12

Graphical User Interface Components: Part 1

Objectives

- To understand the design principles of graphical user interfaces (GUI).
- To be able to build graphical user interfaces.
- To understand the packages containing GUI components and event-handling classes and interfaces.
- To be able to create and manipulate buttons, labels, lists, text fields and panels.
- To understand mouse events and keyboard events.
- To understand and be able to use layout managers.

… the wisest prophets make sure of the event first.
Horace Walpole

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

Speak the affirmative; emphasize your choice by utter ignoring of all that you reject.
Ralph Waldo Emerson

You pays your money and you takes your choice.
Punch

Guess if you can, choose if you dare.
Pierre Corneille

All hope abandon, ye who enter here!
Dante Alighieri

Exit, pursued by a bear.
William Shakespeare

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 647 Saturday, July 7, 2001 2:15 PM

Outline

12.1 Introduction
12.2 Swing Overview
12.3 `JLabel`
12.4 Event-Handling Model
12.5 `JTextField` and `JPasswordField`
12.5.1 How Event Handling Works
12.6 `JButton`
12.7 `JCheckBox` and `JRadioButton`
12.8 `JComboBox`
12.10 Multiple-Selection Lists
12.11 Mouse Event Handling
12.12 Adapter Classes
12.13 Keyboard Event Handling
12.14 Layout Managers
12.14.1 `FlowLayout`
12.14.2 `BorderLayout`
12.14.3 `GridLayout`
12.15 Panels
12.16 (Optional Case Study) Thinking About Objects: Use Cases

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

12.1 Introduction

A *graphical user interface* (*GUI*) presents a pictorial interface to a program. A GUI (pronounced "GOO-EE") gives a program a distinctive "look" and "feel." Providing different programs with a consistent set of intuitive user interface components provides users with a basic level of familiarity with each program before they ever use it. In turn, this reduces the time users require to learn a program and increases their ability to use the program in a productive manner.

Look-and-Feel Observation 12.1

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

As an example of a GUI, Fig. 12.1 contains a Netscape Navigator window with some of its GUI components labeled. In the window, there is a *menu bar* containing *menus* (**File**, **Edit**, **View** etc.). Below the menu bar there is a set of *buttons* that each have a defined task in Netscape Navigator. To the right of the buttons there is a *text field* in which the user can type the name of the World Wide Web site to visit. The menus, buttons and text fields are part of Netscape Navigator's GUI. They enable you to interact with the Navigator program.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 648 Saturday, July 7, 2001 2:15 PM

In this chapter and the next, we demonstrate many GUI components that enable users to interact with your programs.

GUIs are built from *GUI components* (sometimes called *controls* or *widgets*—shorthand notation 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. Several common GUI components are listed in Figure 12.2. In the sections that follow, we discuss each of these GUI components in detail. In the next chapter, we discuss more advanced GUI components.

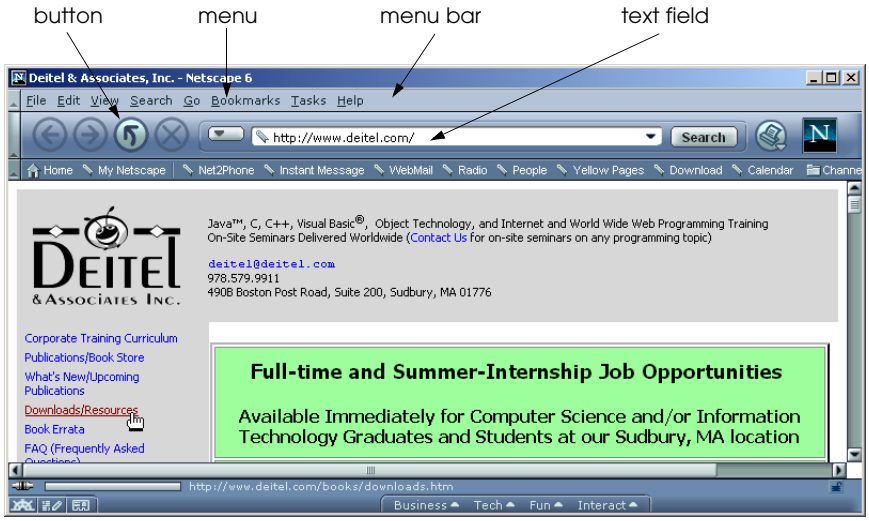


Fig. 12.1 A sample Netscape Navigator window with GUI components.

Component	Description
`JLabel`	An area where uneditable text or icons can be displayed.
`JTextField`	An area in which the user inputs data from the keyboard. The area can also display information.
`JButton`	An area that triggers an event when clicked.
`JCheckBox`	A GUI component that is either selected or not selected.
`JComboBox`	A drop-down list of items from which the user can make a selection by clicking an item in the list or possibly by typing into the box.
`JList`	An area where a list of items is displayed from which the user can make a selection by clicking once on any element in the list. Double-clicking an element in the list generates an action event. Multiple elements can be selected.
`JPanel`	A container in which components can be placed.

Fig. 12.2 Some basic GUI components.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 649 Saturday, July 7, 2001 2:15 PM

12.2 Swing Overview

The classes that create the GUI components of Fig. 12.2 are part of the *Swing GUI components* from package ***javax.swing***. These GUI components became standard in Java with the release of the Java 2 platform version 1.2. Most *Swing components* (as they are commonly called) are written, manipulated and displayed completely in Java (so-called *pure Java* components).

The original GUI components from the *Abstract Windowing Toolkit* package ***java.awt*** (also called the *AWT*) are tied directly to the local platform's graphical user interface capabilities. When a Java program with an AWT GUI executes on different Java platforms, the program's GUI components display differently on each platform. Consider a program that displays an object of type **Button** (package **java.awt**). On a computer running the Microsoft Windows operating system, the **Button** will have the same look and feel as the buttons in other Windows applications. Similarly, on a computer running the Apple Macintosh operating system, the **Button** will have the same look and feel as the buttons in other Macintosh applications. In addition to the differences in appearance, sometimes the manner in which a user interacts with a particular AWT component differs between platforms.

Together, the appearance and how the user interacts with the program are known as that program's *look and feel*. The Swing components allow the programmer to specify a uniform look and feel across all platforms. In addition, Swing enables programs to provide a custom look and feel for each platform or even to change the look and feel while the program is running. For example, a program could enable users to choose their preferred look and feel.

Look-and-Feel Observation 12.2

Swing components are written in Java, so they provide a greater level of portability and flexibility than the original Java GUI components from package ***java.awt****.*

Swing components are often referred to as *lightweight components*—they are written completely in Java so they are not "weighed down" by the complex GUI capabilities of the platform on which they are used. AWT components (many of which parallel the Swing components) that are tied to the local platform are correspondingly called *heavyweight components*—they rely on the local platform's *windowing system* to determine their functionality and their look and feel. Each heavyweight component has a *peer* (from package ***java.awt.peer***) that is responsible for the interactions between the component and the local platform that display and manipulate the component. Several Swing components are still heavyweight components. In particular, subclasses of ***java.awt.Window*** (such as **JFrame** used in several previous chapters) that display windows on the screen and subclasses of **java.applet.Applet** (such as **JApplet**) still require direct interaction with the local windowing system. As such, heavyweight Swing GUI components are less flexible than many of the lightweight components we will demonstrate.

Portability Tip 12.1

The look of a GUI defined with heavyweight GUI components from package ***java.awt*** *may vary across platforms. Heavyweight components "tie" into the "local" platform GUI, which varies from platform to platform.*

Figure 12.3 shows an inheritance hierarchy of the classes that define attributes and behaviors that are common to most Swing components. Each class is displayed with its

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 650 Saturday, July 7, 2001 2:15 PM

fully qualified package name and class name. Much of each GUI component's functionality is derived from these classes. A class that inherits from the ***Component*** class *is a* **Component**. For example, class ***Container*** inherits from class **Component**, and class **Component** inherits from **Object**. Thus, a **Container** *is a* **Component** and *is an* **Object**, and a **Component** *is an* **Object**. A class that inherits from class **Container** *is a* ***Container***. Thus, a ***JComponent*** *is a* **Container**.

Software Engineering Observation 12.1

To use GUI components effectively, the ***javax.swing*** *and* ***java.awt*** *inheritance hierarchies must be understood—especially class* ***Component****, class* ***Container*** *and class* ***JComponent****, which define features common to most Swing components.*

Class **Component** defines the common attributes and behaviors of all subclasses of **Component**. With few exceptions, most GUI components extend class **Component** directly or indirectly. One method that originates in class **Component** that has been used frequently to this point is **paint**. Other methods discussed previously that originated in Component are **repaint** and **update**. It is important to understand the methods of class **Component** because much of the functionality inherited by every subclass of **Component** is defined by the **Component** class originally. Operations common to most GUI components (both Swing and AWT) are found in class **Component**.

Good Programming Practice 12.1

Study the methods of class ***Component*** *in the Java 2 SDK on-line documentation to learn the capabilities common to most GUI components.*

A **Container** is a collection of related components. In applications with **JFrame**s and in applets, we attach components to the content pane, which is an object of class **Container**. Class **Container** defines the common attributes and behaviors for all subclasses of **Container**. One method that originates in class **Container** is **add** for adding components to a **Container**. Another method that originates in class **Container** is **setLayout**, which enables a program to specify the layout manager that helps a **Container** position and size its components.

Good Programming Practice 12.2

Study the methods of class ***Container*** *in the Java 2 SDK on-line documentation to learn the capabilities common to every container for GUI components.*

Class **JComponent** is the superclass to most Swing components. This class defines the common attributes and behaviors of all subclasses of **JComponent**.

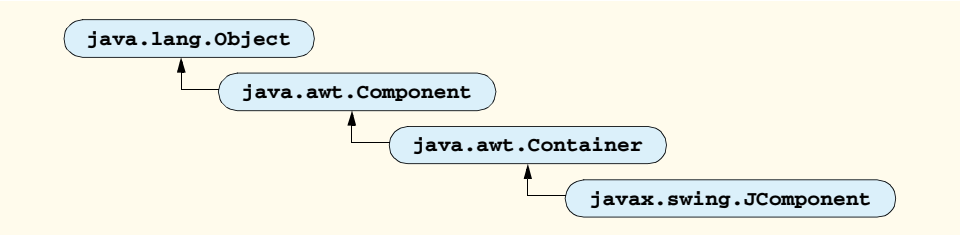


Fig. 12.3 Common superclasses of many of the Swing components.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

Good Programming Practice 12.3

Study the methods of class **`JComponent`** *in the Java 2 SDK on-line documentation to learn the capabilities common to every container for GUI components.*

Swing components that subclass **`JComponent`** have many features, including:

1. A *pluggable look and feel* that can be used to customize the look and feel when the program executes on different platforms.
2. Shortcut keys (called *mnemonics*) for direct access to GUI components through the keyboard.
3. Common event-handling capabilities for cases where several GUI components initiate the same actions in a program.
4. 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.
5. Support for assistive technologies such as braille screen readers for blind people.
6. Support for user interface *localization*—customizing the user interface for display in different languages and cultural conventions.

These are just some of the many features of the Swing components. We discuss several of these features here and in Chapter 13.

12.3 `JLabel`

Labels provide text instructions or information on a GUI. Labels are defined with class ***`JLabel`***—a subclass of **`JComponent`**. A label displays a single line of *read-only text*, an image or both text and an image. Programs rarely change a label's contents after creating it. The application of Figure 12.4 demonstrates several **`JLabel`** features.

```
// Fig. 12.4: LabelTest.java
// Demonstrating the JLabel class.

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

public class LabelTest extends JFrame {
   private JLabel label1, label2, label3;

   // set up GUI
   public LabelTest()
   {
      super( "Testing JLabel" );

```

Fig. 12.4 Demonstrating class **`JLabel`** (part 1 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 652 Saturday, July 7, 2001 2:15 PM


```
      // get content pane and set its layout
      Container container = getContentPane();
      container.setLayout( new FlowLayout() );

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

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

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

      setSize( 275, 170 );
      setVisible( true );
   }

   // execute application
   public static void main( String args[] )
   {
      LabelTest application = new LabelTest();

      application.setDefaultCloseOperation(
         JFrame.EXIT_ON_CLOSE );
   }

}  // end class LabelTest
```

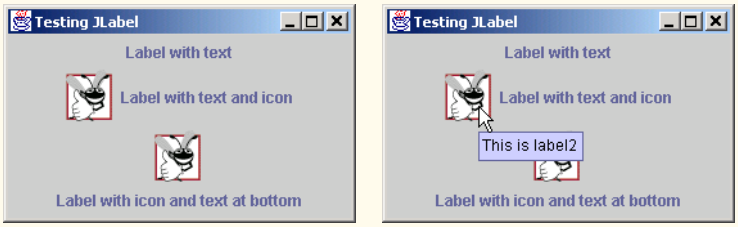


Fig. 12.4 Demonstrating class **JLabel** (part 2 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 653 Saturday, July 7, 2001 2:15 PM

Good Programming Practice 12.4

Study the methods of class ***javax.swing.JLabel*** *in the Java 2 SDK on-line documentation to learn the complete capabilities of the class before using it.*

The program declares three **JLabel**s at line 12. The **JLabel** objects are instantiated in the **LabelTest** constructor (line 15–47). Line 24 creates a **JLabel** object with the text **"Label with text"**. The label displays this text when the label appears on the screen (i.e., when the window is displayed in this program).

Line 25 uses method ***setToolTipText*** (inherited into class **JLabel** from class **JComponent**) to specify the tool tip (see the second screen capture in Fig. 12.4) that is displayed automatically when the user positions the mouse cursor over the label in the GUI. When you execute this program, try positioning the mouse over each label to see its tool tip. Line 26 adds **label1** to the content pane.

Look-and-Feel Observation 12.3

Use tool tips (set with ***JComponent*** *method* ***setToolTipText****) to add descriptive text to your GUI components. This text helps the user determine the GUI component's purpose in the user interface.*

Several Swing components can display images by specifying an ***Icon*** as an argument to their constructor or by using a method that is normally called ***setIcon***. An **Icon** is an object of any class that implements interface ***Icon*** (package **javax.swing**). One such class is ***ImageIcon*** (package **javax.swing**), which supports several image formats, including *Graphics Interchange Format (GIF)*, *Portable Network Graphics (PNG)* and *Joint Photographic Experts Group (JPEG)*. File names for each of these types typically end with **.gif**, **.png** or **.jpg** (or **.jpeg**), respectively. We discuss images in more detail in Chapter 18, Multimedia. Line 30 defines an **ImageIcon** object. The file **bug1.gif** contains the image to load and store in the **ImageIcon** object. This file is assumed to be in the same directory as the program (we will discuss locating the file elsewhere in Chapter 18). The **ImageIcon** object is assigned to **Icon** reference **bug**. Remember, class **ImageIcon** implements interface **Icon**, therefore an **ImageIcon** *is an* **Icon**.

Class **JLabel** supports the display of **Icon**s. Lines 31–32 use another **JLabel** constructor to create a label that displays the text **"Label with text and icon"** and the **Icon** to which **bug** refers and is *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**) defines 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 label can be set with methods ***setHorizontalAlignment*** and ***setVerticalAlignment***, respectively. Line 33 specifies the tool tip text for **label2**. Line 34 adds **label2** to the content pane.

Common Programming Error 12.1

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

Common Programming Error 12.2

Adding to a container a component that has not been instantiated throws a ***NullPointerException****.*

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 654 Saturday, July 7, 2001 2:15 PM

Class **JLabel** provides many methods to configure a label after it has been instantiated. Line 37 creates a **JLabel** and invokes the no-argument (default constructor). Such a label has no text or **Icon**. Line 38 uses **JLabel** method ***setText*** to set the text displayed on the label. A corresponding method ***getText*** retrieves the current text displayed on a label. Line 39 uses **JLabel** method ***setIcon*** to set the **Icon** displayed on the label. A corresponding method ***getIcon*** retrieves the current **Icon** displayed on a label. Lines 40–41 use **JLabel** methods ***setHorizontalTextPosition*** and ***setVerticalTextPosition*** to specify the text position in the label. In this case, the text will be centered horizontally and will appear at the bottom of the label. Thus, the **Icon** will appear above the text. Line 42 sets the tool tip text for the **label3**. Line 43 adds **label3** to the content pane.

12.4 Event-Handling Model

In the preceding section, we did not discuss event handling because there are no specific events for **JLabel** objects. GUIs are *event driven* (i.e., they generate *events* when the user of the program interacts with the GUI). Some common interactions are moving the mouse, clicking the mouse, clicking a button, typing in a text field, selecting an item from a menu, closing a window, etc. When a user interaction occurs, an event is sent to the program. GUI event information is stored in an object of a class that extends **AWTEvent**. Figure 12.5 illustrates a hierarchy containing many of the event classes we use from package **java.awt.event**. Many of these event classes are discussed throughout this chapter and Chapter 13. The event types from package ***java.awt.event*** are used with both AWT and Swing components. Additional event types have also been added that are specific to several types of Swing components. New Swing-component event types are defined in package ***javax.swing.event***.

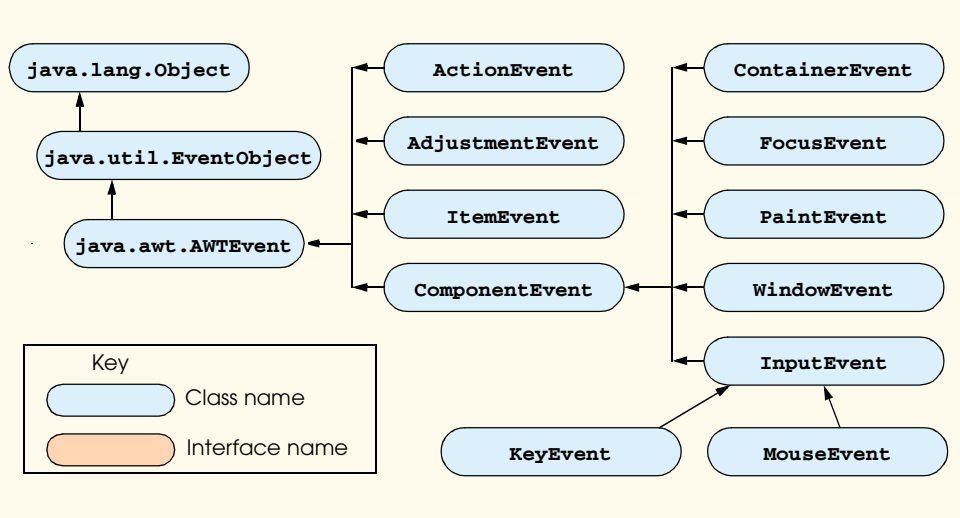


Fig. 12.5 Some event classes of package **java.awt.event**.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 655 Saturday, July 7, 2001 2:15 PM

There are three parts to the event-handling mechanism—the *event source*, the *event object* and the *event listener*. The event source is the particular GUI component with which the user interacts. The event object encapsulates information about the event that occurred. This information includes a reference to the event source and any event-specific information that may be required by the event listener to handle the event. The event listener is an object that is notified by the event source when an event occurs. The event listener receives an event object when it is notified of the event, then uses the object to respond to the event. The event source is required to provide methods that enable listeners to be registered and unregistered. The event source also is required to maintain a list of its registered listeners and be able to notify its listeners when an event occurs.

The programmer must perform two key tasks to process a graphical user interface event in a program—register an *event listener* for the GUI component that is expected to generate the event, and implement an *event handling method* (or set of event-handling methods). Commonly, event-handling methods are called *event handlers*. An event listener for a GUI event is an object of a class that implements one or more of the event-listener interfaces from package **`java.awt.event`** and package **`javax.swing.event`**. Many of the event-listener types are common to both Swing and AWT components. Such types are defined in package **`java.awt.event`**, and many of these are shown in Fig. 12.6. Additional event-listener types that are specific to Swing components are defined in package **`javax.swing.event`**.

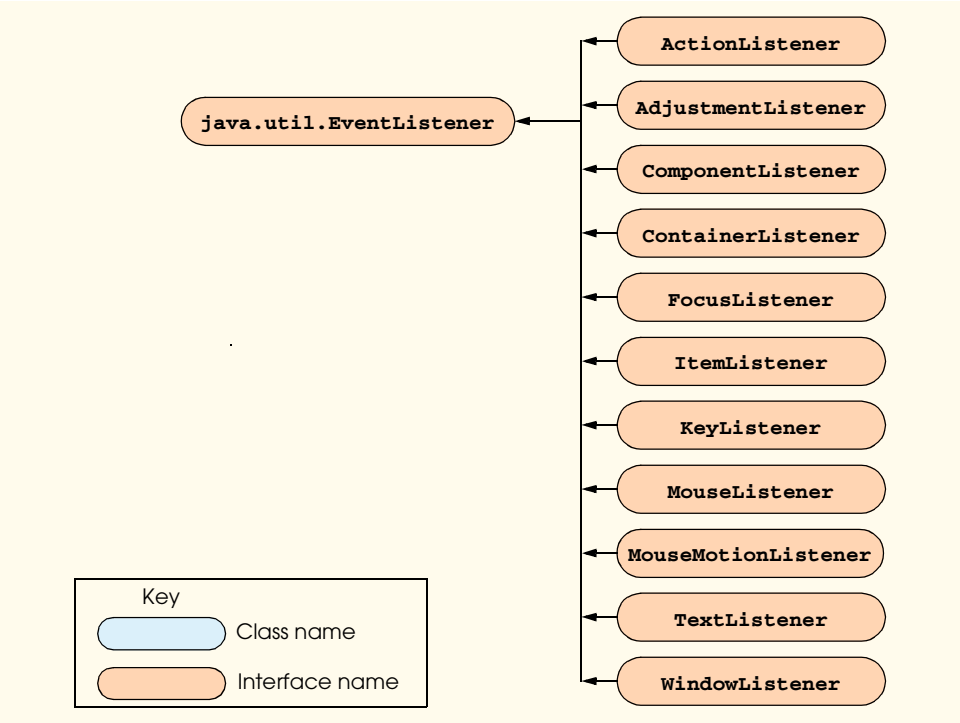


Fig. 12.6 Event-listener interfaces of package **`java.awt.event`**.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 656 Saturday, July 7, 2001 2:15 PM

An event listener object "listens" for specific types of events generated by event sources (normally GUI components) in a program. An event handler is a method that is called in response to a particular type of event. Each event-listener interface specifies one or more event-handling methods that *must* be defined in the class that implements the event-listener interface. Remember that interfaces define **abstract** methods. Any class that implements an interface must define all the methods of that interface; otherwise, the class is an **abstract** class and cannot be used to create objects. The use of event listeners in event handling is known as the *delegation event model*—the processing of an event is delegated to a particular object (the listener) in the program.

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** as opposed to some other event handling method? We answer these questions and diagram the interaction as part of the next example.

12.5 JTextField and JPasswordField

JTextFields and ***JPasswordFields*** (package **javax.swing**) are single-line areas in which text can be entered by the user from the keyboard or text can simply be displayed. A **JPasswordField** shows that a character was typed as the user enters characters, but hides the characters, assuming that they represent a password that should remain known only to the user. When the user types data into a **JTextField** or **JPasswordField** and presses the *Enter* key, an action event occurs. If the program registers an event listener, the listener processes the event and can use the data in the **JTextField** or **JPasswordField** at the time of the event in the program. 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 several methods that are specific to processing passwords.

Common Programming Error 12.3

*Using a lowercase **f** in the class names **JTextField** or **JPasswordField** is a syntax error.*

The application of Fig. 12.7 uses classes **JTextField** and **JPasswordField** to create and manipulate four fields. When the user presses *Enter* in the currently active field (the currently active component "has the *focus*"), a message dialog box containing the text in the field is displayed. When an event occurs in the **JPasswordField**, the password is revealed.

```
// Fig. 12.7: TextFieldTest.java
// Demonstrating the JTextField class.

// Java core packages
import java.awt.*;
import java.awt.event.*;
```

Fig. 12.7 Demonstrating **JTextFields** and **JPasswordFields** (part 1 of 4).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 657 Saturday, July 7, 2001 2:15 PM


```

// Java extension packages
import javax.swing.*;

public class TextFieldTest extends JFrame {
   private JTextField textField1, textField2, textField3;
   private JPasswordField passwordField;

   // set up GUI
   public TextFieldTest()
   {
      super( "Testing JTextField and JPasswordField" );

      Container container = getContentPane();
      container.setLayout( new FlowLayout() );

      // construct textfield with default sizing
      textField1 = new JTextField( 10 );
      container.add( textField1 );

      // construct textfield with default text
      textField2 = new JTextField( "Enter text here" );
      container.add( textField2 );

      // construct textfield with default text and
      // 20 visible elements and no event handler
      textField3 = new JTextField( "Uneditable text field", 20 );
      textField3.setEditable( false );
      container.add( textField3 );

      // construct textfield with default text
      passwordField = new JPasswordField( "Hidden text" );
      container.add( passwordField );

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

      setSize( 325, 100 );
      setVisible( true );
   }

   // execute application
   public static void main( String args[] )
   {
      TextFieldTest application = new TextFieldTest();

      application.setDefaultCloseOperation(
         JFrame.EXIT_ON_CLOSE );
   }
```

Fig. 12.7 Demonstrating **JTextField**s and **JPasswordField**s (part 2 of 4).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 658 Saturday, July 7, 2001 2:15 PM


```

   // private inner class for event handling
   private class TextFieldHandler implements ActionListener {

      // process text field events
      public void actionPerformed( ActionEvent event )
      {
         String string = "";

         // user pressed Enter in JTextField textField1
         if ( event.getSource() == textField1 )
            string = "textField1: " + event.getActionCommand();

         // user pressed Enter in JTextField textField2
         else if ( event.getSource() == textField2 )
            string = "textField2: " + event.getActionCommand();

         // user pressed Enter in JTextField textField3
         else if ( event.getSource() == textField3 )
            string = "textField3: " + event.getActionCommand();

         // user pressed Enter in JTextField passwordField
         else if ( event.getSource() == passwordField ) {
            JPasswordField pwd =
               ( JPasswordField ) event.getSource();
            string = "passwordField: " +
                new String( passwordField.getPassword() );
         }

         JOptionPane.showMessageDialog( null, string );
      }

   }  // end private inner class TextFieldHandler

}  // end class TextFieldTest
```

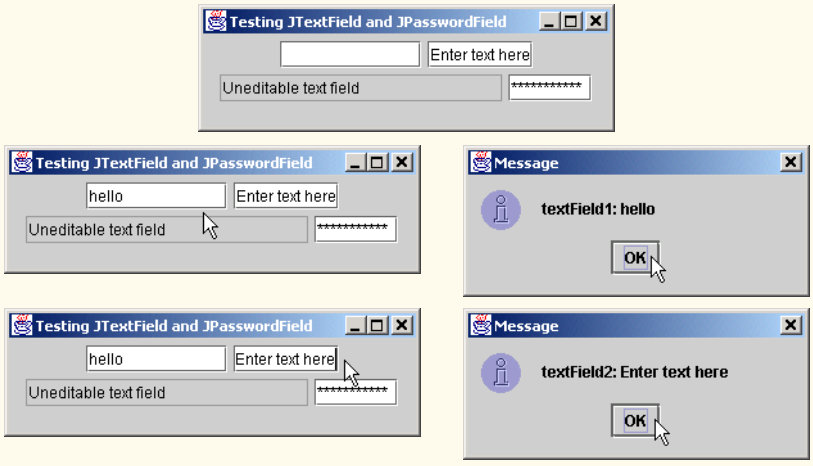


Fig. 12.7 Demonstrating **JTextFields** and **JPasswordFields** (part 3 of 4).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 659 Saturday, July 7, 2001 2:15 PM

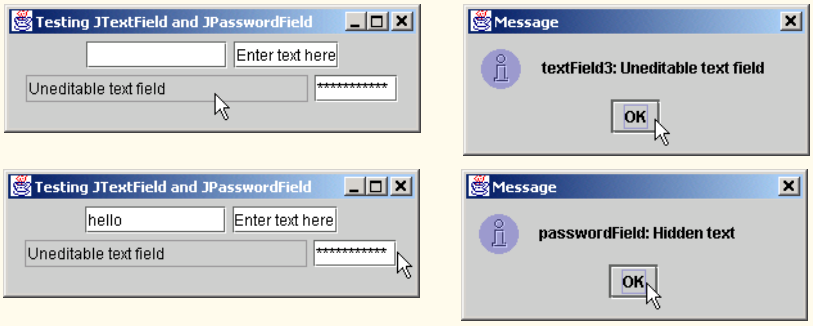


Fig. 12.7 Demonstrating **JTextFields** and **JPasswordFields** (part 4 of 4).

Lines 12–13 declare three references for **JTextFields** (**textField1**, **textField2** and **textField3**) and a **JPasswordField** (**passwordField**). Each of these is instantiated in the constructor (line 16–50). Line 24 defines **JTextField textField1** with **10** columns of text. The width of the text field will be the width in pixels of the average character in the text field's current font multiplied by 10. Line 25 adds **textField1** to the content pane.

Line 28 defines **textField2** with the initial text **"Enter text here"** to display in the text field. The width of the text field is determined by the text. Line 29 adds **textField2** to the content pane.

Line 33 defines **textField3** and call the **JTextField** constructor with two arguments—the default text **"Uneditable text field"** to display in the text field and the number of columns (**20**). The width of the text field is determined by the number of columns specified. Line 34 uses method ***setEditable*** (inherited into **JTextField** from class **JTextComponent**) to indicate that the user cannot modify the text in the text field. Line 35 adds **textField3** to the content pane.

Line 38 defines **JPasswordField passwordField** with the text **"Hidden text"** to display in the text field. The width of the text field is determined by the text. Notice that the text is displayed as a string of asterisks when the program executes. Line 39 adds **passwordField** to the content pane.

For the event-handling in this example, we defined inner class **TextFieldHandler** (lines 62–92), which implements interface **ActionListener** (class **TextFieldHandler** is discussed shortly). Thus, every instance of class **TextFieldHandler** *is an* **ActionListener**. Line 42 defines an instance of class **TextFieldHandler** and assigns it to reference **handler**. This one instance will be used as the event-listener object for the **JTextFields** and the **JPasswordField** in this example.

Lines 43–46 are the event registration statements that specify the event listener object for each of the three **JTextFields** and for the **JPasswordField**. After these statements execute, the object to which **handler** refers is *listening for events* (i.e., it will be notified when an event occurs) on these four objects. The program calls **JTextField** method **addActionListener** to register the event for each component. Method **addActionListener** receives as its argument an **ActionListener** object. Thus, any object of a class that implements interface **ActionListener** (i.e., any object that *is an* **Action-**

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 660 Saturday, July 7, 2001 2:15 PM

Listener) can be supplied as an argument to this method. The object to which **handler** refers *is an* **ActionListener** because its class implements interface **ActionListener**. Now, when the user presses *Enter* in any of these four fields, method **actionPerformed** (line 65–90) in class **TextFieldHandler** is called to handle the event.

Software Engineering Observation 12.2

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

Method **actionPerformed** uses its **ActionEvent** argument's method **getSource** to determine the GUI component with which the user interacted and creates a **String** to display in a message dialog box. **ActionEvent** method **getActionCommand** returns the text in the **JTextField** that generated the event. If the user interacted with the **JPasswordField**, lines 83–84 cast the **Component** reference returned by **event.getSource()** to a **JPasswordField** reference so that lines 85–86 can use **JPasswordField** method ***getPassword*** to obtain the password and create the **String** to display. Method **getPassword** returns the password as an array of type **char** that is used as an argument to a **String** constructor to create a **String**. Line 89 displays a message box indicating the GUI component reference name and the text the user typed in the field.

Note that even an uneditable **JTextField** can generate an event. Simply click the text field, then press *Enter*. Also note that the actual text of the password is displayed when you press *Enter* in the **JPasswordField** (of course, you would normally not do this!).

Common Programming Error 12.4

Forgetting to register an event handler object for a particular GUI component's event type results in no events being handled for that component for that event type.

Using a separate class to define an event listener is a common programming practice for separating the GUI interface from the implementation of its event handler. For the remainder of this chapter and Chapter 13, many programs use separate event-listener classes to process GUI events.

12.5.1 How Event Handling Works

Let us illustrate how the event-handling mechanism works using **textField1** from the preceding example. We have two remaining open questions from Section 12.4:

1. How did the event handler get registered?
2. How does the GUI component know to call **actionPerformed** as opposed to some other event handling method?

The first question is answered by the event registration performed in lines 43–46 of the program. Figure 12.8 diagrams **JTextField** reference **textField1**, the **JTextField** object to which it refers and the listener object that is registered to handle the **JTextField**'s event.

Every **JComponent** has an object of class ***EventListenerList*** (package **javax.swing.event**) called ***listenerList*** as an instance variable. All registered listeners are stored in the **listenerList** (diagramed as an array in Figure 12.8). When the statement

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 661 Saturday, July 7, 2001 2:15 PM

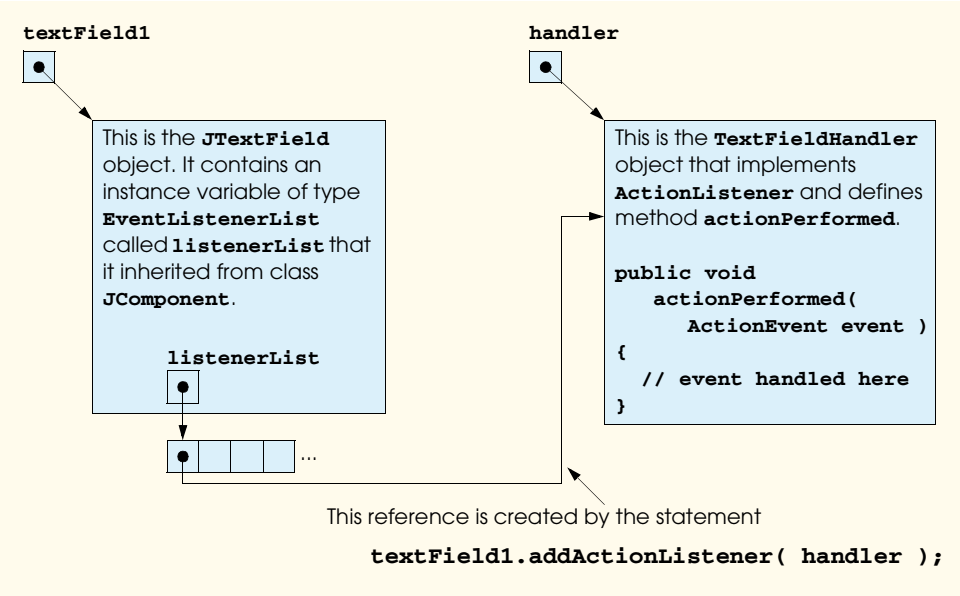


Fig. 12.8 Event registration for **JTextField textField1**.

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

executes in Fig. 12.7, a new entry is placed in the **listenerList** for **JTextField textField1**, indicating both the reference to the listener object and the type of listener (in this case **ActionListener**).

The type is important in answering the second question—how does the GUI component know to call **actionPerformed** rather than another event handling method? Every **JComponent** actually supports several different 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. The dispatching of an event is simply calling the event handling method for each registered listener for that event type.

Each event type has a corresponding event-listener interface. For example, **ActionEvent**s are handled by **ActionListener**s, ***MouseEvent***s are handled by ***MouseListener***s (and ***MouseMotionListener***s as we will see) and ***KeyEvent***s are handled by ***KeyListener***s. When an event is generated by a user interaction with a component, the component is handed a unique *event ID* specifying the event type that occurred. The GUI component uses the event ID to decide the type of listener to which the event should be dispatched and the method to call. In the case of an **ActionEvent**, the event is dispatched to every registered **ActionListener**'s **actionPerformed** method (the only method in interface **ActionListener**). In the case of a **MouseEvent**, the event is dispatched to every registered **MouseListener** (or **MouseMotionListener**, depending on the event that occurs). The event ID of the **MouseEvent** determines which of the seven different mouse event handling methods are called. All of this decision logic is handled for you by the GUI components. We discuss other event types and event-listener interfaces as they are needed with each new component we cover.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 662 Saturday, July 7, 2001 2:15 PM

12.6 JButton

A *button* is a component the user clicks to trigger a specific action. A Java program can use several types of buttons, including *command buttons*, *check boxes*, *toggle buttons* and *radio buttons*. Figure 12.9 shows the inheritance hierarchy of the Swing buttons we cover in this chapter. As you can see in the diagram, all the button types are subclasses of ***AbstractButton*** (package **javax.swing**), which defines many of the features that are common to Swing buttons. In this section, we concentrate on buttons that are typically used to initiate a command. Other button types are covered in the next several sections.

A command button generates an **ActionEvent** when the user clicks the button with the mouse. Command buttons are created with class ***JButton***, which inherits from class **AbstractButton**. The text on the face of a **JButton** is called a *button label*. A GUI can have many **JButton**s, but each button label typically should be unique.

Look-and-Feel Observation 12.4

*Having more than one **JButton** with the same label makes the **JButton**s ambiguous to the user. Be sure to provide a unique label for each button.*

The application of Fig. 12.10 creates two **JButton**s and demonstrates that **JButton**s (like **JLabel**s) support the display of **Icon**s. Event handling for the buttons is performed by a single instance of inner class **ButtonHandler** (defined at lines 53–62).

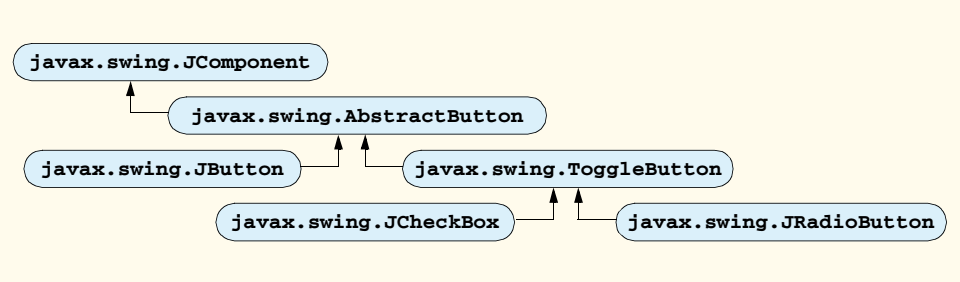


Fig. 12.9 The button hierarchy.

```
1  // Fig. 12.10: ButtonTest.java
2  // Creating JButtons.
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class ButtonTest extends JFrame {
12    private JButton plainButton, fancyButton;
```

Fig. 12.10 Demonstrating command buttons and action events (part 1 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 663 Saturday, July 7, 2001 2:15 PM


```

   // set up GUI
   public ButtonTest()
   {
      super( "Testing Buttons" );

      // get content pane and set its layout
      Container container = getContentPane();
      container.setLayout( new FlowLayout() );

      // create buttons
      plainButton = new JButton( "Plain Button" );
      container.add( plainButton );

      Icon bug1 = new ImageIcon( "bug1.gif" );
      Icon bug2 = new ImageIcon( "bug2.gif" );
      fancyButton = new JButton( "Fancy Button", bug1 );
      fancyButton.setRolloverIcon( bug2 );
      container.add( fancyButton );

      // create an instance of inner class ButtonHandler
      // to use for button event handling
      ButtonHandler handler = new ButtonHandler();
      fancyButton.addActionListener( handler );
      plainButton.addActionListener( handler );

      setSize( 275, 100 );
      setVisible( true );
   }

   // execute application
   public static void main( String args[] )
   {
      ButtonTest application = new ButtonTest();

      application.setDefaultCloseOperation(
         JFrame.EXIT_ON_CLOSE );
   }

   // inner class for button event handling
   private class ButtonHandler implements ActionListener {

      // handle button event
      public void actionPerformed( ActionEvent event )
      {
         JOptionPane.showMessageDialog( null,
            "You pressed: " + event.getActionCommand() );
      }

   }  // end private inner class ButtonHandler

}  // end class ButtonTest
```

Fig. 12.10 Demonstrating command buttons and action events (part 2 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 664 Saturday, July 7, 2001 2:15 PM

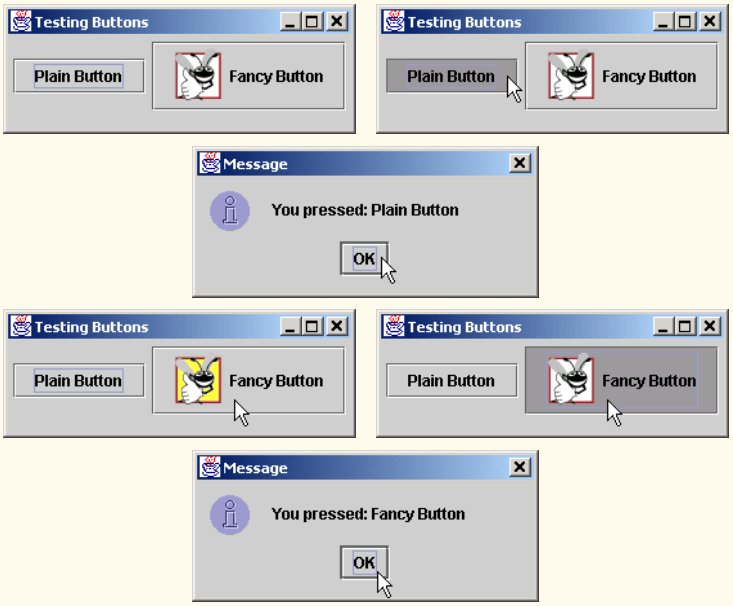


Fig. 12.10 Demonstrating command buttons and action events (part 3 of 3).

Line 12 declares two references to instances of class **`JButton`**—**`plainButton`** and **`fancyButton`**—that are instantiated in the constructor.

Line 24 creates **`plainButton`** with the button label **`"Plain Button"`**. Line 25 adds the button to the content pane.

A **`JButton`** can display **`Icon`**s. To provide the user with an extra level of visual interactivity with the GUI, a **`JButton`** can also have a *rollover* **`Icon`**—an **`Icon`** that is displayed when the mouse is positioned over the button. The icon on the button changes as the mouse moves in and out of the button's area on the screen. Lines 27–28 create two **`ImageIcon`** objects that represent the default **`Icon`** and rollover **`Icon`** for the **`JButton`** created at line 29. Both statements assume the image files are stored in the same directory as the program (this is commonly the case for applications that use images).

Line 29 creates **`fancyButton`** with default text **`"Fancy Button"`** and the **`Icon`** **`bug1`**. By default, the text is displayed to the right of the icon. Line 30 uses method ***`setRolloverIcon`*** (inherited from class **`AbstractButton`** into class **`JButton`**) to specify the image displayed on the button when the user positions the mouse over the button. Line 31 adds the button to the content pane.

Look-and-Feel Observation 12.5

*Using rollover icons for **`JButton`**s provides the user with visual feedback indicating that if they click the mouse, the button's action will occur.*

`JButton`s (like **`JTextField`**s) generate **`ActionEvent`**s. As mentioned previously, an **`ActionEvent`** can be processed by any **`ActionListener`** object. Lines 35–37 register an **`ActionListener`** object for each **`JButton`**. Inner class **`ButtonHandler`** (lines 53–62) defines **`actionPerformed`** to display a message dialog box con-

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 665 Saturday, July 7, 2001 2:15 PM

taining the label for the button that was pressed by the user. **ActionEvent** method **getActionCommand** returns the label on the button that generated the event.

12.7 JCheckBox and JRadioButton

The Swing GUI components contain three types of *state buttons*—***JToggleButton***, ***JCheckBox*** and ***JRadioButton***—that have on/off or true/false values. **JToggleButton**s are frequently used with *toolbars* (sets of small buttons typically located on a bar across the top of a window) and are covered in Chapter 13. Classes **JCheckBox** and **JRadioButton** are subclasses of **JToggleButton**. A **JRadioButton** is different from a **JCheckBox** in that there are normally several **JRadioButton**s that are grouped together and only one of the **JRadioButton**s in the group can be selected (true) at any time. We first discuss class **JCheckBox**.

Look-and-Feel Observation 12.6

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

The application of Fig. 12.11 uses two **JCheckBox** objects to change the font style of the text displayed in a **JTextField**. One **JCheckBox** applies a bold style when selected and the other applies an italic style when selected. If both are selected, the style of the font is bold and italic. When the program initially executes, neither **JCheckBox** is checked (**true**).

```
// Fig. 12.11: CheckBoxTest.java
// Creating Checkbox buttons.

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

public class CheckBoxTest extends JFrame {
   private JTextField field;
   private JCheckBox bold, italic;

   // set up GUI
   public CheckBoxTest()
   {
      super( "JCheckBox Test" );

      // get content pane and set its layout
      Container container = getContentPane();
      container.setLayout( new FlowLayout() );

      // set up JTextField and set its font
      field =
         new JTextField( "Watch the font style change", 20 );
```

Fig. 12.11 Program that creates two **JCheckBox** buttons (part 1 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 666 Saturday, July 7, 2001 2:15 PM


```
      field.setFont( new Font( "Serif", Font.PLAIN, 14 ) );
      container.add( field );

      // create checkbox objects
      bold = new JCheckBox( "Bold" );
      container.add( bold );

      italic = new JCheckBox( "Italic" );
      container.add( italic );

      // register listeners for JCheckBoxes
      CheckBoxHandler handler = new CheckBoxHandler();
      bold.addItemListener( handler );
      italic.addItemListener( handler );

      setSize( 275, 100 );
      setVisible( true );
   }

   // execute application
   public static void main( String args[] )
   {
      CheckBoxTest application = new CheckBoxTest();

      application.setDefaultCloseOperation(
         JFrame.EXIT_ON_CLOSE );
   }

   // private inner class for ItemListener event handling
   private class CheckBoxHandler implements ItemListener {
      private int valBold = Font.PLAIN;
      private int valItalic = Font.PLAIN;

      // respond to checkbox events
      public void itemStateChanged( ItemEvent event )
      {
         // process bold checkbox events
         if ( event.getSource() == bold )

            if ( event.getStateChange() == ItemEvent.SELECTED )
               valBold = Font.BOLD;
            else
               valBold = Font.PLAIN;

         // process italic checkbox events
         if ( event.getSource() == italic )

            if ( event.getStateChange() == ItemEvent.SELECTED )
               valItalic = Font.ITALIC;
            else
               valItalic = Font.PLAIN;

```

Fig. 12.11 Program that creates two **JCheckBox** buttons (part 2 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 667 Saturday, July 7, 2001 2:15 PM


```
79            // set text field font
80            field.setFont(
81               new Font( "Serif", valBold + valItalic, 14 ) );
82         }
83
84      }  // end private inner class CheckBoxHandler
85
86   }  // end class CheckBoxTest
```

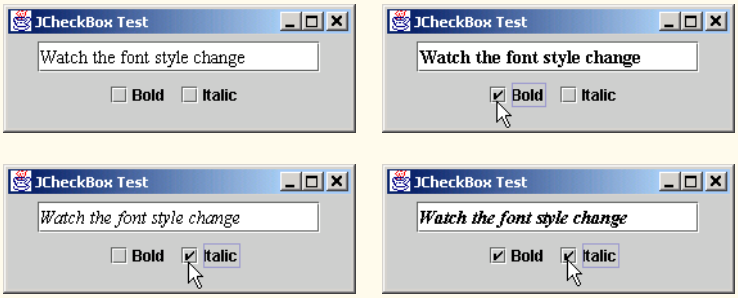


Fig. 12.11 Program that creates two **JCheckBox** buttons (part 3 of 3).

After the **JTextField** is created and initialized, line 27 sets the font of the **JTextField** to **Serif**, **PLAIN** style and **14**-point size. Next, the constructor creates two **JCheckBox** objects with lines 31 and 34. The **String** passed to the constructor is the *check box label* that appears to the right of the **JCheckBox** by default.

When the user clicks a **JCheckBox**, an **ItemEvent** occurs that can be handled by an **ItemListener** (any object of a class that implements interface **ItemListener**). An **ItemListener** must define method ***itemStateChanged***. In this example, the event handling is performed by an instance of inner class **CheckBoxHandler** (lines 56–84). Lines 38–40 create an instance of class **CheckBoxHandler** and register it with method ***addItemListener*** as the **ItemListener** for both the **bold** and **italic** **JCheckBox**es.

Method **itemStateChanged** (lines 61–82) is called when the user clicks either the **bold** or the **italic** checkbox. The method uses **event.getSource()** to determine which **JCheckBox** was clicked. If it was **JCheckBox bold**, the **if/else** structure at lines 66–69 uses **ItemEvent** method ***getStateChange*** to determine the state of the button (**ItemEvent.SELECTED** or **ItemEvent.DESELECTED**). If the state is selected, integer **valBold** is assigned **Font.BOLD**; otherwise, **valBold** is assigned **Font.PLAIN**. A similar **if/else** structure is executed if **JCheckBox italic** is clicked. If the **italic** state is selected, integer **valItalic** is assigned **Font.ITALIC**; otherwise, **valItalic** is assigned **Font.PLAIN**. The sum of **valBold** and **valItalic** is used at lines 80–81 as the style of the new font for the **JTextField**.

Radio buttons (defined with class ***JRadioButton***) are similar to check boxes 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 radio button can be selected at a time. Selecting a different radio button in the group automatically forces all other radio buttons in the group to be deselected. Radio buttons are used to represent a set of *mutually exclusive*

© Copyright 1992-2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

options (i.e., multiple options in the group would not be selected at the same time). The logical relationship between radio buttons is maintained by a ***ButtonGroup*** object (package **javax.swing**). The **ButtonGroup** object itself is not a GUI component. Therefore, a **ButtonGroup** object is not displayed in a user interface. Rather, the individual **JRadioButton** objects from the group are displayed in the GUI.

Common Programming Error 12.5

*Adding a **ButtonGroup** object (or an object of any other class that does not derive from **Component**) to a container is a syntax error.*

The application of Fig. 12.12 is similar to the preceding program. The user can alter the font style of a **JTextField**'s text. The program uses radio buttons that permit only a single font style in the group to be selected at a time.

```
// Fig. 12.12: RadioButtonTest.java
// Creating radio buttons using ButtonGroup and JRadioButton.

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

public class RadioButtonTest extends JFrame {
   private JTextField field;
   private Font plainFont, boldFont, italicFont, boldItalicFont;
   private JRadioButton plainButton, boldButton, italicButton,
      boldItalicButton;
   private ButtonGroup radioGroup;

   // create GUI and fonts
   public RadioButtonTest()
   {
      super( "RadioButton Test" );

      // get content pane and set its layout
      Container container = getContentPane();
      container.setLayout( new FlowLayout() );

      // set up JTextField
      field =
         new JTextField( "Watch the font style change", 25 );
      container.add( field );

      // create radio buttons
      plainButton = new JRadioButton( "Plain", true );
      container.add( plainButton );

      boldButton = new JRadioButton( "Bold", false);
      container.add( boldButton );
```

Fig. 12.12 Creating and manipulating radio button (part 1 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

```

      italicButton = new JRadioButton( "Italic", false );
      container.add( italicButton );

      boldItalicButton = new JRadioButton(
         "Bold/Italic", false );
      container.add( boldItalicButton );

      // register events for JRadioButtons
      RadioButtonHandler handler = new RadioButtonHandler();
      plainButton.addItemListener( handler );
      boldButton.addItemListener( handler );
      italicButton.addItemListener( handler );
      boldItalicButton.addItemListener( handler );

      // create logical relationship between JRadioButtons
      radioGroup = new ButtonGroup();
      radioGroup.add( plainButton );
      radioGroup.add( boldButton );
      radioGroup.add( italicButton );
      radioGroup.add( boldItalicButton );

      // 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 );
      field.setFont( plainFont );

      setSize( 300, 100 );
      setVisible( true );
   }

   // execute application
   public static void main( String args[] )
   {
      RadioButtonTest application = new RadioButtonTest();

      application.setDefaultCloseOperation(
         JFrame.EXIT_ON_CLOSE );
   }

   // private inner class to handle radio button events
   private class RadioButtonHandler implements ItemListener {

      // handle radio button events
      public void itemStateChanged( ItemEvent event )
      {
         // user clicked plainButton
         if ( event.getSource() == plainButton )
            field.setFont( plainFont );

```

Fig. 12.12 Creating and manipulating radio button (part 2 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 670 Saturday, July 7, 2001 2:15 PM


```
91            // user clicked boldButton
92            else if ( event.getSource() == boldButton )
93               field.setFont( boldFont );
94
95            // user clicked italicButton
96            else if ( event.getSource() == italicButton )
97               field.setFont( italicFont );
98
99            // user clicked boldItalicButton
100           else if ( event.getSource() == boldItalicButton )
101              field.setFont( boldItalicFont );
102        }
103
104     }  // end private inner class RadioButtonHandler
105
106  }  // end class RadioButtonTest
```

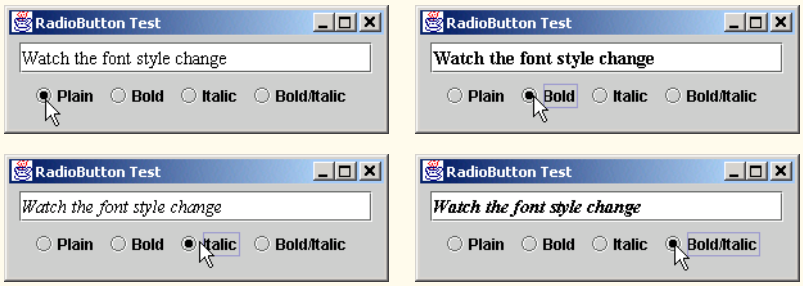


Fig. 12.12 Creating and manipulating radio button (part 3 of 3).

Lines 33–44 in the constructor define each **JRadioButton** object and add it to the application window's content pane. Each **JRadioButton** is initialized with a constructor call like line 33. This constructor supplies 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 is displayed.

JRadioButtons, like **JCheckBox**es, generate **ItemEvent**s when they are clicked. Lines 47–51 create an instance of inner class **RadioButtonHandler** (defined at lines 82–104) and register it to handle the **ItemEvent** generated when the user clicks any one of the **JRadioButton**s.

Line 54 instantiates a **ButtonGroup** object and assigns it to reference **radioGroup**. This object is the "glue" that binds the four **JRadioButton** objects together to form the logical relationship that allows only one of the four buttons to be selected at a time. Lines 55–58 use **ButtonGroup** method ***add*** to associate each of the **JRadioButton**s with **radioGroup**. If more than one selected **JRadioButton** object is added to the group, the first selected **JRadioButton** added will be selected when the GUI is displayed.

Class **RadioButtonHandler** (line 82–104) implements interface **ItemListener** so it can handle item events generated by the **JRadioButton**s. Each **JRadioButton** in the program has an instance of this class (**handler**) registered as its **ItemListener**. When the user clicks a **JRadioButton**, **radioGroup** turns off the previously selected **JRadioButton** and method **itemStateChanged** (line 85–102)

© Copyright 1992-2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 671 Saturday, July 7, 2001 2:15 PM

executes. The method determines which **JRadioButton** was clicked using method **getSource** (inherited indirectly from **EventObject** into **ItemEvent**), then sets the font in the **JTextField** to one of the **Font** objects created in the constructor.

12.8 JComboBox

A *combo box* (sometimes called a *drop-down list*) provides a list of items from which the user can make a selection. Combo boxes are implemented with class ***JComboBox***, which inherits from class **JComponent**. **JComboBox**es generate **ItemEvent**s like **JCheckBox**es and **JRadioButton**s.

The application of Fig. 12.13 uses a **JComboBox** to provide a list of four image file names. When an image file name is selected, the corresponding image is displayed as an **Icon** on a **JLabel**. The screen captures for this program show the **JComboBox** list after the selection was made to illustrate which image file name was selected.

Lines 17–19 declare and initialize array icons with four new **ImageIcon** objects. **String** array **names** (defined on lines 15–16) contains the names of the four image files that are stored in the same directory as the application.

Line 31 creates a **JComboBox** object, using the **String**s in array **names** as the elements in the list. A numeric *index* keeps track of the ordering of items in the **JComboBox**. The first item is added at index 0; the next item is added 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**. When clicked, the **JComboBox** expands into a list from which the user can make a selection.

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 more items in the **JComboBox** than the maximum number of elements that are displayed, the **JComboBox** automatically provides a *scrollbar* (see the first screen capture) that allows the user to view 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 the user can drag the *scroll box* in the middle of the scrollbar up and down to move through the list. To drag the scroll box, hold the mouse button down with the mouse cursor on the scroll box and move the mouse.

```
1  // Fig. 12.13: ComboBoxTest.java
2  // Using a JComboBox to select an image to display.
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class ComboBoxTest extends JFrame {
12    private JComboBox imagesComboBox;
13    private JLabel label;
14
```

Fig. 12.13 Program that uses a **JComboBox** to select an icon (part 1 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

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

   // set up GUI
   public ComboBoxTest()
   {
      super( "Testing JComboBox" );

      // get content pane and set its layout
      Container container = getContentPane();
      container.setLayout( new FlowLayout() );

      // set up JComboBox and register its event handler
      imagesComboBox = new JComboBox( names );
      imagesComboBox.setMaximumRowCount( 3 );

      imagesComboBox.addItemListener(

         // anonymous inner class to handle JComboBox events
         new ItemListener() {

            // handle JComboBox event
            public void itemStateChanged( ItemEvent event )
            {
               // determine whether check box selected
               if ( event.getStateChange() == ItemEvent.SELECTED )
                  label.setIcon( icons[
                     imagesComboBox.getSelectedIndex() ] );
            }

         }  // end anonymous inner class

      ); // end call to addItemListener

      container.add( imagesComboBox );

      // set up JLabel to display ImageIcons
      label = new JLabel( icons[ 0 ] );
      container.add( label );

      setSize( 350, 100 );
      setVisible( true );
   }

   // execute application
   public static void main( String args[] )
   {
      ComboBoxTest application = new ComboBoxTest();

```

Fig. 12.13 Program that uses a **JComboBox** to select an icon (part 2 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 673 Saturday, July 7, 2001 2:15 PM


```
67          application.setDefaultCloseOperation(
68             JFrame.EXIT_ON_CLOSE );
69       }
70
71    }  // end class ComboBoxTest
```

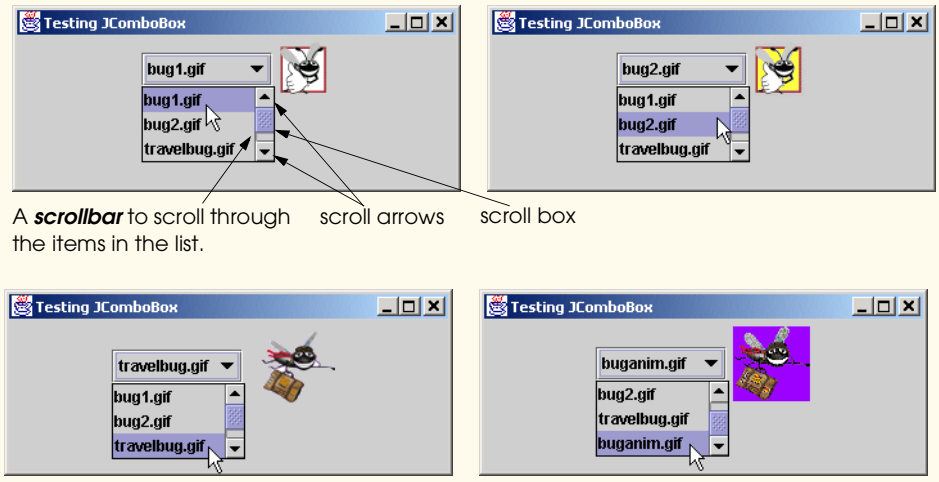


Fig. 12.13 Program that uses a **JComboBox** to select an icon (part 3 of 3).

Look-and-Feel Observation 12.7

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 or applet in which it is used. This will ensure that the list displays correctly when it is expanded by the user.*

Lines 34–50 register an instance of an anonymous inner class that implements **ItemListener** as the listener for **JComboBox images**. When the user makes a selection from **images**, method **itemStateChanged** (line 40–46) sets the **Icon** for **label**. The **Icon** is selected from array **icons** by determining the index number of the selected item in the **JComboBox** with method ***getSelectedIndex*** in line 45. Note that line 43 changes the icon only for a selected item. The reason for the **if** structure here is that for each item that is selected from a **JComboBox**, another item is deselected. Thus, two events occur for each item selected. We wish to display only the icon for the item the user just selected.

12.9 JList

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

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 674 Saturday, July 7, 2001 2:15 PM

The application of Fig. 12.14 creates a **JList** of 13 colors. When a color name is clicked in the **JList**, a ***ListSelectionEvent*** occurs and the application window content pane's background color changes.

```
// Fig. 12.14: ListTest.java
// Selecting colors from a JList.

// Java core packages
import java.awt.*;

// Java extension packages
import javax.swing.*;
import javax.swing.event.*;

public class ListTest extends JFrame {
   private JList colorList;
   private Container container;

   private String colorNames[] = { "Black", "Blue", "Cyan",
      "Dark Gray", "Gray", "Green", "Light Gray", "Magenta",
      "Orange", "Pink", "Red", "White", "Yellow" };

   private Color colors[] = { Color.black, Color.blue,
      Color.cyan, Color.darkGray, Color.gray, Color.green,
      Color.lightGray, Color.magenta, Color.orange, Color.pink,
      Color.red, Color.white, Color.yellow };

   // set up GUI
   public ListTest()
   {
      super( "List Test" );

      // get content pane and set its layout
      container = getContentPane();
      container.setLayout( new FlowLayout() );

      // create a list with items in colorNames array
      colorList = new JList( colorNames );
      colorList.setVisibleRowCount( 5 );

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

      // add a JScrollPane containing JList to content pane
      container.add( new JScrollPane( colorList ) );

      // set up event handler
      colorList.addListSelectionListener(

         // anonymous inner class for list selection events
         new ListSelectionListener() {
```

Fig. 12.14 Selecting colors from a **JList** (part 1 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 675 Saturday, July 7, 2001 2:15 PM


```
49
50              // handle list selection events
51              public void valueChanged( ListSelectionEvent event )
52              {
53                 container.setBackground(
54                    colors[ colorList.getSelectedIndex() ] );
55              }
56
57           }  // end anonymous inner class
58
59        ); // end call to addListSelectionListener
60
61        setSize( 350, 150 );
62        setVisible( true );
63     }
64
65     // execute application
66     public static void main( String args[] )
67     {
68        ListTest application = new ListTest();
69
70        application.setDefaultCloseOperation(
71           JFrame.EXIT_ON_CLOSE );
72     }
73
74  }  // end class ListTest
```

Fig. 12.14 Selecting colors from a **JList** (part 2 of 2).

A **JList** object is instantiated at line 34 and assigned to reference **colorList** in the constructor. The argument to the **JList** constructor is the array of **Object**s (in this case **String**s) to display in the list. Line 35 uses **JList** method ***setVisibleRowCount*** to determine the number of items that are visible in the list.

Lines 38–39 use **JList** method ***setSelectionMode*** to specify the list *selection mode*. Class ***ListSelectionModel*** (package **javax.swing**) defines constants ***SINGLE_SELECTION***, ***SINGLE_INTERVAL_SELECTION*** and ***MULTIPLE_INTERVAL_SELECTION*** to specify a **JList**'s selection mode. A **SINGLE_SELECTION** list allows only one item to be selected at a time. A **SINGLE_INTERVAL_SELECTION** list is a multiple-selection list that allows several items in a contiguous range in the list to be selected. A **MULTIPLE_INTERVAL_SELECTION** list is a multiple-selection list that does not restrict the items that can be selected.

Unlike **JComboBox**, **JList**s *do 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

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 676 Saturday, July 7, 2001 2:15 PM

the automatic scrolling capability for the **JList**. Line 42 adds a new instance of class ***JScrollPane*** to the content pane. The **JScrollPane** constructor receives as its argument the **JComponent** for which it will provide automatic scrolling functionality (in this case **JList colorList**). 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 45–59 use **JList** method ***addListSelectionListener*** to register an instance of an anonymous inner class that implements ***ListSelectionListener*** (defined in package **javax.swing.event**) as the listener for **JList colorList**. When the user makes a selection from **colorList**, method ***valueChanged*** (line 51–55) executes and sets the background color of the content pane with method ***setBackground*** (inherited from class **Component** into class **Container**). The color is selected from the array **colors** with the selected item's index in the list that is returned by **JList** method **getSelectedIndex** (as with arrays, **JList** indexing is zero based).

12.10 Multiple-Selection Lists

A *multiple-selection list* enables the user to select many items from a **JList**. A **SINGLE_INTERVAL_SELECTION** list allows selection of a contiguous range of items in the list by clicking the first item, then holding the *Shift* key while clicking the last item to select in the range. A **MULTIPLE_INTERVAL_SELECTION** list allows continuous range selection as described for a **SINGLE_INTERVAL_SELECTION** list and allows miscellaneous items to be selected by holding the *Ctrl* key (sometimes called to *Control* key)while clicking each item to select. To deselect an item, hold the *Ctrl* key while clicking the item a second time.

The application of Fig. 12.15 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 program, try using the selection techniques described above to select items in both lists.

```
1  // Fig. 12.15: MultipleSelection.java
2  // Copying items from one List to another.
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class MultipleSelection extends JFrame {
12    private JList colorList, copyList;
13    private JButton copyButton;
14
15    private String colorNames[] = { "Black", "Blue", "Cyan",
16       "Dark Gray", "Gray", "Green", "Light Gray",
17       "Magenta", "Orange", "Pink", "Red", "White", "Yellow" };
```

Fig. 12.15 Using a multiple-selection **JList** (part 1 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

```

   // set up GUI
   public MultipleSelection()
   {
      super( "Multiple Selection Lists" );

      // get content pane and set its layout
      Container container = getContentPane();
      container.setLayout( new FlowLayout() );

      // set up JList colorList
      colorList = new JList( colorNames );
      colorList.setVisibleRowCount( 5 );
      colorList.setFixedCellHeight( 15 );
      colorList.setSelectionMode(
         ListSelectionModel.MULTIPLE_INTERVAL_SELECTION );
      container.add( new JScrollPane( colorList ) );

      // create copy button and register its listener
      copyButton = new JButton( "Copy >>>" );

      copyButton.addActionListener(

         // anonymous inner class for button event
         new ActionListener() {

            // handle button event
            public void actionPerformed( ActionEvent event )
            {
               // place selected values in copyList
               copyList.setListData(
                  colorList.getSelectedValues() );
            }

         }  // end anonymous inner class

      ); // end call to addActionListener

      container.add( copyButton );

      // set up JList copyList
      copyList = new JList( );
      copyList.setVisibleRowCount( 5 );
      copyList.setFixedCellWidth( 100 );
      copyList.setFixedCellHeight( 15 );
      copyList.setSelectionMode(
         ListSelectionModel.SINGLE_INTERVAL_SELECTION );
      container.add( new JScrollPane( copyList ) );

      setSize( 300, 120 );
      setVisible( true );
   }

```

Fig. 12.15 Using a multiple-selection **JList** (part 2 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 678 Saturday, July 7, 2001 2:15 PM


```
71      // execute application
72      public static void main( String args[] )
73      {
74         MultipleSelection application = new MultipleSelection();
75
76         application.setDefaultCloseOperation(
77            JFrame.EXIT_ON_CLOSE );
78      }
79
80   }  // end class MultipleSelection
```

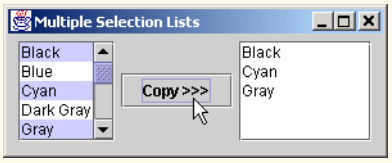


Fig. 12.15 Using a multiple-selection **JList** (part 3 of 3).

Line 29 creates **JList colorList** and initializes it with the **String**s in the array **colorNames**. Line 30 sets the number of visible rows in **colorList** to **5**. Line 31 uses **JList** method ***setFixedCellHeight*** to specify the height in pixels of each item in the **JList**. We do this to ensure that the rows in both **JLists** in the example have the same height. Lines 32–33 specify that **colorList** is a **MULTIPLE_INTERVAL_SELECTION** list. Line 34 adds a new **JScrollPane** containing **colorList** to the content pane. Lines 59–65 perform similar tasks for **JList copyList**, which is defined as a **SINGLE_INTERVAL_SELECTION** list. Line 61 uses **JList** method ***setFixedCellWidth*** to set **copyList**'s width to 100 pixels.

A multiple-selection list does not have a specific event associated with making multiple selections. 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 **JButton copyButton** to trigger the event that copies the selected items in **colorList** to **copyList**.

When the user clicks **copyButton**, method **actionPerformed** (line 45–50) is called. Lines 48–49 use **JList** method ***setListData*** to set the items displayed in **copyList**. Line 49 calls **colorList**'s method ***getSelectedValues***, which returns an array of **Object**s representing the selected items in **colorList**. In this example, the returned array is passed as the argument to **copyList**'s **setListData** method.

Many students ask how reference **copyList** can be used in line 48, when the program does not create the object to which it refers until Line 59. Remember that method **actionPerformed** at lines 45–50 does not execute until the user presses the **copyButton**, which cannot occur until after the call to the constructor completes. At that point in the program's execution, line 59 already has initialized **copyList** with a new **JList** object.

12.11 Mouse Event Handling

This section presents the ***MouseListener*** and ***MouseMotionListener*** event-listener interfaces for handling *mouse events*. Mouse events can be trapped for any GUI com-

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 679 Saturday, July 7, 2001 2:15 PM

ponent that derives from **`java.awt.Component`**. The methods of interfaces **`MouseListener`** and **`MouseMotionListener`** are summarized in Figure 12.16.

Each of the mouse event handling methods takes a ***`MouseEvent`*** object as its argument. A **`MouseEvent`** object contains information about the mouse event that occurred, including the *x*- and *y*-coordinates of the location where the event occurred. The **`MouseListener`** and **`MouseMotionListener`** methods are called automatically when the mouse interacts with a **`Component`** if listener objects are registered for a particular **`Component`**. Method ***`mousePressed`*** is called when a mouse button is pressed with the mouse cursor over a component. Using methods and constants of class ***`InputEvent`*** (the superclass of **`MouseEvent`**), a program can determine which mouse button the user clicked. Method ***`mouseClicked`*** is called whenever a mouse button is released without moving the mouse after a **`mousePressed`** operation. Method ***`mouseReleased`*** is called whenever a mouse button is released. Method ***`mouseEntered`*** is called when the mouse cursor enters the physical boundaries of a **`Component`**. Method ***`mouseExited`*** is called when the mouse cursor leaves the physical boundaries of a **`Component`**. Method ***`mouseDragged`*** is called when the mouse button is pressed and held, and the mouse is moved (a process known as *dragging*). The **`mouseDragged`** event is preceded by a **`mousePressed`** event and followed by a **`mouseReleased`** event. Method ***`mouseMoved`*** is called when the mouse is moved with the mouse cursor over a component (and no mouse buttons pressed).

`MouseListener` and **`MouseMotionListener`** interface methods

Methods of interface ***`MouseListener`***

`public void mousePressed( MouseEvent event )`

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

`public void mouseClicked( MouseEvent event )`

Called when a mouse button is pressed and released on a component without moving the mouse cursor.

`public void mouseReleased( MouseEvent event )`

Called when a mouse button is released after being pressed. This event is always preceded by a **`mousePressed`** event.

`public void mouseEntered( MouseEvent event )`

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

`public void mouseExited( MouseEvent event )`

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

Methods of interface ***`MouseMotionListener`***

`public void mouseDragged( MouseEvent event )`

Called when the mouse button is pressed with the mouse cursor on a component and the mouse is moved. This event is always preceded by a call to **`mousePressed`**.

`public void mouseMoved( MouseEvent event )`

Called when the mouse is moved with the mouse cursor on a component.

Fig. 12.16 **`MouseListener`** and **`MouseMotionListener`** interface methods.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

Look-and-Feel Observation 12.8

Method calls to **`mouseDragged`** *are sent to the* **`MouseMotionListener`** *for the* **`Component`** *on which the drag operation started. Similarly, the* **`mouseReleased`** *method call is sent to the* **`MouseListener`** *for the* **`Component`** *on which the drag operation started.*

The **`MouseTracker`** application (Fig. 12.17) demonstrates the **`MouseListener`** and **`MouseMotionListener`** methods. The application class implements both interfaces so it can listen for its own mouse events. Note that all seven methods from these two interfaces must be defined by the programmer when a class implements both interfaces. The message dialog box in the sample output windows appears when the user moves the mouse into the application window.

```
// Fig. 12.17: MouseTracker.java
// Demonstrating mouse events.

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

public class MouseTracker extends JFrame
   implements MouseListener, MouseMotionListener {

   private JLabel statusBar;

   // set up GUI and register mouse event handlers
   public MouseTracker()
   {
      super( "Demonstrating Mouse Events" );

      statusBar = new JLabel();
      getContentPane().add( statusBar, BorderLayout.SOUTH );

      // application listens to its own mouse events
      addMouseListener( this );
      addMouseMotionListener( this );

      setSize( 275, 100 );
      setVisible( true );
   }

   // MouseListener event handlers

   // handle event when mouse released immediately after press
   public void mouseClicked( MouseEvent event )
   {
      statusBar.setText( "Clicked at [" + event.getX() +
         ", " + event.getY() + "]" );
   }
```

Fig. 12.17 Demonstrating mouse event handling (part 1 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

```

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

   // handle event when mouse released after dragging
   public void mouseReleased( MouseEvent event )
   {
      statusBar.setText( "Released at [" + event.getX() +
         ", " + event.getY() + "]" );
   }

   // handle event when mouse enters area
   public void mouseEntered( MouseEvent event )
   {
      JOptionPane.showMessageDialog( null, "Mouse in window" );
   }

   // handle event when mouse exits area
   public void mouseExited( MouseEvent event )
   {
      statusBar.setText( "Mouse outside window" );
   }

   // MouseMotionListener event handlers

   // handle event when user drags mouse with button pressed
   public void mouseDragged( MouseEvent event )
   {
      statusBar.setText( "Dragged at [" + event.getX() +
         ", " + event.getY() + "]" );
   }

   // handle event when user moves mouse
   public void mouseMoved( MouseEvent event )
   {
      statusBar.setText( "Moved at [" + event.getX() +
         ", " + event.getY() + "]" );
   }

   // execute application
   public static void main( String args[] )
   {
      MouseTracker application = new MouseTracker();

      application.setDefaultCloseOperation(
         JFrame.EXIT_ON_CLOSE );
   }

}  // end class MouseTracker
```

Fig. 12.17 Demonstrating mouse event handling (part 2 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 682 Saturday, July 7, 2001 2:15 PM

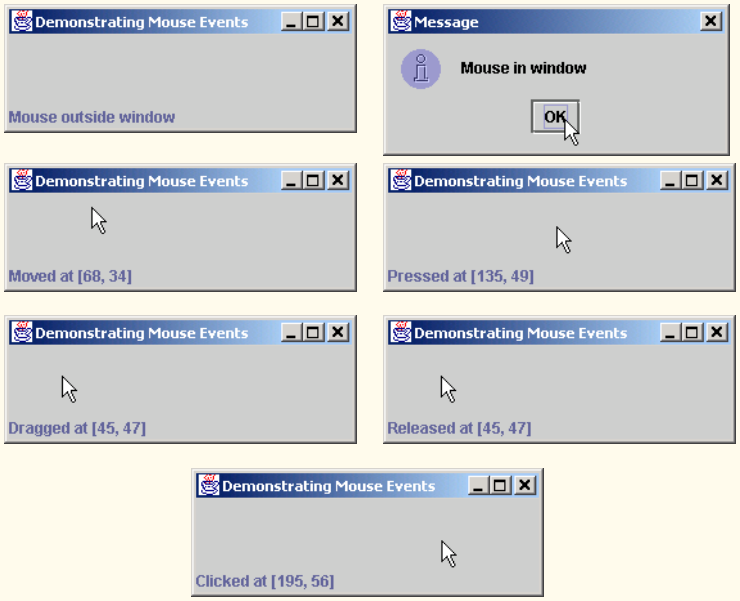


Fig. 12.17 Demonstrating mouse event handling (part 3 of 3).

Each mouse event results in a **`String`** displayed in **`JLabel statusBar`** at the bottom of the window.

Lines 21–22 in the constructor define **`JLabel statusBar`** and attach it to the content pane. Until now, each time we used the content pane, method **`setLayout`** was called to set the content pane's layout manager to a **`FlowLayout`**. This allowed the content pane to display the GUI components we attached to it from left to right. If the GUI components do not fit on one line, the **`FlowLayout`** creates additional lines to continue displaying the GUI components. Actually, the default layout manager is a ***`BorderLayout`*** that divides the content pane's area into five regions—north, south, east, west and center. Line 22 uses a new version of **`Container`** method **`add`** to attach **`statusBar`** to the region ***`BorderLayout.SOUTH`***, which extends across the entire bottom of the content pane. We discuss **`BorderLayout`** and several other layout managers in detail later in this chapter.

Lines 25–26 in the constructor register the **`MouseTracker`** window object as the listener for its own mouse events. Methods ***`addMouseListener`*** and ***`addMouseMotionListener`*** are **`Component`** methods that can be used to register mouse event listeners for an object of any class that extends **`Component`**.

When the mouse enters or exits the application area, method **`mouseEntered`** (lines 56–59) and method **`mouseExited`** (lines 62–65) are called, respectively. Method **`mouseExited`** displays a message in **`statusBar`** indicating that the mouse is outside the application (see the first sample output window). Method **`mouseEntered`** displays a message dialog box indicating that the mouse entered the application window. [Note: Be sure to press *Enter* to dismiss the message dialog, rather than using the mouse. If you use the mouse to dismiss the dialog, when you move the mouse over the window again, **`mouseEntered`** redisplays the dialog. This will prevent you from trying the other mouse events.]

© Copyright 1992-2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 683 Saturday, July 7, 2001 2:15 PM

When any of the other five events occur, they display a message in **statusBar** that includes a **String** that represents the event that occurred and the coordinates where the mouse event occurred. The *x* and *y* coordinates of the mouse when the event occurred are obtained with **MouseEvent** methods ***getX*** and ***getY***, respectively.

12.12 Adapter Classes

Many of the event-listener interfaces provide multiple methods; **MouseListener** and **MouseMotionListener** are examples. It is not always desirable to define every method in an event-listener interface. For example, a program may only need the **mouseClicked** handler from interface **MouseListener** or the **mouseDragged** handler from **MouseMotionListener**. In our windowed applications (subclasses of **JFrame**) terminating the application has been handled with **windowClosing** from interface **WindowListener**, which actually specifies seven window-event-handling methods. For many of the listener interfaces that contain multiple methods, package **java.awt.event** and package **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 every method in the interface. The **java.awt.event** adapter classes are shown in Fig. 12.18 along with the interfaces they implement.

The programmer can extend the adapter class to inherit the default implementation of every method, then override the method(s) needed for event handling. The default implementation of each method in the adapter class has an empty body. This is exactly what we have been doing in each application example that extends **JFrame** and defines method **windowClosing** to handle the closing of the window and termination of the application.

Software Engineering Observation 12.3

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 relationship. Thus, an object of a class that extends an event adapter class is an *object of the corresponding event listener type (e.g., an object of a subclass of* ***MouseAdapter*** is a ***MouseListener****).*

Event adapter class	Implements interface
ComponentAdapter	**ComponentListener**
ContainerAdapter	**ContainerListener**
FocusAdapter	**FocusListener**
KeyAdapter	**KeyListener**
MouseAdapter	**MouseListener**
MouseMotionAdapter	**MouseMotionListener**
WindowAdapter	**WindowListener**

Fig. 12.18 Event adapter classes and the interfaces they implement.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

The **Painter** application of Fig. 12.19 uses the **mouseDragged** event handler to create a simple drawing program. The user can draw pictures with the mouse by dragging the mouse on the background of the window. This example does not use method **mouseMoved**, so our **MouseMotionListener** is defined as a subclass of **MouseMotionAdapter**. This class already defines both **mouseMoved** and **mouseDragged**, so we can simply override **mouseDragged** to provide the drawing functionality.

```
// Fig. 12.19: Painter.java
// Using class MouseMotionAdapter.

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

public class Painter extends JFrame {
   private int xValue = -10, yValue = -10;

   // set up GUI and register mouse event handler
   public Painter()
   {
      super( "A simple paint program" );

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

      addMouseMotionListener(

         // anonymous inner class
         new MouseMotionAdapter() {

            // store drag coordinates and repaint
            public void mouseDragged( MouseEvent event )
            {
               xValue = event.getX();
               yValue = event.getY();
               repaint();
            }

         }  // end anonymous inner class

      ); // end call to addMouseMotionListener

      setSize( 300, 150 );
      setVisible( true );
   }

```

Fig. 12.19 Program that demonstrates adapter classes (part 1 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 685 Saturday, July 7, 2001 2:15 PM


```
45      // draw oval in a 4-by-4 bounding box at the specified
46      // location on the window
47      public void paint( Graphics g )
48      {
49         // we purposely did not call super.paint( g ) here to
50         // prevent repainting
51
52         g.fillOval( xValue, yValue, 4, 4 );
53      }
54
55      // execute application
56      public static void main( String args[] )
57      {
58         Painter application = new Painter();
59
60         application.addWindowListener(
61
62            // adapter to handle only windowClosing event
63            new WindowAdapter() {
64
65               public void windowClosing( WindowEvent event )
66               {
67                  System.exit( 0 );
68               }
69
70            }  // end anonymous inner class
71
72         ); // end call to addWindowListener
73      }
74
75   }  // end class Painter
```

Fig. 12.19 Program that demonstrates adapter classes (part 2 of 2).

The instance variables **xValue** and **yValue** store the coordinates of the **mouseDragged** event. Initially, the coordinates are set outside the window area to prevent an oval from drawing on the background area in the first call to **paint** when the window is displayed. Lines 24–39 register a **MouseMotionListener** to listen for the window's mouse motion events (remember that a call to a method that is not preceded by a reference and a dot operator is really preceded by "**this.**", indicating that the method is called for the current instance of the class at execution time). Lines 27–37 define an anonymous inner class that extends class **MouseMotionAdapter** (which implements **MouseMotionListener**). The anonymous inner class inherits a default implementation of both method **mouseMoved** and method **mouseDragged**. Thus, the anonymous inner class already satisfies the requirement that in all methods an interface must be implemented. However,

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 686 Saturday, July 7, 2001 2:15 PM

the default methods do nothing when they are called. So, we override method **mouseDragged** at lines 30–35 to capture the *x*- and *y*-coordinates of the mouse-dragged event and store them in instance variables **xValue** and **yValue**, then call **repaint** to initiate drawing the next oval on the background (performed by method **paint** at lines 47–53). Note that **paint** does not call the superclass version of **paint** inherited from **JFrame**. The superclass version normally clears the background of the window. Not calling the superclass version enables our program to keep all the ovals on the window at once. However, notice that if you cover the window with another window, only the last oval displayed still appears, because our program does not keep track of all the ovals displayed previously.

Lines 60–72 register a **WindowListener** to listen for the application window's window events (such as closing the window). Lines 63–70 define an anonymous inner class that extends class **WindowAdapter** (which implements **WindowListener**). The anonymous inner class inherits a default implementation of seven different window-event-handler methods. Thus, the anonymous inner class already satisfies the requirement that in all methods an interface must be implemented. However, the default methods do nothing when they are called. So, we override method **windowClosing** at lines 65–68 to terminate the application when the user clicks the application window's close box.

The **MouseDetails** application of Fig. 12.20 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 program is an object of inner class **MouseClickHandler** (lines 47–75) that extends **MouseAdapter** so we can define just the **mouseClicked** method we need in this example.

```
1  // Fig. 12.20: MouseDetails.java
2  // Demonstrating mouse clicks and
3  // distinguishing between mouse buttons.
4
5  // Java core packages
6  import java.awt.*;
7  import java.awt.event.*;
8
9  // Java extension packages
10 import javax.swing.*;
11
12 public class MouseDetails extends JFrame {
13    private int xPos, yPos;
14
15    // set title bar String, register mouse listener and size
16    // and show window
17    public MouseDetails()
18    {
19       super( "Mouse clicks and buttons" );
20
21       addMouseListener( new MouseClickHandler() );
22
23       setSize( 350, 150 );
24       setVisible( true );
25    }
```

Fig. 12.20 Distinguishing among left, center and right mouse-button clicks (part 1 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

```

   // draw String at location where mouse was clicked
   public void paint( Graphics g )
   {
      // call superclass's paint method
      super.paint( g );

      g.drawString( "Clicked @ [" + xPos + ", " + yPos + "]",
         xPos, yPos );
   }

   // execute application
   public static void main( String args[] )
   {
      MouseDetails application = new MouseDetails();

      application.setDefaultCloseOperation(
         JFrame.EXIT_ON_CLOSE );
   }

   // inner class to handle mouse events
   private class MouseClickHandler extends MouseAdapter {

      // handle mouse click event and determine which mouse
      // button was pressed
      public void mouseClicked( MouseEvent event )
      {
         xPos = event.getX();
         yPos = event.getY();

         String title =
            "Clicked " + event.getClickCount() + " time(s)";

         // right mouse button
         if ( event.isMetaDown() )
            title += " with right mouse button";

         // middle mouse button
         else if ( event.isAltDown() )
            title += " with center mouse button";

         // left mouse button
         else
            title += " with left mouse button";

         setTitle( title );  // set title bar of window
         repaint();
      }

   }  // end private inner class MouseClickHandler

}  // end class MouseDetails
```

Fig. 12.20 Distinguishing among left, center and right mouse-button clicks (part 2 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 688 Saturday, July 7, 2001 2:15 PM

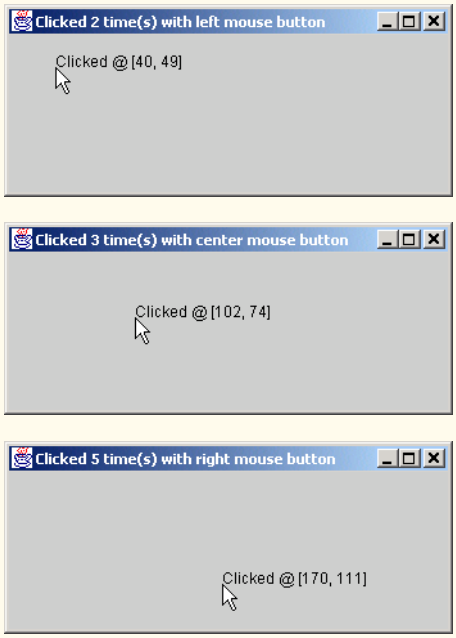


Fig. 12.20 Distinguishing among left, center and right mouse-button clicks (part 3 of 3).

A user of a Java program 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 between mouse buttons on a multi-button mouse or can mimic a multi-button mouse with a combined keystroke and mouse-button click. Figure 12.21 shows the **`InputEvent`** methods used to distinguish between mouse-button clicks. Java assumes that every mouse contains a left mouse button. Thus, it is simple to test for a left-mouse-button click. However, users with a one- or two-button mouse must use a combination of pressing keys on the keyboard and clicking the mouse at the same time to simulate the missing buttons on the mouse. In the case of a one- or two-button mouse, this program assumes that the center mouse button is clicked if the user holds 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, this program assumes that the right mouse button is clicked if the user holds the *Meta* key and clicks the mouse button.

Method **`mouseClicked`** (lines 51–73) first captures the coordinates where the event occurred and stores them in instance variables **`xPos`** and **`yPos`** of class **`MouseDetails`**. Lines 56–57 create a string containing the number of mouse clicks (as returned by **`MouseEvent`** method ***`getClickCount`*** at line 57). The nested **`if`** structure at lines 60–69 uses methods **`isMetaDown`** and **`isAltDown`** to determine which mouse button the user clicked and appends an appropriate string to **`title`** in each case. The resulting string is displayed in the title bar of the window with method ***`setTitle`*** (inherited into class **`JFrame`** from class **`Frame`**) at line 71. Line 72 calls **`repaint`** to initiate a call to **`paint`** to draw a string at the location where the user clicked the mouse.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 689 Saturday, July 7, 2001 2:15 PM

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

Fig. 12.21 **InputEvent** methods that help distinguish among left-, center- and right-mouse-button clicks .

12.13 Keyboard Event Handling

This section presents the **KeyListener** event-listener 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 definitions 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* (e.g., an arrow key, *Home*, *End*, *Page Up*, *Page Down*, a function key, *Num Lock*, *Print Screen*, *Scroll Lock*, *Caps Lock* and *Pause*). Method **keyReleased** is called when the key is released after any **keyPressed** or **keyTyped** event.

Figure 12.22 demonstrates the **KeyListener** methods. Class **KeyDemo** implements the **KeyListener** interface, so all three methods are defined in the application.

```
// Fig. 12.22: KeyDemo.java
// Demonstrating keystroke events.

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

public class KeyDemo extends JFrame implements KeyListener {
   private String line1 = "", line2 = "";
   private String line3 = "";
   private JTextArea textArea;

   // set up GUI
   public KeyDemo()
   {
      super( "Demonstrating Keystroke Events" );
```

Fig. 12.22 Demonstrating key event-handling (part 1 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 690 Saturday, July 7, 2001 2:15 PM


```

      // set up JTextArea
      textArea = new JTextArea( 10, 15 );
      textArea.setText( "Press any key on the keyboard..." );
      textArea.setEnabled( false );
      getContentPane().add( textArea );

      // allow frame to process Key events
      addKeyListener( this );

      setSize( 350, 100 );
      setVisible( true );
   }

   // handle press of any key
   public void keyPressed( KeyEvent event )
   {
      line1 = "Key pressed: " +
         event.getKeyText( event.getKeyCode() );
      setLines2and3( event );
   }

   // handle release of any key
   public void keyReleased( KeyEvent event )
   {
      line1 = "Key released: " +
         event.getKeyText( event.getKeyCode() );
      setLines2and3( event );
   }

   // handle press of an action key
   public void keyTyped( KeyEvent event )
   {
      line1 = "Key typed: " + event.getKeyChar();
      setLines2and3( event );
   }

   // set second and third lines of output
   private void setLines2and3( KeyEvent event )
   {
      line2 = "This key is " +
         ( event.isActionKey() ? "" : "not " ) +
         "an action key";

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

      line3 = "Modifier keys pressed: " +
         ( temp.equals( "" ) ? "none" : temp );

      textArea.setText(
         line1 + "\n" + line2 + "\n" + line3 + "\n" );
   }
```

Fig. 12.22 Demonstrating key event-handling (part 2 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 691 Saturday, July 7, 2001 2:15 PM


```
73
74      // execute application
75      public static void main( String args[] )
76      {
77         KeyDemo application = new KeyDemo();
78
79         application.setDefaultCloseOperation(
80            JFrame.EXIT_ON_CLOSE );
81      }
82
83   }  // end class KeyDemo
```

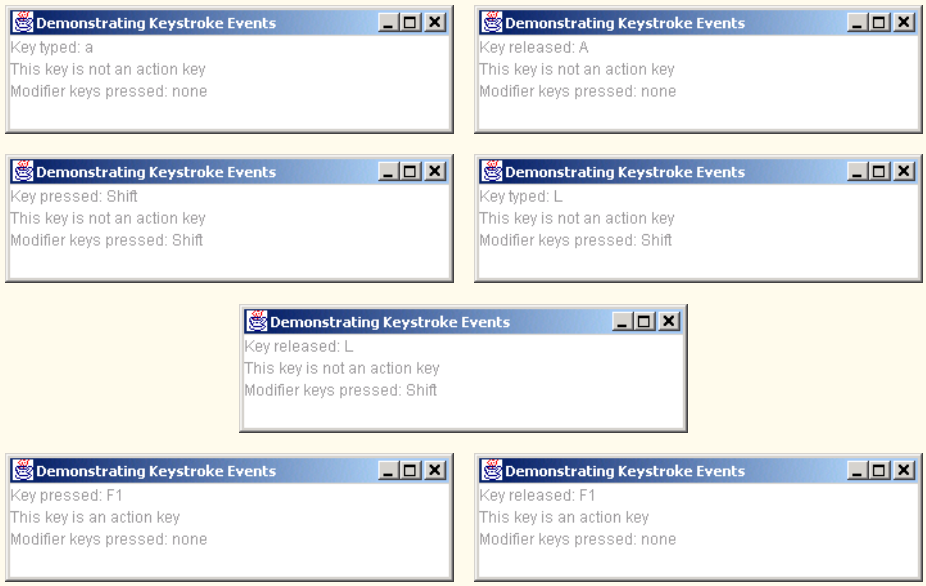


Fig. 12.22 Demonstrating key event-handling (part 3 of 3).

The constructor (lines 17–32) registers the application to handle its own key events with method ***addKeyListener*** at line 28. Method **addKeyListener** is defined in class **Component**, so every subclass of **Component** can notify **KeyListener**s of key events for that **Component**.

Line 25 in the constructor adds **JTextArea textArea** (where the program's output is displayed) to the content pane. Notice, in the screen captures, that **textArea** occupies the entire window. This is due to the content pane's default **BorderLayout** (discussed in Section 12.14.2 and demonstrated in Fig. 12.25). When a single **Component** is added to a **BorderLayout**, the **Component** occupies the entire **Container**.

Methods **keyPressed** (lines 35–40) and **keyReleased** (lines 43–48) use **KeyEvent** method ***getKeyCode*** to get the *virtual key code* of the key that was pressed. Class **KeyEvent** maintains a set of constants—the virtual key code constants—that represent every key on the keyboard. These constants can be compared with the return value of **getKeyCode** to test for individual keys on the keyboard. The value returned by **getKeyCode** is passed to **KeyEvent** method ***getKeyText***, which returns a **String** con-

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 692 Saturday, July 7, 2001 2:15 PM

taining 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 51–55) uses **`KeyEvent`** method ***`getKeyChar`*** to get the Unicode value of the character typed.

All three event handling methods finish by calling method **`setLines2and3`** (lines 58–72) and passing it the **`KeyEvent`** object. This method uses **`KeyEvent`** method ***`isActionKey`*** to determine if the key in the event was an action key. Also, **`InputEvent`** method ***`getModifiers`*** is called to determine if any modifier keys (such as *Shift*, *Alt* and *Ctrl*) were pressed when the key event occurred. The result of this method is passed to **`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 every key on the keyboard. These constants can be used from the key event handlers to determine if 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 if the particular key was pressed during the key event.]

12.14 Layout Managers

Layout managers are provided to arrange GUI components on a container for presentation purposes. The layout managers provide basic layout capabilities that are easier to use than determining the exact position and size of every GUI component. This enables the programmer to concentrate on the basic "look and feel" and lets the layout managers process most of the layout details.

Look-and-Feel Observation 12.9

Most Java programming environments provide GUI design tools that help a programmer graphically design a GUI, then automatically write Java code to create the GUI.

Some GUI designers also allow the programmer to use the layout managers described here and in Chapter 13. Figure 12.23 summarizes the layout managers presented in this chapter. Other layout managers are discussed in Chapter 13.

Layout manager	Description
`FlowLayout`	Default for **`java.awt.Applet`**, **`java.awt.Panel`** and **`javax.swing.JPanel`**. Places components sequentially (left to right) in the order they were added. It is also possible to specify the order of the components using the **`Container`** method **`add`** that takes a **`Component`** and an integer index position as arguments.
`BorderLayout`	Default for the content panes of **`JFrame`**s (and other windows) and **`JApplet`**s. Arranges the components into five areas: North, South, East, West and Center.
`GridLayout`	Arranges the components into rows and columns.

Fig. 12.23 Layout managers.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

Most previous applet and application examples in which we created our own GUI used layout manager ***FlowLayout***. Class **FlowLayout** inherits from class **Object** and implements interface ***LayoutManager***, which defines the methods a layout manager uses to arrange and size GUI components on a container.

12.14.1 FlowLayout

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

The application of Fig. 12.24 creates three **JButton** objects and adds them to the application, using a **FlowLayout** layout manager. The components are automatically center-aligned. 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 is defined with an 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** now appears on a new line.

```
// Fig. 12.24: FlowLayoutDemo.java
// Demonstrating FlowLayout alignments.

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

public class FlowLayoutDemo extends JFrame {
   private JButton leftButton, centerButton, rightButton;
   private Container container;
   private FlowLayout layout;

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

      layout = new FlowLayout();

      // get content pane and set its layout
      container = getContentPane();
      container.setLayout( layout );

```

Fig. 12.24 Program that demonstrates components in **FlowLayout** (part 1 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

```
      // set up leftButton and register listener
      leftButton = new JButton( "Left" );

      leftButton.addActionListener(

         // anonymous inner class
         new ActionListener() {

            // process leftButton event
            public void actionPerformed( ActionEvent event )
            {
               layout.setAlignment( FlowLayout.LEFT );

               // re-align attached components
               layout.layoutContainer( container );
            }

         }  // end anonymous inner class

      ); // end call to addActionListener

      container.add( leftButton );

      // set up centerButton and register listener
      centerButton = new JButton( "Center" );

      centerButton.addActionListener(

         // anonymous inner class
         new ActionListener() {

            // process centerButton event
            public void actionPerformed( ActionEvent event )
            {
               layout.setAlignment( FlowLayout.CENTER );

               // re-align attached components
               layout.layoutContainer( container );
            }
         }
      );

      container.add( centerButton );

      // set up rightButton and register listener
      rightButton = new JButton( "Right" );

      rightButton.addActionListener(

         // anonymous inner class
         new ActionListener() {

```

Fig. 12.24 Program that demonstrates components in **FlowLayout** (part 2 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 695 Saturday, July 7, 2001 2:15 PM


```
79              // process rightButton event
80              public void actionPerformed( ActionEvent event )
81              {
82                 layout.setAlignment( FlowLayout.RIGHT );
83
84                 // re-align attached components
85                 layout.layoutContainer( container );
86              }
87           }
88        );
89
90        container.add( rightButton );
91
92        setSize( 300, 75 );
93        setVisible( true );
94     }
95
96     // execute application
97     public static void main( String args[] )
98     {
99        FlowLayoutDemo application = new FlowLayoutDemo();
100
101       application.setDefaultCloseOperation(
102          JFrame.EXIT_ON_CLOSE );
103    }
104
105 }  // end class FlowLayoutDemo
```

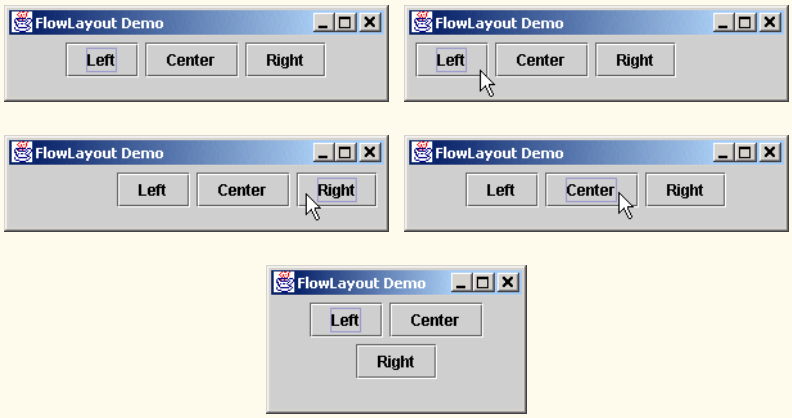


Fig. 12.24 Program that demonstrates components in **`FlowLayout`** (part 3 of 3).

As seen previously, a container's layout is set with method ***`setLayout`*** of class **`Container`**. Line 25 sets the content pane's layout manager to the **`FlowLayout`** defined at line 21. Normally, the layout is set before any GUI components are added to a container.

Look-and-Feel Observation 12.10

Each container can have only one layout manager at a time. (Separate containers in the same program can have different layout managers.)

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

Each button's **actionPerformed** event handler executes two statements. For example, line 38 in method **actionPerformed** for button **left** uses **FlowLayout** method ***setAlignment*** to change the alignment for the **FlowLayout** to a left-aligned (***FlowLayout.LEFT***) **FlowLayout**. Line 41 uses **LayoutManager** interface method ***layoutContainer*** to specify that the content pane 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***, ***FlowLayout.CENTER*** or ***FlowLayout.RIGHT***.

12.14.2 BorderLayout

The ***BorderLayout*** layout manager (the default layout manager for the content pane) arranges components into five regions: ***NORTH***, ***SOUTH***, ***EAST***, ***WEST*** and ***CENTER*** (North corresponds to the top of the container). Class **BorderLayout** inherits from **Object** and implements interface ***LayoutManager2*** (a subinterface of **LayoutManager** that adds several methods for enhanced layout processing).

Up to five components can be added directly to a **BorderLayout**—one for 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 take all remaining space in the layout (this is the reason the **JTextArea** in Fig. 12.22 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 Fig. 12.25 demonstrates the **BorderLayout** layout manager by using five **JButton**s.

```
1  // Fig. 12.25: BorderLayoutDemo.java
2  // Demonstrating BorderLayout.
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class BorderLayoutDemo extends JFrame
12    implements ActionListener {
13
```

Fig. 12.25 Demonstrating components in **BorderLayout** (part 1 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

```
   private JButton buttons[];
   private String names[] = { "Hide North", "Hide South",
      "Hide East", "Hide West", "Hide Center" };
   private BorderLayout layout;

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

      layout = new BorderLayout( 5, 5 );

      // get content pane and set its layout
      Container container = getContentPane();
      container.setLayout( layout );

      // instantiate button objects
      buttons = new JButton[ names.length ];

      for ( int count = 0; count < names.length; count++ ) {
         buttons[ count ] = new JButton( names[ count ] );
         buttons[ count ].addActionListener( this );
      }

      // place buttons in BorderLayout; order not important
      container.add( buttons[ 0 ], BorderLayout.NORTH );
      container.add( buttons[ 1 ], BorderLayout.SOUTH );
      container.add( buttons[ 2 ], BorderLayout.EAST );
      container.add( buttons[ 3 ], BorderLayout.WEST );
      container.add( buttons[ 4 ], BorderLayout.CENTER );

      setSize( 300, 200 );
      setVisible( true );
   }

   // handle button events
   public void actionPerformed( ActionEvent event )
   {
      for ( int count = 0; count < buttons.length; count++ )

         if ( event.getSource() == buttons[ count ] )
            buttons[ count ].setVisible( false );
         else
            buttons[ count ].setVisible( true );

      // re-layout the content pane
      layout.layoutContainer( getContentPane() );
   }

   // execute application
   public static void main( String args[] )
   {
      BorderLayoutDemo application = new BorderLayoutDemo();
```

Fig. 12.25 Demonstrating components in **BorderLayout** (part 2 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 698 Saturday, July 7, 2001 2:15 PM


```
67
68          application.setDefaultCloseOperation(
69             JFrame.EXIT_ON_CLOSE );
70       }
71
72    }  // end class BorderLayoutDemo
```

Fig. 12.25 Demonstrating components in **BorderLayout** (part 3 of 3).

Line 24 in the constructor defines a **BorderLayout**. The arguments specify the number of pixels between components that are arranged horizontally (*horizontal gap space*) and the number of pixels between components that are arranged vertically (*vertical gap space*), respectively. The default **BorderLayout** constructor supplies 0 pixels of gap space horizontally and vertically. Line 28 uses method **setLayout** to set the content pane's layout to **layout**.

Adding **Component**s to a **BorderLayout** requires a different add method from class **Container**, which takes two arguments—the **Component** to add and the region in which the **Component** will be placed. For example, line 39 specifies that the **buttons[0]** should appear in the **NORTH** position. The components can be added in any order, but only one component can be added to each region.

© Copyright 1992-2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 699 Saturday, July 7, 2001 2:15 PM

Look-and-Feel Observation 12.11

If no region is specified when adding a **`Component`** *to a* **`BorderLayout`**, *it is assumed that the* **`Component`** *should be added to region* **`BorderLayout.CENTER`**.

Common Programming Error 12.6

Adding more than one component to a particular region in a **`BorderLayout`** *results in only the last component added being displayed. There is no error message to indicate this problem.*

When the user clicks on a particular **`JButton`** in the layout, method **`actionPerformed`** (lines 50–61) executes. The **`for`** loop at lines 52–57 uses an **`if`**/**`else`** structure to hide the particular **`JButton`** that generated the event. Method ***`setVisible`*** (inherited into **`JButton`** from class **`Component`**) is called with a **`false`** argument 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 to ensure that the **`JButton`** is displayed on the screen. Line 60 uses **`LayoutManager`** method **`layoutContainer`** to recalculate the layout of the content pane. Notice in the screen captures of Fig. 12.25 that certain regions in the **`BorderLayout`** change shape as **`JButton`**s are hidden and displayed in other regions. Try resizing the application window to see how the various regions resize based on the width and height of the window.

12.14.3 `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, etc. Figure 12.26 demonstrates the **`GridLayout`** layout manager using six **`JButton`**s.

```
1  // Fig. 12.26: GridLayoutDemo.java
2  // Demonstrating GridLayout.
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class GridLayoutDemo extends JFrame
12    implements ActionListener {
13
14    private JButton buttons[];
15    private String names[] =
16       { "one", "two", "three", "four", "five", "six" };
17    private boolean toggle = true;
```

Fig. 12.26 Program that demonstrates components in **`GridLayout`**.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 700 Saturday, July 7, 2001 2:15 PM


```
   private Container container;
   private GridLayout grid1, grid2;

   // set up GUI
   public GridLayoutDemo()
   {
      super( "GridLayout Demo" );

      // set up layouts
      grid1 = new GridLayout( 2, 3, 5, 5 );
      grid2 = new GridLayout( 3, 2 );

      // get content pane and set its layout
      container = getContentPane();
      container.setLayout( grid1 );

      // create and add buttons
      buttons = new JButton[ names.length ];

      for ( int count = 0; count < names.length; count++ ) {
         buttons[ count ] = new JButton( names[ count ] );
         buttons[ count ].addActionListener( this );
         container.add( buttons[ count ] );
      }

      setSize( 300, 150 );
      setVisible( true );
   }

   // handle button events by toggling between layouts
   public void actionPerformed( ActionEvent event )
   {
      if ( toggle )
         container.setLayout( grid2 );
      else
         container.setLayout( grid1 );

      toggle = !toggle;   // set toggle to opposite value
      container.validate();
   }

   // execute application
   public static void main( String args[] )
   {
      GridLayoutDemo application = new GridLayoutDemo();

      application.setDefaultCloseOperation(
         JFrame.EXIT_ON_CLOSE );
   }

}  // end class GridLayoutDemo
```

Fig. 12.26 Program that demonstrates components in **GridLayout**.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 701 Saturday, July 7, 2001 2:15 PM

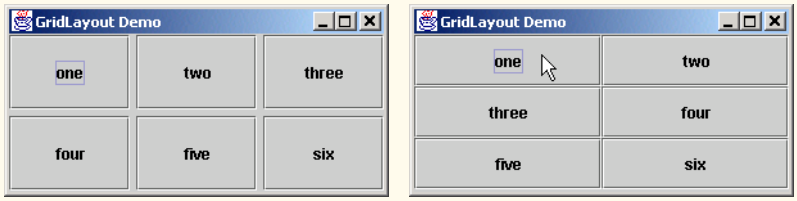


Fig. 12.26 Program that demonstrates components in **GridLayout**.

Lines 27–28 in the constructor define two **GridLayout** objects. The **GridLayout** constructor used at line 27 specifies a **GridLayout** with **2** rows, **3** columns, **5** pixels of horizontal-gap space between **Component**s in the grid and **5** pixels of vertical-gap space between **Component**s in the grid. The **GridLayout** constructor used at line 28 specifies a **GridLayout** with **3** rows, **2** columns and no gap space.

The **JButton** objects in this example initially are arranged using **grid1** (set for the content pane at line 32 with method **setLayout**). The first component is added to the first column of the first row. The next component is added to the second column of the first row, and so on. When a **JButton** is pressed, method **actionPerformed** (lines 48–57) is called. Every call to **actionPerformed** toggles the layout between **grid2** and **grid1**.

Line 56 illustrates another way to relayout a container for which the layout has changed. **Container** method ***validate*** recomputes the container's layout based on the current layout manager for the **Container** and the current set of displayed GUI components.

12.15 Panels

Complex GUIs (like Fig. 12.1) 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. Panels are created with class ***JPanel***—a subclass of **JComponent**. Class **JComponent** inherits from class **java.awt.Container**, so every **JPanel** is a **Container**. Thus **JPanel**s may have components, including other panels, added to them.

The program of Fig. 12.27 demonstrates how a **JPanel** can be used to create a more complex layout for **Component**s.

```
1  // Fig. 12.27: PanelDemo.java
2  // Using a JPanel to help lay out components.
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class PanelDemo extends JFrame {
12    private JPanel buttonPanel;
```

Fig. 12.27 A **JPanel** with five **JButton**s in a **GridLayout** attached to the **SOUTH** region of a **BorderLayout** (part 1 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 702 Saturday, July 7, 2001 2:15 PM


```
   private JButton buttons[];

   // set up GUI
   public PanelDemo()
   {
      super( "Panel Demo" );

      // get content pane
      Container container = getContentPane();

      // create buttons array
      buttons = new JButton[ 5 ];

      // set up panel and set its layout
      buttonPanel = new JPanel();
      buttonPanel.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 ) );
         buttonPanel.add( buttons[ count ] );
      }

      container.add( buttonPanel, BorderLayout.SOUTH );

      setSize( 425, 150 );
      setVisible( true );
   }

   // execute application
   public static void main( String args[] )
   {
      PanelDemo application = new PanelDemo();

      application.setDefaultCloseOperation(
         JFrame.EXIT_ON_CLOSE );
   }

}  // end class PanelDemo
```

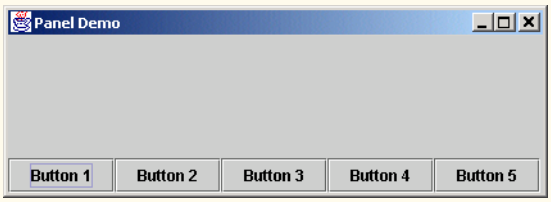


Fig. 12.27 A **JPanel** with five **JButton**s in a **GridLayout** attached to the **SOUTH** region of a **BorderLayout** (part 2 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 703 Saturday, July 7, 2001 2:15 PM

After **JPanel buttonPanel** is created at line 27, lines 28–29 set **buttonPanel**'s layout to a **GridLayout** of one row and five columns (there are five **JButton**s in array **buttons**). The five **JButton**s in array **buttons** are added to the **JPanel** in the loop with line 35. Notice that the buttons are added directly to the **JPanel**—class **JPanel** does not have a content pane like an applet or a **JFrame**. Line 38 uses the content pane's default **BorderLayout** to add **buttonPanel** to the **SOUTH** region. Note that the **SOUTH** region is as tall as the buttons on **buttonPanel**. 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 **JButton**s.

12.16 (Optional Case Study) Thinking About Objects: Use Cases

The previous eight "Thinking About Objects" sections have concentrated on the elevator-simulation model. We have identified and honed the structure and behavior of our system. In this section, we model the interaction between the user and our elevator simulation through the UML use-case diagram, which describes the sets of scenarios that occur between the user and the system.

Use-Case Diagrams

When developers begin a project, they rarely start with as detailed a problem statement as the one we provided in Section 2.9. This document and others are the result of the *object-oriented analysis* (OOA) phase. In this phase you meet with the people who want you to build a system and with the people who will eventually use that system. You use the information gained in these meetings to compile a list of *system requirements*. These requirements guide you and your fellow developers as you design the system. In our case study, the problem statement described the requirements of our elevator simulation in sufficient detail that you did not need to go through an analysis phase. The analysis phase is enormously important—you should consult the references we provide in Section 2.9 to learn more about object-oriented analysis.

The UML provides the *use-case diagram* to facilitate requirements gathering. This diagram models the interactions between the system's external clients and the *use cases* of the system. Each use case represents a different capability that the system provides to clients. For example, automated teller machines typically have several use cases, including "Deposit Money," "Withdraw Money" and "Transfer Funds."

In larger systems, use-case diagrams are indispensable tools that help system designers remain focused on satisfying the users' needs. The goal of the use-case diagram is to show the kinds of interactions users have with a system without providing the details of those interactions: those details are, of course, provided in other UML diagrams.

Figure 12.28 shows the use-case diagram for our elevator simulation. The stick figure represents an *actor*, which, in turn, represents a set of roles that an *external entity*—such as a person or another system—can play. Consider again our automated teller machine example. The actor is a **BankCustomer** who can deposit, withdraw and transfer funds from the ATM. In this sense, **BankCustomer** is more like a class rather than an object—it is not an actual person, but rather describes the roles that a real person—when playing the part of a **BankCustomer**—can perform while interacting with the ATM (deposit, with-

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 704 Saturday, July 7, 2001 2:15 PM

draw and transfer funds). A person is an external entity that can play the part of a **BankCustomer**. In the same manner as an object is an instance of a class, a person playing the part of a **BankCustomer** performing one of its roles (such as making a deposit) is an *instance of actor* **BankCustomer**. For example, when a person named Mary plays the part of a **BankCustomer** making a deposit, Mary—in the role of the depositor—becomes an instance of actor **BankCustomer**. Later in that day, another person named Jon can be another instance of actor **BankCustomer**. In the course of a day, several hundred people might use the ATM machine—some are "depositors", some are "withdrawers" and some are "transferrers," but all of these people are instances of actor **BankCustomer**.

The problem statement in our elevator simulation supplies the actors—"The user requires the ability to create a person in the simulation and situate that person on a given floor." Therefore, the actor of our system is the user who controls the simulation (i.e., the user who clicks the buttons to create new **Person**s in the simulation). An external entity—a real person—plays the part of the user to control the simulation. In our system, the use case is "Create Person," which encompasses creating a **Person** object, then placing that **Person** on either the first or second **Floor**. Figure 12.28 models one actor called "User." The actor's *name* appears underneath the actor.

The *system box* (i.e., the enclosing rectangle in the figure) contains the use cases for the system. Notice that the box is labeled "Elevator Simulation." This title shows that this use-case model focuses on the one use case that our simulation provides to users (i.e., "Create Person"). The UML models each use case as an oval. The system box for a system with multiple use cases would have one oval per use case.

There is a reasonable alternate view of the use case of our elevator simulation. The problem statement from Section 2.9 mentioned that the company requested the elevator simulation to "determine whether the elevator will meet the company's needs." We are designing a simulation of a real-world scenario—the **Person** object in the simulation represents an actual human being using an actual elevator. Thus, we may view the user of the elevator simulation as the user of the elevator. Therefore, specifying a use case from the **Person** object's perspective helps model how a real person uses a real elevator system. We offer the use case of Fig. 12.29, titled "Relocate Person." This use case describes the **Person** moving (relocating) to the other **Floor**. (The **Person** travels to the second **Floor** if starting on the first **Floor** and to the first **Floor** if starting on the second **Floor**.) This use case encompasses all actions that the **Person** performs along his or her journey, such as walking across a **Floor** to the **Elevator**, pressing **Button**s and riding the **Elevator** to the other **Floor**.

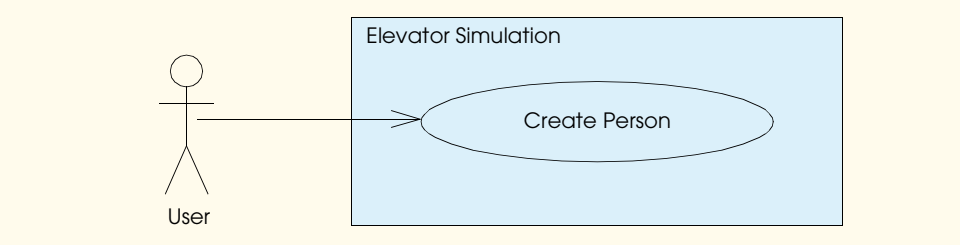


Fig. 12.28 Use-case diagram for elevator simulation from user's perspective.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 705 Saturday, July 7, 2001 2:15 PM

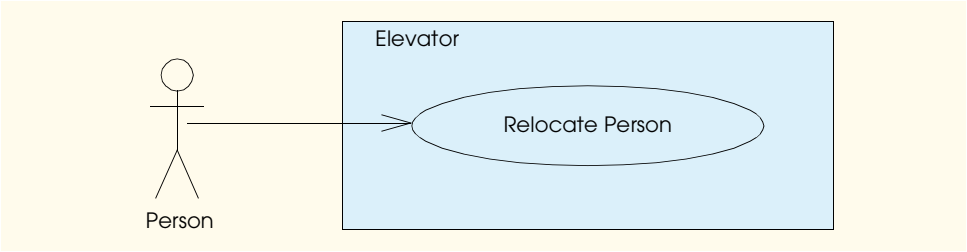


Fig. 12.29 Use-case diagram from the perspective of a **Person**.

We must ensure that our use cases do not model interactions that are too specific between the external client and the system. For example, we do not subdivide each use case into two separate use cases—such as "Create Person on first Floor" and "Create Person on second Floor," or "Relocate Person to first Floor" and "Relocate Person to second Floor"—because the functionality of such use cases is repetitive (i.e., these seemingly alternative use cases are really the same). Improper and repetitive subdivision of use cases can create problems during implementation. For example, if the designer of an automated teller machine separated its "Withdraw Money" use case into "Withdraw Specific Amounts" use cases (e.g., "Withdraw $1.00," "Withdraw $2.00," etc.), there could exist an enormous number of use cases for the system. This would make the implementation tedious. (Our elevator system contains only two floors—separating the use case into two would not cause that much extra work; if our system contained 100 floors, however, creating 100 use cases would be unwieldy.)

Constructing the Graphical User Interface

Our simulation implements both the "Create Person" and "Relocate Person" use cases. We have studied the "Relocate Person" use case through the activity diagram of the **Person** in Fig. 5.29—we implement this use case and the activity diagram in Appendix H when we create class **Person**. We implement the "Create Person" use case through a graphical user interface (GUI). We implement our GUI in class **ElevatorController** (Fig. 12.30, line 17), which is a **JPanel** subclass containing two **JButton** objects—**firstControllerButton** (line 21) and **secondControllerButton** (line 22). Each **JButton** corresponds to a **Floor** on which to place a **Person**.[1] Lines 33–38 instantiate these **JButton**s and add them to the **ElevatorController**.

We discuss in Section 13.17 how class **ElevatorModel** ties together all objects composing our elevator simulation model. Line 25 of class **ElevatorController** declares a reference to the **ElevatorModel**, because the **ElevatorController** allows the user to interact with the model. Lines 42–56 and 60–74 declare two anonymous **ActionListener** objects and register them with **firstFloorControllerButton** and **secondFloorControllerButton**, respectively, for **ActionEvent**s. When the user presses either **JButton**, lines 49–50 and 67–68 of methods **actionPer-**

1. This approach is feasible with only two **Floor**s. If the building had 100 **Floor**s, we might have opted for the user to specify the desired **Floor** in a **JTextField** and press a **JButton** to process the request.

© Copyright 1992-2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 706 Saturday, July 7, 2001 2:15 PM

formed call the **ElevatorModel**'s method **placePersonOnFloor**, which instantiates a **Person** object in the **ElevatorModel** on the specified **Floor**. Method **placePersonOnFloor** takes as an argument a **String** defined in interface **ElevatorConstants** (Fig. 12.31). This interface—used by such classes as **ElevatorController**, **ElevatorModel**, **Elevator**, **Floor** and **ElevatorView**—provides constants that specify the names of **Location**s in our simulation.

```
// ElevatorController.java
// Controller for Elevator Simulation
package com.deitel.jhtp4.elevator.controller;

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

// Deitel packages
import com.deitel.jhtp4.elevator.model.*;
import com.deitel.jhtp4.elevator.event.*;
import com.deitel.jhtp4.elevator.ElevatorConstants;

public class ElevatorController extends JPanel
   implements ElevatorConstants {

   // controller contains two JButtons
   private JButton firstControllerButton;
   private JButton secondControllerButton;

   // reference to model
   private ElevatorModel elevatorModel;

   public ElevatorController( ElevatorModel model )
   {
      elevatorModel = model;
      setBackground( Color.white );

      // add first button to controller
      firstControllerButton = new JButton( "First Floor" );
      add( firstControllerButton );

      // add second button to controller
      secondControllerButton = new JButton( "Second Floor" );
      add( secondControllerButton );

      // anonymous inner class registers to receive ActionEvents
      // from first Controller JButton
      firstControllerButton.addActionListener(
         new ActionListener() {

```

Fig. 12.30 Class **ElevatorController** processes user input (part 1 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 707 Saturday, July 7, 2001 2:15 PM


```
            // invoked when a JButton has been pressed
            public void actionPerformed( ActionEvent event )
            {
               // place Person on first Floor
               elevatorModel.placePersonOnFloor(
                  FIRST_FLOOR_NAME );

               // disable user input
               firstControllerButton.setEnabled( false );
            }
         } // end anonymous inner class
      );

      // anonymous inner class registers to receive ActionEvents
      // from second Controller JButton
      secondControllerButton.addActionListener(
         new ActionListener() {

            // invoked when a JButton has been pressed
            public void actionPerformed( ActionEvent event )
            {
               // place Person on second Floor
               elevatorModel.placePersonOnFloor(
                  SECOND_FLOOR_NAME );

               // disable user input
               secondControllerButton.setEnabled( false );
            }
         } // end anonymous inner class
      );

      // anonymous inner class enables user input on Floor if
      // Person enters Elevator on that Floor
      elevatorModel.addPersonMoveListener(
         new PersonMoveListener() {

            // invoked when Person has entered Elevator
            public void personEntered(
               PersonMoveEvent event )
            {
               // get Floor of departure
               String location =
                  event.getLocation().getLocationName();

               // enable first JButton if first Floor departure
               if ( location.equals( FIRST_FLOOR_NAME ) )
                  firstControllerButton.setEnabled( true );

               // enable second JButton if second Floor
               else
                  secondControllerButton.setEnabled( true );

            } // end method personEntered
```

Fig. 12.30 Class **ElevatorController** processes user input (part 2 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 708 Saturday, July 7, 2001 2:15 PM


```
98
99              // other methods implementing PersonMoveListener
100             public void personCreated(
101                PersonMoveEvent event ) {}
102
103             public void personArrived(
104                PersonMoveEvent event ) {}
105
106             public void personExited(
107                PersonMoveEvent event ) {}
108
109             public void personDeparted(
110                PersonMoveEvent event ) {}
111
112             public void personPressedButton(
113                PersonMoveEvent event ) {}
114
115          } // end anonymous inner class
116       );
117    } // end ElevatorController constructor
118 }
```

Fig. 12.30 Class **ElevatorController** processes user input (part 3 of 3).

Lines 53 and 71 of methods **actionPerformed** disable the respective **JButton**s to prevent the user from creating more than one **Person** per **Floor**. Lines 78–116 of class **ElevatorController** declare an anonymous **PersonMoveListener** that registers with the **ElevatorModel** to reenable the **JButton**s. Method **personEntered** (lines 82–97) of the **PersonMoveListener** reenables the **JButton** associated with the **Floor** that the **Elevator** services—after the **Person** has entered the **Elevator**, the user may place another **Person** on the **Floor**.

We mentioned in Section 9.23 that classes **Elevator** and **Floor** inherited attribute **capacity** from superclass **Location**—in Appendix H, we were going to use this attribute to prevent more than one **Person** from occupying a **Location**. However, the **PersonMoveListener**'s method **personEntered** in class **ElevatorController** prevents the user from creating more than one **Person** per **Floor**. Therefore, we have negated the need for attribute **capacity** in class **Location**. Figure 12.32 is the modified class diagram of Fig. 9.18 removing this attribute.

```
1  // ElevatorConstants.java
2  // Constants used between ElevatorModel and ElevatorView
3  package com.deitel.jhtp4.elevator;
4
5  public interface ElevatorConstants {
6
7     public static final String FIRST_FLOOR_NAME = "firstFloor";
8     public static final String SECOND_FLOOR_NAME = "secondFloor";
9     public static final String ELEVATOR_NAME = "elevator";
10 }
```

Fig. 12.31 Interface **ElevatorConstants** provides **Location** name constants.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 709 Saturday, July 7, 2001 2:15 PM

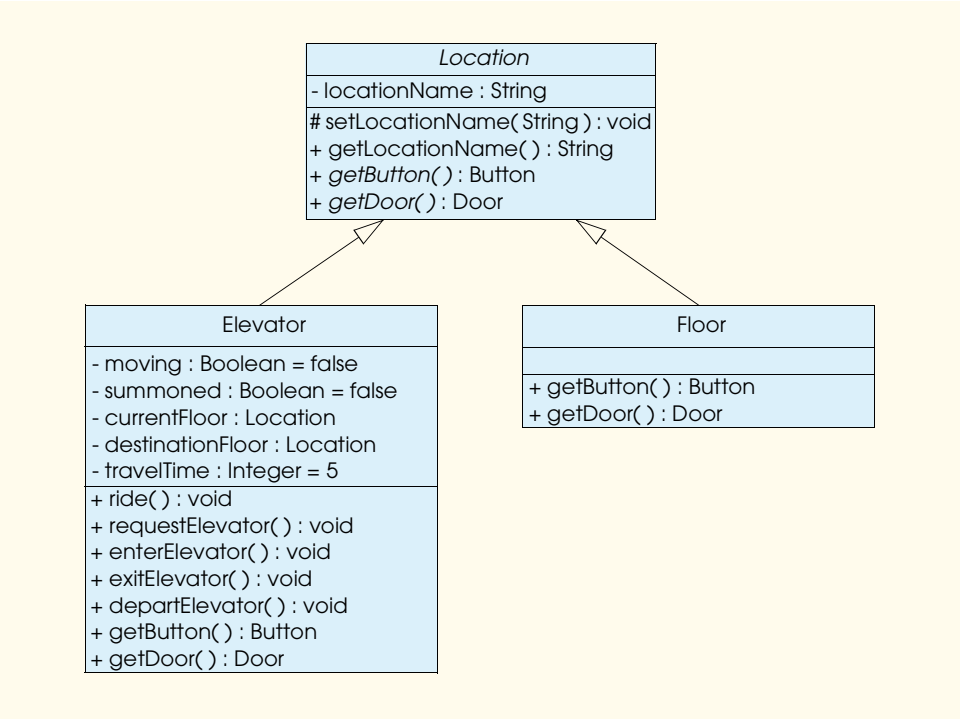


Fig. 12.32 Modified class diagram showing generalization of superclass **`Location`** and subclasses **`Elevator`** and **`Floor`**.

In this section we mentioned that the goal of object-oriented ananlysis is to produce a system-requirements document. We introduced the UML use-case diagram that facilitates gathering system requirements, and we examined the two use cases in our elevator simulation. We implemented our simulator's Graphical User Interface in Java.

This section concludes the discussion on the interaction between the user and the simulation model. In "Thinking About Objects" Section 13.17, we integrate class **`ElevatorController`** with the rest of the simulation. We also introduce the UML Component diagram, which models the **`.class`**, **`.java`**, image and sound files that comprise our system.

SUMMARY

- A graphical user interface (GUI) presents a pictorial interface to a program. A GUI (pronounced "GOO-EE") gives a program a distinctive "look" and "feel."
- By providing different applications with a consistent set of intuitive user interface components, GUIs allow the user to spend more time using the program in a productive manner.
- GUIs are built from GUI components (sometimes called controls or widgets). A GUI component is a visual object with which the user interacts via the mouse or the keyboard.
- Swing GUI components are defined in package **`javax.swing`**. Swing components are written, manipulated and displayed completely in Java.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 710 Saturday, July 7, 2001 2:15 PM

- The original GUI components from the Abstract Windowing Toolkit package **`java.awt`** are tied directly to the local platform's graphical user interface capabilities.
- Swing components are lightweight components. AWT components are tied to the local platform and are called heavyweight components—they must rely on the local platform's windowing system to determine their functionality and their look and feel.
- Several Swing GUI components are heavyweight GUI components: in particular, subclasses of **`java.awt.Window`** (such as **`JFrame`**) that display windows on the screen. Heavyweight Swing GUI components are less flexible than lightweight components.
- Much of each Swing GUI component's functionality is inherited from classes **`Component`**, **`Container`** and **`JComponent`** (the superclass to most Swing components).
- A Container is an area where components can be placed.
- **`JLabel`**s provide text instructions or information on a GUI.
- **`JComponent`** method **`setToolTipText`** specifies the tool tip that is displayed automatically when the user positions the mouse cursor over a **`JComponent`** in the GUI.
- Many Swing components can display images by specifying an **`Icon`** as an argument to their constructor or by using a method **`setIcon`**.
- Class **`ImageIcon`** (package **`javax.swing`**) supports several image formats, including Portable Network Graphics (PNG), Graphics Interchange Format (GIF) and Joint Photographic Experts Group (JPEG).
- Interface **`SwingConstants`** (package **`javax.swing`**) defines a set of common integer constants (such as **`SwingConstants.LEFT`**) that are used with many Swing components.
- By default, the text of a **`JComponent`** appears to the right of the image when the **`JComponent`** contains both text and an image.
- The horizontal and vertical alignments of a **`JLabel`** can be set with methods **`setHorizontalAlignment`** and **`setVerticalAlignment`**. Method **`setText`** sets the text displayed on the label. Method **`getText`** retrieves the current text displayed on a label. Methods **`setHorizontalTextPosition`** and **`setVerticalTextPosition`** specify the text position in a label.
- **`JComponent`** method **`setIcon`** sets the **`Icon`** displayed on a **`JComponent`**. Method **`getIcon`** retrieves the current **`Icon`** displayed on a **`JComponent`**.
- GUIs generate events when the user interacts with the GUI. Information about a GUI event is stored in an object of a class that extends **`AWTEvent`**.
- To process an event, the programmer must register an event listener and implement one or more event handlers.
- The use of event listeners in event handling is known as the delegation event model—the processing of an event is delegated to a particular object in the program.
- 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.
- **`JTextFields`** and **`JPasswordFields`** are single-line areas in which text can be entered by the user from the keyboard or text can simply be displayed. A **`JPasswordField`** shows that a character was typed as the user enters characters, but automatically hides the characters.
- When the user types data into a **`JTextField`** or **`JPasswordField`** and presses the Enter key, an **`ActionEvent`** occurs.
- **`JTextComponent`** method **`setEditable`** determines whether the user can modify the text in a **`JTextComponent`**.
- **`JPasswordField`** method **`getPassword`** returns the password as an array of type **`char`**.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

- Every **JComponent** contains an object of class **EventListenerList** (package **javax.swing.event**) called **listenerList** in which all registered listeners are stored.
- Every **JComponent** supports several different 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. Each event type has a corresponding event-listener interface.
- When an event is generated by a user interaction with a component, the component is handed a unique event ID specifying the event type. The GUI component uses the event ID to decide the type of listener to which the event should be dispatched and the event handler method to call.
- A **JButton** generates an **ActionEvent** when the user clicks the button with the mouse.
- An **AbstractButton** can have a rollover **Icon** that is displayed when the mouse is positioned over the button. The icon changes as the mouse moves in and out of the button's area on the screen. **AbstractButton** method **setRolloverIcon** specifies the image displayed on a button when the user positions the mouse over the button.
- The Swing GUI components contain three state button types—**JToggleButton**, **JCheckBox** and **JRadioButton**—that have on/off or true/false values. Classes **JCheckBox** and **JRadioButton** are subclasses of **JToggleButton**.
- When the user clicks a **JCheckBox**, an **ItemEvent** is generated that can be handled by an **ItemListener**. **ItemListener**s must define method **itemStateChanged**. **ItemEvent** method **getStateChange** determines the state of a **JToggleButton**.
- **JRadioButtons** are similar to **JCheckBox**es in that they have two states—selected and not selected (also called deselected). **JRadioButtons** normally appear as a group in which only one radio button can be selected at a time.
- The logical relationship between radio buttons is maintained by a **ButtonGroup** object.
- The **JRadioButton** constructor supplies 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 is displayed.
- **JRadioButton**s generate **ItemEvent**s when they are clicked.
- **ButtonGroup** method **add** associates a **JRadioButton** with a **ButtonGroup**. If more than one selected **JRadioButton** object is added to the group, the last selected **JRadioButton** added will be selected when the GUI is displayed.
- A **JComboBox** (sometimes called a drop-down list) provides a list of items from which the user can make a selection. **JComboBox**es generate **ItemEvent**s. A numeric index keeps track of the ordering of items in a JComboBox. The first item is added at index 0, the next item is added at index 1 and so forth. The first item added to a JComboBox appears as the currently selected item when the JComboBox is displayed. **JComboBox** method **getSelectedIndex** returns the index number of the selected item.
- A **JList** displays a series of items from which the user may select one or more items. Class **JList** supports single- and multiple-selection lists. When an item is clicked in a **JList**, a **ListSelectionEvent** occurs.
- **JList** method **setVisibleRowCount** determines the number of items that are visible in the list. Method **setSelectionMode** specifies the selection mode for the list.
- Class **JList** does *not* automatically provide a scrollbar if there are more items in the list than the number of visible rows. A **JScrollPane** object is used to provide the automatic scrolling capability for a **JList**.
- A **SINGLE_INTERVAL_SELECTION** list allows selection of a contiguous range of items by clicking the first item, then holding the Shift key while clicking the last item to select in the range.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

- A **MULTIPLE_INTERVAL_SELECTION** list allows continuous range selection as described for a **SINGLE_INTERVAL_SELECTION** list and allows miscellaneous items to be selected by holding the Ctrl key while clicking each item to select.
- **JList** method **setFixedCellHeight** specifies the height in pixels of each item in a **JList**. Method **setFixedCellWidth** sets the width in pixels of a **JList**.
- Normally, an event generated by another GUI component (known as an external event) specifies when the multiple selections in a **JList** should be processed.
- **JList** method **setListData** sets the items displayed in a **JList**. Method **getSelectedValues** returns the selected items as an array of **Object**s.
- Mouse events can be trapped for any GUI component that derives from **java.awt.Component** using **MouseListener**s and **MouseMotionListener**s.
- Each mouse event handling method takes as its argument a **MouseEvent** object containing information about the mouse event and the location where the event occurred.
- Methods **addMouseListener** and **addMouseMotionListener** are **Component** methods used to register mouse event listeners for an object of any class that extends **Component**.
- Many of the event-listener interfaces provide multiple methods. For each, there is a corresponding event-listener adapter class that provides a default implementation of every method in the interface. The programmer can extend the adapter class to inherit the default implementation of every method and simply override the method or methods needed for event handling in the program.
- **MouseEvent** method **getClickCount** returns the number of mouse clicks.
- **InputEvent** methods **isMetaDown** and **isAltDown** are used to determine which mouse button the user clicked.
- **KeyListener**s handle key events that are generated when keys on the keyboard are pressed and released. A **KeyListener** must provide definitions for methods **keyPressed**, **keyReleased** and **keyTyped**, each of which receives a **KeyEvent** as its argument.
- 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 (i.e., an arrow key, *Home*, *End*, *Page Up*, *Page Down*, a function key, *Num Lock*, *Print Screen*, *Scroll Lock*, *Caps Lock* and *Pause*). Method **keyReleased** is called when the key is released after any **keyPressed** or **keyTyped** event.
- **KeyEvent** method **getKeyCode** gets the virtual key code 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** returns a **String** containing the name of the key that corresponds to its virtual key code argument. Method **getKeyChar** gets the Unicode value of the character typed. Method **isActionKey** determines if the key in the event was an action key.
- **InputEvent** method **getModifiers** determines if any modifier keys (such as Shift, Alt and Ctrl) were pressed when the key event occurred. **KeyEvent** method **getKeyModifiersText** produces a string containing the names of the pressed modifier keys.
- Layout managers arrange GUI components on a container for presentation purposes.
- **FlowLayout** lays out components from left to right in the order in which they are added to the container. When the edge of the container is reached, components are continued on the next line.
- **FlowLayout** method **setAlignment** changes the alignment for the **FlowLayout** to **FlowLayout.LEFT**, **FlowLayout.CENTER** or **FlowLayout.RIGHT**.
- The **BorderLayout** layout manager arranges components into five regions: North, South, East, West and Center. One component can be added to each region.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 713 Saturday, July 7, 2001 2:15 PM

- **`LayoutManager`** method **`layoutContainer`** recalculates the layout of its **`Container`** argument.
- The **`GridLayout`** layout manager divides the container into a grid of rows and columns. Components are added to a **`GridLayout`** starting at the top-left cell and proceeding from left to right until the row is full. Then the process continues from left to right on the next row of the grid, and so on.
- **`Container`** method **`validate`** recomputes the container's layout based on the current layout manager for the **`Container`** and the current set of displayed GUI components.
- Panels are created with class **`JPanel`**, which inherits from class JComponent. JPanels may have components, including other panels, added to them.

TERMINOLOGY

`.gif` file name extension
`.jpg` file name extension
"listen" for an event
Abstract Windowing Toolkit
`AbstractButton` class
`ActionEvent` class
`ActionListener` interface
`actionPerformed` method
adapter class
`add` method of **`ButtonGroup`**
`add` method of class **`Container`**
`addItemListener` method
`addKeyListener` method
`addListSelectionListener` method
`addMouseListener` method
`addMouseMotionListener` method
assistive technologies
`BorderLayout` class
`BorderLayout.CENTER`
`BorderLayout.EAST`
`BorderLayout.NORTH`
`BorderLayout.SOUTH`
`BorderLayout.WEST`
button
button label
`ButtonGroup` class
centered
check box
check box label
command button
`Component` class
`ComponentAdapter` class
`ComponentListener` interface
`Container` class
`ContainerAdapter` class
`ContainerListener` interface
control
delegation event model
dispatch an event
dragging
drop-down list
event
event driven
event handler
event ID
event listener
event-listener interface
`EventListenerList` class
`EventObject` class
`FlowLayout` class
`FlowLayout.CENTER`
`FlowLayout.LEFT`
`FlowLayout.RIGHT`
focus
`FocusAdapter` class
`FocusListener` interface
`Font.BOLD`
`Font.ITALIC`
`Font.PLAIN`
`getActionCommand` method
`getClickCount` method
`getIcon` method
`getKeyChar` method of **`KeyEvent`**
`getKeyCode` method of **`KeyEvent`**
`getKeyModifiersText` method
`getKeyText` method of **`KeyEvent`**
`getModifiers` method of **`InputEvent`**
`getPassword` method of **`JPasswordField`**
`getSelectedIndex` method of **`JComboBox`**
`getSelectedIndex` method of **`JList`**
`getSelectedValues` method of **`JList`**
`getSource` method of **`ActionEvent`**
`getStateChange` method of **`ItemEvent`**
`getText` method of **`JLabel`**
`getX` method of **`MouseEvent`**
`getY` method of **`MouseEvent`**

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 714 Saturday, July 7, 2001 2:15 PM

Graphics Interchange Format (GIF)
GridLayout class
GUI component
heavyweight component
horizontal gap space
Icon interface
ImageIcon class
InputEvent class
isActionKey method of **KeyEvent**
isAltDown method of **InputEvent**
isMetaDown method of **InputEvent**
ItemEvent class
ItemEvent.DESELECTED
ItemEvent.SELECTED
ItemListener interface
itemStateChanged method
java.awt package
java.awt.event package
javax.swing package
javax.swing.event package
JButton class
JCheckBox class
JComboBox class
JComponent class
JLabel class
JList class
Joint Photographic Experts Group (JPEG)
JPanel class
JPasswordField class
JRadioButton class
JScrollPane class
JTextComponent class
JTextField class
JToggleButton class
KeyAdapter class
KeyEvent class
KeyListener interface
keyPressed method of **KeyListener**
keyReleased method of **KeyListener**
keyTyped method of **KeyListener**
label
layout manager
layoutContainer method
LayoutManger interface
left aligned
left justified
lightweight component
ListSelectionEvent class
ListSelectionListener interface
ListSelectionModel interface
look and feel
menu
menu bar
MouseAdapter class
mouseClicked method
mouseDragged method
mouseEntered method
MouseEvent class
mouseExited method
MouseListener interface
MouseMotionAdapter class
MouseMotionListener interface
mouseMoved method
mousePressed method
mouseReleased method
multiple-selection list
password
pluggable look and feel
radio button
read-only text
register an event listener
right-aligned
rollover icon
scroll arrow
scroll box
scrollbar
selection mode
setAlignment method
setBackground method
setEditable method
setFixedCellHeight method
setFixedCellWidth method
setHorizontalAlignment method
setHorizontalTextPosition method
setIcon method
setLayout method of class **Container**
setListData method of **JList**
setMaximumRowCount method
setRolloverIcon method
setSelectionMode method
setToolTipText method
setVerticalAlignment method
setVerticalTextPosition method
setVisible method
setVisibleRowCount method
shortcut key (mnemonics)
single-selection list
SwingConstants interface
tool tips
toolbar

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 715 Saturday, July 7, 2001 2:15 PM

user interface localization
`validate` method
`valueChanged` method
vertical gap space
widget (window gadget)
window
`WindowAdapter` class
`windowClosing` method
windowing system
`WindowListener` interface

SELF-REVIEW EXERCISES

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

a) Method _____ is called when the mouse is moved and an event listener is registered to handle the event.
b) Text that cannot be modified by the user is called _____ text.
c) A _____ arranges GUI components on a **`Container`**.
d) The **`add`** method for attaching GUI components is a _____ class method.
e) GUI is an acronym for _____.
f) Method _____ is used to set the layout manager for a container.
g) A **`mouseDragged`** method call is preceded by a _____ method call and followed by a _____ method call.

12.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) **`BorderLayout`** is the default layout manager for a 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) When one is using **`BorderLayout`**, a maximum of five components may be used.

12.3 Find the error(s) in each of the following and explain how to correct it (them).

a)
```
buttonName = JButton( "Caption" );
```
b)
```
JLabel aLabel, JLabel;      // create references
```
c)
```
txtField = new JTextField( 50, "Default Text" );
```
d)
```
Container container = getContentPane();
setLayout( new BorderLayout() );
button1 = new JButton( "North Star" );
button2 = new JButton( "South Pole" );
container.add( button1 );
container.add( button2 );
```

ANSWERS TO SELF-REVIEW EXERCISES

12.1 a) **`mouseMoved`**. b) uneditable (read-only). c) layout manager. d) **`Container`**. e) graphical user interface. f) **`setLayout`**. g) **`mousePressed`**, **`mouseReleased`**.

12.2

a) True.
b) False. Method **`mouseEntered`** is called.
c) False. A **`JPanel`** can be added to another **`JPanel`** because **`JPanel`** derives indirectly from **`Component`**. Therefore, a **`JPanel`** is a **`Component`**. Any **`Component`** can be added to a **`Container`**.
d) False. Only the last button added will be displayed. Remember that only one component can be added to each region in a **`BorderLayout`**.
e) True.

12.3

a) **`new`** is needed to instantiate the object.
b) JLabel is a class name and cannot be used as a variable name.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

c) The arguments passed to the constructor are reversed. The **String** must be passed first.
d) **BorderLayout** has been set and components are being added without specifying the region. Proper **add** statements might be

```
container.add( button1, BorderLayout.NORTH );
container.add( button2, BorderLayout.SOUTH );
```

EXERCISES

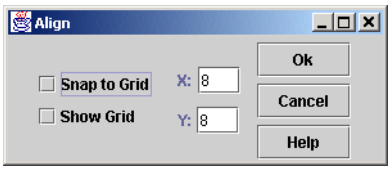
12.4 Fill in the blanks in each of the following statements:
a) The **JTextField** class inherits directly from _____.
b) The layout managers discussed in this chapter are _____, _____ and _____.
c) **Container** method _____ attaches a GUI component to a container.
d) Method _____ is called when a mouse button is released (without moving the mouse).
e) The _____ class is used to create a group of **JRadioButton**s.

12.5 State whether each of the following 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**s provide a series of mutually exclusive options (only one can be **true** at a time).
d) **Graphics** method **setFont** 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 contains a method called **mouseDragged**.

12.6 State whether each of the following is *true* or *false*. If *false*, explain why.
a) A **JApplet** does not have a content pane.
b) A **JPanel** is a **JComponent**.
c) A **JPanel** is a **Component**.
d) A **JLabel** is a **Container**.
e) A **JList** is a **JPanel**.
f) An **AbstractButton** is a **JButton**.
g) A **JTextField** is an **Object**.
h) **ButtonGroup** inherits from JComponent.

12.7 Find any error(s) in each of the following and explain how to correct it (them).
a)
```
import javax.swing.*        // include swing package
```
b)
```
panelObject.GridLayout( 8, 8 ); // set GridLayout
```
c)
```
container.setLayout(
   new FlowLayout( FlowLayout.DEFAULT ) );
```
d)
```
container.add( eastButton, EAST );  // BorderLayout
```

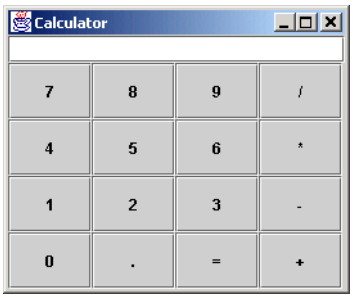
12.8 Create the following GUI. You do not have to provide any functionality.



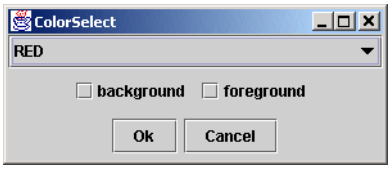
© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 717 Saturday, July 7, 2001 2:15 PM

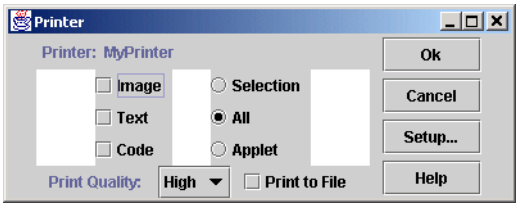
12.9 Create the following GUI. You do not have to provide any functionality.



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



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



12.12 Write a temperature conversion program that converts from Fahrenheit to Celsius. The Fahrenheit temperature should be entered from the keyboard (via a **`JTextField`**). A **`JLabel`** should be used to display the converted temperature. Use the following formula for the conversion:

$$Celsius = 5 / 9 \times (Fahrenheit - 32)$$

12.13 Enhance the temperature conversion program of Exercise 12.12 by adding the Kelvin temperature scale. The program 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 12.12):

$$Kelvin = Celsius + 273$$

12.14 Write an application that allows the user to draw a rectangle by dragging the mouse on the application window. The upper-left coordinate should be the location where the user presses the mouse button, and the lower-right coordinate should be the location where the user releases the mouse button. Also display the area of the rectangle in a **`JLabel`** in the **`SOUTH`** region of a **`BorderLayout`**. Use the following formula for the area:

$$area = width \times height$$

12.15 Modify the program of Exercise 12.14 to draw different shapes. The user should be allowed to choose from an oval, an arc, a line, a rectangle with rounded corners and a predefined polygon. Also display the mouse coordinates in the status bar.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 718 Saturday, July 7, 2001 2:15 PM

12.16 Write a program that will allow the user to draw a shape with the mouse. The shape to draw should be determined by a **`KeyEvent`** using the following keys: *c* draws a circle, *o* draws an oval, *r* draws a rectangle and *l* draws a line. The size and placement of the shape should be determined by the **`mousePressed`** and **`mouseReleased`** events. Display the name of the current shape in a **`JLabel`** in the **`SOUTH`** region of a **`BorderLayout`**. The initial shape should default to a circle.

12.17 Create an application that enables the user to paint a picture. The user should be able to choose the shape to draw, the color in which the shape should appear and whether the shape should be filled with color. Use the graphical user interface components we discussed in this chapter, such as **`JComboBox`**es, **`JRadioButton`**s and **`JCheckBox`**es, to allow the user to select various options. The program should provide a **`JButton`** object that allows the user to erase the window.

12.18 Write a program that uses **`System.out.println`** statements to print out events as they occur. Provide a **`JComboBox`** with a minimum of four items. The user should be able to choose an event to "monitor" from the **`JComboBox`**. When that particular event occurs, display information about the event in a message dialog box. Use method **`toString`** on the event object to convert it to a string representation.

12.19 Write a program that draws a square. As the mouse moves over the drawing area, repaint the square with the upper-left corner of the square following the exact path of the mouse cursor.

12.20 Modify the program of Fig. 12.19 to incorporate colors. In a separate window, provide a "toolbar" of **`JRadioButton`** objects that lists the following six colors: red, black, magenta, blue, green and yellow. The toolbar should be implemented as a subclass of **`JFrame`** called **`ToolBarWindow`** and should consist of six buttons, each with the appropriate color name. When a new color is selected, drawing should occur in the new color. Determine the currently selected color in the **`mousePressed`** event handler of the main window by calling a public method **`getCurrentColor`** on the **`ToolBarWindow`**. [Note: In Chapter 13, we discuss how to combine GUI components and drawing, using separate **`JPanel`**s for each. This provides programs with more flexibility in laying out the components and drawing.]

12.21 Write a program that plays "guess the number" as follows: Your program chooses the number to be guessed by selecting an integer at random in the range 1–1000. The program then displays 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 indicates that the user is getting "colder." A **`JLabel`** should display either "**`Too High`**" or "**`Too Low`**" to help the user zero in on the correct answer. When the user gets the correct answer, "**`Correct!`**" should be displayed and the **`JTextField`** used for input should be changed to 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 editable.

12.22 It is often useful to display the events that occur during the execution of a program to help understand when the events occur and how they are generated. Write a program that enables the user to generate and process every event discussed in this chapter. The program should provide methods from the **`ActionListener`**, **`ItemListener`**, **`ListSelectionListener`**, **`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 a **`String`** that can be displayed. Method **`toString`** creates a **`String`** containing all the information in the event object.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

jhtp4_12.fm Page 719 Saturday, July 7, 2001 2:15 PM

12.23 Modify your solution to Exercise 12.17 to enable the user to select a font and a font size, then type text into a **JTextField**. When the user presses *Enter,* the text should be displayed on the background in the chosen font and size. Modify the program further to allow the user to specify the exact position at which the text should be displayed.

12.24 Write a program that allows the user to select a shape from a **JComboBox**, then draws that shape 20 times with random locations and dimensions in method **paint**. The first item in the **JComboBox** should be the default shape that is displayed the first time **paint** is called.

12.25 Modify Exercise 12.24 to draw each of the 20 randomly sized shapes in a randomly selected color. Use all 13 predefined **Color** objects in an array of **Color**s.

12.26 Modify Exercise 12.25 to allow the user to select the color in which shapes should be drawn from a **JColorChooser** dialog.

12.27 Write a program using methods from interface **MouseListener** that allows the user to press the mouse button, drag the mouse and release the mouse button. When the mouse is released, draw a rectangle with the appropriate upper-left corner, width and height. (*Hint*: The **mousePressed** method should capture the set of coordinates at which the user presses and holds the mouse button initially, and the **mouseReleased** method should capture the set of coordinates at which the user releases the mouse button. Both methods should store the appropriate coordinate values. All calculations of the width, height and upper-left corner should be performed by the **paint** method before the shape is drawn.)

12.28 Modify Exercise 12.27 to provided a "rubber-banding" effect. As the user drags the mouse, the user should be able to see the current size of the rectangle to know exactly what the rectangle will look like when the mouse button is released. (*Hint*: Method **mouseDragged** should perform the same tasks as **mouseReleased**.)

12.29 Modify Exercise 12.28 to allow the user to select which shape to draw. A **JComboBox** should provide options including at least rectangle, oval, line and rounded rectangle.

12.30 Modify Exercise 12.29 to allow the user to select the drawing color from a **JColorChooser** dialog box.

12.31 Modify Exercise 12.30 to allow the user to specify whether a shape should be filled or empty when it is drawn. The user should click a **JCheckBox** to indicate filled or empty.

12.32 *(Painting program)* Using the techniques of Exercise 9.28–Exercise 9.29 and Exercise 12.27–Exercise 12.30 and the graphics techniques of Chapter 11, rewrite Exercise 12.31 to allow the user to draw multiple shapes and store each shape in an array of shapes. (If you feel ambitious, investigate the capabilities of class **Vector** in Chapter 20.) For this program, create your own classes (like those in the class hierarchy described in Exercise 9.28–Exercise 9.29) from which objects will be created to store each shape the user draws. The classes should store the location, dimensions and color of each shape and should indicate whether the shape is filled or unfilled. Your classes should all derive from a class called **MyShape** that has all the common features of every shape type. Every subclass of **MyShape** should have its own method **draw**, which returns **void** and receives a **Graphics** object as its argument. When the application window's **paint** method is called, it should walk through the array of shapes and display each shape by polymorphically calling the shape's **draw** method (passing the **Graphics** object as an argument). Each shape's **draw** method should know how to draw the shape. As a minimum, your program should provide the following classes: **MyLine**, **MyOval**, **MyRect**, **MyRoundRect**. Design the class hierarchy for maximum software reuse, and place all your classes in the package **shapes**. Import this package into your program.

12.33 Modify Exercise 12.32 to provide an **Undo** button that can be used repeatedly to undo the last painting operation. If there are no shapes in the array of shapes, the Undo button should be disabled.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01