An actor entering through the door, you’ve got nothing. But if he enters through the window, you’ve got a situation.
—Billy Wilder

…the force of events wakes slumberous talents.
—Edward Hoagland

You and I would see more interesting photography if they would stop worrying, and instead, apply horse-sense to the problem of recording the look and feel of their own era.
—Jessie Tarbox Beals

Objectives

In this chapter you’ll learn:

■ To create and manipulate sliders, menus, pop-up menus and windows.

■ To programatically change the look-and-feel of a GUI, using Swing’s pluggable look-and-feel.

■ To create a multiple-document interface with JDesktopPane and JInternalFrame.

■ To use additional layout managers.
25.1 **Introduction**

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

We begin our discussion with sliders that enable you to select from a range of integer values. You’ll learn to use menus that enable the user to effectively perform tasks in the program. 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 14), a look-and-feel that simulates **Motif** (a popular UNIX look-and-feel) and one that simulates Microsoft’s Windows look-and-feel.

Many of today’s applications use a multiple-document interface (MDI)—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. The chapter finishes with a series of examples discussing additional layout managers for organizing graphical user interfaces.

Swing is a large and complex topic. There are many more GUI components and capabilities than can be presented here. Several more Swing GUI components are introduced in the remaining chapters of this book as they’re needed.

### 25.2 **JSlider**

**JSliders** enable a user to select from a range of integer values. Class JSlider inherits from JComponent. Figure 25.1 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. 25.1 | JSlider component with horizontal orientation.**
Most Swing GUI components support user interactions through the mouse and the keyboard. For example, if a JSlider has the focus (i.e., it's the currently selected GUI component in the user interface), the left arrow key and right arrow key cause the thumb of the JSlider to decrease or increase by 1, respectively. The down arrow key and up arrow key also cause the thumb 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.

JSide rs 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. 25.2-25.4 allows the user to size a circle drawn on a subclass of JPanel called OvalPanel (Fig. 25.2). 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.

```java
// Fig. 25.2: OvalPanel.java
// A customized JPanel class.
import java.awt.Graphics;
import java.awt.Dimension;
import javax.swing.JPanel;

public class OvalPanel extends JPanel {

    private int diameter = 10; // default diameter of 10

    // draw an oval of the specified diameter
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.fillOval(10, 10, diameter, diameter); // draw circle
    } // end method paintComponent

    // validate and set diameter, then repaint
    public void setDiameter(int newDiameter) {
        // if diameter invalid, default to 10
        diameter = (newDiameter >= 0 ? newDiameter : 10);
    }

    // Fig. 25.2 | JPanel subclass for drawing circles of a specified diameter. (Part 1 of 2.)
```
25.2 JSlider

```java
24    repaint(); // repaint panel
25  } // end method setDiameter
26
27  // used by layout manager to determine preferred size
28  public Dimension getPreferredSize()
29  {
30      return new Dimension( 200, 200 );
31  } // end method getPreferredSize
32
33  // used by layout manager to determine minimum size
34  public Dimension getMinimumSize()
35  {
36      return getPreferredSize();
37  } // end method getMinimumSize
38 } // end class OvalPanel
```

Fig. 25.2 | JPanel subclass for drawing circles of a specified diameter. (Part 2 of 2.)

```java
1  // Fig. 25.3: 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.SwingConstants;
7  import javax.swing.JSlider;
8  import javax.swing.event.ChangeListener;
9  import javax.swing.event.ChangeEvent;
10
11  public class SliderFrame extends JFrame
12  {
13      private JSlider diameterJSlider; // slider to select diameter
14      private 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 ); // set background to yellow
23
24          // set up JSlider to control diameter value
25          diameterJSlider =
26              new JSlider( SwingConstants.HORIZONTAL, 0, 200, 10 );
27          diameterJSlider.setMajorTickSpacing( 10 ); // create tick every 10
28          diameterJSlider.setPaintTicks( true ); // paint ticks on slider
29
30          // register JSlider event listener
31          diameterJSlider.addActionListener(
32              new ChangeListener() // anonymous inner class
33              {
34                  ...
35              }
36          );
```

Fig. 25.3 | JSlider value used to determine the diameter of a circle. (Part 1 of 2.)
Class OvalPanel (Fig. 25.2) 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. Section 24.3 introduced getPreferredSize and getMinimumSize, which are used by some layout managers to determine the size of a component.
Class SliderFrame (Fig. 25.3) 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 diameterSlider to control the diameter of the circle drawn on the OvalPanel. The JSlider constructor takes four arguments. The first argument specifies the orientation of diameterSlider, 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 on-line documentation (download.oracle.com/javase/6/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 aChangeListener to handle diameterSlider’s events. When method stateChanged (lines 36–39) 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.

25.3 Windows: Additional Notes

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 other UNIX applications on that platform.

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 java.awt.swing), which class JFrame implements, declares three constants—DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE and HIDE_ON_CLOSE (the default)—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.

**Performance Tip 25.1**

A window is an expensive system resource. Return it to the system by calling its `dispose` method when the window is no longer needed.

By default, a window is not displayed on the screen until the program invokes the window’s `setVisible` method (inherited from class `java.awt.Component`) with a true argument. A window’s size should be set with a call to method `setSize` (inherited from class `java.awt.Component`). The position of a window when it appears on the screen is specified with method `setLocation` (inherited from class `java.awt.Component`).

**Common Programming Error 25.1**

Forgetting to call method `setVisible` on a window is a runtime logic error—the window is not displayed.

**Common Programming Error 25.2**

Forgetting to call the `setSize` method on a window is a runtime logic error—only the title bar appears.

When the user manipulates the window, this action generates window events. Event listeners are registered for window events with `Window` method `addWindowListener`. Interface `WindowListener` 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 window from being minimized), `windowIconified` (called when the user minimizes a window) and `windowOpened` (called when a program first displays a window on the screen).

### 25.4 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 25.1**

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 `JCheckBoxMenuItem`s. When multiple `JRadioButtonMenuItem`s are maintained as part of a `ButtonGroup`, only one item in the group can be selected at a given time. When a `JRadioButtonMenuItem` is selected, a filled circle appears to the left of the menu item. When another `JRadioButtonMenuItem` is selected, the filled circle of the previously selected menu item is removed.

Using Menus in an Application

Figures 25.5–25.6 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. 25.5) 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).

```java
// Fig. 25.5: MenuFrame.java
// Demonstrating menus.
import java.awt.Color;
import java.awt.Font;
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JLabel;
import javax.swing.SwingConstants;
import javax.swing.ButtonGroup;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JMenu;
import javax.swing.SwingConstants;
import javax.swing.JButton;
import javax.swing.JMenu;
import javax.swing.JMenuBar;

public class MenuFrame extends JFrame {
    {
```
private final Color[] colorValues =
    { Color.BLACK, Color.BLUE, Color.RED, Color.GREEN };

private JRadioButtonMenuItem[] colorItems; // color menu items
private JRadioButtonMenuItem[] fonts; // font menu items
private JCheckBoxMenuItem[] styleItems; // font style menu items
private JLabel displayLabel; // displays sample text
private ButtonGroup fontButtonGroup; // manages font menu items
private ButtonGroup colorButtonGroup; // manages color menu items
private int style; // used to create style for font

// no-argument constructor set up GUI
public MenuFrame()
{
    super( "Using JMenus" );

    JMenu fileMenu = new JMenu( "File" ); // create file menu
    fileMenu.setMnemonic( 'F' ); // set mnemonic to F

    // create About... menu item
    JMenuItem aboutItem = new JMenuItem( "About..." );
    aboutItem.setMnemonic( 'A' ); // set mnemonic to A
    fileMenu.add( aboutItem ); // add about item to file menu
    aboutItem.addActionListener( new ActionListener() // anonymous inner class
    {
        public void actionPerformed( ActionEvent event )
        {
            JOptionPane.showMessageDialog( MenuFrame.this,
                "This is an example of using menus", "About", JOptionPane.PLAIN_MESSAGE );
        }
    } // end anonymous inner class
    ); // end call to addActionListener

    JMenuItem exitItem = new JMenuItem( "Exit" ); // create exit item
    exitItem.setMnemonic( 'x' ); // set mnemonic to x
    fileMenu.add( exitItem ); // add exit item to file menu
    exitItem.addActionListener( new ActionListener() // anonymous inner class
    {
        public void actionPerformed( ActionEvent event )
        {
            System.exit( 0 ); // exit application
        }
    } // end anonymous inner class
    ); // end call to addActionListener

    JMenuBar bar = new JMenuBar(); // create menu bar
    setJMenuBar( bar ); // add menu bar to application

Fig. 25.5 | JMenus and mnemonics. (Part 2 of 5.)
Using Menus with Frames

```java
// array listing string colors
String[] colors = { "Black", "Blue", "Red", "Green" };

// create radio button menu items for colors
ItemHandler itemHandler = new ItemHandler();
// handler for colors
for (int count = 0; count < colors.length; count++) {
    colorItems[count] = new JRadioButtonMenuItem(colors[count]); // create item
    colorMenu.add(colorItems[count]); // add item to color menu
    colorButtonGroup.add(colorItems[count]); // add to group
    colorItems[count].addActionListener(itemHandler);
} // end for

// select first Color item
colorItems[0].setSelected(true);

// array listing font names
String[] fontNames = { "Serif", "Monospaced", "SansSerif" };

// create radio button menu items for font names
fonts = new JRadioButtonMenuItem[fontNames.length];
fontButtonGroup = new ButtonGroup(); // manages font names
for (int count = 0; count < fontNames.length; count++) {
    fonts[count] = new JRadioButtonMenuItem(fontNames[count]); // create item
    fontMenu.add(fonts[count]); // add item to font menu
    fontButtonGroup.add(fonts[count]); // add to button group
    fonts[count].addActionListener(itemHandler); // add handler
} // end for

// select first Font item
fonts[0].setSelected(true);
```

**Fig. 25.5** JMenus and mnemonics. (Part 3 of 5.)
styleItems = new JCheckBoxMenuItem[styleNames.length];
StyleHandler styleHandler = new StyleHandler(); // style handler

// create style checkbox menu items
for (int count = 0; count < styleNames.length; count++)
{
    styleItems[count] = new JCheckBoxMenuItem(styleNames[count]); // for style
    fontMenu.add(styleItems[count]); // add to font menu
    styleItems[count].addItemListener(styleHandler); // handler
}

// set up label to display text
displayJLabel = new JLabel("Sample Text", SwingConstants.CENTER);
displayJLabel.setForeground(colorValues[0]);
displayJLabel.setFont(new Font("Serif", Font.PLAIN, 72));

getContentPane().setBackground(Color.CYAN); // set background
add(displayJLabel, BorderLayout.CENTER); // add displayJLabel

// inner class to handle action events from menu items
private class ItemHandler implements ActionListener
{
    // process color and font selections
    public void actionPerformed(ActionEvent event)
    {
        // process color selection
        for (int count = 0; count < colorItems.length; count++)
        {
            if (colorItems[count].isSelected())
            {
                displayJLabel.setForeground(colorValues[count]);
                break;
            }
        }
        // end if
        // end for

        // process font selection
        for (int count = 0; count < fonts.length; count++)
        {
            if (event.getSource() == fonts[count])
            {
                displayJLabel.setFont(new Font(fonts[count].getText(), style, 72));
            }
        }
        // end if
        // end for

        repaint(); // redraw application
    }
    // end method actionPerformed
}
// end class ItemHandler

Fig. 25.5 | JMenus and mnemonics. (Part 4 of 5.)
25.4 Using Menus with Frames

// inner class to handle item events from checkbox menu items
private class StyleHandler implements ItemListener
{
    // process font style selections
    public void itemStateChanged( ItemEvent e )
    {
        String name = displayJLabel.getFont().getName(); // current Font
        Font font; // new font based on user selections
        // determine which items are checked and create Font
        if ( styleItems[ 0 ].isSelected() &&
             styleItems[ 1 ].isSelected() )
            font = new Font( name, Font.BOLD + Font.ITALIC, 72 );
        else if ( styleItems[ 0 ].isSelected() )
            font = new Font( name, Font.BOLD, 72 );
        else if ( styleItems[ 1 ].isSelected() )
            font = new Font( name, Font.ITALIC, 72 );
        else
            font = new Font( name, Font.PLAIN, 72 );
        displayJLabel.setFont( font );
        repaint(); // redraw application
    } // end method itemStateChanged
} // end class StyleHandler
} // end class MenuFrame

Fig. 25.5 | JMenus and mnemonics. (Part 5 of 5.)

Fig. 25.6 | Test class for MenuFrame. (Part 1 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. 25.6.)

Look-and-Feel Observation 25.2
Mnemonics provide quick access to menu commands and button commands through the keyboard.

Look-and-Feel Observation 25.3
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 JMenuItem 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 47–56 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 (.).

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 dis-
played with class 

Class 

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 exit-Item.

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.

**Common Programming Error 25.3**

Forgetting to set the menu bar with JFrame method setJMenuBar prevents the menu bar from displaying in the JFrame.

**Look-and-Feel Observation 25.4**

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

**Setting Up the Format Menu**

Lines 78–79 create menu formatMenu and set its mnemonic to r. (F is not used because that is the File menu’s mnemonic.)

Lines 84–85 create menu colorMenu (this will be a submenu in the Format menu) 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 will ensure that only one of the menu items in the Color submenu 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 for statement 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 25.5**

A submenu is created by adding a menu as a menu item in another menu. When the mouse is positioned over a submenu (or the submenu’s mnemonic is pressed), the submenu expands to show its menu items.

**Look-and-Feel Observation 25.6**

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

**Look-and-Feel Observation 25.7**

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

Lines 108–126 create the Font submenu and several JRadioButtonMenuItem and select the first element of JRadioButtonMenuItem array fonts. Line 129 creates a JCheck-
BoxMenuItem 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 184–206) 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 157–180) 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 162 uses AbstractButton method isSelected to determine the selected JRadioButtonMenuItem. The if condition at line 172 invokes the event object’s getSource method to get a reference to the JRadioButtonMenuItem that generated the event. Line 175 invokes AbstractButton method getText to obtain the name of the font from the menu item.

StyleHandler method itemStateChanged (lines 187–205) is called if the user selects a JCheckBoxMenuItem in the fontMenu. Lines 193–201 determine which JCheckBoxMenuItems are selected and use their combined state to determine the new font style.

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

Look-and-Feel Observation 25.8
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. 25.7–25.8 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. 25.7, lines 21–69) creates an instance of class ItemHandler (declared in lines 72–87) 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 JRadioButton-
Menu Item at a time and registers its ActionListener (line 38). Line 41 sets the initial background to white by invoking method setBackground.

```java
// Fig. 25.7: PopupFrame.java  
// Demonstrating JPopupMenu.  
import java.awt.Color;  
import java.awt.event.MouseAdapter;  
import java.awt.event.MouseEvent;  
import java.awt.event.ActionListener;  
import java.awt.event.ActionEvent;  
import javax.swing.JFrame;  
import javax.swing.JRadioButtonMenuItem;  
import javax.swing.JPopupMenu;  
import javax.swing.ButtonGroup;  
  
public class PopupFrame extends JFrame  
{  
  private JRadioButtonMenuItem[] items; // holds items for colors  
  private final Color[] colorValues =  
  { Color.BLUE, Color.YELLOW, Color.RED }; // colors to be used  
  private JPopupMenu popupMenu; // allows user to select color  
  // no-argument constructor sets up GUI  
  public PopupFrame()  
  {  
    super( "Using JPopupMenu" );  
    ItemHandler handler = new ItemHandler(); // handler for menu items  
    String[] colors = { "Blue", "Yellow", "Red" }; // array of colors  
    ButtonGroup colorGroup = new ButtonGroup(); // manages color items  
    popupMenu = new JPopupMenu(); // create pop-up menu  
    items = new JRadioButtonMenuItem[ colors.length ]; // color items  
    // construct menu item, add to pop-up menu, enable event handling  
    for ( int count = 0; count < items.length; count++ )  
    {  
      items[ count ] = new JRadioButtonMenuItem( colors[ count ] );  
      popupMenu.add( items[ count ] ); // add item to pop-up menu  
      colorGroup.add( items[ count ] ); // add item to button group  
      items[ count ].addActionListener( handler ); // add handler  
    } // end for  
    setBackground( Color.WHITE ); // set background to white  
    // declare a MouseListener for the window to display pop-up menu  
    addMouseListener(  
      new MouseAdapter() // anonymous inner class  
      {  
        // handle mouse press event  
        public void mousePressed( MouseEvent event )  
        {  
```

**Fig. 25.7** | **JPopupMenu** for selecting colors. (Part 1 of 2.)
checkForTriggerEvent(event); // check for trigger
}

// handle mouse release event
public void mouseReleased(MouseEvent event)
{
    checkForTriggerEvent(event); // check for trigger
}

// determine whether event should trigger pop-up menu
private void checkForTriggerEvent(MouseEvent event)
{
    if (event.isPopupTrigger()) {
        popupMenu.show(event.getComponent(), event.getX(), event.getY());
    }
}

// end method mousePressed

// end method mouseReleased

// determine whether event should trigger pop-up menu
private void checkForTriggerEvent(MouseEvent event)
{
    if (event.isPopupTrigger()) {
        popupMenu.show(event.getComponent(), event.getX(), event.getY());
    }
}

// end anonymous inner class
}
// end PopupFrame constructor

// private inner class to handle menu item events
private class ItemHandler implements ActionListener
{
    // process menu item selections
    public void actionPerformed(ActionEvent event)
    {
        // determine which menu item was selected
        for (int i = 0; i < items.length; i++)
        {
            if (event.getSource() == items[i]) {
                getContentPane().setBackground(colorValues[i]);
                return;
            }
        }
    }
}
// end class ItemHandler

// Fig. 25.8: PopupTest.java
// Testing PopupFrame.
import javax.swing.JFrame;
public class PopupTest
{
    public static void main(String[] args)
    {
        PopupFrame popupFrame = new PopupFrame(); // create PopupFrame
        popupFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
// Fig. 25.8 | Test class for PopupFrame. (Part 1 of 2.)
### 25.6 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 Mac OS X looks like other Mac OS X applications. One running on Microsoft Windows looks like other Windows applications. 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 25.1

GUI components look different on different platforms and may require different amounts of space to display. This could change their layout and alignments.

#### Look-and-Feel Observation 25.9

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 75–86) determines which `JRadioButtonMenuItem` the user selected and sets the background color of the window’s content pane.

---

#### Example Code

```java
popupFrame.setSize(300, 200); // set frame size
popupFrame.setVisible(true); // display frame
```

---

![Image](image.png)
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. Recent versions of Java SE 6 and the upcoming Java SE 7 include the \textit{Nimbus} look-and-feel that we discussed in Section 14.2. Earlier versions of Java used the \textit{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 25.9–25.10 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.

```java
// Fig. 25.9: LookAndFeelFrame.java
// Changing the look-and-feel.
import java.awt.GridLayout;
import java.awt.BorderLayout;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JRadioButton;
import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JComboBox;
import javax.swing.JPanel;
import javax.swing.SwingConstants;

public classLookAndFeelFrame extends JFrame{

private String[] lookNames; // names of look feels
private JRadioButton[] radio; // radio buttons to select look-and-feel
private ButtonGroup group; // group for radio buttons
private JButton button; // displays look of button
private JLabel label; // displays look of label
private JComboBox comboBox; // displays look of combo box

// set up GUI
public LookAndFeelFrame()
{

```

Portability Tip 25.2

\textit{GUI} components on different platforms have different default functionality (e.g., some platforms allow a button with the focus to be “pressed” with the space bar, and some don’t).
```java
super( "Look and Feel Demo" );

// get installed look-and-feel information
looks = UIManager.getInstalledLookAndFeels();
lookNames = new String[ looks.length ];

// get names of installed look-and-feels
for ( int i = 0; i < looks.length; i++ )
    lookNames[i] = looks[i].getName();

JPanel northPanel = new JPanel(); // create north panel
northPanel.setLayout( new GridLayout( 3, 1, 0, 5 ) );

label = new JLabel( "This is a " + lookNames[0] + " look-and-feel",
    SwingConstants.CENTER ); // create label
northPanel.add( label ); // add label to panel

button = new JButton( "JButton" ); // create button
northPanel.add( button ); // add button to panel

comboBox = new JComboBox( lookNames ); // create combobox
northPanel.add( comboBox ); // add combobox to panel

// create array for radio buttons
radio = new JRadioButton[ looks.length ];

JPanel southPanel = new JPanel(); // create south panel
southPanel.setLayout( new GridLayout( rows, 3 ) );
group = new ButtonGroup(); // button group for looks-and-feels
ItemHandler handler = new ItemHandler(); // look-and-feel handler

for ( int count = 0; count < radio.length; count++ )
{
    radio[ count ] = new JRadioButton( lookNames[ count ] );
    radio[ count ].addItemListener( handler ); // add handler
group.add( radio[ count ] ); // add radio button to group
    southPanel.add( radio[ count ] ); // add radio button to panel
}

add( northPanel, BorderLayout.NORTH ); // add north panel
add( southPanel, BorderLayout.SOUTH ); // add south panel

radio[ 0 ].setSelected( true ); // set default selection
}
```

Fig. 25.9 | Look-and-feel of a Swing-based GUI. (Part 2 of 3.)
try // change look-and-feel
{
    // set look-and-feel for this application
    UIManager.setLookAndFeel( looks[ value ].getClassName() );
    // update components in this application
    SwingUtilities.updateComponentTreeUI( this );
} // end try
catch ( Exception exception )
{
    exception.printStackTrace();
} // end catch
} // end method changeTheLookAndFeel

// private inner class to handle radio button events
private class ItemHandler implements ItemListener
{
    // process user's look-and-feel selection
    public void itemStateChanged( ItemEvent event )
    {
        for ( int count = 0; count < radio.length; count++ )
        {
            if ( radio[ count ].isSelected() )
            {
                label.setText( String.format( "This is a %s look-and-feel", lookNames[ count ] ) );
                comboBox.setSelectedIndex( count ); // set combobox index
                changeTheLookAndFeel( count ); // change look-and-feel
            } // end if
        } // end for
    } // end method itemStateChanged
} // end private inner class ItemHandler
} // end class LookAndFeelFrame

Fig. 25.10: LookAndFeelDemo.java
Fig. 25.9 | Look-and-feel of a Swing-based GUI. (Part 3 of 3.)
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 classLookAndFeelInfo (a public static class) that maintains information about a look-and-feel. Line 20 declares an array of type UIManager.LookAndFeelInfo (note the syntax used to identify the static inner classLookAndFeelInfo). Line 34 uses UIManager static methodgetInstalledLookAndFeels to get the array of UIManager.LookAndFeelInfo objects that describe each look-and-feel available on your system.

Performance Tip 25.2

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.

Our utility method changeTheLookAndFeel (lines 81–95) is called by the event handler for the JRadioButton at the bottom of the user interface. The event handler (declared in private inner class ItemHandler at lines 98–114) 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. Method getClassName 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 class is not already loaded, it will be loaded as part of the call to setLookAndFeel. Line 89 invokes 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.

25.7 JDesktopPane and JInternalFrame

Many of today’s applications use a multiple-document interface (MDI)—a main window (called the parent window) containing other windows (called child windows), to manage several open documents that are being processed in parallel. For example, many e-mail programs allow you to have several 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. 25.11–25.12 demonstrates Swing’s JDesktopPane and JInternalFrame classes for implementing multiple-document interfaces.

```java
// Fig. 25.11: DesktopFrame.java
// Demonstrating JDesktopPane.
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.util.Random;
import javax.swing.JFrame;
import javax.swing.JMenuBar;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import javax.swing.JPanel;
import javax.swing.ImageIcon;

public class DesktopFrame extends JFrame {
    private JDesktopPane theDesktop;

    // set up GUI
    public DesktopFrame() {
        super("Using a JDesktopPane");
        JMenu bar = new JMenuBar(); // create menu bar
        JMenu addMenu = new JMenu("Add"); // create Add menu
        JMenuItem newFrame = new JMenuItem("Internal Frame");
        addMenu.add(newFrame); // add new frame item to Add menu
        bar.add(addMenu); // add Add menu to menu bar
    }
}
```

Fig. 25.11 | Multiple-document interface. (Part 1 of 3.)
25.7 **JDesktopPane and JInternalFrame**

```java
setJMenuBar( bar ); // set menu bar for this application

JDesktopPane theDesktop = new JDesktopPane(); // create desktop pane
add( theDesktop ); // add desktop pane to frame

// set up listener for newFrame menu item
newFrame.addActionListener(
    new ActionListener() // anonymous inner class
    {
        // display new internal window
        public void actionPerformed( ActionEvent event )
        {
            // create internal frame
            JInternalFrame frame = new JInternalFrame("Internal Frame", true, true, true, true);
            MyJPanel panel = new MyJPanel(); // create new panel
            frame.add( panel, BorderLayout.CENTER ); // add panel
            frame.pack(); // set internal frame to size of contents
            theDesktop.add( frame ); // attach internal frame
            frame.setVisible( true ); // show internal frame
        } // end method actionPerformed
    } // end anonymous inner class
) ; // end call to addActionListener
} // end DesktopFrame constructor

// class to display an ImageIcon on a panel
class MyJPanel extends JPanel
{
    private static Random generator = new Random();
    private ImageIcon picture; // image to be displayed
    private final static String[] images = {"yellowflowers.png", "purpleflowers.png", "redflowers.png", "redflowers2.png", "lavenderflowers.png"};

    // load image
    public MyJPanel()
    {
        int randomNumber = generator.nextInt( images.length );
        picture = new ImageIcon( images[ randomNumber ] ); // set icon
    } // end MyJPanel constructor

    // display ImageIcon on panel
    public void paintComponent( Graphics g )
    {
        super.paintComponent( g );
        picture.paintIcon( this, g, 0, 0 ); // display icon
    } // end method paintComponent
```

*Fig. 25.11* | Multiple-document interface. (Part 2 of 3.)
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.

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 44–56) 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 JFrame and JApplets, a JInternalFrame has a content pane to which GUI components can be attached. Line 50 (Fig. 25.11) creates an instance of our class MyJPanel (declared at lines 63–91) that is added to the JInternalFrame at line 51.

Fig. 25.12 | Test class for DesktopFrame. (Part 1 of 2.)

```java
// Fig. 25.12: DesktopTest.java
// Demonstrating JDesktopPane.
import javax.swing.JFrame;

public class DesktopTest {
    public static void main( String[] args ) {
        DesktopFrame desktopFrame = new DesktopFrame();
        desktopFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        desktopFrame.setSize( 600, 480 ); // set frame size
        desktopFrame.setVisible( true ); // display frame
    } // end main
} // end class DesktopTest
```
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 86–90) 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:

download.oracle.com/javase/6/docs/api/javax/swing/JInternalFrame.html
download.oracle.com/javase/6/docs/api/javax/swing/JDesktopPane.html
25.8 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. 25.13–25.14 creates one tabbed pane with three tabs. Each tab displays one of the JPanels—panel1, panel2 or panel3.

```java
// Fig. 25.13: JTabbedPaneFrame.java
// Demonstrating JTabbedPane.
import java.awt.BorderLayout;
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JButton;
import javax.swing.SwingConstants;

public class JTabbedPaneFrame extends JFrame {
    public JTabbedPaneFrame()
    {
        super( "JTabbedPane Demo " );
        JTabbedPane tabbedPane = new JTabbedPane(); // create JTabbedPane

        JLabel label1 = new JLabel( "panel one", SwingConstants.CENTER );
        JPanel panel1 = new JPanel(); // create first panel
        panel1.add( label1 ); // add label to panel
        tabbedPane.addTab( "Tab One", null, panel1, "First Panel" );

        JLabel label2 = new JLabel( "panel two", SwingConstants.CENTER );
        JPanel panel2 = new JPanel(); // create second panel
        panel2.setBackground( Color.YELLOW ); // set background to yellow
        panel2.add( label2 ); // add label to panel
        tabbedPane.addTab( "Tab Two", null, panel2, "Second Panel" );

        JLabel label3 = new JLabel( "panel three" );
        JPanel panel3 = new JPanel(); // create third panel
        panel3.setLayout( new BorderLayout() ); // use borderlayout
        panel3.add( new JButton( "North" ), BorderLayout.NORTH );
        panel3.add( new JButton( "West" ), BorderLayout.WEST );
        panel3.add( new JButton( "East" ), BorderLayout.EAST );
    }
}
```

Fig. 25.13 | JTabbedPane used to organize GUI components. (Part 1 of 2.)
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.
25.9 Layout Managers: BoxLayout and GridBagLayout

In Chapter 14, we introduced three layout managers—FlowLayout, BorderLayout and GridLayout. This section presents two additional layout managers (summarized in Fig. 25.15). We discuss them in the examples that follow. We also discuss the extremely flexible GroupLayout in Appendix I.

<table>
<thead>
<tr>
<th>Layout manager</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoxLayout</td>
<td>A layout manager that allows GUI components to be arranged left-to-right or top-to-bottom in a container. Class Box declares a container with BoxLayout as its default layout manager and provides static methods to create a Box with a horizontal or vertical BoxLayout.</td>
</tr>
<tr>
<td>GridBagLayout</td>
<td>A layout manager similar to GridLayout, but the components can vary in size and can be added in any order.</td>
</tr>
</tbody>
</table>

Fig. 25.15 | Additional layout managers.

**BoxLayout Layout Manager**

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. 25.16–25.17 demonstrates BoxLayout and the container class Box that uses BoxLayout as its default layout manager.

```
// Fig. 25.16: BoxLayoutFrame.java
// Demonstrating BoxLayout.
import java.awt.Dimension;
import javax.swing.JFrame;
import javax.swing.Box;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JTabbedPane;

public class BoxLayoutFrame extends JFrame
{
    // set up GUI
    public BoxLayoutFrame()
    {
        super( "Demonstrating BoxLayout" );

        // create Box containers with BoxLayout
        Box horizontal1 = Box.createHorizontalBox();
        Box vertical1 = Box.createVerticalBox();
        Box horizontal2 = Box.createHorizontalBox();
        Box vertical2 = Box.createVerticalBox();

        final int SIZE = 3; // number of buttons on each Box
```

Fig. 25.16 | BoxLayout layout manager. (Part 1 of 2.)
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.

The loop at lines 27–28 adds three `JButton`s to `$horizontal1$`. The for statement at lines 31–35 adds three `JButton`s 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.

The `for` statement at lines 38–42 adds three `JButtons` to `horizontal12`. Before adding
each button, line 40 adds `horizontal glue` to the container with static Box method `cre-
atHorizontalGlue`. 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.

The `for` statement at lines 45–49 adds three `JButtons` to `vertical12`. Before each
button is added, line 47 adds a `rigid area` to the container with static Box method `cre-
atRigidArea`. 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.

Lines 52–53 create a `JPanel` object and set its layout to a BoxLayout in the conven-
tional manner, using Container method `setLayout`. The BoxLayout constructor receives
a reference to the container for which it controls the layout and a constant indicating
whether the layout is horizontal (`BoxLayout.X_AXIS`) or vertical (`BoxLayout.Y_AXIS`).

The `for` statement at lines 55–59 adds three `JButtons` to `panel1`. Before adding each
button, line 57 adds a glue component to the container with static Box method `cre-
atGlue`. This component expands or contracts based on the size of the Box.

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.

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.

**GridLayout Layout Manager**

One of the most powerful predefined layout managers is `GridLayout` (in package `ja-
v.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 `GridLayout` 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 `GridLayout` layout manager can use the row and column numbers to
properly place the components in the grid. Figure 25.18 demonstrates drawing the lines
for the rows and columns over a GUI.

A `GridBagConstraints` object describes how a component is placed in a `Grid-
BagLayout`. Several `GridBagConstraints` fields are summarized in Fig. 25.19.

`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 `GridBag-
**Chapter 25  GUI Components: Part 2**

Constraints constants: **NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, NORTHWEST or CENTER**. The default value is CENTER.

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.

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. 25.18, the JComboBox (displaying “Iron”) has a gridx value of 1 and a gridy value of 2.
Variable `gridwidth` specifies the number of columns a component occupies. The JComboBox occupies two columns. Variable `gridheight` specifies the number of rows a component occupies. The JTextArea on the left side of Fig. 25.18 occupies three rows.

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.

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.

In Fig. 25.18, 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 25.20 shows the GUI of Fig. 25.18 with all weights set to zero.

![Fig. 25.20](image)

**Fig. 25.20** GridBagLayout with the weights set to zero.

The application in Figs. 25.21–25.22 uses the GridBagLayout layout manager to arrange the components of the GUI in Fig. 25.18. The application does nothing except demonstrate how to use GridBagLayout.

```java
// Fig. 25.21: GridBagFrame.java
// Demonstrating GridBagLayout.

import java.awt.Component;
import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
```

**Fig. 25.21** GridBagLayout layout manager. (Part 1 of 3.)
import javax.swing.JButton;
import javax.swing.JComboBox;

public class GridBagFrame extends JFrame {
    private GridBagLayout layout; // layout of this frame
    private GridBagConstraints constraints; // constraints of this layout

    // set up GUI
    public GridBagFrame() {
        super( "GridBagLayout" );
        layout = new GridBagLayout();
        setLayout( layout ); // set frame layout
        constraints = new GridBagConstraints(); // instantiate constraints

        // create GUI components
        JTextArea textArea1 = new JTextArea( "TextArea1", 5, 10 );
        JTextArea textArea2 = new JTextArea( "TextArea2", 2, 2 );
        String[] names = { "Iron", "Steel", "Brass" }; // can grow taller
        JComboBox comboBox = new JComboBox( names );
        JTextField textField = new JTextField( "TextField" );
        JButton button1 = new JButton( "Button 1" );
        JButton button2 = new JButton( "Button 2" );
        JButton button3 = new JButton( "Button 3" );

        // weightx and weighty for textArea1 are both 0: the default
        // anchor for all components is CENTER: the default
        constraints.fill = GridBagConstraints.BOTH;
        addComponent( textArea1, 0, 0, 1, 3 );

        // weightx and weighty for button1 are both 0: the default
        constraints.fill = GridBagConstraints.HORIZONTAL;
        addComponent( button1, 0, 1, 2, 1 );

        // weightx and weighty for comboBox are both 0: the default
        // fill is HORIZONTAL
        addComponent( comboBox, 2, 1, 2, 1 );

        // button2
        constraints.weightx = 1000; // can grow wider
        constraints.weighty = 1; // can grow taller
        addComponent( button2, 1, 1, 1 );

        // fill is BOTH for button3
        constraints.weightx = 0;
        constraints.weighty = 0;
        addComponent( button3, 1, 2, 1 );

        // weightx and weighty for comboBox are both 0: the default
        // fill is HORIZONTAL
        addComponent( comboBox, 2, 1, 2, 1 );

        // button2
        constraints.weightx = 1000; // can grow wider
        constraints.weighty = 1; // can grow taller
        addComponent( button2, 1, 1, 1 );

        // fill is BOTH for button3
        constraints.weightx = 0;
        constraints.weighty = 0;
        addComponent( button3, 1, 2, 1 );
    }
}

Fig. 25.21  |  GridBagLayout layout manager. (Part 2 of 3.)
Fig. 25.22 | Test class for GridBagFrame. (Part 1 of 2.)
The GUI contains three JButton, two JTextArea, 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.

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 JButton 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 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 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 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 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.

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'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. 25.23–25.24 arranges components in GridBagLayout, using these constants.

```java
// Fig. 25.23: GridBagFrame2.java
// Demonstrating GridBagConstraints constants.
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
import java.awt.Component;
import javax.swing.JFrame;
import javax.swing.JComboBox;
import javax.swing.JTextField;
import javax.swing.JList;
import javax.swing.JButton;

public class GridBagFrame2 extends JFrame
{
    private GridBagLayout layout; // layout of this frame
    private GridBagConstraints constraints; // constraints of this layout
}
```

Fig. 25.23  |  GridBagConstraints constants RELATIVE and REMAINDER (Part 1 of 3.)
// set up GUI
public GridBagFrame2()
{
    super("GridBagLayout");
    layout = new GridBagLayout();
    setLayout( layout ); // set frame layout
    constraints = new GridBagConstraints(); // instantiate constraints

    // create GUI components
    String[] metals = {"Copper", "Aluminum", "Silver"};
    JComboBox comboBox = new JComboBox( metals );

    JTextField textField = new JTextField("TextField");

    String[] fonts = {"Serif", "Monospaced"};
    JList list = new JList( fonts );

    String[] names = {"zero", "one", "two", "three", "four"};
    JButton[] buttons = new JButton[ names.length ];

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

    // define GUI component constraints for textField
    constraints.weightx = 1;
    constraints.weighty = 1;
    constraints.fill = GridBagConstraints.BOTH;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    addComponent( textField );

    // buttons[0] -- weightx and weighty are 1: fill is BOTH
    constraints.gridwidth = 1;
    addComponent( buttons[ 0 ] );

    // buttons[1] -- weightx and weighty are 1: fill is BOTH
    constraints.gridwidth = GridBagConstraints.RELATIVE;
    addComponent( buttons[ 1 ] );

    // buttons[2] -- weightx and weighty are 1: fill is BOTH
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    addComponent( buttons[ 2 ] );

    // comboBox -- weightx is 1: fill is BOTH
    constraints.weighty = 0;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    addComponent( comboBox );

    // buttons[3] -- weightx is 1: fill is BOTH
    constraints.weighty = 1;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    addComponent( buttons[ 3 ] );

    constraints.weighty = 0;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    addComponent( comboBox );

    // buttons[3] -- weightx is 1: fill is BOTH
    constraints.weighty = 1;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    addComponent( buttons[ 3 ] );

    constraints.weighty = 0;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    addComponent( comboBox );

    // buttons[3] -- weightx is 1: fill is BOTH
    constraints.weighty = 1;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    addComponent( buttons[ 3 ] );

    constraints.weighty = 0;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    addComponent( comboBox );

    // buttons[3] -- weightx is 1: fill is BOTH
    constraints.weighty = 1;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    addComponent( buttons[ 3 ] );

    constraints.weighty = 0;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    addComponent( comboBox );

    // buttons[3] -- weightx is 1: fill is BOTH
    constraints.weighty = 1;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    addComponent( buttons[ 3 ] );
25.9 Layout Managers: BoxLayout and 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 JButton.

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 com-
ponent 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.

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.

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.

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.

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.

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.

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.

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.

25.10 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. In the next chapter, you’ll learn about multithreading, which allows you to specify that an application should perform multiple tasks at once.

Summary

Section 25.2 JSlider

• JSliders 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. 1001). They also support snap-to ticks (p. 1001)—positioning the thumb (p. 1001) between two tick marks causes snaps it to the closest tick mark.

• JSliders (p. 1001) 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. 1005) of class JSlider returns the current thumb position.
• JSlider method setMajorTickSpacing() sets the spacing for tick marks on a JSlider. Method setPaintTicks(p. 1005) 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. 1005) declares method stateChanged(p. 1005) that can respond to ChangeEvents.

Section 25.3 Windows: Additional Notes
• A window’s (p. 1005) events can be handled by a WindowListener (p. 1006), which provides seven window-event-handling methods—windowActivated, windowClosed, windowClosing, windowDeactivated, windowDeiconified, windowIconified and windowOpened.

Section 25.4 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. 1006).

• A JMenuBar (p. 1006) is a container for menus. A JMenuItem appears in a menu. A JMenu (p. 1006) contains menu items (p. 1007) 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. 1007) 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. 1007) can be selected at a time.

• AbstractButton method setMnemonic (p. 1012) specifies the mnemonic (p. 1007) for a button. Mnemonic characters are normally displayed with an underline.

• A modal dialog box (p. 1012) 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. 1013) can be used to create your own modal or nonmodal dialogs.

Section 25.5 JPopupMenu
• Context-sensitive pop-up menus (p. 1014) are created with class JPopupMenu (p. 1014). The pop-up trigger event (p. 1014) occurs normally when the user presses and releases the right mouse button. MouseEvent method isPopupTrigger (p. 1017) returns true if the pop-up trigger event occurred.

• JPopupMenu method show (p. 1017) displays a JPopupMenu. The first argument specifies the origin component (p. 1017), 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 25.6 Pluggable Look-and-Feel
• Class UIManager.LookAndFeelInfo (p. 1021) maintains information about a look-and-feel.

• UIManager (p. 1021) static method getInstalledLookAIO РеспублиkaandFeels (p. 1021) returns an array of UIManager.LookAndFeelInfo objects that describe the available look-and-feels.

• UIManager static method setLookAndFeel (p. 1021) changes the look-and-feel. SwingUtilities (p. 1022) static method updateComponentTreeUI (p. 1022) changes the look-and-feel of every component attached to its Component argument to the new look-and-feel.

Section 25.7 JDesktopPane and JInternalFrame
• Many of today’s applications use a multiple-document interface (MDI; p. 1022) to manage several open documents that are being processed in parallel. Swing’s JDesktopPane (p. 1022) and JInternalFrame (p. 1022) classes provide support for creating multiple-document interfaces.
Section 25.8 JTabbedPane

- A JTabbedPane (p. 1026) arranges GUI components into layers, of which only one is visible at a time. Users access each layer by clicking its tab.

Section 25.9 Layout Managers: BoxLayout and GridBagLayout

- BoxLayout 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.
- GridBagLayout (p. 1031) is similar to GridLayout, but each component size can vary.
- A GridBagConstraints object (p. 1031) specifies how a component is placed in a GridBagLayout.

Self-Review Exercises

25.1 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.

25.2 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.

25.3 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 );
   } // end method paintComponent
   c) // create a JFrame and display it
   JFrame f = new JFrame( "A Window" );
   f.setVisible( true );

Answers to Self-Review Exercises

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

25.2   a) False. A JFrame does not require any menus.
   b) False. The variable fill belongs to the GridBagConstraints class.
   c) True.
   d) True.

25.3   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.
Exercises

25.4 (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.

25.5 (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.

25.6 (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() );

25.7 (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:
   \[ \text{diameter} = 2 \times \text{radius}, \quad \text{area} = \pi \times \text{radius}^2, \quad \text{circumference} = 2 \times \pi \times \text{radius} \]. Use the constant Math.PI for \( \pi \). All drawing should be done on a subclass of JPanel, and the results of the calculations should be displayed in a read-only JTextArea.

25.8 (Using a JSlider) Enhance the program in Exercise 25.7 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 25.7. All drawing should be done on a subclass of JPanel, and the results of the calculations should be displayed in a read-only JTextArea.

25.9 (Varying weightx and weighty) Explore the effects of varying the weightx and weighty values of the program in Fig. 25.21. 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)?

25.10 (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.]

25.11 (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.
25.12 *(Creating a Color Chooser: Modification)* Modify the `MyColorChooser` class of Exercise 25.11 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.

25.13 *(Creating a Color Chooser: Modification)* Modify the application in Exercise 25.12 to draw the current color as a rectangle on an instance of a subclass of `JPanel` which provides its own `paint-Component` 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.

25.14 *(Drawing Application)* Modify the application in Exercise 25.13 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.

25.15 *(Drawing Application Modification)* Modify the application in Exercise 25.14 to provide the user with the ability to terminate the application by clicking the close box on the window that is displayed and by selecting `Exit` from a `File` menu. Use the techniques shown in Fig. 25.5.

25.16 *(Complete Drawing Application)* Using the techniques developed in this chapter and Chapter 14, create a complete drawing application. The program should use the GUI components from Chapter 14 and Chapter 25 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.