O b j e c t i v e s
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.

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
Chapter 14 GUI Components: Part 1

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.

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 programatically 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](http://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.

```
// Fig. 14.2: Addition.java
// Addition program that uses JOptionPane for input and output.
import javax.swing.JOptionPane; // program uses JOptionPane

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

**Fig. 14.2** | Addition program that uses JOptionPane for input and output. (Part 1 of 2.)
14.3 Simple GUI-Based Input/Output with JOptionPane

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).
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.

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.

### Table: Message dialog types

<table>
<thead>
<tr>
<th>Message dialog type</th>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR_MESSAGE</td>
<td>![Exclamation mark]</td>
<td>Indicates an error.</td>
</tr>
<tr>
<td>INFORMATION_MESSAGE</td>
<td>![Information icon]</td>
<td>Indicates an informational message.</td>
</tr>
<tr>
<td>QUESTION_MESSAGE</td>
<td>![Question mark]</td>
<td>Poses a question. This dialog normally requires a response, such as clicking a Yes or a No button.</td>
</tr>
<tr>
<td>PLAIN_MESSAGE</td>
<td>no icon</td>
<td>A dialog that contains a message, but no icon.</td>
</tr>
</tbody>
</table>

**Fig. 14.3** | JOptionPane static constants for message dialogs.

### Table: GUI components

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

**Fig. 14.4** | Some basic GUI components.
**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.

![Common superclasses of the lightweight Swing components.](image_url)

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.]

```java
// Fig. 14.6: LabelFrame.java
// Demonstrating the JLabel class.
import java.awt.FlowLayout; // specifies how components are arranged
import javax.swing.JFrame; // provides basic window features
import javax.swing.JLabel; // displays text and images
import javax.swing.SwingConstants; // common constants used with Swing
import javax.swing.ImageIcon; // loads images

public class LabelFrame extends JFrame {
    public LabelFrame() {
        private JLabel label1; // JLabel with just text
        private JLabel label2; // JLabel constructed with text and icon
        private JLabel label3; // JLabel with added text and icon
        // LabelFrame constructor adds JLabels to JFrame
        public LabelFrame() {
            // Code to add JLabels
        }
    }
}
```

Fig. 14.6 | JLabels with text and icons. (Part 1 of 2.)
Fig. 14.7 | Test class for JLabels with text and icons. (Part 2 of 2.)

```java
// Fig. 14.7: LabelTest.java
// Testing JLabelFrame.
import javax.swing.JFrame;

public class LabelTest {
    public static void main(String[] args) {
        LabelFrame labelFrame = new LabelFrame();  // create LabelFrame
        labelFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        labelFrame.setSize(260, 180); // set frame size
        labelFrame.setVisible(true); // display frame
    }
}
```

Fig. 14.6 | JLabels with text and icons. (Part 2 of 2.)
Class LabelFrame (Fig. 14.6) is a subclass of JFrame. We’ll use an instance of class LabelFrame to display a window containing three JLabels. 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 superclass 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 label1**

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 label1 to the LabelFrame by passing label1 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 setHorizontalAlignment 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.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEFT</td>
<td>Place text on the left</td>
<td>TOP</td>
<td>Place text at the top</td>
</tr>
<tr>
<td>CENTER</td>
<td>Place text in the center</td>
<td>CENTER</td>
<td>Place text in the center</td>
</tr>
<tr>
<td>RIGHT</td>
<td>Place text on the right</td>
<td>BOTTOM</td>
<td>Place text at the bottom</td>
</tr>
</tbody>
</table>

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—JTextField and JPasswordField (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).

```java
// Fig. 14.9: TextFieldFrame.java
// Demonstrating the JTextField class.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JPasswordField;
import javax.swing.JOptionPane;

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

    // TextFieldFrame constructor adds JTextFields to JFrame
    public TextFieldFrame() {
        super( "Testing JTextField and JPasswordField" );
        setLayout( new FlowLayout() ); // set frame layout
        // construct textfield with 10 columns
        textField1 = new JTextField( 10 );
        add( textField1 ); // add textField1 to JFrame
        // construct textfield with default text
        textField2 = new JTextField( "Enter text here" );
        add( textField2 ); // add textField2 to JFrame
    }
}
```

Fig. 14.9 | JTextFields and JPasswordFields. (Part 1 of 2.)
14.6 Text Fields and an Introduction to Event Handling with Nested Classes

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

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

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

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

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

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

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

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

        // display JTextField content
        JOptionPane.showMessageDialog( null, string );
    }
}

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-
14. TextFields and an Introduction to Event Handling with Nested Classes

ate event handlers for the text fields in top-level class JTextFieldFrame. 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 JTextField 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.

```java
// Fig. 14.10: TextFieldTest.java
// Testing TextFieldFrame.
import javax.swing.JFrame;

public class TextFieldTest {
    public static void main( String[] args ) {
        TextFieldFrame textFieldFrame = new TextFieldFrame();
        textFieldFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        textFieldFrame.setSize( 350, 100 ); // set frame size
    }
}
```

Fig. 14.10 | Test class for TextFieldFrame. (Part 1 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

```java
12    textFieldFrame.setVisible( true ); // display frame
13    } // end main
14    } // end class TextFieldTest
```

![Fig. 14.10](image)  
Test class for TextFieldFrame. (Part 2 of 2.)
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.
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 diagramed listenerList as an array below the JTextField object in Fig. 14.13. When line 43 of Fig. 14.9

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

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

![Swing button hierarchy](image)

**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 `JButton`s, 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 `JButtons` 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
// Fig. 14.15: ButtonFrame.java
// Creating JButtons.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JOptionPane;

public class ButtonFrame extends JFrame {

    private JButton plainJButton; // button with just text
    private JButton fancyJButton; // button with icons

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

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

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

        // create new ButtonHandler for button event handling
        ButtonHandler handler = new ButtonHandler();
        fancyJButton.addActionListener( handler );
        plainJButton.addActionListener( handler );
    } // end ButtonFrame constructor

    // private class for button event handling
    private class ButtonHandler implements ActionListener {
        // handle button event
        public void actionPerformed( ActionEvent event ) {
            JOptionPane.showMessageDialog( this, String.format( "You pressed: %s", event.getActionCommand() ) );
        } // end method actionPerformed
    } // end private inner class ButtonHandler

    // 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
// Fig. 14.16: ButtonTest.java
// Testing ButtonFrame.
import javax.swing.JFrame;

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

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

    // CheckBoxFrame constructor adds JCheckBoxes to JFrame
    public CheckBoxFrame() {
        super( "JCheckBox Test" );
        setLayout( new FlowLayout() ); // set frame layout
        // set up JTextField and set its font
        textField = new JTextField( "Watch the font style change", 20 );
        textField.setFont( new Font( "Serif", Font.PLAIN, 14 ) ); // add textField to JFrame
        boldJCheckBox = new JCheckBox( "Bold" ); // create bold checkbox
        italicJCheckBox = new JCheckBox( "Italic" ); // create italic
        add( boldJCheckBox ); // add bold checkbox to JFrame
        add( italicJCheckBox ); // add italic checkbox to JFrame
        // register listeners for JCheckBoxes
        CheckBoxHandler handler = new CheckBoxHandler();
        boldJCheckBox.addItemListener( handler );
        italicJCheckBox.addItemListener( handler );
    }

    // private inner class for ItemListener event handling
    private class CheckBoxHandler implements ItemListener {
        // respond to checkbox events
        public void itemStateChanged( ItemEvent event ) {
            Font font = null; // stores the new Font
            // determine which CheckBoxes are checked and create Font
            if ( boldJCheckBox.isSelected() && italicJCheckBox.isSelected() )
                font = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
            else if ( boldJCheckBox.isSelected() )
                font = new Font( "Serif", Font.BOLD, 14 );
            else if ( italicJCheckBox.isSelected() )
                font = new Font( "Serif", Font.ITALIC, 14 );
        }
    }
}
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.
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JRadioButton;
import javax.swing.ButtonGroup;

public class RadioButtonFrame extends JFrame {

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

    public RadioButtonFrame() {
        super( "RadioButton Test" ); // set frame layout
        textField = new JTextField("Watch the font style change", 25); // add textField to JFrame
        textField.setFont( plainFont ); // set initial font to plain

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

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

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

        // RadioButtonFrame constructor adds JRadioButtons to JFrame
        // Fig. 14.19 | JRadioButtons and ButtonGroups. (Part 2 of 3.)
14.10 Buttons That Maintain State

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

Fig. 14.20 | Test class for RadioButtonFrame. (Part 1 of 2.)
Chapter 14  GUI Components: Part 1

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-
14.11 JComboBox: Using an Anonymous Inner Class for Event Handling

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.

```java
// Fig. 14.21: ComboBoxFrame.java
// JComboBox that displays a list of image names.
import java.awt.FlowLayout;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JComboBox;
import javax.swing.Icon;
import javax.swing.ImageIcon;

public class ComboBoxFrame extends JFrame {
    private JLabel label; // label to display selected icon
    private JComboBox imagesJComboBox; // combobox to hold names of icons
    private Icon[] icons = {
        new ImageIcon( getClass().getResource(names[0]) ),
        new ImageIcon( getClass().getResource(names[1]) ),
        new ImageIcon( getClass().getResource(names[2]) ),
        new ImageIcon( getClass().getResource(names[3]) ) };

    // ComboBoxFrame constructor adds JComboBox to JFrame
    public ComboBoxFrame() {
        super("Testing JComboBox");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout()); // set frame layout
        imagesJComboBox = new JComboBox(names); // set up JComboBox
        imagesJComboBox.setMaximumRowCount(3); // display three rows
        imagesJComboBox.addItemListener(new ItemListener() { // anonymous inner class
            public void itemStateChanged(ItemEvent event) {
                // determine whether item selected
                if (event.getStateChange() == ItemEvent.SELECTED)
                    label.setIcon(icons[imagesJComboBox.getSelectedIndex()]);
            }
        }); // end anonymous inner class
    }
}
```

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

```java
// Fig. 14.23: ListFrame.java
// JList that displays a list of colors.
import java.awt.FlowLayout;
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.event.ListSelectionListener;
import javax.swing.event.ListSelectionEvent;
import javax.swing.ListSelectionModel;

public class ListFrame extends JFrame {
    private static final String[] colorNames = {
        "Black", "Blue", "Cyan",
        "Orange", "Pink", "Red", "White", "Yellow"};
    private static final Color[] colors = {
        Color.BLACK, Color.BLUE,
        Color.CYAN, Color.DARK_GRAY, Color.GRAY, Color.GREEN,
        Color.LIGHT_GRAY, Color.MAGENTA, Color.ORANGE, Color.PINK,
        Color.RED, Color.WHITE, Color.YELLOW};
}
```

**Fig. 14.23** | JList that displays a list of colors. (Part 1 of 2.)
Fig. 14.24 | Test class for ListFrame.

---

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

public ListFrame()
{
    super( "List Test" );
    setLayout( new FlowLayout() ); // set frame layout
    colorJList = new JList( colorNames ); // create with colorNames
    colorJList.setVisibleRowCount( 5 ); // display five rows at once
    colorJList.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
    add( new JScrollPane( colorJList ) );
    colorJList.addListSelectionListener( new ListSelectionListener() // anonymous inner class
    {
        // handle list selection events
        public void valueChanged( ListSelectionEvent event )
        {
            getContentPane().setBackground(
                colors[ colorJList.getSelectedIndex() ]);
            } // end method valueChanged
        } // end anonymous inner class
    ); // end call to addListSelectionListener
} // end ListFrame constructor

Fig. 14.24 | ListTest.java
// Selecting colors from a JList.
import javax.swing.JFrame;

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

```java
// Fig. 14.25: MultipleSelectionFrame.java
// Copying items from one List to another.
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JButton;
import javax.swing.JScrollPane;
import javax.swing.ListSelectionModel;

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

    // MultipleSelectionFrame constructor
    public MultipleSelectionFrame() {
        super( "Multiple Selection Lists" );
        setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        super.setVisible( true );

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

        copyJButton = new JButton( "Copy >>>" ); // create copy button
        copyJButton.addActionListener( new ActionListener() {
            public void actionPerformed( ActionEvent event ) {
                // place selected values in copyJList
                copyJList.setListData( colorJList.getSelectedValues() );
            }
        } ); // end anonymous inner class
        add( copyJButton ); // add copy button to JFrame

        copyJList = new JList(); // create list to hold copied color names
    }
}
```

Fig. 14.25  |  JList that allows multiple selections. (Part 1 of 2.)
Chapter 14  GUI Components: Part 1

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

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.

<table>
<thead>
<tr>
<th>MouseListener and MouseMotionListener interface methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Methods of interface MouseListener</strong></td>
</tr>
<tr>
<td>public void mousePressed( MouseEvent event )</td>
</tr>
<tr>
<td>Called when a mouse button is pressed while the mouse cursor is on a component.</td>
</tr>
<tr>
<td>public void mouseClicked( MouseEvent event )</td>
</tr>
<tr>
<td>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 <code>mousePressed</code>.</td>
</tr>
<tr>
<td>public void mouseReleased( MouseEvent event )</td>
</tr>
<tr>
<td>Called when a mouse button is released after being pressed. This event is always preceded by a call to <code>mousePressed</code> and one or more calls to <code>mouseDragged</code>.</td>
</tr>
</tbody>
</table>

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

```
// Fig. 14.28: MouseTrackerFrame.java
// Demonstrating mouse events.
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;
import java.awt.event.MouseWheelEvent;
import java.awt.event.MouseWheelListener;

public class MouseTrackerFrame extends JFrame {
    public MouseTrackerFrame() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);

        JPanel panel = new JPanel();
        panel.setLayout(new BorderLayout);
        panel.add(new JButton("A"), BorderLayout.NORTH);
        panel.add(new JButton("B"), BorderLayout.SOUTH);
        panel.add(new JButton("C"), BorderLayout.EAST);
        panel.add(new JButton("D"), BorderLayout.WEST);
        panel.add(new JButton("E"), BorderLayout.CENTER);

        JLabel statusBar = new JLabel("Status Bar");  // attach to JFrame
        panel.add(statusBar, BorderLayout.SOUTH);

        add(panel);
    }

    public MouseTrackerFrame() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 200);
        setLocationRelativeTo(null);
    }

    public static void main(String[] args) {
        MouseTrackerFrame frame = new MouseTrackerFrame();
        frame.setVisible(true);
    }
}
```
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class MouseTrackerFrame extends JFrame {

    private JPanel mousePanel; // panel in which mouse events will occur
    private JLabel statusBar; // label that displays event information

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

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

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

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

    private class MouseHandler implements MouseListener, MouseMotionListener {

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

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

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

    Fig. 14.28 | Mouse event handling. (Part 2 of 3.)
Chapter 14  GUI Components: Part 1

592

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.
14.14 MouseEventHandling

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 MouseListeners and MouseMotionListeners, 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.

Fig. 14.29 | Test class for MouseTrackerFrame.

```java
import javax.swing.JFrame;

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

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.

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

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

```
Fig. 14.31 | Left, center and right mouse-button clicks. (Part 1 of 2.)
```
import java.awt.event.MouseEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MouseDetailsFrame extends JFrame {
    private String details; // String that is displayed in the status bar
    private JLabel statusBar; // JLabel that appears at bottom of window

    // constructor sets title bar String and register mouse listener
    public MouseDetailsFrame() {
        super("Mouse clicks and buttons");
        statusBar = new JLabel("Click the mouse");
        add(statusBar, BorderLayout.SOUTH);
        addMouseListener(new MouseClickHandler()); // add handler
    }

    // inner class to handle mouse events
    private class MouseClickHandler extends MouseAdapter {
        // handle mouse-click event and determine which button was pressed
        public void mouseClicked(MouseEvent event) {
            int xPos = event.getX(); // get x-position of mouse
            int yPos = event.getY(); // get y-position of mouse

            details = String.format("Clicked %d time(s)",
                event.getClickCount());

            if (event.isMetaDown()) // right mouse button
                details += " with right mouse button";
            else if (event.isAltDown()) // middle mouse button
                details += " with center mouse button";
            else // left mouse button
                details += " with left mouse button";

            statusBar.setText(details); // display message in statusBar
        }
    }
}

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

// Fig. 14.32: MouseDetails.java
// Testing MouseDetailsFrame.
import javax.swing.JFrame;

public class MouseDetails {
    public class MouseDetailsFrame extends JFrame {
        private String details; // String that is displayed in the status bar
        private JLabel statusBar; // JLabel that appears at bottom of window

        // constructor sets title bar String and register mouse listener
        public MouseDetailsFrame() {
            super("Mouse clicks and buttons");
            statusBar = new JLabel("Click the mouse");
            add(statusBar, BorderLayout.SOUTH);
            addMouseListener(new MouseClickHandler()); // add handler
        }

        // inner class to handle mouse events
        private class MouseClickHandler extends MouseAdapter {
            // handle mouse-click event and determine which button was pressed
            public void mouseClicked(MouseEvent event) {
                int xPos = event.getX(); // get x-position of mouse
                int yPos = event.getY(); // get y-position of mouse

                details = String.format("Clicked %d time(s)",
                    event.getClickCount());

                if (event.isMetaDown()) // right mouse button
                    details += " with right mouse button";
                else if (event.isAltDown()) // middle mouse button
                    details += " with center mouse button";
                else // left mouse button
                    details += " with left mouse button";

                statusBar.setText(details); // display message in statusBar
            }
        }
    }
}

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

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

**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.*
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).
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.

```java
// Fig. 14.34: PaintPanel.java
// Using class MouseMotionAdapter.
import java.awt.Point;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;
import javax.swing.JPanel;

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

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

    // set up GUI and register mouse event handler
    public PaintPanel()
    {
        // handle frame mouse motion event
        addMouseMotionListener(
            new MouseMotionAdapter() // anonymous inner class
                {
                        // store drag coordinates and repaint
                        public void mouseDragged( MouseEvent event )
                            {
                                if ( pointCount < points.length )
                                    {
                                        //...
                                    }
                }
        );
    }
}
```

Fig. 14.34 | Adapter class used to implement event handlers. (Part 1 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 (JPanel’s 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.

```java
// Fig. 14.35: Painter.java
// Testing PaintPanel.
import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class Painter {
    public static void main( String[] args )
    {
        // create JFrame
        JFrame application = new JFrame( "A simple paint program" );
        PaintPanel paintPanel = new PaintPanel(); // create paint panel
        application.add( paintPanel, BorderLayout.CENTER ); // in center
        // create a label and place it in SOUTH of BorderLayout
        application.add( new JLabel( "Drag the mouse to draw" ), BorderLayout.SOUTH );
        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    }
}
```

Fig. 14.35  |  Test class for PaintPanel. (Part 1 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.
// KeyDemoFrame constructor
public KeyDemoFrame()
{
    super("Demonstrating Keystroke Events");

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

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

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

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

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

// set second and third lines of output
private void setLines2and3(KeyEvent event)
{
    String temp = KeyEvent.getKeyModifiersText(event.getModifiers());

    line2 = String.format("This key is %san action key", (event.isActionKey() ? "" : "not "));

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

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

Fig. 14.36 | Key event handling. (Part 2 of 2.)
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.
14.18 Introduction to Layout Managers

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.

<table>
<thead>
<tr>
<th>Layout manager</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlowLayout</td>
<td>Default for javax.swing.JPanel. 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 Container method add, which takes a Component and an integer index position as arguments.</td>
</tr>
<tr>
<td>BorderLayout</td>
<td>Default for JFrame (and other windows). Arranges the components into five areas: NORTH, SOUTH, EAST, WEST and CENTER.</td>
</tr>
<tr>
<td>GridLayout</td>
<td>Arranges the components into rows and columns.</td>
</tr>
</tbody>
</table>

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 ComponentListener.

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.

```java
// Fig. 14.39: FlowLayoutFrame.java
// Demonstrating FlowLayout alignments.
import java.awt.FlowLayout;
import java.awt.Container;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;

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

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

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

        // set up leftJButton and register listener
        leftJButton = new JButton( "Left" ); // create Left button
        add( leftJButton ); // add Left button to frame
        leftJButton.addActionListener( new ActionListener() // anonymous inner class
        {
            // process leftJButton event
            public void actionPerformed( ActionEvent event )
            {
                layout.setAlignment( FlowLayout.LEFT );
                // realign attached components
                layout.layoutContainer( container );
            }
        } // end anonymous inner class
    } // end call to addActionListener

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

Fig. 14.39  |  FlowLayout allows components to flow over multiple lines. (Part 1 of 2.)
14.18 Introduction to Layout Managers  607

```java
centerJButton.addActionListener(
    new ActionListener() // anonymous inner class
    {
        // process centerJButton event
        public void actionPerformed( ActionEvent event )
        {
            layout.setAlignment( FlowLayout.CENTER );
            layout.layoutContainer( container );
        } // end method actionPerformed
    } // end anonymous inner class
); // end call to addActionListener

// set up rightJButton and register listener
rightJButton = new JButton( "Right" ); // create Right button
add( rightJButton ); // add Right button to frame
rightJButton.addActionListener( 
    new ActionListener() // anonymous inner class
    {
        // process rightJButton event
        public void actionPerformed( ActionEvent event )
        {
            layout.setAlignment( FlowLayout.RIGHT );
            layout.layoutContainer( container );
        } // end method actionPerformed
    } // end anonymous inner class
); // end call to addActionListener
```

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

---

```java
// Fig. 14.40: FlowLayoutDemo.java
// Testing FlowLayoutFrame.
import javax.swing.JFrame;

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

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

```java
// Fig. 14.41: BorderLayoutFrame.java
// Demonstrating BorderLayout.
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;

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

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

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

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

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

    // handle button events
    public void actionPerformed(ActionEvent event) {
        // check event source and lay out content pane correspondingly
        for (JButton button : buttons) {
            if (event.getSource() == button) {
                button.setVisible(false); // hide button clicked
            }
            else {
                button.setVisible(true); // show other buttons
            }
        } // end for
    }
}
```

Fig. 14.41 | BorderLayout containing five buttons. (Part 1 of 2.)
// Fig. 14.42: BorderLayoutDemo.java  
// Testing BorderLayoutFrame.  
import javax.swing.JFrame;  

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

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

Fig. 14.42  |  Test class for BorderLayoutFrame.
14.18 Introduction to Layout Managers

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.

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 LayoutManager 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.
The `GridLayout` constructor used at line 25 specifies a `GridLayout` with 3 rows and 2 columns that uses the default gap space (1 pixel).
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 JButtons 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 JButtons in array buttons). Lines 23–27 add the JButtons 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.

```java
// Fig. 14.45: JPanelFrame.java
// Using a JPanel to help lay out components.
import java.awt.GridLayout;
import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;

public class JPanelFrame extends JFrame {
    private JPanel buttonJPanel; // panel to hold buttons
    private JButton[] buttons; // array of buttons

    // no-argument constructor
    public JPanelFrame() {
        super("Panel Demo");
        buttons = new JButton[5]; // create buttons array
        buttonJPanel = new JPanel(); // set up panel
        buttonJPanel.setLayout( new GridLayout(1, buttons.length ) );

        // create and add buttons
        for ( int count = 0; count < buttons.length; count++ )
            buttons[ count ] = new JButton( "Button "+ ( count + 1 ) );
        buttonJPanel.add( buttons[ count ] ); // add button to panel
    }
    // end PanelFrame constructor
}
// end class JPanelFrame
```

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

```java
// Fig. 14.46: PanelDemo.java
// Testing JPanelFrame.
import javax.swing.JFrame;

public class PanelDemo extends JFrame {
    public static void main( String[] args ) {
        JFrame panelFrame = new JPanelFrame();
        panelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        panelFrame.setSize( 450, 200 ); // set frame size
    }
}
// end class PanelDemo
```

**Fig. 14.46** | Test class for JPanelFrame. (Part 1 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 JTextField, 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.

```java
// Fig. 14.47: TextAreaFrame.java
// Copying selected text from one textarea to another.
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.Box;
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

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

Fig. 14.47 | Copying selected text from one JTextArea to another. (Part 1 of 2.)

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

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

Fig. 14.47 | Copying selected text from one JTextArea to another. (Part 1 of 2.)
// no-argument constructor
public TextAreaFrame() {
    super("TextArea Demo");
    Box box = Box.createHorizontalBox(); // create box
    String demo = "This is a demo string to \n" + "illustrate copying text from one textarea to another textarea using an external event\n";
    TextArea1 = new JTextArea( demo, 10, 15 ); // create textArea1
    box.add( new JScrollPane( textArea1 ) ); // add scrollpane
    copyJButton = new JButton( "Copy >>>" ); // create copy button
    box.add( copyJButton ); // add copy button to box
    copyJButton.addActionListener(
        new ActionListener() // anonymous inner class
            {
                public void actionPerformed( ActionEvent event ) // set text in textArea2 to selected text from textArea1
                    {
                        textArea2.setText( textArea1.getSelectedText() );
                    } // end method actionPerformed
            } // end anonymous inner class
        ); // end call to addActionListener
    TextArea2 = new JTextArea( 10, 15 ); // create second textarea
    TextArea2.setEditable( false ); // disable editing
    box.add( new JScrollPane( textArea2 ) ); // add scrollpane
}
}

// end class TextAreaFrame

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

// Fig. 14.48: TextAreaDemo.java
// Copying selected text from one textarea to another.
import javax.swing.JFrame;

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

Fig. 14.48 | Test class for TextAreaFrame. (Part 1 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-
ereference 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

```java
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
```
to indicate that a scrollbar should always appear, constants

```java
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
```
to indicate that a scrollbar should appear only if necessary (the defaults) and constants

```java
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 JPanel s 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 JPanel s 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 input `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 `Component`s 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).
Chapter 14 GUI Components: Part 1

- **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` 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`.

• `PasswordField` 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.

• `JRadioButton`s 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.

• `JRadioButton`s are used to represent mutually exclusive options (p. 577).

• The logical relationship between `JRadioButton`s 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.

• `JRadioButton`s 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. 583) 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

• JComponent 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 getModifiersText (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.

• **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:

a) Method ________ is called when the mouse is moved with no buttons pressed and an event listener is registered to handle the event.

b) Text that cannot be modified by the user is called ________ text.

c) A(n) ________ arranges GUI components in a Container.

d) The add method for attaching GUI components is a method of class ________.

e) GUI is an acronym for ________.

f) Method ________ is used to specify the layout manager for a container.

g) A mouseDragged method call is preceded by a(n) ________ method call and followed by a(n) ________ method call.

h) Class ________ contains methods that display message dialogs and input dialogs.

i) An input dialog capable of receiving input from the user is displayed with method ________ of class ________.

j) A dialog capable of displaying a message to the user is displayed with method of class ________.

k) Both JTextField and JTextArea directly extend class ________.

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

a) BorderLayout is the default layout manager for a JFrame’s content pane.

b) When the mouse cursor is moved into the bounds of a GUI component, method `mouseOver` is called.

c) A JPanel cannot be added to another JPanel.
d) In a BorderLayout, two buttons added to the NORTH region will be placed side by side.

e) A maximum of five components can be added to a BorderLayout.

f) Inner classes are not allowed to access the members of the enclosing class.

g) A JTextArea’s text is always read-only.

h) Class JTextArea is a direct subclass of class Component.

14.3 Find the error(s) in each of the following statements, and explain how to correct it (them):

a) buttonName = JButton( "Caption" );

b) JLabel aLabel, JLabel; // create references

c) textField = new JTextField( 50, "Default Text" );

d) setLayout( new BorderLayout() );

button1 = new JButton( "North Star" );

button2 = new JButton( "South Pole" );

add( button1 );

add( button2 );

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

a) The JTextField class directly extends class ________.

b) Container method ________ attaches a GUI component to a container.

c) Method ________ is called when a mouse button is released (without moving the mouse).

d) The ________ class is used to create a group of JRadioButton.

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

a) Only one layout manager can be used per Container.

b) GUI components can be added to a Container in any order in a BorderLayout.

c) JRadioButton provides a series of mutually exclusive options (i.e., only one can be true at a time).
d) The `setFont` method is used to set the font for text fields.

e) A `JList` displays a scrollbar if there are more items in the list than can be displayed.

f) A `Mouse` object has a method called `mouseDragged`.

### 14.6 Determine whether each statement is true or false. If false, explain why.

a) A `JPanel` is a `JComponent`.

b) A `JPanel` is a `Component`.

c) A `JLabel` is a `Container`.

d) A `JList` is a `JPanel`.

e) An `AbstractButton` is a `JButton`.

f) A `JTextField` is an `Object`.

g) `ButtonGroup` is a subclass of `JComponent`.

### 14.7 Find any errors in each of the following lines of code, and explain how to correct them.

a) `import javax.swing.JFrame`  

b) `panelObject.GridLayout( 8, 8 );`  
   // set GridLayout

c) `container.setLayout( new FlowLayout( FlowLayout.DEFAULT ) );`  

d) `container.add( eastButton, EAST );`  
   // BorderLayout

### 14.8 Create the following GUI. You do not have to provide any functionality.

### 14.9 Create the following GUI. You do not have to provide any functionality.

### 14.10 Create the following GUI. You do not have to provide any functionality.

### 14.11 Create the following GUI. You do not have to provide any functionality.
14.12 *(Temperature Conversion)* Write a temperature-conversion application that converts from Fahrenheit to Celsius. The Fahrenheit temperature should be entered from the keyboard (via a JTextField). A JLabel should be used to display the converted temperature. Use the following formula for the conversion:

\[ Celsius = \frac{5}{9} \times (Fahrenheit - 32) \]

14.13 *(Temperature-Conversion Modification)* Enhance the temperature-conversion application of Exercise 14.12 by adding the Kelvin temperature scale. The application should also allow the user to make conversions between any two scales. Use the following formula for the conversion between Kelvin and Celsius (in addition to the formula in Exercise 14.12):

\[ Kelvin = Celsius + 273.15 \]

14.14 *(Guess-the-Number Game)* Write an application that plays “guess the number” as follows: Your application chooses the number to be guessed by selecting an integer at random in the range 1–1000. The application then displays the following in a label:

I have a number between 1 and 1000. Can you guess my number?
Please enter your first guess.

A JTextField should be used to input the guess. As each guess is input, the background color should change to either red or blue. Red indicates that the user is getting “warmer,” and blue, “colder.” A JLabel should display either “Too High” or “Too Low” to help the user zero in. When the user gets the correct answer, “Correct!” should be displayed, and the JTextField used for input should be changed to be uneditable. A JButton should be provided to allow the user to play the game again. When the JButton is clicked, a new random number should be generated and the input JTextField changed to be editable.

14.15 *(Displaying Events)* It’s often useful to display the events that occur during the execution of an application. This can help you understand when the events occur and how they’re generated. Write an application that enables the user to generate and process every event discussed in this chapter. The application should provide methods from the ActionListener, ItemListener, ItemSelectionListener, MouseListener, MouseMotionListener and KeyListener interfaces to display messages when the events occur. Use method toString to convert the event objects received in each event handler into Strings that can be displayed. Method toString creates a String containing all the information in the event object.

14.16 *(GUI-Based Craps Game)* Modify the application of Section 6.10 to provide a GUI that enables the user to click a JButton to roll the dice. The application should also display four JLabels and four JTextFields, with one JLabel for each JTextField. The JTextFields should be used to display the values of each die and the sum of the dice after each roll. The point should be displayed in the fourth JTextField when the user does not win or lose on the first roll and should continue to be displayed until the game is lost.

(Optional) GUI and Graphics Case Study Exercise: Expanding the Interface

14.17 *(Interactive Drawing Application)* In this exercise, you’ll implement a GUI application that uses the MyShape hierarchy from GUI and Graphics Case Study Exercise 10.2 to create an interactive drawing application. You’ll create two classes for the GUI and provide a test class that launches the application. The classes of the MyShape hierarchy require no additional changes.

The first class to create is a subclass of JPanel called DrawPanel, which represents the area on which the user draws the shapes. Class DrawPanel should have the following instance variables:

a) An array shapes of type MyShape that will store all the shapes the user draws.
b) An integer shapeCount that counts the number of shapes in the array.
c) An integer shapeType that determines the type of shape to draw.
d) A MyShape currentShape that represents the current shape the user is drawing.
c) A Color currentColor that represents the current drawing color.

f) A boolean filledShape that determines whether to draw a filled shape.

g) A JLabel statusLabel that represents the status bar. The status bar will display the coordinates of the current mouse position.

Class DrawPanel should also declare the following methods:

a) Overridden method paintComponent that draws the shapes in the array. Use instance variable shapeCount to determine how many shapes to draw. Method paintComponent should also call currentShape’s draw method, provided that currentShape is not null.

b) Set methods for the shapeType, currentColor and filledShape.

c) Method clearLastShape should clear the last shape drawn by decrementing instance variable shapeCount. Ensure that shapeCount is never less than zero.

d) Method clearDrawing should remove all the shapes in the current drawing by setting shapeCount to zero.

Methods clearLastShape and clearDrawing should call repaint (inherited from JPanel) to refresh the drawing on the DrawPanel by indicating that the system should call method paintComponent.

Class DrawPanel should also provide event handling to enable the user to draw with the mouse. Create a single inner class that both extends MouseAdapter and implements MouseMotionListener to handle all mouse events in one class.

In the inner class, override method mousePressed so that it assigns currentShape a new shape of the type specified by shapeType and initializes both points to the mouse position. Next, override method mouseReleased to finish drawing the current shape and place it in the array. Set the second point of currentShape to the current mouse position and add currentShape to the array. Instance variable shapeCount determines the insertion index. Set currentShape to null and call method repaint to update the drawing with the new shape.

Override method mouseMoved to set the text of the statusLabel so that it displays the mouse coordinates—this will update the label with the coordinates every time the user moves (but does not drag) the mouse within the DrawPanel. Next, override method mouseDragged so that it sets the second point of currentShape to the current mouse position and calls method repaint. This will allow the user to see the shape while dragging the mouse. Also, update the JLabel in mouseDragged with the current position of the mouse.

Create a constructor for DrawPanel that has a single JLabel parameter. In the constructor, initialize statusLabel with the value passed to the parameter. Also initialize array shapes with 100 entries, shapeCount to 0, shapeType to the value that represents a line, currentShape to null and currentColor to Color.BLACK. The constructor should then set the background color of the DrawPanel to Color.WHITE and register the MouseListener and MouseMotionListener so the JPanel properly handles mouse events.

Next, create a JFrame subclass called DrawFrame that provides a GUI that enables the user to control various aspects of drawing. For the layout of the DrawFrame, we recommend a BorderLayout, with the components in the NORTH region, the main drawing panel in the CENTER region, and a status bar in the SOUTH region, as in Fig. 14.49. In the top panel, create the components listed below. Each component’s event handler should call the appropriate method in class DrawPanel.

a) A button to undo the last shape drawn.

b) A button to clear all shapes from the drawing.

c) A combo box for selecting the color from the 13 predefined colors.

d) A combo box for selecting the shape to draw.

e) A checkbox that specifies whether a shape should be filled or unfilled.

Declare and create the interface components in DrawFrame’s constructor. You’ll need to create the status bar JLabel before you create the DrawPanel, so you can pass the JLabel as an argument to DrawPanel’s constructor. Finally, create a test class that initializes and displays the DrawFrame to execute the application.
14.18 (GUI-Based Version of the ATM Case Study) Reimplement the ATM Case Study of Chapters 12–13 as a GUI-based application. Use GUI components to approximate the ATM user interface shown in Fig. 12.1. For the cash dispenser and the deposit slot use JButton labeled Remove Cash and Insert Envelope. This will enable the application to receive events indicating when the user takes the cash and inserts a deposit envelope, respectively.

Making a Difference

14.19 (Ecofont) Ecofont (www.ecofont.eu/ecofont_en.html)—developed by SPRANQ (a Netherlands-based company)—is a free, open-source computer font designed to reduce by as much as 20% the amount of ink used for printing, thus reducing also the number of ink cartridges used and the environmental impact of the manufacturing and shipping processes (using less energy, less fuel for shipping, and so on). The font, based on sans-serif Verdana, has small circular “holes” in the letters that are not visible in smaller sizes—such as the 9- or 10-point type frequently used. Download Ecofont, then install the font file Spranq_eco_sans_regular.ttf using the instructions from the Ecofont website. Next, develop a GUI-based program that allows you to type in a text string to be displayed in the Ecofont. Create Increase Font Size and Decrease Font Size buttons that allow you to scale up or down by one point at a time. Start with a default font size of 9 points. As you scale up, you’ll be able to see the holes in the letters more clearly. As you scale down, the holes will be less apparent. What is the smallest font size at which you begin to notice the holes?

14.20 (Typing Tutor: Tuning a Crucial Skill in the Computer Age) Typing quickly and correctly is an essential skill for working effectively with computers and the Internet. In this exercise, you’ll build a GUI application that can help users learn to “touch type” (i.e., type correctly without looking at the keyboard). The application should display a virtual keyboard (Fig. 14.50) and should allow the user to watch what he or she is typing on the screen without looking at the actual keyboard. Use JButton to represent the keys. As the user presses each key, the application highlights the corresponding JButton on the GUI and adds the character to a JTextArea that shows what the user has typed so far. [Hint: To highlight a JButton, use its setBackground method to change its background...]

Fig. 14.49 | Interface for drawing shapes.
color. When the key is released, reset its original background color. You can obtain the JButton’s original background color with the setBackground method before you change its color.

You can test your program by typing a pangram—a phrase that contains every letter of the alphabet at least once—such as “The quick brown fox jumped over a lazy dog.” You can find other pangrams on the web.

To make the program more interesting you could monitor the user’s accuracy. You could have the user type specific phrases that you’ve prestored in your program and that you display on the screen above the virtual keyboard. You could keep track of how many keystrokes the user types correctly and how many are typed incorrectly. You could also keep track of which keys the user is having difficulty with and display a report showing those keys.

Fig. 14.50 | Typing tutor.