

13

Graphical User Interface Components: Part 2

Objectives

- To create and manipulate text areas, sliders, menus, popup menus and windows.
- To be able to create customized **JPanel** objects.
- To be able to create a program that can execute as either an applet or an application.
- To be able to change the look-and-feel of a GUI, using Swing's pluggable look-and-feel (PLAF).
- To be able to create a multiple document interface with **JDesktopPane** and **JInternalFrame**.
- To be able to use advanced layout managers.

I claim not to have controlled events, but confess plainly that events have controlled me.

Abraham Lincoln

A good symbol is the best argument, and is a missionary to persuade thousands.

Ralph Waldo Emerson

Capture its reality in paint!

Paul Cézanne



Outline

- 13.1 Introduction
- 13.2 `JTextArea`
- 13.3 Creating a Customized Subclass of `JPanel`
- 13.4 Creating a Self-Contained Subclass of `JPanel`
- 13.5 `JSlider`
- 13.6 Windows
- 13.7 Designing Programs that Execute as Applets or Applications
- 13.8 Using Menus with Frames
- 13.9 Using `JPopupMenu`s
- 13.10 Pluggable Look-and-Feel
- 13.11 Using `JDesktopPane` and `JInternalFrame`
- 13.12 Layout Managers
- 13.13 `BoxLayout` Layout Manager
- 13.14 `CardLayout` Layout Manager
- 13.15 `GridBagLayout` Layout Manager
- 13.16 `GridBagConstraints` Constants `RELATIVE` and `REMAINDER`
- 13.17 (Optional Case Study) Thinking About Objects: Model-View-Controller
- 13.18 (Optional) Discovering Design Patterns: Design Patterns Used in Packages `java.awt` and `javax.swing`
 - 13.18.1 Creational Design Patterns
 - 13.18.2 Structural Design Patterns
 - 13.18.3 Behavioral Design Patterns
 - 13.18.4 Conclusion

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

13.1 Introduction

In this chapter, we continue our study of GUIs. We discuss more advanced components and layout managers and lay the groundwork for building complex GUIs.

We begin our discussion with another text-based GUI component—***JTextArea***—which allows multiple lines of text to be displayed or input. We continue with two examples of *customizing class ***JPanel**** in which we discuss issues that relate to painting on Swing GUI components. Next, we illustrate how to design a Java program that can execute as both an applet and an application. An important aspect of any complete GUI is a system of *menus* that enable the user to effectively perform tasks in the program. The next two examples discuss how to create and use menus. The look-and-feel of a Swing GUI can be uniform across all platforms on which the Java program is executed, or the GUI can be

customized by using Swing's *pluggable look-and-feel (PLAF)*. The next example illustrates how to change between Swing's default *metal look-and-feel*, a look-and-feel that simulates *Motif* (a popular UNIX look-and-feel) and one that simulates Microsoft's Windows look-and-feel. Many of today's applications use a *multiple document interface (MDI)*, [i.e., a main window (often called the *parent window*) containing other windows (often called *child windows*) to manage several open *documents* in parallel. For example, many e-mail programs allow you to have several e-mail windows open at the same time so you can compose and/or read multiple e-mail messages. The next example discusses Swing's classes that provide support for creating multiple document interfaces. Finally, the chapter finishes with a series of examples discussing several advanced layout managers for organizing graphical user interfaces.

Swing is a large and complex topic. There are many more GUI components and capabilities than can be presented here. Several more Swing GUI components are introduced in the remaining chapters of this book as they are needed. Our book *Advanced Java 2 Platform How to Program* discusses other, more advanced Swing components and capabilities.

13.2 JTextArea

JTextAreas provide an area for manipulating multiple lines of text. Like class **JTextField**, class **JTextArea** inherits from **JTextComponent**, which defines common methods for **JTextFields**, **JTextAreas** and several other text-based GUI components.

The application of Fig. 13.1 demonstrates **JTextAreas**. One **JTextArea** displays text that the user can select. The other **JTextArea** is uneditable. Its purpose is to display the text the user selected in the first **JTextArea**. **JTextAreas** do not have action events like **JTextFields**. Often, an *external event*—an event generated by a different GUI component—indicates when to process the text in a **JTextArea**. For example, when typing an e-mail message, you often click a **Send** button to take the text of the message and send it to the recipient. Similarly, when editing a document in a word processor, you normally save the file by selecting a menu item called **Save** or **Save As....** In this program, the button **Copy >>>** generates the external event that copies the selected text in the left **JTextArea** and displays it in the right **JTextArea**.

```

1  // Fig. 13.1: TextAreaDemo.java
2  // Copying selected text from one text area 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 TextAreaDemo extends JFrame {
12     private JTextArea textArea1, textArea2;
13     private JButton copyButton;
14 }

```

Fig. 13.1 Copying selected text from one text area to another (part 1 of 3).

```
15 // set up GUI
16 public TextAreaDemo()
17 {
18     super( "TextArea Demo" );
19
20     Box box = Box.createHorizontalBox();
21
22     String string = "This is a demo string to\n" +
23         "illustrate copying text\n" +
24         "from one TextArea to \n" +
25         "another TextArea using an\n" + "external event\n";
26
27     // set up textArea1
28     JTextArea textArea1 = new JTextArea( string, 10, 15 );
29     box.add( new JScrollPane( textArea1 ) );
30
31     // set up copyButton
32     JButton copyButton = new JButton( "Copy >>>" );
33     copyButton.addActionListener(
34
35         // anonymous inner class to handle copyButton event
36         new ActionListener() {
37
38             // set text in textArea2 to selected
39             // text from textArea1
40             public void actionPerformed((ActionEvent event) )
41             {
42                 textArea2.setText( textArea1.getSelectedText() );
43             }
44
45         } // end anonymous inner class
46
47     ); // end call to addActionListener
48
49     box.add( copyButton );
50
51     // set up textArea2
52     JTextArea textArea2 = new JTextArea( 10, 15 );
53     textArea2.setEditable( false );
54     box.add( new JScrollPane( textArea2 ) );
55
56     // add box to content pane
57     Container container = getContentPane();
58     container.add( box ); // place in BorderLayout.CENTER
59
60     setSize( 425, 200 );
61     setVisible( true );
62 }
63
64 // execute application
65 public static void main( String args[] )
66 {
67     TextAreaDemo application = new TextAreaDemo();
```

Fig. 13.1 Copying selected text from one text area to another (part 2 of 3).

```

68
69     application.setDefaultCloseOperation(
70         JFrame.EXIT_ON_CLOSE );
71     }
72
73 } // end class TextAreaDemo

```

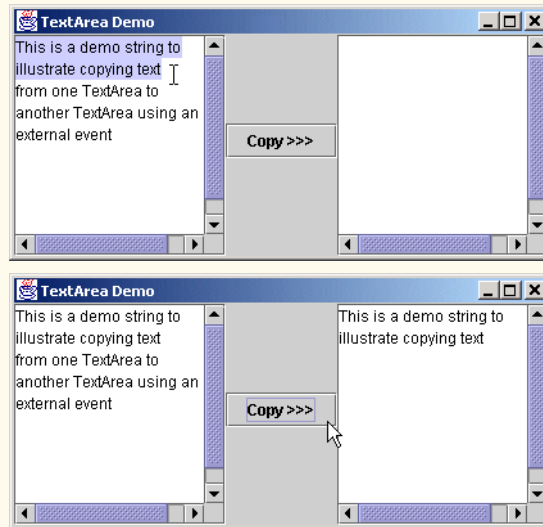


Fig. 13.1 Copying selected text from one text area to another (part 3 of 3).



Look-and-Feel Observation 13.1

Often an external event determines when the text in a **JTextArea** should be processed.

In the constructor method (lines 16–62), line 20 creates a **Box** container (package **javax.swing**) for organizing the GUI components. Class **Box** is a subclass of **Container** that uses a **BoxLayout** layout manager to arrange the GUI components either horizontally or vertically. Section 13.13 discusses **BoxLayout** in detail. Class **Box** provides static method **createHorizontalBox** to create a **Box** that arranges components from left to right in the order that the components are attached.

The application instantiates **JTextArea** objects **textArea1** (line 28) and **textArea2** (line 52). Each **JTextArea** has 10 visible rows and 15 visible columns. Line 28 specifies that **string** should be displayed as the default **JTextArea** content. A **JTextArea** does not provide scrollbars if it cannot display its complete contents. For this reason, line 29 creates a **JScrollPane** object, initializes it with **textArea1** and attaches it to container **box**. By default, horizontal and vertical scrollbars will appear as necessary.

Lines 32–49 instantiate **JButton** object **copyButton** with the label “Copy >>>,” create an anonymous inner class to handle **copyButton**’s **ActionEvent** and add **copyButton** to container **box**. This button provides the external event that determines when the program should copy the selected text in **textArea1** to **textArea2**. When the user clicks **copyButton**, line 42 in **actionPerformed** indicates that method

getSelectedText (inherited into **JTextArea** from **JTextComponent**) should return the *selected text* from **textArea1**. The user selects text by dragging the mouse over the desired text to highlight it. Method **setText** changes the text in **textArea2** to the **String** that method **getSelectedText** returns.

Lines 52–54 create **textArea2** and add it to container **box**. Lines 57–58 obtain the content pane for the window and add **box** to the content pane. Remember that the default layout of the content pane is a **BorderLayout** and that the add method attaches its argument to the **CENTER** of the **BorderLayout** if method **add** does not specify the region.

It is sometimes desirable when text reaches the right side of a **JTextArea** to have the text wrap to the next line. This is referred to as *automatic word wrap*.



Look-and-Feel Observation 13.2

To provide automatic word wrap functionality for a **JTextArea**, invoke **JTextArea** method **setLineWrap** with a **true** argument.

This example uses a **JScrollPane** to provide scrolling functionality for a **JTextArea**. By default, **JScrollPane** provides scrollbars only if they are required. You can set the horizontal and vertical *scrollbar policies* for the **JScrollPane** when a **JScrollPane** is constructed or with methods **setHorizontalScrollBarPolicy** and **setVerticalScrollBarPolicy** of class **JScrollPane** at any time. Class **JScrollPane** provides the constants

```
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS  
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
```

to indicate that a scrollbar should always appear, constants

```
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED  
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
```

to indicate that a scrollbar should appear only if necessary, and constants

```
JScrollPane.VERTICAL_SCROLLBAR_NEVER  
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
```

to indicate that a scrollbar should never appear. If the horizontal scrollbar policy is set to **JScrollPane.HORIZONTAL_SCROLLBAR_NEVER**, a **JTextArea** attached to the **JScrollPane** will exhibit automatic word wrap behavior.

13.3 Creating a Customized Subclass of JPanel1

In Chapter 12, we saw that **JPanel**s can aggregate a set of GUI components for layout purposes. **JPanel**s are quite flexible. Some of their many uses include creating *dedicated drawing areas* and creating areas that receive mouse events. Programs often extend class **JPanel** to create new components. Our next example uses a **JPanel** to create a dedicated drawing area. Dedicated drawing areas help separate drawing from the rest of your graphical user interface. This can be beneficial in Swing graphical user interfaces. If graphics and Swing GUI components are not displayed in the correct order, it is possible that the GUI components will not display correctly. For example, to ensure that graphics and GUI both display correctly, we can separate the GUI and the graphics by creating dedicated drawing areas as subclasses of **JPanel**.



Look-and-Feel Observation 13.3

*Combining graphics and Swing GUI components can lead to incorrect display of the graphics, the GUI components or both. Using **JPanels** for drawing can eliminate this problem by providing a dedicated area for graphics.*

Swing components that inherit from class **JComponent** contain method **paintComponent** that helps them draw properly in the context of a Swing GUI. When customizing a **JPanel** for use as a dedicated drawing area, the subclass should override method **paintComponent** and call the superclass version of **paintComponent** as the first statement in the body of the overridden method. This ensures that painting occurs in the proper order and that Swing's painting mechanism remains intact. An important part of this mechanism is that subclasses of **JComponent** support *transparency*, which can be set with method **setOpaque** (a **false** argument indicates the component is transparent). To paint a component correctly, the program must determine whether the component is transparent. The code that performs this check is in the superclass version of **paintComponent**. When a component is transparent, **paintComponent** will not clear the component's background when the program paints the component. When a component is *opaque*, **paintComponent** clears the background before continuing the painting operation. If the superclass version of **paintComponent** is not called, an opaque GUI component typically will not display correctly on the user interface. Also, if the superclass version is called after performing the customized drawing statements, the results typically will be erased.



Look-and-Feel Observation 13.4

*When overriding a **JComponent**'s **paintComponent** method, the first statement in the body should always be a call to the superclass's original version of the method.*



Common Programming Error 13.1

*When overriding a **JComponent**'s **paintComponent** method, not calling the superclass's original version of **paintComponent** might prevent the GUI component from displaying properly on the GUI.*



Common Programming Error 13.2

*When overriding a **JComponent**'s **paintComponent** method, calling the superclass's **paintComponent** method after other drawing is performed erases the other drawings.*

Classes **JFrame** and **JApplet** are not subclasses of **JComponent**; therefore, they do not contain method **paintComponent**. To draw directly on subclasses of **JFrame** and **JApplet**, override method **paint**.



Look-and-Feel Observation 13.5

*Calling **repaint** for a Swing GUI component indicates that the component should be painted as soon as possible. The background of the GUI component is cleared only if the component is opaque. Most Swing components are transparent by default. **JComponent** method **setOpaque** can be passed a **boolean** argument indicating whether the component is opaque (**true**) or transparent (**false**). The GUI components of package **java.awt** are different from Swing components, in that **repaint** results in a call to **Component** method **update** (which clears the component's background) and **update** calls method **paint** (rather than **paintComponent**).*

The program of Fig. 13.2 and Fig. 13.3 demonstrates a customized subclass of **JPanel**. Class **CustomPanel** (Fig. 13.2) has its own **paintComponent** method that draws a circle or a square, depending on the value passed to **CustomPanel**'s **draw** method. For this purpose, **CustomPanel** line 11 defines constants that enable the program to specify the shape a **CustomPanel** draws on itself with each call to its **paintComponent** method. Class **CustomPanelTest** (Fig. 13.3) creates a **CustomPanel** and a GUI that enable the user to choose which shape to draw.

Class **CustomPanel** contains one instance variable, **shape**, that stores an integer representing the shape to draw. Method **paintComponent** (lines 15–23) draws a shape on the panel. If **shape** is **CIRCLE**, **Graphics** method **fillOval** draws a solid circle. If **shape** is **SQUARE**, **Graphics** method **fillRect** draws a solid square. Method **draw** (lines 26–30) sets instance variable **shape** and calls **repaint** to refresh the **CustomPanel** object. Note that calling **repaint** (which is really **this.repaint()**) for the **CustomPanel** schedules a painting operation for the **CustomPanel**. Method **paintComponent** will be called to repaint the **CustomPanel** and draw the appropriate shape.

```

1  // Fig. 13.2: CustomPanel.java
2  // A customized JPanel class.
3
4  // Java core packages
5  import java.awt.*;
6
7  // Java extension packages
8  import javax.swing.*;
9
10 public class CustomPanel extends JPanel {
11     public final static int CIRCLE = 1, SQUARE = 2;
12     private int shape;
13
14     // use shape to draw an oval or rectangle
15     public void paintComponent( Graphics g )
16     {
17         super.paintComponent( g );
18
19         if ( shape == CIRCLE )
20             g.fillOval( 50, 10, 60, 60 );
21         else if ( shape == SQUARE )
22             g.fillRect( 50, 10, 60, 60 );
23     }
24
25     // set shape value and repaint CustomPanel
26     public void draw( int shapeToDraw )
27     {
28         shape = shapeToDraw;
29         repaint();
30     }
31
32 } // end class CustomPanel

```

Fig. 13.2 Defining a custom drawing area by subclassing **JPanel**.

Class **CustomPanelTest** (Fig. 13.3) instantiates a **CustomPanel** object (line 22 of its constructor) and sets its background color to green, so the **CustomPanel** area is visible on the application. Next, the constructor instantiates **JButton** objects **squareButton** and **circleButton**. Lines 27–40 register an event handler for **squareButton**'s **ActionEvent**. Lines 43–56 register an event handler for **circleButton**'s **ActionEvent**. Lines 35 and 51 each call **CustomPanel** method **draw**. In each case, the appropriate constant (**CustomPanel.SQUARE** or **CustomPanel.CIRCLE**) is passed as an argument to indicate which shape to draw.

```

1  // Fig. 13.3: CustomPanelTest.java
2  // Using a customized Panel object.
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 CustomPanelTest extends JFrame {
12     private JPanel buttonPanel;
13     private CustomPanel myPanel;
14     private JButton circleButton, squareButton;
15
16     // set up GUI
17     public CustomPanelTest()
18     {
19         super( "CustomPanel Test" );
20
21         // create custom drawing area
22         myPanel = new CustomPanel();
23         myPanel.setBackground( Color.green );
24
25         // set up squareButton
26         squareButton = new JButton( "Square" );
27         squareButton.addActionListener(
28
29             // anonymous inner class to handle squareButton events
30             new ActionListener() {
31
32                 // draw a square
33                 public void actionPerformed((ActionEvent event) )
34                 {
35                     myPanel.draw( CustomPanel.SQUARE );
36                 }
37
38             } // end anonymous inner class
39
40         ); // end call to addActionListener
41
42         circleButton = new JButton( "Circle" );

```

Fig. 13.3 Drawing on a customized subclass of class **JPanel** (part 1 of 2).

```

43     circleButton.addActionListener(
44
45         // anonymous inner class to handle circleButton events
46         new ActionListener() {
47
48             // draw a circle
49             public void actionPerformed((ActionEvent event) )
50             {
51                 myPanel.draw( CustomPanel.CIRCLE );
52             }
53
54         } // end anonymous inner class
55
56     ); // end call to addActionListener
57
58     // set up panel containing buttons
59     buttonPanel = new JPanel();
60     buttonPanel.setLayout( new GridLayout( 1, 2 ) );
61     buttonPanel.add( circleButton );
62     buttonPanel.add( squareButton );
63
64     // attach button panel & custom drawing area to content pane
65     Container container = getContentPane();
66     container.add( myPanel, BorderLayout.CENTER );
67     container.add( buttonPanel, BorderLayout.SOUTH );
68
69     setSize( 300, 150 );
70     setVisible( true );
71 }
72
73 // execute application
74 public static void main( String args[] )
75 {
76     CustomPanelTest application = new CustomPanelTest();
77
78     application.setDefaultCloseOperation(
79         JFrame.EXIT_ON_CLOSE );
80 }
81
82 } // end class CustomPanelTest

```

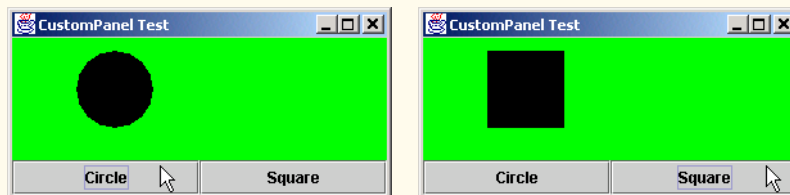


Fig. 13.3 Drawing on a customized subclass of class **JPanel** (part 2 of 2).

For layout of the buttons, **CustomPanelTest** creates **JPanel** **buttonPanel** with a **GridLayout** of one row and two columns (lines 59–60), then attaches the buttons to the panel (lines 61–62). Finally, **CustomPanelTest** adds **myPanel** to the **CENTER**

region of the content pane and adds **buttonPanel** to the **SOUTH** region of the content pane. Note that the **BorderLayout** expands **myPanel** to fill the center region.

13.4 Creating a Self-Contained Subclass of JPanel

JPanels do not support conventional events supported by other GUI components, like buttons, text fields and windows. However, **JPanel**s are capable of recognizing such lower-level events as mouse events and key events. The program of Fig. 13.4 and Fig. 13.5 allows the user to draw an oval on a subclass of **JPanel** by dragging the mouse across the panel. Class **SelfContainedPanel** (Fig. 13.4) listens for its own mouse events and draws an oval on itself in response to those mouse events. The location and size of the oval are determined from the coordinates of the mouse events. The coordinates at which the user presses the mouse button specify the starting point for the oval's bounding box. As the user drags the mouse, the coordinates of the mouse pointer specify another point. Together, the program uses these points to calculate the upper-left *x-y* coordinate, the width and the height of the oval's bounding box. The size of the oval changes continuously while the user drags the mouse. When the user releases the mouse button, the program calculates the final bounding box for the oval and draws the oval. Line 4 of Fig. 13.4 indicates that class **SelfContainedPanel** is in package **com.deitel.jhtp4.ch13** for future reuse. Class **SelfContainedPanelTest** imports **SelfContainedPanel** at line 13 of Fig. 13.5.

```

1  // Fig. 13.4: SelfContainedPanel.java
2  // A self-contained JPanel class that
3  // handles its own mouse events.
4  package com.deitel.jhtp4.ch13;
5
6  // Java core packages
7  import java.awt.*;
8  import java.awt.event.*;
9
10 // Java extension packages
11 import javax.swing.*;
12
13 public class SelfContainedPanel extends JPanel {
14     private int x1, y1, x2, y2;
15
16     // set up mouse event handling for SelfContainedPanel
17     public SelfContainedPanel()
18     {
19         // set up mouse listener
20         addMouseListener(
21
22             // anonymous inner class for mouse pressed and
23             // released event handling
24             new MouseAdapter() {
25
26                 // handle mouse press event
27                 public void mousePressed( MouseEvent event )
28                 {
29                     x1 = event.getX();

```

Fig. 13.4 Customized subclass of **JPanel** that processes mouse events (part 1 of 2).

```

30         y1 = event.getY();
31     }
32
33     // handle mouse release event
34     public void mouseReleased( MouseEvent event )
35     {
36         x2 = event.getX();
37         y2 = event.getY();
38         repaint();
39     }
40
41     } // end anonymous inner class
42
43 ); // end call to addMouseListener
44
45 // set up mouse motion listener
46 addMouseMotionListener(
47
48     // anonymous inner class to handle mouse drag events
49     new MouseMotionAdapter() {
50
51         // handle mouse drag event
52         public void mouseDragged( MouseEvent event )
53         {
54             x2 = event.getX();
55             y2 = event.getY();
56             repaint();
57         }
58
59     } // end anonymous inner class
60
61 ); // end call to addMouseMotionListener
62
63 } // end constructor
64
65 // return preferred width and height of SelfContainedPanel
66 public Dimension getPreferredSize()
67 {
68     return new Dimension( 150, 100 );
69 }
70
71 // paint an oval at the specified coordinates
72 public void paintComponent( Graphics g )
73 {
74     super.paintComponent( g );
75
76     g.drawOval( Math.min( x1, x2 ), Math.min( y1, y2 ),
77               Math.abs( x1 - x2 ), Math.abs( y1 - y2 ) );
78 }
79
80 } // end class SelfContainedPanel

```

Fig. 13.4 Customized subclass of **JPanel** that processes mouse events (part 2 of 2).

Class **SelfContainedPanel** (Fig. 13.4) extends class **JPanel**. Instance variables **x1** and **y1** store the initial coordinates where the **mousePressed** event occurs on the **SelfContainedPanel**. Instance variables **x2** and **y2** store the coordinates where the user drags the mouse or releases the mouse button. All the coordinates are with respect to the upper-left corner of the **SelfContainedPanel**.



Look-and-Feel Observation 13.6

Drawing on any GUI component is performed with coordinates that are measured from the upper-left corner (0, 0) of that GUI component.

The **SelfContainedPanel** constructor (lines 17–63) uses methods **addMouseListener** and **addMouseMotionListener** to register anonymous inner-class objects to handle mouse events and mouse motion events for the **SelfContainedPanel**. Only **mousePressed** (lines 27–31), **mouseReleased** (lines 34–39) and **mouseDragged** (lines 52–57) are overridden to perform tasks. The other mouse event-handling methods are inherited by the anonymous inner classes from the adapter classes **MouseAdapter** and **MouseMotionAdapter**.

By extending class **JPanel**, we are actually creating a new GUI component. Layout managers often use a GUI component's **getPreferredSize** method (inherited from class **java.awt.Component**) to determine the preferred width and height of a component when laying out that component as part of a GUI. If a new component has a preferred width and height, it should override method **getPreferredSize** (lines 66–69) to return that width and height as an object of class **Dimension** (package **java.awt**).



Look-and-Feel Observation 13.7

*The default size of a **JPanel** object is 10 pixels wide and 10 pixels tall.*



Look-and-Feel Observation 13.8

*When subclassing **JPanel** (or any other **JComponent**), override method **getPreferredSize** if the new component should have a specific preferred width and height.*

Method **paintComponent** (lines 72–78) draws an oval, using the current values of instance variables **x1**, **y1**, **x2** and **y2**. The width, height and upper-left corner are determined by the pressing and holding of the mouse button, the dragging of the mouse and releasing of the mouse button on the **SelfContainedPanel** drawing area.

The initial coordinates **x1** and **y1** on the **SelfContainedPanel** drawing area are captured in method **mousePressed** (lines 27–31). As the user drags the mouse after the initial **mousePressed** operation, the program generates a series of calls to **mouseDragged** (lines 52–57) while the user continues to hold the mouse button and move the mouse. Each call captures in variables **x2** and **y2** the current location of the mouse with respect to the upper-left corner of the **SelfContainedPanel** and calls **repaint** to draw the current version of the oval. Drawing is strictly confined to the **SelfContainedPanel**, even if the user drags outside the **SelfContainedPanel** drawing area. Anything drawn off the **SelfContainedPanel** is *clipped*—pixels are not displayed outside the bounds of the **SelfContainedPanel**.

The calculations provided in method **paintComponent** determine the proper upper-left corner, using method **Math.min** twice to find the smaller *x* coordinate and *y* coordinate. The oval's width and height must be positive values or the oval is not displayed.

Method **Math.abs** gets the absolute value of the subtractions **x1 - x2** and **y1 - y2** that determine the width and height of the oval's bounding rectangle, respectively. When the calculations are complete, **paintComponent** draws the oval. The call to the superclass version of **paintComponent** at the beginning of the method guarantees that the previous oval displayed on the **SelfContainedPanel** is erased before the new one is displayed.



Look-and-Feel Observation 13.9

Most Swing GUI components can be transparent or opaque. If a Swing GUI component is opaque, when its **paintComponent** method is called, its background will be cleared. Otherwise, its background will not be cleared. Only opaque components can display a customized background color.



Look-and-Feel Observation 13.10

JPanel objects are opaque by default.

When the user releases the mouse button, method **mouseReleased** (lines 34–39) captures in variables **x2** and **y2** the final location of the mouse and invokes **repaint** to draw the final version of the oval.

Class **SelfContainedPanelTest**'s constructor (lines 21–57 of Fig. 13.5) creates an instance of class **SelfContainedPanel** (line 24) and sets the background color (line 25) of the **SelfContainedPanel** to yellow so that its area is visible against the background of the application window.

```

1  // Fig. 13.5: SelfContainedPanelTest.java
2  // Creating a self-contained subclass of JPanel
3  // that processes its own mouse events.
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 // Deitel packages
13 import com.deitel.jhtp4.ch13.SelfContainedPanel;
14
15 public class SelfContainedPanelTest extends JFrame {
16     private SelfContainedPanel myPanel;
17
18
19     // set up GUI and mouse motion event handlers for
20     // application window
21     public SelfContainedPanelTest()
22     {
23         // set up a SelfContainedPanel
24         myPanel = new SelfContainedPanel();
25         myPanel.setBackground( Color.yellow );
26
27         Container container = getContentPane();

```

Fig. 13.5 Capturing mouse events with a **JPanel** (part 1 of 3).

```

28     container.setLayout( new FlowLayout() );
29     container.add( myPanel );
30
31     // set up mouse motion event handling
32     addMouseMotionListener(
33
34         // anonymous inner class for mouse motion event handling
35         new MouseMotionListener() {
36
37             // handle mouse drag event
38             public void mouseDragged( MouseEvent event )
39             {
40                 setTitle( "Dragging: x=" + event.getX() +
41                     "; y=" + event.getY() );
42             }
43
44             // handle mouse move event
45             public void mouseMoved( MouseEvent event )
46             {
47                 setTitle( "Moving: x=" + event.getX() +
48                     "; y=" + event.getY() );
49             }
50
51         } // end anonymous inner class
52
53     ); // end call to addMouseMotionListener
54
55     setSize( 300, 200 );
56     setVisible( true );
57 }
58
59 // execute application
60 public static void main( String args[] )
61 {
62     SelfContainedPanelTest application =
63         new SelfContainedPanelTest();
64
65     application.setDefaultCloseOperation(
66         JFrame.EXIT_ON_CLOSE );
67 }
68
69 } // end class SelfContainedPanelTest

```

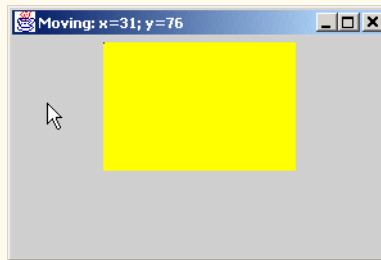


Fig. 13.5 Capturing mouse events with a **JPanel** (part 2 of 3).

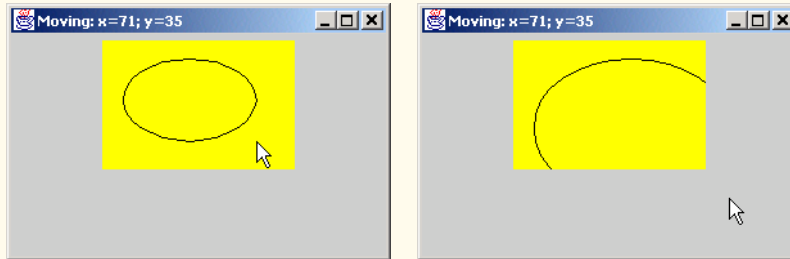


Fig. 13.5 Capturing mouse events with a **JPanel** (part 3 of 3).

We would like this program to distinguish between mouse motion events on the **SelfContainedPanel** and mouse motion events on the application window, so lines 32–53 register an object of an anonymous inner class to handle the application's mouse motion events. Event handlers **mouseDragged** (lines 38–42) and **mouseMoved** (lines 45–49) use method **setTitle** (inherited from class **java.awt.Frame**) to display a **String** in the window's title bar indicating the *x-y* coordinate where the mouse motion event occurred.

When executing this program, try dragging from the background of the application window into the **SelfContainedPanel** area to see that the drag events are sent to the application window rather than the **SelfContainedPanel**. Then, start a new drag operation in the **SelfContainedPanel** area and drag out to the background of the application window to see that the drag events are sent to the **SelfContainedPanel** rather than to the application window.



Look-and-Feel Observation 13.11

A mouse drag operation begins with a mouse-pressed event. All subsequent mouse drag events (until the user releases the mouse button) are sent to the GUI component that received the original mouse-pressed event.

13.5 JSlider

JSliders enable the user to select from a range of integer values. Class **JSlider** inherits from **JComponent**. Figure 13.6 shows a horizontal **JSlider** with *tick marks* and the *thumb* that allows the user to select a value. **JSliders** are highly customizable in that they can display *major tick marks*, *minor tick marks* and labels for the tick marks. They also support *snap-to ticks* where positioning the thumb between two tick marks causes the thumb to *snap* to the closest tick mark.



Fig. 13.6 Horizontal **JSlider** component.

Most Swing GUI components support user interactions through the mouse and the keyboard. For example, if a **JSlider** has the *focus* (i.e., it is the currently selected GUI component in the user interface), the *left arrow key* and *right arrow key* cause the thumb of the **JSlider** to decrease or increase by 1, respectively. The *down arrow key* and *up arrow key* also cause the thumb of the **JSlider** to decrease or increase by 1, respectively. The *PgDn key* (page down) and *PgUp key* (page up) cause the thumb of the **JSlider** to decrease or increase by *block increments* of one-tenth of the range of values, respectively. The *Home key* moves the thumb to the minimum value of the **JSlider** and the *End key* moves the thumb to the maximum value of the **JSlider**.



Look-and-Feel Observation 13.12

Most Swing components support user interactions through the mouse and the keyboard.

JSliders have either a *horizontal orientation* or a *vertical orientation*. For a horizontal **JSlider**, the minimum value is at the extreme left and the maximum value is at the extreme right of the **JSlider**. For a vertical **JSlider**, the minimum value is at the extreme bottom and the maximum value is at the extreme top of the **JSlider**. The relative position of the thumb indicates the current value of the **JSlider**.



Look-and-Feel Observation 13.13

The minimum and maximum value positions on a **JSlider** can be switched by calling the **JSlider** method **setInverted** with boolean argument **true**.

The program of Fig. 13.7 and Fig. 13.8 allows the user to size a circle drawn on a subclass of **JPanel** called **OvalPanel** (Fig. 13.7). The user specifies the diameter of the circle with a horizontal **JSlider**. Application class **SliderDemo** (Fig. 13.8) creates the **JSlider** that controls the diameter of the circle. Class **OvalPanel** is a subclass of **JPanel** that knows how to draw a circle on itself, using its own instance variable **diameter** to determine the diameter of the circle—the **diameter** is used as the width and height of the bounding box in which the circle is displayed. The **diameter** value is set when the user interacts with the **JSlider**. The event handler calls method **setDiameter** in class **OvalPanel** to set the **diameter** and calls **repaint** to draw the new circle. The **repaint** call results in a call to **OvalPanel**'s **paintComponent** method.

Class **OvalPanel** (Fig. 13.7) contains a **paintComponent** method (lines 14–19) that draws a filled oval (a circle in this example), a **setDiameter** method (lines 22–28) that changes the **diameter** of the circle and **repaints** the **OvalPanel**, a **getPreferredSize** method (lines 31–34) that defines the preferred width and height of an **OvalPanel** and a **getMinimumSize** method (lines 37–40) that defines the minimum width and height of an **OvalPanel**.



Look-and-Feel Observation 13.14

If a new GUI component has a minimum width and height (i.e., smaller dimensions would render the component ineffective on the display), override method **getMinimumSize** to return the minimum width and height as an instance of class **Dimension**.



Look-and-Feel Observation 13.15

For many GUI components, method **getMinimumSize** is defined to return the result of a call to that component's **getPreferredSize** method.

```

1  // Fig. 13.7: OvalPanel.java
2  // A customized JPanel class.
3
4  // Java core packages
5  import java.awt.*;
6
7  // Java extension packages
8  import javax.swing.*;
9
10 public class OvalPanel extends JPanel {
11     private int diameter = 10;
12
13     // draw an oval of the specified diameter
14     public void paintComponent( Graphics g )
15     {
16         super.paintComponent( g );
17
18         g.fillOval( 10, 10, diameter, diameter );
19     }
20
21     // validate and set diameter, then repaint
22     public void setDiameter( int newDiameter )
23     {
24         // if diameter invalid, default to 10
25         diameter = ( newDiameter >= 0 ? newDiameter : 10 );
26
27         repaint();
28     }
29
30     // used by layout manager to determine preferred size
31     public Dimension getPreferredSize()
32     {
33         return new Dimension( 200, 200 );
34     }
35
36     // used by layout manager to determine minimum size
37     public Dimension getMinimumSize()
38     {
39         return getPreferredSize();
40     }
41
42 } // end class OvalPanel

```

Fig. 13.7 Custom subclass of **JPanel** for drawing circles of a specified diameter.

Class **SliderDemo**'s constructor (lines 17–54 of Fig. 13.8) instantiates **OvalPanel** object **myPanel** and sets its background color (lines 22–23). Lines 26–27 instantiate **JSlider** object **diameterSlider** to control the diameter of the circle drawn on the **OvalPanel**. The orientation of **diameterSlider** is **HORIZONTAL** (a constant in interface **SwingConstants**). The second and third constructor arguments to the **JSlider** constructor indicate the minimum and maximum integer values in the range of values for this **JSlider**. The last constructor argument indicates that the initial value of the **JSlider** (i.e., where the thumb is displayed) should be 10.

```
1  // Fig. 13.8: SliderDemo.java
2  // Using JSliders to size an oval.
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10 import javax.swing.event.*;
11
12 public class SliderDemo extends JFrame {
13     private JSlider diameterSlider;
14     private OvalPanel myPanel;
15
16     // set up GUI
17     public SliderDemo()
18     {
19         super( "Slider Demo" );
20
21         // set up OvalPanel
22         myPanel = new OvalPanel();
23         myPanel.setBackground( Color.yellow );
24
25         // set up JSlider to control diameter value
26         diameterSlider =
27             new JSlider( SwingConstants.HORIZONTAL, 0, 200, 10 );
28         diameterSlider.setMajorTickSpacing( 10 );
29         diameterSlider.setPaintTicks( true );
30
31         // register JSlider event listener
32         diameterSlider.addChangeListener(
33
34             // anonymous inner class to handle JSlider events
35             new ChangeListener() {
36
37                 // handle change in slider value
38                 public void stateChanged( ChangeEvent e )
39                 {
40                     myPanel.setDiameter( diameterSlider.getValue() );
41                 }
42
43             } // end anonymous inner class
44
45         ); // end call to addChangeListener
46
47         // attach components to content pane
48         Container container = getContentPane();
49         container.add( diameterSlider, BorderLayout.SOUTH );
50         container.add( myPanel, BorderLayout.CENTER );
51
52         setSize( 220, 270 );
```

Fig. 13.8 Using a **JSlider** to determine the diameter of a circle (part 1 of 2).

```

53     setVisible( true );
54 }
55
56 // execute application
57 public static void main( String args[] )
58 {
59     SliderDemo application = new SliderDemo();
60
61     application.setDefaultCloseOperation(
62         JFrame.EXIT_ON_CLOSE );
63 }
64
65 } // end class SliderDemo

```

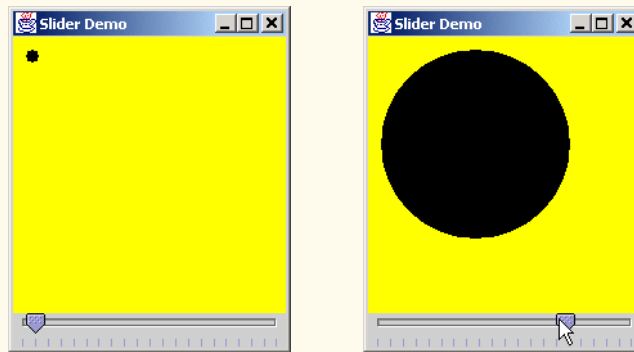


Fig. 13.8 Using a **JSlider** to determine the diameter of a circle (part 2 of 2).

Lines 28–29 customize the appearance of the **JSlider**. Method **setMajorTickSpacing** indicates that each tick mark represents 10 values in the range of values supported by the **JSlider**. Method **setPaintTicks** with a **true** argument indicates that the tick marks should be displayed (they are not displayed by default). See the **JSlider** on-line documentation for more information on methods that are used to customize a **JSlider**'s appearance.

JSliders generate **ChangeEvent**s (package **javax.swing.event**) when the user interacts with a **JSlider**. An object of a class that implements interface **ChangeListener** (package **javax.swing.event**) and defines method **stateChanged** can respond to **ChangeEvent**s. Lines 32–45 register an object of an anonymous inner class that implements **ChangeListener** to handle **diameterSlider**'s events. When method **stateChanged** is called in response to a user interaction, it calls **myPanel**'s **setDiameter** method and passes the current value of the **JSlider** as an argument. Method **getValue** of class **JSlider** returns the current thumb position.

13.6 Windows

From Chapter 9 to this chapter, most applications have used an instance of a subclass of **JFrame** as the application window. In this section, we discuss several important issues regarding **JFrames**.

A **JFrame** is a *window* with a *title bar* and a *border*. Class **JFrame** is a subclass of **java.awt.Frame** (which is a subclass of **java.awt.Window**). As such, **JFrame** is one of the few Swing GUI components that is not a lightweight GUI component. Unlike most Swing components, **JFrame** is not written completely in Java. In fact, when you display a window from a Java program, the window is provided by the local platform's set of GUI components—the window will look like all other windows displayed on that platform. When a Java program executes on a Macintosh and displays a window, the window's title bar and borders will look like other Macintosh applications. When a Java program executes on Microsoft Windows and displays a window, the window's title bar and borders will look like other Microsoft Windows applications. And when a Java program executes on a Unix platform and displays a window, the window's title bar and borders will look like other Unix applications on that platform.

Class **JFrame** supports three operations when the user closes the window. By default, a window is hidden (i.e., removed from the screen) when the user closes a window. This can be controlled with **JFrame** method **setDefaultCloseOperation**. Interface **WindowConstants** (package **javax.swing**) defines three constants for use with this method—**DISPOSE_ON_CLOSE**, **DO_NOTHING_ON_CLOSE** and **HIDE_ON_CLOSE** (the default). Most platforms only allow a limited number of windows to be displayed on the screen. As such, a window is a valuable resource that should be given back to the system when it is no longer needed. Class **Window** (an indirect superclass of **JFrame**) defines method **dispose** for this purpose. When a **Window** is no longer needed in an application, you should explicitly **dispose** of the **Window**. This can be done by calling the **Window**'s **dispose** method or by calling method **setDefaultCloseOperation** with the argument **WindowConstants.DISPOSE_ON_CLOSE**. Also, terminating an application will return window resources to the system. Setting the default close operation to **DO_NOTHING_ON_CLOSE** indicates that you will determine what to do when the user indicates that the window should be closed.



Software Engineering Observation 13.1

Windows are a valuable system resource that should be returned to the system when they are no longer needed.

By default, a window is not displayed on the screen until the program invokes the window's **setVisible** method (inherited from class **java.awt.Component**) with a **true** argument or invokes the window's **show** method, which takes no arguments. Also, a window's size should be set with a call to method **setSize** (inherited from class **java.awt.Component**). The position of a window when it appears on the screen is specified with method **setLocation** (inherited from class **java.awt.Component**).



Common Programming Error 13.3

*Forgetting to call method **show** or method **setVisible** on a window is a run-time logic error; the window is not displayed.*



Common Programming Error 13.4

*Forgetting to call the **setSize** method on a window is a run-time logic error—only the title bar appears.*

All windows generate *window events* when the user manipulates the window. Event listeners are registered for window events with method **addWindowListener** of class

Window. Interface **WindowListener** (implemented by window event listeners) provides seven methods for handling window events—**windowActivated** (called when the user makes a window the active window), **windowClosed** (called after the window is closed), **windowClosing** (called when the user initiates closing of the window), **windowDeactivated** (called when the user makes another window the active window), **windowIconified** (called when the user minimizes a window), **windowDeiconified** (called when the user restores a window from being minimized) and **windowOpened** (called when a program first displays a window on the screen).

Most windows have an icon at the top-left or top-right corner that enables a user to close the window and terminate a program. Most windows also have an icon in the upper-left corner of the window that displays a menu when the user clicks the icon. This menu normally contains a **Close** option to close the window and several other options for manipulating the window.

13.7 Designing Programs that Execute as Applets or Applications

It is sometimes desirable to design a Java program that can execute both as a stand-alone application and as an applet in a Web browser. Such a program can be used to provide the same functionality to users worldwide by making the applet available for download via Web and can be installed on a computer as a stand-alone application. The next example discusses how to create a small program that can execute both as an applet and as an application. [Note: In Chapter 16 and Chapter 17, we discuss several issues that make applets different from applications and security restrictions placed on applets that prevent certain application features from working in an applet.]

Frequently, programs use **JFrames** to create *GUI-based applications*. The **JFrame** provides the space in which the application GUI appears. When the user closes the **JFrame**, the application terminates. In this section, we demonstrate how to convert an applet into a GUI-based application. The program of Fig. 13.9 presents an applet that also can be executed as an application.



Software Engineering Observation 13.2

When designing a program to execute as both an applet and an application, begin by defining it as an applet, because applets have limitations due to security restrictions imposed on them by Web browsers. If the program executes properly as an applet, it can be made to work properly as an application. However, the reverse is not always true.

Our applet class **DrawShapes** presents the user with three buttons which, when pressed, cause an instance of class **DrawPanel** (lines 121–126) to draw a random line, rectangle or oval (depending on which button is pressed). The applet does not contain any new features as far as GUI components, layouts or drawing are concerned. The only new feature is that the **DrawShapes** class now also contains a **main** method (lines 60–99) that can be used to execute the program as an application. We discuss this method in detail below.

The HTML document that loads the applet into the **appletviewer** or a Web browser specifies the applet's width and height as 300 and 200, respectively. When the program executes as an application with the **java** interpreter, you can supply arguments to the program (called *command-line arguments*) that specify the width and height of the application window. For example, the command

```
java DrawShapes 600 400
```

contains two command-line arguments—600 and 400—that specify the width and height of the application window. Java passes the command-line arguments to **main** as the array of **Strings** called **args**, which we have declared in the parameter list of every application's **main** method, but not used until this point. The first argument after the application class name is the first **String** in the array **args**, and the length of the array is the total number of command-line arguments. Line 60 begins the definition of **main** and declares array **args** as an array of **Strings** that allows the application to access the command-line arguments. Line 62 defines variables **width** and **height** that are used to specify the size of the application window.

```

1  // Fig. 13.9: DrawShapes.java
2  // Draw random lines, rectangles and ovals
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 DrawShapes extends JApplet {
12     private JButton choices[];
13     private String names[] = { "Line", "Rectangle", "Oval" };
14     private JPanel buttonPanel;
15     private DrawPanel drawingPanel;
16     private int width = 300, height = 200;
17
18     // initialize applet; set up GUI
19     public void init()
20     {
21         // set up DrawPanel
22         drawingPanel = new DrawPanel( width, height );
23
24         // create array of buttons
25         choices = new JButton[ names.length ];
26
27         // set up panel for buttons
28         buttonPanel = new JPanel();
29         buttonPanel.setLayout(
30             new GridLayout( 1, choices.length ) );
31
32         // set up buttons and register their listeners
33         ButtonHandler handler = new ButtonHandler();
34
35         for ( int count = 0; count < choices.length; count++ ) {
36             choices[ count ] = new JButton( names[ count ] );
37             buttonPanel.add( choices[ count ] );
38             choices[ count ].addActionListener( handler );
39         }

```

Fig. 13.9 Creating a GUI-based application from an applet (part 1 of 4).

```

40
41     // attach components to content pane
42     Container container = getContentPane();
43     container.add( buttonPanel, BorderLayout.NORTH );
44     container.add( drawingPanel, BorderLayout.CENTER );
45 }
46
47 // enables application to specify width of drawing area
48 public void setWidth( int newWidth )
49 {
50     width = ( newWidth >= 0 ? newWidth : 300 );
51 }
52
53 // enables application to specify height of drawing area
54 public void setHeight( int newHeight )
55 {
56     height = ( newHeight >= 0 ? newHeight : 200 );
57 }
58
59 // execute applet as an application
60 public static void main( String args[] )
61 {
62     int width, height;
63
64     // check for command-line arguments
65     if ( args.length != 2 ) {
66         width = 300;
67         height = 200;
68     }
69     else {
70         width = Integer.parseInt( args[ 0 ] );
71         height = Integer.parseInt( args[ 1 ] );
72     }
73
74     // create window in which applet will execute
75     JFrame applicationWindow =
76         new JFrame( "An applet running as an application" );
77
78     applicationWindow.setDefaultCloseOperation(
79         JFrame.EXIT_ON_CLOSE );
80
81     // create one applet instance
82     DrawShapes appletObject = new DrawShapes();
83     appletObject.setWidth( width );
84     appletObject.setHeight( height );
85
86     // call applet's init and start methods
87     appletObject.init();
88     appletObject.start();
89
90     // attach applet to center of window
91     applicationWindow.getContentPane().add( appletObject );
92

```

Fig. 13.9 Creating a GUI-based application from an applet (part 2 of 4).

```

93      // set the window's size
94      applicationWindow.setSize( width, height );
95
96      // showing the window causes all GUI components
97      // attached to the window to be painted
98      applicationWindow.setVisible( true );
99  }
100
101  // private inner class to handle button events
102  private class ButtonHandler implements ActionListener {
103
104      // determine button user pressed and set drawing area's
105      // current choice
106      public void actionPerformed((ActionEvent event) )
107      {
108          for ( int count = 0; count < choices.length; count++ )
109
110              if ( event.getSource() == choices[ count ] ) {
111                  drawingPanel.setCurrentChoice( count );
112                  break;
113              }
114      }
115  } // end private inner class ButtonHandler
116
117  } // end class DrawShapes
118
119  // subclass of JPanel to allow drawing in a separate area
120  class DrawPanel extends JPanel {
121      private int currentChoice = -1; // don't draw first time
122      private int width = 100, height = 100;
123
124      // initialize width and height of DrawPanel
125      public DrawPanel( int newWidth, int newHeight )
126      {
127          width = ( newWidth >= 0 ? newWidth : 100 );
128          height = ( newHeight >= 0 ? newHeight : 100 );
129      }
130
131      // draw line, rectangle or oval based on user's choice
132      public void paintComponent( Graphics g )
133      {
134          super.paintComponent( g );
135
136          switch( currentChoice ) {
137
138              case 0:
139                  g.drawLine( randomX(), randomY(),
140                      randomX(), randomY() );
141                  break;
142          }
143  }

```

Fig. 13.9 Creating a GUI-based application from an applet (part 3 of 4).

```

144         case 1:
145             g.drawRect( randomX(), randomY(),
146                 randomX(), randomY() );
147             break;
148
149         case 2:
150             g.drawOval( randomX(), randomY(),
151                 randomX(), randomY() );
152             break;
153     }
154
155 } // end method paintComponent
156
157 // specify current shape choice and repaint
158 public void setCurrentChoice( int choice )
159 {
160     currentChoice = choice;
161     repaint();
162 }
163
164 // pick random x coordinate
165 private int randomX()
166 {
167     return ( int ) ( Math.random() * width );
168 }
169
170 // pick random y coordinate
171 private int randomY()
172 {
173     return ( int ) ( Math.random() * height );
174 }
175
176 } // end class DrawPanel

```

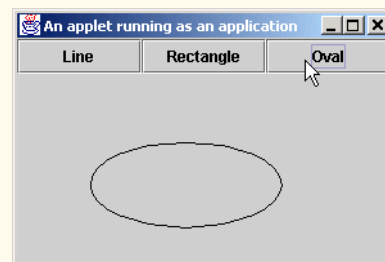
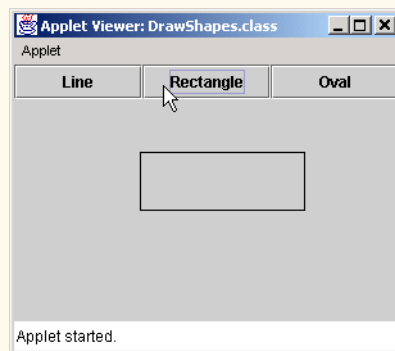


Fig. 13.9 Creating a GUI-based application from an applet (part 4 of 4).

Lines 65–72 determine the initial width and height of the application window. The **if** condition determines the length of array **args**. If the number of elements is not 2, the **width** and **height** are set to 300 and 200 by default. Otherwise, lines 70–71 convert the command-line arguments from **Strings** to **int** values with **parseInt** and use them as the **width**

and **height**. [Note: This program assumes that the user inputs whole-number values for the command-line arguments; if not, an exception will occur. In Chapter 14, we discuss how to make our programs more robust by dealing with improper values when they occur.]

When an applet executes, the window in which it executes is supplied by the applet container (i.e., the **appletviewer** or browser). When a program executes as an application, the application must create its own window (if one is required). Lines 75–76 create the **JFrame** to which the program will attach the applet. As with any **JFrame** that is used as the application's primary window, you should provide a mechanism to terminate the application. Lines 78–79 specify that the application should terminate when the user closes the window.



Software Engineering Observation 13.3

To execute an applet as an application, the application must provide a window in which the applet can be displayed.

When an applet executes in an applet container, the container creates one object of the applet class to execute the applet's tasks. In an application, objects are not created unless the application explicitly contains statements that create objects. Line 82 defines one instance of applet class **DrawShapes**. Notice the call to the no-argument constructor. We did not define a constructor in class **DrawShapes** (applet classes typically do not define constructors). Remember that the compiler provides a default constructor for a class that does not define any constructors. Lines 83–84 call **DrawShapes** methods **setWidth** and **setHeight** to validate the values for **width** and **height** (improper values are set to 300 and 200, respectively).



Software Engineering Observation 13.4

To execute an applet as an application, the application must create an instance of the applet class to execute.

When an applet executes in an applet container, the container guarantees that methods **init**, **start** and **paint** will be called to begin the applet's execution. However, these methods are not special to an application. Method **init** and **start** are not invoked automatically or required by an application. (Method **paint** is part of any window and will be called when it is necessary to repaint the window.) Lines 87–88 invoke **appletObject**'s **init** method to initialize the applet and set up its GUI, then invoke method **start**. [Note: In our example, we did not override method **start**. It is called here to mimic the start-up sequence normally followed for an applet.]



Software Engineering Observation 13.5

*When executing an applet as an application, the application must call **init** and **start** explicitly to simulate the normal applet start-up sequence of method calls.*

When an applet executes in an applet container, the applet is normally attached to the applet container's window. An application must explicitly attach an applet object to the application window. Line 91 obtains a reference to the **applicationWindow**'s content pane and adds the **appletObject** to the default **CENTER** of the content pane's **BorderLayout**. The **appletObject** will occupy the entire window.



Software Engineering Observation 13.6

When one is executing an applet as an application, the application must attach the applet object to its window.

Finally, the application window must be sized and displayed on the screen. Line 94 sets the application window's size, and line 98 displays the window. When a Java program displays any window, all the components attached to the window receive calls to their **paint** methods (if they are heavyweight components) or their **paintComponent** methods (if they are lightweight components). Thus, displaying the application window results in a call to the applet's **paint** method to complete the normal start-up sequence for the applet.

Try executing this program as an applet and as an application to see that it has the same functionality when executed.

13.8 Using Menus with Frames

Menus are an integral part of GUIs. Menus allow the user to perform actions without unnecessarily “cluttering” a graphical user interface with extra GUI components. In Swing GUIs, menus can be attached only to objects of the classes that provide method **setJMenuBar**. Two such classes are **JFrame** and **JApplet**. The classes used to define menus are **JMenuBar**, **JMenuItem**, **JMenu**, **JCheckBoxMenuItem** and class **JRadioButtonMenuItem**.



Look-and-Feel Observation 13.16

Menus simplify GUIs by reducing the number of components the user views.

Class **JMenuBar** (a subclass of **JComponent**) contains the methods necessary to manage a *menu bar*, which is a container for menus.

Class **JMenuItem** (a subclass of **javax.swing.AbstractButton**) contains the methods necessary to manage *menu items*. A menu item is a GUI component inside a menu that, when selected, causes an action to be performed. A menu item can be used to initiate an action or it can be a *submenu* that provides more menu items from which the user can select. Submenus are useful for grouping related menu items in a menu.

Class **JMenu** (a subclass of **javax.swing.JMenuItem**) contains the methods necessary for managing *menus*. Menus contain menu items and are added to menu bars or to other menus as submenus. When a menu is clicked, the menu expands to show its list of menu items. Clicking a menu item generates an action event.

Class **JCheckBoxMenuItem** (a subclass of **javax.swing.JMenuItem**) contains the methods necessary to manage menu items that can be toggled on or off. When a **JCheckBoxMenuItem** is selected, a check appears to the left of the menu item. When the **JCheckBoxMenuItem** is selected again, the check to the left of the menu item is removed.

Class **JRadioButtonMenuItem** (a subclass of **javax.swing.JMenuItem**) contains the methods necessary to manage menu items that can be toggled on or off like **JCheckBoxMenuItem**s. When multiple **JRadioButtonMenuItem**s are maintained as part of a **ButtonGroup**, only one item in the group can be selected at a given time. When a **JRadioButtonMenuItem** is selected, a filled circle appears to the left of the menu item. When another **JRadioButtonMenuItem** is selected, the filled circle to the left of the previously selected menu item is removed.

The application of Fig. 13.10 demonstrates various types of menu items. The program also demonstrates how to specify special characters called mnemonics that can provide quick access to a menu or menu item from the keyboard. Mnemonics can be used with objects of all classes that have subclass **javax.swing.AbstractButton**.

```
1  // Fig. 13.10: MenuTest.java
2  // Demonstrating menus
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 MenuTest extends JFrame {
12     private Color colorValues[] =
13         { Color.black, Color.blue, Color.red, Color.green };
14
15     private JRadioButtonMenuItem colorItems[], fonts[];
16     private JCheckBoxMenuItem styleItems[];
17     private JLabel displayLabel;
18     private ButtonGroup fontGroup, colorGroup;
19     private int style;
20
21     // set up GUI
22     public MenuTest()
23     {
24         super( "Using JMenus" );
25
26         // set up File menu and its menu items
27         JMenu fileMenu = new JMenu( "File" );
28         fileMenu.setMnemonic( 'F' );
29
30         // set up About... menu item
31         JMenuItem aboutItem = new JMenuItem( "About..." );
32         aboutItem.setMnemonic( 'A' );
33
34         aboutItem.addActionListener(
35
36             // anonymous inner class to handle menu item event
37             new ActionListener() {
38
39                 // display message dialog when user selects About...
40                 public void actionPerformed((ActionEvent event) )
41                 {
42                     JOptionPane.showMessageDialog( MenuTest.this,
43                         "This is an example\nof using menus",
44                         "About", JOptionPane.PLAIN_MESSAGE );
45                 }
46
47             } // end anonymous inner class
48
49         ); // end call to addActionListener
50
51         fileMenu.add( aboutItem );
52     }
```

Fig. 13.10 Using JMenus and mnemonics (part 1 of 5).

```
53 // set up Exit menu item
54 JMenuItem exitItem = new JMenuItem( "Exit" );
55 exitItem.setMnemonic( 'x' );
56
57 exitItem.addActionListener(
58
59     // anonymous inner class to handle exitItem event
60     new ActionListener() {
61
62         // terminate application when user clicks exitItem
63         public void actionPerformed((ActionEvent event) )
64         {
65             System.exit( 0 );
66         }
67
68     } // end anonymous inner class
69
70 ); // end call to addActionListener
71
72 fileMenu.add( exitItem );
73
74 // create menu bar and attach it to MenuTest window
75 JMenuBar bar = new JMenuBar();
76 setJMenuBar( bar );
77 bar.add( fileMenu );
78
79 // create Format menu, its submenus and menu items
80 JMenu formatMenu = new JMenu( "Format" );
81 formatMenu.setMnemonic( 'r' );
82
83 // create Color submenu
84 String colors[] = { "Black", "Blue", "Red", "Green" };
85
86 JMenu colorMenu = new JMenu( "Color" );
87 colorMenu.setMnemonic( 'C' );
88
89 colorItems = new JRadioButtonMenuItem[ colors.length ];
90 colorGroup = new ButtonGroup();
91 ItemHandler itemHandler = new ItemHandler();
92
93 // create color radio button menu items
94 for ( int count = 0; count < colors.length; count++ ) {
95     colorItems[ count ] =
96         new JRadioButtonMenuItem( colors[ count ] );
97
98     colorMenu.add( colorItems[ count ] );
99     colorGroup.add( colorItems[ count ] );
100
101     colorItems[ count ].addActionListener( itemHandler );
102 }
103
104 // select first Color menu item
105 colorItems[ 0 ].setSelected( true );
```

Fig. 13.10 Using **JMenus** and mnemonics (part 2 of 5).

```
106
107 // add format menu to menu bar
108 formatMenu.add( colorMenu );
109 formatMenu.addSeparator();
110
111 // create Font submenu
112 String fontNames[] = { "Serif", "Monospaced", "SansSerif" };
113
114 JMenu fontMenu = new JMenu( "Font" );
115 fontMenu.setMnemonic( 'n' );
116
117 fonts = new JRadioButtonMenuItem[ fontNames.length ];
118 fontGroup = new ButtonGroup();
119
120 // create Font radio button menu items
121 for ( int count = 0; count < fontNames.length; count++ ) {
122     fonts[ count ] =
123         new JRadioButtonMenuItem( fontNames[ count ] );
124
125     fontMenu.add( fonts[ count ] );
126     fontGroup.add( fonts[ count ] );
127
128     fonts[ count ].addActionListener( itemHandler );
129 }
130
131 // select first Font menu item
132 fonts[ 0 ].setSelected( true );
133
134 fontMenu.addSeparator();
135
136 // set up style menu items
137 String styleNames[] = { "Bold", "Italic" };
138
139 styleItems = new JCheckBoxMenuItem[ styleNames.length ];
140 StyleHandler styleHandler = new StyleHandler();
141
142 // create style checkbox menu items
143 for ( int count = 0; count < styleNames.length; count++ ) {
144     styleItems[ count ] =
145         new JCheckBoxMenuItem( styleNames[ count ] );
146
147     fontMenu.add( styleItems[ count ] );
148
149     styleItems[ count ].addItemListener( styleHandler );
150 }
151
152 // put Font menu in Format menu
153 formatMenu.add( fontMenu );
154
155 // add Format menu to menu bar
156 bar.add( formatMenu );
157
```

Fig. 13.10 Using **JMenu**s and mnemonics (part 3 of 5).

```

158     // set up label to display text
159     displayLabel = new JLabel(
160         "Sample Text", SwingConstants.CENTER );
161     displayLabel.setForeground( colorValues[ 0 ] );
162     displayLabel.setFont(
163         new Font( "TimesRoman", Font.PLAIN, 72 ) );
164
165     getContentPane().setBackground( Color.cyan );
166     getContentPane().add( displayLabel, BorderLayout.CENTER );
167
168     setSize( 500, 200 );
169     setVisible( true );
170
171 } // end constructor
172
173 // execute application
174 public static void main( String args[] )
175 {
176     MenuTest application = new MenuTest();
177
178     application.setDefaultCloseOperation(
179         JFrame.EXIT_ON_CLOSE );
180 }
181
182 // inner class to handle action events from menu items
183 private class ItemHandler implements ActionListener {
184
185     // process color and font selections
186     public void actionPerformed((ActionEvent event) )
187     {
188         // process color selection
189         for ( int count = 0; count < colorItems.length; count++ )
190
191             if ( colorItems[ count ].isSelected() ) {
192                 displayLabel.setForeground( colorValues[ count ] );
193                 break;
194             }
195
196         // process font selection
197         for ( int count = 0; count < fonts.length; count++ )
198
199             if ( event.getSource() == fonts[ count ] ) {
200                 displayLabel.setFont( new Font(
201                     fonts[ count ].getText(), style, 72 ) );
202                 break;
203             }
204
205         repaint();
206     }
207
208 } // end class ItemHandler
209

```

Fig. 13.10 Using JMenus and mnemonics (part 4 of 5).

```

210 // inner class to handle item events from check box menu items
211 private class StyleHandler implements ItemListener {
212
213     // process font style selections
214     public void itemStateChanged( ItemEvent e )
215     {
216         style = 0;
217
218         // check for bold selection
219         if ( styleItems[ 0 ].isSelected() )
220             style += Font.BOLD;
221
222         // check for italic selection
223         if ( styleItems[ 1 ].isSelected() )
224             style += Font.ITALIC;
225
226         displayLabel.setFont( new Font(
227             displayLabel.getFont().getName(), style, 72 ) );
228
229         repaint();
230     }
231
232 } // end class StyleHandler
233
234 } // end class MenuTest

```

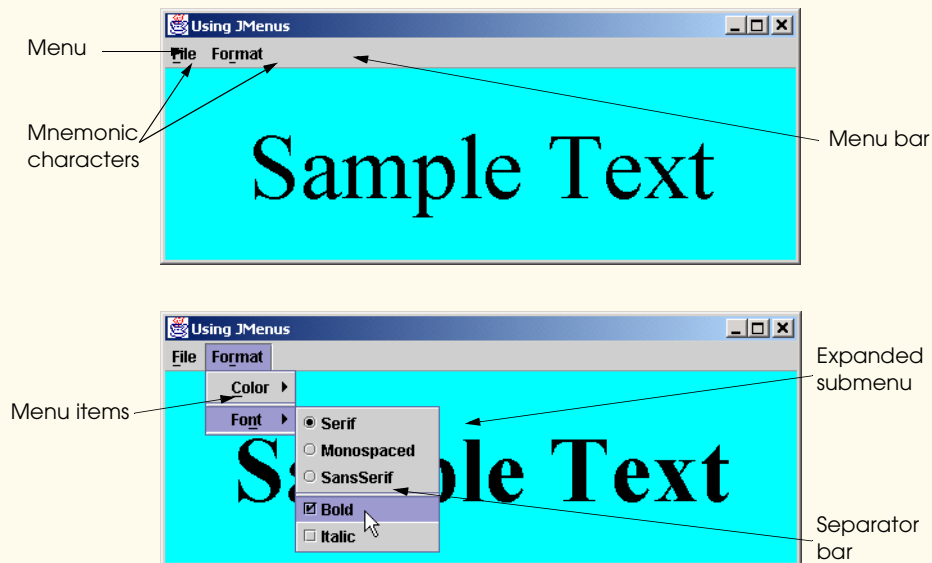


Fig. 13.10 Using **JMenus** and mnemonics (part 5 of 5).

Class **MenuTest** (line 11) is a completely self-contained class—it defines all the GUI components and event handling for the menu items. Most of the code for this application appears in the class's constructor (lines 22–171).

Lines 27–72 set up the **File** menu and attach it to the menu bar. The **File** menu contains an **About...** menu item that displays a message dialog when the menu item is selected and an **Exit** menu item that can be selected to terminate the application.

Line 27 creates **fileMenu** and passes to the constructor the string “**File**” as the name of the menu. Line 28 uses **AbstractButton** method **setMnemonic** (inherited into class **JMenu**) to indicate that **F** is the *mnemonic* for this menu. Pressing the **Alt** key and the letter **F** opens the menu, just as clicking the menu name with the mouse would. In the GUI, the mnemonic character in the menu’s name is displayed with an underline (see the screen captures).



Look-and-Feel Observation 13.17

Mnemonics provide quick access to menu commands and button commands through the keyboard.



Look-and-Feel Observation 13.18

*Different mnemonics should be used for each button or menu item. Normally, the first letter in the label on the menu item or button is used as the mnemonic. If multiple buttons or menu items start with the same letter, choose the next most prominent letter in the name (e.g., **x** is commonly chosen for a button or menu item called **Exit**).*

Lines 31–32 define **JMenuItem** **aboutItem** with the name “**About...**” and set its mnemonic to the letter **A**. This menu item is added to **fileMenu** at line 51. To access the **About...** item through the keyboard, press the **Alt** key and letter **F** to open the **File** menu, then press **A** to select the **About...** menu item. Lines 34–49 create an **ActionListener** to process **aboutItem**’s action event. Lines 42–44 display a message dialog box. In most prior uses of **showMessageDialog**, the first argument has been **null**. The purpose of the first argument is to specify the *parent window* for the dialog box. The parent window helps determine where the dialog box will be displayed. If the parent window is specified as **null**, the dialog box appears in the center of the screen. If the parent window is not **null**, the dialog box appears centered over the specified parent window. In this example, the program specifies the parent window with **MenuTest.this**—the **this** reference of class **MenuTest**. When using the **this** reference in an inner class, specifying **this** by itself refers to the inner-class object. To reference the outer-class object’s **this** reference, qualify **this** with the outer-class name and a dot operator (**.**).

Dialog boxes can be either *modal* or *modeless*. A *modal dialog box* does not allow any other window in the application to be accessed until the dialog box is dismissed. A *modeless dialog box* allows other windows to be accessed while the dialog is displayed. By default, the dialogs displayed with class **JOptionPane** are modal dialogs. Class **JDialog** can be used to create your own modeless or modal dialogs.

Lines 54–72 define menu item **exitItem**, set its mnemonic to **x**, register an **ActionListener** that terminates the application when the user selects **exitItem** and add **exitItem** to the **fileMenu**.

Lines 75–77 create the **JMenuBar**, attach it to the application window with **JFrame** method **setJMenuBar** and use **JMenuBar** method **add** to attach the **fileMenu** to the menu bar.



Common Programming Error 13.5

*Forgetting to set the menu bar with **JFrame** method **setJMenuBar** results in the menu bar not being displayed on the **JFrame**.*



Look-and-Feel Observation 13.19

*Menus normally appear left to right in the order that they are added to a **JMenuBar**.*

Lines 80–81 create menu **formatMenu** and set its mnemonic to **r** (**F** is not used because that is the **File** menu's mnemonic).

Lines 86–87 create menu **colorMenu** (this will be a submenu in the **Format** menu) and set its mnemonic to **C**. Line 89 creates **JRadioButtonMenuItem** array **colorItems** that refers to the menu items in **colorMenu**. Line 90 creates the **ButtonGroup** **colorGroup**, which ensures that only one of the menu items in the **Color** submenu is selected at a time. Line 91 defines an instance of inner class **ItemHandler** (defined at lines 183–208) that responds to selections from the **Color** submenu and the **Font** submenu (discussed shortly). The **for** structure at lines 94–102 creates each **JRadioButtonMenuItem** in array **colorItems**, adds each menu item to **colorMenu**, adds each menu item to **colorGroup** and registers the **ActionListener** for each menu item.

Line 105 uses **AbstractButton** method **setSelected** to select the first element in the **colorItems** array. Line 108 adds the **colorMenu** as a submenu of the **formatMenu**.



Look-and-Feel Observation 13.20

Adding a menu as a menu item in another menu automatically makes the added menu a submenu. When the mouse is positioned over a submenu (or the submenu's mnemonic is pressed), the submenu expands to show its menu items.

Line 109 adds a *separator* line to the menu. The separator appears as a horizontal line in the menu.



Look-and-Feel Observation 13.21

Separators can be added to a menu to group menu items logically.



Look-and-Feel Observation 13.22

*Any lightweight GUI component (i.e., a component that subclasses **JComponent**) can be added to a **JMenu** or to a **JMenuBar**.*

Lines 114–132 create the **Font** submenu and several **JRadioButtonMenuItems** and select the first element of **JRadioButtonMenuItem** array **fonts**. Line 139 creates a **JCheckBoxMenuItem** array to represent the menu items for specifying bold and italic styles for the fonts. Line 140 defines an instance of inner class **StyleHandler** (defined at lines 211–232) to respond to the **JCheckBoxMenuItem** events. The **for** structure at lines 143–150 creates each **JCheckBoxMenuItem**, adds each menu item to **fontMenu** and registers the **ItemListener** for each menu item. Line 153 adds **fontMenu** as a submenu of **formatMenu**. Line 156 adds the **formatMenu** to **bar**.

Lines 159–163 create a **JLabel** for which the **Format** menu items control the font, font color and font style. The initial foreground color is set to the first element of array **colorValues** (**Color.black**) and the initial font is set to **TimesRoman** with **PLAIN** style and **72**-point size. Line 165 sets the background color of the window's content pane to **Color.cyan**, and line 166 attaches the **JLabel** to the **CENTER** of the content pane's **BorderLayout**.

Method **actionPerformed** of class **ItemHandler** (lines 186–206) uses two **for** structures to determine which font or color menu item generated the event and sets the font or color of the **JLabel display**, respectively. The **if** condition at line 191 uses **AbstractButton** method **isSelected** to determine the selected **JRadioButtonMenuItem**. The **if** condition at line 199 uses **EventSource** method **getSource** to get a reference to the **JRadioButtonMenuItem** that generated the event. Line 201 uses **AbstractButton** method **getText** to obtain the name of the font from the menu item.

The program calls method **itemStateChanged** of class **StyleHandler** (lines 214–230) if the user selects a **JCheckBoxMenuItem** in the **fontMenu**. Lines 219 and 223 determine whether either or both of the **JCheckBoxMenuItems** are selected and use their combined state to determine the new style of the font.

13.9 Using JPopupMenu

Many of today's computer applications provide so-called *context-sensitive popup menus*. In Swing, such menus are created with class **JPopupMenu** (a subclass of **JComponent**). These menus provide options that are specific to the component for which the *popup trigger event* was generated. On most systems, the popup trigger event occurs when the user presses and releases the right mouse button.



Look-and-Feel Observation 13.23

The popup trigger event is platform specific. On most platforms that use a mouse with multiple mouse buttons, the popup trigger event occurs when the user clicks the right mouse button.

Figure 13.11 creates a **JPopupMenu** that allows the user to select one of three colors and change the background color of the window. When the user clicks the right mouse button on the **PopupTest** window's background, a **JPopupMenu** containing colors appears. If the user clicks one of the **JRadioButtonMenuItems** that represents a color, method **actionPerformed** of class **ItemHandler** changes the background color of the window's content pane.

```

1  // Fig. 13.11: PopupTest.java
2  // Demonstrating JPopupMenu
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 PopupTest extends JFrame {
12
13     private JRadioButtonMenuItem items[];
14     private Color colorValues[] =
15         { Color.blue, Color.yellow, Color.red };
16
17     private JPopupMenu popupMenu;
18

```

Fig. 13.11 Using a **PopupMenu** object (part 1 of 3).

```
19 // set up GUI
20 public PopupTest()
21 {
22     super( "Using JPopupMenu" );
23
24     ItemHandler handler = new ItemHandler();
25     String colors[] = { "Blue", "Yellow", "Red" };
26
27     // set up popup menu and its items
28     ButtonGroup colorGroup = new ButtonGroup();
29     popupMenu = new JPopupMenu();
30     items = new JRadioButtonMenuItem[ 3 ];
31
32     // construct each menu item and add to popup menu; also
33     // enable event handling for each menu item
34     for ( int count = 0; count < items.length; count++ ) {
35         items[ count ] =
36             new JRadioButtonMenuItem( colors[ count ] );
37
38         popupMenu.add( items[ count ] );
39         colorGroup.add( items[ count ] );
40
41         items[ count ].addActionListener( handler );
42     }
43
44     getContentPane().setBackground( Color.white );
45
46     // define a MouseListener for the window that displays
47     // a JPopupMenu when the popup trigger event occurs
48     addMouseListener(
49
50         // anonymous inner class to handle mouse events
51         new MouseAdapter() {
52
53             // handle mouse press event
54             public void mousePressed( MouseEvent event )
55             {
56                 checkForTriggerEvent( event );
57             }
58
59             // handle mouse release event
60             public void mouseReleased( MouseEvent event )
61             {
62                 checkForTriggerEvent( event );
63             }
64
65             // determine whether event should trigger popup menu
66             private void checkForTriggerEvent( MouseEvent event )
67             {
68                 if ( event.isPopupTrigger() )
69                     popupMenu.show( event.getComponent(),
70                                     event.getX(), event.getY() );
71             }
72         }
73     );
74 }
```

Fig. 13.11 Using a **PopupMenu** object (part 2 of 3).

```

72
73         } // end anonymous inner clas
74
75     ); // end call to addMouseListener
76
77     setSize( 300, 200 );
78     setVisible( true );
79 }
80
81 // execute application
82 public static void main( String args[] )
83 {
84     PopupTest application = new PopupTest();
85
86     application.setDefaultCloseOperation(
87         JFrame.EXIT_ON_CLOSE );
88 }
89
90 // private inner class to handle menu item events
91 private class ItemHandler implements ActionListener {
92
93     // process menu item selections
94     public void actionPerformed((ActionEvent event) )
95     {
96         // determine which menu item was selected
97         for ( int i = 0; i < items.length; i++ )
98             if ( event.getSource() == items[ i ] ) {
99                 getContentPane().setBackground(
100                     colorValues[ i ] );
101                 repaint();
102                 return;
103             }
104     }
105 } // end private inner class ItemHandler
106
107 } // end class PopupTest
108

```

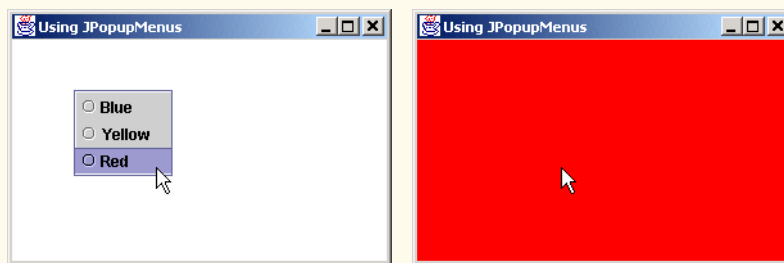


Fig. 13.11 Using a **PopupMenu** object (part 3 of 3).

The constructor for class **PopupTest** (lines 20–79) defines the **JPopupMenu** at line 29. The **for** structure at lines 34–42 creates **JRadioButtonMenuItem**s to add to the **JPopupMenu**, adds them to the **JPopupMenu** (line 38), adds them to **ButtonGroup**

colorGroup (to maintain one selected **JRadioButtonMenuItem** at a time) and registers an **ActionListener** for each menu item.

Lines 48–75 register an instance of an anonymous inner class that extends **MouseAdapter** to handle the mouse events of the application window. Methods **mousePressed** (lines 54–57) and **mouseReleased** (lines 60–63) check for the popup-trigger event. Each method calls **private** utility method **checkForTriggerEvent** (lines 66–71) to determine whether the popup-trigger event occurred. **MouseEvent** method **isPopupTrigger** returns **true** if the popup-trigger event occurred. If so, method **show** of class **JPopupMenu** displays the **JPopupMenu**. The first argument to method **show** specifies the *origin component*, whose position helps determine where the **JPopupMenu** will appear on the screen. The last two arguments are the *x-y* coordinate from the origin component's upper-left corner at which the **JPopupMenu** should appear.



Look-and-Feel Observation 13.24

Displaying a **JPopupMenu** for the popup-trigger event of multiple different GUI components requires registering mouse event handlers to check for the popup-trigger event for each of those GUI components.

When the user selects a menu item from the popup menu, class **ItemHandler**'s (lines 91–106) method **actionPerformed** (lines 94–104) determines which **JRadioButtonMenuItem** the user selected, then sets the background color of the window's content pane.

13.10 Pluggable Look-and-Feel

A program that uses Java's Abstract Windowing Toolkit GUI components (package **java.awt**) takes on the look-and-feel of the platform on which the program executes. A Java program running on a Macintosh looks like other programs running on a Macintosh. A Java program running on Microsoft Windows looks like other programs running on Microsoft Windows. A Java program running on a UNIX platform looks like other programs running on that UNIX platform. This could be desirable, because it allows users of the program on each platform to use the GUI components with which they are already familiar. However, this also introduces interesting portability issues.



Portability Tip 13.1

Programs that use Java's Abstract Windowing Toolkit GUI components (package **java.awt**) take on the look-and-feel of the platform on which they execute.



Portability Tip 13.2

GUI components on each platform have different looks that can require different amounts of space to display. This could change the layout and alignments of GUI components.



Portability Tip 13.3

GUI components on each platform have different default functionality (e.g., some platforms allow a button with the focus to be "pressed" with the space bar, and some do not).

Swing's lightweight GUI components eliminate many of these issues by providing uniform functionality across platforms and by defining a uniform cross-platform look-and-feel (known as the metal look-and-feel). Swing also provides the flexibility to customize

the look-and-feel to appear as a Microsoft Windows-style look-and-feel or a Motif-style (UNIX) look-and-feel.

The program of Fig. 13.12 demonstrates how to change the look-and-feel of a Swing GUI. The program creates several GUI components so you can see the change in the look-and-feel of several GUI components at the same time. The first output window shows the standard metal look-and-feel, the second output window shows the Motif look-and-feel, and the third output window shows the Windows look-and-feel.

```

1  // Fig. 13.12: LookAndFeelDemo.java
2  // Changing the look and feel.
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 LookAndFeelDemo extends JFrame {
12
13     private String strings[] = { "Metal", "Motif", "Windows" };
14     private UIManager.LookAndFeelInfo looks[];
15     private JRadioButton radio[];
16     private ButtonGroup group;
17     private JButton button;
18     private JLabel label;
19     private JComboBox comboBox;
20
21     // set up GUI
22     public LookAndFeelDemo()
23     {
24         super( "Look and Feel Demo" );
25
26         Container container = getContentPane();
27
28         // set up panel for NORTH of BorderLayout
29         JPanel northPanel = new JPanel();
30         northPanel.setLayout( new GridLayout( 3, 1, 0, 5 ) );
31
32         // set up label for NORTH panel
33         label = new JLabel( "This is a Metal look-and-feel",
34                             SwingConstants.CENTER );
35         northPanel.add( label );
36
37         // set up button for NORTH panel
38         button = new JButton( "JButton" );
39         northPanel.add( button );
40
41         // set up combo box for NORTH panel
42         comboBox = new JComboBox( strings );
43         northPanel.add( comboBox );

```

Fig. 13.12 Changing the look-and-feel of a Swing-based GUI (part 1 of 3).

```
44
45 // attach NORTH panel to content pane
46 container.add( northPanel, BorderLayout.NORTH );
47
48 // create array for radio buttons
49 radio = new JRadioButton[ strings.length ];
50
51 // set up panel for SOUTH of BorderLayout
52 JPanel southPanel = new JPanel();
53 southPanel.setLayout(
54     new GridLayout( 1, radio.length ) );
55
56 // set up radio buttons for SOUTH panel
57 group = new ButtonGroup();
58 ItemHandler handler = new ItemHandler();
59
60 for ( int count = 0; count < radio.length; count++ ) {
61     radio[ count ] = new JRadioButton( strings[ count ] );
62     radio[ count ].addItemListener( handler );
63     group.add( radio[ count ] );
64     southPanel.add( radio[ count ] );
65 }
66
67 // attach SOUTH panel to content pane
68 container.add( southPanel, BorderLayout.SOUTH );
69
70 // get installed look-and-feel information
71 looks = UIManager.getInstalledLookAndFeels();
72
73 setSize( 300, 200 );
74 setVisible( true );
75
76 radio[ 0 ].setSelected( true );
77 }
78
79 // use UIManager to change look-and-feel of GUI
80 private void changeTheLookAndFeel( int value )
81 {
82     // change look and feel
83     try {
84         UIManager.setLookAndFeel(
85             looks[ value ].getClassName() );
86         SwingUtilities.updateComponentTreeUI( this );
87     }
88
89     // process problems changing look and feel
90     catch ( Exception exception ) {
91         exception.printStackTrace();
92     }
93 }
94
```

Fig. 13.12 Changing the look-and-feel of a Swing-based GUI (part 2 of 3).

```

95  // execute application
96  public static void main( String args[] )
97  {
98      LookAndFeelDemo application = new LookAndFeelDemo();
99
100     application.setDefaultCloseOperation(
101         JFrame.EXIT_ON_CLOSE );
102 }
103
104 // private inner class to handle radio button events
105 private class ItemHandler implements ItemListener {
106
107     // process user's look-and-feel selection
108     public void itemStateChanged( ItemEvent event )
109     {
110         for ( int count = 0; count < radio.length; count++ )
111
112             if ( radio[ count ].isSelected() ) {
113                 label.setText( "This is a " +
114                     strings[ count ] + " look-and-feel" );
115                 comboBox.setSelectedIndex( count );
116
117                 changeTheLookAndFeel( count );
118             }
119     }
120 } // end private inner class ItemHandler
121
122 } // end class LookAndFeelDemo

```

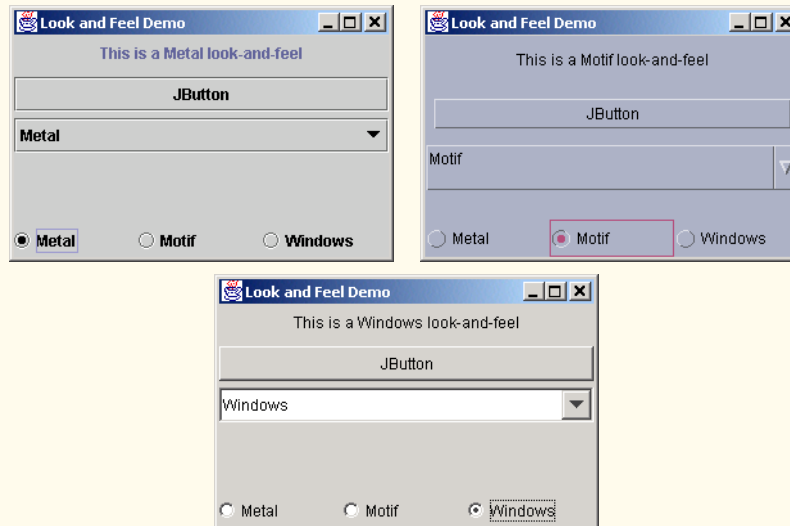


Fig. 13.12 Changing the look-and-feel of a Swing-based GUI (part 3 of 3).

All the GUI components and event handling in this example have been covered before, so we concentrate on the mechanism for changing the look-and-feel in this example.

Class **UIManager** (package **javax.swing**) contains **public static** inner class **LookAndFeelInfo** that is used to maintain information about a look-and-feel. Line 14 declares an array of type **UIManager.LookAndFeelInfo** (notice the syntax used to access the inner class **LookAndFeelInfo**). Line 71 uses **static** method **getInstalledLookAndFeels** of class **UIManager** to get the array of **UIManager.LookAndFeelInfo** objects that describe the installed look-and-feels.



Performance Tip 13.1

Each look-and-feel is represented by a Java class. **UIManager** method **getInstalledLookAndFeels** does not load each class. Rather, it provides access to the names of each look-and-feel, so a choice of look-and-feel can be made (presumably one time at program start-up). This reduces the overhead of loading additional classes that the program will not use.

Utility method **changeTheLookAndFeel** (lines 80–93) is called by the event handler (defined in **private** inner class **ItemHandler** at lines 105–121) for the **JRadioButtons** at the bottom of the user interface. The event handler passes an integer representing the element in array **looks** that should be used to change the look-and-feel. Lines 84–85 use **static** method **setLookAndFeel** of class **UIManager** to change the look-and-feel. Method **getClassName** of class **UIManager.LookAndFeelInfo** determines the name of the look-and-feel class that corresponds to the **UIManager.LookAndFeelInfo**. If the look-and-feel class is not already loaded, it will be loaded as part of the call to **setLookAndFeel**. Line 86 uses **static** method **updateComponentTreeUI** of class **SwingUtilities** (package **javax.swing**) to change the look-and-feel of every component attached to its argument (this instance of class **LookAndFeelDemo**) to the new look-and-feel.

The preceding two statements appear in a special block of code called a **try block**. This code is part of the *exception-handling mechanism* discussed in detail in the next chapter. This code is required in case lines 84–85 attempt to change the look-and-feel to a look-and-feel that does not exist. Lines 90–92 complete the exception-handling mechanism with a **catch handler** that simply processes this problem (if it occurs) by printing an error message at the command line.

13.11 Using JDesktopPane and JInternalFrame

Many of today's applications use a *multiple document interface (MDI)* [i.e., a main window (often called the *parent window*) containing other windows (often called *child windows*)] to manage several open *documents* that are being processed in parallel. For example, many e-mail programs allow you to have several e-mail windows open at the same time so you can compose and/or read multiple e-mail messages. Similarly, many word processors allow the user to open multiple documents in separate windows so the user can switch between the documents without having to close the current document to open another document. The program of Fig. 13.13 demonstrates Swing's **JDesktopPane** and **JInternalFrame** classes, which provide support for creating multiple document interfaces. The child windows simply display an image of the cover of this book.

Lines 20–27 define a **JMenuBar**, a **JMenu** and a **JMenuItem**, add the **JMenuItem** to the **JMenu**, add the **JMenu** to the **JMenuBar** and set the **JMenuBar** for the application window. When the user selects the **JMenuItem newFrame**, the program creates and displays a new **JInternalFrame**.

```
1 // Fig. 13.13: DesktopTest.java
2 // Demonstrating JDesktopPane.
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 DesktopTest extends JFrame {
12     private JDesktopPane theDesktop;
13
14     // set up GUI
15     public DesktopTest()
16     {
17         super( "Using a JDesktopPane" );
18
19         // create menu bar, menu and menu item
20         JMenuBar bar = new JMenuBar();
21         JMenu addMenu = new JMenu( "Add" );
22         JMenuItem newFrame = new JMenuItem( "Internal Frame" );
23
24         addMenu.add( newFrame );
25         bar.add( addMenu );
26
27         setJMenuBar( bar );
28
29         // set up desktop
30         theDesktop = new JDesktopPane();
31         getContentPane().add( theDesktop );
32
33         // set up listener for newFrame menu item
34         newFrame.addActionListener(
35
36             // anonymous inner class to handle menu item event
37             new ActionListener() {
38
39                 // display new internal window
40                 public void actionPerformed((ActionEvent event) {
41
42                     // create internal frame
43                     JFrame frame = new JFrame(
44                         "Internal Frame", true, true, true, true );
45
46                     // attach panel to internal frame content pane
47                     Container container = frame.getContentPane();
48                     MyJPanel panel = new MyJPanel();
49                     container.add( panel, BorderLayout.CENTER );
50
51                     // set size internal frame to size of its contents
52                     frame.pack();
53                 }
41
```

Fig. 13.13 Creating a multiple document interface (part 1 of 3).

```

54         // attach internal frame to desktop and show it
55         theDesktop.add( frame );
56         frame.setVisible( true );
57     }
58
59     } // end anonymous inner class
60
61     ); // end call to addActionListener
62
63     setSize( 600, 440 );
64     setVisible( true );
65
66 } // end constructor
67
68 // execute application
69 public static void main( String args[] )
70 {
71     DesktopTest application = new DesktopTest();
72
73     application.setDefaultCloseOperation(
74         JFrame.EXIT_ON_CLOSE );
75 }
76
77 } // end class DesktopTest
78
79 // class to display an ImageIcon on a panel
80 class MyJPanel extends JPanel {
81     private ImageIcon imageIcon;
82
83     // load image
84     public MyJPanel()
85     {
86         imageIcon = new ImageIcon( "jhtp4.png" );
87     }
88
89     // display imageIcon on panel
90     public void paintComponent( Graphics g )
91     {
92         // call superclass paintComponent method
93         super.paintComponent( g );
94
95         // display icon
96         imageIcon.paintIcon( this, g, 0, 0 );
97     }
98
99     // return image dimensions
100    public Dimension getPreferredSize()
101    {
102        return new Dimension( imageIcon.getIconWidth(),
103                               imageIcon.getIconHeight() );
104    }
105
106 } // end class MyJPanel

```

Fig. 13.13 Creating a multiple document interface (part 2 of 3).

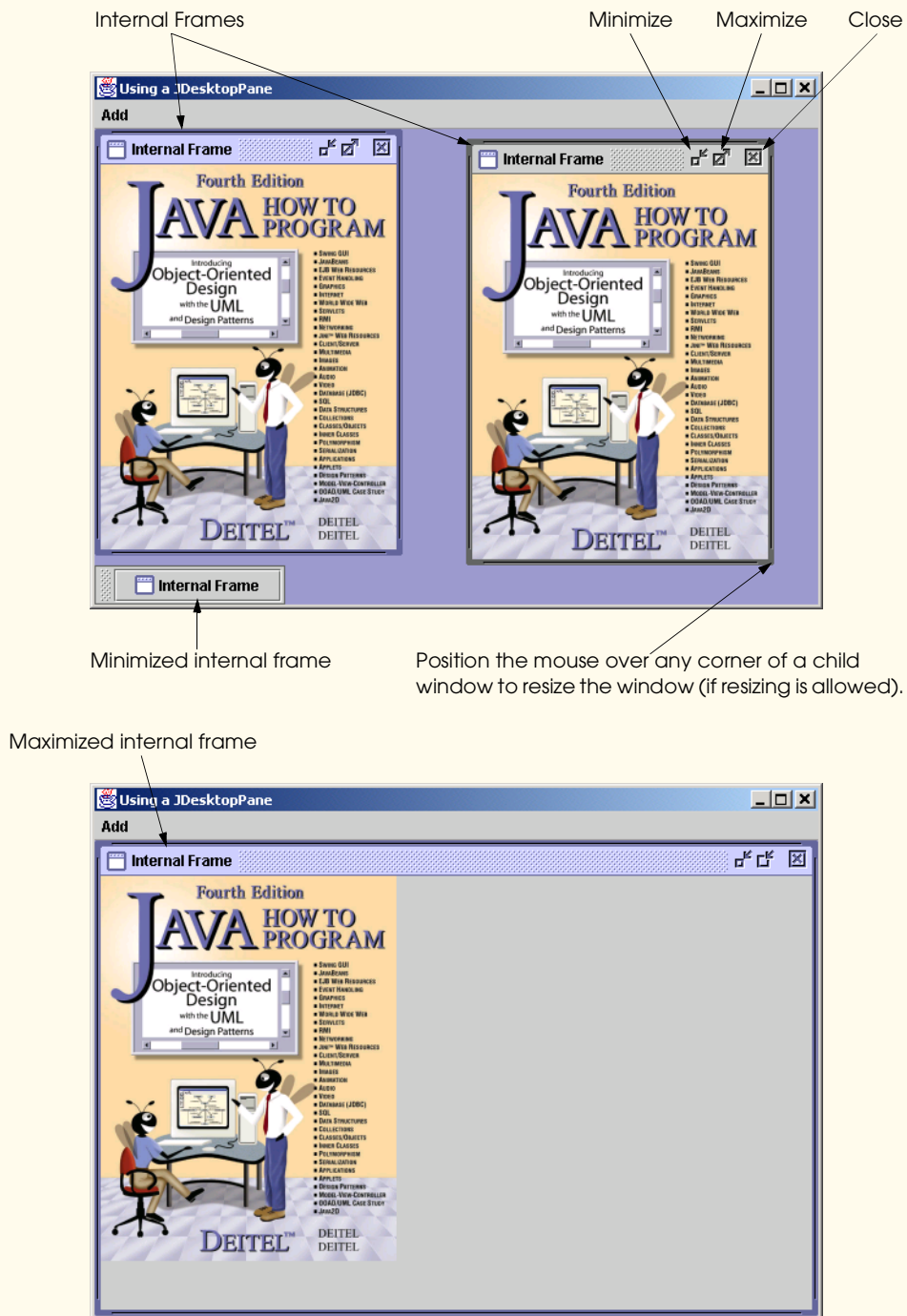


Fig. 13.13 Creating a multiple document interface (part 3 of 3).

Line 30 creates **JDesktopPane** (package **javax.swing**) reference **theDesktop** and assigns it a new **JDesktopPane** object. The **JDesktopPane** object manages the **JInternalFrame** child windows displayed in the **JDesktopPane**. Line 31 adds the **JDesktopPane** to the application window's content pane.

Lines 34–61 register an instance of an anonymous inner class that implements **ActionListener** to handle the event when the user selects the **newFrame** menu item. When the event occurs, method **actionPerformed** (lines 40–57) creates a **JInternalFrame** object with lines 43–44. The **JInternalFrame** constructor used here requires five arguments—a **String** for the title bar of the internal window, a **boolean** indicating whether the internal frame should be resizable by the user, a **boolean** indicating whether the internal frame should be closable by the user, a **boolean** indicating whether the internal frame should be maximizable by the user and a **boolean** indicating whether the internal frame should be minimizable by the user. For each of the boolean arguments, a **true** value indicates that the operation should be allowed.

As with **JFrames** and **JApplets**, a **JInternalFrame** has a content pane to which GUI components can be attached. Line 47 gets a reference to the **JInternalFrame**'s content pane. Line 48 creates an instance of our class **MyJPanel** (defined at lines 80–106) that is added to the **JInternalFrame**'s content pane at line 49.

Line 52 uses **JInternalFrame** method **pack** to set the size of the child window. Method **pack** uses the preferred sizes of the components on the content pane to determine the window's size. Class **MyJPanel** defines method **getPreferredSize** to specify the panel's preferred size. Line 55 adds the **JInternalFrame** to the **JDesktopPane**, and line 56 displays the **JInternalFrame**.

Classes **JInternalFrame** and **JDesktopPane** provide many methods for managing child windows. See the on-line API documentation for a complete list of these methods.

13.12 Layout Managers

In the preceding chapter, we introduced three layout managers—**FlowLayout**, **BorderLayout** and **GridLayout**. This section presents three additional layout managers (summarized in Figure 13.14). We discuss these layout managers in the sections that follow.

Layout Manager	Description
BoxLayout	A layout manager that allows GUI components to be arranged left-to-right or top-to-bottom in a container. Class Box defines a container with BoxLayout as its default layout manager and provides static methods to create a Box with a horizontal or vertical BoxLayout .
CardLayout	A layout manager that stacks components like a deck of cards. If a component in the deck is a container, it can use any layout manager. Only the component at the "top" of the deck is visible.
GridBagLayout	A layout manager similar to GridLayout . Unlike GridLayout , each component size can vary and components can be added in any order.

Fig. 13.14 Additional layout managers.

13.13 BorderLayout Layout Manager

The **BoxLayout** layout manager arranges GUI components horizontally along the *x*-axis or vertically along the *y*-axis of a container. The program of Fig. 13.15 demonstrates **BoxLayout** and the container class **Box** that uses **BoxLayout** as its default layout manager.

In the constructor for class **BoxLayoutDemo**, lines 19–20 obtain a reference to the content pane and set its layout to a **BorderLayout** with 30 pixels of horizontal and 30 pixels of vertical gap space between components. The space is to help isolate each of the containers with **BoxLayout** in this example.

Lines 23–28 define an array of **Box** container references called **boxes** and initialize each element of the array with **Box** objects. Elements 0 and 2 of the array are initialized with **static** method **createHorizontalBox** of class **Box**, which returns a **Box** container with a horizontal **BoxLayout** (GUI components are arranged left-to-right). Elements 1 and 3 of the array are initialized with **static** method **createVerticalBox** of class **Box**, which returns a **Box** container with a vertical **BoxLayout** (GUI components are arranged top-to-bottom).

```

1  // Fig. 13.15: BoxLayoutDemo.java
2  // Demonstrating BoxLayout.
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 BoxLayoutDemo extends JFrame {
12
13     // set up GUI
14     public BoxLayoutDemo()
15     {
16         super( "Demonstrating BoxLayout" );
17         final int SIZE = 3;
18
19         Container container = getContentPane();
20         container.setLayout( new BorderLayout( 30, 30 ) );
21
22         // create Box containers with BoxLayout
23         Box boxes[] = new Box[ 4 ];
24
25         boxes[ 0 ] = Box.createHorizontalBox();
26         boxes[ 1 ] = Box.createVerticalBox();
27         boxes[ 2 ] = Box.createHorizontalBox();
28         boxes[ 3 ] = Box.createVerticalBox();
29
30         // add buttons to boxes[ 0 ]
31         for ( int count = 0; count < SIZE; count++ )
32             boxes[ 0 ].add( new JButton( "boxes[0]: " + count ) );

```

Fig. 13.15 Demonstrating the **BoxLayout** layout manager (part 1 of 3).

```

33
34 // create strut and add buttons to boxes[ 1 ]
35 for ( int count = 0; count < SIZE; count++ ) {
36     boxes[ 1 ].add( Box.createVerticalStrut( 25 ) );
37     boxes[ 1 ].add( new JButton( "boxes[1]: " + count ) );
38 }
39
40 // create horizontal glue and add buttons to boxes[ 2 ]
41 for ( int count = 0; count < SIZE; count++ ) {
42     boxes[ 2 ].add( Box.createHorizontalGlue() );
43     boxes[ 2 ].add( new JButton( "boxes[2]: " + count ) );
44 }
45
46 // create rigid area and add buttons to boxes[ 3 ]
47 for ( int count = 0; count < SIZE; count++ ) {
48     boxes[ 3 ].add(
49         Box.createRigidArea( new Dimension( 12, 8 ) ) );
50     boxes[ 3 ].add( new JButton( "boxes[3]: " + count ) );
51 }
52
53 // create vertical glue and add buttons to panel
54 JPanel panel = new JPanel();
55 panel.setLayout(
56     new BoxLayout( panel, BoxLayout.Y_AXIS ) );
57
58 for ( int count = 0; count < SIZE; count++ ) {
59     panel.add( Box.createGlue() );
60     panel.add( new JButton( "panel: " + count ) );
61 }
62
63 // place panels on frame
64 container.add( boxes[ 0 ], BorderLayout.NORTH );
65 container.add( boxes[ 1 ], BorderLayout.EAST );
66 container.add( boxes[ 2 ], BorderLayout.SOUTH );
67 container.add( boxes[ 3 ], BorderLayout.WEST );
68 container.add( panel, BorderLayout.CENTER );
69
70 setSize( 350, 300 );
71 setVisible( true );
72
73 } // end constructor
74
75 // execute application
76 public static void main( String args[] )
77 {
78     BoxLayoutDemo application = new BoxLayoutDemo();
79
80     application.setDefaultCloseOperation(
81         JFrame.EXIT_ON_CLOSE );
82 }
83
84 } // end class BoxLayoutDemo

```

Fig. 13.15 Demonstrating the **BoxLayout** layout manager (part 2 of 3).

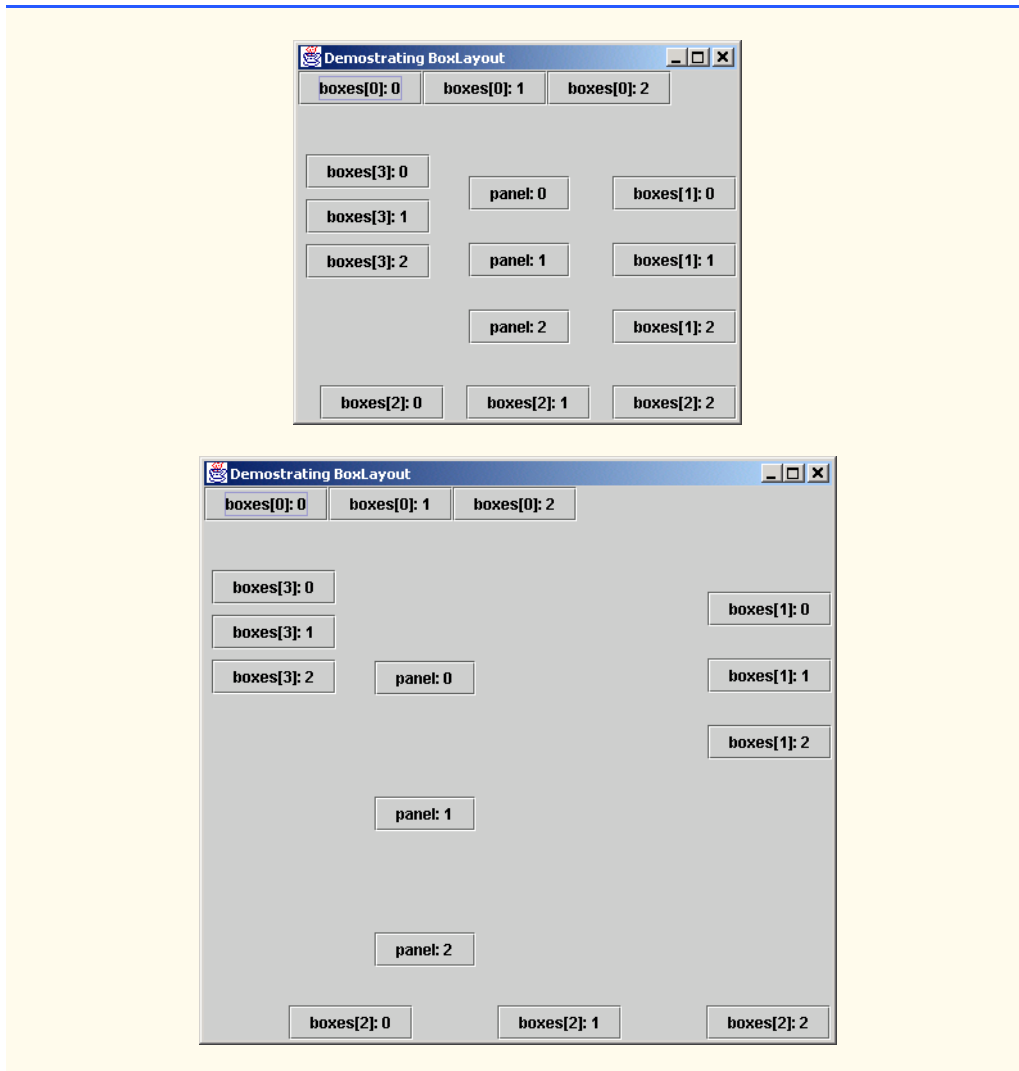


Fig. 13.15 Demonstrating the **BoxLayout** layout manager (part 3 of 3).

The **for** structure at lines 31–32 adds three **JButtons** to **boxes [0]** (a horizontal **Box**). The **for** structure at lines 35–38 adds three **JButtons** to **boxes [1]** (a vertical **Box**). Before adding each button, line 36 adds a *vertical strut* to the container with **static** method **createVerticalStrut** of class **Box**. A vertical strut is an invisible GUI component that has a fixed pixel height and is used to guarantee a fixed amount of space between GUI components. The argument to method **createVerticalStrut** determines the height of the strut in pixels. Class **Box** also defines method **createHorizontalStrut** for horizontal **BoxLayouts**.

The **for** structure at lines 41–44 adds three **JButtons** to **boxes [2]** (a horizontal **Box**). Before adding each button, line 42 adds *horizontal glue* to the container with **static** method **createHorizontalGlue** of class **Box**. Horizontal glue is an invis-

ible GUI component that can be used between fixed-size GUI components to occupy additional space. Normally, extra space appears to the right of the last horizontal GUI component or below the last vertical GUI component in a **BoxLayout**. Glue allows the extra space to be placed between GUI components. Class **Box** also defines method **createVerticalGlue** for vertical **BoxLayouts**.

The **for** structure at lines 47–51 adds three **JButtons** to **boxes[3]** (a vertical **Box**). Before adding each button, lines 48–49 add a *rigid area* to the container with **static** method **createRigidArea** of class **Box**. A rigid area is an invisible GUI component that always has a fixed pixel width and height. The argument to method **createRigidArea** is a **Dimension** object that specifies the width and height of the rigid area.

Lines 54–56 create a **JPanel** object and set its layout in the conventional manner, using **Container** method **setLayout**. The **BoxLayout** constructor receives a reference to the container for which it controls the layout and a constant indicating whether the layout is horizontal (**BoxLayout.X_AXIS**) or vertical (**BoxLayout.Y_AXIS**).

The **for** structure at lines 58–61 adds three **JButtons** to **panel**. Before adding each button, line 59 adds a glue component to the container with **static** method **createGlue** of class **Box**. This component expands or contracts based on the size of the **Box**.

The **Box** containers and the **JPanel** are attached to the content pane's **BorderLayout** at lines 64–68. Try executing the application. When the window appears, resize the window to see how the glue components, strut components and rigid area affect the layout in each container.

13.14 CardLayout Layout Manager

The **CardLayout layout manager** arranges components into a “deck” of cards where only the top card is visible. Any card in the deck can be placed at the top of the deck at any time by using methods of class **CardLayout**. Each card is usually a container, such as a panel, and each card can use any layout manager. Class **CardLayout** inherits from **Object** and implements the **LayoutManager2** interface.

The program of Fig. 13.16 creates five panels. **JPanel deck** uses the **CardLayout** layout manager to control the card that is displayed. **JPanel card1**, **card2** and **card3** are the individual cards in **deck**. **JPanel buttons** contains four buttons (with labels **First card**, **Next card**, **Previous card** and **Last card**) that enable the user to manipulate the deck. When the user clicks the **First card** button, the first card in **deck** (i.e., **card1**) is displayed. When the user clicks the **Last card** button, the last card (i.e., **card3**) in **deck** is displayed. Each time the user clicks the **Previous card** button, the previous card in **deck** is displayed. Each time the user clicks the **Next card** button, the next card in **deck** is displayed. Clicking the **Previous card** button or the **Next card** button repeatedly allows the user to cycle through the **deck** of cards. Application class **CardDeck** implements **ActionListener**, so the action events generated by the **JButtons** on **JPanel buttons** are handled by the application in its **actionPerformed** method.

Class **CardDeck** declares a reference of type **CardLayout** called **cardManager** (line 13). This reference is used to invoke **CardLayout** methods that manipulate the cards in the deck.

```
1 // Fig. 13.16: CardDeck.java
2 // Demonstrating CardLayout.
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 CardDeck extends JFrame implements ActionListener {
12
13     private CardLayout cardManager;
14     private JPanel deck;
15     private JButton controls[];
16     private String names[] = { "First card", "Next card",
17         "Previous card", "Last card" };
18
19     // set up GUI
20     public CardDeck()
21     {
22         super( "CardLayout " );
23
24         Container container = getContentPane();
25
26         // create the JPanel with CardLayout
27         deck = new JPanel();
28         cardManager = new CardLayout();
29         deck.setLayout( cardManager );
30
31         // set up card1 and add it to JPanel deck
32         JLabel label1 =
33             new JLabel( "card one", SwingConstants.CENTER );
34         JPanel card1 = new JPanel();
35         card1.add( label1 );
36         deck.add( card1, label1.getText() ); // add card to deck
37
38         // set up card2 and add it to JPanel deck
39         JLabel label2 =
40             new JLabel( "card two", SwingConstants.CENTER );
41         JPanel card2 = new JPanel();
42         card2.setBackground( Color.yellow );
43         card2.add( label2 );
44         deck.add( card2, label2.getText() ); // add card to deck
45
46         // set up card3 and add it to JPanel deck
47         JLabel label3 = new JLabel( "card three" );
48         JPanel card3 = new JPanel();
49         card3.setLayout( new BorderLayout() );
50         card3.add( new JButton( "North" ), BorderLayout.NORTH );
51         card3.add( new JButton( "West" ), BorderLayout.WEST );
52         card3.add( new JButton( "East" ), BorderLayout.EAST );
53         card3.add( new JButton( "South" ), BorderLayout.SOUTH );
```

Fig. 13.16 Demonstrating the **CardLayout** layout manager (part 1 of 3).

```

54     card3.add( label3, BorderLayout.CENTER );
55     deck.add( card3, label3.getText() ); // add card to deck
56
57     // create and layout buttons that will control deck
58     JPanel buttons = new JPanel();
59     buttons.setLayout( new GridLayout( 2, 2 ) );
60     controls = new JButton[ names.length ];
61
62     for ( int count = 0; count < controls.length; count++ ) {
63         controls[ count ] = new JButton( names[ count ] );
64         controls[ count ].addActionListener( this );
65         buttons.add( controls[ count ] );
66     }
67
68     // add JPanel deck and JPanel buttons to the applet
69     container.add( buttons, BorderLayout.WEST );
70     container.add( deck, BorderLayout.EAST );
71
72     setSize( 450, 200 );
73     setVisible( true );
74
75 } // end constructor
76
77 // handle button events by switching cards
78 public void actionPerformed((ActionEvent event) )
79 {
80     // show first card
81     if ( event.getSource() == controls[ 0 ] )
82         cardManager.first( deck );
83
84     // show next card
85     else if ( event.getSource() == controls[ 1 ] )
86         cardManager.next( deck );
87
88     // show previous card
89     else if ( event.getSource() == controls[ 2 ] )
90         cardManager.previous( deck );
91
92     // show last card
93     else if ( event.getSource() == controls[ 3 ] )
94         cardManager.last( deck );
95 }
96
97 // execute application
98 public static void main( String args[] )
99 {
100     CardDeck cardDeckDemo = new CardDeck();
101
102     cardDeckDemo.setDefaultCloseOperation(
103         JFrame.EXIT_ON_CLOSE );
104 }
105
106 } // end class CardDeck

```

Fig. 13.16 Demonstrating the **CardLayout** layout manager (part 2 of 3).

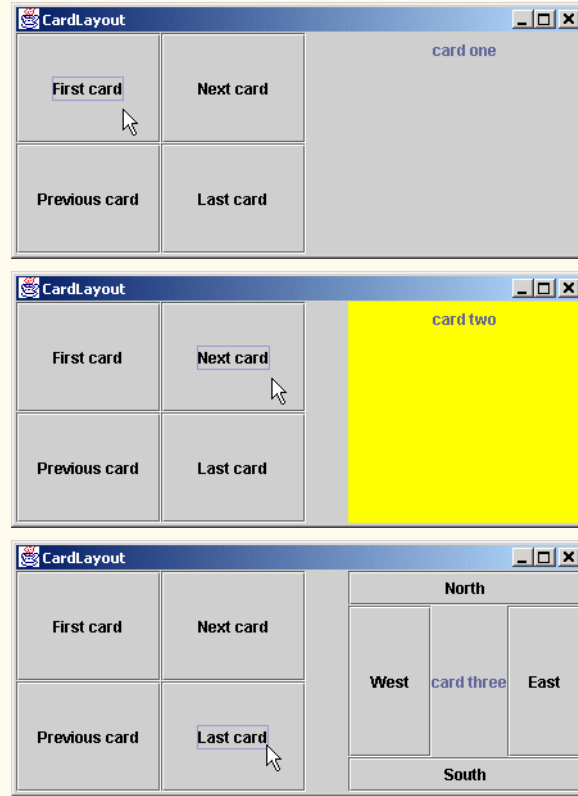


Fig. 13.16 Demonstrating the **CardLayout** layout manager (part 3 of 3).

The constructor method (lines 20–75) builds the GUI. Lines 27–29 create **JPanel** **deck**, create **CardLayout** object **cardManager** and set the layout manager for **deck** to **cardManager**. Next, the constructor creates the **JPanel**s **card1**, **card2**, and **card3** and their GUI components. As we set up each card, we add the card to **deck**, using **Container** method **add** with two arguments—a **Component** and a **String**. The **Component** is the **JPanel** object that represents the card. The **String** argument identifies the card. For example, line 36 adds **JPanel** **card1** to **deck** and uses **JLabel** **label1**'s label as the **String** identifier for the card. **JPanel**s **card2** and **card3** are added to **deck** at lines 44–55. Next, the constructor creates the **JPanel** **buttons** and its **JButton** objects (lines 58–66). Line 64 registers the **ActionListener** for each **JButton**. Finally, **buttons** and **deck** are added to the content pane's **WEST** and **EAST** regions, respectively.

Method **actionPerformed** (lines 78–95) determines which **JButton** generated the event by using **EventObject** method **getSource**. **CardLayout** methods **first**, **previous**, **next** and **last** are used to display the particular card corresponding to which **JButton** the user pressed. Method **first** displays the first card added to the deck. Method **previous** displays the previous card in the deck. Method **next** displays the next card in the deck. Method **last** displays the last card in the deck. Note that **deck** is passed to each of these methods.

13.15 GridBagLayout Layout Manager

The most complex and most powerful of the predefined layout managers is **GridBagLayout**. This layout is similar to **GridLayout** because **GridBagLayout** also arranges components in a grid. However, **GridBagLayout** is more flexible. The components can vary in size (i.e., they can occupy multiple rows and columns) and can be added in any order.

The first step in using **GridBagLayout** is determining the appearance of the GUI. This step does not involve any programming; all that is needed is a piece of paper. First, draw the GUI. Next draw a grid over the GUI dividing the components into rows and columns. The initial row and column numbers should be 0 so the **GridBagLayout** layout manager can properly place the components in the grid. The row and column numbers will be used to place each component in an exact position in the grid. Figure 13.17 demonstrates drawing the lines for the rows and columns over a GUI.

To use **GridBagLayout**, a **GridBagConstraints** object must be constructed. This object specifies how a component is placed in a **GridBagLayout**. Several **GridBagConstraints** instance variables are summarized in Fig. 13.18.

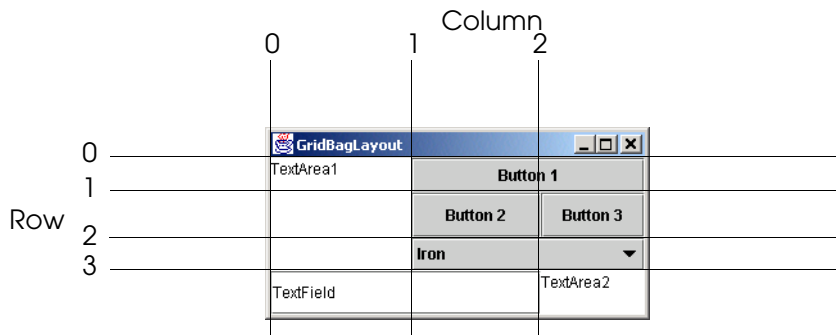


Fig. 13.17 Designing a GUI that will use **GridBagLayout**.

GridBagConstraints Instance Variable	Description
gridx	The column in which the component will be placed.
gridy	The row in which the component will be placed.
gridwidth	The number of columns the component occupies.
gridheight	The number of rows the component occupies.
weightx	The portion of extra space to allocate horizontally. The components can become wider when extra space is available.
weighty	The portion of extra space to allocate vertically. The components can become taller when extra space is available.

Fig. 13.18 **GridBagConstraints** instance variables.

Variables **gridx** and **gridy** specify the row and column where the upper-left corner of the component is placed in the grid. Variable **gridx** corresponds to the column and the variable **gridy** corresponds to the row. In Fig. 13.17, the **JComboBox** (displaying “Iron”) has a **gridx** value of 1 and a **gridy** value of 2.

Variable **gridwidth** specifies the number of columns a component occupies. In Fig. 13.17, the **JComboBox** button occupies two columns. Variable **gridheight** specifies the number of rows a component occupies. In Fig. 13.17, the **JTextArea** on the left side of the window occupies three rows.

Variable **weightx** specifies how to distribute extra horizontal space to components in a **GridBagLayout** when the container is resized. A zero value indicates that the component does not grow horizontally on its own. However, if the component spans a column containing a component with nonzero **weightx** value, the component with zero **weightx** value will grow horizontally in the same proportion as the other component(s) in the same column. This is because each component must be maintained in the same row and column in which it was originally placed.

Variable **weighty** specifies how to distribute extra vertical space to components in a **GridBagLayout** when the container is resized. A zero value indicates that the component does not grow vertically on its own. However, if the component spans a row containing a component with nonzero **weighty** value, the component with zero **weighty** value grows vertically in the same proportion as the other component(s) in the same row.

In Fig. 13.17, the effects of **weighty** and **weightx** cannot easily be seen until the container is resized and additional space becomes available. Components with larger weight values occupy more of the additional space than components with smaller weight values. The exercises explore the effects of varying **weightx** and **weighty**.

Components should be given nonzero positive weight values—otherwise the components will “huddle” together in the middle of the container. Figure 13.19 shows the GUI of Fig. 13.17—where all weights have been set to zero.



Common Programming Error 13.6

Using a negative value for either **weightx** or **weighty** is a logic error.

GridBagConstraints instance variable **fill** specifies how much of the component’s area (the number of rows and columns the component occupies in the grid) is occupied. The variable **fill** is assigned one of the following **GridBagConstraints** constants: **NONE**, **VERTICAL**, **HORIZONTAL** or **BOTH**. The default value is **NONE**, which indicates that the component will not grow in either direction. **VERTICAL** indicates that the component will grow vertically. **HORIZONTAL** indicates that the component will grow horizontally. **BOTH** indicates that the component will grow in both directions.

GridBagConstraints instance variable **anchor** specifies the location of the component in the area when the component does not fill the entire area. The variable **anchor** is assigned one of the following **GridBagConstraints** constants: **NORTH**, **NORTHEAST**, **EAST**, **SOUTHEAST**, **SOUTH**, **SOUTHWEST**, **WEST**, **NORTHWEST** or **CENTER**. The default value is **CENTER**.

The program of Fig. 13.20 uses the **GridBagLayout** layout manager to arrange the components in the GUI of Fig. 13.17. The program does nothing other than demonstrate how to use **GridBagLayout**.

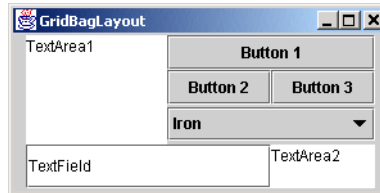


Fig. 13.19 **GridBagLayout** with the weights set to zero.

```

1  // Fig. 13.20: GridBagDemo.java
2  // Demonstrating GridBagLayout.
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 GridBagDemo extends JFrame {
12     private Container container;
13     private GridBagLayout layout;
14     private GridBagConstraints constraints;
15
16     // set up GUI
17     public GridBagDemo()
18     {
19         super( "GridBagLayout" );
20
21         container = getContentPane();
22         layout = new GridBagLayout();
23         container.setLayout( layout );
24
25         // instantiate gridbag constraints
26         constraints = new GridBagConstraints();
27
28         // create GUI components
29         JTextArea textArea1 = new JTextArea( "TextArea1", 5, 10 );
30         JTextArea textArea2 = new JTextArea( "TextArea2", 2, 2 );
31
32         String names[] = { "Iron", "Steel", "Brass" };
33         JComboBox comboBox = new JComboBox( names );
34
35         JTextField textField = new JTextField( "TextField" );
36         JButton button1 = new JButton( "Button 1" );
37         JButton button2 = new JButton( "Button 2" );
38         JButton button3 = new JButton( "Button 3" );

```

Fig. 13.20 Demonstrating the **GridBagLayout** layout manager (part 1 of 3).

```

39
40     // textArea1
41     // weightx and weighty are both 0: the default
42     // anchor for all components is CENTER: the default
43     constraints.fill = GridBagConstraints.BOTH;
44     addComponent( textArea1, 0, 0, 1, 3 );
45
46     // button1
47     // weightx and weighty are both 0: the default
48     constraints.fill = GridBagConstraints.HORIZONTAL;
49     addComponent( button1, 0, 1, 2, 1 );
50
51     // comboBox
52     // weightx and weighty are both 0: the default
53     // fill is HORIZONTAL
54     addComponent( comboBox, 2, 1, 2, 1 );
55
56     // button2
57     constraints.weightx = 1000; // can grow wider
58     constraints.weighty = 1;   // can grow taller
59     constraints.fill = GridBagConstraints.BOTH;
60     addComponent( button2, 1, 1, 1, 1 );
61
62     // button3
63     // fill is BOTH
64     constraints.weightx = 0;
65     constraints.weighty = 0;
66     addComponent( button3, 1, 2, 1, 1 );
67
68     // textField
69     // weightx and weighty are both 0, fill is BOTH
70     addComponent( textField, 3, 0, 2, 1 );
71
72     // textArea2
73     // weightx and weighty are both 0, fill is BOTH
74     addComponent( textArea2, 3, 2, 1, 1 );
75
76     setSize( 300, 150 );
77     setVisible( true );
78 }
79
80 // method to set constraints on
81 private void addComponent( Component component,
82     int row, int column, int width, int height )
83 {
84     // set gridx and gridy
85     constraints.gridx = column;
86     constraints.gridy = row;
87
88     // set gridwidth and gridheight
89     constraints.gridwidth = width;
90     constraints.gridheight = height;
91

```

Fig. 13.20 Demonstrating the **GridBagLayout** layout manager (part 2 of 3).

```

92      // set constraints and add component
93      layout.setConstraints( component, constraints );
94      container.add( component );
95  }
96
97  // execute application
98  public static void main( String args[] )
99  {
100      GridBagDemo application = new GridBagDemo();
101
102      application.setDefaultCloseOperation(
103          JFrame.EXIT_ON_CLOSE );
104  }
105
106 } // end class GridBagDemo

```

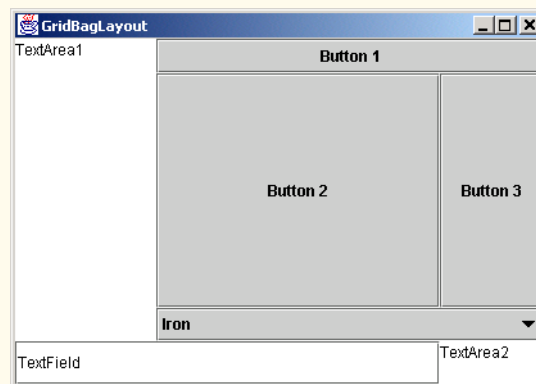
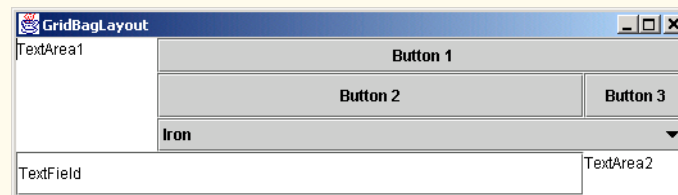
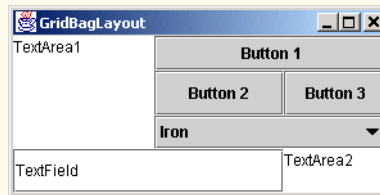


Fig. 13.20 Demonstrating the **GridBagLayout** layout manager (part 3 of 3).

The GUI consists of three **JButtons**, two **JTextAreas**, a **JComboBox** and a **JTextField**. The layout manager for the content pane is **GridBagLayout**. Lines 22–23 instantiate the **GridBagLayout** object and set the layout manager for the content pane to **layout**. The **GridBagConstraints** object used to determine the location and size

of each component in the grid is instantiated with line 26. Lines 23 through 30 instantiate each of the GUI components that will be added to the content pane.

JTextArea textArea1 is the first component added to the **GridBagLayout** (line 44). The values for **weightx** and **weighty** values are not specified in **gbConstraints**, so each has the value zero by default. Thus, the **JTextArea** will not resize itself even if space is available. However, the **JTextArea** spans multiple rows, so the vertical size is subject to the **weighty** values of **JButtons button2** and **button3**. When either **button2** or **button3** is resized vertically based on its **weighty** value, the **JTextArea** is also resized.

Line 43 sets variable **fill** in **constraints** to **GridBagConstraints.BOTH**, causing the **JTextArea** to always fill its entire allocated area in the grid. An **anchor** value is not specified in **constraints**, so the default **CENTER** is used. We do not use variable **anchor** in this program, so all components will use the default. Line 36 calls our utility method **addComponent** method (defined at lines 81–95). The **JTextArea** object, the row, the column, the number of columns to span and the number of rows to span are passed as arguments.

Method **addComponent**'s parameters are a **Component** reference **component** and integers **row**, **column**, **width** and **height**. Lines 85–86 set the **GridBagConstraints** variables **gridx** and **gridy**. The **gridx** variable is assigned the column in which the **Component** will be placed, and the **gridy** value is assigned the row in which the **Component** will be placed. Lines 89–90 set the **GridBagConstraints** variables **gridwidth** and **gridheight**. The **gridwidth** variable specifies the number of columns the **Component** will span in the grid and the **gridheight** variable specifies the number of rows the **Component** will span in the grid. Line 93 sets the **GridBagConstraints** for a component in the **GridBagLayout**. Method **setConstraints** of class **GridBagLayout** takes a **Component** argument and a **GridBagConstraints** argument. Method **add** (line 94) is used to add the component to the content pane.

JButton object **button1** is the next component added (lines 48–49). The values of **weightx** and **weighty** are still zero. The **fill** variable is set to **HORIZONTAL**—the component will always fill its area in the horizontal direction. The vertical direction is not filled. The **weighty** value is zero, so the button will become taller only if another component in the same row has a nonzero **weighty** value. **JButton b1** is located at row 0, column 1. One row and two columns are occupied.

JComboBox comboBox is the next component added (line 54). The **weightx** and **weighty** values are zero, and the **fill** variable is set to **HORIZONTAL**. The **JComboBox** button will grow only in the horizontal direction. Note that the **weightx**, **weighty** and **fill** variables remain set in **gbConstraints** until they are changed. The **JComboBox** button is placed at row 2, column 1. One row and two columns are occupied.

JButton object **button2** is the next component added (lines 57–60). **JButton button2** is given a **weightx** value of **1000** and a **weighty** value of **1**. The area occupied by the button is capable of growing in the vertical and horizontal directions. The **fill** variable is set to **BOTH**, which specifies that the button will always fill the entire area. When the window is resized, **b2** will grow. The button is placed at row 1, column 1. One row and one column are occupied.

JButton button3 is added next (lines 64–66). Both the **weightx** value and **weighty** value are set to zero, and the value of **fill** is **BOTH**. **JButton button3** will

grow if the window is resized; it is affected by the weight values of **button2**. Note that the **weightx** value for **button2** is much larger than **button3**. When resizing occurs, **button2** will occupy a larger percentage of the new space. The button is placed at row 1, column 2. One row and one column are occupied.

Both the **JTextField** (line 70) and **JTextArea textArea2** (line 74) have a **weightx** value 0 and a **weighty** value 0. The value of **fill** is **BOTH**. The **JTextField** is placed at row 3, column 0, and the **JTextArea** is placed at row 3, column 2. The **JTextField** occupies one row and two columns. The **JTextArea** occupies one row and one column.

When you execute this application, try resizing the window to see how the constraints for each GUI component affect its position and size in the window.

13.16 GridBagConstraints Constants **RELATIVE** and **REMAINDER**

A variation of **GridBagLayout** does not use **gridx** and **gridy**. Rather, **GridbagConstraints** constants **RELATIVE** and **REMAINDER** are used in their place. **RELATIVE** specifies that the next-to-last component in a particular row should be placed to the right of the previous component in that row. **REMAINDER** specifies that a component is the last component in a row. Any component that is not the second-to-last or last component on a row must specify values for **GridbagConstraints** variables **gridwidth** and **gridheight**. Class **GridBagDemo2** in Fig. 13.21 arranges components in **GridBagLayout**, using these constants.

```

1  // Fig. 13.21: GridBagDemo2.java
2  // Demonstrating GridBagLayout constants.
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 GridBagDemo2 extends JFrame {
12     private GridBagLayout layout;
13     private GridBagConstraints constraints;
14     private Container container;
15
16     // set up GUI
17     public GridBagDemo2()
18     {
19         super( "GridBagLayout" );
20
21         container = getContentPane();
22         layout = new GridBagLayout();
23         container.setLayout( layout );
24     }

```

Fig. 13.21 Demonstrating the **GridBagConstraints** constants **RELATIVE** and **REMAINDER** (part 1 of 3).

```

25      // instantiate gridbag constraints
26      constraints = new GridBagConstraints();
27
28      // create GUI components
29      String metals[] = { "Copper", "Aluminum", "Silver" };
30      JComboBox comboBox = new JComboBox( metals );
31
32      JTextField textField = new JTextField( "TextField" );
33
34      String fonts[] = { "Serif", "Monospaced" };
35      JList list = new JList( fonts );
36
37      String names[] =
38      { "zero", "one", "two", "three", "four" };
39      JButton buttons[] = new JButton[ names.length ];
40
41      for ( int count = 0; count < buttons.length; count++ )
42          buttons[ count ] = new JButton( names[ count ] );
43
44      // define GUI component constraints
45      // textField
46      constraints.weightx = 1;
47      constraints.weighty = 1;
48      constraints.fill = GridBagConstraints.BOTH;
49      constraints.gridwidth = GridBagConstraints.REMAINDER;
50      addComponent( textField );
51
52      // buttons[0] -- weightx and weighty are 1: fill is BOTH
53      constraints.gridwidth = 1;
54      addComponent( buttons[ 0 ] );
55
56      // buttons[1] -- weightx and weighty are 1: fill is BOTH
57      constraints.gridwidth = GridBagConstraints.RELATIVE;
58      addComponent( buttons[ 1 ] );
59
60      // buttons[2] -- weightx and weighty are 1: fill is BOTH
61      constraints.gridwidth = GridBagConstraints.REMAINDER;
62      addComponent( buttons[ 2 ] );
63
64      // comboBox -- weightx is 1: fill is BOTH
65      constraints.weighty = 0;
66      constraints.gridwidth = GridBagConstraints.REMAINDER;
67      addComponent( comboBox );
68
69      // buttons[3] -- weightx is 1: fill is BOTH
70      constraints.weighty = 1;
71      constraints.gridwidth = GridBagConstraints.REMAINDER;
72      addComponent( buttons[ 3 ] );
73
74      // buttons[4] -- weightx and weighty are 1: fill is BOTH
75      constraints.gridwidth = GridBagConstraints.RELATIVE;
76      addComponent( buttons[ 4 ] );

```

Fig. 13.21 Demonstrating the `GridBagConstraints` constants `RELATIVE` and `REMAINDER` (part 2 of 3).

```

77
78     // list -- weightx and weighty are 1: fill is BOTH
79     constraints.gridwidth = GridBagConstraints.REMAINDER;
80     addComponent( list );
81
82     setSize( 300, 200 );
83     setVisible( true );
84
85 } // end constructor
86
87 // addComponent is programmer-defined
88 private void addComponent( Component component )
89 {
90     layout.setConstraints( component, constraints );
91     container.add( component ); // add component
92 }
93
94 // execute application
95 public static void main( String args[] )
96 {
97     GridBagDemo2 application = new GridBagDemo2();
98
99     application.setDefaultCloseOperation(
100         JFrame.EXIT_ON_CLOSE );
101 }
102
103 } // end class GridBagDemo2

```

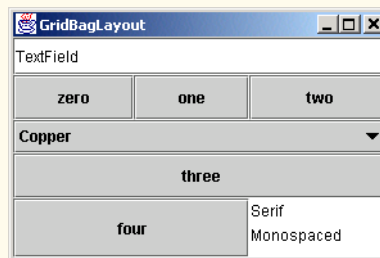


Fig. 13.21 Demonstrating the **GridBagConstraints** constants **RELATIVE** and **REMAINDER** (part 3 of 3).

Lines 22–23 construct a **GridBagLayout** and set the content pane’s layout manager to **GridBagLayout**. The components that are placed in **GridBagLayout** are each constructed (lines 29–42). The components are five **JButtons**, one **JTextField**, one **JList** and one **JComboBox**.

The **JTextField** is added first (lines 46–50). The **weightx** and **weighty** values are set to 1. The **fill** variable is set to **BOTH**. Line 49 specifies that the **JTextField** is the last component on the line. The **JTextField** is added to the content pane with a call to our utility method **addComponent** (defined at lines 88–92). Method **addComponent** takes a **Component** argument and uses **GridBagLayout** method **setConstraints** to set the constraints for the **Component**. Method **add** attaches the component to the content pane.

JButton buttons[0] (lines 53–54) has **weightx** and **weighty** values of 1. The **fill** variable is **BOTH**. Because **buttons[0]** is not one of the last two components on the row, it is given a **gridwidth** of 1 so it will occupy one column. The **JButton** is added to the content pane with a call to utility method **addComponent**.

JButton buttons[1] (lines 57–58) has **weightx** and **weighty** values of 1. The **fill** variable is **BOTH**. Line 57 specifies that the **JButton** is to be placed relative to the previous component. The **Button** is added to the **JFrame** with a call to **addComponent**.

JButton buttons[2] (lines 61–62) has **weightx** and **weighty** values of 1. The **fill** variable is **BOTH**. This **JButton** is the last component on the line, so **REMAINDER** is used. The **JButton** is added to the content pane with a call to utility method **addComponent**.

The **JComboBox** button (lines 65–67) has a **weightx** 1 and a **weighty** 0. The **JComboBox** will not grow in the vertical direction. The **JComboBox** is the only component on the line, so **REMAINDER** is used. The **JComboBox** is added to the content pane with a call to utility method **addComponent**.

JButton buttons[3] (lines 70–72) has **weightx** and **weighty** values of 1. The **fill** variable is **BOTH**. This **JButton** is the only component on the line, so **REMAINDER** is used. The **JButton** is added to the content pane with a call to utility method **addComponent**.

JButton buttons[4] (lines 75–76) has **weightx** and **weighty** values of 1. The **fill** variable is **BOTH**. This **JButton** is the next-to-last component on the line, so **RELATIVE** is used. The **JButton** is added to the content pane with a call to utility method **addComponent**.

The **JList** component (lines 79–80) has **weightx** and **weighty** values of 1. The **fill** variable is **BOTH**. The **JList** is added to the content pane with a call to utility method **addComponent**.

13.17 (Optional Case Study) Thinking About Objects: Model-View-Controller

Design patterns describe proven strategies for building reliable object-oriented software systems. Our case study adheres to the *Model-View-Controller* (MVC) architecture, which uses several design patterns.¹ MVC divides system responsibilities into three parts:

1. the *model*, which contains all program data and logic;
2. the *view*, which provides a visual presentation of the model and
3. the *controller*, which defines the system behavior by sending user input to the model.

Using the controller, the user changes the data in the model. The model then informs the view of the change in data. The view changes its visual presentation to reflect the changes in the model.

For example, in our simulation, the user adds a **Person** to the model by pressing either the **First Floor** or **Second Floor JButton** in the controller (see Fig. 2.22–

1. For those readers who seek further study in design patterns and MVC architecture, we encourage you to read our “Discovering Design Patterns” material in Sections 1.16, 9.24, 13.18, 15.13, 17.11 and 21.12

Fig. 2.24). The model then notifies the view of the **Person**'s creation. The view, in response to this notification, displays a **Person** on a **Floor**. The model is unaware of how the view displays the **Person**, and the view is unaware of how or why the model created the **Person**.

The MVC architecture helps construct reliable and easily modifiable systems. If we desire text-based output rather than graphical output for the elevator simulation, we may create an alternate view to produce text-based output, without altering the model or the controller. We could also provide a three-dimensional view that uses a first-person perspective to allow the user to “take part” in the simulation; such views are commonly employed in virtual-reality-based systems.

Elevator-Simulation MVC

We now apply the MVC architecture to our elevator simulation. Every UML diagram we have provided to this point (with the exception of the use-case diagram) relates to the model of our elevator system. We provide a “higher-level” class diagram of the simulation in Fig. 13.19. Class **ElevatorSimulation**—a **JFrame** subclass—aggregates one instance each of classes **ElevatorModel**, **ElevatorView** and **ElevatorController** to create the **ElevatorSimulation** application. As mentioned in “Thinking About Objects” Section 10.22, the rectangle with the upper-right corner “folded over” represents a note in the UML. In this case, each note points to a specific class (through a dotted line) to describe that class' role in the system. Classes **ElevatorModel**, **ElevatorView** and **ElevatorController** encapsulate all objects comprising the model, view and controller portions of our simulation, respectively.

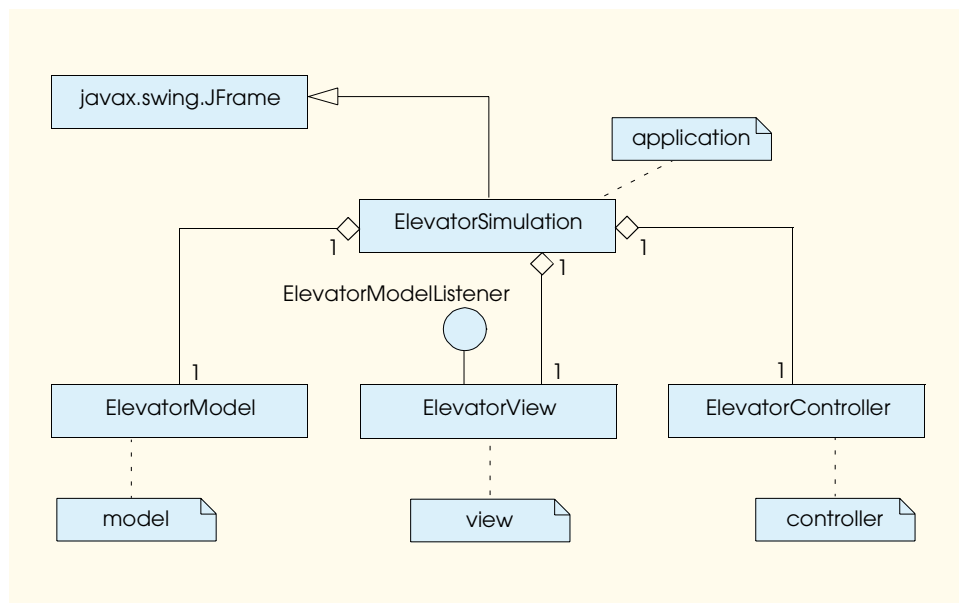


Fig. 13.19 Class diagram of the elevator simulation.

We showed in Fig. 9.38 that class **ElevatorModel** is an aggregation of several classes. To save space, we do not repeat this aggregation in Fig. 13.19. Class **ElevatorView** is also an aggregation of several classes—we expand the class diagram of **ElevatorView** in “Thinking About Objects” Section 22.9 to show these additional classes. Class **ElevatorController**, as described in Section 12.16, represents the simulation controller. Note that class **ElevatorView** implements interface **ElevatorModelListener**, which implements all interfaces used in our simulation, so the **ElevatorView** can receive all events from the model.



Software Engineering Observation 13.7

When appropriate, partition the system class diagram into several smaller class diagrams, so each diagram represents a unique subsystem.

Class **ElevatorSimulation** contains no attributes other than its references to an **ElevatorModel** object, an **ElevatorView** object and an **ElevatorController** object. The only behavior for class **ElevatorSimulation** is to start the program—therefore, in Java, class **ElevatorSimulation** contains a **static main** method that calls the constructor, which instantiates the **ElevatorModel**, **ElevatorView** and **ElevatorController** objects. We implement class **ElevatorSimulation** in Java later in this section.

Component Diagrams

Figure 13.19 helps us construct another diagram of the UML that we use to design our system—the *component diagram*. The component diagram models the “pieces”—called *components*—that the system needs to perform its tasks. These pieces include binary executables, compiled **.class** files, **.java** files, images, packages, resources, etc. We present the component diagram for our simulation in Fig. 13.20.

In Fig. 13.20, each box that contains the two small white boxes overlapping its left side is a *component*. A component in the UML is drawn similar to a plug (the two overlapping boxes represent the plug’s prongs)—a component may be “plugged-in” to other systems without having to change the component. Our system contains five components: **ElevatorSimulation.class**, **ElevatorSimulation.java**, **ElevatorModel.java**, **ElevatorView.java** and **ElevatorController.java**.

In Fig. 13.20, the graphics that resemble folders (boxes with tabs in their upper-left corners) represent *packages* in the UML. We can group classes, objects, components, use cases, etc., in a package. In this diagram (and in the remainder of our case study), the UML packages refer to Java packages (introduced in Section 8.5). In our discussion, we use lower-case bold-face Courier type for package names. The packages in our system are **model**, **view** and **controller**. Component **ElevatorSimulation.java** contains one instance each of all components in these packages. Currently, each package contains only one component—a **.java** file. The **model** package contains **ElevatorModel.java**, the **view** package contains **ElevatorView.java** and the **controller** package contains **ElevatorController.java**. We add components to each package in the appendices, when we implement each class from our class diagrams into a component (**.java** file).

The dotted arrows in Fig. 13.20 indicate a *dependency* between two components—the direction of the arrow indicates the “depends on” relationship. A dependency describes the

relationship between components in which changes in one component affect another component. For example, component **ElevatorSimulation.class** depends on component **ElevatorSimulation.java**, because a change in **ElevatorSimulation.java** affects **ElevatorSimulation.class** when **ElevatorSimulation.java** is compiled. Section 12.16 mentioned that the **ElevatorController** object contains a reference to the **ElevatorModel** object (to place **Persons** on **Floors**). Therefore, **ElevatorController.java** depends on **ElevatorModel.java**.



Software Engineering Observation 13.8

A component diagram's dependencies help designers group components for reuse in future systems. For example, in our simulation, designers can reuse **ElevatorModel.java** in other systems without having to reuse **ElevatorView.java** (and vice versa), because these components do not depend on each other. However, if a designer wanted to reuse **ElevatorController.java**, the designer would have to reuse **ElevatorModel.java**.

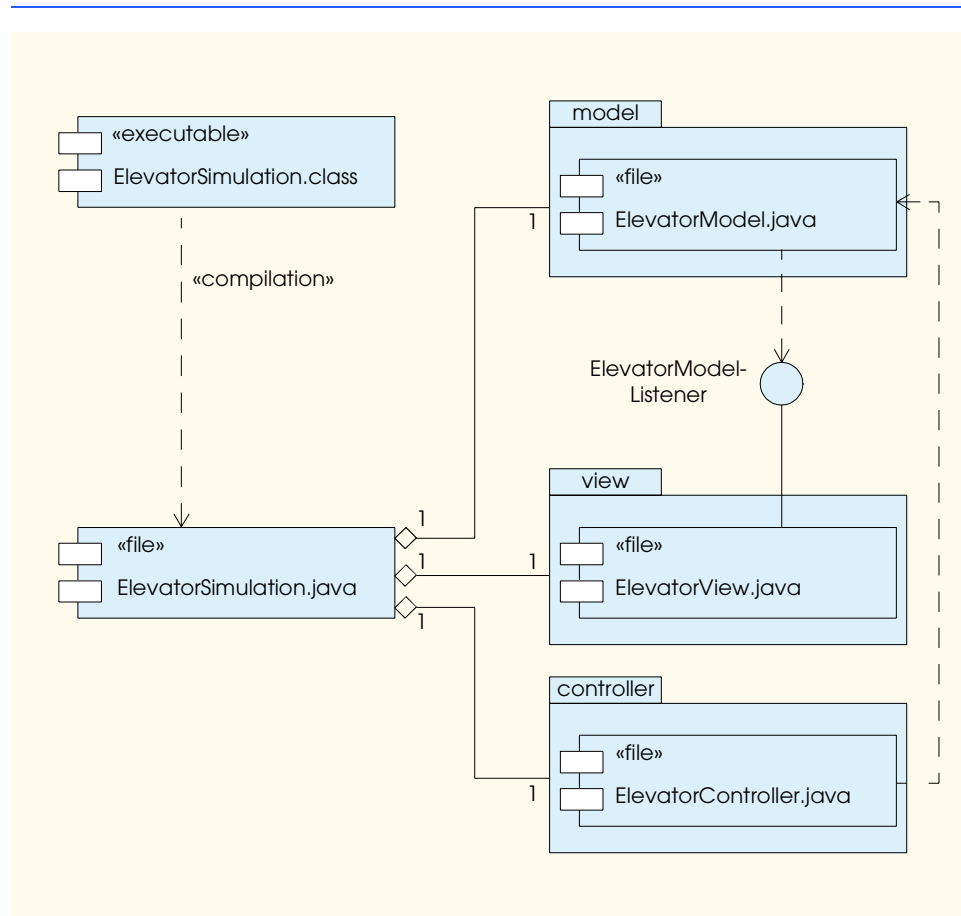


Fig. 13.20 Component diagram for elevator simulation.

According to Fig. 13.20, **ElevatorModel.java** and **ElevatorView.java** do not depend on each other—they communicate through interface **ElevatorModelListener**, which implements all interfaces in the simulation. **ElevatorView.java** realizes interface **ElevatorModelListener**, and **ElevatorModel.java** depends on interface **ElevatorModelListener**.

Figure 13.20 contains several *stereotypes*—words placed in *guillemets* (« ») indicating an element's role. We mentioned the «interface» stereotype in “Thinking About Objects” Section 11.10. The «compilation» stereotype describes the dependency between **ElevatorSimulation.class** and **ElevatorSimulation.java**—**ElevatorSimulation.java** compiles to **ElevatorSimulation.class**. The «executable» stereotype specifies that a component is an application, and the «file» stereotype specifies that a component is a file containing source code for the executable.

Implementation: **ElevatorSimulation.java**

We use the component diagram of Fig. 13.20, the class diagram of Fig. 13.19 and the use-case diagram of Fig. 12.28 to implement **ElevatorSimulation.java** (Fig. 13.21). Lines 12–14 import packages **model**, **view** and **controller** as specified in Fig. 13.20.

```

1  // ElevatorSimulation.java
2  // Application with Elevator Model, View, and Controller (MVC)
3  package com.deitel.jhtp4.elevator;
4
5  // Java core packages
6  import java.awt.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 // Deitel packages
12 import com.deitel.jhtp4.elevator.model.*;
13 import com.deitel.jhtp4.elevator.view.*;
14 import com.deitel.jhtp4.elevator.controller.*;
15
16 public class ElevatorSimulation extends JFrame {
17
18     // model, view and controller
19     private ElevatorModel model;
20     private ElevatorView view;
21     private ElevatorController controller;
22
23     // constructor instantiates model, view, and controller
24     public ElevatorSimulation()
25     {
26         super( "Deitel Elevator Simulation" );
27
28         // instantiate model, view and controller
29         model = new ElevatorModel();
30         view = new ElevatorView();
31         controller = new ElevatorController( model );

```

Fig. 13.21 Class **ElevatorSimulation** is the application for the elevator simulation (part 1 of 2).

```

32
33     // register View for Model events
34     model.setElevatorModelListener( view );
35
36     // add view and controller to ElevatorSimulation
37     getContentPane().add( view, BorderLayout.CENTER );
38     getContentPane().add( controller, BorderLayout.SOUTH );
39
40 } // end ElevatorSimulation constructor
41
42 // main method starts program
43 public static void main( String args[] )
44 {
45     // instantiate ElevatorSimulation
46     ElevatorSimulation simulation = new ElevatorSimulation();
47     simulation.setDefaultCloseOperation( EXIT_ON_CLOSE );
48     simulation.pack();
49     simulation.setVisible( true );
50 }
51 }

```

Fig. 13.21 Class **ElevatorSimulation** is the application for the elevator simulation (part 2 of 2).

The class diagram of Fig. 13.19 specifies that class **ElevatorSimulation** is a subclass of **javax.swing.JFrame**—line 16 declares class **ElevatorSimulation** as a **public** class extending class **JFrame**. Lines 19–21 implement class **ElevatorSimulation**'s aggregation of class **ElevatorModel**, the **ElevatorView** and the **ElevatorController** (shown in Fig. 13.19) by declaring one object from each class. Lines 29–31 of the **ElevatorSimulation** constructor initialize these objects.

Figure 13.19 and Fig. 13.20 specify that the **ElevatorView** is an **ElevatorModelListener** for the **ElevatorModel**. Line 34 registers the **ElevatorView** as a listener for **ElevatorModelEvents**, so the **ElevatorView** can receive events from the **ElevatorModel** and properly represent the state of the model. Lines 37–38 add the **ElevatorView** and the **ElevatorController** to the **ElevatorSimulation**. According to the stereotypes in Fig. 13.20, **ElevatorSimulation.java** compiles to **ElevatorSimulation.class**, which is executable—lines 43–50 provide method **main** that runs the application.

We have completed the design of the components of our system. “Thinking About Objects” Section 15.12 concludes the design of the model by solving the interaction problems encountered in Fig. 10.26. Finally, Section 22.9 completes the design of the view and describes in greater detail how the **ElevatorView** receives events from the **ElevatorModel**. These last two sections will prepare you for the walkthrough of our elevator-simulation implementation in Appendices G, H and I.

13.18 (Optional) Discovering Design Patterns: Design Patterns Used in Packages **java.awt** and **javax.swing**

We continue our discussion from Section 9.24 on design patterns. This section introduces those design patterns associated with Java GUI components. After reading this section, you

should understand better how these components take advantage of design patterns and how developers integrate design patterns with Java GUI applications.

13.18.1 Creational Design Patterns

Now, we continue our treatment of creational design patterns, which provide ways to instantiate objects in a system.

Factory Method

Suppose we are designing a system that opens an image from a specified file. Several different image formats exist, such as GIF and JPEG. We can use method **createImage** of class **java.awt.Component** to create an **Image** object. For example, to create a JPEG and GIF image in an object of a **Component** subclass—such as a **JPanel** object, we pass the name of the image file to method **createImage**, which returns an **Image** object that stores the image data. We can create two **Image** objects, each which contains data for two images having entirely different structures. For example, a JPEG image can hold up to 16.7 million colors, whereas a GIF image can hold up to only 256. Also, a GIF image can contain transparent pixels that are not rendered on screen, whereas a JPEG image cannot contain transparent pixels.

Class **Image** is an abstract class that represents an image we can display on screen. Using the parameter passed by the programmer, method **createImage** determines the specific **Image** subclass from which to instantiate the **Image** object. We can design systems to allow the user to specify which image to create, and method **createImage** will determine the subclass from which to instantiate the **Image**. If the parameter passed to method **createImage** references a JPEG file, method **createImage** instantiates and returns an object of an **Image** subclass suitable for JPEG images. If the parameter references a GIF file, **createImage** instantiates and returns an object of an **Image** subclass suitable for GIF images.

Method **createImage** is an example of the *Factory Method design pattern*. The sole purpose of this *factory method* is to create objects by allowing the system to determine which class to instantiate at run time. We can design a system that allows a user to specify what type of image to create at run time. Class **Component** might not be able to determine which **Image** subclass to instantiate until the user specifies the image to load. For more information on method **createImage**, visit

[www.java.sun.com/j2se/1.3/docs/api/java/awt/Component.html#createImage\(java.awt.image.ImageProducer\)](http://www.java.sun.com/j2se/1.3/docs/api/java/awt/Component.html#createImage(java.awt.image.ImageProducer))

13.18.2 Structural Design Patterns

We now discuss three more structural design patterns. The Adapter design pattern helps objects with incompatible interfaces collaborate with one another. The Bridge design pattern helps designers enhance platform independence in their systems. The Composite design pattern provides a way for designers to organize and manipulate objects.

Adapter

The *Adapter design pattern* provides an object with a new interface that *adapts* to another object's interface, allowing both objects to collaborate with one another. The adapter in this

pattern is similar to an adapter for a plug on an electrical device—electrical sockets in Europe are different from those in the United States, so an adapter is needed to plug an American device into a European electrical socket and vice versa.

Java provides several classes that use the Adapter design pattern. Objects of these classes act as adapters between objects that generate certain events and those objects that handle these events. For example, a **MouseAdapter**, which we explained in Section 12.12, adapts an object that generates **MouseEvent**s to an object that handles **MouseEvent**s.

Bridge

Suppose we are designing class **Button** for both the Windows and Macintosh operating systems. Class **Button** contains specific button information such as an **ActionListener** and a **String** label. We design classes **Win32Button** and **MacButton** to extend class **Button**. Class **Win32Button** contains “look-and-feel” information on how to display a **Button** on the Windows operating system and class **MacButton** contains “look-and-feel” information on how to display a **Button** on the Macintosh operating system.

Two problems arise from this approach. First, if we create new **Button** subclasses, we must create corresponding **Win32Button** and **MacButton** subclasses. For example, if we create class **ImageButton** (a **Button** with an overlapping **Image**) that extends class **Button**, we must create additional subclasses **Win32ImageButton** and **MacImageButton**. In fact, we must create **Button** subclasses for every operating system we wish to support, which increases development time. The second problem is that when a new operating system enters the market, we must create additional **Button** subclasses specific to that operating system.

The *Bridge design pattern* avoids these problems by separating an abstraction (e.g., a **Button**) and its implementations (e.g., **Win32Button**, **MacButton**, etc.) into different class hierarchies. For example, the Java AWT classes use the Bridge design pattern to enable designers to create AWT **Button** subclasses without needing to create corresponding subclasses for each operating system. Each AWT **Button** maintains a reference to a **ButtonPeer**, which is the superclass for platform-specific implementations, such as **Win32ButtonPeer**, **MacButtonPeer**, etc. When a programmer creates a **Button** object, class **Button** calls factory method **createButton** of class **Toolkit** to create the platform-specific **ButtonPeer** object. The **Button** object stores a reference to its **ButtonPeer**—this reference is the “bridge” in the Bridge design pattern. When the programmer invokes methods on the **Button** object, the **Button** object invokes the appropriate method on its **ButtonPeer** to fulfill the request. If a designer creates a **Button** subclass called **ImageButton**, the designer does not need to create a corresponding **Win32ImageButton** or **MacImageButton** with platform-specific image-drawing capabilities. An **ImageButton** “is a” **Button**. Therefore, when an **ImageButton** needs to display its image, the **ImageButton** uses its **ButtonPeer**’s **Graphics** object to render the image on each platform. This design pattern enables designers to create new cross-platform GUI components using a “bridge” to hide platform-specific details.



Portability Tip 13.2

Designers often use the Bridge design pattern to enhance the platform independence of their systems. This design pattern enables designers to create new cross-platform components using a “bridge” to hide platform-specific details.

Composite

Designers often organize components into hierarchical structures (e.g., a hierarchy of directories and files on a hard drive)—each node in the structure represents a component (e.g., a file or directory). Each node can contain references to other nodes. A node is called a *branch* if it contains a reference to one or more nodes (e.g., a directory containing files). A node is called a *leaf* if it does not contain a reference to another node (e.g., a file). Occasionally, a structure contains objects from several different classes (e.g., a directory can contain files and directories). When an object—called a *client*—wants to traverse the structure, the client must determine the particular class for each node. Making this determination can be time consuming, and the structure can become hard to maintain.

In the *Composite design pattern*, each component in a hierarchical structure implements the same interface or extends a common superclass. This polymorphism (introduced in Section 9.10) ensures that clients can traverse all elements—branch or leaf—uniformly in the structure. Using this pattern, a client traversing the structure does not have to determine each component type, because all components implement the same interface or extend the same superclass.

Java GUI components use the Composite design pattern. Consider the Swing component class **JPanel**, which extends class **JComponent**. Class **JComponent** extends class **java.awt.Container**, which extends class **java.awt.Component** (Fig. 13.22). Class **Container** provides method **add**, which appends a **Component** object (or **Component** subclass object) to that **Container** object. Therefore, a **JPanel** object may be added to any object of a **Component** subclass, and any object from a **Component** subclass may be added to that **JPanel** object. A **JPanel** object can contain any GUI component while remaining unaware of that component's specific type.

A client, such as a **JPanel** object, can traverse all components uniformly in the hierarchy. For example, if the **JPanel** object calls method **repaint** of superclass **Container**, method **repaint** displays the **JPanel** object and all components added to the **JPanel** object. Method **repaint** does not have to determine each component's type, because all components inherit from superclass **Container**, which contains method **repaint**.

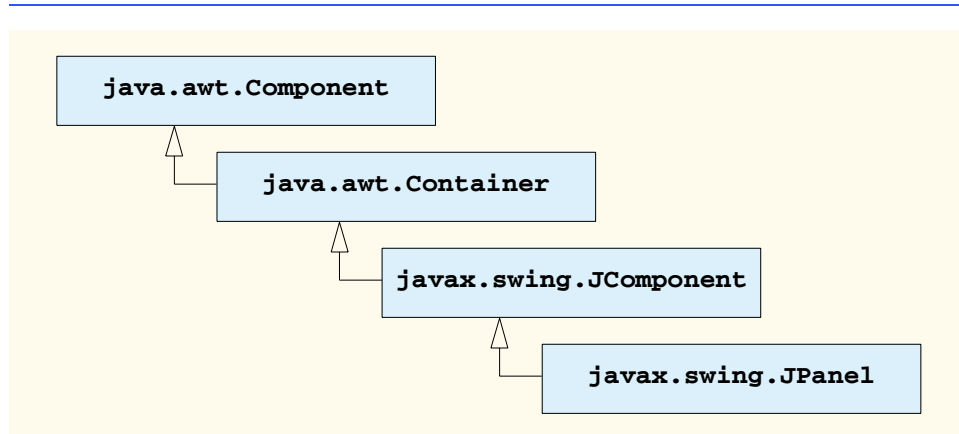


Fig. 13.22 Inheritance hierarchy for class **JPanel**.

13.18.3 Behavioral Design Patterns

In this section, we continue our discussion on behavioral design patterns. We discuss the Chain-of-Responsibility, Command, Observer, Strategy and Template Method design patterns.

Chain-of-Responsibility

In object-oriented systems, objects interact by sending messages to one another. Often, a system needs to determine at run time the object that will handle a particular message. For example, consider the design of a three-line office phone system. When a person calls the office, the first line handles the call—if the first line is busy, the second line handles the call, and if the second line is busy, the third line handles the call. If all lines in the system are busy, an automated speaker instructs that person to wait for the next available line—when a line becomes available, that line handles the call.

The *Chain-of-Responsibility design pattern* enables a system to determine at run time the object that will handle a message. This pattern allows an object to send a message to several objects in a *chain* of objects. Each object in the chain either may handle the message or pass the message to the next object in the chain. For instance, the first line in the phone system is the first object in the chain of responsibility, the second line is the second object, the third line is the third object, and the automated speaker is the fourth object. Note that this mechanism is not the final object in the chain—the next available line handles the message, and that line is the final object in the chain. The chain is created dynamically in response to the presence or absence of specific message handlers.

Several Java AWT GUI components use the Chain-of-Responsibility design pattern to handle certain events. For example, class `java.awt.Button` overrides method `processEvent` of class `java.awt.Component` to process `AWTEvent` objects. Method `processEvent` attempts to handle the `AWTEvent` upon receiving this event as an argument. If method `processEvent` determines that the `AWTEvent` is an `ActionEvent` (i.e., the `Button` has been pressed), the method handles the event by invoking method `processActionEvent`, which informs any `ActionListener` registered with the `Button` that the `Button` has been pressed. If method `processEvent` determines that the `AWTEvent` is not an `ActionEvent`, the method is unable to handle the event and passes the `AWTEvent` to method `processEvent` of superclass `Component` (the next object in the chain).

Command

Applications often provide users with several ways to perform a given task. For example, in a word processor there might be an **Edit** menu with menu items for cutting, copying and pasting text. There could also be a toolbar and/or a popup menu offering the same items. The functionality the application provides is the same in each case—the different interface components for invoking the functionality are provided for the user's convenience. However, the same GUI component instance (e.g., `JButton`) cannot be used for menus and toolbars and popup menus, so the developer must code the same functionality three times. If there were many such interface items, repeating this functionality would become tedious and error-prone.

The *Command design pattern* solves this problem by enabling developers to specify the desired functionality (e.g., copying text) once in a reusable object; that functionality can

then be added to a menu, toolbar, popup menu or other mechanisms. This design pattern is called Command because it defines a user command, or instruction. This pattern allows a designer to encapsulate a command, so that the command may be used among several objects.

Observer

Suppose we want to design a program for viewing bank account information. This system includes class **BankStatementData** to store data pertaining to bank statements, and classes **TextDisplay**, **BarGraphDisplay** and **PieChartDisplay** to display the data.² Figure 13.23 shows the design for our system. Class **TextDisplay** displays the data in text format, class **BarGraphDisplay** displays the data in bar-graph format and class **PieChartDisplay** displays the data as a pie chart. We want to design the system so that the **BankStatementData** object notifies the objects displaying the data of a change in the data. We also want to design the system to loosen *coupling*—the degree to which classes depend on each other in a system.



Software Engineering Observation 13.9

Loosely-coupled classes are easier to reuse and modify than are tightly-coupled classes, which depend heavily on each other. A modification in a class in a tightly-coupled system usually results in modifying other classes in that system. A modification to one of a group of loosely-coupled classes would require little or no modification to the other classes in the group.

The *Observer design pattern* is appropriate for systems like that of Fig. 13.23. This pattern promotes loose coupling between a *subject* object and *observer* objects—a subject notifies the observers when the subject changes state. When notified by the subject, the observers change in response to the change in the subject. In our example, the **BankStatementData** object is the subject, and the objects displaying the data are the observers. A subject can notify several observers; therefore, the subject contains a one-to-many relationship with the observers.

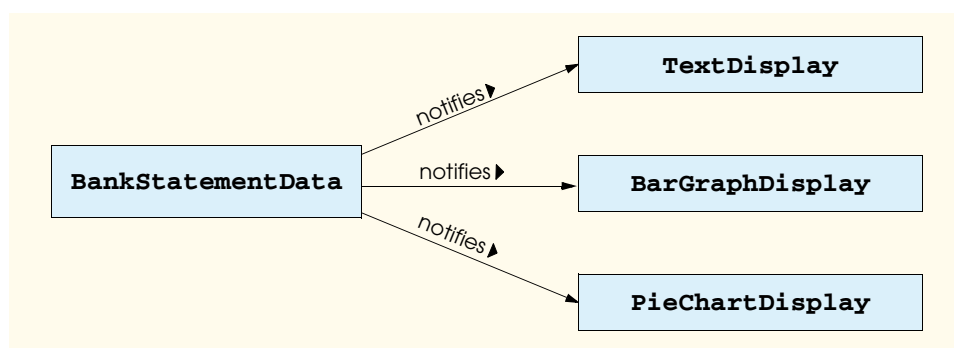


Fig. 13.23 Basis for the Observer design pattern.

2. This approach is the basis for the Model-View-Controller architecture pattern, discussed in Sections 13.17 and 17.11.

The Java API contains classes that use the Observer design pattern. Class `java.util.Observable` represents a subject. Class `Observable` provides method `addObserver`, which takes a `java.util.Observer` argument. Interface `Observer` allows the `Observable` object to notify the `Observer` when the `Observable` object changes state. The `Observer` can be an instance of any class that implements interface `Observer`; because the `Observable` object invokes methods defined in interface `Observer`, the objects remain loosely coupled. If a developer changes the way in which a particular `Observer` responds to changes in the `Observable` object, the developer does not need to change the `Observable` object. The `Observable` object interacts with its `Observers` only through interface `Observer`, which enables the loose coupling.

The optional elevator simulation case study in the “Thinking About Objects” sections uses the Observer design pattern to allow the `ElevatorModel` object (the subject) to notify the `ElevatorView` object (the observer) of changes in the `ElevatorModel`. The simulation does not use class `Observable` and interface `Observer` from the Java library—rather, it uses a custom interface `ElevatorModelListener` that provides functionality similar to that of interface `Observable`.

The Swing GUI components use the Observer design pattern. GUI components collaborate with their listeners to respond to user interactions. For example, an `ActionListener` observes state changes in a `JButton` (the subject) by registering to handle that `JButton`’s events. When pressed by the user, the `JButton` notifies its `ActionListener` objects (the observers) that the `JButton`’s state has changed (i.e., the `JButton` has been pressed).

Strategy

The *Strategy design pattern* is similar to the State design pattern (discussed in Section 9.24.3). We mentioned that the State design pattern contains a state object, which encapsulates the state of a context object. The Strategy design pattern contains a *strategy object*, which is analogous to the State design pattern’s state object. The key difference between a state object and a strategy object is that the strategy object encapsulates an *algorithm* rather than state information.

For example, `java.awt.Container` components implement the Strategy design pattern using `LayoutManagers` (discussed in Section 12.14) as strategy objects. In package `java.awt`, classes `FlowLayout`, `BorderLayout` and `GridLayout` implement interface `LayoutManager`. Each class uses method `addLayoutComponent` to add GUI components to a `Container` object—however, each method uses a different algorithm to display these GUI components: a `FlowLayout` displays components in a left-to-right sequence; a `BorderLayout` displays components in five regions; and a `GridLayout` displays components in row-column format.

Class `Container` contains a reference to a `LayoutManager` object (the strategy object). Because an interface reference (i.e., the reference to the `LayoutManager` object) can hold references to objects of classes that implement that interface (i.e., the `FlowLayout`, `BorderLayout` or `GridLayout` objects), the `LayoutManager` object can reference a `FlowLayout`, `BorderLayout` or `GridLayout` at any one time. Class `Container` can change this reference through method `setLayout` to select different layouts at run time.

Class `FlowLayoutDemo` (Fig. 12.24) demonstrates the application of the Strategy pattern—line 16 declares a new `FlowLayout` object and line 19 invokes the `Con-`

tainer object's method **setLayout** to assign the **FlowLayout** object to the **Container** object. In this example, the **FlowLayout** provides the strategy for laying out the components.

Template Method

The *Template Method design pattern* also deals with algorithms. The Strategy design pattern allows several objects to contain distinct algorithms. However, the Template Method design pattern requires all objects to share a single algorithm defined by a superclass.

For example, consider the design of Fig. 13.23, which we mentioned in the Observer design pattern discussion. Objects of classes **TextDisplay**, **BarGraphDisplay** and **PieChartDisplay** use the same basic algorithm for acquiring and displaying the data—get all statements from the **BankStatementData** object, parse the statements then display the statements. The Template Method design pattern allows us to create an abstract superclass called **BankStatementDisplay** that provides the central algorithm for displaying the data. In this example, the algorithm comprises abstract methods **getData**, **parseData** and **displayData** comprise the algorithm. Classes **TextDisplay**, **BarGraphDisplay** and **PieChartDisplay** extend class **BankStatementDisplay** to inherit the algorithm, so each object can use the same algorithm. Each **BankStatementDisplay** subclass then overrides each method in a way specific to that subclass, because each class implements the algorithm differently from one another. For example, classes **TextDisplay**, **BarGraphDisplay** and **PieChartDisplay** might get and parse the data identically, but each class displays that data differently.

The Template Method design pattern allows us to extend the algorithm to other **BankStatementDisplay** subclasses—e.g., we could create classes, such as **LineGraphDisplay** or class **3DimensionalDisplay**, that use the same algorithm inherited from class **BankStatementDisplay**.

13.18.4 Conclusion

In this “Discovering Design Patterns” section, we discussed how Swing components take advantage of design patterns and how developers can integrate design patterns with GUI applications in Java. In “Discovering Design Patterns” Section 15.13, we discuss concurrency design patterns, which are particularly useful for developing multithreaded systems.

SUMMARY

- **JTextAreas** provide an area for manipulating multiple lines of text. Like class **JTextField**, class **JTextArea** inherits from **JTextComponent**.
- An external event (i.e., an event generated by a different GUI component) normally indicates when the text in a **JTextArea** should be processed.
- Scrollbars are provided for a **JTextArea** by attaching it to a **JScrollPane** object.
- Method **getSelectedText** returns the selected text from a **JTextArea**. Text is selected by dragging the mouse over the desired text to highlight it.
- Method **setText** sets the text in a **JTextArea**.
- To provide automatic word wrap in a **JTextArea**, attach it to a **JScrollPane** with horizontal scrollbar policy **JScrollPane.HORIZONTAL_SCROLLBAR_NEVER**.

- The horizontal and vertical scrollbar policies for a **JScrollPane** are set when a **JScrollPane** is constructed or with methods **setHorizontalScrollBarPolicy** and **setVerticalScrollBarPolicy** of class **JScrollPane**.
- A **JPanel** can be used as a dedicated drawing area that can receive mouse events and is often extended to create new GUI components.
- Swing components that inherit from class **JComponent** contain method **paintComponent**, which helps them draw properly in the context of a Swing GUI. **JComponent** method **paintComponent** should be overridden to call to the superclass version of **paintComponent** as the first statement in its body.
- Classes **JFrame** and **JApplet** are not subclasses of **JComponent**; therefore, they do not contain method **paintComponent** (they have method **paint**).
- Calling **repaint** for a Swing GUI component indicates that the component should be painted as soon as possible. The background of the GUI component is cleared only if the component is opaque. Most Swing components are transparent by default. **JComponent** method **setOpaque** can be passed a **boolean** argument indicating whether the component is opaque (**true**) or transparent (**false**). The GUI components of package **java.awt** are different from Swing components in that **repaint** results in a call to **Component** method **update** (which clears the component's background) and **update** calls method **paint** (rather than **paintComponent**).
- Method **setTitle** displays a **String** in a window's title bar.
- Drawing on any GUI component is performed with coordinates that are measured from the upper-left corner (0, 0) of that GUI component.
- Layout managers often use a GUI component's **getPreferredSize** method to determine the preferred width and height of a component when laying out that component as part of a GUI. If a new component has a preferred width and height, it should override method **getPreferredSize** to return that width and height as an object of class **Dimension** (package **java.awt**).
- The default size of a **JPanel** object is 0 pixels wide and 0 pixels tall.
- A mouse drag operation begins with a mouse-pressed event. All subsequent mouse drag events (for which **mouseDragged** will be called) are sent to the GUI component that received the original mouse-pressed event.
- **JSliders** enable the user to select from a range of integer values. **JSliders** can display major tick marks, minor tick marks and labels for the tick marks. They also support snap-to ticks, where positioning the thumb between two tick marks causes the thumb to *snap* to the closest tick mark.
- Most Swing GUI components support user interactions through both the mouse and the keyboard.
- If a **JSlider** has the focus, the left arrow key and right arrow key cause the thumb of the **JSlider** to decrease or increase by 1. The down arrow key and up arrow key also cause the thumb of the **JSlider** to decrease or increase by 1, respectively. The *PgDn* key (page down) and *PgUp* key (page up) cause the thumb of the **JSlider** to decrease or increase by block increments of one-tenth of the range of values, respectively. The *Home* key moves the thumb to the minimum value of the **JSlider** and the *End* key moves the thumb to the maximum value of the **JSlider**.
- **JSliders** have either a horizontal orientation or a vertical orientation. For a horizontal **JSlider**, the minimum value is at the extreme left and the maximum value is at the extreme right of the **JSlider**. For a vertical **JSlider**, the minimum value is at the extreme bottom and the maximum value is at the extreme top of the **JSlider**. The relative position of the thumb indicates the current value of the **JSlider**.
- Method **setMajorTickSpacing** of class **JSlider** sets the spacing for tick marks on a **JSlider**. Method **setPaintTicks** with a **true** argument indicates that the tick marks should be displayed.

- **JSliders** generate **ChangeEvent**s (package **javax.swing.event**) when the user interacts with a **JSlider**. A **ChangeListener** (package **javax.swing.event**) defines method **stateChanged** that can respond to **ChangeEvent**s.
- Method **getValue** of class **JSlider** returns the current thumb position.
- A **JFrame** is a window with a title bar and a border. Class **JFrame** is a subclass of **java.awt.Frame** (which is a subclass of **java.awt.Window**).
- Class **JFrame** supports three operations when the user closes the window. By default, a **JFrame** is hidden when the user closes a window. This can be controlled with **JFrame** method **setDefaultCloseOperation**. Interface **WindowConstants** (package **javax.swing**) defines three constants for use with this method—**DISPOSE_ON_CLOSE**, **DO_NOTHING_ON_CLOSE** and **HIDE_ON_CLOSE** (the default).
- By default, a window is not displayed on the screen until its **setVisible** method is called with **true** as an argument. A window can also be displayed by calling its **show** method.
- A window's size should be set with a call to method **setSize**. The position of a window when it appears on the screen is specified with method **setLocation**.
- All windows generate window events when the user manipulates the window. Event listeners are registered for window events with method **addWindowListener** of class **Window**. The **WindowListener** interface provides seven methods for handling window events—**windowActivated** (called when the window is made active by clicking the window), **windowClosed** (called after the window is closed), **windowClosing** (called when the user initiates closing of the window), **windowDeactivated** (called when another window is made active), **windowIconified** (called when the user minimizes a window), **windowDeiconified** (called when a window is restored from being minimized) and **windowOpened** (called when a window is first displayed on the screen).
- The command-line arguments are automatically passed to **main** as the array of **Strings** called **args**. The first argument after the application class name is the first **String** in the array **args**, and the length of the array is the total number of command-line arguments.
- **Menus** are an integral part of GUIs. Menus allow the user to perform actions without unnecessarily "cluttering" a graphical user interface with extra GUI components.
- In Swing GUIs, menus can be attached only to objects of the classes that provide method **setJMenuBar**. Two such classes are **JFrame** and **JApplet**.
- The classes used to define menus are **JMenuBar**, **JMenuItem**, **JMenu**, **JCheckBoxMenuItem** and class **JRadioButtonMenuItem**.
- A **JMenuBar** is a container for menus.
- A **JMenuItem** is a GUI component inside a menu that, when selected, causes an action to be performed. A **JMenuItem** can be used to initiate an action or it can be a *submenu* that provides more menu items from which the user can select.
- A **JMenu** contains menu items and can be added to a **JMenuBar** or to other **JMenu**s as submenus. When a menu is clicked, the menu expands to show its list of menu items.
- When a **JCheckBoxMenuItem** is selected, a check appears to the left of the menu item. When the **JCheckBoxMenuItem** is selected again, the check to the left of the menu item is removed.
- When multiple **JRadioButtonMenuItems** are maintained as part of a **ButtonGroup**, only one item in the group can be selected at a given time. When a **JRadioButtonMenuItem** is selected, a filled circle appears to the left of the menu item. When another **JRadioButtonMenuItem** is selected, the filled circle to the left of the previously selected menu item is removed.
- **JFrame** method **setJMenuBar** attaches a menu bar to a **JFrame**.

- **AbstractButton** method **setMnemonic** (inherited into class **JMenu**) specifies the mnemonic for an **AbstractButton** object. Pressing the *Alt* key and the mnemonic performs the **AbstractButton**'s action (in the case of a menu, it opens the menu).
- Mnemonic characters are normally displayed with an underline.
- Dialog boxes can be either modal or modeless. A modal dialog box does not allow any other window in the application to be accessed until the dialog box is dismissed. A modeless dialog box allows other windows to be accessed while the dialog is displayed. By default, the dialogs displayed with class **JOptionPane** are modal dialogs. Class **JDialog** can be used to create your own modeless or modal dialogs.
- **JMenu** method **addSeparator** adds a separator line to a menu.
- Context-sensitive popup menus are created with class **JPopupMenu**. These menus provide options that are specific to the component for which the popup-trigger event was generated. On most systems, the popup-trigger event occurs when the user presses and releases the right mouse button.
- **MouseEvent** method **isPopupTrigger** returns **true** if the popup-trigger event occurred.
- Method **show** of class **JPopupMenu** displays a **JPopupMenu**. The first argument to method **show** specifies the origin component, whose position helps determine where the **JPopupMenu** will appear on the screen. The last two arguments are the x and y coordinates from the origin component's upper-left corner at which the **JPopupMenu** should appear.
- Class **UIManager** contains a **public static** inner class called **LookAndFeelInfo** that is used to maintain information about a look-and-feel.
- **UIManager static** method **getInstalledLookAndFeels** gets an array of **UIManager.LookAndFeelInfo** objects that describe the installed look-and-feels.
- **UIManager static** method **setLookAndFeel** changes the look-and-feel.
- **SwingUtilities static** method **updateComponentTreeUI** changes the look-and-feel of every component attached to its **Component** argument to the new look-and-feel.
- Many of today's applications use a multiple document interface (MDI) [i.e., a main window (often called the parent window) containing other windows (often called child windows)] to manage several open documents that are being processed in parallel.
- Swing's **JDesktopPane** and **JInternalFrame** classes provide support for creating multiple document interfaces.
- **BoxLayout** is a layout manager that allows GUI components to be arranged left-to-right or top-to-bottom in a container. Class **Box** defines a container with **BoxLayout** as its default layout manager and provides static methods to create a **Box** with a horizontal or vertical **BoxLayout**.
- **CardLayout** is a layout manager that stacks components like a deck of cards. Each container in the stack can use any layout manager. Only the container at the "top" of the deck is visible.
- **GridBagLayout** is a layout manager similar to **GridLayout**. Unlike with **GridLayout**, each component size can vary, and components can be added in any order.
- **Box static** method **createHorizontalBox** returns a **Box** container with a horizontal **BoxLayout**. **Box static** method **createVerticalBox** of class **Box** returns a **Box** container with a vertical **BoxLayout**.
- **Box static** method **createVerticalStrut** adds a *vertical strut* to a container. A vertical strut is an invisible GUI component that has a fixed pixel height and is used to guarantee a fixed amount of space between GUI components. Class **Box** also defines method **createHorizontalStrut** for horizontal **BoxLayouts**.
- **Box static** method **createHorizontalGlue** adds horizontal glue to a container. Horizontal glue is an invisible GUI component that can be used between fixed-size GUI components to

occupy additional space. Class **Box** also defines method **createVerticalGlue** for vertical **BoxLayout**s.

- **Box** static method **createRigidArea** adds a rigid area to a container. A rigid area is an invisible GUI component that always has a fixed pixel width and height.
- The **BoxLayout** constructor receives a reference to the container for which it controls the layout and a constant indicating whether the layout is horizontal (**BoxLayout.X_AXIS**) or vertical (**BoxLayout.Y_AXIS**).
- **CardLayout** methods **first**, **previous**, **next** and **last** are used to display a particular card. Method **first** displays the first card. Method **previous** displays the previous card. Method **next** displays the next card. Method **last** displays the last card.
- To use **GridBagLayout**, a **GridBagConstraints** object must be used to specify how a component is placed in a **GridBagLayout**.
- Method **setConstraints** of class **GridBagLayout** takes a **Component** argument and a **GridBagConstraints** argument and sets the constraints of the **Component**.

TERMINOLOGY

addSeparator method of class **JMenu**

addWindowListener method of **Window**

anchor variable of **GridBagConstraints**

automatic word wrap

Box class

BoxLayout layout manager

BoxLayout.X_AXIS

BoxLayout.Y_AXIS

CardLayout layout manager

ChangeEvent class

ChangeListener interface

child window

command-line arguments

context-sensitive popup menu

createHorizontalBox method of **Box**

createHorizontalGlue method of **Box**

createHorizontalStrut method of **Box**

createRigidArea method of **Box**

createVerticalBox method of **Box**

createVerticalGlue method of **Box**

createVerticalStrut method of **Box**

dedicated drawing area

Dimension class

dispose method of class **Window**

external event

fill variable of **GridBagConstraints**

first method of **CardLayout**

getClassName method

getInstalledLookAndFeel method

getMinimumSize method of **Component**

getPreferredSize method of **Component**

getSelectedText method

getValue method of class **JSlider**

GridBagConstraints class

GridBagConstraints.BOTH

GridBagConstraints.CENTER

GridBagConstraints.EAST

GridBagConstraints.HORIZONTAL

GridBagConstraints.NONE

GridBagConstraints.NORTH

GridBagConstraints.NORTHEAST

GridBagConstraints.NORTHWEST

GridBagConstraints.RELATIVE

GridBagConstraints.REMAINDER

GridBagConstraints.SOUTH

GridBagConstraints.SOUTHEAST

GridBagConstraints.SOUTHWEST

GridBagConstraints.VERTICAL

GridBagConstraints.WEST

GridBagLayout layout manager

gridheight variable

gridwidth variable

gridx variable of **GridBagConstraints**

gridy variable of **GridBagConstraints**

isPopupTrigger method of **MouseEvent**

JCheckBoxMenuItem class

JDesktopPane class

JInternalFrame class

JMenu class

JMenuBar class

JMenuItem class

JPopupMenu class

JRadioButtonMenuItem class

JSlider class

JTextArea class

JTextComponent class

labels for tick marks	setSelected method of AbstractButton
last method of class CardLayout	setTitle method of class Frame
major tick mark	setVerticalScrollBarPolicy method
menu	show method of class JPopupMenu
menu bar	snap-to ticks
menu item	submenu
metal look-and-feel	super.paintComponent(g);
method setMnemonic of AbstractButton	SwingConstants.HORIZONTAL
minor tick mark	SwingConstants.VERTICAL
mnemonic	thumb of a JSlider
modal dialog	tick mark
modeless dialog	UIManager class
Motif look-and-feel	UIManager.LookAndFeelInfo class
multiple document interface (MDI)	updateComponentTreeUI method
next method of class CardLayout	vertical strut
paintComponent method of JComponent	weightx variable of GridBagConstraints
parent window	weighty variable of GridBagConstraints
pluggable look-and-feel (PLAF)	windowActivated method
previous method of class CardLayout	windowClosed method
scrollbar policies for a JScrollPane	windowClosing method
setConstraints method	WindowConstants.DISPOSE_ON_CLOSE
setDefaultCloseOperation method	windowDeactivated method
setHorizontalScrollBarPolicy method	windowDeiconified method
setJMenuBar method	windowIconified method
setLookAndFeel method	WindowListener interface
setMajorTickSpacing method	windowOpened method
setOpaque method of class JComponent	Windows look-and-feel
setPaintTicks method of class JSlider	

SELF-REVIEW EXERCISES

- 13.1** Fill in the blanks in each of the following statements:
- The _____ class is used to create a menu object.
 - The _____ method places a separator bar in a menu.
 - Passing **false** to a **TextArea**'s _____ method prevents its text from being modified by the user.
 - JSlider** events are handled by the _____ method of interface _____.
 - The **GridBagConstraints** instance variable _____ is set to **CENTER** by default.
- 13.2** State whether each of the following is *true* or *false*. If *false*, explain why.
- When the programmer creates a **JFrame**, a minimum of one menu must be created and added to the **JFrame**.
 - The variable **fill** belongs to the **GridBagLayout** class.
 - JFrames** and applets cannot be used together in the same program.
 - The top-left corner of a **JFrame** or applet has a coordinate of (0, 0).
 - A **JTextArea**'s text is always read-only.
 - Class **JTextArea** is a direct subclass of class **Component**.
 - The default layout for a **Box** is **BoxLayout**.
- 13.3** Find the error(s) in each of the following and explain how to correct the error(s).
- JMenubar b;**
 - mySlider = JSlider(1000, 222, 100, 450);**

```

c) gbc.fill = GridBagConstraints.NORTHWEST; // set fill
d) // override to paint on a customized Swing component
   public void paintcomponent( Graphics g )
   {
       g.drawString( "HELLO", 50, 50 );
   }
e) // create a JFrame and display it
   JFrame f = new JFrame( "A Window" );
   f.setVisible( true );

```

ANSWERS TO SELF-REVIEW EXERCISES

13.1 a) **JMenu**. b) **addSeparator**. c) **setEditable**. d) **stateChanged**, **ChangeListener**. e) **anchor**.

13.2 a) False. A **JFrame** does not require any menus.
 b) False. The variable **fill** belongs to the **GridBagConstraints** class.
 c) False. They can be used together.
 d) True.
 e) False. **JTextAreas** are editable by default.
 f) False. **JTextArea** derives from class **JTextComponent**.
 g) True.

13.3 a) **JMenuBar** should be **JMenuBar**.
 b) The first argument to the constructor should be either **SwingConstants.HORIZONTAL** or **SwingConstants.VERTICAL**, and the **new** operator must be used after the **=** operator.
 c) The constant should be either **BOTH**, **HORIZONTAL**, **VERTICAL** or **NONE**.
 d) **paintcomponent** should be **paintComponent** and the method should call **super.paintComponent(g)** as its first statement.
 e) The **JFrame**'s **setSize** method must also be called to determine the size of the window.

EXERCISES

13.4 Fill in the blanks in each of the following statements:
 a) A dedicated drawing area can be defined as a subclass of _____.
 b) A **JMenuItem** that is a **JMenu** is called a _____.
 c) Both **JTextField**s and **JTextArea**s inherit directly from class _____.
 d) The _____ method attaches a **JMenuBar** to a **JFrame**.
 e) Container class _____ has a default **BoxLayout**.
 f) A _____ manages a set of child windows defined with class **JInternalFrame**.

13.5 State whether each of the following is *true* or *false*. If *false*, explain why.
 a) Menus require a **JMenuBar** object so they can be attached to a **JFrame**.
 b) A **JPanel** object is capable of receiving mouse events.
 c) **CardLayout** is the default layout manager for a **JFrame**.
 d) Method **setEditable** is a **JTextComponent** method.
 e) The **GridBagLayout** layout manager implements **LayoutManager**.
 f) **JPanel** objects are containers to which other GUI components can be attached.
 g) Class **JFrame** inherits directly from class **Container**.
 h) **JApplets** can contain menus.

13.6 Find the error(s) in each of the following. Explain how to correct the error(s).
 a) `x.add( new JMenuItem( "Submenu Color" ) ); // create submenu`

- b) `container.setLayout( m = new GridbagLayout() );`
 c) `String s = JTextArea.getText();`

13.7 Write a program that displays a circle of random size and calculates and displays the area, radius, diameter and circumference. Use the following equations: $diameter = 2 \times radius$, $area = \pi \times radius^2$, $circumference = 2 \times \pi \times radius$. Use the constant `Math.PI` for pi (π). All drawing should be done on a subclass of `JPanel`, and the results of the calculations should be displayed in a read-only `JTextArea`.

13.8 Enhance the program of Exercise 13.7 by allowing the user to alter the radius with a `JSlider`. The program should work for all radii in the range from 100 to 200. As the radius changes, the diameter, area and circumference should be updated and displayed. The initial radius should be 150. Use the equations of Exercise 13.7. All drawing should be done on a subclass of `JPanel`, and the results of the calculations should be displayed in a read-only `JTextArea`.

13.9 Explore the effects of varying the `weightx` and `weighty` values of the program of Fig. 13.20. What happens when a component has a nonzero weight, but is not allowed to fill the whole area (i.e., the `fill` value is not `BOTH`)?

13.10 Write a program that uses the `paintComponent` method to draw the current value of a `JSlider` on a subclass of `JPanel`. In addition, provide a `JTextField` where a specific value can be entered. The `JTextField` should display the current value of the `JSlider` at all times. A `JLabel` should be used to identify the `JTextField`. The `JSlider` methods `setValue` and `getValue` should be used. [Note: The `setValue` method is a `public` method that does not return a value and takes one integer argument—the `JSlider` value, which determines the position of the thumb.]

13.11 Modify the program of Fig. 13.16 to use a single `JComboBox` instead of the four separate `JButtons`. Each “card” should not be modified.

13.12 Modify the program of Fig. 13.16 by adding a minimum of two new “cards” to the deck.

13.13 Define a subclass of `JPanel` called `MyColorChooser` that provides three `JSlider` objects and three `JTextField` objects. Each `JSlider` represents the values from 0 to 255 for the red, green and blue parts of a color. Use the red, green and blue values as the arguments to the `Color` constructor to create a new `Color` object. Display the current value of each `JSlider` in the corresponding `JTextField`. When the user changes the value of the `JSlider`, the `JTextField` should be changed accordingly. Define class `MyColorChooser` so it can be reused in other applications or applets. Use your new GUI component as part of an applet that displays the current `Color` value by drawing a filled rectangle.

13.14 Modify the `MyColorChooser` class of Exercise 13.13 to allow the user to type an integer value into a `JTextField` to set the red, green or blue value. When the user presses *Enter* in the `JTextField`, the corresponding `JSlider` should be set to the appropriate value.

13.15 Modify the applet of Exercise 13.14 to draw the current color as a rectangle on an instance of a subclass of `JPanel` called `DrawPanel`. Class `DrawPanel` should provide its own `paintComponent` method to draw the rectangle and should provide *set* methods to set the red, green and blue values for the current color. When any *set* method is invoked for the class `DrawPanel`, the object should automatically *repaint* itself.

13.16 Modify the applet of Exercise 13.15 to allow the user to drag the mouse across the `DrawPanel` to draw a shape in the current color. Enable the user to choose what shape to draw.

13.17 Modify the program of Exercise 13.16 to enable the program to run as an application. The existing applet’s code should be modified only by adding a `main` method to launch the application in its own `JFrame`. Provide the user with the ability to terminate the application by clicking the close

box on the window that is displayed and by selecting **Exit** from a **File** menu. Use the techniques shown in Fig. 13.9.

13.18 (*Complete Drawing Application*) Using the techniques developed in Exercise 12.27–Exercise 12.33 and Exercise 13.13–Exercise 13.17, create a complete drawing program that can execute as both an applet and an application. The program should use the GUI components of Chapter 12 and Chapter 13 to enable the user to select the shape, color and fill characteristics. Each shape should be stored in an array of **MyShape** objects, where **MyShape** is the superclass in your hierarchy of shape classes (see Exercise 9.28 and Exercise 9.29). Use a **JDesktopPane** and **JInternalFrames** to allow the user to create multiple separate drawings in separate child windows. Create the user interface as a separate child window containing all the GUI components that allow the user to determine the characteristics of the shape to be drawn. The user can then click in any **JInternalFrame** to draw the shape.

13.19 A company pays its employees as managers (who receive a fixed weekly salary), hourly workers (who receive a fixed hourly wage for up to the first 40 hours they work and “time-and-a-half,” i.e., 1.5 times their hourly wage, for overtime hours worked), commission workers (who receive \$250 plus 5.7% of their gross weekly sales) or pieceworkers (who receive a fixed amount of money per item for each of the items they produce—each pieceworker in this company works on only one type of item). Write an application to compute the weekly pay for each employee. Each type of employee has its own pay code: Managers have paycode 1, hourly workers have code 2, commission workers have code 3 and pieceworkers have code 4. Use a **switch** to compute each employee’s pay based on that employee’s paycode. Use a **CardLayout** to display the appropriate GUI components that allow the user to enter the facts your program needs to calculate each employee’s pay based on that employee’s paycode.