintComponent` clears its background before the component is displayed.
- The transparency of a Swing lightweight component can be set with method `setOpaque` (p. 597); a `false` argument indicates that the component is transparent).
- Class `Point` (p. 599) package `java.awt` represents an *x-y* coordinate.
- Class `Graphics` (p. 599) is used to draw.
- MouseEvent method `getPoint` (p. 599) obtains the `Point` where a mouse event occurred.
- Method `repaint` (p. 599), inherited indirectly from class `Component`, indicates that a component should be refreshed on the screen as soon as possible.
- Method `paintComponent` receives a `Graphics` parameter and is called automatically whenever a lightweight component needs to be displayed on the screen.
- `Graphics` method `fillOval` (p. 600) draws a solid oval. The first two arguments are the upper-left *x-y* coordinate of the bounding box, and the last two are the bounding box's width and height.

Section 14.17 Key Event Handling

- Interface `KeyListener` (p. 570) is used to handle key events (p. 570) that are generated when keys on the keyboard are pressed and released. Method `addKeyListener` of class `Component` (p. 601) registers a `KeyListener`.
- `KeyEvent` (p. 570) method `getKeyCode` (p. 603) gets the virtual key code (p. 603) of the key that was pressed. Class `KeyEvent` maintains a set of virtual key-code constants that represent every key on the keyboard.
- `KeyEvent` method `getKeyText` (p. 604) returns a string containing the name of the key that was pressed.
- `KeyEvent` method `getKeyChar` (p. 604) gets the Unicode value of the character typed.
- `KeyEvent` method `isActionKey` (p. 604) determines whether the key in an event was an action key (p. 601).
- `InputEvent` method `getModifiers` (p. 604) determines whether any modifier keys (such as *Shift*, *Alt* and *Ctrl*) were pressed when the key event occurred.
- `KeyEvent` method `getKeyModifiersText` (p. 604) returns a string containing the pressed modifier keys.

Section 14.18 Introduction to Layout Managers

- Layout managers (p. 559) arrange GUI components in a container for presentation purposes.
- All layout managers implement the interface `LayoutManager` (p. 604) of package `java.awt`.
- Container method `setLayout` (p. 559) specifies the layout of a container.

- `FlowLayout` (p. 559) places components left to right in the order in which they're added to the container. When the container's edge is reached, components continue to display on the next line. `FlowLayout` allows GUI components to be left aligned, centered (the default) and right aligned.
- `FlowLayout` method `setAlignment` (p. 608) changes the alignment for a `FlowLayout`.
- `BorderLayout` (p. 592) the default for a `JFrame`) arranges components into five regions: NORTH, SOUTH, EAST, WEST and CENTER. NORTH corresponds to the top of the container.
- A `BorderLayout` limits a Container to containing at most five components—one in each region.
- `GridLayout` (p. 611) divides a container into a grid of rows and columns.
- Container method `validate` (p. 613) recomputes a container's layout based on the current layout manager for the Container and the current set of displayed GUI components.

Section 14.19 Using Panels to Manage More Complex Layouts

- Complex GUIs often consist of multiple panels with different layouts. Every `JPanel` may have components, including other panels, attached to it with Container method `add`.

Section 14.20 JTextArea

- A `JTextArea` (p. 615) may contain multiple lines of text. `JTextArea` is a subclass of `JTextComponent`.
- Class `Box` (p. 617) is a subclass of `Container` that uses a `BoxLayout` layout manager (p. 617) to arrange the GUI components either horizontally or vertically.
- `Box` static method `createHorizontalBox` (p. 617) creates a `Box` that arranges components from left to right in the order that they're attached.
- Method `getSelectedText` (p. 617) returns the selected text from a `JTextArea`.
- You can set the horizontal and vertical scrollbar policies (p. 617) of a `JScrollPane` when it's constructed. `JScrollPane` methods `setHorizontalScrollBarPolicy` (p. 618), and `setVerticalScrollBarPolicy` (p. 618) can be used to change the scrollbar policies at any time.

Self-Review Exercises

14.1 Fill in the blanks in each of the following statements:

- Method _____ is called when the mouse is moved with no buttons pressed and an event listener is registered to handle the event.
- Text that cannot be modified by the user is called _____ text.
- A(n) _____ arranges GUI components in a Container.
- The add method for attaching GUI components is a method of class _____.
- GUI is an acronym for _____.
- Method _____ is used to specify the layout manager for a container.
- A `mouseDragged` method call is preceded by a(n) _____ method call and followed by a(n) _____ method call.
- Class _____ contains methods that display message dialogs and input dialogs.
- An input dialog capable of receiving input from the user is displayed with method _____ of class _____.
- A dialog capable of displaying a message to the user is displayed with method of class _____.
- Both `JTextFields` and `JTextAreas` directly extend class _____.

14.2 Determine whether each statement is *true* or *false*. If *false*, explain why.

- `BorderLayout` is the default layout manager for a `JFrame`'s content pane.
- When the mouse cursor is moved into the bounds of a GUI component, method `mouseOver` is called.
- A `JPanel` cannot be added to another `JPanel`.