```
125         group.add(blackItem);
126         group.add(redItem);
127         group.add(blueItem);
128
129         // Create a check box menu item to make the text
130         // visible or invisible.
131         visibleItem = new JCheckBoxMenuItem("Visible", true);
132         visibleItem.setMnemonic(KeyEvent.VK_V);
133         visibleItem.addActionListener(new VisibleListener());
134
135         // Create a JMenu object for the Text menu.
136         textMenu = new JMenu("Text");
137         textMenu.setMnemonic(KeyEvent.VK_T);
138
139         // Add the menu items to the Text menu.
140         textMenu.add(blackItem);
141         textMenu.add(redItem);
142         textMenu.add(blueItem);
143         textMenu.addSeparator();    // Add a separator bar.
144         textMenu.add(visibleItem);
145      }
146
147      /**
148         Private inner class that handles the event that
149         is generated when the user selects Exit from
150         the File menu.
151      */
152
153      private class ExitListener implements ActionListener
154      {
155         public void actionPerformed(ActionEvent e)
156         {
157            System.exit(0);
158         }
159      }
160
161      /**
162         Private inner class that handles the event that
163         is generated when the user selects a color from
164         the Text menu.
165      */
166
167      private class ColorListener implements ActionListener
168      {
169         public void actionPerformed(ActionEvent e)
170         {
171            if (blackItem.isSelected())
172               messageLabel.setForeground(Color.BLACK);
```

```
173          else if (redItem.isSelected())
174             messageLabel.setForeground(Color.RED);
175          else if (blueItem.isSelected())
176             messageLabel.setForeground(Color.BLUE);
177       }
178    }
179
180    /**
181       Private inner class that handles the event that
182       is generated when the user selects Visible from
183       the Text menu.
184    */
185
186    private class VisibleListener implements ActionListener
187    {
188       public void actionPerformed(ActionEvent e)
189       {
190          if (visibleItem.isSelected())
191             messageLabel.setVisible(true);
192          else
193             messageLabel.setVisible(false);
194       }
195    }
196
197    /**
198       The main method creates an instance of the
199       MenuWindow class, which causes it to display
200       its window.
201    */
202
203    public static void main(String[] args)
204    {
205       MenuWindow mw = new MenuWindow();
206    }
207 }
```

Let's take a closer look at the `MenuWindow` class. Before we examine how the menu system is constructed, we should explain the code in lines 38 through 44. Lines 38 through 40 create the `messageLabel` component and align its text in the label's center. Then, in lines 43 and 44, the `setPreferredSize` method is called. The `setPreferredSize` method is inherited from the `JComponent` class, and it establishes a component's preferred size. It is called the *preferred size* because the layout manager adjusts the component's size when necessary. Normally, a label's preferred size is determined automatically, depending on its contents. We want to make this label larger, however, so the window will be larger when it is packed around the label.

The `setPreferredSize` method accepts a `Dimension` object as its argument. A `Dimension` object specifies a component's width and height. The first argument to the `Dimension` class

constructor is the component's width, and the second argument is the component's height. In this class, the `LABEL_WIDTH` and `LABEL_HEIGHT` constants are defined with the values 400 and 200 respectively. So, this statement sets the label's preferred size to 400 pixels wide by 200 pixels high. (The `Dimension` class is part of the `java.awt` package.) Notice from Figure 13-24 that this code does not affect the size of the text that is displayed in the label, only the size of the label component.

To create the menu system, the constructor calls the `buildMenuBar` method in line 53. Inside this method, the statement in line 67 creates a `JMenuBar` component and assigns its address to the `menuBar` variable. The `JMenuBar` component acts as a container for `JMenu` components. The menu bar in this application has two menus: File and Text.

Next, the statement in line 70 calls the `buildFileMenu` method. The `buildFileMenu` method creates the File menu, which has only one item: Exit. The statement in line 89 creates a `JMenuItem` component for the Exit item, which is referenced by the `exitItem` variable. The `String` that is passed to the `JMenuItem` constructor is the text that will appear on a menu for this menu item. The statement in line 90 assigns the x key as a mnemonic to the `exitItem` component. Then, line 91 creates an action listener for the component (an instance of `ExitListener`, a private inner class), which causes the application to end.

Next, line 94 creates a `JMenu` object for the File menu. Notice that the name of the menu is passed as an argument to the `JMenu` constructor. Line 95 assigns the F key to the File menu as a mnemonic. The last statement in the `buildFileMenu` method, in line 98, adds `exitItem` to the `fileMenu` component.

Back in the `buildMenuBar` method, the statement in line 71 calls the `buildTextMenu` method. The `buildTextMenu` method builds the Text menu, which has three radio button menu items (Black, Red, and Blue), a separator bar, and a check box menu item (Visible). The code in lines 111 through 121 creates the radio button menu items, assigns mnemonic keys to them, and adds an action listener to each.

The `JRadioButtonItem` constructor accepts a `String` argument, which is the menu item's text. By default, a radio button menu item is not initially selected. The constructor can also accept an optional second argument, which is a `boolean` value indicating whether the item should be initially selected. Notice that in line 111, `true` is passed as the second argument to the `JRadioButtonItem` constructor. This causes the Black menu item to be initially selected.

Next, in lines 124 through 127, a button group is created and the radio button menu items are added to it. As with `JRadioButton` components, `JRadioButtonMenuItem` components may be grouped in a `ButtonGroup` object. As a result, only one of the grouped menu items may be selected at a time. When one is selected, any other menu item in the group is deselected.

Next, the Visible item, a check box menu item, is created in line 131. Notice that `true` is passed as the second argument to the constructor. This causes the item to be initially selected. A mnemonic key is assigned in line 132, and an action listener is added to the component in line 133.

Line 136 creates a `JMenu` component for the Text menu, and line 137 assigns a mnemonic key to it. Lines 140 through 142 add the `blackItem`, `redItem`, and `blueItem` radio button menu items to the Text menu. In line 143, the `addSeparator` method is called to add a

separator bar to the menu. Because the `addSeparator` method is called just after the `blueItem` component is added and just before the `visibleItem` component is added, it will appear between the Blue and Visible items on the menu. Line 144 adds the Visible item to the Text menu.

Back in the `buildMenuBar` method, in lines 74 and 75, the File menu and Text menu are added to the menu bar. In line 78, the `setJMenuBar` method is called, passing `menuBar` as an argument. The `setJMenuBar` method is a `JFrame` method that places a menu bar in a frame. You pass a `JMenuBar` component as the argument. When the `JFrame` is displayed, the menu bar will appear at its top.

Figure 13-26 shows how the class's window appears with the File menu and the Text menu opened. Selecting a color from the Text menu causes an instance of the `ColorListener` class to execute its `actionPerformed` method, which changes the color of the text. Selecting the Visible item causes an instance of the `VisibleListener` class to execute its `actionPerformed` method, which toggles the label's visibility.

Figure 13-26 The window with the File menu and Text menu opened

The window with the File menu opened.

The window with the Text menu opened.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

13.12 Briefly describe each of the following menu system items:
- a) Menu bar
- b) Menu item
- c) Check box menu item
- d) Radio button menu item
- e) Submenu
- f) Separator bar

13.13 What class do you use to create a regular menu item? What do you pass to the class constructor?

13.14 What class do you use to create a radio button menu item? What do you pass to the class constructor? How do you cause it to be initially selected?

13.15 How do you create a relationship between radio button menu items so that only one may be selected at a time?

13.16 What class do you use to create a check box menu item? What do you pass to the class constructor? How do you cause it to be initially selected?

13.17 What class do you use to create a menu? What do you pass to the class constructor?

13.18 What class do you use to create a menu bar?

13.19 How do you place a menu bar in a `JFrame`?

13.20 What type of event do menu items generate when selected by the user?

13.21 How do you change the size of a component such as a `JLabel` after it has been created?

13.22 What arguments do you pass to the `Dimension` class constructor?

13.9 More about Text Components: Text Areas and Fonts

CONCEPT: **A text area is a multi-line text field that can accept several lines of text input. Components that inherit from the `JComponent` class have a `setFont` method that allows you to change the font and style of the component's text.**

Text Areas

In Chapter 12 you were introduced to the `JTextField` class, which is used to create text fields. A text field is a component that allows the user to enter a single line of text. A text area is like a text field that can accept multiple lines of input. You use the `JTextArea` class to create a text area. Here is the general format of two of the class's constructors:

```
JTextArea(int rows, int columns)
JTextArea(String text, int rows, int columns)
```

In both constructors, `rows` is the number of rows or lines of text that the text area is to display, and `columns` is the number of columns or characters that are to be displayed per line. In the second constructor, `text` is a string that the text area will initially display. For example, the following statement creates a text area with 20 rows and 40 columns:

```
JTextArea textInput = new JTextArea(20, 40);
```

The following statement creates a text area with 20 rows and 40 columns that will initially display the text stored in the `String` object `info`:

```
JTextArea textInput = new JTextArea(info, 20, 40);
```

As with the `JTextField` class, the `JTextArea` class provides the `getText` and `setText` methods for getting and setting the text contained in the component. For example, the following statement gets the text stored in the `textInput` text area and stores it in the `String` object `userText`:

```
String userText = textInput.getText();
```

The following statement stores the text that is in the `String` object `info` in the `textInput` text area:

```
textInput.setText(info);
```

`JTextArea` components do not automatically display scroll bars. To display scroll bars on a `JTextArea` component, you must add it to the scroll pane. As you already know, you create a scroll pane with the `JScrollPane` class. Here is an example of code that creates a text area and adds it to a scroll pane:

```
JTextArea textInput = new JTextArea(20, 40);
JScrollPane scrollPane = new JScrollPane(textInput);
```

The `JScrollPane` object displays both vertical and horizontal scroll bars on a text area. By default, the scroll bars are not displayed until they are needed; however, you can alter this behavior with two of the `JScrollPane` class's methods. The `setHorizontalScrollBarPolicy` method takes an `int` argument that specifies when a horizontal scroll bar should appear in the scroll pane. You can pass one of the following constants as an argument:

- **`JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED`.** This is the default setting. A horizontal scroll bar is displayed only when there is not enough horizontal space to display the text contained in the text area.
- **`JScrollPane.HORIZONTAL_SCROLLBAR_NEVER`.** This setting prevents a horizontal scroll bar from being displayed in the text area.
- **`JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS`.** With this setting, a horizontal scroll bar is always displayed, even when it is not needed.

The `setVerticalScrollBarPolicy` method also takes an `int` argument, which specifies when a vertical scroll bar should appear in the scroll pane. You can pass one of the following constants as an argument:

- **`JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED`.** This is the default setting. A vertical scroll bar is displayed only when there is not enough vertical space to display the text contained in the text area.

- **`JScrollPane.VERTICAL_SCROLLBAR_NEVER`.** This setting prevents a vertical scroll bar from being displayed in the text area.
- **`JScrollPane.VERTICAL_SCROLLBAR_ALWAYS`.** With this setting, a vertical scroll bar is always displayed, even when it is not needed.

For example, the following code specifies that a vertical scroll bar should always appear on a scroll pane's component, but a horizontal scroll bar should not appear:

```
scrollPane.setHorizontalScrollBarPolicy(
                JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
scrollPane.setVerticalScrollBarPolicy(
                 JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
```

Figure 13-27 shows a text area without scroll bars, a text area with a vertical scroll bar, and a text area with both a horizontal and a vertical scroll bar.

Figure 13-27 Text areas with and without scroll bars

By default, `JTextArea` components do not perform line wrapping. This means that when text is entered into the component and the end of a line is reached, the text does not wrap around to the next line. If you want line wrapping, you use the `JTextArea` class's `setLineWrap` method to turn it on. The method accepts a `boolean` argument. If you pass `true`, line wrapping is turned on. If you pass `false`, line wrapping is turned off. Here is an example of a statement that turns a text area's line wrapping on:

```
textInput.setLineWrap(true);
```

There are two different styles of line wrapping: word wrapping and character wrapping. When word wrapping is performed, the line breaks always occur between words, never in the middle of a word. When character wrapping is performed, lines are broken between characters. This means that lines can be broken in the middle of a word. You specify the style of line wrapping that you prefer with the `JTextArea` class's `setWrapStyleWord` method. This method accepts a `boolean` argument. If you pass `true`, the text area will perform word wrapping. If you pass `false`, the text area will perform character wrapping. The default style is character wrapping.

Fonts

The appearance of a component's text is determined by the text's font, style, and size. The font is the name of the typeface—the style can be plain, bold, and/or italic—and the size is the size of the text in points. To change the appearance of a component's text you use the component's `setFont` method, which is inherited from the `JComponent` class. The general format of the method is as follows:

```
void setFont(Font appearance)
```

You pass a `Font` object as an argument to this method. The `Font` class constructor has the following general format:

```
Font(String fontName, int style, int size);
```

The first argument is the name of a font. Although the fonts that are available vary from system to system, Java guarantees that you will have Dialog, DialogInput, Monospaced, SansSerif, and Serif. Figure 13-28 shows an example of each of these.

Figure 13-28 Examples of fonts

The second argument to the `Font` constructor is an `int` that represents the style of the text. The `Font` class provides the following constants that you can use: `Font.PLAIN`, `Font.BOLD`, and `Font.ITALIC`. The third argument is the size of the text in points. (There are 72 points per inch, so a 72-point font has a height of one inch. Ten- and twelve-point fonts are normally used for most applications.) Here is an example of a statement that changes the text of a label to a 24-point bold serif font:

```
label.setFont(new Font("Serif", Font.BOLD, 24));
```

You can combine styles by mathematically adding them. For example, the following statement changes a label's text to a 24-point bold and italic serif font:

```
label.setFont(new Font("Serif", Font.BOLD + Font.ITALIC, 24));
```

Figure 13-29 shows an example of the serif font in plain, bold, italic, and bold plus italic styles. The following code was used to create the labels:

```
JLabel label1 = new JLabel("Serif Plain", SwingConstants.CENTER);
label1.setFont(new Font("Serif", Font.PLAIN, 24));

JLabel label2 = new JLabel("Serif Bold", SwingConstants.CENTER);
label2.setFont(new Font("Serif", Font.BOLD, 24));
```

```
JLabel label3 = new JLabel("Serif Italic", SwingConstants.CENTER);
label3.setFont(new Font("Serif", Font.ITALIC, 24));

JLabel label4 = new JLabel("Serif Bold + Italic",
                           SwingConstants.CENTER);
label4.setFont(new Font("Serif", Font.BOLD + Font.ITALIC, 24));
```

Figure 13-29 Examples of serif plain, bold, italic, and bold plus italic

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

13.23 What arguments do you pass to the `JTextArea` constructor?

13.24 How do you retrieve the text that is stored in a `JTextArea` component?

13.25 Does the `JTextArea` component automatically display scroll bars? If not, how do you accomplish this?

13.26 What is line wrapping? What are the two styles of line wrapping? How do you turn a `JTextArea` component's line wrapping on? How do you select a line wrapping style?

13.27 What type of argument does a component's `setFont` method accept?

13.28 What are the arguments that you pass to the `Font` class constructor?

See the Simple Text Editor Application case study on this book's companion Web site (www.pearsonhighered.com/gaddis) for an in-depth example that uses menus and other topics from this chapter.

13.10 Sliders

CONCEPT: A slider is a component that allows the user to adjust a number graphically within a range of values.

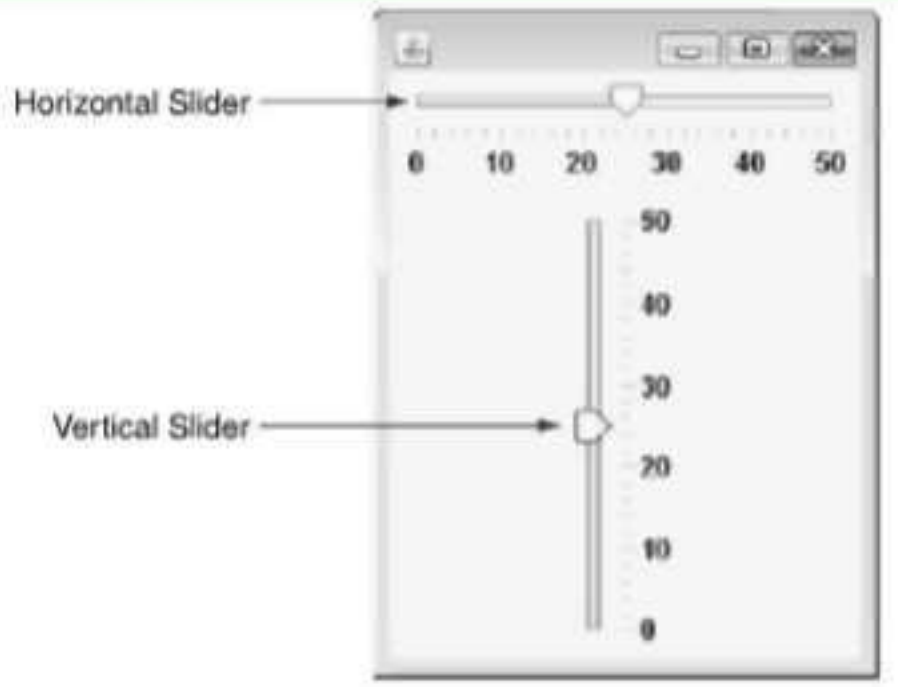
Sliders, which are created from the `JSlider` class, display an image of a "slider knob" that can be dragged along a track. Sliders can be horizontally or vertically oriented, as shown in Figure 13-30.

A slider is designed to represent a range of numeric values. At one end of the slider is the range's minimum value and at the other end is the range's maximum value. Both of the

sliders shown in Figure 13-30 represent a range of 0 through 50. Sliders hold a numeric value in a field, and as the user moves the knob along the track, the numeric value is adjusted accordingly. Notice that the sliders in Figure 13-30 have accompanying tick marks. At every tenth value, a major tick mark is displayed along with a label indicating the value at that tick mark. Between the major tick marks are minor tick marks, which in this example are displayed at every second value. The appearance of tick marks, their spacing, and the appearance of labels can be controlled through methods in the `JSlider` class. The `JSlider` constructor has the following general format:

```
JSlider(int orientation, int minValue,
        int maxValue, int initialValue)
```

Figure 13-30 A horizontal and a vertical slider



The first argument is an `int` specifying the slider's orientation. You should use one of the constants `JSlider.HORIZONTAL` or `JSlider.VERTICAL`. The second argument is the minimum value of the slider's range and the third argument is the maximum value of the slider's range. The fourth argument is the initial value of the slider, which determines the initial position of the slider's knob. For example, the following code could be used to create the sliders shown in Figure 13-30:

```
JSlider slider1 = new JSlider(JSlider.HORIZONTAL, 0, 50, 25);
JSlider slider2 = new JSlider(JSlider.VERTICAL, 0, 50, 25);
```

You set the major and minor tick mark spacing with the methods `setMajorTickSpacing` and `setMinorTickSpacing`. Each of these methods accepts an `int` argument that specifies the intervals of the tick marks. For example, the following code sets the `slider1` object's major tick mark spacing at 10, and its minor tick mark spacing at 2:

```
slider1.setMajorTickSpacing(10);
slider1.setMinorTickSpacing(2);
```

If the `slider1` component's range is 0 through 50, then these statements would cause major tick marks to be displayed at values 0, 10, 20, 30, 40, and 50. Minor tick marks would be displayed at values 2, 4, 6, and 8, then at values 12, 14, 16, and 18, and so forth.

By default, tick marks are not displayed, and setting their spacing does not cause them to be displayed. You display tick marks by calling the `setPaintTicks` method, which accepts a `boolean` argument. If you pass `true`, then tick marks are displayed. If you pass `false`, they are not displayed. Here is an example:

```
slider1.setPaintTicks(true);
```

By default, labels are not displayed either. You display numeric labels on the slider component by calling the `setPaintLabels` method, which accepts a `boolean` argument. If you pass `true`, then numeric labels are displayed at the major tick marks. If you pass `false`, labels are not displayed. Here is an example:

```
slider1.setPaintLabels(true);
```

When the knob's position is moved, the slider component generates a change event. To handle the change event, you must write a change listener class. When you write a change listener class, it must meet the following requirements:

- It must implement the `ChangeListener` interface. This interface is in the `javax.swing.event` package.
- It must have a method named `stateChanged`. This method must take an argument of the `ChangeEvent` type.

To retrieve the current value stored in a `JSlider`, use the `getValue` method. This method returns the slider's value as an `int`. Here is an example:

```
currentValue = slider1.getValue();
```

The `TempConverter` class shown in Code Listing 13-7 demonstrates the `JSlider` component. This class displays the window shown in Figure 13-31. Two temperatures are initially shown: 32.0 degrees Fahrenheit and 0.0 degrees Celsius. A slider, which has the range of 0 through 100, allows you to adjust the Celsius temperature and immediately see the Fahrenheit conversion. The `main` method creates an instance of the class and displays the window.

Figure 13-31 Window displayed by the `TempConverterWindow` class

Code Listing 13-7 (TempConverter.java)

```
1  import javax.swing.*;
2  import javax.swing.event.*;
3  import java.awt.*;
4  import java.text.DecimalFormat;
5
6  /**
7     This class displays a window with a slider component.
8     The user can convert the Celsius temperatures from
9     0 through 100 to Fahrenheit by moving the slider.
10 */
11
12 public class TempConverter extends JFrame
13 {
14    private JLabel label1, label2;          // Message labels
15    private JTextField fahrenheitTemp;      // Fahrenheit temp
16    private JTextField celsiusTemp;         // Celsius temp
17    private JPanel fpanel;                  // Fahrenheit panel
18    private JPanel cpanel;                  // Celsius panel
19    private JPanel sliderPanel;             // Slider panel
20    private JSlider slider;                 // Temperature adjuster
21
22    /**
23       Constructor
24    */
25
26    public TempConverter()
27    {
28       // Set the title.
29       setTitle("Temperatures");
30
31       // Specify an action for the close button.
32       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
33
34       // Create the message labels.
35       label1 = new JLabel("Fahrenheit: ");
36       label2 = new JLabel("Celsius: ");
37
38       // Create the read-only text fields.
39       fahrenheitTemp = new JTextField("32.0", 10);
40       fahrenheitTemp.setEditable(false);
41       celsiusTemp = new JTextField("0.0", 10);
42       celsiusTemp.setEditable(false);
43
44       // Create the slider.
45       slider = new JSlider(JSlider.HORIZONTAL, 0, 100, 0);
```

```
46          slider.setMajorTickSpacing(20);     // Major tick every 20
47          slider.setMinorTickSpacing(5);      // Minor tick every 5
48          slider.setPaintTicks(true);         // Display tick marks
49          slider.setPaintLabels(true);        // Display numbers
50          slider.addChangeListener(new SliderListener());
51
52          // Create panels and place the components in them.
53          fpanel = new JPanel();
54          fpanel.add(label1);
55          fpanel.add(fahrenheitTemp);
56          cpanel = new JPanel();
57          cpanel.add(label2);
58          cpanel.add(celsiusTemp);
59          sliderPanel = new JPanel();
60          sliderPanel.add(slider);
61
62          // Create a GridLayout manager.
63          setLayout(new GridLayout(3, 1));
64
65          // Add the panels to the content pane.
66          add(fpanel);
67          add(cpanel);
68          add(sliderPanel);
69
70          // Pack and display the frame.
71          pack();
72          setVisible(true);
73       }
74
75       /**
76          Private inner class to handle the change events
77          that are generated when the slider is moved.
78       */
79
80       private class SliderListener implements ChangeListener
81       {
82          public void stateChanged(ChangeEvent e)
83          {
84             double fahrenheit, celsius;
85             DecimalFormat fmt = new DecimalFormat("0.0");
86
87             // Get the slider value.
88             celsius = slider.getValue();
89
90             // Convert the value to Fahrenheit.
91             fahrenheit = (9.0 / 5.0) * celsius + 32.0;
92
93             // Store the Celsius temp in its display field.
```

```
94            celsiusTemp.setText(
95                        Double.toString(celsius));
96
97            // Store the Fahrenheit temp in its display field.
98            fahrenheitTemp.setText(fmt.format(fahrenheit));
99         }
100     }
101
102     /*
103        The main method creates an instance of the
104        class, which displays a window with a slider.
105     */
106
107     public static void main(String[] args)
108     {
109        new TempConverter();
110     }
111  }
```

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

13.29 What type of event does a `JSlider` generate when its slider knob is moved?

13.30 What `JSlider` methods do you use to perform each of these operations?

a) Establish the spacing of major tick marks.
b) Establish the spacing of minor tick marks.
c) Cause tick marks to be displayed.
d) Cause labels to be displayed.

13.11 Look and Feel

CONCEPT: A GUI application's appearance is determined by its look and feel. Java allows you to select an application's look and feel.

Most operating systems' GUIs have their own unique appearance and style conventions. For example, if a Windows user switches to a Macintosh, UNIX, or Linux system, the first thing he or she is likely to notice is the difference in the way the GUIs on each system appear. The appearance of a particular system's GUI is known as its look and feel.

Java allows you to select the look and feel of a GUI application. The default look and feel for Java is called *Ocean*. This is the look and feel that you have seen in all of the GUI applications that we have written in this book. Some of the other look and feel choices are Metal, Motif, and Windows. Metal was the default look and feel for previous versions of Java. Motif is similar to a UNIX look and feel. Windows is the look and feel of the Windows operating system. Figure 13-32 shows how the `TempConverterWindow` class window, presented earlier in this chapter, appears in each of these looks and feels.

NOTE: Ocean is actually a special theme of the Metal look and feel.

NOTE: Currently the Windows look and feel is available only on computers running the Microsoft Windows operating system.

Figure 13-32 Metal, Motif, and Windows looks and feels

To change an application's look and feel, you call the `UIManager` class's static `setLookAndFeel` method. Java has a class for each look and feel, and this method takes the fully qualified class name for the desired look and feel as its argument. The class name must be passed as a string. Table 13-1 lists the fully qualified class names for the Metal, Motif, and Windows looks and feels.

Table 13-1 Look and feel class names

Class Name	Look and Feel
`"javax.swing.plaf.metal.MetalLookAndFeel"`	Metal
`"com.sun.java.swing.plaf.motif.MotifLookAndFeel"`	Motif
`"com.sun.java.swing.plaf.windows.WindowsLookAndFeel"`	Windows

When you call the `UIManager.setLookAndFeel` method, any components that have already been created need to be updated. You do this by calling the `SwingUtilities.updateComponentTreeUI` method, passing a reference to the component that you want to update as an argument.

The `UIManager.setLookAndFeel` method throws a number of exceptions. Specifically, it throws `ClassNotFoundException`, `InstantiationException`, `IllegalAccessException`, and `UnsupportedLookAndFeelException`. Unless you want to trap each of these types of exceptions, you can simply trap exceptions of type `Exception`. Here is an example of code that can be run from a `JFrame` object that changes its look and feel to Motif:

```
try
{
    UIManager.setLookAndFeel(
         "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
    SwingUtilities.updateComponentTreeUI(this);
}
catch (Exception e)
{
    JOptionPane.showMessageDialog(null, "Error setting " +
                                        "the look and feel.");
    System.exit(0);
}
```

And here is an example of code that can be run from a `JFrame` object that changes its look and feel to Windows:

```
try
{
    UIManager.setLookAndFeel(
         "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
    SwingUtilities.updateComponentTreeUI(this);
}
catch (Exception e)
{
    JOptionPane.showMessageDialog(null, "Error setting " +
                                        "the look and feel.");
    System.exit(0);
}
```

13.12 Common Errors to Avoid

- **Only retrieving the first selected item from a list component in which multiple items have been selected.** If multiple items have been selected in a list component, the `getSelectedValue` method returns only the first selected item. Likewise, the `getSelectedIndex` method returns only the index of the first selected item. You should use the `getSelectedValues` or `getSelectedIndices` methods instead.
- **Using 1 as the beginning index for a list or combo box.** The indices for a list or combo box start at 0, not 1.
- **Forgetting to add a list or text area to a scroll pane.** The `JList` and `JTextArea` components do not automatically display scroll bars. You must add these components to a scroll pane object in order for them to display scroll bars.
- **Using the add method instead of the constructor to add a component to a scroll pane.** To add a component to a scroll pane, you must pass a reference to the component as an argument to the `JScrollPane` constructor.
- **Adding a component to a scroll pane and then adding the component (not the scroll pane) to another container, such as a panel.** If you add a component to a scroll pane and then intend to add that same component to a panel or other container, you must add the scroll pane instead of the component. Otherwise, the scroll bars will not appear on the component.

- **Forgetting to call the `setEditable` method to give a combo box a text field.** By default, a combo box is the combination of a button and a list. To make it a combination of a text field and a list, you must call the `setEditable` method and pass `true` as an argument.
- **Trying to open an image file of an unsupported type.** Currently, an `ImageIcon` object can open image files that are stored in JPEG, GIF, or PNG formats.
- **Loading an image into an existing `JLabel` component and clipping part of the image.** If you have not explicitly set the preferred size of a `JLabel` component, it resizes itself automatically when you load an image into it. The `JFrame` that encloses the `JLabel` does not automatically resize, however. You must call the `JFrame` object's `pack` method or `setPreferredSize` method to resize it.
- **Assigning the same mnemonic to more than one component.** If you assign the same mnemonic to more than one component in a window, it works only for the first component that you assigned it to.
- **Forgetting to add menu items to a `JMenu` component, and `JMenu` components to a `JMenuBar` component.** After you create a menu item, you must add it to a `JMenu` component in order for it to be displayed on the menu. Likewise, `JMenu` components must be added to a `JMenuBar` component in order to be displayed on the menu bar.
- **Not calling the `JFrame` object's `setJMenuBar` method to place the menu bar.** To display a menu bar, you must call the `setJMenuBar` method and pass it as an argument.
- **Not grouping `JRadioButtonMenuItems` in a `ButtonGroup` object.** Just like regular radio button components, you must group radio button menu items in a button group in order to create a mutually exclusive relationship among them.

Review Questions and Exercises

Multiple Choice and True/False

1. You can use this method to make a text field read-only.
   a. `setReadOnly`
   b. `setChangeable`
   c. `setUneditable`
   d. `setEditable`
2. A `JList` component generates this type of event when the user selects an item.
   a. action event
   b. item event
   c. list selection event
   d. list change event
3. To display a scroll bar with a `JList` component, you must _____.
   a. do nothing; the `JList` automatically appears with scroll bars if necessary
   b. add the `JList` component to a `JScrollPane` component
   c. call the `setScrollBar` method
   d. none of the above; you cannot display a scroll bar with a `JList` component

4. This is the `JList` component's default selection mode.
   a. single selection
   b. single interval selection
   c. multiple selection
   d. multiple interval selection
5. A list selection listener must have this method.
   a. `valueChanged`
   b. `selectionChanged`
   c. `actionPerformed`
   d. `itemSelected`
6. The `ListSelectionListener` interface is in this package.
   a. `java.awt`
   b. `java.awt.event`
   c. `javax.swing.event`
   d. `javax.event`
7. This `JList` method returns −1 if no item in the list is selected.
   a. `getSelectedValue`
   b. `getSelectedItem`
   c. `getSelectedIndex`
   d. `getSelection`
8. A `JComboBox` component generates this type of event when the user selects an item.
   a. action event
   b. item event
   c. list selection event
   d. list change event
9. You can pass an instance of this class to the `JLabel` constructor if you want to display an image in the label.
   a. `ImageFile`
   b. `ImageIcon`
   c. `JLabelImage`
   d. `JImageFile`
10. This method can be used to store an image in a `JLabel` or a `JButton` component.
    a. `setImage`
    b. `storeImage`
    c. `getIcon`
    d. `setIcon`
11. This is text that appears in a small box when the user holds the mouse cursor over a component.
    a. mnemonic
    b. instant message
    c. tool tip
    d. pop-up mnemonic

12. This is a key that activates a component just as if the user clicked it with the mouse.
    a. mnemonic
    b. key activator
    c. tool tip
    d. click simulator
13. To display an open file or save file dialog box, you use this class.
    a. `JFileChooser`
    b. `JOpenSaveDialog`
    c. `JFileDialog`
    d. `JFileOptionPane`
14. To display a dialog box that allows the user to select a color, you use this class.
    a. `JColor`
    b. `JColorDialog`
    c. `JColorChooser`
    d. `JColorOptionPane`
15. You use this class to create a menu bar.
    a. `MenuBar`
    b. `JMenuBar`
    c. `JMenu`
    d. `JBar`
16. You use this class to create a radio button menu item.
    a. `JMenuItem`
    b. `JRadioButton`
    c. `JRadioButtonItem`
    d. `JRadioButtonMenuItem`
17. You use this method to place a menu bar on a `JFrame`.
    a. `setJMenuBar`
    b. `setMenuBar`
    c. `placeMenuBar`
    d. `setJMenu`
18. The `setPreferredSize` method accepts this as its argument(s).
    a. a `Size` object
    b. two `int` values
    c. a `Dimension` object
    d. one `int` value
19. Components of this class are multi-line text fields.
    a. `JMultiLineTextField`
    b. `JTextArea`
    c. `JTextField`
    d. `JEditField`

20. This method is inherited from `JComponent` and changes the appearance of a component's text.
    a. `setAppearance`
    b. `setTextAppearance`
    c. `setFont`
    d. `setText`
21. This method sets the intervals at which major tick marks are displayed on a `JSlider` component.
    a. `setMajorTickSpacing`
    b. `setMajorTickIntervals`
    c. `setTickSpacing`
    d. `setIntervals`
22. **True or False:** You can use code to change the contents of a read-only text field.
23. **True or False:** A `JList` component automatically appears with a line border drawn around it.
24. **True or False:** In single interval selection mode, the user may select multiple items from a `JList` component.
25. **True or False:** With an editable combo box the user may only enter a value that appears in the component's list.
26. **True or False:** You can store either text or an image in a `JLabel` object, but not both.
27. **True or False:** You can store large images as well as small ones in a `JLabel` component.
28. **True or False:** Mnemonics are useful for users who are good with the keyboard.
29. **True or False:** A `JMenuBar` object acts as a container for `JMenu` components.
30. **True or False:** A `JMenu` object cannot contain other `JMenu` objects.
31. **True or False:** A `JTextArea` component does not automatically display scroll bars.
32. **True or False:** By default, a `JTextArea` component does not perform line wrapping.
33. **True or False:** A `JSlider` component generates an action event when the slider knob is moved.
34. **True or False:** By default, a `JSlider` component displays labels and tick marks.
35. **True or False:** When labels are displayed on a `JSlider` component, they are displayed on the major tick marks.

Find the Error

1.
```
// Create a read-only text field.
JTextField textField = new JTextField(10);
textField.setEditable(true);
```

2.
```
// Create a black 1-pixel border around list, a JList component.
list.setBorder(Color.BLACK, 1);
```

3.
```
// Create a JList and add it to a scroll pane.
// Assume that array already exists.
JList list = new JList(array);
JScrollPane scrollPane = new JScrollPane();
scrollPane.add(list);
```

4.
```
// Assume that nameBox is a combo box and is properly set up
// with a list of names to choose from.
// Get value of the selected item.
String selectedName = nameBox.getSelectedIndex();
```

5.
```
JLabel label = new JLabel("Have a nice day!");
label.setImage(image);
```

6.
```
// Add a menu to the menu bar.
JMenuBar menuBar = new JMenuBar(menuItem);
```

7.
```
// Create a text area with 20 columns and 5 rows.
JTextArea textArea = new JTextArea (20, 5);
```

Algorithm Workbench

1. Give an example of code that creates a read-only text field.
2. Write code that creates a list with the following items: Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, and Sunday.
3. Write code that adds a scroll bar to the list you created in your answer to Algorithm Workbench 2.
4. Assume that the variable `myList` references a `JList` component, and `selection` is a `String` variable. Write code that assigns the selected item in the `myList` component to the `selection` variable.
5. Assume that the variable `myComboBox` references an uneditable combo box, and `selectionIndex` is an `int` variable. Write code that assigns the index of the selected item in the `myComboBox` component to the `selectionIndex` variable.
6. Write code that stores the image in the file *dog.jpg* in a label.
7. Assume that `label` references an existing `JLabel` object. Write code that stores the image in the file *picture.gif* in the label.
8. Write code that creates a button with the text "Open File." Assign the O key as a mnemonic and assign "This button opens a file" as the component's tool tip.
9. Write code that displays a file open dialog box. If the user selects a file, the code should store the file's path and name in a `String` variable.
10. Write code that creates a text area displaying 10 rows and 15 columns. The text area should be capable of displaying scroll bars, when necessary. It should also perform word style line wrapping.
11. Write the code that creates a menu bar with one menu named File. The File menu should have the F key assigned as a mnemonic. The File menu should have three menu items: Open, Print, and Exit. Assign mnemonic keys of your choice to each of these items. Register an instance of the `OpenListener` class as an action listener for the Open menu item, an instance of the `PrintListener` class as an action listener for the Print menu item, and an instance of the `ExitListener` class as an action listener for the Exit menu item. Assume these classes have already been created.
12. Write code that creates a `JSlider` component. The component should be horizontally oriented and its range should be 0 through 1000. Labels and tick marks should be displayed. Major tick marks should appear at every 100th number, and minor tick marks should appear at every 25th number. The initial value of the slider should be set at 500.

Short Answer

1. What selection mode should you select if you want the user to select a single item only in a list?
2. You want to provide 20 items in a list for the user to select from. Which component would take up less space, a `JList` or a `JComboBox`?
3. What is the difference between an uneditable combo box and an editable combo box? Which one is a combo box by default?
4. Describe how you can store both an image and text in a `JLabel` component.
5. What is a mnemonic? How does the user use it?
6. What happens when the mnemonic that you assign to a component is a letter that appears in the component's text?
7. What is a tool tip? What is its purpose?
8. What do you do to a group of radio button menu items so that only one of them can be selected at a time?
9. When a checked menu item shows a check mark next to it, what happens when the user clicks on it?
10. What fonts does Java guarantee you have?
11. Why would a `JSlider` component be ideal when you want the user to enter a number, but you want to make sure that the number is within a range?
12. What are the standard GUI looks and feels that are available in Java?

Programming Challenges

MyProgrammingLab™ *Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

1. Scrollable Tax Calculator

Create an application that allows you to enter the amount of a purchase and then displays the amount of sales tax on that purchase. Use a slider to adjust the tax rate between 0 percent and 10 percent.

VideoNote
The Image Viewer Problem

2. Image Viewer

Write an application that allows the user to view image files. The application should use either a button or a menu item that displays a file chooser. When the user selects an image file, it should be loaded and displayed.

3. Dorm and Meal Plan Calculator

A university has the following dormitories:

Allen Hall: $1,500 per semester
Pike Hall: $1,600 per semester
Farthing Hall: $1,200 per semester
University Suites: $1,800 per semester

The university also offers the following meal plans:

7 meals per week: $560 per semester
14 meals per week: $1,095 per semester
Unlimited meals: $1,500 per semester

Create an application with two combo boxes. One should hold the names of the dormitories, and the other should hold the meal plans. The user should select a dormitory and a meal plan, and the application should show the total charges for the semester.

4. Skateboard Designer

The Skate Shop sells the skateboard products listed in Table 13-2.

Table 13-2 Skateboard products

Decks	Truck Assemblies	Wheels
The Master Thrasher $60	7.75 inch axle $35	51 mm $20
The Dictator $45	8 inch axle $40	55 mm $22
The Street King $50	8.5 inch axle $45	58 mm $24
		61 mm $28

In addition, the Skate Shop sells the following miscellaneous products and services:

Grip tape: $10
Bearings: $30
Riser pads: $2
Nuts & bolts kit: $3

Create an application that allows the user to select one deck, one truck assembly, and one wheel set from either list components or combo boxes. The application should also have a list component that allows the user to select multiple miscellaneous products. The application should display the subtotal, the amount of sales tax (at 6 percent), and the total of the order.

5. Shopping Cart System

Create an application that works like a shopping cart system for a bookstore. In this chapter's source code folder (available on the book's companion Web site at www.pearsonhighered.com/gaddis), you will find a file named *BookPrices.txt*. This file contains the names and prices of various books, formatted in the following fashion:

I Did It Your Way, 11.95
The History of Scotland, 14.50
Learn Calculus in One Day, 29.95
Feel the Stress, 18.50

Each line in the file contains the name of a book, followed by a comma, followed by the book's retail price. When your application begins execution, it should read the contents of the file and store the book titles in a list component. The user should be able to select a title from the list and add it to a shopping cart, which is simply another list component. The application should have buttons or menu items that allow the user to remove items from the shopping cart, clear the shopping cart of all selections, and check out. When the user

checks out, the application should calculate and display the subtotal of all the books in the shopping cart, the sales tax (which is 6 percent of the subtotal), and the total.

6. Cell Phone Packages

Cell Solutions, a cell phone provider, sells the following packages:

300 minutes per month: $45.00 per month
800 minutes per month: $65.00 per month
1500 minutes per month: $99.00 per month

The provider sells the following phones (a 6 percent sales tax applies to the sale of a phone):

Model 100: $29.95
Model 110: $49.95
Model 200: $99.95

Customers may also select the following options:

Voice mail: $5.00 per month
Text messaging: $10.00 per month

Write an application that displays a menu system. The menu system should allow the user to select one package, one phone, and any of the options desired. As the user selects items from the menu, the application should show the prices of the items selected.

7. Shade Designer

A custom window shade designer charges a base fee of $50 per shade. In addition, charges are added for certain styles, sizes, and colors as follows:

Styles:

Regular shades: Add $0
Folding shades: Add $10
Roman shades: Add $15

Sizes:

25 inches wide: Add $0
27 inches wide: Add $2
32 inches wide: Add $4
40 inches wide: Add $6

Colors:

Natural: Add $5
Blue: Add $0
Teal: Add $0
Red: Add $0
Green: Add $0

Create an application that allows the user to select the style, size, color, and number of shades from lists or combo boxes. The total charges should be displayed.

8. Conference Registration System

Create an application that calculates the registration fees for a conference. The general conference registration fee is $895 per person, and student registration is $495 per person. There is also an optional opening night dinner with a keynote speech for $30 per person. In addition, the optional preconference workshops listed in Table 13-3 are available.

Table 13-3 Optional preconference workshops

Workshop	Fee
Introduction to E-commerce	$295
The Future of the Web	$295
Advanced Java Programming	$395
Network Security	$395

The application should allow the user to select the registration type, the optional opening night dinner and keynote speech, and as many preconference workshops as desired. The total cost should be displayed.

9. Dice Simulator

Write a GUI application that simulates a pair of dice, similar to that shown in Figure 13-33. Each time the button is clicked, the application should roll the dice, using random numbers to determine the value of each die. (This chapter's source code folder contains images that you can use to display the dice.)

Figure 13-33 Dice simulator

10. Card Dealer

This chapter's source code folder contains images for a complete deck of poker cards. Write a GUI application, similar to the one shown in Figure 13-34, that randomly selects a card from the deck and displays it each time the user clicks the button. When a card has been selected, it is removed from the deck and cannot be selected again. Display a message when no more cards are left in the deck.

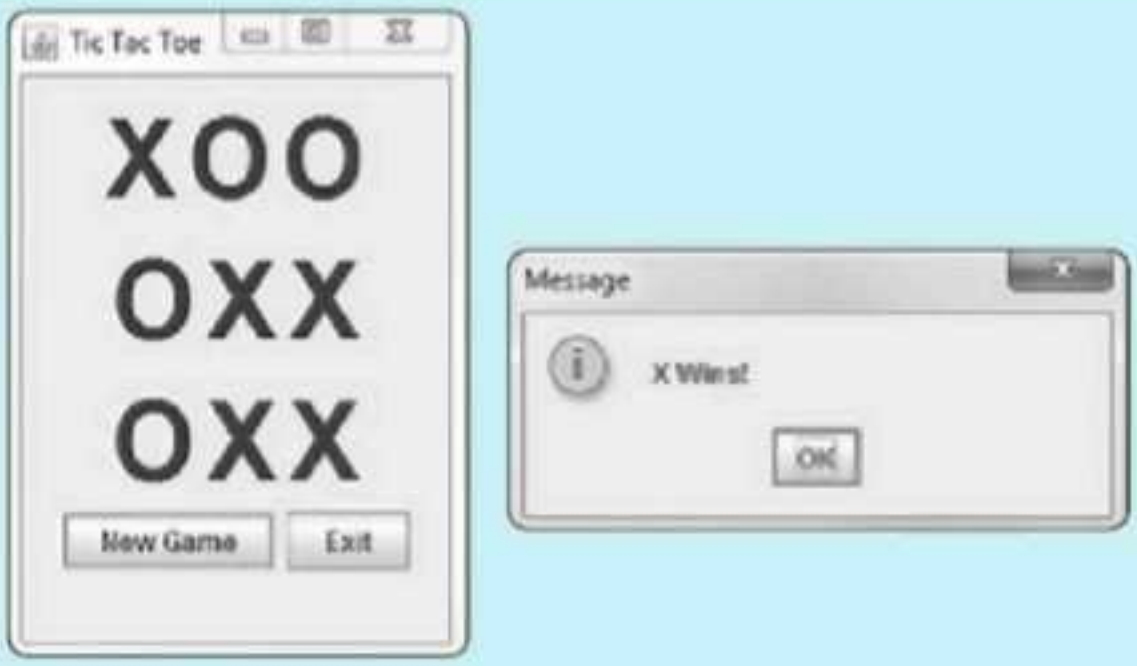
Figure 13-34 Card dealer

11. Tic Tac Toe Simulator

Create a GUI application that simulates a game of tic tac toe. Figure 13-35 shows an example of the application's window. The window shown in the figure uses nine large `JLabel` components to display the Xs and Os.

One approach in designing this application is to use a two-dimensional `int` array to simulate the game board in memory. When the user clicks the *New Game* button, the application should step through the array, storing a random number in the range of 0 through 1 in each element. The number 0 represents the letter O, and the number 1 represents the letter X. The `JLabel` components should then be updated to display the game board. The application should display a message indicating whether player X won, player Y won, or the game was a tie.

Figure 13-35 The Tic Tac Toe application



CHAPTER 14

Applets and More

TOPICS

14.1 Introduction to Applets

CONCEPT: An applet is a Java program that is associated with a Web page and is executed in a Web browser as part of that Web page.

Recall from Chapter 1 that there are two types of programs you can create with Java: applications and applets. An *application* is a stand-alone program that runs on your computer. So far in this book we have concentrated exclusively on writing applications.

Applets are Java programs that are usually part of a Web site. If a user opens the Web site with a Java-enabled browser, the applet is executed inside the browser window. It appears to the user that the applet is part of the Web site. This is how it works: Applets are stored on a Web server along with the site's Web pages. When a user accesses a Web page on a server with his or her browser, any applets associated with the Web page are transmitted over the Internet from the server to the user's system. This is illustrated in Figure 14-1. Once the applets are transmitted, the user's system executes them.

Applets are important because they can be used to extend the capabilities of a Web page. Web pages are normally written in Hypertext Markup Language (HTML). HTML is limited, however, because it merely describes the content and layout of a Web page, and creates links to other files and Web pages. HTML does not have sophisticated abilities such as performing math calculations and interacting with the user. A programmer can write a Java applet to perform these types of operations and associate it with a Web page. When someone visits the Web page, the applet is downloaded to the visitor's browser and executed.

Figure 14-1 Applets are transmitted along with Web pages

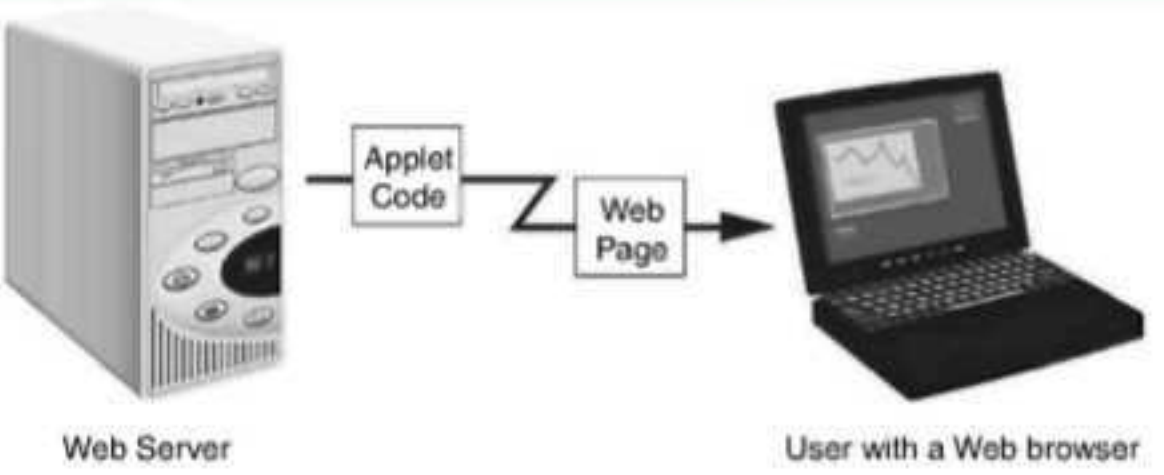


Figure 14-2 shows an example of a Web page that has an applet. In the figure, the Web page is being viewed with Internet Explorer. This Web page briefly explains the Fahrenheit and Celsius temperature scales. The area with the text boxes and the button at the bottom of the page is generated by an applet. To see a Fahrenheit temperature converted to Celsius, the user can enter the Fahrenheit temperature into the top text box and click the Convert button. The Celsius temperature will be displayed in the read-only text box.

An applet does not have to be on a Web server in order to be executed. The Web page shown in Figure 14-2 is in the source code folder *Chapter 14\TempConverter*. Open the *TempConverter.html* file in your Web browser to try it. Later in this chapter we will take a closer look at this Web page and its applet.

Figure 14-2 A Web page with an applet

Most Web browsers have a special version of the JVM for running applets. For security purposes, this version of the JVM greatly restricts what an applet can do. Here is a summary of the restrictions placed on applets:

- Applets cannot delete files, read the contents of files, or create files on the user's system.
- Applets cannot run any other program on the user's system.
- Applets cannot execute operating system procedures on the user's system.
- Applets cannot retrieve information about the user's system, or the user's identity.
- Applets cannot make network connections with any system except the server from which the applet was transmitted.
- If an applet displays a window, it will automatically have a message such as "Warning: Applet Window" displayed in it. This lets the user know that the window was not displayed by an application on his or her system.

These restrictions might seem severe, but they are necessary to prevent malicious code from attacking or spying on unsuspecting users. If an applet attempts to violate one of these restrictions, an exception is thrown.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

14.1 How is an applet that is associated with a Web page executed on a user's system?

14.2 Why do applets run in a restricted environment?

14.2 A Brief Introduction to HTML

CONCEPT: **When creating a Web page, you use Hypertext Markup Language (HTML) to create a file that can be read and processed by a Web browser.**

Hypertext Markup Language (HTML) is the language that Web pages are written in. Although it is beyond the scope of this book to teach you everything about HTML, this section will give you enough of the fundamentals so that you can write simple Web pages. You will need to know a little about HTML in order to run Java applets. If you are already familiar with HTML, this section is optional.

Before we continue, let's look at the meanings of the terms *hypertext* and *markup language*.

Hypertext

Web pages can contain regular text and hypertext, which are both displayed in the browser window. In addition, *hypertext* can contain a link to another Web page, or perhaps another location in the same Web page. When the user clicks on the hypertext, it loads the Web page or the location that the hypertext is linked to.

Markup Language

Although HTML is called a language, it is not a programming language like Java. Instead, HTML is a *markup language*. It allows you to "mark up" a text file by inserting special instructions. These instructions tell the browser how to format the text and create any hypertext links.

To make a Web page, you create a text file that contains HTML instructions, which are known as *tags*, as well as the text that should be displayed on the Web page. The resulting file is known as an *HTML document*, and it is usually saved with the *.html* file name extension. When a Web browser reads the HTML document, the tags instruct it how to format the text, where to place images, what to do when the user clicks on a link, and more.

Most HTML tags come in pairs. The first is known as the opening tag and the second is known as the closing tag. The general format of a simple tag is as follows:

```
<tag_name>
Text
</tag_name>
```

In this general format, `tag_name` is the name of the tag. The opening tag is `<tag_name>` and the closing tag is `</tag_name>`. Both the opening and closing tags are enclosed in angle brackets (< >). Notice that in the closing tag, the tag name is preceded by a forward slash (/). The `Text` that appears between the opening and closing tags is text that is formatted or modified by the tags.

Document Structure Tags

Some of the HTML tags are used to establish the structure of an HTML document. The first of the structure tags that you should learn is the `<html></html>` tag. This tag marks the beginning and ending of an HTML document. Everything that appears between these tags, including other tags, is the content of the Web page. When you are writing an HTML document, place an `<html>` tag at the very beginning, and an `</html>` tag at the very end.

The next tag is `<head></head>`. Everything that appears between `<head>` and `</head>` is considered part of the document head. The *document head* is a section of the HTML file that contains information about the document. For example, key words that search engines use to identify a document are often placed in the document's head. The only thing that we will use the document head for is to display a title in the Web browser's title bar. You do this with the `<title></title>` tag. Any text that you place between `<title>` and `</title>` becomes the title of the page and is displayed in the browser's title bar. Code Listing 14-1 shows the contents of an HTML document with the title "My First Web Page".

Notice that the `<title></title>` tag is inside of the `<head></head>` tag. The only output displayed by this Web page is the title. Figure 14-3 shows how this Web page appears when opened in a browser.

Code Listing 14-1 (BasicWebPage1.html)

```
<html>
<head>
    <title>My First Web Page</title>
</head>
</html>
```

Figure 14-3 Web page with a title only

After the document head comes the document body, which is enclosed in the <body></body> tag. The *document body* contains all of the tags and text that produce output in the browser window. Code Listing 14-2 shows an HTML document with text placed in its body. Figure 14-4 shows the document when opened in a browser.

Code Listing 14-2 (BasicWebPage2.html)

```
<html>
<head>
    <title>Java Applications and Applets</title>
</head>
<body>
    There are two types of programs you can create with Java: applications
    and applets. An application is a stand-alone program that runs on your
    computer. Applets are Java programs that are usually part of a Web site.
    They are stored on a Web server along with the site's Web pages. When a
    remote user accesses a Web page with his or her browser, any applets
```

```
    associated with the Web page are transmitted over the Internet from the
    server to the remote user's system.
</body>
</html>
```

Figure 14-4 Web page produced by *BasicWebPage2.html*

Text Formatting Tags

The text displayed in the Web page in Figure 14-4 is unformatted, which means it appears as plain text. There are many HTML tags that you can use to change the appearance of text. For example, there are six different header tags that you can use to format text as a heading of some type. The `<h1></h1>` tag creates a level one header. A level one header appears in boldface, and is much larger than regular text. The `<h2></h2>` tag creates a level two header. A level two header also appears in boldface, but is smaller than a level one header. This pattern continues with the `<h3></h3>`, `<h4></h4>`, `<h5></h5>`, and `<h6></h6>` tags. The higher a header tag's level number is, the smaller the text that it formats appears. For example, look at the following HTML:

```
<h1>This is an h1 Header</h1>
<h2>This is an h2 Header</h2>
<h3>This is an h3 Header</h3>
<h4>This is an h4 Header</h4>
<h5>This is an h5 Header</h5>
<h6>This is an h6 Header</h6>
This is regular unformatted text.
```

When this appears in the body of an HTML document, it produces the Web page shown in Figure 14-5.

You can use the `<center></center>` tag to center a line of text in the browser window. To demonstrate, we will add the following line to the document that was previously shown in Code Listing 14-2:

```
<center><h1>Java</h1></center>
```

Figure 14-5 Header levels

This will cause the word "Java" to appear centered and as a level one header. The modified document is shown in Code Listing 14-3, and the Web page it produces is shown in Figure 14-6.

Code Listing 14-3 (BasicWebPage3.html)

```
<html>
<head>
    <title>Java Applications and Applets</title>
</head>
<body>
    <center>
        <h1>Java</h1>
    </center>
    There are two types of programs you can create with Java: applications
    and applets. An application is a stand-alone program that runs
    on your computer. Applets are Java programs that are usually
    part of a Web site. They are stored on a Web server along with
    the site's Web pages. When a remote user accesses a Web page
    with his or her browser, any applets associated with the Web
    page are transmitted over the Internet from the server to the
    remote user's system.
</body>
</html>
```

Figure 14-6 Web page produced by *BasicWebPage3.html*

Notice that in the HTML document, the word "Java" is enclosed in two sets of tags: the `<center>` tags and the `<h1>` tags. It doesn't matter which set of tags is used first. If we had written the line as follows, we would have gotten the same result:

```
<h1><center>Java</center></h1>
```

You can display text in boldface by using the `<b></b>` tag, and in italics by using the `<i></i>` tag. For example, the following will cause the text "Hello World" to be displayed in boldface:

```
<b>Hello World</b>
```

The following will cause "Hello World" to be displayed in italics:

```
<i>Hello World</i>
```

The following will display "Hello World" in boldface and italics:

```
<b><i>Hello World</i></b>
```

Creating Breaks in Text

We will look at three HTML tags that are used to create breaks in a document's text. These three tags are unique from the ones we previously studied because they do not occur in pairs. When you use one of these tags, you only insert an opening tag.

The `<br />` tag causes a line break to appear at the point in the text where it is inserted. It is often necessary to insert `<br />` tags in an HTML document because the browser usually ignores the newline characters that are created when you press the Enter key. For example, if the following line appears in the body of an HTML document, it will cause the output shown in Figure 14-7.

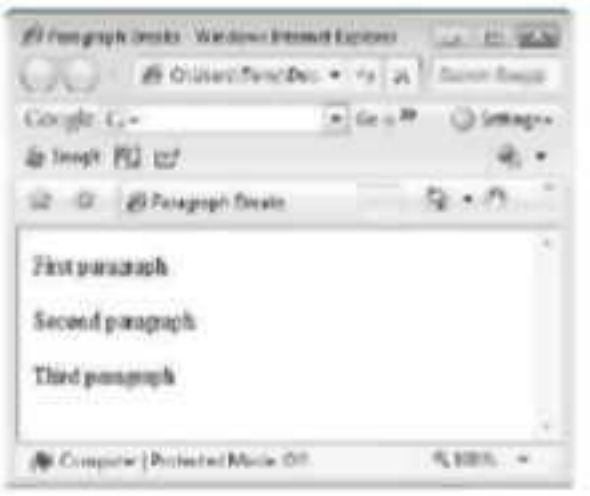
```
First line<br />Second line<br />Third line
```

Figure 14-7 Line breaks in an HTML document

The <p /> tag causes a paragraph break to appear at the point in the text where it is inserted. A paragraph break typically inserts more space into the text than a line break. For example, if the following line appears in the body of an HTML document, it will cause the output shown in Figure 14-8.

```
First paragraph<p />Second paragraph<p />Third paragraph
```

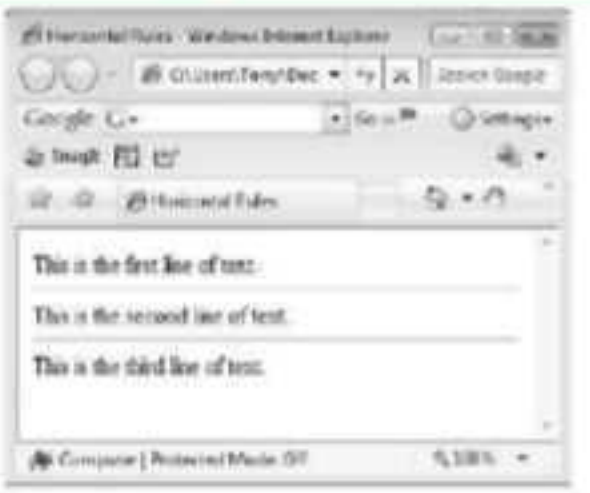
Figure 14-8 Paragraph breaks in an HTML document



The <hr /> tag causes a horizontal rule to appear at the point in the text where it is inserted. A horizontal rule is a thin, horizontal line that is drawn across the Web page. For example, if the following text appears in the body of an HTML document, it will cause the output shown in Figure 14-9.

```
This is the first line of text.
<hr />
This is the second line of text.
<hr />
This is the third line of text.
```

Figure 14-9 Horizontal rules in a Web page



The HTML document shown in Code Listing 14-4 demonstrates each of the tags we have discussed. The Web page it produces is shown in Figure 14-10.

Code Listing 14-4 (BasicWebPage4.html)

```
<html>
<head>
    <title>Java Applications and Applets</title>
</head>
<body>
    <center>
        <h1>Java</h1>
    </center>
    There are two types of programs you can create with Java: applications
    and applets.
    <p />
    <b>Applications</b>
    <br />
    An <i>application</i> is a stand-alone program that runs on
    your computer.
    <p />
    <b>Applets</b>
    <br />
    <i>Applets</i> are Java programs that are usually part of a
    Web site. They are stored on a Web server along with the site's
    Web pages. When a remote user accesses a Web page with his or
    her browser, any applets associated with the Web page are
    transmitted over the Internet from the server to the remote
    user's system.
    <hr />
</body>
</html>
```

Figure 14-10 Web page produced by *BasicWebPage4.html*

Inserting Links

As previously mentioned, a link is some element in a Web page that can be clicked on by the user. When the user clicks the link, another Web page is displayed, or some sort of action is initiated. We now look at how to insert a simple link that causes another Web page to be displayed. The tag that is used to insert a link has the following general format:

```
<a href="Address">Text</a>
```

The *Text* that appears between the opening and closing tags is the text that will be displayed in the Web page. When the user clicks on this text, the Web page that is located at *Address* will be displayed in the browser. This address is often referred to as a *uniform resource locator* (*URL*). Notice that the address is enclosed in quotation marks. Here is an example:

```
<a href="http://www.pearsonhighered.com/gaddis/">Click here to go to
the textbook's web site.</a>
```

The HTML document shown in Code Listing 14-5 uses this link, and Figure 14-11 shows how the page appears in the browser.

Code Listing 14-5 (LinkDemo.html)

```
<html>
<head>
    <title>Link Demonstration</title>
</head>
<body>
    This demonstrates a link.
    <br />
    <a href="http://www.aw.com/gaddis">Click here to go to
    the textbook's web site.</a>
</body>
</html>
```

The text that is displayed by a link is usually highlighted in some way to let the user know that it is not ordinary text. In Figure 14-11, the link text is underlined. When the user clicks on this text, the browser displays the Web page at `www.aw.com/gaddis`

Figure 14-11 Web page produced by *LinkDemo.html*

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

14.3 What tag marks the beginning and end of an HTML document?

14.4 What tag marks the beginning and end of an HTML document's head section?

14.5 What statement would you use in an HTML document to display the text "My Web Page" in the browser's title bar? What section of the HTML document would this statement be written in?

14.6 What tag marks the beginning and end of an HTML document's body section?

14.7 What statement would you write in an HTML document to display the text "Student Roster" as a level one header?

14.8 What statement would you write in an HTML document to display the text "My Resume" in bold and centered on the page?

14.9 What statement would you write in an HTML document to display the text "Hello World" in bold and italic?

14.10 What tag causes a line break? What tag causes a paragraph break? What tag displays a horizontal rule?

14.11 Suppose you wanted to display the text "Click Here" as a link to the Web site `http://java.sun.com`. What statement would you write to create the text?

14.3 Creating Applets with Swing

CONCEPT: **You extend a class from `JApplet` to create an applet, just as you extend a class from `JFrame` to create a GUI application.**

By now you know almost everything necessary to create an applet. That is because applets are very similar to GUI applications. You can think of an applet as a GUI application that runs under the control of a Web browser. Instead of displaying its own window, an applet

appears in the browser's window. The differences between GUI application code and applet code are summarized here:

- A GUI application class inherits from `JFrame`. An applet class inherits from `JApplet`. The `JApplet` class is part of the `javax.swing` package.
- A GUI application class has a constructor that creates other components and sets up the GUI. An applet class does not normally have a constructor. Instead, it has a method named `init` that performs the same operations as a constructor. The `init` method accepts no arguments and has a `void` return type.
- The following methods, which are commonly called in a GUI application's constructor, are not called in an applet:

```
setTitle
setSize
setDefaultCloseOperation
pack
setVisible
```

VideoNote
Creating an Applet

The methods listed here are used in a GUI application to affect the application's window in some way. They are not usually applicable to an applet because the applet does not have a window of its own.

- There is no static `main` method needed to create an instance of the applet class. The browser creates an instance of the class automatically.

Let's look at a simple applet. Code Listing 14-6 shows an applet that displays a label.

Code Listing 14-6 (SimpleApplet.java)

```
1 import javax.swing.*;
2 import java.awt.*;
3
4 /**
5    This is a simple applet.
6 */
7
8 public class SimpleApplet extends JApplet
9 {
10    /**
11       The init method sets up the applet, much
12       like a constructor.
13    */
14
15    public void init()
16    {
17       // Create a label.
18       JLabel label =
19            new JLabel("This is my very first applet.");
20
```

```
21        // Set the layout manager.
22        setLayout(new FlowLayout());
23
24        // Add the label to the content pane.
25        add(label);
26     }
27  }
```

This code is very much like a regular GUI application. Although this class extends `JApplet` instead of `JFrame`, you still add components to the content pane and use layout managers in the same way.

Running an Applet

The process of running an applet is different from that of running an application. To run an applet, you create an HTML document with an `applet` tag, which has the following general format:

```
<applet code="Filename.class" width=Wide height=High></applet>
```

In the general format, *`Filename.class`* is the name of the applet's *.class* file. This is the file that contains the compiled byte code. Note that you do not specify the *.java* file, which contains the Java source code. You can optionally specify a path along with the file name. If you specify only the file name, it is assumed that the file is in the same directory as the HTML document. *`Wide`* is the width of the applet in pixels, and *`High`* is the height of the applet in pixels. When a browser processes an `applet` tag, it loads specified byte code and executes it in an area that is the size specified by the *`Wide`* and *`High`* values.

The HTML document shown in Code Listing 14-7 uses an `applet` tag to load the applet shown in Code Listing 14-6. This document specifies that the applet should be displayed in an area that is 200 pixels wide by 50 pixels high. Figure 14-12 shows this document when it is displayed in a Web browser.

Code Listing 14-7 (SimpleApplet.html)

```
<html>
<head>
    <title>A Simple Applet</title>
</head>
<body>
    <applet code="SimpleApplet.class" width="200" height="50">
    </applet>
</body>
</html>
```

Figure 14-12 The Web page produced by *SimpleApplet.html*

NOTE: When you load a Web page that uses an applet into your browser, you will most likely get a security warning. For example, Figure 14-13 shows the warning you get from Internet Explorer. To run the applet, click the warning message and then select Allow Blocked Content . . . from the pop-up menu that appears.

Figure 14-13 Security warning in Internet Explorer

Running an Applet with `appletviewer`

The Sun JDK comes with an applet viewer program that loads and executes an applet without the need for a Web browser. This program can be run from a command prompt with the `appletviewer` command. When you run the program, you specify the name of an HTML document as a command line argument. For example, the following command passes `SimpleApplet.html` as the command line argument:

```
appletviewer SimpleApplet.html
```

This command executes any applet that is referenced by an `applet` tag in the file *SimpleApplet.html*. The window shown in Figure 14-14 will be displayed.

Figure 14-14 Applet executed by `appletviewer`

NOTE: The applet viewer does not display any output generated by text or tags in the HTML document. It only executes applets. If the applet viewer opens an HTML document with more than one `applet` tag, it will execute each applet in a separate window.

Handling Events in an Applet

In an applet, events are handled with event listeners exactly as they are in GUI applications. To demonstrate, we will examine the `TempConverter` class, which is shown in Code Listing 14-8. This class is the applet displayed in the Web page we examined at the beginning of this chapter. It has a text field where the user can enter a Fahrenheit temperature and a Convert button that converts the temperature to Celsius and displays it in a read-only text field. The temperature conversion is performed in an action listener class that handles the button's action events.

Code Listing 14-8 (`TempConverter.java`)

```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4  import java.text.DecimalFormat;
5
6  /**
7     The TempConverter class is an applet that converts
8     Fahrenheit temperatures to Celsius.
9  */
10
11 public class TempConverter extends JApplet
12 {
13    private JPanel fPanel;                 // To hold a text field
14    private JPanel cPanel;                 // To hold a text field
15    private JPanel buttonPanel;            // To hold a button
16    private JTextField fahrenheit;         // Fahrenheit temperature
17    private JTextField celsius;            // Celsius temperature
18
19    /**
20       init method
21    */
22
23    public void init()
24    {
25       // Build the panels.
26       buildFpanel();
```

```
27          buildCpanel();
28          buildButtonPanel();
29
30          // Create a layout manager.
31          setLayout(new GridLayout(3, 1));
32
33          // Add the panels to the content pane.
34          add(fPanel);
35          add(cPanel);
36          add(buttonPanel);
37       }
38
39       /**
40          The buildFpanel method creates a panel with a text
41          field in which the user can enter a Fahrenheit
42          temperature.
43       */
44
45       private void buildFpanel()
46       {
47          // Create the panel.
48          fPanel = new JPanel();
49
50          // Create a label to display a message.
51          JLabel message1 =
52                    new JLabel("Fahrenheit Temperature: ");
53
54          // Create a text field for the Fahrenheit temp.
55          fahrenheit = new JTextField(10);
56
57          // Create a layout manager for the panel.
58          fPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));
59
60          // Add the label and text field to the panel.
61          fPanel.add(message1);
62          fPanel.add(fahrenheit);
63       }
64
65       /**
66          The buildCpanel method creates a panel that
67          displays the Celsius temperature in a
68          read-only text field.
69       */
70
71       private void buildCpanel()
72       {
73          // Create the panel.
74          cPanel = new JPanel();
75
```

```
76          // Create a label to display a message.
77          JLabel message2 =
78                  new JLabel("Celsius Temperature: ");
79
80          // Create a text field for the Celsius temp.
81          celsius = new JTextField(10);
82
83          // Make the text field read-only.
84          celsius.setEditable(false);
85
86          // Create a layout manager for the panel.
87          cPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));
88
89          // Add the label and text field to the panel.
90          cPanel.add(message2);
91          cPanel.add(celsius);
92       }
93
94       /**
95          The buildButtonPanel method creates a panel with
96          a button that converts the Fahrenheit temperature
97          to Celsius.
98       */
99
100      private void buildButtonPanel()
101      {
102         // Create the panel.
103         buttonPanel = new JPanel();
104
105         // Create a button with the text "Convert".
106         JButton convButton = new JButton("Convert");
107
108         // Add an action listener to the button.
109         convButton.addActionListener(new ButtonListener());
110
111         // Add the button to the panel.
112         buttonPanel.add(convButton);
113      }
114
115      /**
116         Private inner class that handles the action event
117         that is generated when the user clicks the convert
118         button.
119      */
120
121      private class ButtonListener implements ActionListener
122      {
123         public void actionPerformed(ActionEvent e)
```

```
124       {
125          double ftemp, ctemp;  // To hold the temperatures
126
127          // Create a DecimalFormat object to format numbers.
128          DecimalFormat formatter = new DecimalFormat("0.0");
129
130          // Get the Fahrenheit temperature and convert it
131          // to a double.
132          ftemp = Double.parseDouble(fahrenheit.getText());
133
134          // Calculate the Celsius temperature.
135          ctemp = (5.0 / 9.0) * (ftemp - 32);
136
137          // Display the Celsius temperature.
138          celsius.setText(formatter.format(ctemp));
139       }
140    }
141 }
```

Code Listing 14-9 shows the contents of `TempConverter.html`, an HTML document that uses this applet. Figure 14-15 shows the Web page produced by this document. In the figure, the user has entered a Fahrenheit temperature and converted it to Celsius.

Code Listing 14-9 (TempConverter.html)

```
<html>
<head>
    <title>Fahrenheit and Celsius Temperatures</title>
</head>
<body>
    <center>
        <h1>Fahrenheit and Celsius Temperatures</h1>
    </center>
    Fahrenheit and Celsius are two temperature scales in use today.
    The <b>Fahrenheit</b> scale was developed by the German physicist
    Daniel Gabriel Fahrenheit (1686 - 1736). In the Fahrenheit scale,
    water freezes at 32 degrees and boils at 212 degrees. The
    <b>Celsius</b> scale was developed by Swedish astronomer Andres Celsius
    (1701 - 1744). In the Celsius scale, water freezes at 0 degrees and
    boils at 100 degrees. The Celsius to Fahrenheit conversion formula
    is:
    <p />
    <i>C</i> = (5/9) * (<i>F</i> - 32)
    <p />
    where <i>F</i> is the Fahrenheit temperature. You can also use
    this Web page to convert Fahrenheit temperatures to Celsius.
    Just enter a Fahrenheit temperature in the text box below, then
```

```
        click on the Convert button.
        <p />
        <applet code="TempConverter.class" width="300" height="150">
        </applet>
        <hr />
    </body>
    </html>
```

Figure 14-15 Web page produced by *TempConverter.html*

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

14.12 Instead of `JFrame`, an applet class is extended from what class?

14.13 Instead of a constructor, an applet class uses what method?

14.14 Why is there no need for a static `main` method to create an instance of an applet class?

14.15 Suppose the file *MyApplet.java* contains the Java source code for an applet. What tag would you write in an HTML document to run the applet in an area that is 400 pixels wide by 200 pixels high?

14.4 Using AWT for Portability

CONCEPT: Applets that use Swing components may be incompatible with some browsers. If you want to make sure that an applet is compatible with all Java-enabled browsers, use AWT components instead of Swing.

Java provides two libraries of classes that GUI components may be created from. Recall from Chapter 12 that these libraries are AWT and Swing. AWT is the original library that has been part of Java since its earliest version. Swing is an improved library that was introduced with Java 2. All of the GUI applications in Chapters 12 and 13, as well as the applets we have studied so far in this chapter, use Swing classes for their components.

Some browsers, do not directly support the Swing classes in applets. These browsers require a *plug-in*, which is software that extends or enhances another program, in order to run applets that use Swing components. Fortunately, this plug-in is automatically installed on a computer when the Sun JDK is installed. If you have installed the JDK, you should be able to write applets that use Swing and run them with no problems.

If you are writing an applet for other people to run on their computers, however, there is no guarantee that they will have the required plug-in. If this is the case, you should use the AWT classes instead of the Swing classes for the components in your applet. Fortunately, the AWT component classes are very similar to the Swing classes, so learning to use them is simple if you already know how to use Swing.

There is a corresponding AWT class for each of the Swing classes that you have learned so far. The names of the AWT classes are the same as those of the Swing classes, except the AWT class names do not start with the letter `J`. For example, the AWT class to create a frame is named `Frame`, and the AWT class to create a panel is named `Panel`. Table 14-1 lists several of the AWT classes. All of these classes are in the `java.awt` package.

Table 14-1 Several AWT classes

AWT Class	Description	Corresponding Swing Class
`Applet`	Used as a superclass for all applets. Unlike `JApplet` objects, `Applet` objects do not have a content pane.	`JApplet`
`Frame`	Creates a frame container that may be displayed as a window. Unlike `JFrame` objects, `Frame` objects do not have a content pane.	`JFrame`
`Panel`	Creates a panel container.	`JPanel`
`Button`	Creates a button that may be clicked.	`JButton`
`Label`	Creates a label that displays text.	`JLabel`
`TextField`	Creates a single line text field, which the user may type into.	`JTextField`
`Checkbox`	Creates a check box that may be selected or deselected.	`JCheckBox`

The Swing classes were intentionally designed with constructors and methods that are similar to those of their AWT counterparts. In addition, events are handled in the same way for each set of classes. This makes it easy for you to use either set of classes without learning a completely different syntax for each. For example, Code Listing 14-10 shows a version of the `TempConverter` applet that has been rewritten to use AWT components instead of Swing components.

Code Listing 14-10 (AWTTempConverter.java)

```
1  import java.applet.Applet;
2  import java.awt.*;
3  import java.awt.event.*;
4  import java.text.DecimalFormat;
5
6  /**
7     The AWTTempConverter class is an applet that converts
8     Fahrenheit temperatures to Celsius.
9  */
10
11 public class AWTTempConverter extends Applet
12 {
13    private Panel fPanel;          // To hold a text field
14    private Panel cPanel;          // To hold a text field
15    private Panel buttonPanel;     // To hold a button
16    private TextField fahrenheit;  // Fahrenheit temperature
17    private TextField celsius;     // Celsius temperature
18
19    /**
20       init method
21    */
22
23    public void init()
24    {
25       // Build the panels.
26       buildFpanel();
27       buildCpanel();
28       buildButtonPanel();
29
30       // Create a layout manager.
31       setLayout(new GridLayout(3, 1));
32
33       // Add the panels to the applet.
34       add(fPanel);
35       add(cPanel);
36       add(buttonPanel);
37    }
38
```

```
39    /**
40       The buildFpanel method creates a panel with a text
41       field in which the user can enter a Fahrenheit
42       temperature.
43    */
44
45    private void buildFpanel()
46    {
47       // Create the panel.
48       fPanel = new Panel();
49
50       // Create a label to display a message.
51       Label message1 =
52                 new Label("Fahrenheit Temperature: ");
53
54       // Create a text field for the Fahrenheit temp.
55       fahrenheit = new TextField(10);
56
57       // Create a layout manager for the panel.
58       fPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));
59
60       // Add the label and text field to the panel.
61       fPanel.add(message1);
62       fPanel.add(fahrenheit);
63    }
64
65    /**
66       The buildCpanel method creates a panel that
67       displays the Celsius temperature in a
68       read-only text field.
69    */
70
71    private void buildCpanel()
72    {
73       // Create the panel.
74       cPanel = new Panel();
75
76       // Create a label to display a message.
77       Label message2 =
78                new Label("Celsius Temperature: ");
79
80       // Create a text field for the Celsius temp.
81       celsius = new TextField(10);
82
83       // Make the text field read-only.
84       celsius.setEditable(false);
85
86       // Create a layout manager for the panel.
```

```
87          cPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));
88
89          // Add the label and text field to the panel.
90          cPanel.add(message2);
91          cPanel.add(celsius);
92       }
93
94       /**
95          The buildButtonPanel method creates a panel with
96          a button that converts the Fahrenheit temperature
97          to Celsius.
98       */
99
100
101      private void buildButtonPanel()
102      {
103         // Create the panel.
104         buttonPanel = new Panel();
105
106         // Create a button with the text "Convert".
107         Button convButton = new Button("Convert");
108
109         // Add an action listener to the button.
110         convButton.addActionListener(new ButtonListener());
111
112         // Add the button to the panel.
113         buttonPanel.add(convButton);
114      }
115
116      /**
117         Private inner class that handles the action event
118         that is generated when the user clicks the convert
119         button.
120      */
121
122      private class ButtonListener implements ActionListener
123      {
124         public void actionPerformed(ActionEvent e)
125         {
126            double ftemp, ctemp;  // To hold the temperatures
127
128            // Create a DecimalFormat object to format numbers.
129            DecimalFormat formatter = new DecimalFormat("0.0");
130
131            // Get the Fahrenheit temperature and convert it
132            // to a double.
133            ftemp = Double.parseDouble(fahrenheit.getText());
134
```

```
135          // Calculate the Celsius temperature.
136          ctemp = (5.0 / 9.0) * (ftemp - 32);
137
138          // Display the Celsius temperature.
139          celsius.setText(formatter.format(ctemp));
140       }
141    }
142 }
```

The only modifications that were made were as follows:

- The `JApplet`, `JPanel`, `JLabel`, `JTextField`, and `JButton` classes were replaced with the `Applet`, `Panel`, `Label`, `TextField`, and `Button` classes.
- The `import javax.swing.*;` statement was removed.

To run the applet in a browser, the `APPLET` tag in the *TempConverter.html* file must be modified to read as follows:

```
<applet code="AWTTempConverter.class" width=300 height=150>
</applet>
```

Once this change is made, the *TempConverter.html* file produces the Web page shown in Figure 14-16.

Figure 14-16 Web page running the `AWTTempConverter` applet

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

14.16 To create an applet using AWT, what class do you inherit your applet class from?

14.17 In Swing, if an object's class extends `JFrame` or `JApplet`, you add components to its content pane. How do you add components to an object if its class extends `Frame` or `Applet`?

14.5 Drawing Shapes

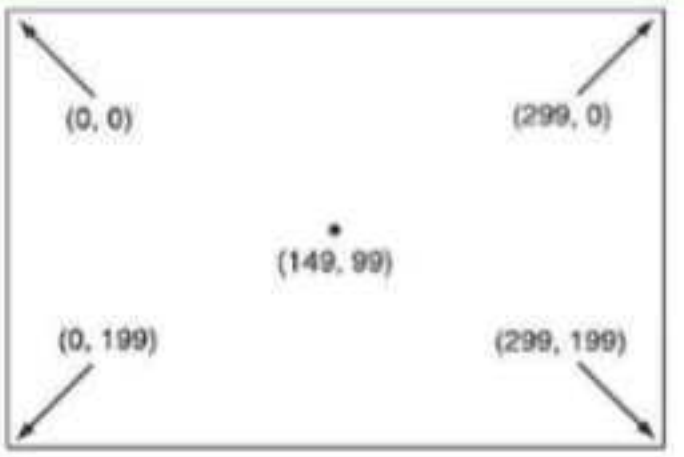
CONCEPT: **Components have an associated `Graphics` object that may be used to draw lines and shapes.**

In addition to displaying standard components such as buttons and labels, Java allows you to draw lines and graphical shapes such as rectangles, ovals, and arcs. These lines and shapes are drawn directly on components. This allows a frame or a panel to become a canvas for your drawings. Before we examine how to draw graphics on a component, however, we must discuss the *XY* coordinate system. You use the *XY* coordinate system to specify the location of your graphics.

The *XY* Coordinate System

The location of each pixel in a component is identified with an *X* coordinate and a *Y* coordinate. The coordinates are usually written in the form (*X*, *Y*). The *X* coordinate identifies a pixel's horizontal location, and the *Y* coordinate identifies its vertical location. The coordinates of the pixel in the upper-left corner of a component are usually (0, 0). The *X* coordinates increase from left to right, and the *Y* coordinates increase from top to bottom. For example, Figure 14-17 illustrates a component such as a frame or a panel that is 300 pixels wide by 200 pixels high. The *X* and *Y* coordinates of the pixels in each corner, as well as the pixel in the center of the component are shown. The pixel in the center of the component has an *X* coordinate of 149 and a *Y* component of 99.

Figure 14-17 *X* and *Y* coordinates on a 300 pixel wide by 200 pixel high component



When you draw a line or shape on a component, you must indicate its position using *X* and *Y* coordinates.

Graphics Objects

Each component has an internal object that inherits from the `Graphics` class, which is part of the `java.awt` package. This object has numerous methods for drawing graphical shapes on the surface of the component. Table 14-2 lists some of these methods.

Table 14-2 Some of the `Graphics` class methods

Method	Description
`void setColor(Color c)`	Sets the drawing color for this object to that specified by the argument.
`Color getColor()`	Returns the current drawing color for this object.
`void drawLine(int x1, int y1, int x2, int y2)`	Draws a line on the component starting at the coordinate (*x1*, *y1*) and ending at the coordinate (*x2*, *y2*). The line will be drawn in the current drawing color.
`void drawRect(int x, int y, int width, int height)`	Draws the outline of a rectangle on the component. The upper-left corner of the rectangle will be at the coordinate (*x*, *y*). The *width* parameter specifies the rectangle's width in pixels, and *height* specifies the rectangle's height in pixels. The rectangle will be drawn in the current drawing color.
`void fillRect(int x, int y, int width, int height)`	Draws a filled rectangle. The parameters are the same as those used by the `drawRect` method. The rectangle will be filled with the current drawing color.
`void drawOval(int x, int y, int width, int height)`	Draws the outline of an oval on the component. The shape and size of the oval is determined by an invisible rectangle that encloses it. The upper-left corner of the rectangle will be at the coordinate (*x*, *y*). The *width* parameter specifies the rectangle's width in pixels, and *height* specifies the rectangle's height in pixels. The oval will be drawn in the current drawing color.
`void fillOval(int x, int y, int width, int height)`	Draws a filled oval. The parameters are the same as those used by the `drawOval` method. The oval will be filled in the current drawing color.
`void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)`	This method draws an arc, which is considered to be part of an oval. The shape and size of the oval are determined by an invisible rectangle that encloses it. The upper-left corner of the rectangle will be at the coordinate (*x*, *y*). The *width* parameter specifies the rectangle's width in pixels, and *height* specifies the rectangle's height in pixels. The arc begins at the angle *startAngle*, and ends at the angle *arcAngle*. The arc will be drawn in the current drawing color.
`void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)`	This method draws a filled arc. The parameters are the same as those used by the `drawArc` method. The arc will be filled with the current drawing color.

(table continues next page)

Table 14-2 Some of the `Graphics` class methods (continued)

Method	Description
`void drawPolygon(int[] xPoints, int[] yPoints, int numPoints)`	This method draws the outline of a closed polygon on the component. The *`xPoints`* array contains the *X*-coordinates for each vertex, and the *`yPoints`* array contains the *Y* coordinates for each vertex. The argument passed into *`numPoints`* is the number of vertices in the polygon.
`void fillPolygon(int[] xPoints, int[] yPoints, int numPoints)`	This method draws a filled polygon. The parameters are the same as those used by the `drawPolygon` method. The polygon will be filled with the current drawing color.
`void drawString(String str, int x, int y)`	Draws the string passed into *`str`* using the current font. The bottom left of the string is drawn at the coordinates passed into *x* and *y*.
`void setFont(Font f)`	Sets the current font, which is used by the `drawString` method.

In order to call any of these methods, you must get a reference to a component's `Graphics` object. One way to do this is to override the `paint` method. You can override the `paint` method in any class that extends as follows:

- `JApplet`
- `JFrame`
- Any AWT class, including `Applet` and `Frame`

The `paint` method is responsible for displaying, or "painting," a component on the screen. This method is automatically called when the component is first displayed and is called again any time the component needs to be redisplayed. For example, when the component is completely or partially obscured by another window, and the obscuring window is moved, then the component's `paint` method is called to redisplay it. The header for the `paint` method is:

```
public void paint(Graphics g)
```

Notice that the method's argument is a `Graphics` object. When this method is called for a particular component, the `Graphics` object that belongs to that component is automatically passed as an argument. By overriding the `paint` method, you can use the `Graphics` object argument to draw your own graphics on the component. For example, look at the applet class in Code Listing 14-11.

This class inherits from `JApplet`, and it overrides the `paint` method. The `Graphics` object that is passed into the `paint` method's `g` parameter is the object that is responsible for drawing the entire applet window. Notice that in line 29 the method first calls the superclass version of the `paint` method, passing the object `g` as an argument. When overriding the `paint` method, you should always call the superclass's `paint` method before doing anything else. This ensures that the component will be displayed properly on the screen.

Code Listing 14-11 (LineDemo.java)

```
1 import javax.swing.*;
2 import java.awt.*;
3
4 /**
5    This class is an applet that demonstrates how lines
6    can be drawn.
7 */
8
9 public class LineDemo extends JApplet
10 {
11    /**
12       init method
13    */
14
15    public void init()
16    {
17       // Set the background color to white.
18       getContentPane().setBackground(Color.white);
19    }
20
21    /**
22       paint method
23       @param g The applet's Graphics object.
24    */
25
26    public void paint(Graphics g)
27    {
28       // Call the superclass paint method.
29       super.paint(g);
30
31       // Draw a red line from (20, 20) to (280, 280).
32       g.setColor(Color.red);
33       g.drawLine(20, 20, 280, 280);
34
35       // Draw a blue line from (280, 20) to (20, 280).
36       g.setColor(Color.blue);
37       g.drawLine(280, 20, 20, 280);
38    }
39 }
```

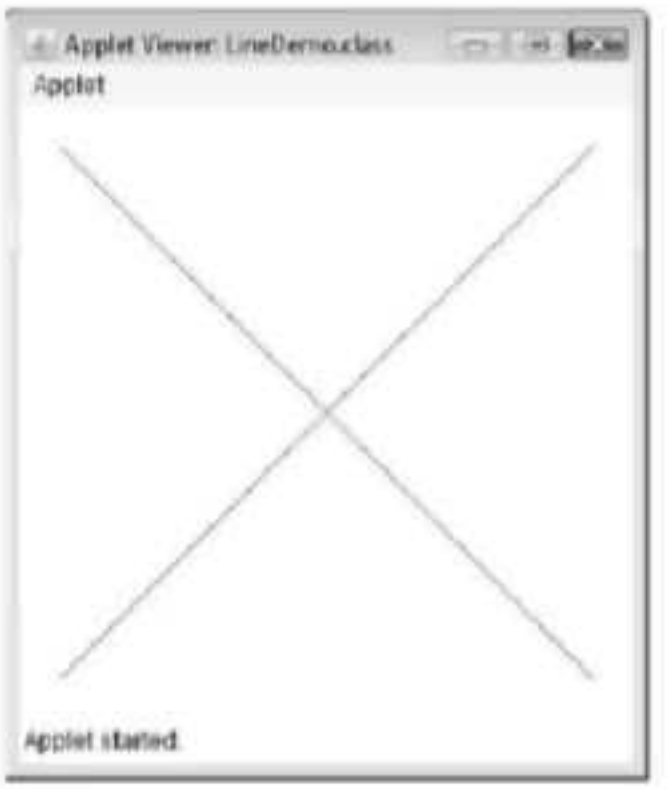
In line 32 the method sets the drawing color to red. In line 33 a line is drawn from the coordinates (20, 20) to (280, 280). This is a diagonal line drawn from the top-left area of the applet window to the bottom-right area. Next, in line 36, the drawing color is set to blue. In line 37 a line is drawn from (280, 20) to (20, 280). This is also a diagonal line. It is drawn from the top-right area of the applet window to the bottom-left area.

We can use the *LineDemo.html* file, which is in the same folder as the applet class, to execute the applet. The following line in the file runs the applet in an area that is 300 pixels wide by 300 pixels high:

```
<applet code="LineDemo.class" width=300 height=300>
</applet>
```

Figure 14-18 shows the applet running in the applet viewer.

Figure 14-18 `LineDemo` applet



Notice that the `paint` method is not explicitly called by the applet. It is automatically called when the applet first executes. As previously mentioned, it is also called any time the applet window needs to be redisplayed.

Code Listing 14-12 shows the `RectangleDemo` class, an applet that draws two rectangles: one as a black outline and one filled with red. Each rectangle is 120 pixels wide and 120 pixels high. The file *RectangleDemo.html*, which is in the same folder as the applet class, executes the applet with the following tag:

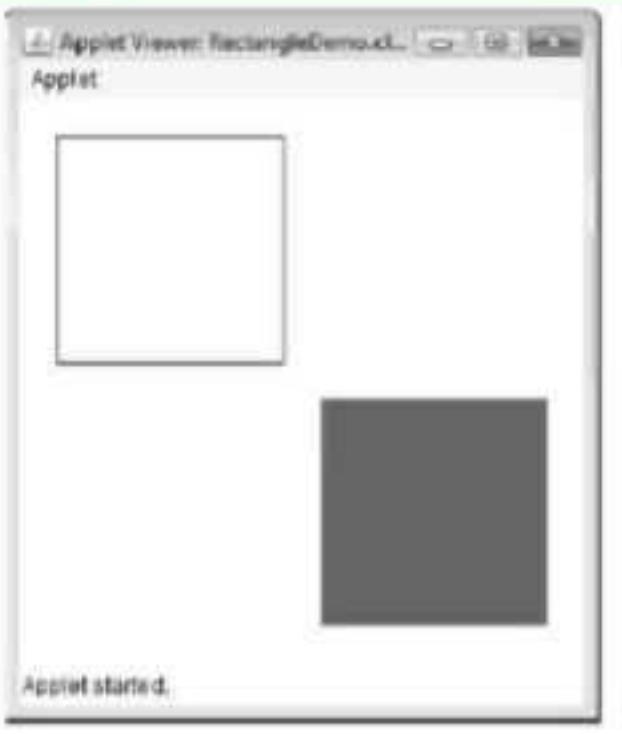
```
<applet code="RectangleDemo.class" width=300 height=300>
</applet>
```

Figure 14-19 shows the applet running in the applet viewer.

Code Listing 14-12 (RectangleDemo.java)

```
1  import javax.swing.*;
2  import java.awt.*;
3
4  /**
5     This class is an applet that demonstrates how
6     rectangles can be drawn.
7  */
8
9  public class RectangleDemo extends JApplet
10 {
11    /**
12       init method
13    */
14
15    public void init()
16    {
17       // Set the background color to white.
18       getContentPane().setBackground(Color.white);
19    }
20
21    /**
22       paint method
23       @param g The applet's Graphics object.
24    */
25
26    public void paint(Graphics g)
27    {
28       // Call the superclass paint method.
29       super.paint(g);
30
31       // Draw a black unfilled rectangle.
32       g.setColor(Color.black);
33       g.drawRect(20, 20, 120, 120);
34
35       // Draw a red filled rectangle.
36       g.setColor(Color.red);
37       g.fillRect(160, 160, 120, 120);
38    }
39 }
```

Figure 14-19 `RectangleDemo` applet



Code Listing 14-13 shows the `OvalDemo` class, an applet that draws two ovals. An oval is enclosed in an invisible rectangle that establishes the boundaries of the oval. The width and height of the enclosing rectangle defines the shape and size of the oval. This is illustrated in Figure 14-20.

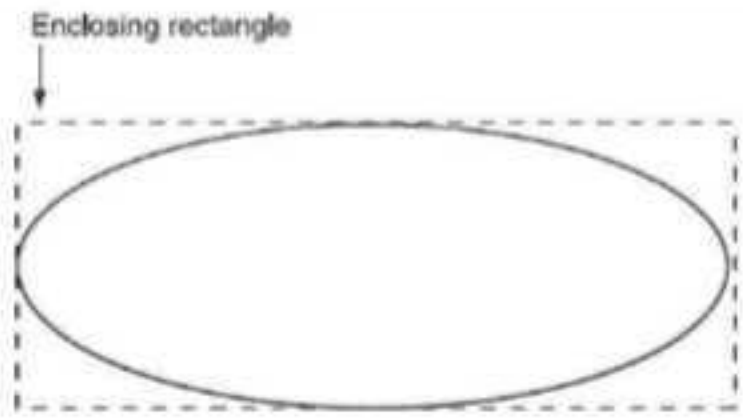
When you call the `drawOval` or `fillOval` method, you pass the *X* and *Y* coordinates of the enclosing rectangle's upper-left corner, and the width and height of the enclosing rectangle as arguments.

Code Listing 14-13 (`OvalDemo.java`)

```
1 import javax.swing.*;
2 import java.awt.*;
3
4 /**
5    This class is an applet that demonstrates how
6    ovals can be drawn.
7 */
8
9 public class OvalDemo extends JApplet
10 {
11    /**
12       init method
13    */
14
15    public void init()
16    {
```

```
17        // Set the background color to white.
18        getContentPane().setBackground(Color.white);
19     }
20
21     /**
22        paint method
23        @param g The applet's Graphics object.
24     */
25
26     public void paint(Graphics g)
27     {
28        // Call the superclass paint method.
29        super.paint(g);
30
31        // Draw a black unfilled oval.
32        g.setColor(Color.black);
33        g.drawOval(20, 20, 120, 75);
34
35        // Draw a green filled oval.
36        g.setColor(Color.green);
37        g.fillOval(80, 160, 180, 75);
38     }
39 }
```

Figure 14-20 An oval and its enclosing rectangle

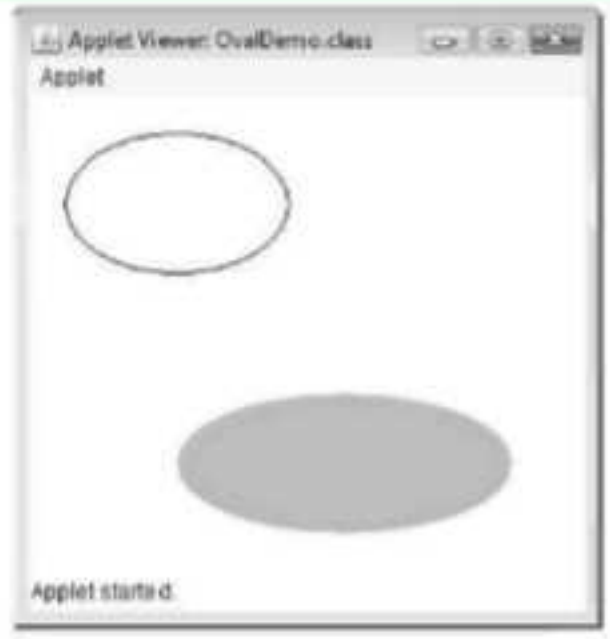


The file *OvalDemo.html*, which is in the same folder as the applet class, executes the applet with the following tag:

```
<applet code="OvalDemo.class" width=300 height=255>
</applet>
```

Figure 14-21 shows the applet running in the applet viewer.

Figure 14-21 `OvalDemo` applet



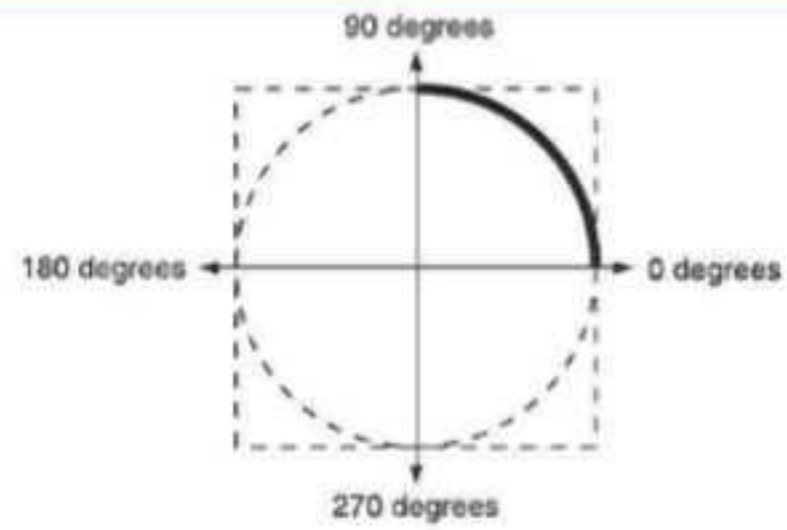
TIP: To draw a circle, simply draw an oval with an enclosing rectangle that is square. In other words, the enclosing rectangle's width and height should be the same.

The `drawArc` method draws an arc, which is part of an oval. You pass the same arguments to `drawArc` as you do to `drawOval`, plus two additional arguments: the arc's starting angle and ending angle. The angles are measured in degrees, with 0 degrees being at the 3 o'clock position. For example, look at the following statement:

```
g.drawArc(20, 20, 100, 100, 0, 90);
```

This statement creates an enclosing rectangle with its upper-left corner at (20, 20) and with a width and height of 100 pixels each. The oval constructed from this enclosing rectangle is a circle. The arc that is drawn is the part of the oval that starts at 0 degrees and ends at 90 degrees. Figure 14-22 illustrates this arc. The dashed lines show the enclosing rectangle and the oval. The thick black line shows the arc that will be drawn.

Figure 14-22 An arc



Code Listing 14-14 shows the `ArcDemo` class, which is an applet that draws four arcs: two unfilled and two filled. The filled arcs are drawn with the `fillArc` method.

The file *ArcDemo.html*, which is in the same folder as the applet class, executes the applet with the following tag:

```
<applet code="ArcDemo.class" width=300 height=220>
</applet>
```

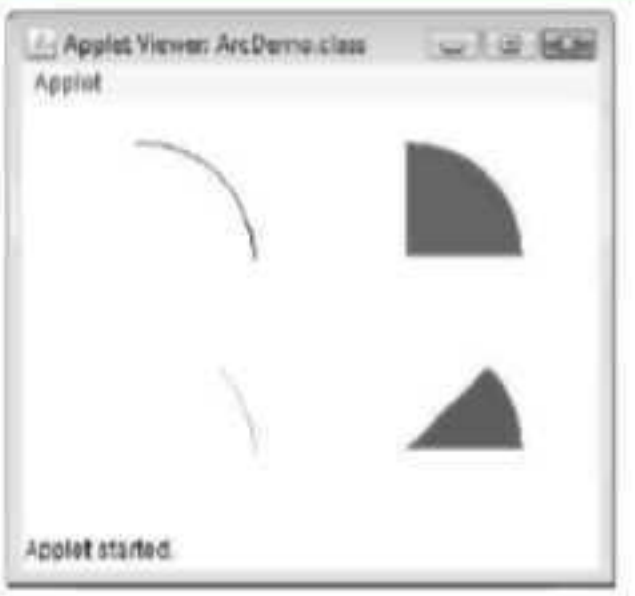
Figure 14-23 shows the applet running in the applet viewer.

Code Listing 14-14 (ArcDemo.java)

```
1  import javax.swing.*;
2  import java.awt.*;
3
4  /**
5     This class is an applet that demonstrates how
6     arcs can be drawn.
7  */
8
9  public class ArcDemo extends JApplet
10 {
11    /**
12       init method
13    */
14
15    public void init()
16    {
17       // Set the background color to white.
18       getContentPane().setBackground(Color.white);
19    }
20
21    /**
22       paint method
23       @param g The applet's Graphics object.
24    */
25
26    public void paint(Graphics g)
27    {
28       // Call the superclass paint method.
29       super.paint(g);
30
31       // Draw a black unfilled arc from 0 degrees
32       // to 90 degrees.
33       g.setColor(Color.black);
34       g.drawArc(0, 20, 120, 120, 0, 90);
35
36       // Draw a red filled arc from 0 degrees
37       // to 90 degrees.
```

```
38          g.setColor(Color.red);
39          g.fillArc(140, 20, 120, 120, 0, 90);
40
41          // Draw a green unfilled arc from 0 degrees
42          // to 45 degrees.
43          g.setColor(Color.green);
44          g.drawArc(0, 120, 120, 120, 0, 45);
45
46          // Draw a blue filled arc from 0 degrees
47          // to 45 degrees.
48          g.setColor(Color.blue);
49          g.fillArc(140, 120, 120, 120, 0, 45);
50      }
51  }
```

Figure 14-23 ArcDemo applet



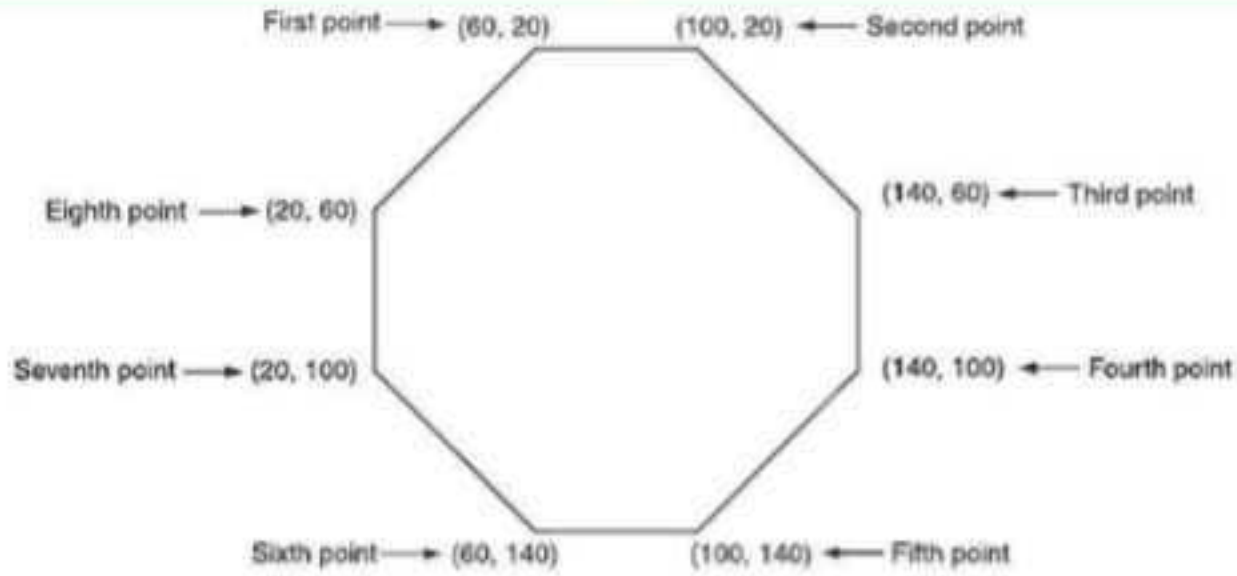
The `drawPolygon` method draws an outline of a closed polygon and the `fillPolygon` method draws a closed polygon filled with the current drawing color. A polygon is constructed of multiple line segments that are connected. The point where two line segments are connected is called a *vertex*. These methods accept two `int` arrays as arguments. The first array contains the *X* coordinates of each vertex, and the second array contains the *Y* coordinates of each vertex. The third argument is an `int` that specifies the number of vertices, or connecting points.

For example, suppose we use the following arrays as arguments for the *X* and *Y* coordinates of a polygon:

```
int[] xCoords = {60, 100, 140, 140, 100, 60, 20, 20 };
int[] yCoords = {20, 20, 60, 100, 140, 140, 100, 60 };
```

The first point specified by these arrays is (60, 20), the second point is (100, 20), and so forth. There are a total of eight points specified by these arrays, and if we connect each of these points we get the octagon shown in Figure 14-24.

Figure 14-24 Points of each vertex in an octagon



If the last point specified in the arrays is different from the first point, as in this example, then the two points are automatically connected to close the polygon. The `PolygonDemo` class in Code Listing 14-15 draws a filled polygon using these arrays as arguments.

Code Listing 14-15 (PolygonDemo.java)

```
1 import javax.swing.*;
2 import java.awt.*;
3
4 /**
5    This class is an applet that demonstrates how a
6    polygon can be drawn.
7 */
8
9 public class PolygonDemo extends JApplet
10 {
11    /**
12       init method
13    */
14
15    public void init()
16    {
17       // Set the background color to white.
18       getContentPane().setBackground(Color.white);
19    }
20
21    /**
22       paint method
23       @param g The applet's Graphics object.
24    */
25
26    public void paint(Graphics g)
```

```
27    {
28       int[] xCoords = {60, 100, 140, 140,
29                        100, 60, 20, 20 };
30       int[] yCoords = {20, 20, 60, 100,
31                        140, 140, 100, 60 };
32
33       // Call the superclass paint method.
34       super.paint(g);
35
36       // Set the drawing color.
37       g.setColor(Color.red);
38
39       // Draw the polygon.
40       g.fillPolygon(xCoords, yCoords, 8);
41    }
42 }
```

The file *PolygonDemo.html*, which is in the same folder as the applet class, executes the applet with the following tag:

```
<applet code="PolygonDemo.class" width=160 height=160>
</applet>
```

Figure 14-25 shows the applet running in the applet viewer.

Figure 14-25 `PolygonDemo` applet

The `drawString` method draws a string as a graphic. The string is specified by its first argument, a `String` object. The *X* and *Y* coordinates of the lower-left point of the string are specified by the second and third arguments. For example, assuming that `g` references a `Graphics` object, the following statement draws the string "Hello World", starting at the coordinates 100, 50:

```
g.drawstring("Hello World", 100, 50);
```

You can set the font for the string with the `setFont` method. This method accepts a `Font` object as its argument. Here is an example:

```
g.setFont(new Font("Serif", Font.ITALIC, 20));
```

The `Font` class was covered in Chapter 13. Recall that the `Font` constructor's arguments are the name of a font, the font's style, and the font's size in points. You can combine font styles with the + operator, as follows:

```
g.setFont(new Font("Serif", Font.BOLD + Font.ITALIC, 24));
```

The `GraphicStringDemo` class in Code Listing 14-16 demonstrates the `drawString` method. It draws the same octagon that the `PolygonDemo` class drew, and then draws the string "STOP" over it to create a stop sign. The string is drawn in a bold 35-point sanserif font.

Code Listing 14-16 **(GraphicStringDemo.java)**

```
1  import javax.swing.*;
2  import java.awt.*;
3
4  /**
5     This class is an applet that demonstrates how a
6     string can be drawn.
7  */
8
9  public class GraphicStringDemo extends JApplet
10 {
11    /**
12       init method
13    */
14
15    public void init()
16    {
17       // Set the background color to white.
18       getContentPane().setBackground(Color.white);
19    }
20
21    /**
22       paint method
23       @param g The applet's Graphics object.
24    */
25
26    public void paint(Graphics g)
27    {
28       int[] xCoords = {60, 100, 140, 140,
29                        100, 60, 20, 20 };
30       int[] yCoords = {20, 20, 60, 100,
31                        140, 140, 100, 60 };
32
```

```
33        // Call the superclass paint method.
34        super.paint(g);
35
36        // Set the drawing color.
37        g.setColor(Color.red);
38
39        // Draw the polygon.
40        g.fillPolygon(xCoords, yCoords, 8);
41
42        // Set the drawing color to white.
43        g.setColor(Color.white);
44
45        // Set the font and draw "STOP".
46        g.setFont(new Font("SansSerif", Font.BOLD, 35));
47        g.drawString("STOP", 35, 95);
48     }
49 }
```

The file *GraphicStringDemo.html*, which is in the same folder as the applet class, executes the applet with the following tag:

```
<applet code="GraphicStringDemo.class" width=160 height=160>
</applet>
```

Figure 14-26 shows the applet running in the applet viewer.

Figure 14-26 `GraphicStringDemo` applet

The `repaint` Method

As previously mentioned, you do not call a component's `paint` method. It is automatically called when the component must be redisplayed. Sometimes, however, you might want to force the application or applet to call the `paint` method. You do this by calling the `repaint` method, which has the following header:

```
public void repaint()
```

The `repaint` method clears the surface of the component and then calls the `paint` method. You will see an applet that uses this method in a moment.

Drawing on Panels

Each of the preceding examples uses the entire `JApplet` window as a canvas for drawing. Sometimes, however, you might want to confine your drawing space to a smaller region within the window, such as a panel. To draw on a panel, you simply get a reference to the panel's `Graphics` object and then use that object's methods to draw. The resulting graphics are drawn only on the panel.

Getting a reference to a `JPanel` component's `Graphics` object is similar to the technique you saw in the previous examples. Instead of overriding the `JPanel` object's `paint` method, however, you should override its `paintComponent` method. This is true not only for `JPanel` objects, but also for all Swing components except `JApplet` and `JFrame`. The `paintComponent` method serves for `JPanel` and most other Swing objects the same purpose as the `paint` method: It is automatically called when the component needs to be redisplayed. When it is called, the component's `Graphics` object is passed as an argument. Here is the method's header:

```
public void paintComponent(Graphics g)
```

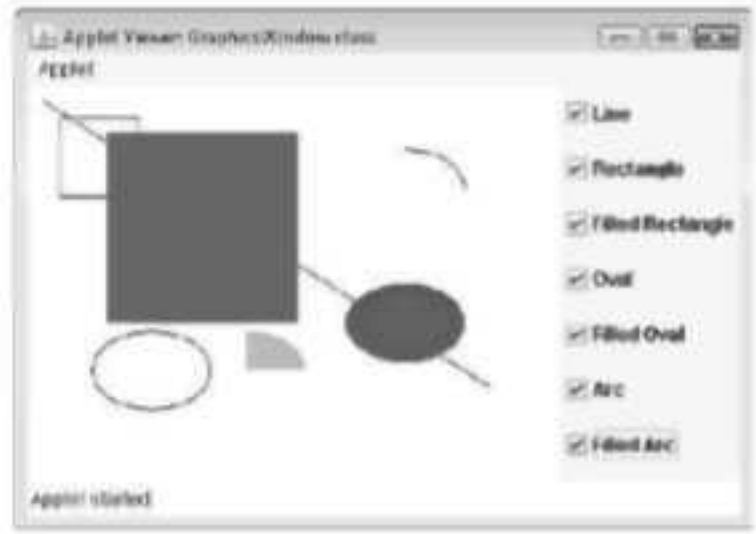
When you override this method, first you should call the superclass's `paintComponent` method to ensure that the component is properly displayed. Here is an example call to the superclass's version of the method:

```
super.paintComponent(g);
```

After this you can call any of the `Graphics` object's methods to draw on the component. As an example, we look at the `GraphicsWindow` class in Code Listing 14-17. When this applet is run (via the *GraphicsWindow.html* file, which is in the same folder as the applet class), the window shown in Figure 14-27 is displayed. A set of check boxes is displayed in a `JPanel` component on the right side of the window. The white area that occupies the majority of the window is a `DrawingPanel` object. The `DrawingPanel` class inherits from `JPanel`, and its code is shown in Code Listing-14-18. When one of the check boxes is selected, a shape appears in the `DrawingPanel` object. Figure 14-28 shows how the applet window appears when all of the check boxes are selected.

Figure 14-27 `GraphicsWindow` applet

Figure 14-28 GraphicsWindow applet with all graphics selected



Code Listing 14-17 (GraphicsWindow.java)

```
1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.event.*;
4
5 /**
6    This class displays a drawing panel and a set of
7    check boxes that allow the user to select shapes.
8    The selected shapes are drawn on the drawing panel.
9 */
10
11 public class GraphicsWindow extends JApplet
12 {
13    // Declare an array of check box components
14    private JCheckBox[] checkBoxes;
15
16    // The following titles array contains the
17    // titles of the check boxes.
18    private String[] titles = { "Line", "Rectangle",
19                                "Filled Rectangle",
20                                "Oval", "Filled Oval",
21                                "Arc", "Filled Arc" };
22
23    // The following will reference a panel to contain
24    // the check boxes.
25    private JPanel checkBoxPanel;
26
27    // The following will reference an instance of the
28    // DrawingPanel class. This will be a panel to draw on.
29    private DrawingPanel drawingPanel;
```

```
30
31    /**
32       init method
33    */
34
35    public void init()
36    {
37       // Build the check box panel.
38       buildCheckBoxPanel();
39
40       // Create the drawing panel.
41       drawingPanel = new DrawingPanel(checkBoxes);
42
43       // Add the check box panel to the east region
44       // and the drawing panel to the center region.
45       add(checkBoxPanel, BorderLayout.EAST);
46       add(drawingPanel, BorderLayout.CENTER);
47    }
48
49    /**
50       The buildCheckBoxPanel method creates the array of
51       check box components and adds them to a panel.
52    */
53
54    private void buildCheckBoxPanel()
55    {
56       // Create the panel.
57       checkBoxPanel = new JPanel();
58       checkBoxPanel.setLayout(new GridLayout(7, 1));
59
60       // Create the check box array.
61       checkBoxes = new JCheckBox[7];
62
63       // Create the check boxes and add them to the panel.
64       for (int i = 0; i < checkBoxes.length; i++)
65       {
66          checkBoxes[i] = new JCheckBox(titles[i]);
67          checkBoxes[i].addItemListener(
68                                      new CheckBoxListener());
69          checkBoxPanel.add(checkBoxes[i]);
70       }
71    }
72
73    /**
74       A private inner class to respond to changes in the
75       state of the check boxes.
76    */
77
```

```
78    private class CheckBoxListener implements ItemListener
79    {
80       public void itemStateChanged(ItemEvent e)
81       {
82          drawingPanel.repaint();
83       }
84    }
85 }
```

Code Listing 14-18 (DrawingPanel.java)

```
1  import javax.swing.*;
2  import java.awt.*;
3
4  /**
5     This class creates a panel that example shapes are
6     drawn on.
7  */
8
9  public class DrawingPanel extends JPanel
10 {
11    // Declare a check box array.
12    private JCheckBox[] checkBoxArray;
13
14    /**
15       Constructor
16    */
17
18    public DrawingPanel(JCheckBox[] cbArray)
19    {
20       // Reference the check box array.
21       checkBoxArray = cbArray;
22
23       // Set the background color to white.
24       setBackground(Color.white);
25
26       // Set the preferred size of the panel.
27       setPreferredSize(new Dimension(300, 200));
28    }
29
30    /**
31       paintComponent method
32       @param g The panel's Graphics object.
33    */
34
35    public void paintComponent(Graphics g)
```

```
36    {
37       // Call the superclass paintComponent method.
38       super.paintComponent(g);
39
40       // Draw the selected shapes.
41       if (checkBoxArray[0].isSelected())
42       {
43          g.setColor(Color.black);
44          g.drawLine(10, 10, 290, 190);
45       }
46       if (checkBoxArray[1].isSelected())
47       {
48          g.setColor(Color.black);
49          g.drawRect(20, 20, 50, 50);
50       }
51       if (checkBoxArray[2].isSelected())
52       {
53          g.setColor(Color.red);
54          g.fillRect(50, 30, 120, 120);
55       }
56       if (checkBoxArray[3].isSelected())
57       {
58          g.setColor(Color.black);
59          g.drawOval(40, 155, 75, 50);
60       }
61       if (checkBoxArray[4].isSelected())
62       {
63          g.setColor(Color.blue);
64          g.fillOval(200, 125, 75, 50);
65       }
66       if (checkBoxArray[5].isSelected())
67       {
68          g.setColor(Color.black);
69          g.drawArc(200, 40, 75, 50, 0, 90);
70       }
71       if (checkBoxArray[6].isSelected())
72       {
73          g.setColor(Color.green);
74          g.fillArc(100, 155, 75, 50, 0, 90);
75       }
76    }
77 }
```

Let's take a closer look at the applet's code. First, notice in lines 14 through 21 of the `GraphicsWindow` class (in Code Listing 14-17) that two of the class's fields are array reference variables. The `checkBoxes` variable references an array of `JCheckBox` components, and the `titles` variable references an array of strings. The strings in the `titles` array are the titles that the check boxes will display.

The first statement in the `init` method, line 38, is a call to the `buildCheckBoxPanel` method, which creates a panel for the check boxes, creates the array of check boxes, adds an item listener to each element of the array, and adds each element to the panel.

After the `buildCheckBoxPanel` method executes, the `init` method creates a `DrawingPanel` object with the statement in line 41. Notice that the `checkBoxes` variable is passed to the `DrawingPanel` constructor. The `drawingPanel` object needs a reference to the array so its `paintComponent` method can determine which check boxes are selected and draw the corresponding shape.

The only times that the `paintComponent` method is automatically called is when the component is initially displayed and when the component needs to be redisplayed. In order to display a shape immediately when the user selects a check box, we need the check box item listener to force the `paintComponent` method to be called. This is accomplished by the statement in line 82, in the `CheckBoxListener` class's `itemStateChanged` method. This statement calls the `drawingPanel` object's `repaint` method, which causes the `drawingPanel` object's surface to be cleared, and then causes the object's `paintComponent` method to execute. Because it is in the item listener, it is executed each time the user clicks on a check box.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

14.18 In an AWT component, or a class that extends `JApplet` or `JFrame`, if you want to get a reference to the `Graphics` object, do you override the `paint` or `paintComponent` method?

14.19 In a `JPanel` object, do you override the `paint` or `paintComponent` method to get a reference to the `Graphics` object?

14.20 When are the `paint` and `paintComponent` method called?

14.21 In the `paint` or `paintComponent` method, what should be done before anything else?

14.22 How do you force the `paint` or `paintComponent` method to be called?

14.23 When using a `Graphics` object to draw an oval, what invisible shape is the oval enclosed in?

14.24 What values are contained in the two arrays that are passed to a `Graphics` object's `drawPolygon` method?

14.25 What `Graphics` class methods do you use to perform the following tasks?

a) draw a line
b) draw a filled rectangle
c) draw a filled oval
d) draw a filled arc
e) set the drawing color
f) draw a rectangle
g) draw an oval
h) draw an arc
i) draw a string
j) set the font

14.6 Handling Mouse Events

CONCEPT: Java allows you to create listener classes that handle events generated by the mouse.

Handling Mouse Events

The mouse generates two types of events: mouse events and mouse motion events. To handle mouse events you create a *mouse listener* class and/or a *mouse motion listener* class. A mouse listener class can respond to any of the follow events:

- The mouse button is pressed.
- The mouse button is released.
- The mouse button is clicked (pressed, then released without moving the mouse).
- The mouse cursor enters a component's screen space.
- The mouse cursor exits a component's screen space.

A mouse listener class must implement the `MouseListener` interface, which is in the `java.awt.event` package. The class must also have the methods listed in Table 14-3.

Table 14-3 Methods required by the `MouseListener` interface

Method	Description
`public void mousePressed(MouseEvent e)`	If the mouse cursor is over the component and the mouse button is pressed, this method is called.
`public void mouseClicked(MouseEvent e)`	A mouse click is defined as pressing the mouse button and releasing it without moving the mouse. If the mouse cursor is over the component and the mouse is clicked on, this method is called.
`public void mouseReleased(MouseEvent e)`	This method is called when the mouse button is released after it has been pressed. The `mousePressed` method is always called before this method.
`public void mouseEntered(MouseEvent e)`	This method is called when the mouse cursor enters the screen area belonging to the component.
`public void mouseExited(MouseEvent e)`	This method is called when the mouse cursor leaves the screen area belonging to the component.

Notice that each of the methods listed in Table 14-3 accepts a `MouseEvent` object as its argument. The `MouseEvent` object contains data about the mouse event. We will use two

of the `MouseEvent` object's methods: `getX` and `getY`. These methods return the *X* and *Y* coordinates of the mouse cursor when the event occurs.

Once you create a mouse listener class, you can register it with a component using the `addMouseListener` method, which is inherited from the `Component` class. The appropriate methods in the mouse listener class are automatically called when their corresponding mouse events occur.

A mouse motion listener class can respond to the following events:

- The mouse is dragged (the button is pressed and the mouse is moved while the button is held down).
- The mouse is moved.

A mouse motion listener class must implement the `MouseMotionListener` interface, which is in the `java.awt.event` package. The class must also have the methods listed in Table 14-4. Notice that each of these methods also accepts a `MouseEvent` object as an argument.

Table 14-4 Methods required by the `MouseMotionListener` interface

Method	Description
`public void mouseDragged(MouseEvent e)`	The mouse is dragged when its button is pressed and the mouse is moved while the button is held down. This method is called when a dragging operation begins over the component. The `mousePressed` method is always called just before this method.
`public void mouseMoved(MouseEvent e)`	This method is called when the mouse cursor is over the component and it is moved.

Once you create a mouse motion listener class, you can register it with a component using the `addMouseMotionListener` method, which is inherited from the `Component` class. The appropriate methods in the mouse motion listener class are automatically called when their corresponding mouse events occur.

The `MouseEvents` class, shown in Code Listing 14-19, is an applet that demonstrates both a mouse listener and a mouse motion listener. The file *MouseEvents.html*, which is in the same folder as the applet class, can be used to start the applet. Figure 14-29 shows the applet running. The window displays a group of read-only text fields that represent the different mouse and mouse motion events. When an event occurs, the corresponding text field turns yellow. The last two text fields constantly display the mouse cursor's *X* and *Y* coordinates. Run this applet and experiment by clicking the mouse inside the window, dragging the mouse, moving the mouse cursor in and out of the window, and moving the mouse cursor over the text fields.