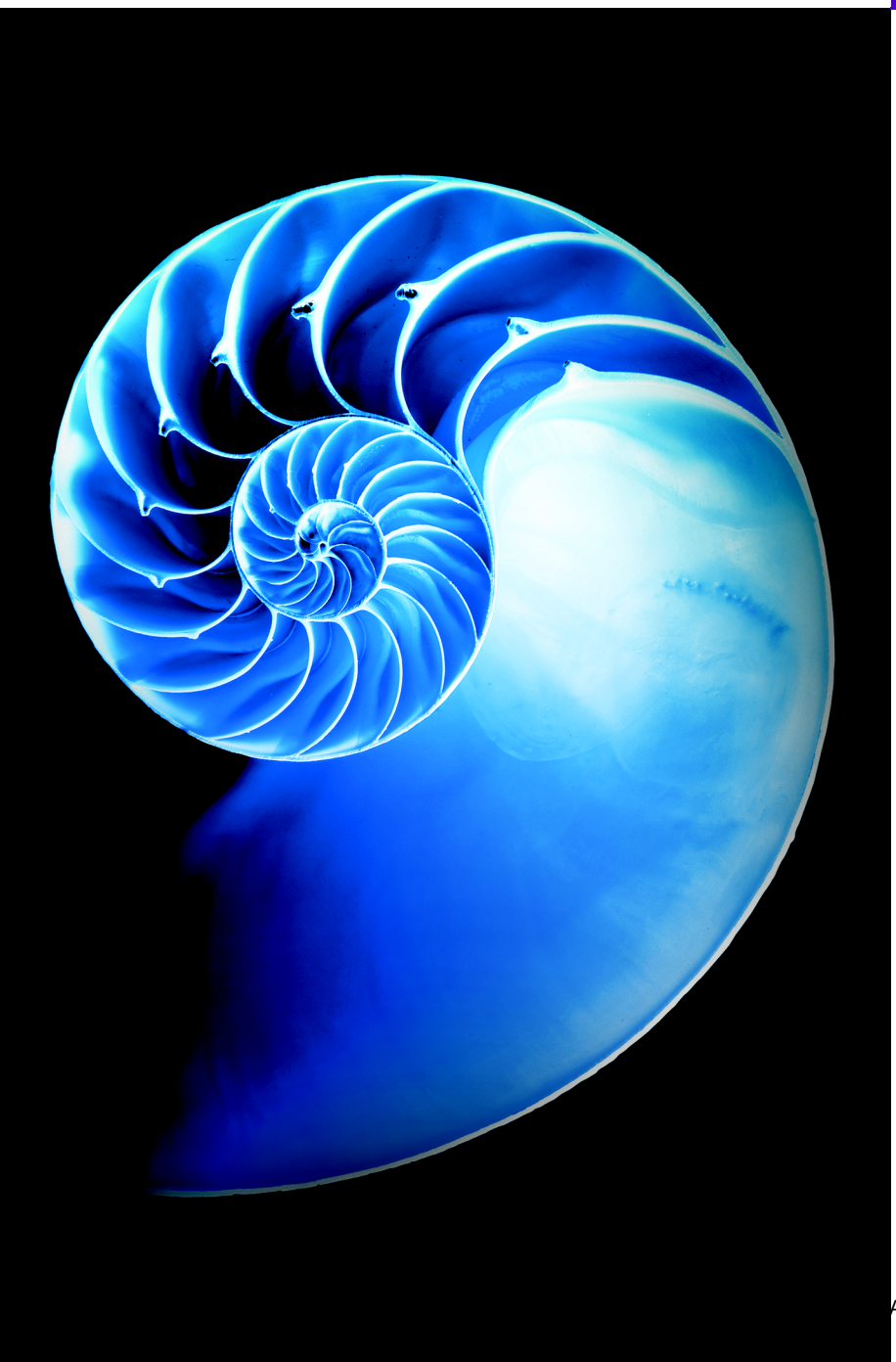
Swing GUI Components: Part 2

35



Objectives

In this chapter you'll:

- Create and manipulate sliders, menus, pop-up menus and windows.
- Programatically change the look-and-feel of a GUI, using Swing's pluggable look-and-feel.
- Create a multiple-document interface with `JDesktopPane` and `JInternalFrame`.
- Use additional layout managers `BoxLayout` and `GridBagLayout`.

Outline

35.1 Introduction

[*Note:* JavaFX (Chapters 12, 13 and 22) is Java's GUI, graphics and multimedia API of the future. This chapter is provided *as is* for those still interested in Swing GUIs.]

In this chapter, we continue our study of Swing GUIs. We discuss additional components and layout managers and lay the groundwork for building more complex GUIs. We begin with sliders for selecting from a range of integer values, then discuss additional details of windows. Next, you'll use menus to organize an application's commands.

The look-and-feel of a Swing GUI can be uniform across all platforms on which a Java program executes, or the GUI can be customized by using Swing's **pluggable look-and-feel** (**PLAF**). We provide an example that illustrates how to change between Swing's default metal look-and-feel (which looks and behaves the same across platforms), the Nimbus look-and-feel (introduced in Chapter 26), a look-and-feel that simulates **Motif** (a UNIX look-and-feel) and one that simulates the Microsoft Windows look-and-feel.

Many of today's applications use a multiple-document interface (MDI)—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 that you can compose or read multiple e-mail messages. We demonstrate Swing's classes for creating multiple-document interfaces. Finally, you'll learn about additional layout managers for organizing graphical user interfaces. We use several more Swing GUI components in later chapters as they're needed.

Swing is now considered a legacy technology. For GUIs, graphics and multimedia in new Java apps, you should use the features presented in this book's JavaFX chapters.

8 *Java SE 8: Implementing Event Listeners with Lambdas*

Throughout this chapter, we use anonymous inner classes and nested classes to implement event handlers so that the examples can compile and execute with both Java SE 7 and Java SE 8. In many of the examples, you could implement the functional event-listener interfaces with Java SE 8 lambdas (as demonstrated in Section 17.16).

35.2 JSlider

JSliders enable a user to select from a range of integer values. Class `JSlider` inherits from `JComponent`. Figure 35.1 shows a horizontal `JSlider` with **tick marks** and the **thumb** that allows a user to select a value. `JSliders` can be customized to display *major tick marks*,

minor tick marks and labels for the tick marks. They also support **snap-to ticks**, which cause the *thumb*, when positioned between two tick marks, to snap to the closest one.

Fig. 35.1 | `JSlider` component with horizontal orientation.

Most Swing GUI components support mouse and keyboard interactions—e.g., if a `JSlider` has the focus (i.e., it's the currently selected GUI component in the user interface), pressing the left arrow key or right arrow key causes the `JSlider`'s thumb to decrease or increase by 1, respectively. The down arrow key and up arrow key also cause the thumb to decrease or increase by 1 tick, respectively. The *PgDn* (page down) *key* and *PgUp* (page up) *key* cause the thumb 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`.

`JSlider`s have either a horizontal or a vertical orientation. For a horizontal `JSlider`, the minimum value is at the left end and the maximum is at the right end. For a vertical `JSlider`, the minimum value is at the bottom and the maximum is at the top. The minimum and maximum value positions on a `JSlider` can be reversed by invoking `JSlider` method **`setInverted`** with `boolean` argument `true`. The relative position of the thumb indicates the current value of the `JSlider`.

The program in Figs. 35.2–35.4 allows the user to size a circle drawn on a subclass of `JPanel` called `OvalPanel` (Fig. 35.2). The user specifies the circle's diameter with a horizontal `JSlider`. Class `OvalPanel` 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.

```
1  // Fig. 22.2: OvalPanel.java
2  // A customized JPanel class.
3  import java.awt.Graphics;
4  import java.awt.Dimension;
5  import javax.swing.JPanel;
6
7  public class OvalPanel extends JPanel
8  {
9     private int diameter = 10; // default diameter
10
11    // draw an oval of the specified diameter
12    @Override
13    public void paintComponent(Graphics g)
14    {
15       super.paintComponent(g);
```

Fig. 35.2 | `JPanel` subclass for drawing circles of a specified diameter. (Part 1 of 2.)

```
16          g.fillOval(10, 10, diameter, diameter);
17       }
18
19       // validate and set diameter, then repaint
20       public void setDiameter(int newDiameter)
21       {
22          // if diameter invalid, default to 10
23          diameter = (newDiameter >= 0 ? newDiameter : 10);
24          repaint(); // repaint panel
25       }
26
27       // used by layout manager to determine preferred size
28       public Dimension getPreferredSize()
29       {
30          return new Dimension(200, 200);
31       }
32
33       // used by layout manager to determine minimum size
34       public Dimension getMinimumSize()
35       {
36          return getPreferredSize();
37       }
38    } // end class OvalPanel
```

Fig. 35.2 | `JPanel` subclass for drawing circles of a specified diameter. (Part 2 of 2.)

```
1  // Fig. 22.3: SliderFrame.java
2  // Using JSliders to size an oval.
3  import java.awt.BorderLayout;
4  import java.awt.Color;
5  import javax.swing.JFrame;
6  import javax.swing.JSlider;
7  import javax.swing.SwingConstants;
8  import javax.swing.event.ChangeListener;
9  import javax.swing.event.ChangeEvent;
10
11 public class SliderFrame extends JFrame
12 {
13    private final JSlider diameterJSlider; // slider to select diameter
14    private final OvalPanel myPanel; // panel to draw circle
15
16    // no-argument constructor
17    public SliderFrame()
18    {
19       super("Slider Demo");
20
21       myPanel = new OvalPanel(); // create panel to draw circle
22       myPanel.setBackground(Color.YELLOW);
23
24       // set up JSlider to control diameter value
25       diameterJSlider =
26          new JSlider(SwingConstants.HORIZONTAL, 0, 200, 10);
```

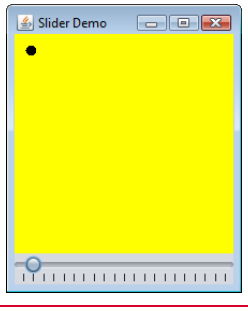
Fig. 35.3 | `JSlider` value used to determine the diameter of a circle. (Part 1 of 2.)

```
27          diameterJSlider.setMajorTickSpacing(10); // create tick every 10
28          diameterJSlider.setPaintTicks(true); // paint ticks on slider
29
30          // register JSlider event listener
31          diameterJSlider.addChangeListener(
32             new ChangeListener() // anonymous inner class
33             {
34                // handle change in slider value
35                @Override
36                public void stateChanged(ChangeEvent e)
37                {
38                   myPanel.setDiameter(diameterJSlider.getValue());
39                }
40             }
41          );
42
43          add(diameterJSlider, BorderLayout.SOUTH);
44          add(myPanel, BorderLayout.CENTER);
45       }
46    } // end class SliderFrame
```

Fig. 35.3 | `JSlider` value used to determine the diameter of a circle. (Part 2 of 2.)

```
1  // Fig. 22.4: SliderDemo.java
2  // Testing SliderFrame.
3  import javax.swing.JFrame;
4
5  public class SliderDemo
6  {
7     public static void main(String[] args)
8     {
9        SliderFrame sliderFrame = new SliderFrame();
10       sliderFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       sliderFrame.setSize(220, 270);
12       sliderFrame.setVisible(true);
13    }
14 } // end class SliderDemo
```

a) Initial GUI with default circle



b) GUI after the user moves the `JSlider`'s thumb to the right

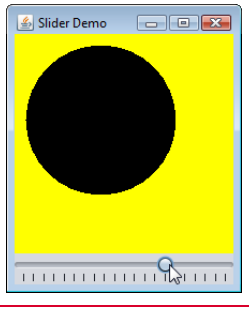


Fig. 35.4 | Test class for `SliderFrame`.

Class `OvalPanel` (Fig. 35.2) contains a `paintComponent` method (lines 12–17) that draws a filled oval (a circle in this example), a `setDiameter` method (lines 20–25) that

changes the circle's `diameter` and `repaints` the `OvalPanel`, a `getPreferredSize` method (lines 28–31) that returns the preferred width and height of an `OvalPanel` and a `getMinimumSize` method (lines 34–37) that returns an `OvalPanel`'s minimum width and height. Methods `getPreferredSize` and `getMinimumSize` are used by some layout managers to determine the size of a component.

Class `SliderFrame` (Fig. 35.3) creates the `JSlider` that controls the diameter of the circle. Class `SliderFrame`'s constructor (lines 17–45) creates `OvalPanel` object `myPanel` (line 21) and sets its background color (line 22). Lines 25–26 create `JSlider` object `diameterJSlider` to control the diameter of the circle drawn on the `OvalPanel`. The `JSlider` constructor takes four arguments. The first specifies the orientation of `diameterJSlider`, which is `HORIZONTAL` (a constant in interface `SwingConstants`). The second and third arguments indicate the minimum and maximum integer values in the range of values for this `JSlider`. The last argument indicates that the initial value of the `JSlider` (i.e., where the thumb is displayed) should be `10`.

Lines 27–28 customize the appearance of the `JSlider`. Method **`setMajorTickSpacing`** indicates that each major 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 aren't displayed by default). For other methods that are used to customize a `JSlider`'s appearance, see the `JSlider` online documentation (`docs.oracle.com/javase/8/docs/api/javax/swing/JSlider.html`).

`JSliders` generate **`ChangeEvents`** (package `javax.swing.event`) in response to user interactions. An object of a class that implements interface **`ChangeListener`** (package `javax.swing.event`) and declares method **`stateChanged`** can respond to `ChangeEvents`. Lines 31–41 register a `ChangeListener` to handle `diameterJSlider`'s events. When method `stateChanged` (lines 35–39) is called in response to a user interaction, line 38 calls `myPanel`'s `setDiameter` method and passes the current value of the `JSlider` as an argument. `JSlider` method **`getValue`** returns the current thumb position.

35.3 Understanding Windows in Java

A `JFrame` is a **window** with a **title bar** and a **border**. Class `JFrame` is a subclass of `Frame` (package `java.awt`), which is a subclass of `Window` (package `java.awt`). As such, `JFrame` is one of the *heavyweight* Swing GUI components. When you display a window from a Java program, the window is provided by the local platform's windowing toolkit, and therefore the window will look like every other window displayed on that platform. When a Java application executes on a Macintosh and displays a window, the window's title bar and borders will look like those of other Macintosh applications. When a Java application executes on a Microsoft Windows system and displays a window, the window's title bar and borders will look like those of other Microsoft Windows applications. And when a Java application executes on a UNIX platform and displays a window, the window's title bar and borders will look like those of other UNIX applications on that platform.

Returning Window Resources to the System

By default, when the user closes a `JFrame` window, it's hidden (i.e., removed from the screen), but you can control this with `JFrame` method **`setDefaultCloseOperation`**. Interface **`WindowConstants`** (package `javax.swing`), which class `JFrame` implements, declares three constants—`DISPOSE_ON_CLOSE`, `DO_NOTHING_ON_CLOSE` and `HIDE_ON_CLOSE` (the de-

fault)—for use with this method. Some platforms allow only a limited number of windows to be displayed on the screen. Thus, a window is a valuable resource that should be given back to the system when it's no longer needed. Class `Window` (an indirect superclass of `JFrame`) declares method **`dispose`** for this purpose. When a `Window` is no longer needed in an application, you should explicitly dispose of it. This can be done by calling the `Window`'s `dispose` method or by calling method `setDefaultCloseOperation` with the argument `WindowConstants.DISPOSE_ON_CLOSE`. Terminating an application also returns window resources to the system. Using `DO_NOTHING_ON_CLOSE` indicates that the program will determine what to do when the user attempts to close the window. For example, the program might want to ask whether to save a file's changes before closing a window.

Displaying and Positioning Windows

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. 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`).

Window Events

When the user manipulates the window, this action generates **window events**. Event listeners are registered for window events with `Window` method **`addWindowListener`**. The **`WindowListener`** interface provides seven window-event-handling methods—**`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), **`windowDeiconified`** (called when the user restores a minimized window), **`windowIconified`** (called when the user minimizes a window) and **`windowOpened`** (called when a program first displays a window on the screen).

35.4 Using Menus with Frames

Menus are an integral part of GUIs. They allow the user to perform actions without unnecessarily cluttering a GUI with extra 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 declare menus are `JMenuBar`, `JMenu`, `JMenuItem`, `JCheckBoxMenuItem` and class `JRadioButtonMenuItem`.

Look-and-Feel Observation 35.1

Menus simplify GUIs because components can be hidden within them. These components will be visible only when the user looks for them by selecting the menu.

Overview of Several Menu-Related Components

Class **`JMenuBar`** (a subclass of `JComponent`) contains the methods necessary to manage a **menu bar**, which is a container for menus. 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, it expands to show its list of menu items.

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 event. 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 `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 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 `JCheckBoxMenuItems`. 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 of the previously selected menu item is removed.

Using Menus in an Application

Figures 35.5–35.6 demonstrate various menu items and 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 all subclasses of `javax.swing.AbstractButton`. Class `MenuFrame` (Fig. 35.5) creates the GUI and handles the menu-item events. Most of the code in this application appears in the class's constructor (lines 34–151).

```
1  // Fig. 22.5: MenuFrame.java
2  // Demonstrating menus.
3  import java.awt.Color;
4  import java.awt.Font;
5  import java.awt.BorderLayout;
6  import java.awt.event.ActionListener;
7  import java.awt.event.ActionEvent;
8  import java.awt.event.ItemListener;
9  import java.awt.event.ItemEvent;
10 import javax.swing.JFrame;
11 import javax.swing.JRadioButtonMenuItem;
12 import javax.swing.JCheckBoxMenuItem;
13 import javax.swing.JOptionPane;
14 import javax.swing.JLabel;
15 import javax.swing.SwingConstants;
16 import javax.swing.ButtonGroup;
17 import javax.swing.JMenu;
18 import javax.swing.JMenuItem;
19 import javax.swing.JMenuBar;
20
21 public class MenuFrame extends JFrame
22 {
23    private final Color[] colorValues =
24       {Color.BLACK, Color.BLUE, Color.RED, Color.GREEN};
```

Fig. 35.5 | `JMenus` and mnemonics. (Part 1 of 5.)

```
25      private final JRadioButtonMenuItem[] colorItems; // color menu items
26      private final JRadioButtonMenuItem[] fonts; // font menu items
27      private final JCheckBoxMenuItem[] styleItems; // font style menu items
28      private final JLabel displayJLabel; // displays sample text
29      private final ButtonGroup fontButtonGroup; // manages font menu items
30      private final ButtonGroup colorButtonGroup; // manages color menu items
31      private int style; // used to create style for font
32
33      // no-argument constructor set up GUI
34      public MenuFrame()
35      {
36         super("Using JMenus");
37
38         JMenu fileMenu = new JMenu("File"); // create file menu
39         fileMenu.setMnemonic('F'); // set mnemonic to F
40
41         // create About... menu item
42         JMenuItem aboutItem = new JMenuItem("About...");
43         aboutItem.setMnemonic('A'); // set mnemonic to A
44         fileMenu.add(aboutItem); // add about item to file menu
45         aboutItem.addActionListener(
46            new ActionListener() // anonymous inner class
47            {
48               // display message dialog when user selects About...
49               @Override
50               public void actionPerformed(ActionEvent event)
51               {
52                  JOptionPane.showMessageDialog(MenuFrame.this,
53                     "This is an example\nof using menus",
54                     "About", JOptionPane.PLAIN_MESSAGE);
55               }
56            }
57         );
58
59         JMenuItem exitItem = new JMenuItem("Exit"); // create exit item
60         exitItem.setMnemonic('x'); // set mnemonic to x
61         fileMenu.add(exitItem); // add exit item to file menu
62         exitItem.addActionListener(
63            new ActionListener() // anonymous inner class
64            {
65               // terminate application when user clicks exitItem
66               @Override
67               public void actionPerformed(ActionEvent event)
68               {
69                  System.exit(0); // exit application
70               }
71            }
72         );
73
74         JMenuBar bar = new JMenuBar(); // create menu bar
75         setJMenuBar(bar); // add menu bar to application
76         bar.add(fileMenu); // add file menu to menu bar
77
```

Fig. 35.5 | `JMenus` and mnemonics. (Part 2 of 5.)

```
78          JMenu formatMenu = new JMenu("Format"); // create format menu
79          formatMenu.setMnemonic('r'); // set mnemonic to r
80
81          // array listing string colors
82          String[] colors = { "Black", "Blue", "Red", "Green" };
83
84          JMenu colorMenu = new JMenu("Color"); // create color menu
85          colorMenu.setMnemonic('C'); // set mnemonic to C
86
87          // create radio button menu items for colors
88          colorItems = new JRadioButtonMenuItem[colors.length];
89          colorButtonGroup = new ButtonGroup(); // manages colors
90          ItemHandler itemHandler = new ItemHandler(); // handler for colors
91
92          // create color radio button menu items
93          for (int count = 0; count < colors.length; count++)
94          {
95             colorItems[count] =
96                new JRadioButtonMenuItem(colors[count]); // create item
97             colorMenu.add(colorItems[count]); // add item to color menu
98             colorButtonGroup.add(colorItems[count]); // add to group
99             colorItems[count].addActionListener(itemHandler);
100         }
101
102         colorItems[0].setSelected(true); // select first Color item
103
104         formatMenu.add(colorMenu); // add color menu to format menu
105         formatMenu.addSeparator(); // add separator in menu
106
107         // array listing font names
108         String[] fontNames = { "Serif", "Monospaced", "SansSerif" };
109         JMenu fontMenu = new JMenu("Font"); // create font menu
110         fontMenu.setMnemonic('n'); // set mnemonic to n
111
112         // create radio button menu items for font names
113         fonts = new JRadioButtonMenuItem[fontNames.length];
114         fontButtonGroup = new ButtonGroup(); // manages font names
115
116         // create Font radio button menu items
117         for (int count = 0; count < fonts.length; count++)
118         {
119            fonts[count] = new JRadioButtonMenuItem(fontNames[count]);
120            fontMenu.add(fonts[count]); // add font to font menu
121            fontButtonGroup.add(fonts[count]); // add to button group
122            fonts[count].addActionListener(itemHandler); // add handler
123         }
124
125         fonts[0].setSelected(true); // select first Font menu item
126         fontMenu.addSeparator(); // add separator bar to font menu
127
128         String[] styleNames = { "Bold", "Italic" }; // names of styles
129         styleItems = new JCheckBoxMenuItem[styleNames.length];
130         StyleHandler styleHandler = new StyleHandler(); // style handler
```

Fig. 35.5 | `JMenus` and mnemonics. (Part 3 of 5.)

```
131
132        // create style checkbox menu items
133        for (int count = 0; count < styleNames.length; count++)
134        {
135           styleItems[count] =
136              new JCheckBoxMenuItem(styleNames[count]); // for style
137           fontMenu.add(styleItems[count]); // add to font menu
138           styleItems[count].addItemListener(styleHandler); // handler
139        }
140
141        formatMenu.add(fontMenu); // add Font menu to Format menu
142        bar.add(formatMenu); // add Format menu to menu bar
143
144        // set up label to display text
145        displayJLabel = new JLabel("Sample Text", SwingConstants.CENTER);
146        displayJLabel.setForeground(colorValues[0]);
147        displayJLabel.setFont(new Font("Serif", Font.PLAIN, 72));
148
149        getContentPane().setBackground(Color.CYAN); // set background
150        add(displayJLabel, BorderLayout.CENTER); // add displayJLabel
151     } // end MenuFrame constructor
152
153     // inner class to handle action events from menu items
154     private class ItemHandler implements ActionListener
155     {
156        // process color and font selections
157        @Override
158        public void actionPerformed(ActionEvent event)
159        {
160           // process color selection
161           for (int count = 0; count < colorItems.length; count++)
162           {
163              if (colorItems[count].isSelected())
164              {
165                 displayJLabel.setForeground(colorValues[count]);
166                 break;
167              }
168           }
169
170           // process font selection
171           for (int count = 0; count < fonts.length; count++)
172           {
173              if (event.getSource() == fonts[count])
174              {
175                 displayJLabel.setFont(
176                    new Font(fonts[count].getText(), style, 72));
177              }
178           }
179
180           repaint(); // redraw application
181        }
182     } // end class ItemHandler
183
```

Fig. 35.5 | `JMenus` and mnemonics. (Part 4 of 5.)

```
184    // inner class to handle item events from checkbox menu items
185    private class StyleHandler implements ItemListener
186    {
187       // process font style selections
188       @Override
189       public void itemStateChanged(ItemEvent e)
190       {
191          String name = displayJLabel.getFont().getName(); // current Font
192          Font font; // new font based on user selections
193
194          // determine which items are checked and create Font
195          if (styleItems[0].isSelected() &&
196                styleItems[1].isSelected())
197             font = new Font(name, Font.BOLD + Font.ITALIC, 72);
198          else if (styleItems[0].isSelected())
199             font = new Font(name, Font.BOLD, 72);
200          else if (styleItems[1].isSelected())
201             font = new Font(name, Font.ITALIC, 72);
202          else
203             font = new Font(name, Font.PLAIN, 72);
204
205          displayJLabel.setFont(font);
206          repaint(); // redraw application
207       }
208    }
209 } // end class MenuFrame
```

Fig. 35.5 | `JMenus` and mnemonics. (Part 5 of 5.)

```
1  // Fig. 22.6: MenuTest.java
2  // Testing MenuFrame.
3  import javax.swing.JFrame;
4
5  public class MenuTest
6  {
7     public static void main(String[] args)
8     {
9        MenuFrame menuFrame = new MenuFrame();
10       menuFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       menuFrame.setSize(500, 200);
12       menuFrame.setVisible(true);
13    }
14 } // end class MenuTest
```

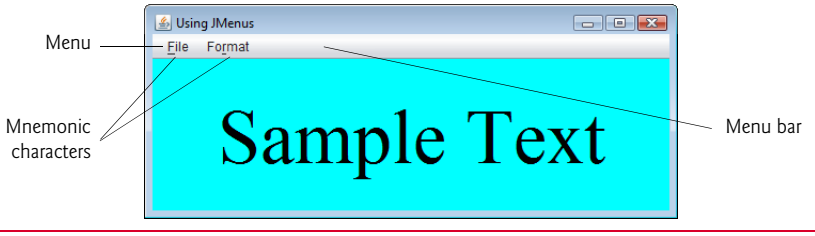


Fig. 35.6 | Test class for `MenuFrame`. (Part 1 of 2.)

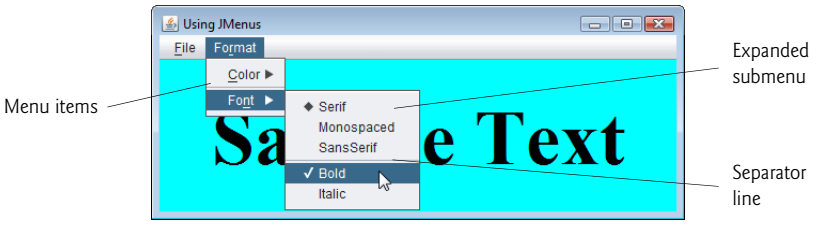


Fig. 35.6 | Test class for `MenuFrame`. (Part 2 of 2.)

Setting Up the File *Menu*

Lines 38–76 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 38 creates a `JMenu` and passes to the constructor the string `"File"` as the name of the menu. Line 39 uses `JMenu` method **`setMnemonic`** (inherited from class `AbstractButton`) 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 in Fig. 35.6.)

Look-and-Feel Observation 35.2

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

Look-and-Feel Observation 35.3

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 several 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 an* ***Exit*** *button or menu item). Mnemonics are case insensitive.*

Lines 42–43 create `JMenuItem` `aboutItem` with the text "`About...`" and set its mnemonic to the letter `A`. This menu item is added to `fileMenu` at line 44 with `JMenu` method **`add`**. To access the **About...** menu 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 46–56 create an `ActionListener` to process `aboutItem`'s action event. Lines 52–54 display a message dialog box. In most prior uses of `showMessageDialog`, the first argument was `null`. The purpose of the first argument is to specify the **parent window** that 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. Otherwise, it appears centered over the specified parent window. In this example, the program specifies the parent window with `MenuFrame.this`—the `this` reference of the `MenuFrame` object. 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 (`.`).

Recall that dialog boxes are typically modal. A modal dialog box does not allow any other window in the application to be accessed until the dialog box is dismissed. The dialogs displayed with class `JOptionPane` are modal dialogs. Class **`JDialog`** can be used to create your own modal or nonmodal dialogs.

Lines 59–72 create menu item `exitItem`, set its mnemonic to `x`, add it to `fileMenu` and register an `ActionListener` that terminates the program when the user selects `exitItem`. Lines 74–76 create the `JMenuBar`, attach it to the window with `JFrame` method `setJMenuBar` and use `JMenuBar` method **`add`** to attach the `fileMenu` to the `JMenuBar`.

Look-and-Feel Observation 35.4

Menus appear left to right in the order they're added to a `JMenuBar`.

Setting Up the Format *Menu*

Lines 78–79 create the `formatMenu` and set its mnemonic to `r`. `F` is not used because that's the **File** menu's mnemonic. Lines 84–85 create `colorMenu` (this will be a submenu in **Format**) and set its mnemonic to `C`. Line 88 creates `JRadioButtonMenuItem` array `colorItems`, which refers to the menu items in `colorMenu`. Line 89 creates `ButtonGroup colorButtonGroup`, which ensures that only one of the **Color** submenu items is selected at a time. Line 90 creates an instance of inner class `ItemHandler` (declared at lines 154–181) that responds to selections from the **Color** and **Font** submenus (discussed shortly). The loop at lines 93–100 creates each `JRadioButtonMenuItem` in array `colorItems`, adds each menu item to `colorMenu` and to `colorButtonGroup` and registers the `ActionListener` for each menu item.

Line 102 invokes `AbstractButton` method **`setSelected`** to select the first element in array `colorItems`. Line 104 adds `colorMenu` as a submenu of `formatMenu`. Line 105 invokes `JMenu` method **`addSeparator`** to add a horizontal **separator** line to the menu.

Look-and-Feel Observation 35.5

A submenu is created by adding a menu as a menu item in another menu.

Look-and-Feel Observation 35.6

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

Look-and-Feel Observation 35.7

Any `JComponent` can be added to a `JMenu` or to a `JMenuBar`.

Lines 108–126 create the **Font** submenu and several `JRadioButtonMenuItems` and select the first element of `JRadioButtonMenuItem` array `fonts`. Line 129 creates a `JCheckBoxMenuItem` array to represent the menu items for specifying bold and italic styles for the fonts. Line 130 creates an instance of inner class `StyleHandler` (declared at lines 185–208) to respond to the `JCheckBoxMenuItem` events. The `for` statement at lines 133–139 creates each `JCheckBoxMenuItem`, adds it to `fontMenu` and registers its `ItemListener`. Line 141 adds `fontMenu` as a submenu of `formatMenu`. Line 142 adds the `formatMenu` to `bar` (the menu bar).

Creating the Rest of the GUI and Defining the Event Handlers

Lines 145–147 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`) by invoking `JComponent` method **`setForeground`**. The initial font is set to `Serif` with `PLAIN` style and `72`-point size. Line 149 sets the background color of

the window's content pane to cyan, and line 150 attaches the `JLabel` to the `CENTER` of the content pane's `BorderLayout`.

`ItemHandler` method `actionPerformed` (lines 157–181) uses two `for` statements to determine which font or color menu item generated the event and sets the font or color of the `JLabel displayLabel`, respectively. The `if` condition at line 163 uses `AbstractButton` method **`isSelected`** to determine the selected `JRadioButtonMenuItem`. The `if` condition at line 173 invokes the event object's `getSource` method to get a reference to the `JRadioButtonMenuItem` that generated the event. Line 176 invokes `AbstractButton` method `getText` to obtain the name of the font from the menu item.

`StyleHandler` method `itemStateChanged` (lines 188–207) is called if the user selects a `JCheckBoxMenuItem` in the `fontMenu`. Lines 195–203 determine which `JCheckBoxMenuItems` are selected and use their combined state to determine the new font style.

35.5 JPopupMenu

Applications often provide **context-sensitive pop-up menus** for several reasons—they can be convenient, there might not be a menu bar and the options they display can be specific to individual on-screen components. 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** occurred—on most systems, when the user presses and releases the right mouse button.

Look-and-Feel Observation 35.8

The pop-up trigger event is platform specific. On most platforms that use a mouse with multiple buttons, the pop-up trigger event occurs when the user clicks the right mouse button on a component that supports a pop-up menu.

The application in Figs. 35.7–35.8 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 `PopupFrame` window's background, a `JPopupMenu` containing colors appears. If the user clicks a `JRadioButtonMenuItem` for a color, `ItemHandler` method `actionPerformed` changes the background color of the window's content pane.

Line 25 of the `PopupFrame` constructor (Fig. 35.7, lines 21–70) creates an instance of class `ItemHandler` (declared in lines 73–89) that will process the item events from the menu items in the pop-up menu. Line 29 creates the `JPopupMenu`. The `for` statement (lines 33–39) creates a `JRadioButtonMenuItem` object (line 35), adds it to `popupMenu` (line 36), adds it to `ButtonGroup colorGroup` (line 37) to maintain one selected `JRadioButtonMenuItem` at a time and registers its `ActionListener` (line 38). Line 41 sets the initial background to white by invoking method `setBackground`.

```
1  // Fig. 22.7: PopupFrame.java
2  // Demonstrating JPopupMenus.
3  import java.awt.Color;
4  import java.awt.event.MouseAdapter;
5  import java.awt.event.MouseEvent;
6  import java.awt.event.ActionListener;
```

Fig. 35.7 | `JPopupMenu` for selecting colors. (Part 1 of 3.)

```
7  import java.awt.event.ActionEvent;
8  import javax.swing.JFrame;
9  import javax.swing.JRadioButtonMenuItem;
10 import javax.swing.JPopupMenu;
11 import javax.swing.ButtonGroup;
12
13 public class PopupFrame extends JFrame
14 {
15    private final JRadioButtonMenuItem[] items; // holds items for colors
16    private final Color[] colorValues =
17       { Color.BLUE, Color.YELLOW, Color.RED }; // colors to be used
18    private final JPopupMenu popupMenu; // allows user to select color
19
20    // no-argument constructor sets up GUI
21    public PopupFrame()
22    {
23       super("Using JPopupMenus");
24
25       ItemHandler handler = new ItemHandler(); // handler for menu items
26       String[] colors = { "Blue", "Yellow", "Red" };
27
28       ButtonGroup colorGroup = new ButtonGroup(); // manages color items
29       popupMenu = new JPopupMenu(); // create pop-up menu
30       items = new JRadioButtonMenuItem[colors.length];
31
32       // construct menu item, add to pop-up menu, enable event handling
33       for (int count = 0; count < items.length; count++)
34       {
35          items[count] = new JRadioButtonMenuItem(colors[count]);
36          popupMenu.add(items[count]); // add item to pop-up menu
37          colorGroup.add(items[count]); // add item to button group
38          items[count].addActionListener(handler); // add handler
39       }
40
41       setBackground(Color.WHITE);
42
43       // declare a MouseListener for the window to display pop-up menu
44       addMouseListener(
45          new MouseAdapter() // anonymous inner class
46          {
47             // handle mouse press event
48             @Override
49             public void mousePressed(MouseEvent event)
50             {
51                checkForTriggerEvent(event);
52             }
53
54             // handle mouse release event
55             @Override
56             public void mouseReleased(MouseEvent event)
57             {
58                checkForTriggerEvent(event);
59             }
```

Fig. 35.7 | `JPopupMenu` for selecting colors. (Part 2 of 3.)

```
60
61             // determine whether event should trigger pop-up menu
62             private void checkForTriggerEvent(MouseEvent event)
63             {
64                if (event.isPopupTrigger())
65                   popupMenu.show(
66                      event.getComponent(), event.getX(), event.getY());
67             }
68          }
69       );
70    } // end PopupFrame constructor
71
72    // private inner class to handle menu item events
73    private class ItemHandler implements ActionListener
74    {
75       // process menu item selections
76       @Override
77       public void actionPerformed(ActionEvent event)
78       {
79          // determine which menu item was selected
80          for (int i = 0; i < items.length; i++)
81          {
82             if (event.getSource() == items[i])
83             {
84                getContentPane().setBackground(colorValues[i]);
85                return;
86             }
87          }
88       }
89    } // end private inner class ItemHandler
90 } // end class PopupFrame
```

Fig. 35.7 | `JPopupMenu` for selecting colors. (Part 3 of 3.)

```
1  // Fig. 22.8: PopupTest.java
2  // Testing PopupFrame.
3  import javax.swing.JFrame;
4
5  public class PopupTest
6  {
7     public static void main(String[] args)
8     {
9        PopupFrame popupFrame = new PopupFrame();
10       popupFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       popupFrame.setSize(300, 200);
12       popupFrame.setVisible(true);
13    }
14 } // end class PopupTest
```

Fig. 35.8 | Test class for `PopupFrame`. (Part 1 of 2.)

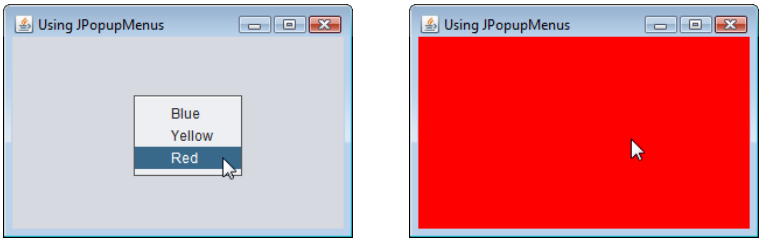


Fig. 35.8 | Test class for `PopupFrame`. (Part 2 of 2.)

Lines 44–69 register a `MouseListener` to handle the mouse events of the application window. Methods `mousePressed` (lines 48–52) and `mouseReleased` (lines 55–59) check for the pop-up trigger event. Each method calls private utility method `checkForTriggerEvent` (lines 62–67) to determine whether the pop-up trigger event occurred. If it did, `MouseEvent` method **`isPopupTrigger`** returns `true`, and `JPopupMenu` method **`show`** 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* coordinates (measured from the origin component's upper-left corner) at which the `JPopupMenu` is to appear.

Look-and-Feel Observation 35.9

Displaying a `JPopupMenu` for the pop-up trigger event of multiple GUI components requires registering mouse-event handlers for each of those GUI components.

When the user selects a menu item from the pop-up menu, class `ItemHandler`'s method `actionPerformed` (lines 76–88) determines which `JRadioButtonMenuItem` the user selected and sets the background color of the window's content pane.

35.6 Pluggable Look-and-Feel

A program that uses Java's AWT GUI components (package `java.awt`) takes on the look-and-feel of the platform on which the program executes. A Java application running on a macOS looks like other macOS applications, one running on Microsoft Windows looks like other Windows applications, and one running on a Linux platform looks like other applications on that Linux platform. This is sometimes desirable, because it allows users of the application on each platform to use GUI components with which they're already familiar. However, it also introduces interesting portability issues.

Portability Tip 35.1

GUI components often look different on different platforms (fonts, font sizes, component borders, etc.) and might require different amounts of space to display. This could change their layout and alignments.

Portability Tip 35.2

GUI components on different platforms have might different default functionality—e.g., not all platforms allow a button with the focus to be "pressed" with the space bar.

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. Section 26.2 introduced the *Nimbus* look-and-feel. Earlier versions of Java used the **metal look-and-feel**, which is still the default. Swing also provides the flexibility to customize the look-and-feel to appear as a Microsoft Windows-style look-and-feel (only on Windows systems), a Motif-style (UNIX) look-and-feel (across all platforms) or a Macintosh look-and-feel (only on Mac systems).

Figures 35.9–35.10 demonstrate a way to change the look-and-feel of a Swing GUI. It creates several GUI components, so you can see the change in their look-and-feel at the same time. The output windows show the Metal, Nimbus, CDE/Motif, Windows and Windows Classic look-and-feels that are available on Windows systems. The installed look-and-feels will vary by platform.

We've covered the GUI components and event-handling concepts in this example previously, so we focus here on the mechanism for changing the look-and-feel. Class **`UIManager`** (package `javax.swing`) contains nested class **`LookAndFeelInfo`** (a `public static` class) that maintains information about a look-and-feel. Line 20 (Fig. 35.9) declares an array of type `UIManager.LookAndFeelInfo` (note the syntax used to identify the `static` inner class `LookAndFeelInfo`). Line 34 uses `UIManager` `static` method **`getInstalledLookAndFeels`** to get the array of `UIManager.LookAndFeelInfo` objects that describe each look-and-feel available on your system.

Performance Tip 35.1

Each look-and-feel is represented by a Java class. `UIManager` method `getInstalledLookAndFeels` does not load each class. Rather, it provides the names of the available look-and-feel classes so that a choice can be made (presumably once at program start-up). This reduces the overhead of having to load all the look-and-feel classes even if the program will not use some of them.

```
1  // Fig. 22.9: LookAndFeelFrame.java
2  // Changing the look-and-feel.
3  import java.awt.GridLayout;
4  import java.awt.BorderLayout;
5  import java.awt.event.ItemListener;
6  import java.awt.event.ItemEvent;
7  import javax.swing.JFrame;
8  import javax.swing.UIManager;
9  import javax.swing.JRadioButton;
10 import javax.swing.ButtonGroup;
11 import javax.swing.JButton;
12 import javax.swing.JLabel;
13 import javax.swing.JComboBox;
14 import javax.swing.JPanel;
15 import javax.swing.SwingConstants;
16 import javax.swing.SwingUtilities;
17
18 public class LookAndFeelFrame extends JFrame
19 {
```

Fig. 35.9 | Look-and-feel of a Swing-based GUI. (Part 1 of 3.)

```
20      private final UIManager.LookAndFeelInfo[] looks;
21      private final String[] lookNames; // look-and-feel names
22      private final JRadioButton[] radio; // for selecting look-and-feel
23      private final ButtonGroup group; // group for radio buttons
24      private final JButton button; // displays look of button
25      private final JLabel label; // displays look of label
26      private final JComboBox<String> comboBox; // displays look of combo box
27
28      // set up GUI
29      public LookAndFeelFrame()
30      {
31         super("Look and Feel Demo");
32
33         // get installed look-and-feel information
34         looks = UIManager.getInstalledLookAndFeels();
35         lookNames = new String[looks.length];
36
37         // get names of installed look-and-feels
38         for (int i = 0; i < looks.length; i++)
39            lookNames[i] = looks[i].getName();
40
41         JPanel northPanel = new JPanel();
42         northPanel.setLayout(new GridLayout(3, 1, 0, 5));
43
44         label = new JLabel("This is a " + lookNames[0] + " look-and-feel",
45            SwingConstants.CENTER);
46         northPanel.add(label);
47
48         button = new JButton("JButton");
49         northPanel.add(button);
50
51         comboBox = new JComboBox<String>(lookNames);
52         northPanel.add(comboBox);
53
54         // create array for radio buttons
55         radio = new JRadioButton[looks.length];
56
57         JPanel southPanel = new JPanel();
58
59         // use a GridLayout with 3 buttons in each row
60         int rows = (int) Math.ceil(radio.length / 3.0);
61         southPanel.setLayout(new GridLayout(rows, 3));
62
63         group = new ButtonGroup(); // button group for look-and-feels
64         ItemHandler handler = new ItemHandler(); // look-and-feel handler
65
66         for (int count = 0; count < radio.length; count++)
67         {
68            radio[count] = new JRadioButton(lookNames[count]);
69            radio[count].addItemListener(handler); // add handler
70            group.add(radio[count]); // add radio button to group
71            southPanel.add(radio[count]); // add radio button to panel
72         }
```

Fig. 35.9 | Look-and-feel of a Swing-based GUI. (Part 2 of 3.)

```
73
74          add(northPanel, BorderLayout.NORTH); // add north panel
75          add(southPanel, BorderLayout.SOUTH); // add south panel
76
77          radio[0].setSelected(true); // set default selection
78       } // end LookAndFeelFrame constructor
79
80       // use UIManager to change look-and-feel of GUI
81       private void changeTheLookAndFeel(int value)
82       {
83          try // change look-and-feel
84          {
85             // set look-and-feel for this application
86             UIManager.setLookAndFeel(looks[value].getClassName());
87
88             // update components in this application
89             SwingUtilities.updateComponentTreeUI(this);
90          }
91          catch (Exception exception)
92          {
93             exception.printStackTrace();
94          }
95       }
96
97       // private inner class to handle radio button events
98       private class ItemHandler implements ItemListener
99       {
100         // process user's look-and-feel selection
101         @Override
102         public void itemStateChanged(ItemEvent event)
103         {
104            for (int count = 0; count < radio.length; count++)
105            {
106               if (radio[count].isSelected())
107               {
108                  label.setText(String.format(
109                     "This is a %s look-and-feel", lookNames[count]));
110                  comboBox.setSelectedIndex(count); // set combobox index
111                  changeTheLookAndFeel(count); // change look-and-feel
112               }
113            }
114         }
115      } // end private inner class ItemHandler
116   } // end class LookAndFeelFrame
```

Fig. 35.9 | Look-and-feel of a Swing-based GUI. (Part 3 of 3.)

```
1  // Fig. 22.10: LookAndFeelDemo.java
2  // Changing the look-and-feel.
3  import javax.swing.JFrame;
4
```

Fig. 35.10 | Test class for `LookAndFeelFrame`. (Part 1 of 2.)

```
 5  public class LookAndFeelDemo
 6  {
 7     public static void main(String[] args)
 8     {
 9        LookAndFeelFrame lookAndFeelFrame = new LookAndFeelFrame();
10        lookAndFeelFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        lookAndFeelFrame.setSize(400, 220);
12        lookAndFeelFrame.setVisible(true);
13     }
14  } // end class LookAndFeelDemo
```

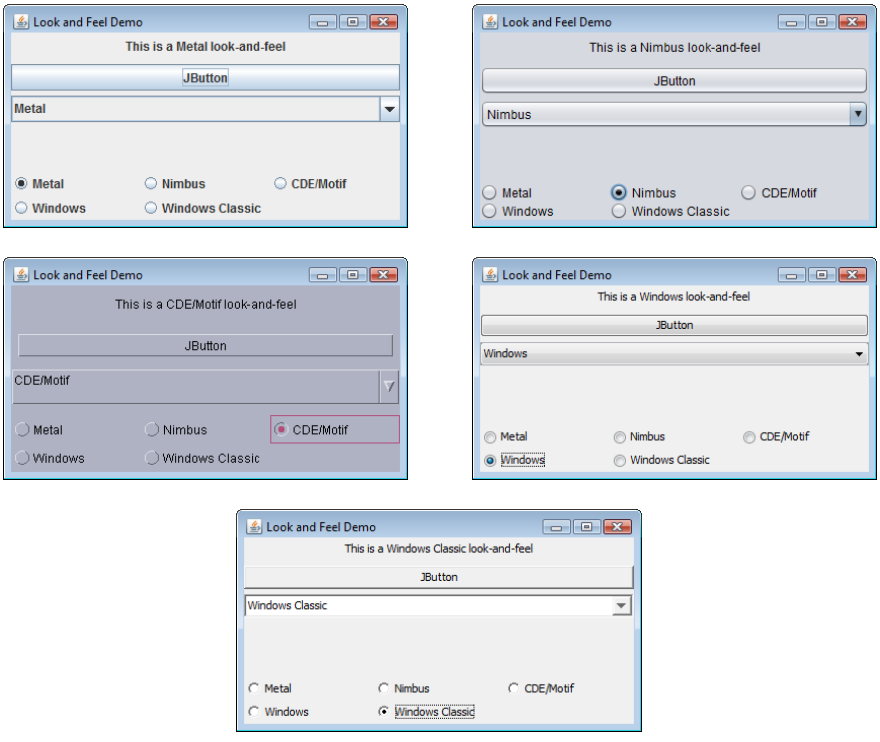


Fig. 35.10 | Test class for `LookAndFeelFrame`. (Part 2 of 2.)

Our utility method `changeTheLookAndFeel` (lines 81–95) is called by the event handler for the `JRadioButtons` at the bottom of the user interface. The event handler (declared in `private` inner class `ItemHandler` at lines 98–115) passes an integer representing the element in array `looks` that should be used to change the look-and-feel. Line 86 invokes `static` method **`setLookAndFeel`** of `UIManager` to change the look-and-feel. The **`getClassName`** method of class `UIManager.LookAndFeelInfo` determines the name of the look-and-feel class that corresponds to the `UIManager.LookAndFeelInfo` object. If the look-and-feel is not already loaded, it will be loaded as part of the call to `setLookAndFeel`. Line 89 invokes the `static` method **`updateComponentTreeUI`** of class **`SwingUtilities`** (package `javax.swing`) to change the look-and-feel of every GUI component attached to its argument (`this` instance of our application class `LookAndFeelFrame`) to the new look-and-feel.

35.7 JDesktopPane and JInternalFrame

A **multiple-document interface** (**MDI**) is a main window (called the **parent window**) containing other windows (called **child windows**) and is often used to manage several open documents. For example, many e-mail programs allow you to have several windows open at the same time, so you can compose or read multiple e-mail messages simultaneously. Similarly, many word processors allow the user to open multiple documents in separate windows within a main window, making it possible to switch between them without having to close one to open another. The application in Figs. 35.11–35.12 demonstrates Swing's **`JDesktopPane`** and **`JInternalFrame`** classes for implementing multiple-document interfaces.

```
1  // Fig. 22.11: DesktopFrame.java
2  // Demonstrating JDesktopPane.
3  import java.awt.BorderLayout;
4  import java.awt.Dimension;
5  import java.awt.Graphics;
6  import java.awt.event.ActionListener;
7  import java.awt.event.ActionEvent;
8  import java.util.Random;
9  import javax.swing.JFrame;
10 import javax.swing.JDesktopPane;
11 import javax.swing.JMenuBar;
12 import javax.swing.JMenu;
13 import javax.swing.JMenuItem;
14 import javax.swing.JInternalFrame;
15 import javax.swing.JPanel;
16 import javax.swing.ImageIcon;
17
18 public class DesktopFrame extends JFrame
19 {
20    private final JDesktopPane theDesktop;
21
22    // set up GUI
23    public DesktopFrame()
24    {
25       super("Using a JDesktopPane");
26
27       JMenuBar bar = new JMenuBar();
28       JMenu addMenu = new JMenu("Add");
29       JMenuItem newFrame = new JMenuItem("Internal Frame");
30
31       addMenu.add(newFrame); // add new frame item to Add menu
32       bar.add(addMenu); // add Add menu to menu bar
33       setJMenuBar(bar); // set menu bar for this application
34
35       theDesktop = new JDesktopPane();
36       add(theDesktop); // add desktop pane to frame
37
38       // set up listener for newFrame menu item
39       newFrame.addActionListener(
```

Fig. 35.11 | Multiple-document interface. (Part 1 of 2.)

```
40            new ActionListener() // anonymous inner class
41            {
42               // display new internal window
43               @Override
44               public void actionPerformed(ActionEvent event)
45               {
46                  // create internal frame
47                  JInternalFrame frame = new JInternalFrame(
48                     "Internal Frame", true, true, true, true);
49
50                  MyJPanel panel = new MyJPanel();
51                  frame.add(panel, BorderLayout.CENTER);
52                  frame.pack(); // set internal frame to size of contents
53
54                  theDesktop.add(frame); // attach internal frame
55                  frame.setVisible(true); // show internal frame
56               }
57            }
58         );
59      } // end DesktopFrame constructor
60   } // end class DesktopFrame
61
62   // class to display an ImageIcon on a panel
63   class MyJPanel extends JPanel
64   {
65      private static final SecureRandom generator = new SecureRandom();
66      private final ImageIcon picture; // image to be displayed
67      private final static String[] images = { "yellowflowers.png",
68         "purpleflowers.png", "redflowers.png", "redflowers2.png",
69         "lavenderflowers.png" };
70
71      // load image
72      public MyJPanel()
73      {
74         int randomNumber = generator.nextInt(images.length);
75         picture = new ImageIcon(images[randomNumber]); // set icon
76      }
77
78      // display imageIcon on panel
79      @Override
80      public void paintComponent(Graphics g)
81      {
82         super.paintComponent(g);
83         picture.paintIcon(this, g, 0, 0); // display icon
84      }
85
86      // return image dimensions
87      public Dimension getPreferredSize()
88      {
89         return new Dimension(picture.getIconWidth(),
90            picture.getIconHeight());
91      }
92   } // end class MyJPanel
```

Fig. 35.11 | Multiple-document interface. (Part 2 of 2.)

Lines 27–33 create 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 application creates and displays a new `JInternalFrame` object containing an image.

Line 35 assigns `JDesktopPane` (package `javax.swing`) variable `theDesktop` a new `JDesktopPane` object that will be used to manage the `JInternalFrame` child windows. Line 36 adds the `JDesktopPane` to the `JFrame`. By default, the `JDesktopPane` is added to the center of the content pane's `BorderLayout`, so the `JDesktopPane` expands to fill the entire application window.

```
1  // Fig. 22.12: DesktopTest.java
2  // Demonstrating JDesktopPane.
3  import javax.swing.JFrame;
4
5  public class DesktopTest
6  {
7     public static void main(String[] args)
8     {
9        DesktopFrame desktopFrame = new DesktopFrame();
10       desktopFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       desktopFrame.setSize(600, 480);
12       desktopFrame.setVisible(true);
13    }
14 } // end class DesktopTest
```

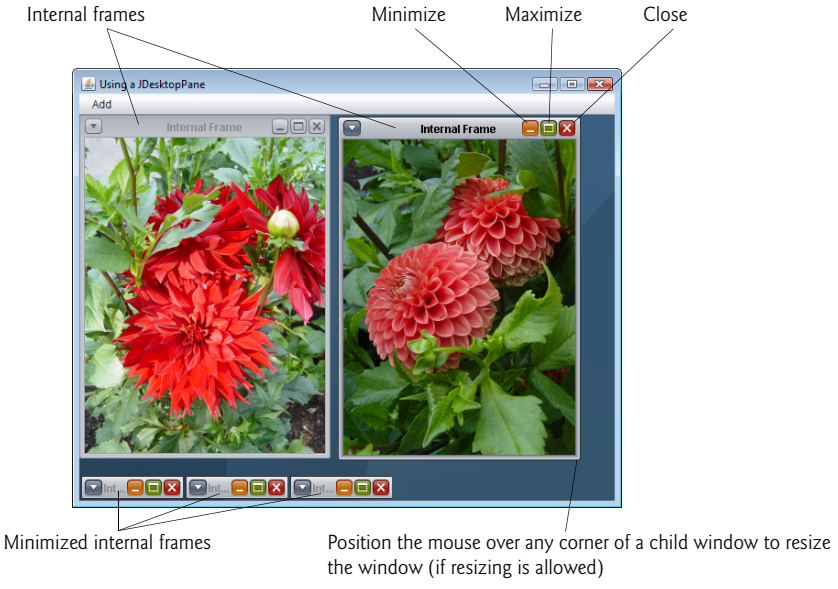


Fig. 35.12 | Test class for `DeskTopFrame`. (Part 1 of 2.)

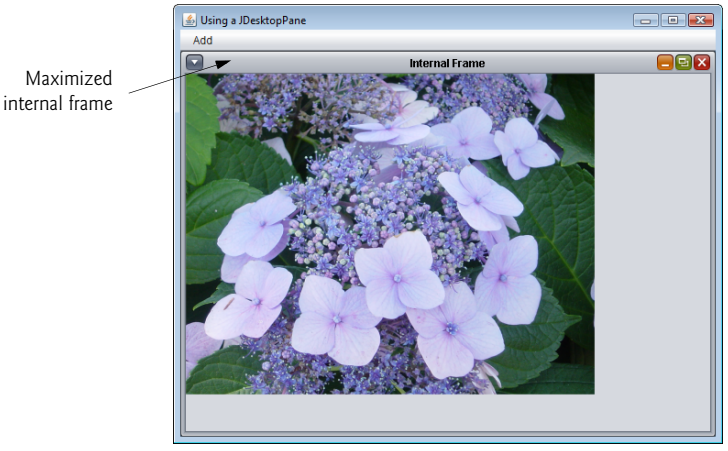


Fig. 35.12 | Test class for `DeskTopFrame`. (Part 2 of 2.)

Lines 39–58 register an `ActionListener` to handle the event when the user selects the `newFrame` menu item. When the event occurs, method `actionPerformed` (lines 43–56) creates a `JInternalFrame` object in lines 47–48. The `JInternalFrame` constructor used here takes five arguments—a `String` for the title bar of the internal window, a `boolean` indicating whether the internal frame can be resized by the user, a `boolean` indicating whether the internal frame can be closed by the user, a `boolean` indicating whether the internal frame can be maximized by the user and a `boolean` indicating whether the internal frame can be minimized by the user. For each of the `boolean` arguments, a `true` value indicates that the operation should be allowed (as is the case here).

As with `JFrames` and `JApplets`, a `JInternalFrame` has a content pane to which GUI components can be attached. Line 50 creates an instance of our class `MyJPanel` (declared at lines 63–91) that is added to the `JInternalFrame` at line 51.

Line 52 uses `JInternalFrame` method **`pack`** to set the size of the child window. Method `pack` uses the preferred sizes of the components to determine the window's size. Class `MyJPanel` declares method `getPreferredSize` (lines 87–91) to specify the panel's preferred size for use by the `pack` method. Line 54 adds the `JInternalFrame` to the `JDesktopPane`, and line 55 displays the `JInternalFrame`.

Classes `JInternalFrame` and `JDesktopPane` provide many methods for managing child windows. See the `JInternalFrame` and `JDesktopPane` online API documentation for complete lists of these methods:

```
docs.oracle.com/javase/8/docs/api/javax/swing/JInternalFrame.html
docs.oracle.com/javase/8/docs/api/javax/swing/JDesktopPane.html
```

35.8 JTabbedPane

A **`JTabbedPane`** arranges GUI components into layers, of which only one is visible at a time. Users access each layer via a tab—similar to folders in a file cabinet. When the user clicks a tab, the appropriate layer is displayed. The tabs appear at the top by default but also can be positioned at the left, right or bottom of the `JTabbedPane`. Any component

can be placed on a tab. If the component is a container, such as a panel, it can use any layout manager to lay out several components on the tab. Class `JTabbedPane` is a subclass of `JComponent`. The application in Figs. 35.13–35.14 creates one tabbed pane with three tabs. Each tab displays one of the `JPanel`s—`panel1`, `panel2` or `panel3`.

```
1  // Fig. 22.13: JTabbedPaneFrame.java
2  // Demonstrating JTabbedPane.
3  import java.awt.BorderLayout;
4  import java.awt.Color;
5  import javax.swing.JFrame;
6  import javax.swing.JTabbedPane;
7  import javax.swing.JLabel;
8  import javax.swing.JPanel;
9  import javax.swing.JButton;
10 import javax.swing.SwingConstants;
11
12 public class JTabbedPaneFrame extends JFrame
13 {
14    // set up GUI
15    public JTabbedPaneFrame()
16    {
17       super("JTabbedPane Demo ");
18
19       JTabbedPane tabbedPane = new JTabbedPane(); // create JTabbedPane
20
21       // set up pane11 and add it to JTabbedPane
22       JLabel label1 = new JLabel("panel one", SwingConstants.CENTER);
23       JPanel panel1 = new JPanel();
24       panel1.add(label1);
25       tabbedPane.addTab("Tab One", null, panel1, "First Panel");
26
27       // set up panel2 and add it to JTabbedPane
28       JLabel label2 = new JLabel("panel two", SwingConstants.CENTER);
29       JPanel panel2 = new JPanel();
30       panel2.setBackground(Color.YELLOW);
31       panel2.add(label2);
32       tabbedPane.addTab("Tab Two", null, panel2, "Second Panel");
33
34       // set up panel3 and add it to JTabbedPane
35       JLabel label3 = new JLabel("panel three");
36       JPanel panel3 = new JPanel();
37       panel3.setLayout(new BorderLayout());
38       panel3.add(new JButton("North"), BorderLayout.NORTH);
39       panel3.add(new JButton("West"), BorderLayout.WEST);
40       panel3.add(new JButton("East"), BorderLayout.EAST);
41       panel3.add(new JButton("South"), BorderLayout.SOUTH);
42       panel3.add(label3, BorderLayout.CENTER);
43       tabbedPane.addTab("Tab Three", null, panel3, "Third Panel");
44
45       add(tabbedPane); // add JTabbedPane to frame
46    }
47 } // end class JTabbedPaneFrame
```

Fig. 35.13 | `JTabbedPane` used to organize GUI components.

```
1  // Fig. 22.14: JTabbedPaneDemo.java
2  // Demonstrating JTabbedPane.
3  import javax.swing.JFrame;
4
5  public class JTabbedPaneDemo
6  {
7     public static void main(String[] args)
8     {
9        JTabbedPaneFrame tabbedPaneFrame = new JTabbedPaneFrame();
10       tabbedPaneFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       tabbedPaneFrame.setSize(250, 200);
12       tabbedPaneFrame.setVisible(true);
13    }
14 } // end class JTabbedPaneDemo
```

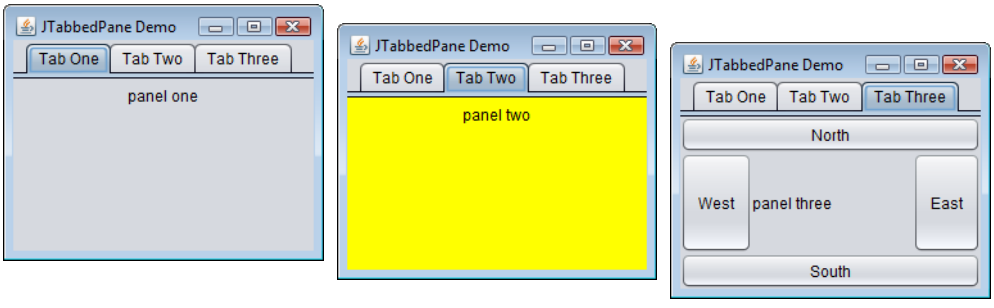


Fig. 35.14 | Test class for `JTabbedPaneFrame`.

The constructor (lines 15–46) builds the GUI. Line 19 creates an empty `JTabbedPane` with default settings—that is, tabs across the top. If the tabs do not fit on one line, they'll wrap to form additional lines of tabs. Next the constructor creates the `JPanel`s `panel1`, `panel2` and `panel3` and their GUI components. As we set up each panel, we add it to `tabbedPane`, using `JTabbedPane` method **`addTab`** with four arguments. The first argument is a `String` that specifies the title of the tab. The second argument is an `Icon` reference that specifies an icon to display on the tab. If the `Icon` is a `null` reference, no image is displayed. The third argument is a `Component` reference that represents the GUI component to display when the user clicks the tab. The last argument is a `String` that specifies the tool tip for the tab. For example, line 25 adds `JPanel panel1` to `tabbedPane` with title `"Tab One"` and the tool tip `"First Panel"`. `JPanel`s `panel2` and `panel3` are added to `tabbedPane` at lines 32 and 43. To view a tab, click it with the mouse or use the arrow keys to cycle through the tabs.

35.9 BoxLayout Layout Manager

In Chapter 26, we introduced three layout managers—`FlowLayout`, `BorderLayout` and `GridLayout`. This section and Section 35.10 present two additional layout managers (summarized in Fig. 35.15). We discuss them in the examples that follow.

Layout manager	Description
`BoxLayout`	Allows GUI components to be arranged left-to-right or top-to-bottom in a container. Class `Box` declares a container that uses `BoxLayout` and provides `static` methods to create a `Box` with a horizontal or vertical `BoxLayout`.
`GridBagLayout`	Similar to `GridLayout`, but the components can vary in size and can be added in any order.

Fig. 35.15 | Additional layout managers.

The `BoxLayout` layout manager (in package `javax.swing`) arranges GUI components horizontally along a container's *x*-axis or vertically along its *y*-axis. The application in Figs. 35.16–35.17 demonstrate `BoxLayout` and the container class `Box` that uses `BoxLayout` as its default layout manager.

```
1  // Fig. 22.16: BoxLayoutFrame.java
2  // Demonstrating BoxLayout.
3  import java.awt.Dimension;
4  import javax.swing.JFrame;
5  import javax.swing.Box;
6  import javax.swing.JButton;
7  import javax.swing.BoxLayout;
8  import javax.swing.JPanel;
9  import javax.swing.JTabbedPane;
10
11 public class BoxLayoutFrame extends JFrame
12 {
13    // set up GUI
14    public BoxLayoutFrame()
15    {
16       super("Demonstrating BoxLayout");
17
18       // create Box containers with BoxLayout
19       Box horizontal1 = Box.createHorizontalBox();
20       Box vertical1 = Box.createVerticalBox();
21       Box horizontal2 = Box.createHorizontalBox();
22       Box vertical2 = Box.createVerticalBox();
23
24       final int SIZE = 3; // number of buttons on each Box
25
26       // add buttons to Box horizontal1
27       for (int count = 0; count < SIZE; count++)
28          horizontal1.add(new JButton("Button " + count));
29
30       // create strut and add buttons to Box vertical1
31       for (int count = 0; count < SIZE; count++)
32       {
33          vertical1.add(Box.createVerticalStrut(25));
34          vertical1.add(new JButton("Button " + count));
35       }
```

Fig. 35.16 | `BoxLayout` layout manager. (Part 1 of 2.)

```
36
37          // create horizontal glue and add buttons to Box horizontal2
38          for (int count = 0; count < SIZE; count++)
39          {
40             horizontal2.add(Box.createHorizontalGlue());
41             horizontal2.add(new JButton("Button " + count));
42          }
43
44          // create rigid area and add buttons to Box vertical2
45          for (int count = 0; count < SIZE; count++)
46          {
47             vertical2.add(Box.createRigidArea(new Dimension(12, 8)));
48             vertical2.add(new JButton("Button " + count));
49          }
50
51          // create vertical glue and add buttons to panel
52          JPanel panel = new JPanel();
53          panel.setLayoutnew BoxLayout(panel, BoxLayout.Y_AXIS)  ();
54
55          for (int count = 0; count < SIZE; count++)
56          {
57             panel.add(Box.createGlue());
58             panel.add(new JButton("Button " + count));
59          }
60
61          // create a JTabbedPane
62          JTabbedPane tabs = new JTabbedPane(
63             JTabbedPane.TOP, JTabbedPane.SCROLL_TAB_LAYOUT);
64
65          // place each container on tabbed pane
66          tabs.addTab("Horizontal Box", horizontal1);
67          tabs.addTab("Vertical Box with Struts", vertical1);
68          tabs.addTab("Horizontal Box with Glue", horizontal2);
69          tabs.addTab("Vertical Box with Rigid Areas", vertical2);
70          tabs.addTab("Vertical Box with Glue", panel);
71
72          add(tabs); // place tabbed pane on frame
73       } // end BoxLayoutFrame constructor
74    } // end class BoxLayoutFrame
```

Fig. 35.16 | `BoxLayout` layout manager. (Part 2 of 2.)

```
1  // Fig. 22.17: BoxLayoutDemo.java
2  // Demonstrating BoxLayout.
3  import javax.swing.JFrame;
4
5  public class BoxLayoutDemo
6  {
7     public static void main(String[] args)
8     {
9        BoxLayoutFrame boxLayoutFrame = new BoxLayoutFrame();
```

Fig. 35.17 | Test class for `BoxLayoutFrame`. (Part 1 of 2.)

```
10          boxLayoutFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11          boxLayoutFrame.setSize(400, 220);
12          boxLayoutFrame.setVisible(true);
13       }
14    } // end class BoxLayoutDemo
```

Fig. 35.17 | Test class for `BoxLayoutFrame`. (Part 2 of 2.)

Creating `Box` *Containers*

Lines 19–22 create `Box` containers. References `horizontal1` and `horizontal2` are initialized with `static` `Box` method `createHorizontalBox`, which returns a `Box` container with a horizontal `BoxLayout` in which GUI components are arranged left-to-right. Variables `vertical1` and `vertical2` are initialized with `static` `Box` method **`createVerticalBox`**, which returns references to `Box` containers with a vertical `BoxLayout` in which GUI components are arranged top-to-bottom.

Struts

The loop at lines 27–28 adds three `JButtons` to `horizontal1`. The `for` statement at lines 31–35 adds three `JButtons` to `vertical1`. Before adding each button, line 33 adds a **vertical strut** to the container with `static` `Box` method **`createVerticalStrut`**. 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 `int` argument to method `createVerticalStrut` determines the height of the strut in pixels. When the container is resized, the distance between GUI components separated by struts does not change. Class `Box` also declares method **`createHorizontalStrut`** for horizontal `BoxLayouts`.

Glue

The `for` statement at lines 38–42 adds three `JButtons` to `horizontal2`. Before adding each button, line 40 adds **horizontal glue** to the container with `static` `Box` method **`createHorizontalGlue`**. Horizontal glue is an invisible 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 one in a `BoxLayout`. Glue allows the extra space to be placed between GUI components. When the container is resized, components separated by glue components remain the same size, but the glue stretches or contracts to occupy the space between them. Class `Box` also declares method **`createVerticalGlue`** for vertical `BoxLayouts`.

Rigid Areas

The `for` statement at lines 45–49 adds three `JButtons` to `vertical2`. Before each button is added, line 47 adds a **rigid area** to the container with `static` `Box` method **`createRigidArea`**. 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 area's width and height.

Setting a `BoxLayout` for a `Container`

Lines 52–53 create a `JPanel` object and set its layout to a `BoxLayout` 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`**).

Adding `Glue` and `JButtons`

The `for` statement at lines 55–59 adds three `JButtons` to `panel`. Before adding each button, line 57 adds a glue component to the container with `static` `Box` method **`createGlue`**. This component expands or contracts based on the size of the `Box`.

Creating the `JTabbedPane`

Lines 62–63 create a `JTabbedPane` to display the five containers in this program. The argument **`JTabbedPane.TOP`** sent to the constructor indicates that the tabs should appear at the top of the `JTabbedPane`. The argument **`JTabbedPane.SCROLL_TAB_LAYOUT`** specifies that the tabs should wrap to a new line if there are too many to fit on one line.

Attaching the `Box` Containers and `JPanel` to the `JTabbedPane`

The `Box` containers and the `JPanel` are attached to the `JTabbedPane` at lines 66–70. 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 on each tab.

35.10 `GridBagLayout` Layout Manager

One of the most powerful predefined layout managers is **`GridBagLayout`** (in package `java.awt`). This layout is similar to `GridLayout` in that it arranges components in a grid, but it's 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. For this step you need only a piece of paper. Draw the GUI, then draw a grid over it, dividing

the components into rows and columns. The initial row and column numbers should be 0, so that the `GridBagLayout` layout manager can use the row and column numbers to properly place the components in the grid. Figure 35.18 demonstrates drawing the lines for the rows and columns over a GUI.

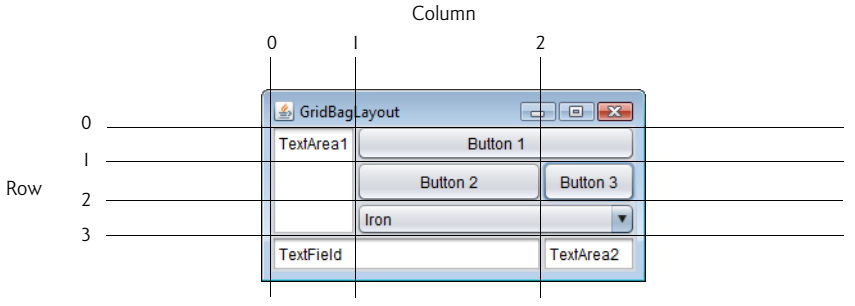


Fig. 35.18 | Designing a GUI that will use `GridBagLayout`.

GridBagConstraints

A **`GridBagConstraints`** object describes how a component is placed in a `GridBagLayout`. Several `GridBagConstraints` fields are summarized in Fig. 35.19.

Field	Description
`anchor`	Specifies the relative position (`NORTH`, `NORTHEAST`, `EAST`, `SOUTHEAST`, `SOUTH`, `SOUTHWEST`, `WEST`, `NORTHWEST`, `CENTER`) of the component in an area that it does not fill.
`fill`	Resizes the component in the specified direction (`NONE`, `HORIZONTAL`, `VERTICAL`, `BOTH`) when the display area is larger than the component.
`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 amount of extra space to allocate horizontally. The grid slot can become wider when extra space is available.
`weighty`	The amount of extra space to allocate vertically. The grid slot can become taller when extra space is available.

Fig. 35.19 | `GridBagConstraints` fields.

GridBagConstraints Field anchor

`GridBagConstraints` field **`anchor`** specifies the relative position of the component in an area that it does not fill. 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`.

`GridBagConstraints` Field `fill`

`GridBagConstraints` field `fill` defines how the component grows if the area in which it can be displayed is larger than the component. 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 it will grow vertically. `HORIZONTAL` indicates that it will grow horizontally. `BOTH` indicates that it will grow in both directions.

`GridBagConstraints` Fields `gridx` and `gridy`

Variables **`gridx`** and **`gridy`** specify where the upper-left corner of the component is placed in the grid. Variable `gridx` corresponds to the column, and variable `gridy` corresponds to the row. In Fig. 35.18, the `JComboBox` (displaying "`Iron`") has a `gridx` value of 1 and a `gridy` value of 2.

`GridBagConstraints` Field `gridwidth`

Variable **`gridwidth`** specifies the number of columns a component occupies. The `JComboBox` occupies two columns. Variable **`gridheight`** specifies the number of rows a component occupies. The `JTextArea` on the left side of Fig. 35.18 occupies three rows.

`GridBagConstraints` Field `weightx`

Variable **`weightx`** specifies how to distribute extra horizontal space to grid slots in a `GridBagLayout` when the container is resized. A zero value indicates that the grid slot 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 that column. This is because each component must be maintained in the same row and column in which it was originally placed.

`GridBagConstraints` Field `weighty`

Variable **`weighty`** specifies how to distribute extra vertical space to grid slots in a `GridBagLayout` when the container is resized. A zero value indicates that the grid slot 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.

Effects of `weightx` and `weighty`

In Fig. 35.18, 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 those with smaller weight values.

Components should be given nonzero positive weight values—otherwise they'll "huddle" together in the middle of the container. Figure 35.20 shows the GUI of Fig. 35.18 with all weights set to zero.

Demonstrating `GridBagLayout`

The application in Figs. 35.21–35.22 uses the `GridBagLayout` layout manager to arrange the components of the GUI in Fig. 35.18. The application does nothing except demonstrate how to use `GridBagLayout`.

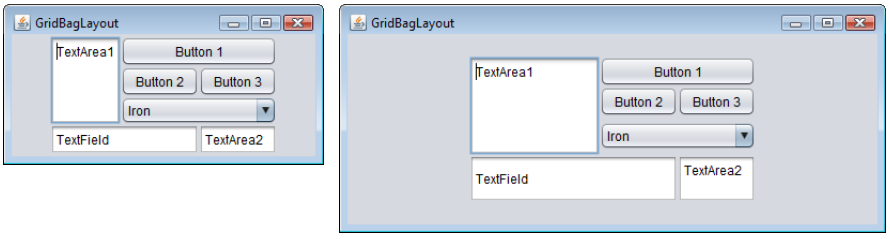


Fig. 35.20 | `GridBagLayout` with the weights set to zero.

```
1  // Fig. 22.21: GridBagFrame.java
2  // Demonstrating GridBagLayout.
3  import java.awt.GridBagLayout;
4  import java.awt.GridBagConstraints;
5  import java.awt.Component;
6  import javax.swing.JFrame;
7  import javax.swing.JTextArea;
8  import javax.swing.JTextField;
9  import javax.swing.JButton;
10 import javax.swing.JComboBox;
11
12 public class GridBagFrame extends JFrame
13 {
14    private final GridBagLayout layout; // layout of this frame
15    private final GridBagConstraints constraints; // layout's constraints
16
17    // set up GUI
18    public GridBagFrame()
19    {
20       super("GridBagLayout");
21       layout = new GridBagLayout();
22       setLayout(layout); // set frame layout
23       constraints = new GridBagConstraints(); // instantiate constraints
24
25       // create GUI components
26       JTextArea textArea1 = new JTextArea("TextArea1", 5, 10);
27       JTextArea textArea2 = new JTextArea("TextArea2", 2, 2);
28
29       String[] names = { "Iron", "Steel", "Brass" };
30       JComboBox<String> comboBox = new JComboBox<String>(names);
31
32       JTextField textField = new JTextField("TextField");
33       JButton button1 = new JButton("Button 1");
34       JButton button2 = new JButton("Button 2");
35       JButton button3 = new JButton("Button 3");
36
37       // weightx and weighty for textArea1 are both 0: the default
38       // anchor for all components is CENTER: the default
39       constraints.fill = GridBagConstraints.BOTH;
40       addComponent(textArea1, 0, 0, 1, 3);
```

Fig. 35.21 | `GridBagLayout` layout manager. (Part 1 of 2.)

```
41
42          // weightx and weighty for button1 are both 0: the default
43          constraints.fill = GridBagConstraints.HORIZONTAL;
44          addComponent(button1, 0, 1, 2, 1);
45
46          // weightx and weighty for comboBox are both 0: the default
47          // fill is HORIZONTAL
48          addComponent(comboBox, 2, 1, 2, 1);
49
50          // button2
51          constraints.weightx = 1000;  // can grow wider
52          constraints.weighty = 1;     // can grow taller
53          constraints.fill = GridBagConstraints.BOTH;
54          addComponent(button2, 1, 1, 1, 1);
55
56          // fill is BOTH for button3
57          constraints.weightx = 0;
58          constraints.weighty = 0;
59          addComponent(button3, 1, 2, 1, 1);
60
61          // weightx and weighty for textField are both 0, fill is BOTH
62          addComponent(textField, 3, 0, 2, 1);
63
64          // weightx and weighty for textArea2 are both 0, fill is BOTH
65          addComponent(textArea2, 3, 2, 1, 1);
66       } // end GridBagFrame constructor
67
68       // method to set constraints on
69       private void addComponent(Component component,
70          int row, int column, int width, int height)
71       {
72          constraints.gridx = column;
73          constraints.gridy = row;
74          constraints.gridwidth = width;
75          constraints.gridheight = height;
76          layout.setConstraints(component, constraints); // set constraints
77          add(component); // add component
78       }
79    } // end class GridBagFrame
```

Fig. 35.21 | `GridBagLayout` layout manager. (Part 2 of 2.)

```
1  // Fig. 22.22: GridBagDemo.java
2  // Demonstrating GridBagLayout.
3  import javax.swing.JFrame;
4
5  public class GridBagDemo
6  {
7     public static void main(String[] args)
8     {
9        GridBagFrame gridBagFrame = new GridBagFrame();
```

Fig. 35.22 | Test class for `GridBagFrame`. (Part 1 of 2.)

```
10          gridBagFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11          gridBagFrame.setSize(300, 150);
12          gridBagFrame.setVisible(true);
13       }
14    } // end class GridBagDemo
```

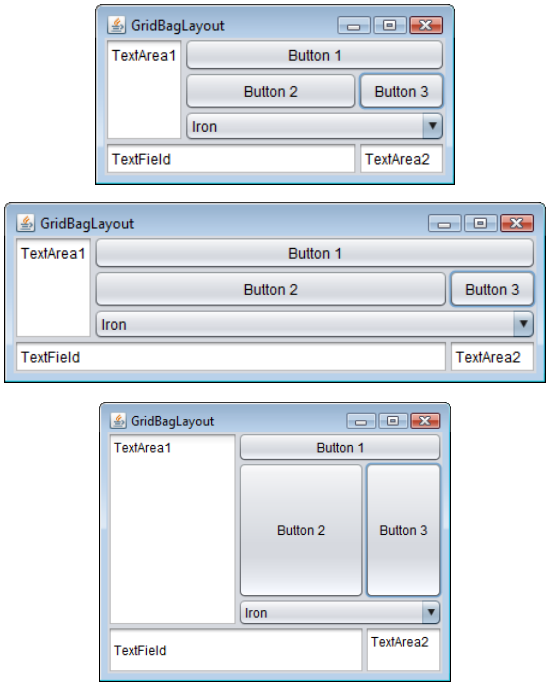


Fig. 35.22 | Test class for `GridBagFrame`. (Part 2 of 2.)

GUI Overview

The GUI contains three `JButtons`, two `JTextAreas`, a `JComboBox` and a `JTextField`. The layout manager is `GridBagLayout`. Lines 21–22 create the `GridBagLayout` object and set the layout manager for the `JFrame` to `layout`. Line 23 creates the `GridBagConstraints` object used to determine the location and size of each component in the grid. Lines 26–35 create each GUI component that will be added to the content pane.

`JTextArea textArea1`

Lines 39–40 configure `JTextArea textArea1` and add it to the content pane. The values for `weightx` and `weighty` values are not specified in `constraints`, so each has the value zero by default. Thus, the `JTextArea` will not resize itself even if space is available. However, it spans multiple rows, so the vertical size is subject to the `weighty` values of `JButtons` `button2` and `button3`. When either button is resized vertically based on its `weighty` value, the `JTextArea` is also resized.

Line 39 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

application, so all the components will use the default. Line 40 calls our utility method `addComponent` (declared at lines 69–78). The `JTextArea` object, the row, the column, the number of columns to span and the number of rows to span are passed as arguments.

`JButton button1`

`JButton button1` is the next component added (lines 43–44). By default, the `weightx` and `weighty` values 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 button1` is located at row 0, column 1. One row and two columns are occupied.

`JComboBox comboBox`

`JComboBox comboBox` is the next component added (line 48). By default, 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. The `weightx`, `weighty` and `fill` variables retain the values set in `constraints` until they're changed. The `JComboBox` button is placed at row 2, column 1. One row and two columns are occupied.

`JButton button2`

`JButton button2` is the next component added (lines 51–54). It's 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, `button2` will grow. The button is placed at row 1, column 1. One row and one column are occupied.

`JButton button3`

`JButton button3` is added next (lines 57–59). 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's affected by the weight values of `button2`. The `weightx` value for `button2` is much larger than that for `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.

`JTextField textField` and `JTextArea textArea2`

Both the `JTextField textField` (line 62) and `JTextArea textArea2` (line 65) have a `weightx` value of 0 and a `weighty` value of 0. The value of `fill` is `BOTH`. The `JTextField` is placed at row 3, column 0, and the `JTextArea` at row 3, column 2. The `JTextField` occupies one row and two columns, the `JTextArea` one row and one column.

Method `addComponent`

Method `addComponent`'s parameters are a `Component` reference `component` and integers `row`, `column`, `width` and `height`. Lines 72–73 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 74–75 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 76 sets the GridBagConstraints for a component in the GridBagLayout. Method setConstraints of class GridBagLayout takes a Component argument and a GridBagConstraints argument. Line 77 adds the component to the JFrame.

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.

GridBagConstraints Constants RELATIVE and REMAINDER

Instead of gridx and gridy, a variation of GridBagLayout uses GridBagConstraints constants RELATIVE and REMAINDER. RELATIVE specifies that the next-to-last component in a particular row should be placed to the right of the previous component in the 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. The application in Figs. 35.23–35.24 arranges components in GridBagLayout, using these constants.

```
1  // Fig. 22.23: GridBagFrame2.java
2  // Demonstrating GridBagLayout constants.
3  import java.awt.GridBagLayout;
4  import java.awt.GridBagConstraints;
5  import java.awt.Component;
6  import javax.swing.JFrame;
7  import javax.swing.JComboBox;
8  import javax.swing.JTextField;
9  import javax.swing.JList;
10 import javax.swing.JButton;
11
12 public class GridBagFrame2 extends JFrame
13 {
14    private final GridBagLayout layout; // layout of this frame
15    private final GridBagConstraints constraints; // layout's constraints
16
17    // set up GUI
18    public GridBagFrame2()
19    {
20       super("GridBagLayout");
21       layout = new GridBagLayout();
22       setLayout(layout); // set frame layout
23       constraints = new GridBagConstraints(); // instantiate constraints
24
25       // create GUI components
26       String[] metals = { "Copper", "Aluminum", "Silver" };
27       JComboBox comboBox = new JComboBox(metals);
28
29       JTextField textField = new JTextField("TextField");
30
31       String[] fonts = { "Serif", "Monospaced" };
32       JList list = new JList(fonts);
33
```

Fig. 35.23 | GridBagConstraints constants RELATIVE and REMAINDER. (Part 1 of 2.)

```
34          String[] names = { "zero", "one", "two", "three", "four" };
35          JButton[] buttons = new JButton[names.length];
36
37          for (int count = 0; count < buttons.length; count++)
38             buttons[count] = new JButton(names[count]);
39
40          // define GUI component constraints for textField
41          constraints.weightx = 1;
42          constraints.weighty = 1;
43          constraints.fill = GridBagConstraints.BOTH;
44          constraints.gridwidth = GridBagConstraints.REMAINDER;
45          addComponent(textField);
46
47          // buttons[0] -- weightx and weighty are 1: fill is BOTH
48          constraints.gridwidth = 1;
49          addComponent(buttons[0]);
50
51          // buttons[1] -- weightx and weighty are 1: fill is BOTH
52          constraints.gridwidth = GridBagConstraints.RELATIVE;
53          addComponent(buttons[1]);
54
55          // buttons[2] -- weightx and weighty are 1: fill is BOTH
56          constraints.gridwidth = GridBagConstraints.REMAINDER;
57          addComponent(buttons[2]);
58
59          // comboBox -- weightx is 1: fill is BOTH
60          constraints.weighty = 0;
61          constraints.gridwidth = GridBagConstraints.REMAINDER;
62          addComponent(comboBox);
63
64          // buttons[3] -- weightx is 1: fill is BOTH
65          constraints.weighty = 1;
66          constraints.gridwidth = GridBagConstraints.REMAINDER;
67          addComponent(buttons[3]);
68
69          // buttons[4] -- weightx and weighty are 1: fill is BOTH
70          constraints.gridwidth = GridBagConstraints.RELATIVE;
71          addComponent(buttons[4]);
72
73          // list -- weightx and weighty are 1: fill is BOTH
74          constraints.gridwidth = GridBagConstraints.REMAINDER;
75          addComponent(list);
76       } // end GridBagFrame2 constructor
77
78       // add a component to the container
79       private void addComponent(Component component)
80       {
81          layout.setConstraints(component, constraints);
82          add(component); // add component
83       }
84    } // end class GridBagFrame2
```

Fig. 35.23 | `GridBagConstraints` constants `RELATIVE` and `REMAINDER`. (Part 2 of 2.)

```
1  // Fig. 22.24: GridBagDemo2.java
2  // Demonstrating GridBagLayout constants.
3  import javax.swing.JFrame;
4
5  public class GridBagDemo2
6  {
7     public static void main(String[] args)
8     {
9        GridBagFrame2 gridBagFrame = new GridBagFrame2();
10       gridBagFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11       gridBagFrame.setSize(300, 200);
12       gridBagFrame.setVisible(true);
13    }
14 } // end class GridBagDemo2
```

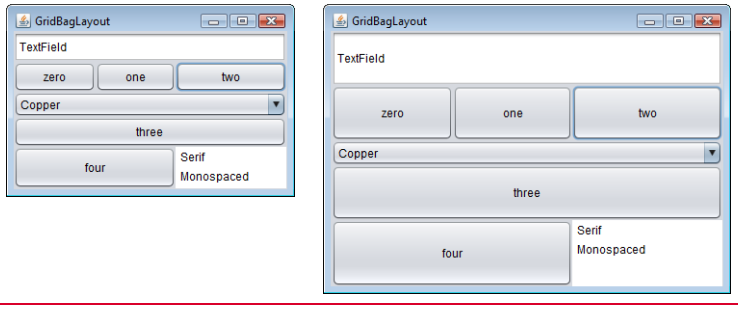


Fig. 35.24 | Test class for `GridBagDemo2`.

Setting the `JFrame`'s Layout to a `GridBagLayout`

Lines 21–22 create a `GridBagLayout` and use it to set the `JFrame`'s layout manager. The components that are placed in `GridBagLayout` are created in lines 27–38—they are a `JComboBox`, a `JTextField`, a `JList` and five `JButtons`.

Configuring the `JTextField`

The `JTextField` is added first (lines 41–45). The `weightx` and `weighty` values are set to 1. The `fill` variable is set to `BOTH`. Line 44 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` (declared at lines 79–83). 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.

Configuring `JButton buttons[0]`

`JButton buttons[0]` (lines 48–49) 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's given a `gridwidth` of 1 and so will occupy one column. The `JButton` is added to the content pane with a call to utility method `addComponent`.

Configuring `JButton buttons[1]`

`JButton buttons[1]` (lines 52–53) has `weightx` and `weighty` values of 1. The `fill` variable is `BOTH`. Line 52 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`.

Configuring `JButton buttons[2]`

`JButton buttons[2]` (lines 56–57) 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 `addComponent`.

Configuring `JComboBox`

The `JComboBox` (lines 60–62) has a `weightx` of 1 and a `weighty` of 0. The `JComboBox` will not grow vertically. 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 `addComponent`.

Configuring `JButton buttons[3]`

`JButton buttons[3]` (lines 65–67) 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 `addComponent`.

Configuring `JButton buttons[4]`

`JButton buttons[4]` (lines 70–71) 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 `addComponent`.

Configuring `JList`

The `JList` (lines 74–75) has `weightx` and `weighty` values of 1. The `fill` variable is `BOTH`. The `JList` is added to the content pane with a call to `addComponent`.

35.11 Wrap-Up

This chapter completes our introduction to GUIs. In this chapter, we discussed additional GUI topics, such as menus, sliders, pop-up menus, multiple-document interfaces, tabbed panes and Java's pluggable look-and-feel. All these components can be added to existing applications to make them easier to use and understand. We also presented additional layout managers for organizing and sizing GUI components.

Summary

Section 22.2 `JSlider`

- `JSliders` (p. 2) enable you to select from a range of integer values. They can display major and minor tick marks, and labels for the tick marks (p. 2). They also support snap-to ticks (p. 3)—positioning the thumb (p. 2) between two tick marks snaps it to the closest tick mark.

- `JSliders` have either horizontal or vertical orientation. For a horizontal `JSlider`, the minimum value is at the extreme left and the maximum value at the extreme right. For a vertical `JSlider`, the minimum value is at the extreme bottom and the maximum value at the extreme top. The position of the thumb indicates the current value of the `JSlider`. Method `getValue` (p. 6) of class `JSlider` returns the current thumb position.
- `JSlider` method `setMajorTickSpacing` () sets the spacing for tick marks on a `JSlider`. Method `setPaintTicks` (p. 6) with a `true` argument indicates that the tick marks should be displayed.
- `JSliders` generate `ChangeEvents` when the user interacts with a `JSlider`. A `ChangeListener` (p. 6) declares method `stateChanged` (p. 6) that can respond to `ChangeEvents`.

Section 35.3 Understanding Windows in Java

- A window's (p. 6) events can be handled by a `WindowListener` (p. 7), which provides seven window-event-handling methods—`windowActivated`, `windowClosed`, `windowClosing`, `windowDeactivated`, `windowDeiconified`, `windowIconified` and `windowOpened`.

Section 35.4 Using Menus with Frames

- Menus neatly organize commands in a GUI. In Swing GUIs, menus can be attached only to objects of classes with method `setJMenuBar` (p. 7).
- A `JMenuBar` (p. 7) is a container for menus. A `JMenuItem` appears in a menu. A `JMenu` (p. 7) contains menu items and can be added to a `JMenuBar` or to other `JMenus` as submenus.
- When a menu is clicked, it expands to show its list of menu items.
- When a `JCheckBoxMenuItem` (p. 8) is selected, a check appears to the left of the menu item. When the `JCheckBoxMenuItem` is selected again, the check is removed.
- In a `ButtonGroup`, only one `JRadioButtonMenuItem` (p. 8) can be selected at a time.
- `AbstractButton` method `setMnemonic` (p. 13) specifies the mnemonic (p. 8) for a button. Mnemonic characters are normally displayed with an underline.
- A modal dialog box (p. 13) does not allow access to any other window in the application until the dialog is dismissed. The dialogs displayed with class `JOptionPane` are modal dialogs. Class `JDialog` (p. 13) can be used to create your own modal or nonmodal dialogs.

Section 22.5 `JPopupMenu`

- Context-sensitive pop-up menus (p. 15) are created with class `JPopupMenu`. The pop-up trigger event occurs normally when the user presses and releases the right mouse button. `MouseEvent` method `isPopupTrigger` (p. 18) returns `true` if the pop-up trigger event occurred.
- `JPopupMenu` method `show` (p. 18) displays a `JPopupMenu`. The first argument specifies the origin component, which helps determine where the `JPopupMenu` will appear. The last two arguments are the coordinates from the origin component's upper-left corner, at which the `JPopupMenu` appears.

Section 35.6 Pluggable Look-and-Feel

- Class `UIManager.LookAndFeelInfo` (p. 19) maintains information about a look-and-feel.
- `UIManager` (p. 19) `static` method `getInstalledLookAndFeels` (p. 19) returns an array of `UIManager.LookAndFeelInfo` objects that describe the available look-and-feels.
- `UIManager` `static` method `setLookAndFeel` (p. 22) changes the look-and-feel. `SwingUtilities` (p. 22) `static` method `updateComponentTreeUI` (p. 22) changes the look-and-feel of every component attached to its `Component` argument to the new look-and-feel.

Section 22.7 `JDesktopPane` and `JInternalFrame`

- Many of today's applications use a multiple-document interface (MDI; p. 23) to manage several open documents that are being processed in parallel. Swing's `JDesktopPane` (p. 23) and `JInternalFrame` (p. 23) classes provide support for creating multiple-document interfaces.

Section 22.8 `JTabbedPane`

- A `JTabbedPane` (p. 26) arranges GUI components into layers, of which only one is visible at a time. Users access each layer by clicking its tab.

Section 22.9 `BoxLayout` Layout Manager

- `BoxLayout`)p. 939) arranges GUI components left-to-right or top-to-bottom in a container.
- Class `Box` represents a container with `BoxLayout` as its default layout manager and provides `static` methods to create a `Box` with a horizontal or vertical `BoxLayout`.

Section 22.10 `GridBagLayout` Layout Manager

- `GridBagLayout` (p. 32) is similar to `GridLayout`, but each component size can vary.
- A `GridBagConstraints` object (p. 33) specifies how a component is placed in a `GridBagLayout`.

Self-Review Exercises

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

a) The _____ class is used to create a menu object.
b) The _____ method of class `JMenu` places a separator bar in a menu.
c) `JSlider` events are handled by the _____ method of interface _____.
d) The `GridBagConstraints` instance variable _____ is set to `CENTER` by default.

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

a) When the programmer creates a `JFrame`, a minimum of one menu must be created and added to the `JFrame`.
b) The variable `fill` belongs to the `GridBagLayout` class.
c) Drawing on a GUI component is performed with respect to the (0, 0) upper-left corner coordinate of the component.
d) The default layout for a `Box` is `BoxLayout`.

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

a)
```
JMenubar b;
```
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

35.1 a) `JMenu`. b) `addSeparator`. c) `stateChanged`, `ChangeListener`. d) `anchor`.

35.2 a) False. A `JFrame` does not require any menus.

b) False. The variable `fill` belongs to the `GridBagConstraints` class.
c) True.
d) True.

35.3 a) `JMenubar` should be `JMenuBar`.
b) The first argument to the constructor should be `SwingConstants.HORIZONTAL` or `SwingConstants.VERTICAL`, and the keyword `new` must be used after the `=` operator. Also, the minimum value should be less than the maximum and the initial value should be in range.
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 establish the size of the window.

Exercises

35.4 *(Fill-in-the-Blanks)* Fill in the blanks in each of the following statements:
a) A `JMenuItem` that is a `JMenu` is called a(n) _____.
b) Method _____ attaches a `JMenuBar` to a `JFrame`.
c) Container class _____ has a default `BoxLayout`.
d) A(n) _____ manages a set of child windows declared with class `JInternalFrame`.

35.5 *(True or False)* 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) `BoxLayout` is the default layout manager for a `JFrame`.
c) `JApplets` can contain menus.

35.6 *(Find the Code Errors)* 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(new GridbagLayout());
```

35.7 *(Display a Circle and Its Attributes)* 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 × *radius*, *area* = π × *radius*^{2}, *circumference* = 2 × π × *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`.

35.8 *(Using a `JSlider`)* Enhance the program in Exercise 35.7 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 from Exercise 35.7. All drawing should be done on a subclass of `JPanel`, and the results of the calculations should be displayed in a read-only `JTextArea`.

35.9 *(Varying `weightx` and `weighty`)* Explore the effects of varying the `weightx` and `weighty` values of the program in Fig. 35.21. What happens when a slot has a nonzero weight but is not allowed to fill the whole area (i.e., the `fill` value is not `BOTH`)?

35.10 *(Synchronizing a `JSlider` and a `JTextField`)* 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. Changing the value in the `JTextField` should also update the `JSlider`. 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.]

35.11 ***(Creating a Color Chooser)*** Declare 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 these 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. Use your new GUI component as part of an application that displays the current `Color` value by drawing a filled rectangle.

35.12 ***(Creating a Color Chooser: Modification)*** Modify the `MyColorChooser` class of Exercise 35.11 to allow the user to enter 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.

35.13 ***(Creating a Color Chooser: Modification)*** Modify the application in Exercise 35.12 to draw the current color as a rectangle on an instance of a subclass of `JPanel` which provides its own `paintComponent` method to draw the rectangle and provides *set* methods to set the red, green and blue values for the current color. When any *set* method is invoked, the drawing panel should automatically `repaint` itself.

35.14 ***(Drawing Application)*** Modify the application in Exercise 35.13 to allow the user to drag the mouse across the drawing panel (a subclass of `JPanel`) to draw a shape in the current color. Enable the user to choose what shape to draw.

35.15 ***(Drawing Application Modification)*** Modify the application in Exercise 35.14 to allow the user 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. 35.5.

35.16 ***(Complete Drawing Application)*** Using the techniques developed in this chapter and Chapter 26, create a complete drawing application. The program should use the GUI components from Chapter 26 and this chapter 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. 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.