CHAPTER 11

ADVANCED USER INTERFACES

CHAPTER GOALS

To use layout managers to arrange user-interface components in a container

To become familiar with common user-interface components, such as radio buttons, check boxes, and menus

To build programs that handle events generated by user-interface components

To browse the Java documentation effectively

CHAPTER CONTENTS

The graphical applications with which you are familiar have many visual gadgets for information entry: buttons, scroll bars, menus, and so on. In this chapter, you will learn how to use the most common user-interface components in the Java Swing toolkit, and how to search the Java documentation for information about other components. You will also learn more about event handling, so you can use timer events in animations and process mouse events in interactive graphical programs.

11.1 Layout Management

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

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

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

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

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

A layout manager arranges user-interface components.

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 components are laid out.

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.

Another commonly used layout manager is the **border layout**. The border layout groups components into five areas: center, north, south, west, and east (see Figure 1). Each area can hold a single component, or it can be empty.

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);
```

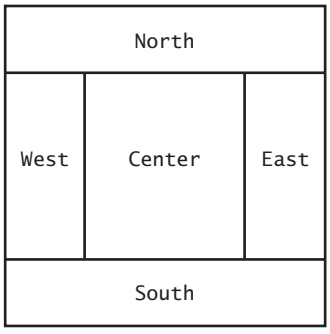


Figure 1
Components Expand to Fill Space in the Border Layout

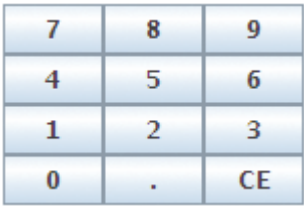


Figure 2 The Grid Layout

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

The **grid layout** manager arranges components in a grid with a fixed number of rows and columns. All components are resized so that they all have the same width and height. 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 example, Cay S. Horstmann and Gary Cornell, *Core Java 2 Volume 1: Fundamentals,* 8th edition (Prentice Hall, 2008), for more information. Java 6 introduced a *group layout* that is designed for use by interactive tools—see Programming Tip 11.1 on page W520.

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 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.

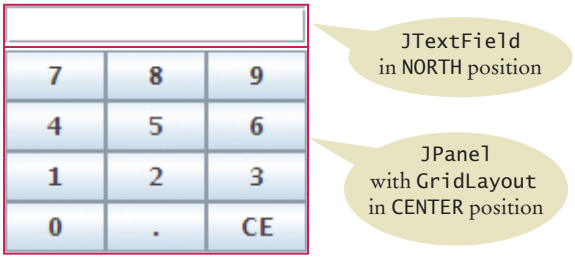


Figure 3 Nesting Panels

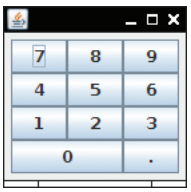
The following code produces the arrangement in Figure 3:

ONLINE EXAMPLE

The code for a calculator's user interface.

```
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);
```

1. What happens if you place two buttons in the northern position of a border layout? Try it out with a small program.
2. How do you add two buttons to the northern position of a frame so that they are shown next to each other?
3. How can you stack three buttons one above the other?
4. What happens when you place one button in the northern position of a border layout and another in the center position? Try it out with a small program if you aren't sure.
5. Some calculators have a double-wide 0 button, as shown below. How can you achieve that?



Practice It Now you can try these exercises at the end of the chapter: R11.1, R11.3, P11.1.

11.2 Choices

In the following sections, you will see how to present a finite set of choices to the user. Which Swing component you use depends on whether the choices are mutually exclusive or not, and on the amount of space you have for displaying the choices.

11.2.1 Radio Buttons

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

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,

In an old fashioned radio, pushing down one station button released the others.

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.

Add radio buttons to a ButtonGroup so that only one button in the group is selected at any time.

To create a set of radio buttons, first create each button individually, and then add all buttons in 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 in 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; }
```

Unfortunately, there is no convenient way of finding out which button in a group is currently selected. You have to call `isSelected` on each button. Because users will expect one radio button in a radio button group to be selected, call `setSelected(true)` on the default radio button before making the enclosing frame visible.

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

If you have multiple button groups, it is a good idea to group them together visually. It is a good idea to use a panel for each set of radio buttons, 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.

Figure 4 A Combo Box, Check Boxes, and Radio Buttons

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:

```
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"));
```

11.2.2 Check Boxes

For a binary choice, use a check box.

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.

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

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

Because check box settings do not exclude each other, you do not place a set of check boxes inside a button group.

As with radio buttons, you use the `isSelected` method to find out whether a check box is currently checked or not.

11.2.3 Combo Boxes

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

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).

Figure 5 An Open Combo Box

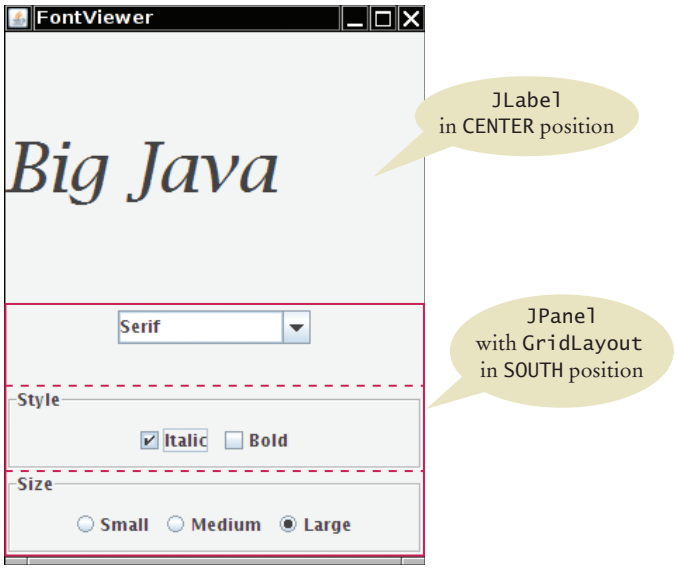


Figure 6 The Components of the `FontFrame`

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");
. . .
```

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, in our example, 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 label with the new font.

Figure 6 shows how the components are arranged in the frame.

section_2/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 FontFrame();
11       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12       frame.setTitle("FontViewer");
13       frame.setVisible(true);
14    }
15 }
```

section_2/FontFrame.java

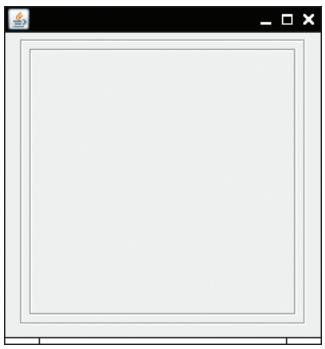
```
1  import java.awt.BorderLayout;
2  import java.awt.Font;
3  import java.awt.GridLayout;
4  import java.awt.event.ActionEvent;
5  import java.awt.event.ActionListener;
6  import javax.swing.ButtonGroup;
7  import javax.swing.JButton;
8  import javax.swing.JCheckBox;
9  import javax.swing.JComboBox;
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 sample and a control panel
19    to change the font of the text.
20 */
21 public class FontFrame extends JFrame
22 {
23    private static final int FRAME_WIDTH = 300;
24    private static final int FRAME_HEIGHT = 400;
25
26    private JLabel label;
27    private JCheckBox italicCheckBox;
28    private JCheckBox boldCheckBox;
29    private JRadioButton smallButton;
30    private JRadioButton mediumButton;
31    private JRadioButton largeButton;
32    private JComboBox facenameCombo;
33    private ActionListener listener;
34
35    /**
36       Constructs the frame.
37    */
38    public FontFrame()
39    {
40       // Construct text sample
41       label = new JLabel("Big Java");
42       add(label, BorderLayout.CENTER);
43
44       // This listener is shared among all components
45       listener = new ChoiceListener();
46
47       createControlPanel();
48       setLabelFont();
```

```
49          setSize(FRAME_WIDTH, FRAME_HEIGHT);
50       }
51
52       class ChoiceListener implements ActionListener
53       {
54          public void actionPerformed(ActionEvent event)
55          {
56             setLabelFont();
57          }
58       }
59
60       /**
61          Creates the control panel to change the font.
62       */
63       public void createControlPanel()
64       {
65          JPanel facenamePanel = createComboBox();
66          JPanel sizeGroupPanel = createCheckBoxes();
67          JPanel styleGroupPanel = createRadioButtons();
68
69          // Line up component panels
70
71          JPanel controlPanel = new JPanel();
72          controlPanel.setLayout(new GridLayout(3, 1));
73          controlPanel.add(facenamePanel);
74          controlPanel.add(sizeGroupPanel);
75          controlPanel.add(styleGroupPanel);
76
77          // Add panels to content pane
78
79          add(controlPanel, BorderLayout.SOUTH);
80       }
81
82       /**
83          Creates the combo box with the font style choices.
84          @return the panel containing the combo box
85       */
86       public JPanel createComboBox()
87       {
88          facenameCombo = new JComboBox();
89          facenameCombo.addItem("Serif");
90          facenameCombo.addItem("SansSerif");
91          facenameCombo.addItem("Monospaced");
92          facenameCombo.setEditable(true);
93          facenameCombo.addActionListener(listener);
94
95          JPanel panel = new JPanel();
96          panel.add(facenameCombo);
97          return panel;
98       }
99
100      /**
101         Creates the check boxes for selecting bold and italic styles.
102         @return the panel containing the check boxes
103      */
104      public JPanel createCheckBoxes()
105      {
106         italicCheckBox = new JCheckBox("Italic");
107         italicCheckBox.addActionListener(listener);
108
```

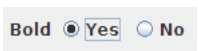
```
109         boldCheckBox = new JCheckBox("Bold");
110         boldCheckBox.addActionListener(listener);
111
112         JPanel panel = new JPanel();
113         panel.add(italicCheckBox);
114         panel.add(boldCheckBox);
115         panel.setBorder(new TitledBorder(new EtchedBorder(), "Style"));
116
117         return panel;
118      }
119
120      /**
121         Creates the radio buttons to select the font size.
122         @return the panel containing the radio buttons
123      */
124      public JPanel createRadioButtons()
125      {
126         smallButton = new JRadioButton("Small");
127         smallButton.addActionListener(listener);
128
129         mediumButton = new JRadioButton("Medium");
130         mediumButton.addActionListener(listener);
131
132         largeButton = new JRadioButton("Large");
133         largeButton.addActionListener(listener);
134         largeButton.setSelected(true);
135
136         // Add radio buttons to button group
137
138         ButtonGroup group = new ButtonGroup();
139         group.add(smallButton);
140         group.add(mediumButton);
141         group.add(largeButton);
142
143         JPanel panel = new JPanel();
144         panel.add(smallButton);
145         panel.add(mediumButton);
146         panel.add(largeButton);
147         panel.setBorder(new TitledBorder(new EtchedBorder(), "Size"));
148
149         return panel;
150      }
151
152      /**
153         Gets user choice for font name, style, and size
154         and sets the font of the text sample.
155      */
156      public void setLabelFont()
157      {
158         // Get font name
159         String facename = (String) facenameCombo.getSelectedItem();
160
161         // Get font style
162
163         int style = 0;
164         if (italicCheckBox.isSelected())
165         {
166            style = style + Font.ITALIC;
167         }
```

```
168         if (boldCheckBox.isSelected())
169         {
170            style = style + Font.BOLD;
171         }
172
173         // Get font size
174
175         int size = 0;
176
177         final int SMALL_SIZE = 24;
178         final int MEDIUM_SIZE = 36;
179         final int LARGE_SIZE = 48;
180
181         if (smallButton.isSelected()) { size = SMALL_SIZE; }
182         else if (mediumButton.isSelected()) { size = MEDIUM_SIZE; }
183         else if (largeButton.isSelected()) { size = LARGE_SIZE; }
184
185         // Set font of text field
186
187         label.setFont(new Font(facename, style, size));
188         label.repaint();
189      }
190   }
```

6. What is the advantage of a `JComboBox` over a set of radio buttons? What is the disadvantage?
7. What happens when you put two check boxes into a button group? Try it out if you are not sure.
8. How can you nest two etched borders, like this?



9. Why do all user-interface components in the `FontFrame` class share the same listener?
10. Why was the combo box placed inside a panel? What would have happened if it had been added directly to the control panel?
11. How could the following user interface be improved?



Practice It Now you can try these exercises at the end of the chapter: R11.11, P11.3, P11.4.

HOW TO 11.1 Laying Out a User Interface

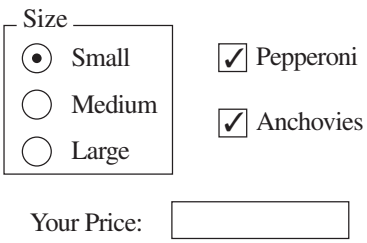
A graphical user interface is made up of components such as buttons and text fields. The Swing library uses containers and layout managers to arrange these components. This How To explains how to group components into containers and how to pick the right layout managers.

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

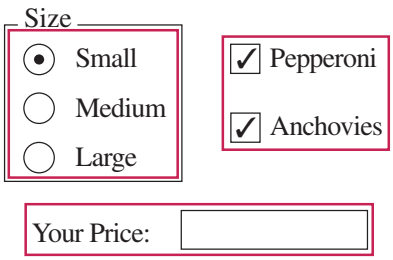
- Three radio buttons
- Two check boxes
- A label: "Your Price:"
- A text field
- A border



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:



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 with 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

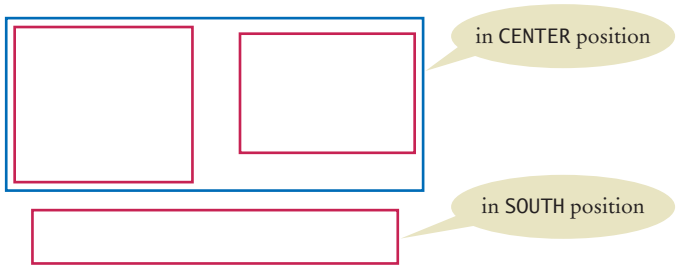
Step 4 Group the groups together.

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:

- 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 their panels. Continue in this fashion until you reach the outermost containers, which you add to the frame.

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

```
JPanel radioButtonPanel = new JPanel();
radioButtonPanel.setLayout(new GridLayout(3, 1));
radioButtonPanel.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 by default
pricePanel.add(new JLabel("Your Price: "));
pricePanel.add(priceTextField);
```

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

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

Programming Tip 11.1

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 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.

Java 6 introduced `GroupLayout`, a powerful layout manager that was specifically designed to be used by GUI builders. The free NetBeans development environment, available from `http://netbeans.org`, makes use of this layout manager—see Figure 7.

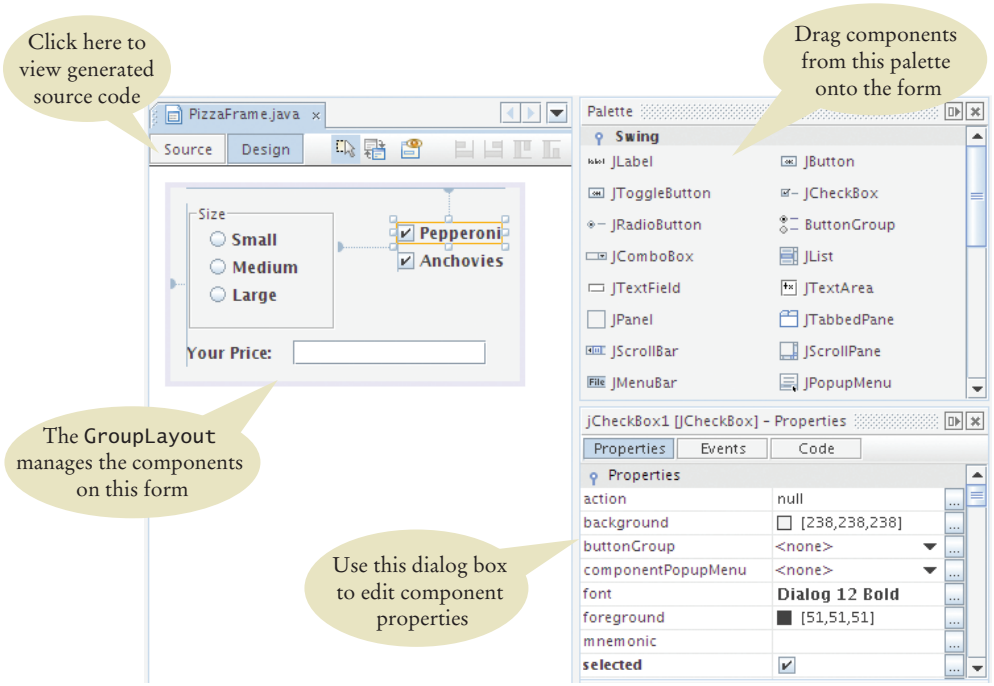


Figure 7 A GUI Builder

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.

WORKED EXAMPLE 11.1 **Programming a Working Calculator**

In this Worked Example, we implement arithmetic and scientific operations for a calculator. The sample program in Section 11.1 showed how to lay out the buttons for a simple calculator, and we use that program as a starting point.

11.3 Menus

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

Anyone who has ever used a graphical user interface is familiar with pull-down menus (see Figure 8). At the top of the frame is a *menu bar* that contains the top-level menus. Each menu is a collection of *menu items* and *submenus*.

The sample program for this section builds up a small but typical menu and traps the action events from the menu items. The program allows the user to specify the font for a label by selecting a face name, font size, and font style. In Java it is easy to create these menus.

You add the menu bar to the frame:

```
public class MyFrame extends JFrame
{
   public MyFrame()
   {
      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);
      . . .
   }
   . . .
}
```

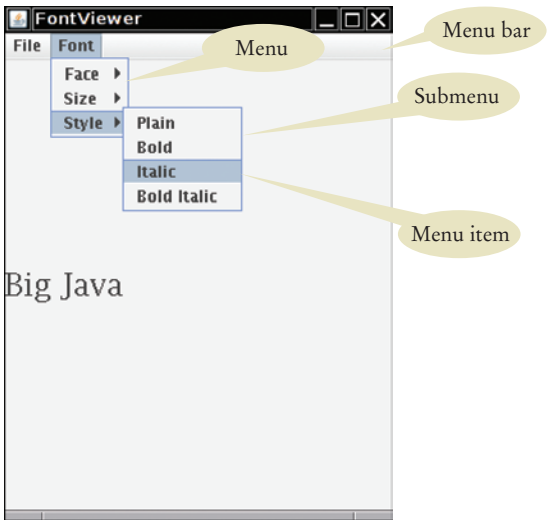


Figure 8
Pull-Down Menus

Available online in WileyPLUS and at www.wiley.com/college/horstmann.

A menu provides a list of available choices.

Menus are then added to the menu bar:

```
JMenu fileMenu = new JMenu("File");
JMenu fontMenu = new JMenu("Font");
menuBar.add(fileMenu);
menuBar.add(fontMenu);
```

You add menu items and submenus with the `add` method:

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

JMenu styleMenu = new JMenu("Style");
fontMenu.add(styleMenu); // A submenu
```

Menu items generate action events.

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:

```
ActionListener listener = new ExitItemListener();
exitItem.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.

To keep the program readable, it is a good idea to use a separate method for each menu or set of related menus. For example,

```
public JMenu createFaceMenu()
{
   JMenu menu = new JMenu("Face");
   menu.add(createFaceItem("Serif"));
   menu.add(createFaceItem("SansSerif"));
   menu.add(createFaceItem("Monospaced"));
   return menu;
}
```

Now consider the `createFaceItem` method. It has a string parameter variable for the name of the font face. When the item is selected, its action listener needs to

1. Set the current face name to the menu item text.
2. Make a new font from the current face, size, and style, and apply it to the label.

We have three menu items, one for each supported face name. Each of them needs to set a different name in the first step. Of course, we can make three listener classes `SerifListener`, `SansSerifListener`, and `MonospacedListener`, but that is not very elegant. After all, the actions only vary by a single string. We can store that string inside the listener class and then make three objects of the same listener class:

```
class FaceItemListener implements ActionListener
{
   private String name;

   public FaceItemListener(String newName) { name = newName; }
```

```
      public void actionPerformed(ActionEvent event)
      {
         faceName = name; // Sets an instance variable of the frame class
         setLabelFont();
      }
   }
```

Now we can install a listener object with the appropriate name:

```
public JMenuItem createFaceItem(String name)
{
   JMenuItem item = new JMenuItem(name);
   ActionListener listener = new FaceItemListener(name);
   item.addActionListener(listener);
   return item;
}
```

This approach is still a bit tedious. We can do better by using a local inner class (see Special Topic 10.2). When we move the declaration of the inner class inside the `createFaceItem` method, the `actionPerformed` method can access the `name` parameter variable directly. However, we need to observe a technical rule. Because `name` is a local variable, it must be declared as `final` to be accessible from an inner class method.

```
public JMenuItem createFaceItem(final String name)
// Final variables can be accessed from an inner class method
{
   class FaceItemListener implements ActionListener // A local inner class
   {
      public void actionPerformed(ActionEvent event)
      {
         facename = name; // Accesses the local variable name
         setLabelFont();
      }
   }

   JMenuItem item = new JMenuItem(name);
   ActionListener listener = new FaceItemListener();
   item.addActionListener(listener);
   return item;
}
```

The same strategy is used for the `createSizeItem` and `createStyleItem` methods.

section_3/FontViewer2.java

```
1  import javax.swing.JFrame;
2
3  /**
4     This program uses a menu to display font effects.
5  */
6  public class FontViewer2
7  {
8     public static void main(String[] args)
9     {
10       JFrame frame = new FontFrame2();
11       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12       frame.setTitle("FontViewer");
13       frame.setVisible(true);
14    }
15 }
```

section_3/FontFrame2.java

```
1  import java.awt.BorderLayout;
2  import java.awt.Font;
3  import java.awt.event.ActionEvent;
4  import java.awt.event.ActionListener;
5  import javax.swing.JFrame;
6  import javax.swing.JLabel;
7  import javax.swing.JMenu;
8  import javax.swing.JMenuBar;
9  import javax.swing.JMenuItem;
10
11 /**
12    This frame has a menu with commands to change the font
13    of a text sample.
14 */
15 public class FontFrame2 extends JFrame
16 {
17    private static final int FRAME_WIDTH = 300;
18    private static final int FRAME_HEIGHT = 400;
19
20    private JLabel label;
21    private String facename;
22    private int fontstyle;
23    private int fontsize;
24
25    /**
26       Constructs the frame.
27    */
28    public FontFrame2()
29    {
30       // Construct text sample
31       label = new JLabel("Big Java");
32       add(label, BorderLayout.CENTER);
33
34       // Construct menu
35       JMenuBar menuBar = new JMenuBar();
36       setJMenuBar(menuBar);
37       menuBar.add(createFileMenu());
38       menuBar.add(createFontMenu());
39
40       facename = "Serif";
41       fontsize = 24;
42       fontstyle = Font.PLAIN;
43
44       setLabelFont();
45       setSize(FRAME_WIDTH, FRAME_HEIGHT);
46    }
47
48    class ExitItemListener implements ActionListener
49    {
50       public void actionPerformed(ActionEvent event)
51       {
52          System.exit(0);
53       }
54    }
55
56    /**
57       Creates the File menu.
```

```
58          @return the menu
59       */
60       public JMenu createFileMenu()
61       {
62          JMenu menu = new JMenu("File");
63          JMenuItem exitItem = new JMenuItem("Exit");
64          ActionListener listener = new ExitItemListener();
65          exitItem.addActionListener(listener);
66          menu.add(exitItem);
67          return menu;
68       }
69
70       /**
71          Creates the Font submenu.
72          @return the menu
73       */
74       public JMenu createFontMenu()
75       {
76          JMenu menu = new JMenu("Font");
77          menu.add(createFaceMenu());
78          menu.add(createSizeMenu());
79          menu.add(createStyleMenu());
80          return menu;
81       }
82
83       /**
84          Creates the Face submenu.
85          @return the menu
86       */
87       public JMenu createFaceMenu()
88       {
89          JMenu menu = new JMenu("Face");
90          menu.add(createFaceItem("Serif"));
91          menu.add(createFaceItem("SansSerif"));
92          menu.add(createFaceItem("Monospaced"));
93          return menu;
94       }
95
96        /**
97          Creates the Size submenu.
98          @return the menu
99       */
100      public JMenu createSizeMenu()
101      {
102         JMenu menu = new JMenu("Size");
103         menu.add(createSizeItem("Smaller", -1));
104         menu.add(createSizeItem("Larger", 1));
105         return menu;
106      }
107
108      /**
109         Creates the Style submenu.
110         @return the menu
111      */
112      public JMenu createStyleMenu()
113      {
114         JMenu menu = new JMenu("Style");
115         menu.add(createStyleItem("Plain", Font.PLAIN));
116         menu.add(createStyleItem("Bold", Font.BOLD));
```

```
117        menu.add(createStyleItem("Italic", Font.ITALIC));
118        menu.add(createStyleItem("Bold Italic", Font.BOLD
119              + Font.ITALIC));
120        return menu;
121     }
122
123     /**
124        Creates a menu item to change the font face and set its action listener.
125        @param name the name of the font face
126        @return the menu item
127     */
128     public JMenuItem createFaceItem(final String name)
129     {
130        class FaceItemListener implements ActionListener
131        {
132           public void actionPerformed(ActionEvent event)
133           {
134              facename = name;
135              setLabelFont();
136           }
137        }
138
139        JMenuItem item = new JMenuItem(name);
140        ActionListener listener = new FaceItemListener();
141        item.addActionListener(listener);
142        return item;
143     }
144
145     /**
146        Creates a menu item to change the font size
147        and set its action listener.
148        @param name the name of the menu item
149        @param increment the amount by which to change the size
150        @return the menu item
151     */
152     public JMenuItem createSizeItem(String name, final int increment)
153     {
154        class SizeItemListener implements ActionListener
155        {
156           public void actionPerformed(ActionEvent event)
157           {
158              fontsize = fontsize + increment;
159              setLabelFont();
160           }
161        }
162
163        JMenuItem item = new JMenuItem(name);
164        ActionListener listener = new SizeItemListener();
165        item.addActionListener(listener);
166        return item;
167     }
168
169     /**
170        Creates a menu item to change the font style
171        and set its action listener.
172        @param name the name of the menu item
173        @param style the new font style
174        @return the menu item
175     */
```

```
176    public JMenuItem createStyleItem(String name, final int style)
177    {
178       class StyleItemListener implements ActionListener
179       {
180          public void actionPerformed(ActionEvent event)
181          {
182             fontstyle = style;
183             setLabelFont();
184          }
185       }
186
187       JMenuItem item = new JMenuItem(name);
188       ActionListener listener = new StyleItemListener();
189       item.addActionListener(listener);
190       return item;
191    }
192
193    /**
194       Sets the font of the text sample.
195    */
196    public void setLabelFont()
197    {
198       Font f = new Font(facename, fontstyle, fontsize);
199       label.setFont(f);
200    }
201 }
```

12. Why do `JMenu` objects not generate action events?
13. Can you add a menu item directly to the menu bar? Try it out. What happens?
14. Why is the `increment` parameter variable in the `createSizeItem` method declared as `final`?
15. Why can't the `createFaceItem` method simply set the `faceName` instance variable, like this:

```
class FaceItemListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      setLabelFont();
   }
}

public JMenuItem createFaceItem(String name)
{
   JMenuItem item = new JMenuItem(name);
   faceName = name;
   ActionListener listener = new FaceItemListener();
   item.addActionListener(listener);
   return item;
}
```

16. In this program, the font specification (name, size, and style) is stored in instance variables. Why was this not necessary in the program of the previous section?

Practice It Now you can try these exercises at the end of the chapter: R11.12, P11.6, P11.7.

11.4 Exploring the Swing Documentation

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

In order to use the Swing library effectively, you need to study the API 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 may find the documentation intimidating at first glance, though. The purpose of this section is to show you how you can use the documentation to your advantage without being overwhelmed.

As an example, consider a program for mixing colors by specifying the red, green, and blue values. How can you specify the colors? Of course, you could supply three text fields, but sliders would be more convenient for users of your program (see Figure 9).

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. Or you can run the sample application included in the Java Development Kit that shows off all Swing components (see Figure 10). 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.

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?

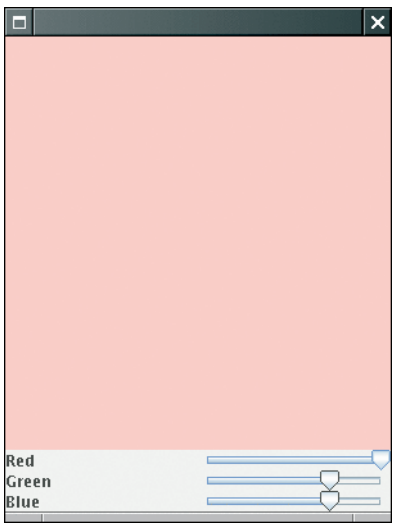
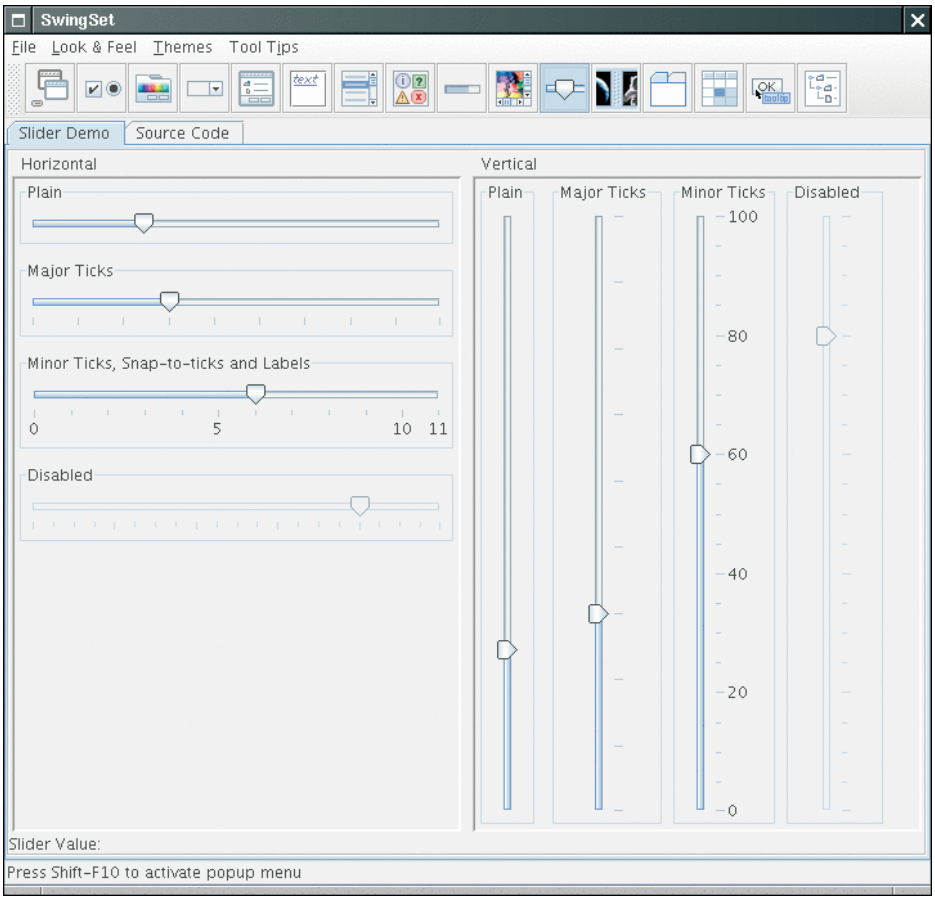


Figure 9 A Color Viewer with Sliders

Figure 10
The SwingSet Demo



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 11. 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

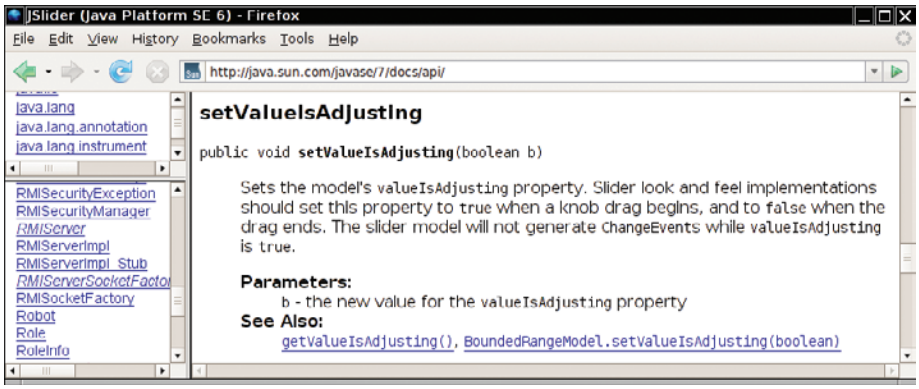


Figure 11 A Mysterious Method Description from the API Documentation

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.

Let's 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.

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

```
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* 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 let's make 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.

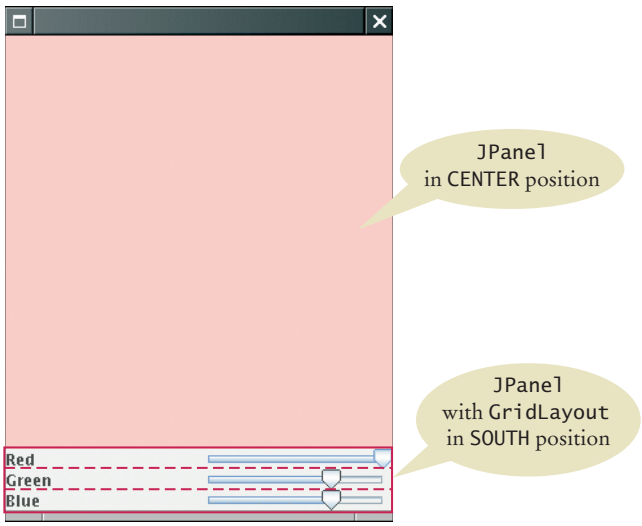


Figure 12 The Components of the Color Viewer Frame

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. After having mastered the basics, you may want to explore the capabilities of the component further, for example by adding tick marks—see Exercise P11.9.

Figure 12 shows how the components are arranged in the frame.

section_4/ColorViewer.java

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

section_4/ColorFrame.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 ColorFrame extends JFrame
12 {
13    private static final int FRAME_WIDTH = 300;
14    private static final int FRAME_HEIGHT = 400;
```

```
15
16    private JPanel colorPanel;
17    private JSlider redSlider;
18    private JSlider greenSlider;
19    private JSlider blueSlider;
20
21    public ColorFrame()
22    {
23       colorPanel = new JPanel();
24
25       add(colorPanel, BorderLayout.CENTER);
26       createControlPanel();
27       setSampleColor();
28       setSize(FRAME_WIDTH, FRAME_HEIGHT);
29    }
30
31    class ColorListener implements ChangeListener
32    {
33       public void stateChanged(ChangeEvent event)
34       {
35          setSampleColor();
36       }
37    }
38
39    public void createControlPanel()
40    {
41       ChangeListener listener = new ColorListener();
42
43       redSlider = new JSlider(0, 255, 255);
44       redSlider.addChangeListener(listener);
45
46       greenSlider = new JSlider(0, 255, 175);
47       greenSlider.addChangeListener(listener);
48
49       blueSlider = new JSlider(0, 255, 175);
50       blueSlider.addChangeListener(listener);
51
52       JPanel controlPanel = new JPanel();
53       controlPanel.setLayout(new GridLayout(3, 2));
54
55       controlPanel.add(new JLabel("Red"));
56       controlPanel.add(redSlider);
57
58       controlPanel.add(new JLabel("Green"));
59       controlPanel.add(greenSlider);
60
61       controlPanel.add(new JLabel("Blue"));
62       controlPanel.add(blueSlider);
63
64       add(controlPanel, BorderLayout.SOUTH);
65    }
66
67    /**
68       Reads the slider values and sets the panel to
69       the selected color.
70    */
71    public void setSampleColor()
72    {
73       // Read slider values
74
```

```
75         int red = redSlider.getValue();
76         int green = greenSlider.getValue();
77         int blue = blueSlider.getValue();
78
79         // Set panel background to selected color
80
81         colorPanel.setBackground(new Color(red, green, blue));
82         colorPanel.repaint();
83      }
84   }
```

17. 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.
18. Why does a slider emit change events and not action events?

Practice It Now you can try these exercises at the end of the chapter: R11.14, P11.2, P11.9.

11.5 Using Timer Events for Animations

In this section we introduce timer events and show how you can use them to implement simple animations.

The `Timer` class in the `javax.swing` package generates a sequence of action events, spaced at even time intervals. (You can think of a timer as an invisible button that is automatically clicked.) This is useful whenever you want to send continuous updates to a component. For example, in an animation, you may want to update a scene ten times per second and redisplay the image to give the illusion of movement.

A timer generates action events at fixed intervals.

When you use a timer, you specify the frequency of the events and an object of a class that implements the `ActionListener` interface. Place whatever action you want to occur inside the `actionPerformed` method. Finally, start the timer.

```
class MyListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      Action that is executed at each timer event
   }
}

MyListener listener = new MyListener();
Timer t = new Timer(interval, listener);
t.start();
```

Then the timer calls the `actionPerformed` method of the `listener` object every `interval` milliseconds.

A Swing timer notifies a listener with each "tick".

Our sample program will display a moving rectangle. We first supply a `RectangleComponent` class with a `moveRectangleBy` method that moves the rectangle by a given amount.

section_5/RectangleComponent.java

```
1  import java.awt.Graphics;
2  import javax.swing.JComponent;
3
4  /**
5     This component displays a rectangle that can be moved.
6  */
7  public class RectangleComponent extends JComponent
8  {
9     private static final int RECTANGLE_WIDTH = 20;
10    private static final int RECTANGLE_HEIGHT = 30;
11
12    private int xLeft;
13    private int yTop;
14
15    public RectangleComponent()
16    {
17       xLeft = 0;
18       yTop = 0;
19    }
20
21    public void paintComponent(Graphics g)
22    {
23       g.fillRect(xLeft, yTop, RECTANGLE_WIDTH, RECTANGLE_HEIGHT);
24    }
25
26    /**
27       Moves the rectangle by a given amount.
28       @param dx the amount to move in the x-direction
29       @param dy the amount to move in the y-direction
30    */
31    public void moveRectangleBy(int dx, int dy)
32    {
33       xLeft = xLeft + dx;
34       yTop = yTop + dy;
35       repaint();
36    }
37 }
```

To make an animation, the timer listener should update and repaint a component several times per second.

Note the call to `repaint` in the `moveRectangleBy` method. This call is necessary to ensure that the component is repainted after the position of the rectangle has been changed. The call to `repaint` forces a call to the `paintComponent` method. The `paintComponent` method redraws the component, causing the rectangle to appear at the updated location.

The `actionPerformed` method of the timer listener moves the rectangle one pixel down and to the right:

```
scene.moveRectangleBy(1, 1);
```

Because the `actionPerformed` method is called many times per second, the rectangle appears to move smoothly across the frame.

section_5/RectangleFrame.java

```
1  import java.awt.event.ActionEvent;
2  import java.awt.event.ActionListener;
3  import javax.swing.JFrame;
4  import javax.swing.Timer;
5
6  /**
7     This frame contains a moving rectangle.
8  */
9  public class RectangleFrame extends JFrame
10 {
11    private static final int FRAME_WIDTH = 300;
12    private static final int FRAME_HEIGHT = 400;
13
14    private RectangleComponent scene;
15
16    class TimerListener implements ActionListener
17    {
18       public void actionPerformed(ActionEvent event)
19       {
20          scene.moveRectangleBy(1, 1);
21       }
22    }
23
24    public RectangleFrame()
25    {
26       scene = new RectangleComponent();
27       add(scene);
28
29       setSize(FRAME_WIDTH, FRAME_HEIGHT);
30
31       ActionListener listener = new TimerListener();
32
33       final int DELAY = 100; // Milliseconds between timer ticks
34       Timer t = new Timer(DELAY, listener);
35       t.start();
36    }
37 }
```

19. Why does a timer require a listener object?
20. How can you make the rectangle move backwards?
21. Describe two ways of modifying the program so that the rectangle moves twice as fast.
22. How can you make a car move instead of a rectangle?
23. How can you make two rectangles move in parallel in the scene?
24. What would happen if you omitted the call to `repaint` in the `moveRectangleBy` method?

Practice It Now you can try these exercises at the end of the chapter: P11.12, P11.13, P11.14.

11.6 Mouse Events

You use a mouse listener to capture mouse events.

If you write programs that show drawings, and you want users to manipulate the drawings with a mouse, then you need to listen to mouse events. Mouse listeners are more complex than action listeners, the listeners that process button clicks and timer ticks.

A mouse listener must implement the `MouseListener` interface, which contains the following five methods:

In Swing, a mouse event isn't a gathering of rodents; it's notification of a mouse click by the program user.

```
public interface MouseListener
{
   void mousePressed(MouseEvent event);
      // Called when a mouse button has been pressed on a component
   void mouseReleased(MouseEvent event);
      // Called when a mouse button has been released on a component
   void mouseClicked(MouseEvent event);
      // Called when the mouse has been clicked on a component
   void mouseEntered(MouseEvent event);
      // Called when the mouse enters a component
   void mouseExited(MouseEvent event);
      // Called when the mouse exits a component
}
```

The `mousePressed` and `mouseReleased` methods are called whenever a mouse button is pressed or released. If a button is pressed and released in quick succession, and the mouse has not moved, then the `mouseClicked` method is called as well. The `mouseEntered` and `mouseExited` methods can be used to highlight a user-interface component whenever the mouse is pointing inside it.

The most commonly used method is `mousePressed`. Users generally expect that their actions are processed as soon as the mouse button is pressed.

You add a mouse listener to a component by calling the `addMouseListener` method:

```
public class MyMouseListener implements MouseListener
{
   public void mousePressed(MouseEvent event)
   {
      int x = event.getX();
      int y = event.getY();
      Process mouse event at (x, y)
   }

   // Do-nothing methods
   public void mouseReleased(MouseEvent event) {}
   public void mouseClicked(MouseEvent event) {}
   public void mouseEntered(MouseEvent event) {}
   public void mouseExited(MouseEvent event) {}
}

MouseListener listener = new MyMouseListener();
component.addMouseListener(listener);
```

In our sample program, a user clicks on a component containing a rectangle. Whenever the mouse button is pressed, the rectangle is moved to the mouse location. We

first enhance the `RectangleComponent` class and add a `moveRectangleTo` method to move the rectangle to a new position.

section_6/RectangleComponent2.java

```
1  import java.awt.Graphics;
2  import java.awt.Rectangle;
3  import javax.swing.JComponent;
4
5  /**
6     This component displays a rectangle that can be moved.
7  */
8  public class RectangleComponent2 extends JComponent
9  {
10    private static final int RECTANGLE_WIDTH = 20;
11    private static final int RECTANGLE_HEIGHT = 30;
12
13    private int xLeft;
14    private int yTop;
15
16    public RectangleComponent2()
17    {
18       xLeft = 0;
19       yTop = 0;
20    }
21
22    public void paintComponent(Graphics g)
23    {
24       g.fillRect(xLeft, yTop, RECTANGLE_WIDTH, RECTANGLE_HEIGHT);
25    }
26
27    /**
28       Moves the rectangle to the given location.
29       @param x the x-position of the new location
30       @param y the y-position of the new location
31    */
32    public void moveRectangleTo(int x, int y)
33    {
34       xLeft = x;
35       yTop = y;
36       repaint();
37    }
38 }
```

Note the call to `repaint` in the `moveRectangleTo` method. As you saw before, this call causes the component to repaint itself and show the rectangle in the new position.

Now, add a mouse listener to the component. Whenever the mouse is pressed, the listener moves the rectangle to the mouse location.

```
class MousePressListener implements MouseListener
{
   public void mousePressed(MouseEvent event)
   {
      int x = event.getX();
      int y = event.getY();
      scene.moveRectangleTo(x, y);
   }
   . . .
}
```

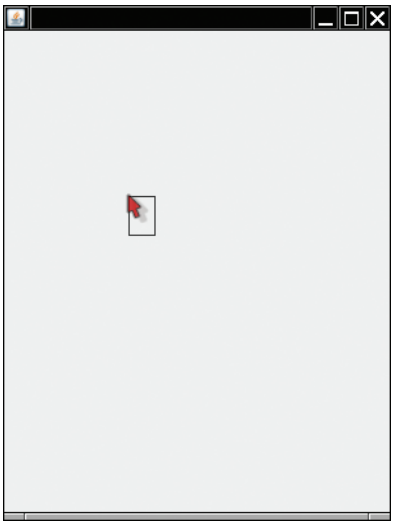


Figure 13 Clicking the Mouse Moves the Rectangle

It often happens that a particular listener specifies actions only for one or two of the listener methods. Nevertheless, all five methods of the interface must be implemented. The unused methods are simply implemented as do-nothing methods.

Go ahead and run the `RectangleViewer2` program. Whenever you click the mouse inside the frame, the top-left corner of the rectangle moves to the mouse pointer (see Figure 13).

section_6/RectangleViewer2.java

```
1  import javax.swing.JFrame;
2
3  /**
4     This program displays a rectangle that can be moved with the mouse.
5  */
6  public class RectangleViewer2
7  {
8     public static void main(String[] args)
9     {
10       JFrame frame = new RectangleFrame2();
11       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12       frame.setVisible(true);
13    }
14 }
```

section_6/RectangleFrame2.java

```
1  import java.awt.event.MouseListener;
2  import java.awt.event.MouseEvent;
3  import javax.swing.JFrame;
4
5  /**
6     This frame contains a moving rectangle.
7  */
8  public class RectangleFrame2 extends JFrame
9  {
```

```
10     private static final int FRAME_WIDTH = 300;
11     private static final int FRAME_HEIGHT = 400;
12
13     private RectangleComponent2 scene;
14
15     class MousePressListener implements MouseListener
16     {
17        public void mousePressed(MouseEvent event)
18        {
19           int x = event.getX();
20           int y = event.getY();
21           scene.moveRectangleTo(x, y);
22        }
23
24        // Do-nothing methods
25        public void mouseReleased(MouseEvent event) {}
26        public void mouseClicked(MouseEvent event) {}
27        public void mouseEntered(MouseEvent event) {}
28        public void mouseExited(MouseEvent event) {}
29     }
30
31     public RectangleFrame2()
32     {
33        scene = new RectangleComponent2();
34        add(scene);
35
36        MouseListener listener = new MousePressListener();
37        scene.addMouseListener(listener);
38
39        setSize(FRAME_WIDTH, FRAME_HEIGHT);
40     }
41  }
```

25. Why was the `moveRectangleBy` method in `RectangleComponent2` replaced with a `moveRectangleTo` method?
26. Why must the `MousePressListener` class supply five methods?
27. How could you change the behavior of the program so that a new rectangle is added whenever the mouse is clicked?

Practice It Now you can try these exercises at the end of the chapter: R11.21, P11.22, P11.23.

Special Topic 11.1

Keyboard Events

If you program a game, you may want to process keystrokes, such as the arrow keys. Add a key listener to the component on which you draw the game scene. The `KeyListener` interface has three methods. As with a mouse listener, you are most interested in key press events, and you can leave the other two methods empty. Your key listener class should look like this:

```
class MyKeyListener implements KeyListener
{
   public void keyPressed(KeyEvent event)
   {
      String key = KeyStroke.getKeyStrokeForEvent(event).toString();
      key = key.replace("pressed ", "");
      Process key.
```

```
   }

   // Do-nothing methods
   public void keyReleased(KeyEvent event) {}
   public void keyTyped(KeyEvent event) {}
}
```

Whenever the program user presses a key, a key event is generated.

The call `KeyStroke.getKeyStrokeForEvent(event).toString()` turns the event object into a text description of the key, such as `"pressed LEFT"`. In the next line, we eliminate the `"pressed "` prefix. The remainder is a string such as `"LEFT"` or `"A"` that describes the key that was pressed. You can find a list of all key names in the API documentation of the `KeyStroke` class.

ONLINE EXAMPLE

A complete program that uses the arrow keys to move a rectangle.

As always, remember to attach the listener to the event source:

```
KeyListener listener = new MyKeyListener();
scene.addKeyListener(listener);
```

In order to receive key events, your component must call

```
scene.setFocusable(true);
```

Special Topic 11.2

Event Adapters

In the preceding section you saw how to install a mouse listener in a mouse event source and how the listener methods are called when an event occurs. Usually, a program is not interested in all listener notifications. For example, a program may only be interested in mouse clicks and may not care that these mouse clicks are composed of "mouse pressed" and "mouse released" events. Of course, the program could supply a listener that declares all those methods in which it has no interest as "do-nothing" methods, for example:

```
class MouseClickListener implements MouseListener
{
   public void mouseClicked(MouseEvent event)
   {
      Mouse click action
   }

   // Four do-nothing methods
   public void mouseEntered(MouseEvent event) {}
   public void mouseExited(MouseEvent event) {}
   public void mousePressed(MouseEvent event) {}
   public void mouseReleased(MouseEvent event) {}
}
```

To avoid this labor, some friendly soul has created a `MouseAdapter` class that implements the `MouseListener` interface such that all methods do nothing. You can *extend* that class, inheriting the do-nothing methods and overriding the methods that you care about, like this:

```
class MouseClickListener extends MouseAdapter
{
   public void mouseClicked(MouseEvent event)
   {
      Mouse click action
   }
}
```

There is also a `KeyAdapter` that implements the `KeyListener` interface (see Special Topic 11.1), providing three do-nothing methods.

WORKED EXAMPLE 11.2

Adding Mouse and Keyboard Support to the Bar Chart Creator

In this Worked Example, we will enhance the bar chart creator of Worked Example 10.1 and add support for mouse and keyboard operations.

VIDEO EXAMPLE 11.1

Designing a Baby Naming Program

In this Video Example, you will see how to design a user interface for a program that suggests baby names.

CHAPTER SUMMARY

Learn how to arrange multiple components in a container.

- User-interface components are arranged by placing them inside containers. Containers can be placed inside larger containers.
- Each container has a layout manager that directs the arrangement of its components.
- Three useful layout managers are the border layout, flow layout, and grid layout.
- When adding a component to a container with the border layout, specify the `NORTH`, `SOUTH`, `WEST`, `EAST`, or `CENTER` position.
- The content pane of a frame has a border layout by default. A panel has a flow layout by default.

Select among the Swing components for presenting choices to the user.

- For a small set of mutually exclusive choices, use a group of radio buttons or a combo box.
- Add radio buttons to a `ButtonGroup` so that only one button in the group is selected at any time.
- You can place a border around a panel to group its contents visually.
- For a binary choice, use a check box.
- For a large set of choices, use a combo box.
- Radio buttons, check boxes, and combo boxes generate action events, just as buttons do.

Implement menus in a Swing program.

- A frame contains a menu bar. The menu bar contains menus. A menu contains submenus and menu items.
- Menu items generate action events.

Available online in WileyPLUS and at `www.wiley.com/college/horstmann`.

Use the Swing documentation.

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

Use timer events to implement animations.

- A timer generates action events at fixed intervals.
- To make an animation, the timer listener should update and repaint a component several times per second.

Write programs that process mouse events.

- You use a mouse listener to capture mouse events.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

```
java.awt.BorderLayout
   CENTER
   EAST
   NORTH
   SOUTH
   WEST
java.awt.Component
   addKeyListener
   addMouseListener
   setFocusable
java.awt.Container
   setLayout
java.awt.FlowLayout
java.awt.Font
   BOLD
   ITALIC
java.awt.GridLayout
java.awt.event.KeyEvent
java.awt.event.KeyListener
   keyPressed
   keyReleased
   keyTyped
java.awt.event.MouseEvent
   getX
   getY
java.awt.event.MouseListener
   mouseClicked
   mouseEntered
   mouseExited
   mousePressed
   mouseReleased
javax.swing.AbstractButton
   isSelected
   setSelected
javax.swing.ButtonGroup
   add
javax.swing.JCheckBox
javax.swing.JComboBox
   addItem
   getSelectedItem
   isEditable
   setEditable
   setSelectedItem
javax.swing.JComponent
   setBorder
   setFocusable
   setFont
javax.swing.JFrame
   setJMenuBar
javax.swing.JMenu
   add
javax.swing.JMenuBar
   add
javax.swing.JMenuItem
javax.swing.JRadioButton
javax.swing.JSlider
   addChangeListener
   getValue
javax.swing.KeyStroke
   getKeyStrokeForEvent
javax.swing.Timer
   start
   stop
javax.swing.border.EtchedBorder
javax.swing.border.TitledBorder
javax.swing.event.ChangeEvent
javax.swing.event.ChangeListener
   stateChanged
```

REVIEW EXERCISES

■ **R11.1** Can you use a flow layout for the components in a frame? If yes, how?

■ **R11.2** What is the advantage of a layout manager over telling the container "place this component at position (x, y)"?

■■ **R11.3** 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.

■■ **R11.4** 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.

■■ **R11.5** What happens when you add a button to a container that uses a border layout and omit the position? Try it out and explain.

■■ **R11.6** What happens when you try to add a button to another button? Try it out and explain.

■■ **R11.7** The control panel in Section 11.4 uses a grid layout manager. Explain a drawback of the grid that is apparent in Figure 12. What could you do to overcome this drawback?

■■■ **R11.8** What is the difference between the grid layout and the grid bag layout?

■■■ **R11.9** 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.

■ **R11.10** What is the difference between radio buttons and check boxes?

■ **R11.11** Why do you need a button group for radio buttons but not for check boxes?

■ **R11.12** What is the difference between a menu bar, a menu, and a menu item?

■ **R11.13** When browsing through the Java documentation for more information about sliders, we ignored the `JSlider` constructor with no arguments. Why? Would it have worked in our sample program?

■ **R11.14** How do you construct a vertical slider? Consult the Swing documentation for an answer.

■■ **R11.15** Why doesn't a `JComboBox` send out change events?

■■■ **R11.16** What component would you use to show a set of choices, as in a combo box, but so that several items are visible at the same time? Run the Swing demo application or look at a book with Swing example programs to find the answer.

■■ **R11.17** How many Swing user-interface components are there? Look at the Java documentation to get an approximate answer.

■■ **R11.18** How many methods does the `JProgressBar` component have? Be sure to count inherited methods. Look at the Java documentation.

■ **R11.19** What is the difference between an `ActionEvent` and a `MouseEvent`?

■■ **R11.20** What information does an action event object carry? What additional information does a mouse event object carry? *Hint:* Check the API documentation.

■■ **R11.21** Why does the `ActionListener` interface have only one method, whereas the `MouseListener` has five methods?

PROGRAMMING EXERCISES

■ **P11.1** 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.

■■ **P11.2** Add icons to the buttons of Exercise P11.1. Use a `JButton` constructor with an `Icon` argument and supply an `ImageIcon`.

■ **P11.3** 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.

■ **P11.4** Write an application with three check boxes labeled "Red", "Green", and "Blue" that adds a red, green, or blue component to the background color of a panel in the center of the frame. This application can display a total of eight color combinations.

■ **P11.5** Write an application with a combo box containing three items labeled "Red", "Green", and "Blue" that change the background color of a panel in the center of the frame to red, green, or blue.

■ **P11.6** Write an application with a Color menu and menu items labeled "Red", "Green", and "Blue" that change the background color of a panel in the center of the frame to red, green, or blue.

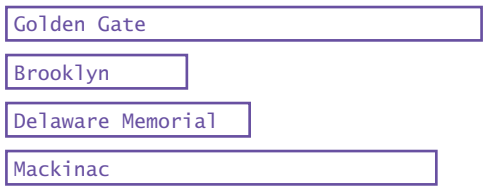
■ **P11.7** Write a program that displays a number of rectangles at random positions. Supply menu items "Fewer" and "More" that generate fewer or more random rectangles. Each time the user selects "Fewer", the count should be halved. Each time the user clicks on "More", the count should be doubled.

■■ **P11.8** Modify the program of Exercise P11.7 to replace the buttons with a slider to generate more or fewer random rectangles.

■■ **P11.9** Modify the slider program in Section 11.4 to add a set of tick marks to each slider that show the exact slider position.

■■■ **P11.10** Enhance the font viewer program to allow the user to select different font faces. Research the API documentation to find out how to find the available fonts on the user's system.

■■■ **P11.11** 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.

■ **P11.12** Write a program that uses a timer to print the current time once a second. *Hint:* The following code prints the current time:

```
Date now = new Date();
System.out.println(now);
```

The `Date` class is in the `java.util` package.

■■■ **P11.13** Change the `RectangleComponent` for the animation in Section 11.5 so that the rectangle bounces off the edges of the component rather than simply moving outside.

■■ **P11.14** Change the rectangle animation in Section 11.5 so that it shows two rectangles moving in opposite directions.

■■ **P11.15** Write a program that animates a car so that it moves across a frame.

■■■ **P11.16** Write a program that animates two cars moving across a frame in opposite directions (but at different heights so that they don't collide.)

■■■ **P11.17** 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.

■ **P11.18** Change the `RectangleComponent` for the mouse listener program in Section 11.6 so that a new rectangle is added to the component whenever the mouse is clicked. *Hint:* Store all points on which the user clicked, and draw all rectangles in the `paintComponent` method.

■ **P11.19** Write a program that prompts the user to enter the *x*- and *y*-positions of a center point and a radius, using text fields.When the user clicks a "Draw" button, draw a circle with that center and radius in a component.

■■ **P11.20** Write a program that allows the user to specify a circle by typing the radius in a text field and then clicking on the center. Note that you don't need a "Draw" button.

■ **P11.21** Write a program that allows the user to specify a circle with two mouse presses, the first one on the center and the second on a point on the periphery. *Hint:* In the mouse press handler, you must keep track of whether you already received the center point in a previous mouse press.

■■■ **P11.22** Write a program that allows the user to specify a triangle with three mouse presses. After the first mouse press, draw a small dot. After the second mouse press, draw a line joining the first two points. After the third mouse press, draw the entire triangle. The fourth mouse press erases the old triangle and starts a new one.

■■■ **P11.23** Implement a program that allows two players to play tic-tac-toe. Draw the game grid and an indication of whose turn it is (X or O). Upon the next click, check that the mouse click falls into an empty location, fill the location with the mark of the current player, and give the other player a turn. If the game is won, indicate the winner. Also supply a button for starting over.

■■■ **P11.24** Write a program that lets users design bar charts with a mouse. When the user clicks inside a bar, the next mouse click extends the length of the bar to the *x*-coordinate of the mouse click. (If it is at or near 0, the bar is removed.) When the user clicks below the last bar, a new bar is added whose length is the *x*-coordinate of the mouse click.

■■ **Business P11.25** Write a program with a graphical interface that allows the user to convert an amount of money between U.S. dollars (USD), euros (EUR), and British pounds (GBP). The user interface should have the following elements: a text box to enter the amount to be converted, two combo boxes to allow the user to select the currencies, a button to make the conversion, and a label to show the result. Display a warning if the user does not choose different currencies. Use the following conversion rates:

1 EUR is equal to 1.42 USD.

1 GBP is equal to 1.64 USD.

1 GBP is equal to 1.13 EUR.

■■ **Business P11.26** Write a program with a graphical interface that implements a login window with text fields for the user name and password. When the login is successful, hide the login window and open a new window with a welcome message. Follow these rules for validating the password:

1. The user name is not case sensitive.
2. The password is case sensitive.
3. The user has three opportunities to enter valid credentials.

Otherwise, display an error message and terminate the program. When the program starts, read the file `users.txt`. Each line in that file contains a username and password, separated by a space. You should make a `users.txt` file for testing your program.

■■ **Business P11.27** In Exercise P11.26, the password is shown as it is typed. Browse the Swing documentation to find an appropriate component for entering a password. Improve the solution of Exercise P11.26 by using this component instead of a text field. Each time the user types a letter, show a ■ character.

ANSWERS TO SELF-CHECK QUESTIONS

1. Only the second one is displayed.
2. First add them to a panel, then add the panel to the north end of a frame.
3. Place them inside a panel with a `GridLayout` that has three rows and one column.
4. The button in the north stretches horizontally to fill the width of the frame. The height of the northern area is the normal height.
5. To get the double-wide button, put it in the south of a panel with border layout whose center has a 3 × 2 grid layout with the keys 7, 8, 4, 5, 1, 2. Put that panel in the west of another border layout panel whose eastern area has a 4 × 1 grid layout with the remaining keys.
6. 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.
7. If one of them is checked, the other one is unchecked. You should use radio buttons if that is the behavior you want.
8. You can't nest borders, but you can nest panels with borders:

```
JPanel p1 = new JPanel();
p1.setBorder(new EtchedBorder());
JPanel p2 = new JPanel();
p2.setBorder(new EtchedBorder());
p1.add(p2);
```

9. When any of the component settings is changed, the program simply queries all of them and updates the label.
10. To keep it from growing too large. It would have grown to the same width and height as the two panels below it.
11. Instead of using radio buttons with two choices, use a checkbox.
12. When you open a menu, you have not yet made a selection. Only `JMenuItem` objects correspond to selections.
13. Yes, you can—`JMenuItem` is a subclass of `JMenu`. The item shows up on the menu bar.When you click on it, its listener is called. But the behavior feels unnatural for a menu bar and is likely to confuse users.
14. The parameter variable is accessed in a method of an inner class.
15. Then the `faceName` variable is set when the menu item is added to the menu, not when the user selects the menu.
16. In the previous program, the user-interface components effectively served as storage for the font specification. Their current settings were used to construct the font. But a menu doesn't save settings; it just generates an action.
17. `JColorChooser`.
18. Action events describe one-time changes, such as button clicks. Change events describe continuous changes.
19. The timer needs to call some method whenever the time interval expires. It calls the `actionPerformed` method of the listener object.
20. Call `scene.moveRectangleBy(-1, -1)` in the `actionPerformed` method.
21. You can cut the timer delay in half (to 50 milliseconds between ticks), or you can double the distance by which the rectangle moves, by calling `scene.moveRectangleBy(2, 2)`.
22. The component class would need to draw a car at positon (x, y) instead of a rectangle.
23. There are two entirely different ways:
    a. Add a second `RectangleComponent` to the frame, using a grid layout. Change the `actionPerformed` method of the `TimerListener` to call `moveRectangleBy` on both components.
    b. Draw a second rectangle in the `paintComponent` method of `RectangleComponent`.
24. The moved rectangles won't be painted, and the rectangle will appear to be stationary until the frame is repainted for an external reason.
25. Because you know the current mouse position, not the amount by which the mouse has moved.
26. It implements the `MouseListener` interface, which has five methods.
27. The `RectangleComponent2` class needs to keep track of the locations of multiple rectangles. It can do that with an array list of `Point` or `Rectangle` objects. The `paintComponent` method needs to draw them all. Replace the `moveRectangleTo` method with an `addRectangleAt` method that adds a rectangle at a given (x, y) position.