

Chapter 18 Graphical User Interfaces

CHAPTER GOALS

- G** To understand the use of layout managers to arrange user-interface components in a container
- G** To become familiar with common user-interface components, such as buttons, combo boxes, and menus
- G** To build programs that handle events from user-interface components
 - To learn how to browse the Java documentation

In this chapter, we will delve more deeply into graphical user interface programming. The graphical applications with which you are familiar have many visual gadgets for information entry: buttons, scroll bars, menus, etc. In this chapter, you will learn how to use the most common user-interface components in the Java Swing user-interface toolkit. Swing has many more components than can be mastered in a first course, and even the basic components have advanced options that can't be covered here. In fact, few programmers try to learn everything about a particular user-interface component. It is more important to understand the concepts and to search the Java documentation for the details. This chapter walks you through one example to show you how the Java documentation is organized and how you can rely on it for your programming.

787

788

18.1 Layout Management

Up to now, you have had limited control over the layout of user-interface components. You learned how to add components to a panel. The panel arranged the components from the left to the right. However, in many applications, you need more sophisticated arrangements.

User-interface components are arranged by placing them inside containers. Containers can be placed inside larger containers.

Java Concepts, 5th Edition

In Java, you build up user interfaces by adding components into containers such as panels. Each container has its own *layout manager*, which determines how the components are laid out.

Each container has a layout manager that directs the arrangement of its components.

By default, a `JPanel` uses a *flow layout*. A flow layout simply arranges its components from left to right and starts a new row when there is no more room in the current row.

Three useful layout managers are the border layout, flow layout, and grid layout.

Another commonly used layout manager is the *border layout*. The border layout groups the container into five areas: center, north, west, south, and east (see [Figure 1](#)). Not all of the areas need to be occupied.

When adding a component to a container with the border layout, specify the NORTH, EAST, SOUTH, WEST, or CENTER position.

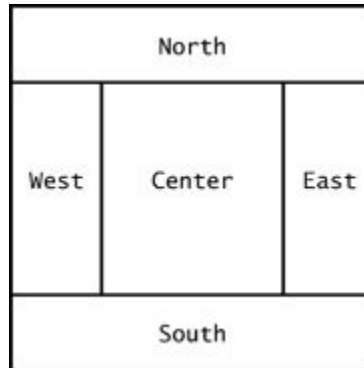
The border layout is the default layout manager for a frame (or, more technically, the frame's content pane). But you can also use the border layout in a panel:

```
panel.setLayout(new BorderLayout());
```

Now the panel is controlled by a border layout, not the flow layout. When adding a component, you specify the position, like this:

```
panel.add(component, BorderLayout.NORTH);
```

788

Figure 1

Components Expand to Fill Space in the Border Layout

The content pane of a frame has a border layout by default. A panel has a flow layout by default.

The *grid layout* is a third layout that is sometimes useful. The grid layout arranges components in a grid with a fixed number of rows and columns, resizing each of the components so that they all have the same size. Like the border layout, it also expands each component to fill the entire allotted area. (If that is not desirable, you need to place each component inside a panel.) [Figure 2](#) shows a number pad panel that uses a grid layout. To create a grid layout, you supply the number of rows and columns in the constructor, then add the components, row by row, left to right:

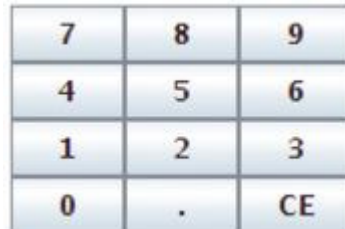
```
JPanel buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout(4, 3));
buttonPanel.add(button7);
buttonPanel.add(button8);
buttonPanel.add(button9);
buttonPanel.add(button4);
. . .
```

Sometimes you want to have a tabular arrangement of the components where columns have different sizes or one component spans multiple columns. A more complex layout manager called the *grid bag layout* can handle these situations. The grid bag layout is quite complex to use, however, and we do not cover it in this book; see, for

Java Concepts, 5th Edition

example, [1] for more information. Java 6 introduces a group layout that is designed for use by interactive tools—see [Productivity Hint 18.1](#).

Figure 2



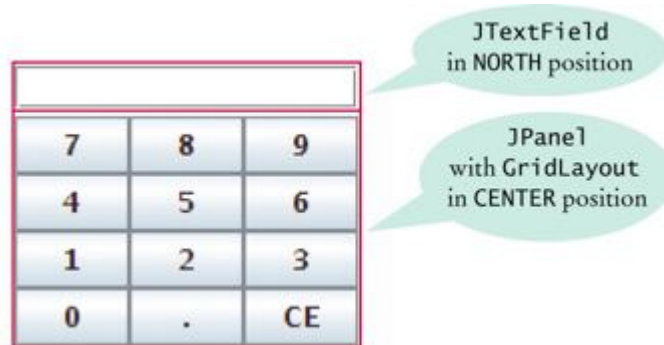
7	8	9
4	5	6
1	2	3
0	.	CE

The Grid Layout

Fortunately, you can create acceptable-looking layouts in nearly all situations by nesting panels. You give each panel an appropriate layout manager. Panels don't have visible borders, so you can use as many panels as you need to organize your components. [Figure 3](#) shows an example; the keypad from the ATM GUI in [Chapter 12](#). The keypad buttons are contained in a panel with grid layout. That panel is itself contained in a larger panel with border layout. The text field is in the northern position of the larger panel. The following code produces this arrangement:

```
JPanel keypadPanel = new JPanel();
keypadPanel.setLayout(new BorderLayout());
buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout(4, 3));
buttonPanel.add(button7);
buttonPanel.add(button8);
// . . .
keypadPanel.add(buttonPanel, BorderLayout.CENTER);
JTextField display = new JTextField();
keypadPanel.add(display, BorderLayout.NORTH);
```

Figure 3



Nesting Panels

SELF CHECK

1. How do you add two buttons to the north area of a frame?
2. How can you stack three buttons on top of each other?

18.2 Choices

18.2.1 Radio Buttons

For a small set of mutually exclusive choices, use a group of radio buttons or a combo box.

In this section you will see how to present a finite set of choices to the user. If the choices are mutually exclusive, use a set of *radio buttons*. In a radio button set, only one button can be selected at a time. When the user selects another button in the same set, the previously selected button is automatically turned off. (These buttons are called radio buttons because they work like the station selector buttons on a car radio: If you select a new station, the old station is automatically deselected.) For example, in [Figure 4](#), the font sizes are mutually exclusive. You can select small, medium, or large, but not a combination of them.

790

791

Add radio buttons into a `ButtonGroup` so that only one button in the group is on at any time.

To create a set of radio buttons, first create each button individually, and then add all buttons of the set to a `ButtonGroup` object:

```
JRadioButton smallButton = new
JRadioButton("Small");
JRadioButton mediumButton = new
JRadioButton("Medium");
JRadioButton largeButton = new
JRadioButton("Large");

ButtonGroup group = new ButtonGroup();
group.add(smallButton);
group.add(mediumButton);
group.add(largeButton);
```

Note that the button group does *not* place the buttons close to each other on the container. The purpose of the button group is simply to find out which buttons to turn off when one of them is turned on. It is still your job to arrange the buttons on the screen.

The `isSelected` method is called to find out whether a button is currently selected or not. For example,

```
if (largeButton.isSelected()) size = LARGE_SIZE;
```

Call `setSelected(true)` on one of the radio buttons in a radio button group before making the enclosing frame visible.

If you have multiple button groups, it is a good idea to group them together visually. You probably use panels to build up your user interface, but the panels themselves are invisible. You can add a *border* to a panel to make it visible. In [Figure 4](#), for example; the panels containing the Size radio buttons and Style check boxes have borders.

You can place a border around a panel to group its contents visually.

Figure 4



A Combo Box, Check Boxes, and Radio Buttons

791

There are a large number of border types. We will show only a couple of variations and leave it to the border enthusiasts to look up the others in the Swing documentation. The `EtchedBorder` class yields a border with a three-dimensional, etched effect. You can add a border to any component, but most commonly you apply it to a panel:

792

```
JPanel panel = new JPanel();  
panel.setBorder(new EtchedBorder());
```

If you want to add a title to the border (as in [Figure 4](#)), you need to construct a `TitledBorder`. You make a titled border by supplying a basic border and then the title you want. Here is a typical example:

```
panel.setBorder(new TitledBorder(new  
EtchedBorder(), "Size"));
```

18.2.2 Check Boxes

A check box is a user-interface component with two states: checked and unchecked. You use a group of check boxes when one selection does not exclude another. For example, the choices for “Bold” and “Italic” in [Figure 4](#) are not exclusive. You can choose either, both, or neither. Therefore, they are implemented as a set of separate check boxes. Radio buttons and check boxes have different visual appearances. Radio buttons are round and have a black dot when selected. Check boxes are square and have a check mark when selected. (Strictly speaking, the appearance depends on the chosen look and feel. It is possible to create a different look and feel in which check boxes have a different shape or in which they give off a particular sound when selected.)

For a binary choice, use a check box.

You construct a check box by giving the name in the constructor:

```
JCheckBox italicCheckBox = new JCheckBox("Italic");
```

Do not place check boxes inside a button group.

18.2.3 Combo Boxes

If you have a large number of choices, you don't want to make a set of radio buttons, because that would take up a lot of space. Instead, you can use a *combo box*. This component is called a combo box because it is a combination of a list and a text field. The text field displays the name of the current selection. When you click on the arrow to the right of the text field of a combo box, a list of selections drops down, and you can choose one of the items in the list (see [Figure 5](#)).

For a large set of choices, use a combo box.

If the combo box is *editable*, you can also type in your own selection. To make a combo box editable, call the `setEditable` method.

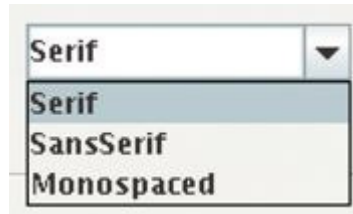
You add strings to a combo box with the `addItem` method.


```
JComboBox facenameCombo = new JComboBox();  
facenameCombo.addItem("Serif");  
facenameCombo.addItem("SansSerif");  
. . .
```

792

793

Figure 5



An Open Combo Box

You get the item that the user has selected by calling the `getSelectedItem` method. However, because combo boxes can store other objects in addition to strings, the `getSelectedItem` method has return type `Object`. Hence you must cast the returned value back to `String`.

```
String selectedString  
    = (String) facenameCombo.getSelectedItem();
```

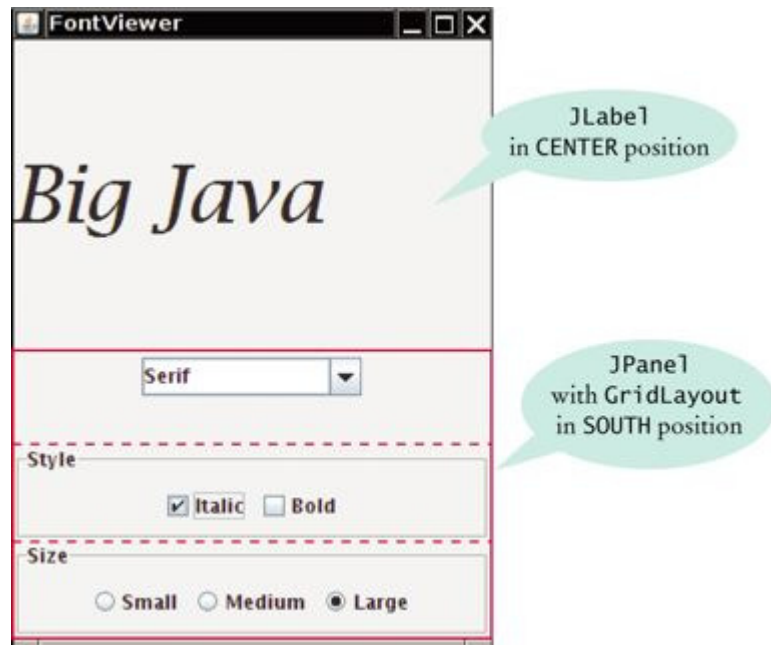
You can select an item for the user with the `setSelectedItem` method.

Radio buttons, check boxes, and combo boxes generate action events, just as buttons do.

Radio buttons, check boxes, and combo boxes generate an `ActionEvent` whenever the user selects an item. In the following program, we don't care which component was clicked—all components notify the same listener object. Whenever the user clicks on any one of them, we simply ask each component for its current content, using the `isSelected` and `getSelectedItem` methods. We then redraw the text sample with the new font.

[Figure 6](#) shows how the components are arranged in the frame. [Figure 7](#) shows the UML diagram.

Figure 6

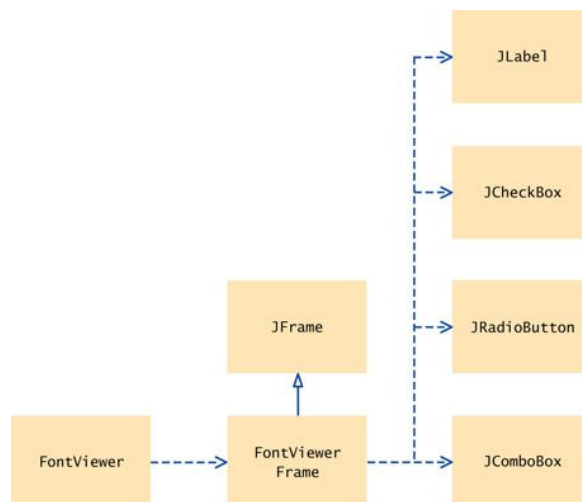


The Components of the FontViewerFrame

793

Figure 7

794



Classes of the Font Viewer Program

ch18/choice/FontViewer.java

```
1  import javax.swing.JFrame;
2
3  /**
4   * This program allows the user to view font
effects.
5   */
6  public class FontViewer
7  {
8      public static void main(String[] args)
9      {
10         JFrame frame = new FontViewerFrame();
11         frame.setDefaultCloseOperation(JFrame.EXIT_ON
12         frame.setTitle("FontViewer");
13         frame.setVisible(true);
14     }
15 }
```

794

ch18/choice/FontViewerFrame.java

```
1  import java.awt.BorderLayout;
2  import java.awt.Font;
3  import java.awt.GridLayout;
4  import java.awt.event.ActionEvent;
5  import java.awt.event.ActionListener;
6  import javax.swing.ButtonGroup;
7  import javax.swing.JButton;
8  import javax.swing.JCheckBox;
9  import javax.swing.JComboBox;
10 import javax.swing.JFrame;
11 import javax.swing.JLabel;
12 import javax.swing.JPanel;
13 import javax.swing.JRadioButton;
14 import javax.swing.border.EtchedBorder;
15 import javax.swing.border.TitledBorder;
16
17 /**
18  * This frame contains a text field and a
control panel
19  * to change the font of the text.
```

795

```
20  */
21  public class FontViewerFrame extends JFrame
22  {
23      /**
24       Constructs the frame.
25      */
26      public FontViewerFrame()
27      {
28          // Construct text sample
29          sampleField = new JLabel("Big Java");
30          add(sampleField, BorderLayout.CENTER);
31
32          // This listener is shared among all
components
33          class ChoiceListener implements
ActionListener
34          {
35              public void
actionPerformed(ActionEvent event)
36              {
37                  setSampleFont();
38              }
39          }
40
41          listener = new ChoiceListener();
42
43          createControlPanel();
44          setSampleFont();
45          setSize(FRAME_WIDTH, FRAME_HEIGHT);
46      }
47
48      /**
49       Creates the control panel to change the
font.
50      */
51      public void createControlPanel()
52      {
53          JPanel facenamePanel = createComboBox();
54          JPanel sizeGroupPanel =
createCheckBoxes();
55          JPanel styleGroupPanel =
createRadioButtons();
56
```

795

796

```
57         // Line up component panels
58
59         JPanel controlPanel = new JPanel();
60         controlPanel.setLayout(new
GridLayout(3, 1));
61         controlPanel.add(facenamePanel);
62         controlPanel.add(sizeGroupPanel);
63         controlPanel.add(styleGroupPanel);
64
65         // Add panels to content pane
66
67         add(controlPanel, BorderLayout.SOUTH);
68     }
69
70     /**
71      Creates the combo box with the font
style choices.
72      @return the panel containing the combo
box
73      */
74     public JPanel createComboBox()
75     {
76         facenameCombo = new JComboBox();
77         facenameCombo.addItem("Serif");
78         facenameCombo.addItem("SansSerif");
79         facenameCombo.addItem("Monospaced");
80         facenameCombo.setEditable(true);
81         facenameCombo.addActionListener(listener);
82
83         JPanel panel = new JPanel();
84         panel.add(facenameCombo);
85         return panel;
86     }
87
88     /**
89      Creates the check boxes for selecting
bold and italic styles.
90      @return the panel containing the check
boxes
91      */
92     public JPanel createCheckBoxes()
93     {
```

```
94         italicCheckBox = new
JCheckBox("Italic");
95         italicCheckBox.addActionListener(listener);
96
97         boldCheckBox = new JCheckBox("Bold");
98         boldCheckBox.addActionListener(listener);
99
100        JPanel panel = new JPanel();
101        panel.add(italicCheckBox);
102        panel.add(boldCheckBox);
103        panel.setBorder(
104            new TitledBorder(new
EtchedBorder(), "Style"));
105
```

796

```
106        return panel;
107    }
108
109    /**
110     * Creates the radio buttons to select the
font size.
111     * @return the panel containing the radio
buttons
112     */
113    public JPanel createRadioButtons()
114    {
115        smallButton = new JRadioButton("Small");
116        smallButton.addActionListener(listener);
117
118        mediumButton = new
JRadioButton("Medium");
119        mediumButton.addActionListener(listener);
120
121        largeButton = new JRadioButton("Large");
122        largeButton.addActionListener(listener);
123        largeButton.setSelected(true);
124
125        // Add radio buttons to button group
126
127        ButtonGroup group = new ButtonGroup();
128        group.add(smallButton);
129        group.add(mediumButton);
130        group.add(largeButton);
131
```

797

```
132     JPanel panel = new JPanel();
133     panel.add(smallButton);
134     panel.add(mediumButton);
135     panel.add(largeButton);
136     panel.setBorder(
137         new TitledBorder(new
EtchedBorder(), "Size"));
138
139     return panel;
140 }
141
142 /**
143     Gets user choice for font name, style,
and size
144     and sets the font of the text sample.
145 */
146 public void setSampleFont()
147 {
148     // Get font name
149     String facename
150         = (String)
facenameCombo.getSelectedItem();
151
152     // Get font style
153
154     int style = 0;
155     if (italicCheckBox.isSelected())
156         style = style + Font.ITALIC;
157     if (boldCheckBox.isSelected())
158         style = style + Font.BOLD;
159
160     // Get font size
161
162     int size = 0;
163
164     final int SMALL_SIZE = 24;
165     final int MEDIUM_SIZE = 36;
166     final int LARGE_SIZE = 48;
167
168     if (smallButton.isSelected())
169         size = SMALL_SIZE;
170     else if (mediumButton.isSelected())
171         size = MEDIUM_SIZE;
```

797

798

```
172         else if (largeButton.isSelected())
173             size = LARGE_SIZE;
174
175         // Set font of text field
176
177         sampleField.setFont(new Font(facename,
178 style, size));
179         sampleField.repaint();
180     }
181
182     private JLabel sampleField;
183     private JCheckBox italicCheckBox;
184     private JCheckBox boldCheckBox;
185     private JRadioButton smallButton;
186     private JRadioButton mediumButton;
187     private JRadioButton largeButton;
188     private JComboBox facenameCombo;
189     private ActionListener listener;
190
191     private static final int FRAME_WIDTH = 300;
192     private static final int FRAME_HEIGHT = 400;
193 }
```

SELF CHECK

- [3.](#) What is the advantage of a JComboBox over a set of radio buttons? What is the disadvantage?
- [4.](#) Why do all user interface components in the FontViewerFrame class share the same listener?
- [5.](#) Why was the combo box placed inside a panel? What would have happened if it had been added directly to the control panel?

798

799

How To 18.1: Laying Out a User Interface

A graphical user interface is made up of components such as buttons and text fields. The Swing library uses containers and layout managers to arrange these components. This How To explains how to group components into containers and how to pick the right layout managers.

Java Concepts, 5th Edition

Step 1 Make a sketch of your desired component layout.

Draw all the buttons, labels, text fields, and borders on a sheet of paper. Graph paper works best.

Here is an example—a user interface for ordering pizza. The user interface contains

- Three radio buttons
- Two check boxes
- A label: “Your Price:”
- A text field
- A border

Size

☒ Small ☒ Pepperoni

☐ Medium ☒ Anchovies

☐ Large

Your Price:

Step 2 Find groupings of adjacent components with the same layout.

Usually, the component arrangement is complex enough that you need to use several panels, each with its own layout manager. Start by looking at adjacent components that are arranged top to bottom or left to right. If several components are surrounded by a border, they should be grouped together.

Here are the groupings from the pizza user interface:

Size

☒ Small ☒ Pepperoni

☐ Medium ☒ Anchovies

☐ Large

Your Price:

Step 3 Identify layouts for each group.

When components are arranged horizontally, choose a flow layout. When components are arranged vertically, use a grid layout. The grid in this layout has as many rows as there are components, and it has one column.

In the pizza user interface example, you would choose

- A (3, 1) grid layout for the radio buttons
- A (2, 1) grid layout for the check boxes
- A flow layout for the label and text field

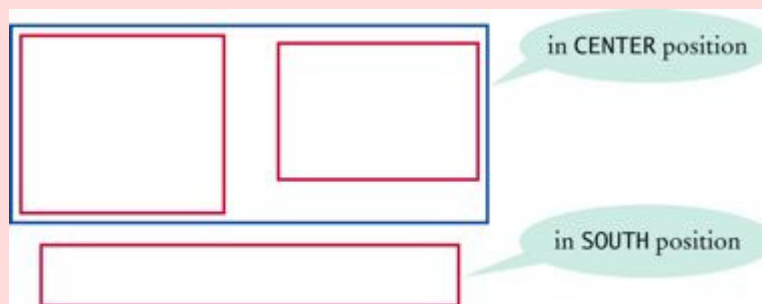
Step 4 Group the groups together.

Look at each group as one blob, and group the blobs together into larger groups, just as you grouped the components in the preceding step. If you note one large blob surrounded by smaller blobs, you can group them together in a border layout.

You may have to repeat the grouping again if you have a very complex user interface. You are done if you have arranged all groups in a single container.

For example, the three component groups of the pizza user interface can be arranged as follows:

- A group containing the first two component groups, placed in the center of a container with a border layout
- The third component group, in the southern area of that container



In this step, you may run into a couple of complications. The group “blobs” tend to vary in size more than the individual components. If you place them inside a grid layout, the grid layout forces them all to be the same size. Also, you occasionally would like a component from one group to line up with a component from another group, but there is no way for you to communicate that intent to the layout managers.

These problems can be overcome by using more sophisticated layout managers or implementing a custom layout manager. However, those techniques are beyond the scope of this book. Sometimes, you may want to start over with Step 1, using a component layout that is easier to manage. Or you can decide to live with minor imperfections of the layout. Don't worry about achieving the perfect layout—after all, you are learning programming, not user-interface design.

Step 5 Write the code to generate the layout.

This step is straightforward but potentially tedious, especially if you have a large number of components.

Start by constructing the components. Then construct a panel for each component group and set its layout manager if it is not a flow layout (the default for panels). Add a border to the panel if required. Finally, add the components to the panel. Continue in this fashion until you reach the outermost containers, which you add to the frame.

800

801

Here is an outline of the code required for the pizza user interface.

```
JPanel radioButtonPanel = new JPanel();
radioButtonPanel.setLayout(new GridLayout(3, 1));
radioButton.setBorder(
    new TitledBorder(new EtchedBorder(),
        "Size"));
radioButtonPanel.add(smallButton);
radioButtonPanel.add(mediumButton);
radioButtonPanel.add(largeButton);

JPanel checkBoxPanel = new JPanel();
checkBoxPanel.setLayout(new GridLayout(2, 1));
checkBoxPanel.add(pepperoniButton());
checkBoxPanel.add(anchoviesButton());
```

```
JPanel pricePanel = new JPanel(); // Uses
FlowLayout
pricePanel.add(new JLabel("Your Price:"));
pricePanel.add(priceTextField);

JPanel centerPanel = new JPanel(); // Uses
FlowLayout
centerPanel.add(radiusButtonPanel);
centerPanel.add(checkBoxPanel);

// Frame uses BorderLayout by default
add(centerPanel, BorderLayout.CENTER);
add(pricePanel, BorderLayout.SOUTH);
```

Of course, you also need to add event handlers to the components. That is the topic of How To 10.1.

PRODUCTIVITY HINT 18.1: Use a GUI Builder

As you have seen, implementing even a simple graphical user interface in Java is quite tedious. You have to write a lot of code for constructing components, using layout managers, and providing event handlers. Most of the code is boring and repetitive.

A GUI builder takes away much of the tedium. Most GUI builders help you in three ways:

- You drag and drop components onto a panel. The GUI builder writes the layout management code for you.
- You customize components with a dialog box, setting properties such as fonts, colors, text, and so on. The GUI builder writes the customization code for you.
- You provide event handlers by picking the event to process and providing just the code snippet for the listener method. The GUI builder writes the boilerplate code for attaching a listener object.

801

Java 6 introduced `GroupLayout`, a powerful layout manager that was specifically designed to be used by GUI builders. The free NetBeans

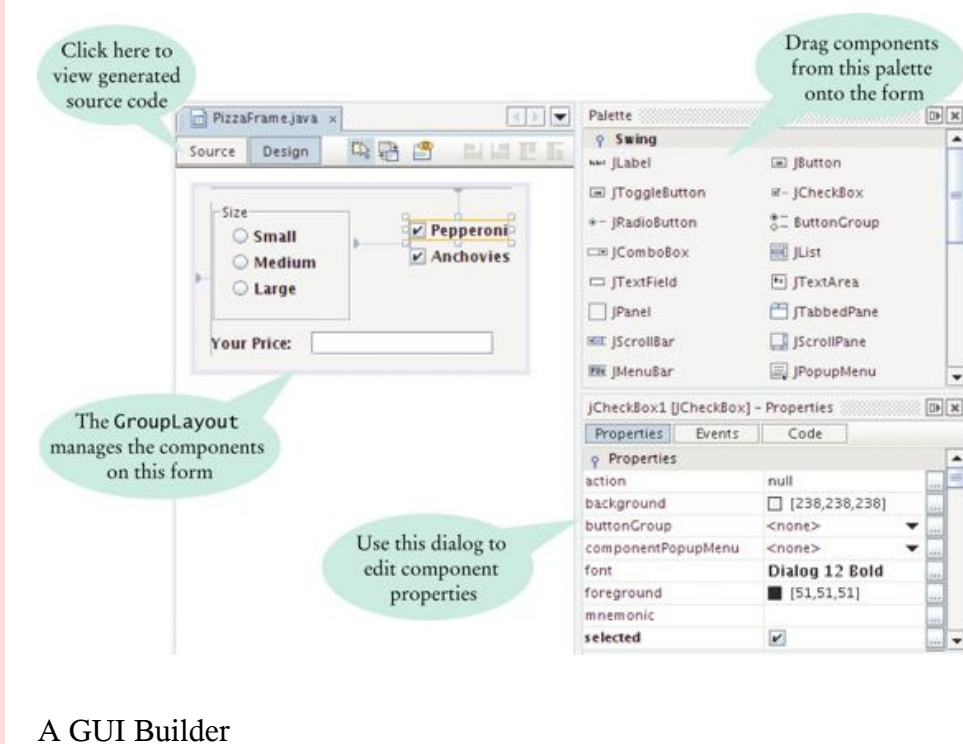
802

Java Concepts, 5th Edition

development environment, available from <http://netbeans.org>, makes use of this layout manager—see [Figure 8](#).

If you need to build a complex user interface, you will find that learning to use a GUI builder is a very worthwhile investment. You will spend less time writing boring code, and you will have more fun designing your user interface and focusing on the functionality of your program.

Figure 8



18.3 Menus

Anyone who has ever used a graphical user interface is familiar with pull-down menus (see [Figure 9](#)). In Java it is easy to create these menus.

A frame contains a menu bar. The menu bar contains menus. A menu contains submenus and menu items.

Java Concepts, 5th Edition

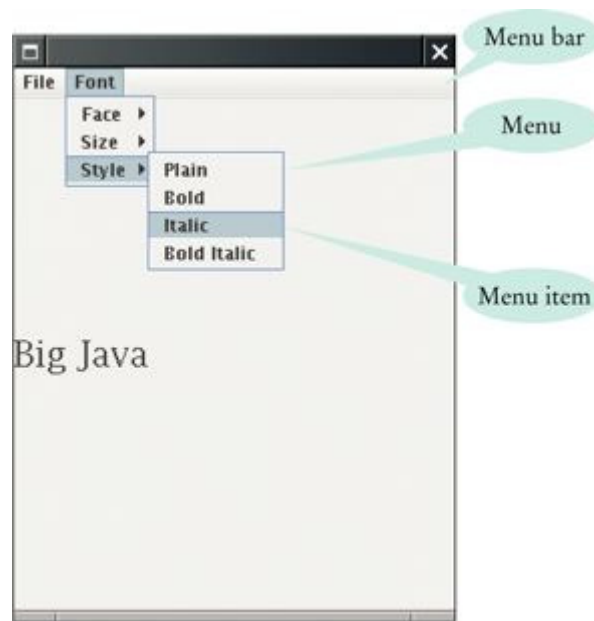
The container for the top-level menu items is called a *menu bar*. A *menu* is a collection of *menu items* and more menus (submenus). You add menu items and submenus with the add method:

```
JMenuItem fileExitItem = new JMenuItem("Exit");  
fileMenu.add(fileExitItem);
```

802

803

Figure 9



Pull-Down Menus

A menu item has no further submenus. When the user selects a menu item, the menu item sends an action event. Therefore, you want to add a listener to each menu item:

```
fileExitItem.addActionListener(listener);
```

You add action listeners only to menu items, not to menus or the menu bar. When the user clicks on a menu name and a submenu opens, no action event is sent.

Menu items generate action events.

The following program builds up a small but typical menu and traps the action events from the menu items. To keep the program readable, it is a good idea to use a separate method for each menu or set of related menus. Have a look at the `createFaceItem` method, which creates a menu item to change the font face. The same listener class takes care of three cases, with the name parameters varying for each menu item. The same strategy is used for the `createSizeItem` and `createStyleItem` methods.

ch18/menu/FontViewer2.java

```
1  import javax.swing.JFrame;
2
3  /**
4   * This program uses a menu to display font
5   * effects.
6   */
7  public class FontViewer2
8  {
9      public static void main(String[] args)
10     {
11         JFrame frame = new FontViewer2Frame();
12         frame.setDefaultCloseOperation(JFrame.EXIT_ON_
13     }
14 }
```

803

ch18/menu/FontViewer2Frame.java

```
1  import java.awt.BorderLayout;
2  import java.awt.Font;
3  import java.awt.GridLayout;
4  import java.awt.event.ActionEvent;
5  import java.awt.event.ActionListener;
6  import javax.swing.ButtonGroup;
7  import javax.swing.JButton;
8  import javax.swing.JCheckBox;
9  import javax.swing.JComboBox;
10 import javax.swing.JFrame;
11 import javax.swing.JLabel;
12 import javax.swing.JMenu;
```

804

```
13 import javax.swing.JMenuBar;
14 import javax.swing.JMenuItem;
15 import javax.swing.JPanel;
16 import javax.swing.JRadioButton;
17 import javax.swing.border.EtchedBorder;
18 import javax.swing.border.TitledBorder;
19
20 /**
21     This frame has a menu with commands to
change the font
22     of a text sample.
23 */
24 public class FontViewer2Frame extends JFrame
25 {
26     /**
27         Constructs the frame.
28     */
29     public FontViewer2Frame()
30     {
31         // Construct text sample
32         sampleField = new JLabel("Big Java");
33         add(sampleField, BorderLayout.CENTER);
34
35         // Construct menu
36         JMenuBar menuBar = new JMenuBar();
37         setJMenuBar(menuBar);
38         menuBar.add(createFileMenu());
39         menuBar.add(createFontMenu());
40
41         facename = "Serif";
42         fontsize = 24;
43         fontstyle = Font.PLAIN;
44
45         setSampleFont();
46         setSize(FRAME_WIDTH, FRAME_HEIGHT);
47     }
48
49     /**
50         Creates the File menu.
51         @return the menu
52     */
53     public JMenu createFileMenu()
54     {
```

804

805


```
55         JMenu menu = new JMenu("File");
56         menu.add(createFileExitItem());
57         return menu;
58     }
59
60     /**
61      * Creates the File->Exit menu item and
62      * sets its action listener.
63      * @return the menu item
64      */
65     public JMenuItem createFileExitItem()
66     {
67         JMenuItem item = new JMenuItem("Exit");
68         class MenuItemListener implements
69             ActionListener
70         {
71             public void
72             actionPerformed(ActionEvent event)
73             {
74                 System.exit(0);
75             }
76         }
77         ActionListener listener = new
78             MenuItemListener();
79         item.addActionListener(listener);
80         return item;
81     }
82
83     /**
84      * Creates the Font submenu.
85      * @return the menu
86      */
87     public JMenu createFontMenu()
88     {
89         JMenu menu = new JMenu("Font");
90         menu.add(createFaceMenu());
91         menu.add(createSizeMenu());
92         menu.add(createStyleMenu());
93         return menu;
94     }
95
96     /**
97      * Creates the Face submenu.
```

```
94      @return the menu
95      */
96      public JMenu createFaceMenu()
97      {
98          JMenu menu = new JMenu("Face");
99          menu.add(createFaceItem("Serif"));
100         menu.add(createFaceItem("SansSerif"));
101         menu.add(createFaceItem("Monospaced"));
102         return menu;
103     }
104
105     /**
106      Creates the Size submenu.
107      @return the menu
108     */
```

805

```
109     public JMenu createSizeMenu()
110     {
111         JMenu menu = new JMenu("Size");
112         menu.add(createSizeItem("Smaller", -1));
113         menu.add(createSizeItem("Larger", 1));
114         return menu;
115     }
116
117     /**
118      Creates the Style submenu.
119      @return the menu
120     */
121     public JMenu createStyleMenu()
122     {
123         JMenu menu = new JMenu("Style");
124         menu.add(createStyleItem("Plain",
Font.PLAIN));
125         menu.add(createStyleItem("Bold",
Font.BOLD));
126         menu.add(createStyleItem("Italic",
Font.ITALIC));
127         menu.add(createStyleItem("Bold Italic",
Font.BOLD
128             + Font.ITALIC));
129         return menu;
130     }
131
132     /**
```

806

```
133      Creates a menu item to change the font
134      face and set its action listener.
135      @param name the name of the font face
136      @return the menu item
137      */
138      public JMenuItem createFaceItem(final
String name)
139      {
140          JMenuItem item = new JMenuItem(name);
141          class MenuItemListener implements
ActionListener
142          {
143              public void
actionPerformed(ActionEvent event)
144              {
145                  facename = name;
146                  setSampleFont();
147              }
148          }
149          ActionListener listener = new
MenuItemListener();
150          item.addActionListener(listener);
151          return item;
152      }
153      /**
154      Creates a menu item to change the font
155      size
156      and set its action listener.
157      @param name the name of the menu item
158      @param ds the amount by which to change
159      the size
160      @return the menu item
161      */
162      public JMenuItem createSizeItem(String
name, final int ds)
163      {
164          JMenuItem item = new JMenuItem(name);
165          class MenuItemListener implements
ActionListener
166          {
167              public void
actionPerformed(ActionEvent event)
```

806

807

```
166         {
167             fontsize = fontsize + ds;
168             setSampleFont();
169         }
170     }
171     ActionListener listener = new
MenuItemListener();
172     item.addActionListener(listener);
173     return item;
174 }
175
176 /**
177     Creates a menu item to change the font
style
178     and set its action listener.
179     @param name the name of the menu item
180     @param style the new font style
181     @return the menu item
182 */
183 public JMenuItem createStyleItem(String
name, final int style)
184 {
185     JMenuItem item = new JMenuItem(name);
186     class MenuItemListener implements
ActionListener
187     {
188         public void
actionPerformed(ActionEvent event)
189         {
190             fontstyle = style;
191             setSampleFont();
192         }
193     }
194     ActionListener listener = new
MenuItemListener();
195     item.addActionListener(listener);
196     return item;
197 }
198
199 /**
200     Sets the font of the text sample.
201 */
202 public void setSampleFont()
```

```
203     {
204         Font f = new Font(facename, fontstyle,
205             fontsize);
206         sampleField.setFont(f);
207         sampleField.repaint();
208     }
209     private JLabel sampleField;
210     private String facename;
211     private int fontstyle;
212     private int fontsize;
213
214     private static final int FRAME_WIDTH = 300;
215     private static final int FRAME_HEIGHT = 400;
216 }
```

807

SELF CHECK

808

- [6.](#) Why do JMenu objects not generate action events?
- [7.](#) Why is the name parameter in the createFaceItem method declared as final?

18.4 Exploring the Swing Documentation

In the preceding sections, you saw the basic properties of the most common user-interface components. We purposefully omitted many options and variations to simplify the discussion. You can go a long way by using only the simplest properties of these components. If you want to implement a more sophisticated effect, you can look inside the Swing documentation. You will probably find the documentation quite intimidating at first glance, though. The purpose of this section is to show you how you can use the documentation to your advantage without becoming overwhelmed.

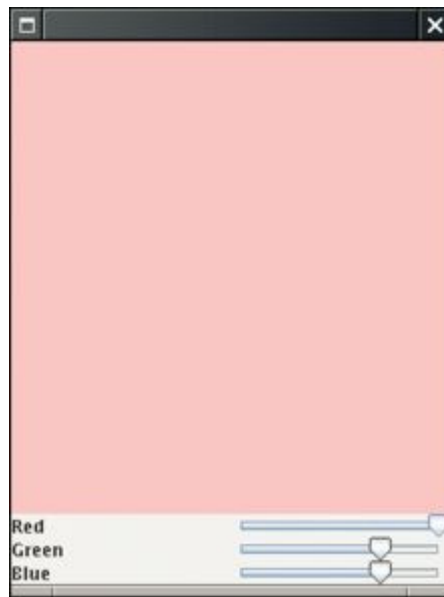
You should learn to navigate the API documentation to find out more about user-interface components.

Recall the `Color` class that was introduced in [Chapter 2](#). Every combination of red, green, and blue values represents a different color. It should be fun to mix your own colors, with a slider for the red, green, and blue values (see [Figure 10](#)).

Java Concepts, 5th Edition

The Swing user interface toolkit has a large set of user-interface components. How do you know if there is a slider? You can buy a book that illustrates all Swing components, such as [2]. Or you can run the sample application included in the Java Development Kit that shows off all Swing components (see [Figure 11](#)). Or you can look at the names of all of the classes that start with `J` and decide that `JSlider` may be a good candidate.

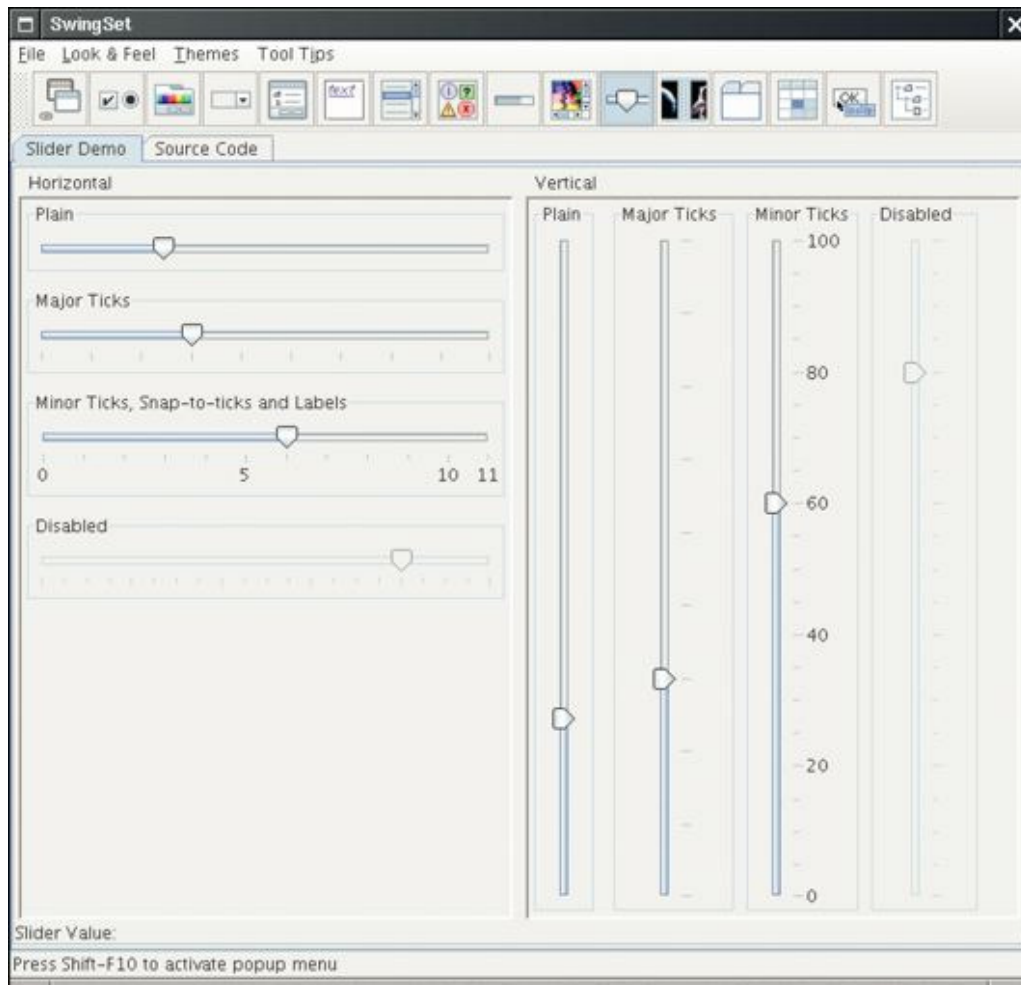
Figure 10



A Color Viewer

808

Figure 11



The SwingSet Demo

Next, you need to ask yourself a few questions:

- How do I construct a `JSlider`?
- How can I get notified when the user has moved it?
- How can I tell to which value the user has set it?

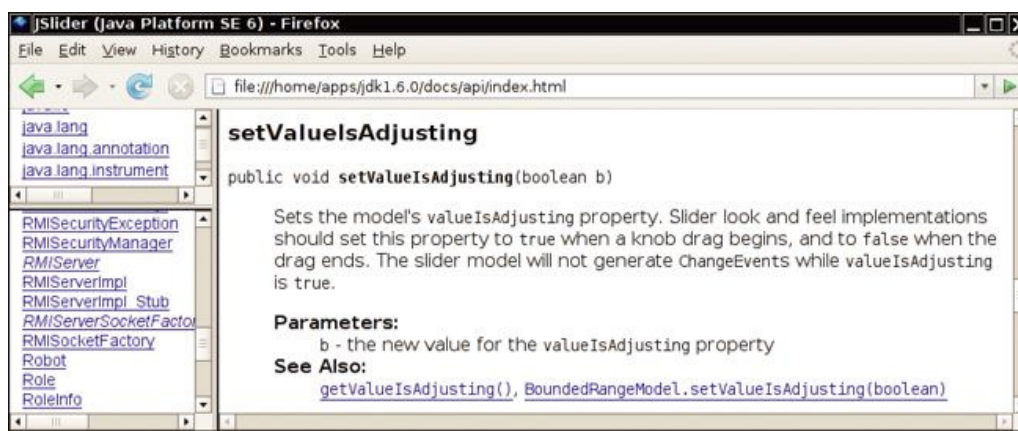
If you can answer these questions, then you can put a slider to good use. Once you have mastered sliders, you can fritter away more time and find out how to set tick marks or otherwise enhance the visual beauty of your creation.

When you look at the documentation of the `JSlider` class, you will probably not be happy. There are over 50 methods in the `JSlider` class and over 250 inherited methods, and some of the method descriptions look downright scary, such as the one in [Figure 12](#). Apparently some folks out there are concerned about the `valueIsAdjusting` property, whatever that may be, and the designers of this class felt it necessary to supply a method to tweak that property. Until you too feel that need, your best bet is to ignore this method. As the author of an introductory book, it pains me to tell you to ignore certain facts. But the truth of the matter is that the Java library is so large and complex that nobody understands it in its entirety, not even the designers of Java themselves. You need to develop the ability to separate fundamental concepts from ephemeral minutiae. For example, it is important that you understand the concept of event handling. Once you understand the concept, you can ask the question, “What event does the slider send when the user moves it?” But it is not important that you memorize how to set tick marks or that you know how to implement a slider with a custom look and feel.

809

810

Figure 12



A Mysterious Method Description from the API Documentation

Java Concepts, 5th Edition

Let us go back to our fundamental questions. In Java 6, there are six constructors for the `JSlider` class. You want to learn about one or two of them. You must strike a balance somewhere between the trivial and the bizarre. Consider

```
public JSlider()  
    Creates a horizontal slider with the range 0 to  
    100 and an initial value of 50.
```

Maybe that is good enough for now, but what if you want another range or initial value? It seems too limited.

On the other side of the spectrum, there is

```
public JSlider(BoundedRangeModel brm)  
    Creates a horizontal slider using the specified  
    BoundedRangeModel.
```

Whoa! What is that? You can click on the `BoundedRangeModel` link to get a long explanation of this class. This appears to be some internal mechanism for the Swing implementors. Let's try to avoid this constructor if we can. Looking further, we find

```
public JSlider(int min, int max, int value)  
    Creates a horizontal slider using the specified  
    min, max, and value.
```

This sounds general enough to be useful and simple enough to be usable. You might want to stash away the fact that you can have vertical sliders as well.

810

Next, you want to know what events a slider generates. There is no `addActionListener` method. That makes sense. Adjusting a slider seems different from clicking a button, and Swing uses a different event type for these events. There is a method

811

```
public void addChangeListener(ChangeListener l)
```

Click on the `ChangeListener` link to find out more about this interface. It has a single method

```
void stateChanged(ChangeEvent e)
```

Apparently, that method is called whenever the user moves the slider. What is a `ChangeEvent`? Once again, click on the link, to find out that this event class has *no*

Java Concepts, 5th Edition

methods of its own, but it inherits the `getSource` method from its superclass `EventObject`. The `getSource` method tells us which component generated this event, but we don't need that information—we know that the event came from the slider.

Now we have a plan: Add a change event listener to each slider. When the slider is changed, the `stateChanged` method is called. Find out the new value of the slider. Recompute the color value and repaint the color panel. That way, the color panel is continually repainted as the user moves one of the sliders.

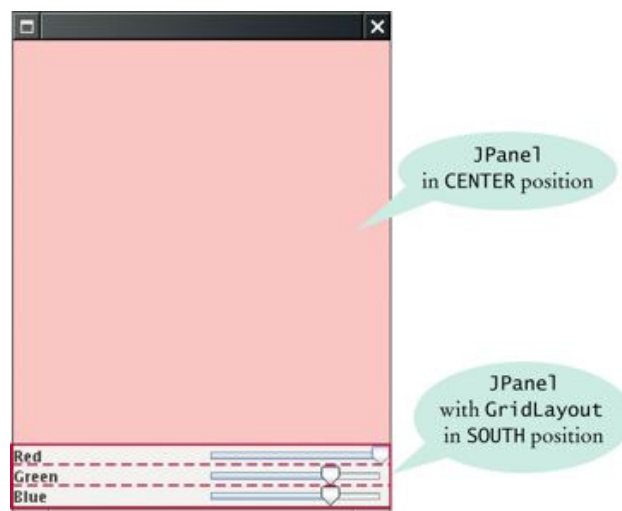
To compute the color value, you will still need to get the current value of the slider. Look at all the methods that start with `get`. Sure enough, you find

```
public int getValue()  
    Returns the slider's value.
```

Now you know everything you need to write the program. The program uses one new Swing component and one event listener of a new type. Of course, now that you have “tasted blood”, you may want to add those tick marks—see Exercise P18.10.

[Figure 13](#) shows how the components are arranged in the frame. [Figure 14](#) shows the UML diagram.

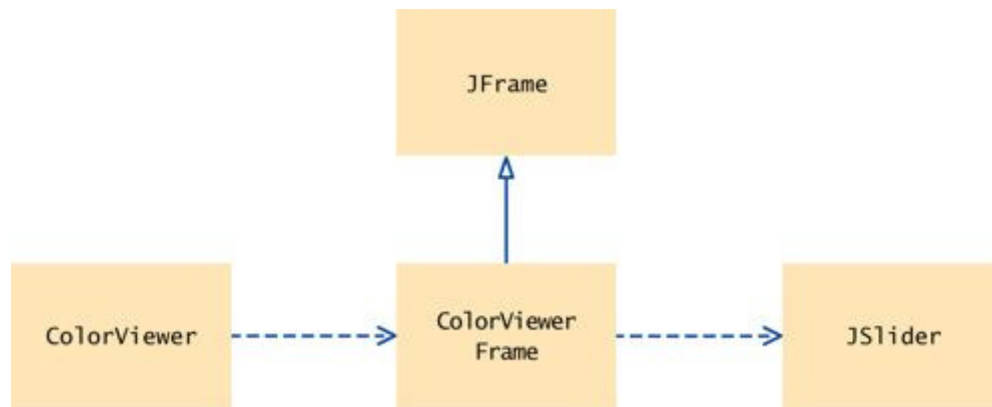
Figure 13



The Components of the `ColorViewerFrame`

811

Figure 14



Classes of the Color Viewer Program

ch18/slider/ColorViewer.java

```
1  import javax.swing.JFrame;
2
3  public class ColorViewer
4  {
5      public static void main(String[] args)
6      {
7          ColorViewerFrame frame = new
ColorViewerFrame();
8          frame.setDefaultCloseOperation(JFrame.EXIT_ON_C
9          frame.setVisible(true);
10     }
11 }
```

ch18/slider/ColorViewerFrame.java

```
1  import java.awt.BorderLayout;
2  import java.awt.Color;
3  import java.awt.GridLayout;
4  import javax.swing.JFrame;
5  import javax.swing.JLabel;
6  import javax.swing.JPanel;
7  import javax.swing.JSlider;
```

```
8 import javax.swing.event.ChangeListener;
9 import javax.swing.event.ChangeEvent;
10
11 public class ColorViewerFrame extends JFrame
12 {
13     public ColorViewerFrame()
14     {
15         colorPanel = new JPanel();
16
17         add(colorPanel, BorderLayout.CENTER);
18         createControlPanel();
19         setSampleColor();
20         setSize(FRAME_WIDTH, FRAME_HEIGHT);
21     }
```

812

```
22
23     public void createControlPanel()
24     {
25         class ColorListener implements
ChangeListener
26         {
27             public void stateChanged(ChangeEvent
event)
28             {
29                 setSampleColor();
30             }
31         }
32
33         ChangeListener listener = new
ColorListener();
34
35         redSlider = new JSlider(0, 255, 175);
36         redSlider.addChangeListener(listener);
37
38         greenSlider = new JSlider(0, 255, 175);
39         greenSlider.addChangeListener(listener);
40
41         blueSlider = new JSlider(0, 255, 175);
42         blueSlider.addChangeListener(listener);
43
44         JPanel controlPanel = new JPanel();
45         controlPanel.setLayout(new GridLayout(3,
2));
46
```

813

```
47     controlPanel.add(new JLabel("Red"));
48     controlPanel.add(redSlider);
49
50     controlPanel.add(new JLabel("Green"));
51     controlPanel.add(greenSlider);
52
53     controlPanel.add(new JLabel("Blue"));
54     controlPanel.add(blueSlider);
55
56     add(controlPanel, BorderLayout.SOUTH);
57 }
58
59 /**
60  Reads the slider values and sets the
panel to
61  the selected color.
62  */
63 public void setSampleColor()
64 {
65     // Read slider values
66
67     int red = redSlider.getValue();
68     int green = greenSlider.getValue();
69     int blue = blueSlider.getValue();
70
71     // Set panel background to selected color
72
73     colorPanel.setBackground(new Color(red,
green, blue));
74     colorPanel.repaint();
75 }
76
77 private JPanel colorPanel;
78 private JSlider redSlider;
79 private JSlider greenSlider;
80 private JSlider blueSlider;
81
82 private static final int FRAME_WIDTH = 300;
83 private static final int FRAME_HEIGHT = 400;
84 }
```

813

814

SELF CHECK

- [8.](#) Suppose you want to allow users to pick a color from a color dialog box. Which class would you use? Look in the API documentation.
- [9.](#) Why does a slider emit change events and not action events?

CHAPTER SUMMARY

1. User-interface components are arranged by placing them inside containers. Containers can be placed inside larger containers.
2. Each container has a layout manager that directs the arrangement of its components.
3. Three useful layout managers are the border layout, flow layout, and grid layout.
4. When adding a component to a container with the border layout, specify the NORTH, EAST, SOUTH, WEST, or CENTER position.
5. The content pane of a frame has a border layout by default. A panel has a flow layout by default.
6. For a small set of mutually exclusive choices, use a group of radio buttons or a combo box.
7. Add radio buttons into a `ButtonGroup` so that only one button in the group is on at any time.
8. You can place a border around a panel to group its contents visually.
9. For a binary choice, use a check box.
10. For a large set of choices, use a combo box.
11. Radio buttons, check boxes, and combo boxes generate action events, just as buttons do.

814

12. A frame contains a menu bar. The menu bar contains menus. A menu contains submenus and menu items.
13. Menu items generate action events.
14. You should learn to navigate the API documentation to find out more about user-interface components.

FURTHER READING

1. Cay S. Horstmann and Gary Cornell, *Core Java 2 Volume 1: Fundamentals*, 7th edition, Prentice Hall, 2004.
2. Kim Topley, *Core Java Foundation Classes*, 2nd edition, Prentice Hall, 2002.

CLASSES, OBJECTS, AND METHODS INTRODUCED IN THIS CHAPTER

```
java.awt.BorderLayout
    CENTER
    EAST
    NORTH
    SOUTH
    WEST
java.awt.Container
    setLayout
java.awt.FlowLayout
java.awt.Font
java.awt.GridLayout
javax.swing.AbstractButton
    isSelected
    setSelected
javax.swing.ButtonGroup
    add
javax.swing.ImageIcon
javax.swing.JCheckBox
javax.swing.JComboBox
    addItem
    getSelectedItem
    isEditable
```

```
setEditable
javax.swing.JComponent
setBorder
setFont
javax.swing.JFrame
setJMenuBar
javax.swing.JMenu
add
javax.swing.JMenuBar
add
javax.swing.JMenuItem
javax.swing.JRadioButton
javax.swing.JScrollPane
javax.swing.JSlider
addChangeListener
getValue
javax.swing.border.EtchedBorder
javax.swing.border.TitledBorder
javax.swing.event.ChangeEvent
javax.swing.event.ChangeListener
stateChanged
```

815

816

REVIEW EXERCISES

★**G Exercise R18.1.** Can you use a flow layout for the components in a frame?
If yes, how?

★**G Exercise R18.2.** What is the advantage of a layout manager over telling the container “place this component at position (x, y)”?

★★**G Exercise R18.3.** What happens when you place a single button into the CENTER area of a container that uses a border layout? Try it out, by writing a small sample program, if you aren't sure of the answer.

★★**G Exercise R18.4.** What happens if you place multiple buttons directly into the SOUTH area, without using a panel? Try it out, by writing a small sample program, if you aren't sure of the answer.

★★**G Exercise R18.5.** What happens when you add a button to a container that uses a border layout and omit the position? Try it out and explain.

- ★★G Exercise R18.6. What happens when you try to add a button to another button? Try it out and explain.
- ★★G Exercise R18.7. The `ColorViewerFrame` uses a grid layout manager. Explain a drawback of the grid that is apparent from [Figure 13](#). What could you do to overcome this drawback?
- ★★★G Exercise R18.8. What is the difference between the grid layout and the grid bag layout?
- ★★★G Exercise R18.9. Can you add icons to check boxes, radio buttons, and combo boxes? Browse the Java documentation to find out. Then write a small test program to verify your findings.
- ★G Exercise R18.10. What is the difference between radio buttons and check boxes?
- ★G Exercise R18.11. Why do you need a button group for radio buttons but not for check boxes?
- ★G Exercise R18.12. What is the difference between a menu bar, a menu, and a menu item?
- ★G Exercise R18.13. When browsing through the Java documentation for more information about sliders, we ignored the `JSlider` default constructor. Why? Would it have worked in our sample program?
- ★G Exercise R18.14. How do you construct a vertical slider? Consult the Swing documentation for an answer.
- ★★G Exercise R18.15. Why doesn't a `JComboBox` send out change events?
- ★★★G Exercise R18.16. What component would you use to show a set of choices, just as in a combo box, but so that several items are visible at the same time? Run the Swing demo app or look at a book with Swing example programs to find the answer.
- ★★G Exercise R18.17. How many Swing user interface components are there? Look at the Java documentation to get an approximate answer.

★★G **Exercise R18.18.** How many methods does the `JProgressBar` component have? Be sure to count inherited methods. Look at the Java documentation.

 Additional review exercises are available in WileyPLUS.

PROGRAMMING EXERCISES

★G **Exercise P18.1.** Write an application with three buttons labeled “Red”, “Green”, and “Blue” that changes the background color of a panel in the center of the frame to red, green, or blue.

★★G **Exercise P18.2.** Add icons to the buttons of Exercise P18.1.

★★G **Exercise P18.3.** Write a calculator application. Use a grid layout to arrange buttons for the digits and for the $+$ $-$ \times \div operations. Add a text field to display the result.

★G **Exercise P18.4.** Write an application with three radio buttons labeled “Red”, “Green”, and “Blue” that changes the background color of a panel in the center of the frame to red, green, or blue.

★G **Exercise P18.5.** Write an application with three check boxes labeled “Red”, “Green”, and “Blue” that adds a red, green, or blue component to the the background color of a panel in the center of the frame. This application can display a total of eight color combinations.

★G **Exercise P18.6.** Write an application with a combo box containing three items labeled “Red”, “Green”, and “Blue” that changes the background color of a panel in the center of the frame to red, green, or blue.

★G **Exercise P18.7.** Write an application with a Color menu and menu items labeled “Red”, “Green”, and “Blue” that changes the background color of a panel in the center of the frame to red, green, or blue.

★G **Exercise P18.8.** Write a program that displays a number of rectangles at random positions. Supply buttons “Fewer” and “More” that generate fewer or more random rectangles. Each time the user clicks on “Fewer”, the

Java Concepts, 5th Edition

count should be halved. Each time the user clicks on “More”, the count should be doubled.

★★G **Exercise P18.9.** Modify the program of Exercise P18.8 to replace the buttons with a slider to generate fewer or more random rectangles.

★★G **Exercise P18.10.** In the slider test program, add a set of tick marks to each slider that show the exact slider position.

★★★G **Exercise P18.11.** Enhance the font viewer program to allow the user to select different fonts. Research the API documentation to find out how to find the available fonts on the user's system.

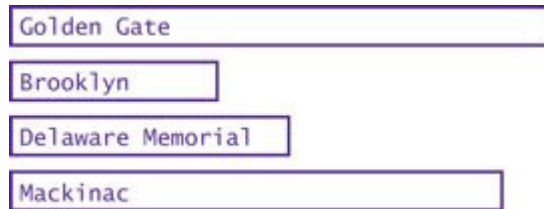
Additional programming exercises are available in WileyPLUS.

817

818

PROGRAMMING PROJECTS

★★★G **Project 18.1.** Write a program that lets users design charts such as the following:



Use appropriate components to ask for the length, label, and color, then apply them when the user clicks an “Add Item” button. Allow the user to switch between bar charts and pie charts.

★★★G **Project 18.2.** Write a program that displays a scrolling message in a panel. Use a timer for the scrolling effect. In the timer's action listener, move the starting position of the message and repaint. When the message has left the window, reset the starting position to the other corner. Provide a user interface to customize the message text, font, foreground and background colors, and the scrolling speed and direction.

ANSWERS TO SELF-CHECK QUESTIONS

1. First add them to a panel, then add the panel to the north end of a frame.
2. Place them inside a panel with a `GridLayout` that has three rows and one column.
3. If you have many options, a set of radio buttons takes up a large area. A combo box can show many options without using up much space. But the user cannot see the options as easily.
4. When any of the component settings is changed, the program simply queries all of them and updates the label.
5. To keep it from growing too large. It would have grown to the same width and height as the two panels below it.
6. When you open a menu, you have not yet made a selection. Only `JMenuItem` objects correspond to selections.
7. The parameter variable is accessed in a method of an inner class.
8. `JColorChooser`.
9. Action events describe one-time changes, such as button clicks. Change events describe continuous changes.