After this statement executes, the `JPanel` referenced by `panel` will have an empty border of five pixels around each edge.

NOTE: In case you've skipped ahead to this chapter, the `BorderFactory` methods are static, which means that you call them without creating an instance of the `BorderFactory` class. (You simply write *`BorderFactory.`* before the method name to call the method.) This is similar to the way the `Math` class and wrapper class methods we have discussed are called. Static methods are covered in Chapter 8.

Line Borders

A line border is a line of a specified color and thickness that appears around the edges of a component. To create a line border, call the `BorderFactory` class's `createLineBorder` method. Here is the method's general format:

```
BorderFactory.createLineBorder(Color color, int thickness);
```

The arguments passed into `color` and `thickness` specify the color of the line and the size of the line in pixels. The method returns a reference to a `Border` object. The following is an example of a statement that uses the method. Assume that the `panel` variable references a `JPanel` object.

```
panel.setBorder(BorderFactory.createLineBorder(Color.RED, 1));
```

After this statement executes, the `JPanel` referenced by `panel` will have a red line border that is one pixel thick around its edges.

Titled Borders

A titled border is an etched border with a title displayed on it. To create a titled border, call the `BorderFactory` class's `createTitledBorder` method. Here is the method's general format:

```
BorderFactory.createTitledBorder(String title);
```

The argument passed into `title` is the text that is to be displayed as the border's title. The method returns a reference to a `Border` object. The following is an example of a statement that uses the method. Assume that the `panel` variable references a `JPanel` object.

```
panel.setBorder(BorderFactory.createTitledBorder("Choices"));
```

After this statement executes, the `JPanel` referenced by `panel` will have an etched border with the title "Choices" displayed on it.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

12.26 What method do you use to set a border around a component?

12.27 What is the preferred way of creating a `Border` object?

12.6 Focus on Problem Solving: Extending Classes from `JPanel`

CONCEPT: By writing a class that is extended from the `JPanel` class, you can create a custom panel component that can hold other components and their related code.

In the applications that you have studied so far in this chapter, we have used the `extends JFrame` clause in the class header to extend the class from the `JFrame` class. Recall that the extended class is then a specialized version of the `JFrame` class, and we use its constructor to create the panels, buttons, and all of the other components needed. This approach works well for simple applications. But for applications that use many components, this approach can be cumbersome. Bundling all of the code and event listeners for a large number of components into a single class can lead to a large and complex class. A better approach is to encapsulate smaller groups of related components and their event listeners into their own classes.

A commonly used technique is to extend a class from the `JPanel` class. This allows you to create your own specialized panel component, which can contain other components and related code such as event listeners. A complex application that uses numerous components can be constructed from several specialized panel components. In this section we will examine such an application.

The Brandi's Bagel House Application

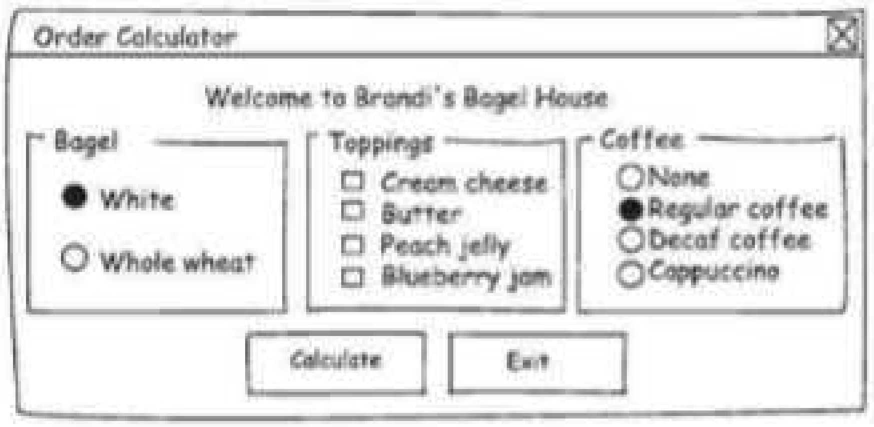
Brandi's Bagel House has a bagel and coffee delivery service for the businesses in her neighborhood. Customers may call in and order white and whole wheat bagels with a variety of toppings. In addition, customers may order three different types of coffee. (Delivery for coffee alone is not available, however.) Here is a complete price list:

Bagels:	*White bagel $1.25, whole wheat bagel $1.50*
Toppings:	*Cream cheese $0.50, butter $0.25, peach jelly $0.75, blueberry jam $0.75*
Coffee:	*Regular coffee $1.25, decaf coffee $1.25, cappuccino $2.00*

Brandi, the owner, needs an "order calculator" application that her staff can use to calculate the price of an order as it is called in. The application should display the subtotal, the amount of a 6 percent sales tax, and the total of the order. Figure 12-30 shows a sketch of the application's window. The user selects the type of bagel, toppings, and coffee, then clicks the Calculate button. A dialog box appears displaying the subtotal, amount of sales tax, and total. The user can exit the application by clicking either the Exit button or the standard close button in the upper-right corner.

The layout shown in the sketch can be achieved using a `BorderLayout` manager with the window's content pane. The label that displays "Welcome to Brandi's Bagel House" is in the north region, the radio buttons for the bagel types are in the west region, the check boxes for the toppings are in the center region, the radio buttons for the coffee selection are in the east region, and the Calculate and Exit buttons are in the south region. To construct this window we create the following specialized panel classes that are extended from `JPanel`:

Figure 12-30 Sketch of the Order Calculator window



- **GreetingsPanel.** This panel contains the label that appears in the window's north region.
- **BagelPanel.** This panel contains the radio buttons for the types of bagels.
- **ToppingPanel.** This panel contains the check boxes for the types of bagels.
- **CoffeePanel.** This panel contains the radio buttons for the coffee selections.

(We will not create a specialized panel for the Calculate and Exit buttons. The reason is explained later.) After these classes have been created, we can create objects from them and add the objects to the correct regions of the window's content pane. Let's take a closer look at each of these classes.

The GreetingPanel Class

The GreetingPanel class holds the label displaying the text "Welcome to Brandi's Bagel House". Code Listing 12-16 shows the class, which extends JPanel.

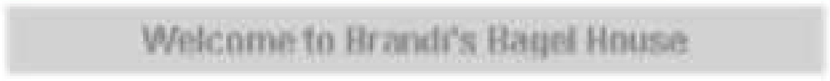
Code Listing 12-16 **(GreetingPanel.java)**

```
import javax.swing.*;

/**
   The GreetingPanel class displays a greeting in a panel.
*/

public class GreetingPanel extends JPanel
{
   private JLabel greeting; // To display a greeting

   /**
      Constructor
   */

   public GreetingPanel()
   {
      // Create the label.
18          greeting = new JLabel("Welcome to Brandi's Bagel House");
19
20          // Add the label to this panel.
21          add(greeting);
22       }
23    }
```

In line 21 the `add` method is called to add the `JLabel` component referenced by `greeting`. Notice that we are calling the method without an object reference and a dot preceding it. This is because the method was inherited from the `JPanel` class, and we can call it just as if it were written into the `GreetingPanel` class declaration.

When we create an instance of this class, we are creating a `JPanel` component that displays a label with the text "Welcome to Brandi's Bagel House". Figure 12-31 shows how the component will appear when it is placed in the window's north region.

Figure 12-31 Appearance of the `GreetingPanel` component


Welcome to Brandi's Bagel House

The `BagelPanel` Class

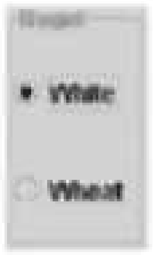
The `BagelPanel` class holds the radio buttons for the types of bagels. Notice that this panel uses a `GridLayout` manager with two rows and one column. Code Listing 12-17 shows the class, which is extended from `JPanel`.

Code Listing 12-17 **(`BagelPanel.java`)**

```
import javax.swing.*;
import java.awt.*;

/**
   The BagelPanel class allows the user to select either
   a white or whole wheat bagel.
*/

public class BagelPanel extends JPanel
{
   // The following constants are used to indicate
   // the cost of each type of bagel.
   public final double WHITE_BAGEL = 1.25;
   public final double WHEAT_BAGEL = 1.50;

   private JRadioButton whiteBagel;  // To select white
   private JRadioButton wheatBagel;  // To select wheat
   private ButtonGroup bg;          // Radio button group

   /**
      Constructor
   */

   public BagelPanel()
   {
      // Create a GridLayout manager with
      // two rows and one column.
      setLayout(new GridLayout(2, 1));

      // Create the radio buttons.
      whiteBagel = new JRadioButton("White", true);
      wheatBagel = new JRadioButton("Wheat");

      // Group the radio buttons.
      bg = new ButtonGroup();
      bg.add(whiteBagel);
      bg.add(wheatBagel);

      // Add a border around the panel.
      setBorder(BorderFactory.createTitledBorder("Bagel"));

      // Add the radio buttons to the panel.
      add(whiteBagel);
      add(wheatBagel);
   }

   /**
      getBagelCost method
      @return The cost of the selected bagel.
   */

   public double getBagelCost()
   {
      double bagelCost = 0.0;

      if (whiteBagel.isSelected())
         bagelCost = WHITE_BAGEL;
      else
         bagelCost = WHEAT_BAGEL;

      return bagelCost;
   }
}
```

Notice that the `whiteBagel` radio button is automatically selected when it is created. This is the default choice. This class does not have an inner event listener class because we do not want to execute any code when the user selects a bagel. Instead, we want this class to be able to report the cost of the selected bagel. That is the purpose of the `getBagelCost` method, which returns the cost of the selected bagel as a `double`. (This method will be called by the Calculate button's event listener.) Figure 12-32 shows how the component appears when it is placed in the window's west region.

Figure 12-32 Appearance of the `BagelPanel` component



The `ToppingPanel` Class

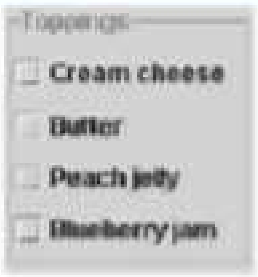
The `ToppingPanel` class holds the check boxes for the available toppings. Code Listing 12-18 shows the class, which is also extended from `JPanel`.

Code Listing 12-18 **(ToppingPanel.java)**

```
import javax.swing.*;
import java.awt.*;

/**
   The ToppingPanel class allows the user to select
   the toppings for the bagel.
*/

public class ToppingPanel extends JPanel
{
   // The following constants are used to indicate
   // the cost of toppings.
   public final double CREAM_CHEESE = 0.50;
   public final double BUTTER = 0.25;
   public final double PEACH_JELLY = 0.75;
   public final double BLUEBERRY_JAM = 0.75;

   private JCheckBox creamCheese;     // To select cream cheese
   private JCheckBox butter;          // To select butter
   private JCheckBox peachJelly;      // To select peach jelly
   private JCheckBox blueberryJam;    // To select blueberry jam

   /**
      Constructor
   */

   public ToppingPanel()
   {
      // Create a GridLayout manager with
      // four rows and one column.
      setLayout(new GridLayout(4, 1));

      // Create the check boxes.
      creamCheese = new JCheckBox("Cream cheese");
      butter = new JCheckBox("Butter");
      peachJelly = new JCheckBox("Peach jelly");
      blueberryJam = new JCheckBox("Blueberry jam");

      // Add a border around the panel.
      setBorder(BorderFactory.createTitledBorder("Toppings"));

      // Add the check boxes to the panel.
      add(creamCheese);
      add(butter);
      add(peachJelly);
      add(blueberryJam);
   }

   /**
      getToppingCost method
      @return The cost of the selected toppings.
   */

   public double getToppingCost()
   {
      double toppingCost = 0.0;

      if (creamCheese.isSelected())
         toppingCost += CREAM_CHEESE;
      if (butter.isSelected())
         toppingCost += BUTTER;
      if (peachJelly.isSelected())
         toppingCost += PEACH_JELLY;
      if (blueberryJam.isSelected())
         toppingCost += BLUEBERRY_JAM;

      return toppingCost;
   }
}
```

As with the `BagelPanel` class, this class does not have an inner event listener class because we do not want to execute any code when the user selects a topping. Instead, we want this class to be able to report the total cost of all the selected toppings. That is the purpose of the `getToppingCost` method, which returns the cost of all the selected toppings as a `double`. (This method will be called by the Calculate button's event listener.) Figure 12-33 shows how the component appears when it is placed in the window's center region.

Figure 12-33 Appearance of the `ToppingPanel` component



The `CoffeePanel` Class

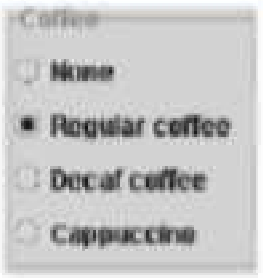
The `CoffeePanel` class holds the radio buttons for the available coffee selections. Code Listing 12-19 shows the class, which extends `JPanel`.

Code Listing 12-19 (`CoffeePanel.java`)

```java
import javax.swing.*;
import java.awt.*;

/**
   The CoffeePanel class allows the user to select coffee.
*/

public class CoffeePanel extends JPanel
{
   // The following constants are used to indicate
   // the cost of coffee.
   public final double NO_COFFEE = 0.0;
   public final double REGULAR_COFFEE = 1.25;
   public final double DECAF_COFFEE = 1.25;
   public final double CAPPUCCINO = 2.00;

   private JRadioButton noCoffee;       // To select no coffee
   private JRadioButton regularCoffee;  // To select regular coffee
   private JRadioButton decafCoffee;    // To select decaf
   private JRadioButton cappuccino;     // To select cappuccino
   private ButtonGroup bg;              // Radio button group

   /**
      Constructor
   */

   public CoffeePanel()
   {
      // Create a GridLayout manager with
      // four rows and one column.
      setLayout(new GridLayout(4, 1));

      // Create the radio buttons.
      noCoffee = new JRadioButton("None");
      regularCoffee = new JRadioButton("Regular coffee", true);
      decafCoffee = new JRadioButton("Decaf coffee");
      cappuccino = new JRadioButton("Cappuccino");

      // Group the radio buttons.
      bg = new ButtonGroup();
      bg.add(noCoffee);
      bg.add(regularCoffee);
      bg.add(decafCoffee);
      bg.add(cappuccino);

      // Add a border around the panel.
      setBorder(BorderFactory.createTitledBorder("Coffee"));

      // Add the radio buttons to the panel.
      add(noCoffee);
      add(regularCoffee);
      add(decafCoffee);
      add(cappuccino);
   }

   /**
      getCoffeeCost method
      @return The cost of the selected coffee.
   */

   public double getCoffeeCost()
   {
      double coffeeCost = 0.0;

      if (noCoffee.isSelected())
         coffeeCost = NO_COFFEE;
      else if (regularCoffee.isSelected())
         coffeeCost = REGULAR_COFFEE;
      else if (decafCoffee.isSelected())
         coffeeCost = DECAF_COFFEE;
      else if (cappuccino.isSelected())
         coffeeCost = CAPPUCCINO;

         return coffeeCost;
      }
   }
```

As with the `BagelPanel` and `ToppingPanel` classes, this class does not have an inner event listener class because we do not want to execute any code when the user selects coffee. Instead, we want this class to be able to report the cost of the selected coffee. The `getCoffeeCost` method returns the cost of the selected coffee as a `double`. (This method will be called by the Calculate button's event listener.) Figure 12-34 shows how the component appears when it is placed in the window's east region.

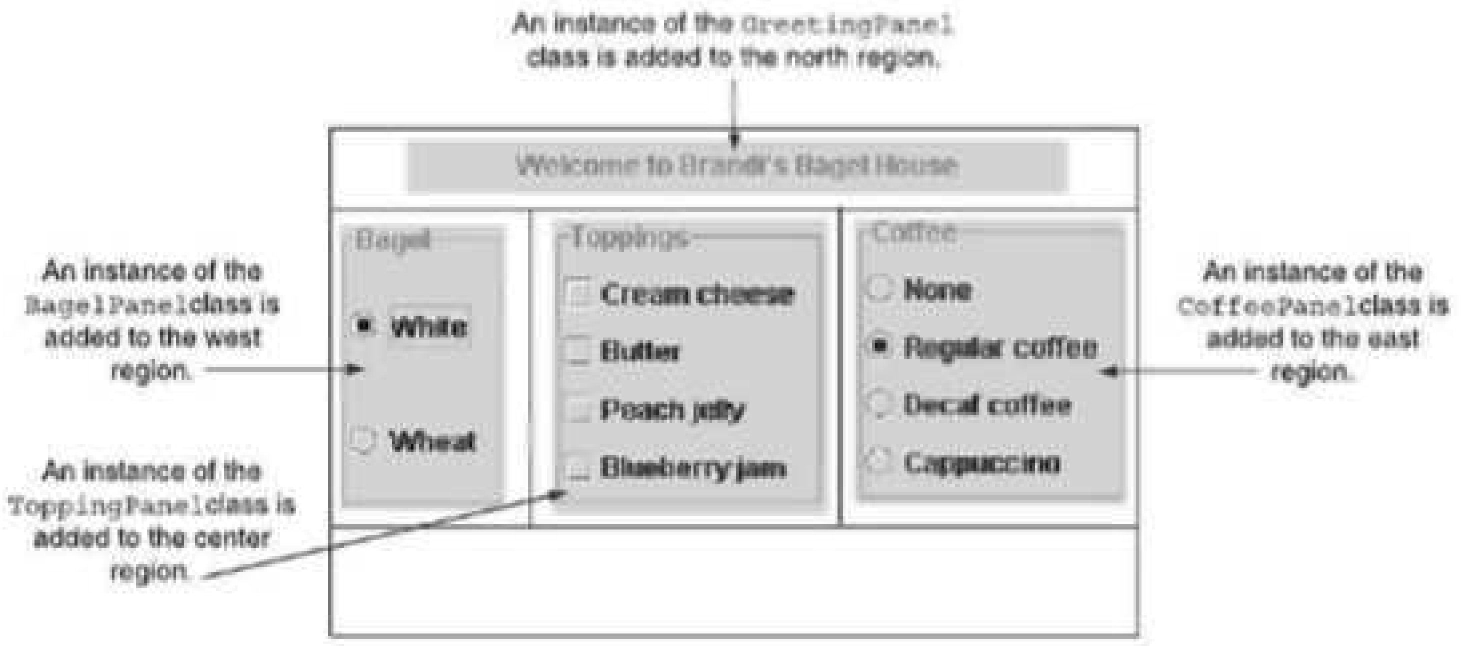
Figure 12-34 Appearance of the `CoffeePanel` component



Putting It All Together

The last step in creating this application is to write a class that builds the application's window and adds the Calculate and Exit buttons. This class, which we name `OrderCalculatorGUI`, is extended from `JFrame` and uses a `BorderLayout` manager with its content pane. Figure 12-35 shows how instances of the `GreetingPanel`, `BagelPanel`, `ToppingPanel`, and `CoffeePanel` classes are placed in the content pane.

Figure 12-35 Placement of the custom panels


An instance of the GreetingPanel class is added to the north region.
Welcome to Brandi's Bagel House
Bagel
White
Wheat
Toppings
Cream cheese
Butter
Peach jelly
Blueberry jam
Coffee
None
Regular coffee
Decaf coffee
Cappuccino
An instance of the BagelPanel class is added to the west region.
An instance of the ToppingPanel class is added to the center region.
An instance of the CoffeePanel class is added to the east region.

We have not created a custom panel class to hold the Calculate and Exit buttons. The reason is that the Calculate button's event listener must call the `getBagelCost`, `getToppingCost`, and `getCoffeeCost` methods. In order to call those methods, the event listener must have access to the `BagelPanel`, `ToppingPanel`, and `CoffeePanel` objects that are created in the `OrderCalculatorGUI` class. The approach taken in this example is to have the `OrderCalculatorGUI` class itself create the buttons. The code for the `OrderCalculatorGUI` class is shown in Code Listing 12-20.

Code Listing 12-20 (`OrderCalculatorGUI.java`)

```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.text.DecimalFormat;

/**
   The OrderCalculatorGUI class creates the GUI for the
   Brandi's Bagel House application.
*/

public class OrderCalculatorGUI extends JFrame
{
   private BagelPanel bagels;     // Bagel panel
   private ToppingPanel toppings; // Topping panel
   private CoffeePanel coffee;    // Coffee panel
   private GreetingPanel banner;  // To display a greeting
   private JPanel buttonPanel;    // To hold the buttons
   private JButton calcButton;    // To calculate the cost
   private JButton exitButton;    // To exit the application
   private final double TAX_RATE = 0.06; // Sales tax rate

   /**
      Constructor
   */

   public OrderCalculatorGUI()
   {
      // Display a title.
      setTitle("Order Calculator");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create a BorderLayout manager.
      setLayout(new BorderLayout());

      // Create the custom panels.
      banner = new GreetingPanel();
      bagels = new BagelPanel();
      toppings = new ToppingPanel();
      coffee = new CoffeePanel();

      // Create the button panel.
      buildButtonPanel();

      // Add the components to the content pane.
      add(banner, BorderLayout.NORTH);
      add(bagels, BorderLayout.WEST);
      add(toppings, BorderLayout.CENTER);
      add(coffee, BorderLayout.EAST);
      add(buttonPanel, BorderLayout.SOUTH);

      // Pack the contents of the window and display it.
      pack();
      setVisible(true);
   }

   /**
      The buildButtonPanel method builds the button panel.
   */

   private void buildButtonPanel()
   {
      // Create a panel for the buttons.
      buttonPanel = new JPanel();

      // Create the buttons.
      calcButton = new JButton("Calculate");
      exitButton = new JButton("Exit");

      // Register the action listeners.
      calcButton.addActionListener(new CalcButtonListener());
      exitButton.addActionListener(new ExitButtonListener());

      // Add the buttons to the button panel.
      buttonPanel.add(calcButton);
      buttonPanel.add(exitButton);
   }

   /**
      Private inner class that handles the event when
      the user clicks the Calculate button.
   */

   private class CalcButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         // Variables to hold the subtotal, tax, and total
         double subtotal, tax, total;

         // Calculate the subtotal.
         subtotal = bagels.getBagelCost() +
                    toppings.getToppingCost() +
                    coffee.getCoffeeCost();

         // Calculate the sales tax.
         tax = subtotal * TAX_RATE;

         // Calculate the total.
         total = subtotal + tax;

         // Create a DecimalFormat object to format output.
         DecimalFormat dollar = new DecimalFormat("0.00");

         // Display the charges.
         JOptionPane.showMessageDialog(null, "Subtotal: $" +
                       dollar.format(subtotal) + "\n" +
                       "Tax: $" + dollar.format(tax) + "\n" +
                       "Total: $" + dollar.format(total));
      }
   }

   /**
      Private inner class that handles the event when
      the user clicks the Exit button.
   */

   private class ExitButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         System.exit(0);
      }
   }

   /**
      main method
   */

```

```
    public static void main(String[] args)
    {
       new OrderCalculatorGUI();
    }
}
```

When the application runs, the window shown in Figure 12-36 appears. Figure 12-37 shows the `JOptionPane` dialog box that is displayed when the user selects a wheat bagel with butter, cream cheese, and decaf coffee.

Figure 12-36 The Order Calculator window

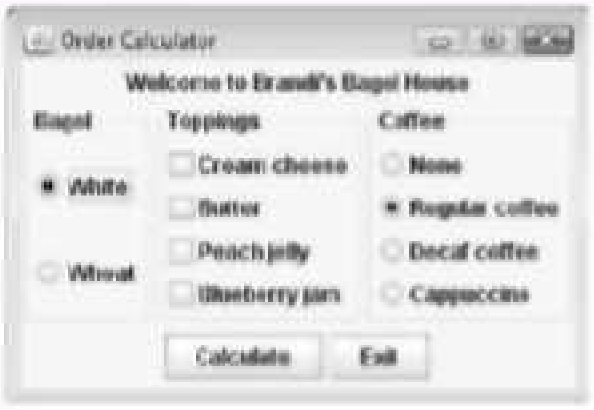
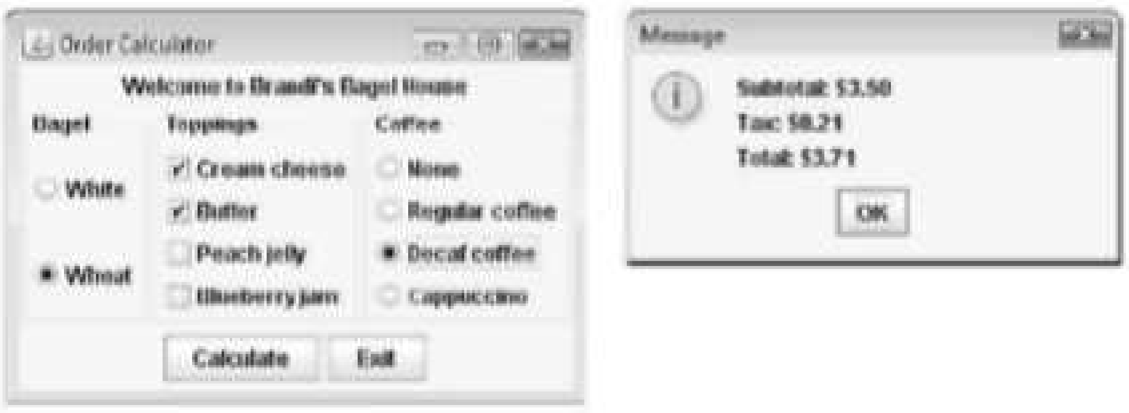


Figure 12-37 The subtotal, tax, and total displayed



```
Subtotal: $3.50
Tax: $0.21
Total: $3.71
```

12.7 Splash Screens

CONCEPT: A splash screen is a graphic image that is displayed while an application loads into memory and starts up.

Most major applications display a splash screen, which is a graphic image that is displayed while the application is loading into memory. Splash screens usually show company logos and keep the user's attention while the application starts up. Splash screens are particularly important for large applications that take a long time to load, because they assure the user that the program is not malfunctioning.

Beginning with Java 6, you can display splash screens with your Java applications. First, you have to use a graphics program to create the image that you want to display. Java supports splash screens in the GIF, PNG, or JPEG formats. (If you are using Windows, you can create images with Microsoft Paint, which supports all of these formats.)

To display the splash screen you use the `java` command in the following way when you run the application:

```
java -splash:GraphicFileName ClassFileName
```

`GraphicFileName` is the name of the file that contains the graphic image, and *`ClassFileName`* is the name of the *.class* file that you are running. For example, in the same source code folder as the *Brandi's Bagel House* application, you will find a file named *BrandiLogo.jpg*. This image, which is shown in Figure 12-38, is a logo for the *Brandi's Bagel House* application. To display the splash screen when the application starts, you would use the following command:

```
java splash:BrandiLogo.jpg Bagel
```

When you run this command, the graphic file will immediately be displayed in the center of the screen. It will remain displayed until the application's window appears.

Figure 12-38 Splash screen for the *Brandi's Bagel House* application

12.8 Using Console Output to Debug a GUI Application

CONCEPT: When debugging a GUI application, you can use `System.out.println` to send diagnostic messages to the console.

When an application is not performing correctly, programmers sometimes write statements that display *diagnostic messages* into the application. For example, if an application is not giving the correct result for a calculation, diagnostic messages can be displayed at various points in the program's execution showing the values of all the variables used in the calculation. If the trouble is caused by a variable that has not been properly initialized, or that has not been assigned the correct value, the diagnostic messages reveal this problem. This helps the programmer to see what is going on "under the hood" while an application is running.

The `System.out.println` method can be a valuable tool for displaying diagnostic messages in a GUI application. Because the `System.out.println` method sends its output to the console, diagnostic messages can be displayed without interfering with the application's GUI windows.

Code Listing 12-21 shows an example. This is a modified version of the `KiloConverter` class, discussed earlier in this chapter. Inside the `actionPerformed` method, which is in the `CalcButtonListener` inner class, calls to the `System.out.println` method have been written. The new code, which appears in lines 99 through 104 and 113 through 115, is shown in bold. These new statements display the value that the application has retrieved from the text field, and is working within its calculation. (This file is stored in the source code folder *Chapter 12\KiloConverter Phase 3*.)

Code Listing 12-21 **(KiloConverter.java)**

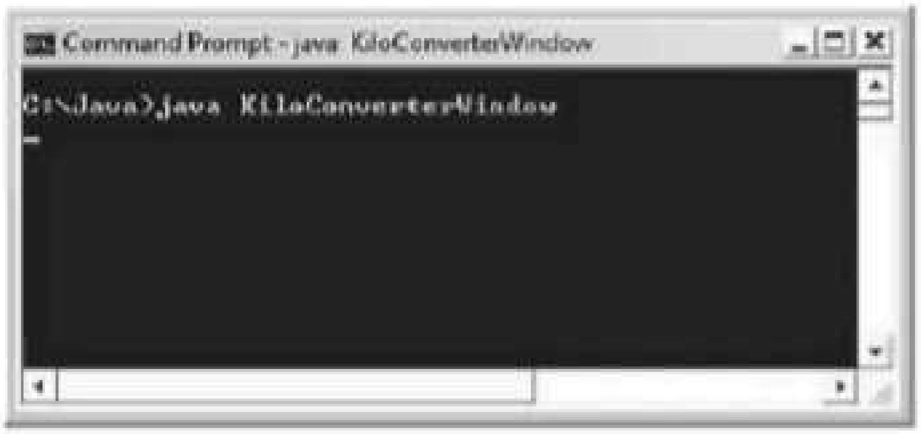
```
 1 import javax.swing.*;     // Needed for Swing classes
 2 import java.awt.event.*;  // Needed for ActionListener Interface
 3
 4 /**
 5    The KiloConverter class displays a JFrame that
 6    lets the user enter a distance in kilometers. When
 7    the Calculate button is clicked, a dialog box is
 8    displayed with the distance converted to miles.
 9 */
10
11 public class KiloConverter extends JFrame
12 {
13    private JPanel panel;               // To reference a panel
14    private JLabel messageLabel;        // To reference a label
15    private JTextField kiloTextField; // To reference a text field
16    private JButton calcButton;         // To reference a button
17    private final int WINDOW_WIDTH = 310;  // Window width
18    private final int WINDOW_HEIGHT = 100; // Window height
19
20    /**
21       Constructor
22    */
23
24    public KiloConverter()
25    {
26       // Set the window title.
27       setTitle("Kilometer Converter");
28
29       // Set the size of the window.
30       setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
31
32       // Specify what happens when the close button is clicked.
33       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```

      // Build the panel and add it to the frame.
      buildPanel();

      // Add the panel to the frame's content pane.
      add(panel);

      // Display the window.
      setVisible(true);
   }

   /**
      The buildPanel method adds a label, a text field,
      and a button to a panel.
   */

   private void buildPanel()
   {
      // Create a label to display instructions.
      messageLabel = new JLabel("Enter a distance " +
                                "in kilometers");

      // Create a text field 10 characters wide.
      kiloTextField = new JTextField(10);

      // Create a button with the caption "Calculate".
      calcButton = new JButton("Calculate");

      // Add an action listener to the button.
      calcButton.addActionListener(new CalcButtonListener());

      // Create a JPanel object and let the panel
      // field reference it.
      panel = new JPanel();

      // Add the label, text field, and button
      // components to the panel.
      panel.add(messageLabel);
      panel.add(kiloTextField);
      panel.add(calcButton);
   }

   /**
      CalcButtonListener is an action listener class for
      the Calculate button.
   */

   private class CalcButtonListener implements ActionListener
   {
      /**
         The actionPerformed method executes when the user
         clicks on the Calculate button.
         @param e The event object.
      */

      public void actionPerformed(ActionEvent e)
      {
         final double CONVERSION = 0.6214;
         String input;  // To hold the user's input
         double miles;  // The number of miles

         // Get the text entered by the user into the
         // text field.
         input = kiloTextField.getText();

         // For debugging, display the text entered, and
         // its value converted to a double.
         System.out.println("Reading " + input +
                            " from the text field.");
         System.out.println("Converted value: " +
                            Double.parseDouble(input));

         // Convert the input to miles.
         miles = Double.parseDouble(input) * CONVERSION;

         // Display the result.
         JOptionPane.showMessageDialog(null, input +
                 " kilometers is " + miles + " miles.");

         // For debugging, display a message indicating
         // the application is ready for more input.
         System.out.println("Ready for the next input.");
      }
   } // End of CalcButtonListener class

   /**
      The main method creates an instance of the
      KiloConverter class, which displays
      its window on the screen.
   */

   public static void main(String[] args)
   {
      new KiloConverter();
   }
}
```

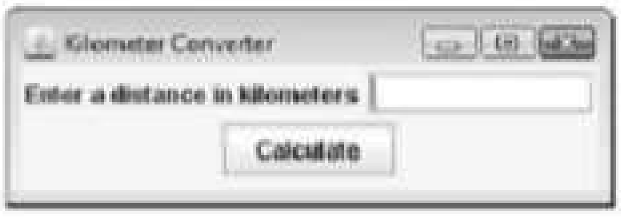
Let's take a closer look. In lines 101 and 102 a message is displayed to the console showing the value that was read from the text field. In lines 103 and 104 another message is displayed showing the value after it is converted to a `double`. Then, in line 115, a message is displayed indicating that the application is ready for its next input. Figure 12-39 shows an example session with the application on a computer running Microsoft Windows. Both the console window and the application windows are shown.

Figure 12-39 Messages displayed to the console during the application's execution

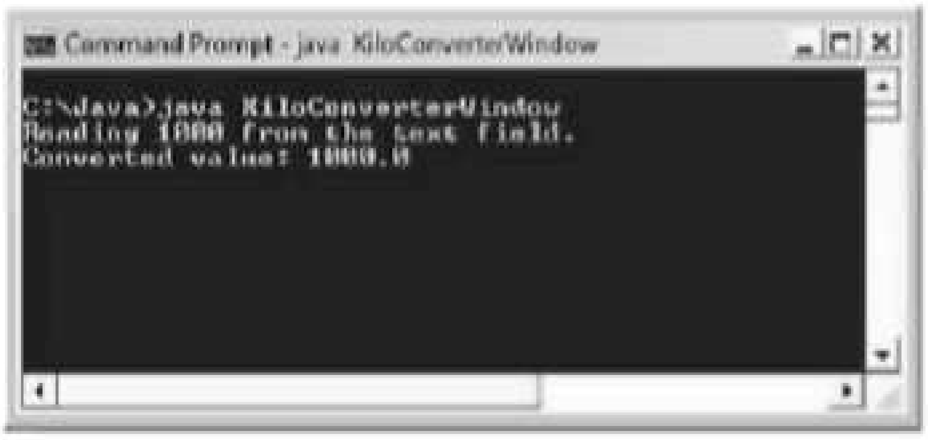
1. A command is typed in the console window to execute the application. The application's window appears.



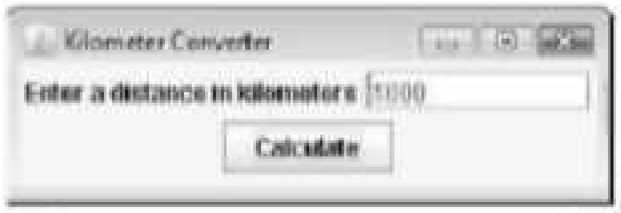
```
C:\Java>java KiloConverterWindow
_
```



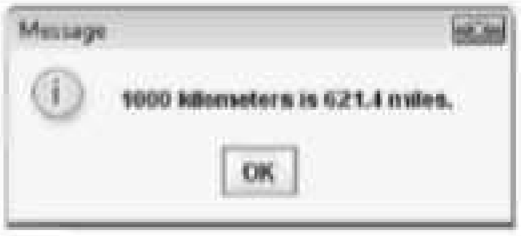
2. The user types a value into the text field and clicks the Calculate button. Debugging messages appear in the console window, and a message dialog appears showing the value converted to miles.



```
C:\Java>java KiloConverterWindow
Reading 1000 from the text field.
Converted value: 1000.0
```

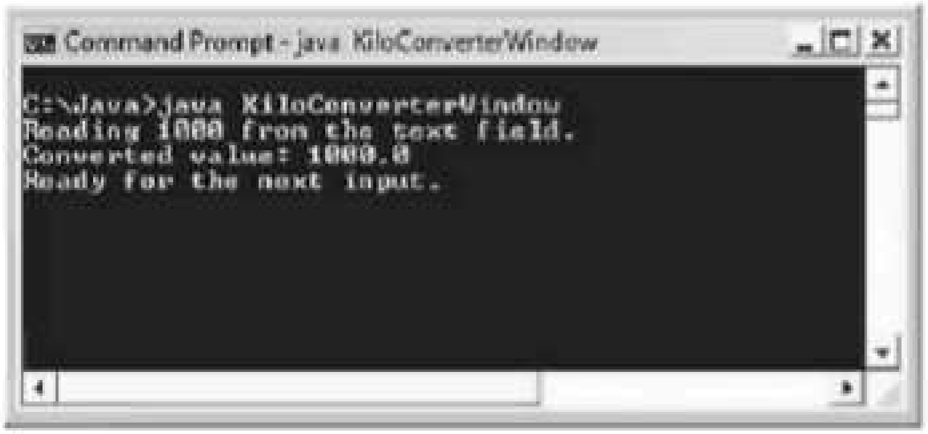


1000

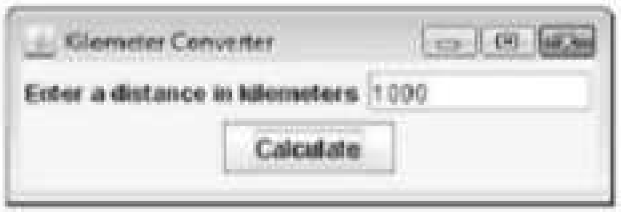


1000 kilometers is 621.4 miles.

3. The user dismisses the dialog box and a message is displayed in the console window indicating that the application is ready for the next input.



```
C:\Java>java KiloConverterWindow
Reading 1000 from the text field.
Converted value: 1000.0
Ready for the next input.
```



1000

The messages that are displayed to the console are meant for only the programmer to see, while he or she is debugging the application. Once the programmer is satisfied that the application is running correctly, the calls to `System.out.println` can be taken out.

12.9 Common Errors to Avoid

- **Misspelling `javax.swing` in an `import` statement.** Don't forget the letter x that appears after `java` in this import statement.
- **Forgetting to specify the action taken when the user clicks on a `JFrame`'s close button.** By default, a window is hidden from view when the close button is clicked, but the application is not terminated. If you wish to exit the application when a `JFrame's` close button is clicked, you must call the `setDefaultCloseOperation` method and pass `JFrame.EXIT_ON_CLOSE` as the argument.
- **Forgetting to write an event listener for each event you wish an application to respond to.** In order to respond to an event, you must write an event listener that implements the proper type of interface, registered to the component that generates the event.
- **Forgetting to register an event listener.** Even if you write an event listener, it will not execute unless it has been registered with the correct component.
- **When writing an event listener method that is required by an interface, not using the method header specified by the interface.** The header of an `actionPerformed` method must match that specified by the `ActionListener` interface. Also, the header of an `itemStateChanged` method must match that specified by the `ItemListener` method.
- **Placing components directly into the regions of a container governed by a `BorderLayout` manager when you do not want the components resized or you want to add more than one component per region.** If you do not want the components that you place in a `BorderLayout` region to be resized, place them in a `JPanel` component and then add the `JPanel` component to the region.
- **Placing components directly into the cells of a container governed by a `GridLayout` manager when you do not want the components resized or you want to add more than one component per cell.** If you do not want the components that you place in a `GridLayout` cell to be resized, place them in a `JPanel` component, and then add the `JPanel` component to the cell.
- **Forgetting to add `JRadioButton` components to a `ButtonGroup` object.** A mutually exclusive relationship is created between radio buttons only when they are added to a `ButtonGroup` object.

Review Questions and Exercises

Multiple Choice and True/False

1. With Swing, you use this class to create a frame.
   a. `Frame`
   b. `SwingFrame`
   c. `JFrame`
   d. `JavaFrame`
2. This is the part of a `JFrame` object that holds the components that have been added to the `JFrame` object.
   a. content pane
   b. viewing area
   c. component array
   d. object collection

3. This is a `JPanel` object's default layout manager.
   a. `BorderLayout`
   b. `GridLayout`
   c. `FlowLayout`
   d. None
4. This is the default layout manager for a `JFrame` object's content pane.
   a. `BorderLayout`
   b. `GridLayout`
   c. `FlowLayout`
   d. None
5. If a container is governed by a `BorderLayout` manager and you add a component to it, but you do not pass the second argument specifying the region, this is the region in which the component will be added.
   a. north
   b. south
   c. east
   d. center
6. Components in this/these regions of a `BorderLayout` manager are resized horizontally so they fill up the entire region.
   a. north and south
   b. east and west
   c. center only
   d. north, south, east, and west
7. Components in this/these regions of a `BorderLayout` manager are resized vertically so they fill up the entire region.
   a. north and south
   b. east and west
   c. center only
   d. north, south, east, and west
8. Components in this/these regions of a `BorderLayout` manager are resized both horizontally and vertically so they fill up the entire region.
   a. north and south
   b. east and west
   c. center only
   d. north, south, east, and west
9. This is the default alignment of a `FlowLayout` manager.
   a. left
   b. center
   c. right
   d. no alignment
10. Adding radio button components to this type of object creates a mutually exclusive relationship between them.
    a. `MutualExclude`
    b. `RadioGroup`
    c. `LogicalGroup`
    d. `ButtonGroup`

11. You use this class to create `Border` objects.
    a. `BorderFactory`
    b. `BorderMaker`
    c. `BorderCreator`
    d. `BorderSource`
12. **True or False:** A panel cannot be displayed by itself.
13. **True or False:** You can place multiple components inside a `GridLayout` cell.
14. **True or False:** You can place multiple components inside a `BorderLayout` region.
15. **True or False:** You can place multiple components inside a container governed by a `FlowLayout` manager.
16. **True or False:** You can place a panel inside a region governed by a `BorderLayout` manager.
17. **True or False:** A component placed in a `GridLayout` manager's cell will not be resized to fill up any extra space in the cell.
18. **True or False:** You normally add `JCheckBox` components to a `ButtonGroup` object.
19. **True or False:** A mutually exclusive relationship is automatically created among all `JRadioButton` components in the same container.
20. **True or False:** You can write a class that extends the `JPanel` class.

Find the Error

1. The following statement is in a class that uses Swing components:

```
import java.swing.*;
```

2. The following is an inner class that will be registered as an action listener for a `JButton` component:

```
private class ButtonListener implements ActionListener
{
   public void actionPerformed()
   {
      // Code appears here.
   }
}
```

3. The intention of the following statement is to give the `panel` object a `GridLayout` manager with 10 columns and 5 rows:

```
panel.setLayout(new GridLayout(10, 5));
```

4. The `panel` variable references a `JPanel` governed by a `BorderLayout` manager. The following statement attempts to add the button component to the north region of panel:

```
panel.add(button, NORTH);
```

5. The `panel` variable references a `JPanel` object. The intention of the following statement is to create a titled border around `panel`:

```
panel.setBorder(new BorderFactory("Choices"));
```

Algorithm Workbench

1. The variable `myWindow` references a `JFrame` object. Write a statement that sets the size of the object to 500 pixels wide and 250 pixels high.
2. The variable `myWindow` references a `JFrame` object. Write a statement that causes the application to end when the user clicks on the `JFrame` object's close button.
3. The variable `myWindow` references a `JFrame` object. Write a statement that displays the object's window on the screen.
4. The variable `myButton` references a `JButton` object. Write the code to set the object's background color to white and foreground color to red.
5. Assume that a class inherits from the `JFrame` class. Write code that can appear in the class constructor, which gives the content pane a `FlowLayout` manager. Components added to the content pane should be aligned with the left edge of each row.
6. Assume that a class inherits from the `JFrame` class. Write code that can appear in the class constructor, which gives the content pane a `GridLayout` manager with five rows and 10 columns.
7. Assume that the variable `panel` references a `JPanel` object that uses a `BorderLayout` manager. In addition, the variable `button` references a `JButton` object. Write code that adds the `button` object to the `panel` object's west region.
8. Write code that creates three radio buttons with the text "Option 1", "Option 2", and "Option 3". The radio button that displays the text "Option 1" should be initially selected. Make sure these components are grouped so that a mutually exclusive relationship exists among them.
9. Assume that panel references a `JPanel` object. Write code that creates a two pixel thick blue line border around it.

Short Answer

1. If you do not change the default close operation, what happens when the user clicks on the close button on a `JFrame` object?
2. Why is it sometimes necessary to place a component inside a panel and then place the panel inside a container governed by a `BorderLayout` manager?
3. In what type of situation would you present a group of items to the user with radio buttons? With check boxes?
4. How can you create a specialized panel component that can be used to hold other components and their related code?

Programming Challenges

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

1. Retail Price Calculator

Create a GUI application where the user enters the wholesale cost of an item and its markup percentage into text fields. (For example, if an item's wholesale cost is $5 and its markup percentage is 100 percent, then its retail price is $10.) The application should have a button that displays the item's retail price when clicked.

2. Monthly Sales Tax

VideoNote The Monthly Sales Tax Problem

A retail company must file a monthly sales tax report listing the total sales for the month, and the amount of state and county sales tax collected. The state sales tax rate is 4 percent and the county sales tax rate is 2 percent. Create a GUI application that allows the user to enter the total sales for the month into a text field. From this figure, the application should calculate and display the following:

- The amount of county sales tax
- The amount of state sales tax
- The total sales tax (county plus state)

In the application's code, represent the county tax rate (0.02) and the state tax rate (0.04) as named constants.

3. Property Tax

A county collects property taxes on the assessment value of property, which is 60 percent of the property's actual value. If an acre of land is valued at $10,000, its assessment value is $6,000. The property tax is then $0.64 for each $100 of the assessment value. The tax for the acre assessed at $6,000 will be $38.40. Create a GUI application that displays the assessment value and property tax when a user enters the actual value of a property.

4. Travel Expenses

Create a GUI application that calculates and displays the total travel expenses of a business person on a trip. Here is the information that the user must provide:

- Number of days on the trip
- Amount of airfare, if any
- Amount of car rental fees, if any
- Number of miles driven, if a private vehicle was used
- Amount of parking fees, if any
- Amount of taxi charges, if any
- Conference or seminar registration fees, if any
- Lodging charges, per night

The company reimburses travel expenses according to the following policy:

- $37 per day for meals
- Parking fees, up to $10.00 per day
- Taxi charges up to $20.00 per day
- Lodging charges up to $95.00 per day
- If a private vehicle is used, $0.27 per mile driven

The application should calculate and display the following:

- Total expenses incurred by the business person
- The total allowable expenses for the trip
- The excess that must be paid by the business person, if any
- The amount saved by the business person if the expenses are under the total allowed

5. Theater Revenue

A movie theater only keeps a percentage of the revenue earned from ticket sales. The remainder goes to the movie company. Create a GUI application that allows the user to enter the following data into text fields:

- Price per adult ticket
- Number of adult tickets sold
- Price per child ticket
- Number of child tickets sold

The application should calculate and display the following data for one night's box office business at a theater:

- **Gross revenue for adult tickets sold.** This is the amount of money taken in for all adult tickets sold.
- **Net revenue for adult tickets sold.** This is the amount of money from adult ticket sales left over after the payment to the movie company has been deducted.
- **Gross revenue for child tickets sold.** This is the amount of money taken in for all child tickets sold.
- **Net revenue for child tickets sold.** This is the amount of money from child ticket sales left over after the payment to the movie company has been deducted.
- **Total gross revenue.** This is the sum of gross revenue for adult and child tickets sold.
- **Total net revenue.** This is the sum of net revenue for adult and child tickets sold.

Assume the theater keeps 20 percent of its box office receipts. Use a constant in your code to represent this percentage.

6. Joe's Automotive

Joe's Automotive performs the following routine maintenance services:

- Oil change—$26.00
- Lube job—$18.00
- Radiator flush—$30.00
- Transmission flush—$80.00
- Inspection—$15.00
- Muffler replacement—$100.00
- Tire rotation—$20.00

Joe also performs other nonroutine services and charges for parts and for labor ($20 per hour). Create a GUI application that displays the total for a customer's visit to Joe's.

7. Long Distance Calls

A long-distance provider charges the following rates for telephone calls:

Rate Category	Rate per Minute
Daytime (6:00 A.M. through 5:59 P.M.)	$0.07
Evening (6:00 P.M. through 11:59 P.M.)	$0.12
Off-Peak (12:00 A.M. through 5:59 A.M.)	$0.05

Create a GUI application that allows the user to select a rate category (from a set of radio buttons), and enter the number of minutes of the call into a text field. A dialog box should display the charge for the call.

8. Latin Translator

Look at the following list of Latin words and their meanings.

Latin	English
sinister	left
dexter	right
medium	center

Write a GUI application that translates the Latin words to English. The window should have three buttons, one for each Latin word. When the user clicks a button, the program displays the English translation in a label.

9. MPG Calculator

Write a GUI application that calculates a car's gas mileage. The application should let the user enter the number of gallons of gas the car holds, and the number of miles it can be driven on a full tank. When a *Calculate MPG* button is clicked, the application should display the number of miles that the car may be driven per gallon of gas. Use the following formula to calculate MPG:

$$MPG = \frac{Miles}{Gallons}$$

10. Celsius to Fahrenheit

Write a GUI application that converts Celsius temperatures to Fahrenheit temperatures. The user should be able to enter a Celsius temperature, click a button, and then see the equivalent Fahrenheit temperature. Use the following formula to make the conversion:

$$F = \frac{9}{5}C + 32$$

F is the Fahrenheit temperature and C is the Celsius temperature.

CHAPTER 13 Advanced GUI Applications

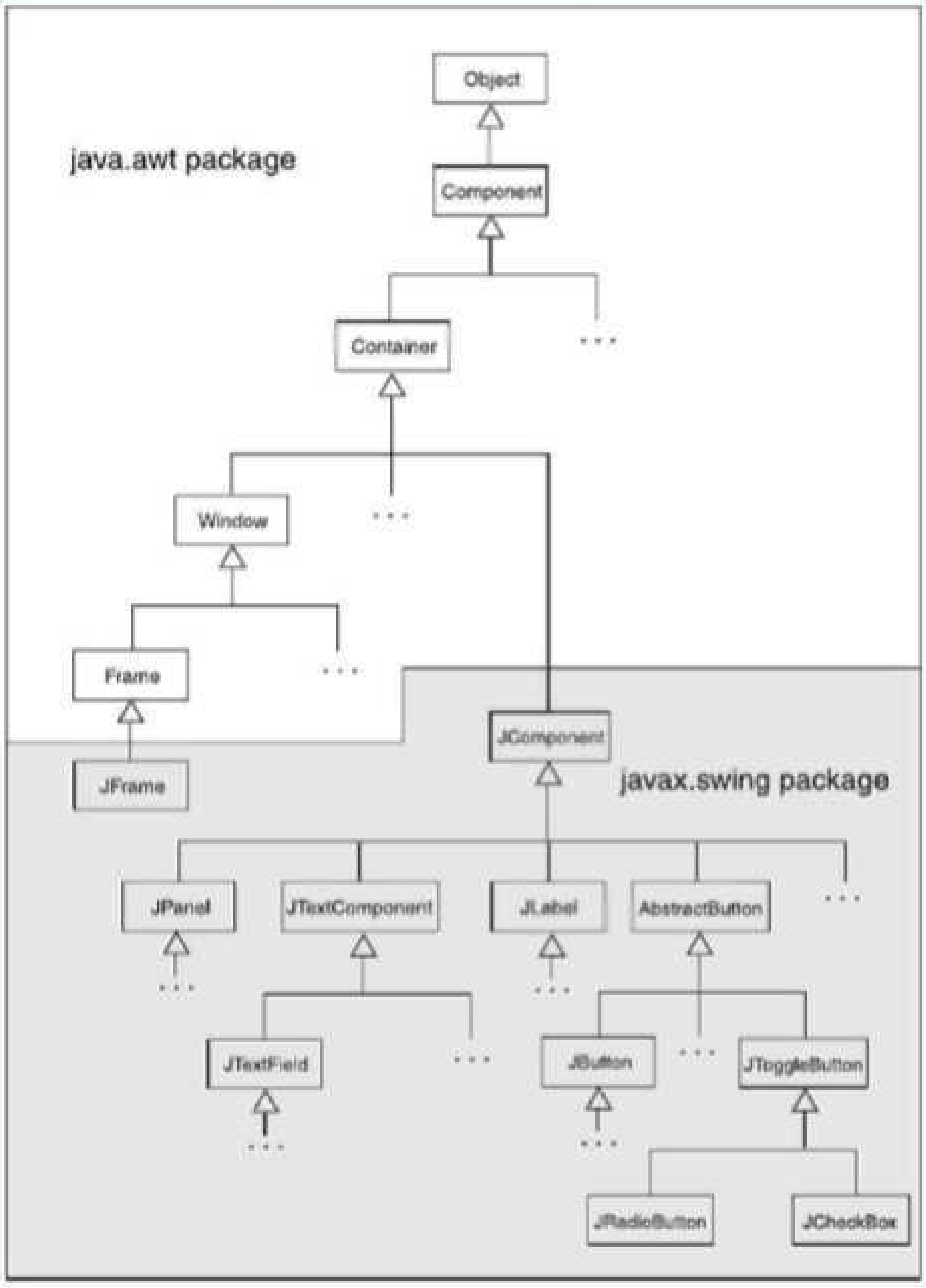
TOPICS

13.1 The Swing and AWT Class Hierarchy

Now that you have used some of the fundamental GUI components, let's look at how they fit into the class hierarchy. Figure 13-1 shows the parts of the Swing and AWT class hierarchy that contain the `JFrame`, `JPanel`, `JLabel`, `JTextField`, `JButton`, `JRadioButton`, and `JCheckBox` classes. Because of the inheritance relationships that exist, there are many other classes in the figure as well.

The classes that are in the unshaded top part of the figure are AWT classes and are in the `java.awt` package. The classes that are in the shaded bottom part of the figure are Swing classes and are in the `javax.swing` package. Notice that all of the components we have dealt with ultimately inherit from the `Component` class.

Figure 13-1 Part of the Swing and AWT class hierarchy



13.2 Read-Only Text Fields

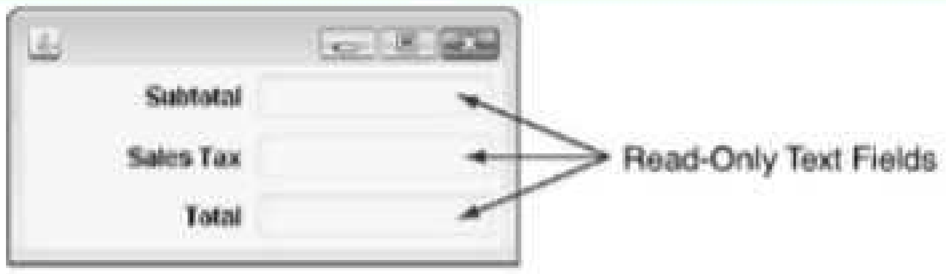
CONCEPT: A read-only text field displays text that can be changed by code in the application, but cannot be edited by the user.

A read-only text field is not a new component, but a different way to use the `JTextField` component. The `JTextField` component has a method named `setEditable`, which has the following general format:

```
setEditable(boolean editable)
```

You pass a `boolean` argument to this method. By default a text field is editable, which means that the user can enter data into it. If you call the `setEditable` method and pass `false` as the argument, then the text field becomes read-only. This means it is not editable by the user. Figure 13-2 shows a window that has three read-only text fields.

Figure 13-2 A window with three read-only text fields



The following code could be used to create the read-only text fields shown in the figure:

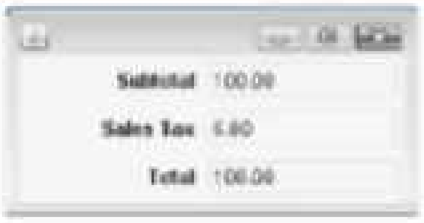
```
// Create a read-only text field for the subtotal.
JTextField subtotalField = new JTextField(10);
subtotalField.setEditable(false);

// Create a read-only text field for the sales tax.
JTextField taxField = new JTextField(10);
taxField.setEditable(false);

// Create a read-only text field for the total.
JTextField totalField = new JTextField(10);
totalField.setEditable(false);
```

A read-only text field looks like a label with a border drawn around it. You can use the `setText` method to display data inside it. Here is an example:

```
subtotalField.setText("100.00");
taxField.setText("6.00");
totalField.setText("106.00");
```

This code causes the text fields to appear as shown in Figure 13-3.

Figure 13-3 Read-only text fields with data displayed



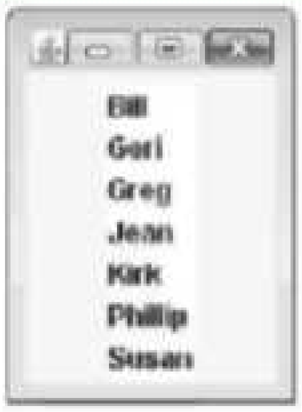
13.3 Lists

CONCEPT: A list component displays a list of items and allows the user to select an item from the list.

VideoNote The `JList` Component

A list is a component that displays a list of items and also allows the user to select one or more items from the list. Java provides the `JList` component for creating lists. Figure 13-4 shows an example. The `JList` component in the figure shows a list of names. At runtime, the user may select an item in the list, which causes the item to appear highlighted. In the figure, the first name is selected.

Figure 13-4 A `JList` component



When you create an instance of the `JList` class, you pass an array of objects to the constructor. Here is the general format of the constructor call:

```
JList (Object[] array)
```

The `JList` component uses the array to create the list of items. In this text we always pass an array of `String` objects to the `JList` constructor. For example, the list component shown in Figure 13-4 could be created with the following code:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
JList nameList = new JList(names);
```

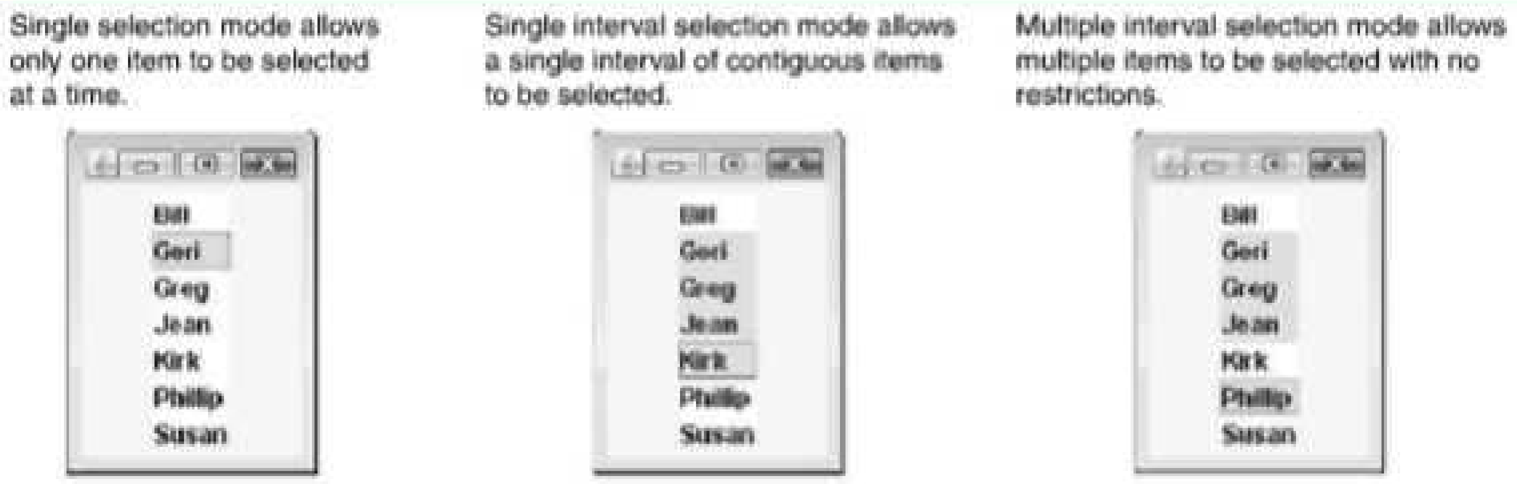
Selection Modes

The `JList` component can operate in any of the following selection modes:

- Single Selection Mode. In this mode only one item can be selected at a time. When an item is selected, any other item that is currently selected is deselected.
- Single Interval Selection Mode. In this mode multiple items can be selected, but they must be in a single interval. An interval is a set of contiguous items.
- Multiple Interval Selection Mode. In this mode multiple items may be selected with no restrictions. This is the default selection mode.

Figure 13-5 shows an example of a list in each type of selection mode.

Figure 13-5 Selection modes


Single selection mode allows only one item to be selected at a time.
Single interval selection mode allows a single interval of contiguous items to be selected.
Multiple interval selection mode allows multiple items to be selected with no restrictions.
Bill
Geri
Greg
Jean
Kirk
Phillip
Susan

The default mode is multiple interval selection. To keep our applications simple, we will use single selection mode for now. You change a `JList` component's selection mode with the `setSelectionMode` method. The method accepts an `int` argument that determines the selection mode.

The `ListSelectionModel` class, which is in the `javax.swing` package, provides the following constants that you can use as arguments to the `setSelectionMode` method:

- `ListSelectionModel.SINGLE_SELECTION`
- `ListSelectionModel.SINGLE_INTERVAL_SELECTION`
- `ListSelectionModel.MULTIPLE_INTERVAL_SELECTION`

Assuming that `nameList` references a `JList` component, the following statement sets the component to single selection mode:

```
nameList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

Responding to List Events

When an item in a `JList` object is selected it generates a list selection event. You handle list selection events with a list selection listener class, which must meet the following requirements:

- It must implement the `ListSelectionListener` interface.
- It must have a method named `valueChanged`. This method must take an argument of the `ListSelectionEvent` type.

NOTE: The `ListSelectionListener` interface is in the `javax.swing.event` package, so you must have an `import` statement for that package in your source code.

Once you have written a list selection listener class, you create an object of that class and then pass it as an argument to the `JList` component's `addListSelectionListener` method. When the `JList` component generates an event, it automatically executes the `valueChanged` method of the list selection listener object, passing the event object as an argument. You will see an example in a moment.

Retrieving the Selected Item

You may use either the getSelectedValue method or the getSelectedIndex method to determine which item in a list is currently selected. The getSelectedValue method returns a reference to the item that is currently selected. For example, assume that nameList references the JList component shown earlier in Figure 13-4. The following code retrieves a reference to the name that is currently selected and assigns it to the selectedName variable:

```
String selectedName;
selectedName = (String) nameList.getSelectedValue();
```

Note that the return value of the getSelectedValue method is an Object reference. In this code we had to cast the return value to the String type in order to store it in the selectedName variable. If no item in the list is selected, the method returns null.

The getSelectedIndex method returns the index of the selected item, or −1 if no item is selected. Internally, the items that are stored in a list are numbered. Each item's number is called its index. The first item (which is the item stored at the top of the list) has the index 0, the second item has the index 1, and so forth. You can use the index of the selected item to retrieve the item from an array. For example, assume that the following code was used to build the nameList component shown in Figure 13-4:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
JList nameList = new JList(names);
```

Because the names array holds the values displayed in the namesList component, the following code could be used to determine the selected item:

```
int index;
String selectedName;
index = nameList.getSelectedIndex();
if (index != -1)
   selectedName = names[index];
```

The ListWindow class shown in Code Listing 13-1 demonstrates the concepts we have discussed so far. It uses a JList component with a list selection listener. When an item is selected from the list, it is displayed in a read-only text field. The main method creates an instance of the ListWindow class, which displays the window shown on the left in Figure 13-6. After the user selects October from the list, the window appears as that shown on the right in the figure.

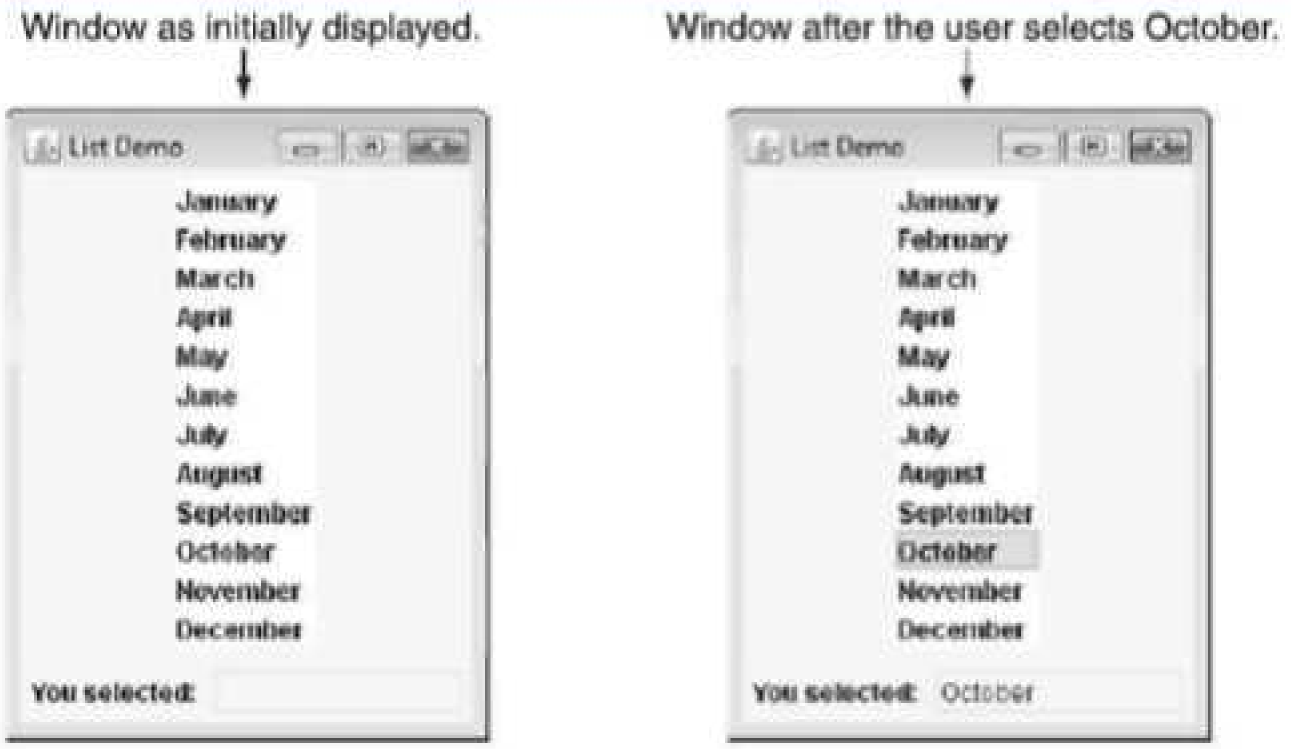
Code Listing 13-1 (ListWindow.java)

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;

/**
   This class demonstrates the List Component.
*/
```

```

public class ListWindow extends JFrame
{
   private JPanel monthPanel;           // To hold components
   private JPanel selectedMonthPanel;   // To hold components
   private JList monthList;             // The months
   private JTextField selectedMonth;    // The selected month
   private JLabel label;                // A message

   // The following array holds the values that will
   // be displayed in the monthList list component.
   private String[] months = { "January", "February",
                               "March", "April", "May", "June", "July",
                               "August", "September", "October", "November",
                               "December" };

   /**
      Constructor
   */

   public ListWindow()
   {
      // Set the title.
      setTitle("List Demo");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Add a BorderLayout manager.
      setLayout(new BorderLayout());

      // Build the month and selectedMonth panels.
      buildMonthPanel();
      buildSelectedMonthPanel();

      // Add the panels to the content pane.
      add(monthPanel, BorderLayout.CENTER);
      add(selectedMonthPanel, BorderLayout.SOUTH);

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildMonthPanel method adds a list containing
      the names of the months to a panel.
   */

   private void buildMonthPanel()
   {
      // Create a panel to hold the list.
      monthPanel = new JPanel();

      // Create the list.
      monthList = new JList(months);

      // Set the selection mode to single selection.
      monthList.setSelectionMode(
                 ListSelectionModel.SINGLE_SELECTION);

      // Register the list selection listener.
      monthList.addListSelectionListener(
                                  new ListListener());

      // Add the list to the panel.
      monthPanel.add(monthList);
   }

   /**
      The buildSelectedMonthPanel method adds an
      uneditable text field to a panel.
   */

   private void buildSelectedMonthPanel()
   {
      // Create a panel to hold the text field.
      selectedMonthPanel = new JPanel();

      // Create the label.
      label = new JLabel("You selected: ");

      // Create the text field.
      selectedMonth = new JTextField(10);

      // Make the text field uneditable.
      selectedMonth.setEditable(false);

      // Add the label and text field to the panel.
      selectedMonthPanel.add(label);
      selectedMonthPanel.add(selectedMonth);
   }

   /**
      Private inner class that handles the event when
      the user selects an item from the list.
```

```
    */

    private class ListListener
                    implements ListSelectionListener
    {
       public void valueChanged(ListSelectionEvent e)
       {
          // Get the selected month.
          String selection =
                (String) monthList.getSelectedValue();

          // Put the selected month in the text field.
          selectedMonth.setText(selection);
       }
    }

    /**
       The main method creates an instance of the
       ListWindow class which causes it to display
       its window.
    */

      public static void main(String[] args)
      {
          new ListWindow();
      }
}
```

Figure 13-6 Window displayed by the `ListWindow` class



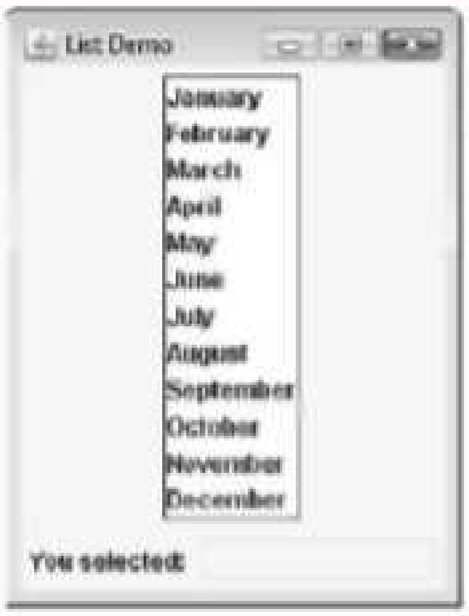
Placing a Border around a List

As with other components, you can use the **setBorder** method, which was discussed in Chapter 12, to draw a border around a **JList**. For example the following statement can be used to draw a black 1-pixel thick line border around the **monthList** component:

```
monthList.setBorder(BorderFactory.createLineBorder(Color.BLACK, 1));
```

This code will cause the list to appear as shown in Figure 13-7.

Figure 13-7 List with a line border



Adding a Scroll Bar to a List

By default, a list component is large enough to display all of the items it contains. Sometimes a list component contains too many items to be displayed at once, however. Most GUI applications display a scroll bar on list components that contain a large number of items. The user simply uses the scroll bar to scroll through the list of items.

List components do not automatically display a scroll bar. To display a scroll bar on a list component, you must follow the following general steps:

1. Set the number of visible rows for the list component.
2. Create a scroll pane object and add the list component to it.
3. Add the scroll pane object to any other containers, such as panels.

Let's take a closer look at how these steps can be used to apply a scroll bar to the list component created in the following code:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
JList nameList = new JList(names);
```

First, we establish the size of the list component with the **JList** class's **setVisibleRowCount** method. The following statement sets the number of visible rows in the **nameList** component to three:

```
nameList.setVisibleRowCount(3);
```

This statement causes the `nameList` component to display only three items at a time.

Next, we create a scroll pane object and add the list component to it. A scroll pane object is a container that displays scroll bars on any component it contains. In Java we use the `JScrollPane` class to create a scroll pane object. We pass the object that we wish to add to the scroll pane as an argument to the `JScrollPane` constructor. The following statement demonstrates:

```
JScrollPane scrollPane = new JScrollPane(nameList);
```

This statement creates a `JScrollPane` object and adds the `nameList` component to it.

Next, we add the scroll pane object to any other containers that are necessary for our GUI. For example, the following code adds the scroll pane to a `JPanel`, which is then added to the `JFrame` object's content pane:

```
// Create a panel and add the scroll pane to it.
JPanel panel = new JPanel();
panel.add(scrollPane);

// Add the panel to this JFrame object's contentPane.
add(panel);
```

When the list component is displayed, it will appear as shown in Figure 13-8.

Although the list component displays only three items at a time, the user can scroll through all of the items it contains.

The `ListWindowWithScroll` class shown in Code Listing 13-2 is a modification of the `ListWindow` class. In this class, the `monthList` component shows only six items at a time, but displays a scroll bar. The code shown in bold is the new lines that are used to add the scroll bar to the list. The `main` method creates an instance of the class, which displays the window shown in Figure 13-9.

Figure 13-8 List component with a scroll bar

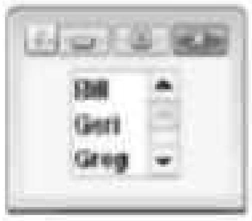
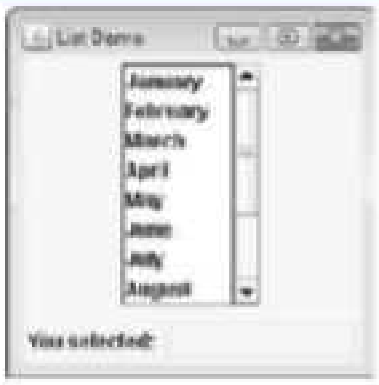


Figure 13-9 List component with scroll bars



Code Listing 13-2 (ListWindowWithScroll.java)

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;

/**
   This class demonstrates the List Component.
*/

public class ListWindowWithScroll extends JFrame
{
   private JPanel monthPanel;           // To hold components
   private JPanel selectedMonthPanel;   // To hold components
   private JList monthList;             // The months
   private JScrollPane scrollPane;      // A scroll pane
   private JTextField selectedMonth;    // The selected month
   private JLabel label;                // A message

   // The following array holds the values that will
   // be displayed in the monthList list component.
   private String[] months = { "January", "February",
             "March", "April", "May", "June", "July",
             "August", "September", "October", "November",
             "December" };

   /**
      Constructor
   */

   public ListWindowWithScroll()
   {
      // Set the title.
      setTitle("List Demo");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Add a BorderLayout manager.
      setLayout(new BorderLayout());

      // Build the month and selectedMonth panels.
      buildMonthPanel();
      buildSelectedMonthPanel();

      // Add the panels to the content pane.
      add(monthPanel, BorderLayout.CENTER);
      add(selectedMonthPanel, BorderLayout.SOUTH);

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildMonthPanel method adds a list containing
      the names of the months to a panel.
   */

   private void buildMonthPanel()
   {
      // Create a panel to hold the list.
      monthPanel = new JPanel();

      // Create the list.
      monthList = new JList(months);

      // Set the selection mode to single selection.
      monthList.setSelectionMode(
                ListSelectionModel.SINGLE_SELECTION);

      // Register the list selection listener.
      monthList.addListSelectionListener(
                                 new ListListener());

      // Set the number of visible rows to 6.
      monthList.setVisibleRowCount(6);

      // Add the list to a scroll pane.
      scrollPane = new JScrollPane(monthList);

      // Add the scroll pane to the panel.
      monthPanel.add(scrollPane);
   }

   /**
      The buildSelectedMonthPanel method adds an
      uneditable text field to a panel.
   */

   private void buildSelectedMonthPanel()
   {
      // Create a panel to hold the text field.
      selectedMonthPanel = new JPanel();

      // Create the label.
      label = new JLabel("You selected: ");

      // Create the text field.
      selectedMonth = new JTextField(10);

      // Make the text field uneditable.
      selectedMonth.setEditable(false);

      // Add the label and text field to the panel.
      selectedMonthPanel.add(label);
      selectedMonthPanel.add(selectedMonth);
   }

   /**
      Private inner class that handles the event when
      the user selects an item from the list.
   */

   private class ListListener
                       implements ListSelectionListener
   {
      public void valueChanged(ListSelectionEvent e)
      {
         // Get the selected month.
         String selection =
                (String) monthList.getSelectedValue();

         // Put the selected month in the text field.
         selectedMonth.setText(selection);
      }
   }

   /**
      The main method creates an instance of the
      ListWindowWithScroll class which causes it
      to display its window.
   */

     public static void main(String[] args)
     {
        new ListWindowWithScroll();
     }
}
```

NOTE: By default, when a `JList` component is added to a `JScrollPane` object, the scroll bar is only displayed when there are more items in the list than there are visible rows.

NOTE: When a `JList` component is added to a `JScrollPane` object, a border will automatically appear around the list.

Adding Items to an Existing `JList` Component

The `JList` class's `setListData` method allows you to store items in an existing `JList` component. Here is the method's general format:

```
void setListData(Object[] data)
```

The argument passed into `data` is an array of objects that will become the items displayed in the `JList` component. Any items that are currently displayed in the component will be replaced by the new items.

In addition to replacing the existing items in a list, you can use this method to add items to an empty list. You can create an empty list by passing no argument to the `JList` constructor. Here is an example:

```
JList nameList = new JList();
```

This statement creates an empty `JList` component referenced by the `nameList` variable. You can then add items to the list, as shown here:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
nameList.setListData(names);
```

Multiple Selection Lists

For simplicity, the previous examples used a `JList` component in single selection mode. Recall that the other two selection modes are single interval and multiple interval. Both of these modes allow the user to select multiple items. Let's take a closer look at each of these modes.

Single Interval Selection Mode

You put a `JList` component in single interval selection mode by passing the constant `ListSelectionModel.SINGLE_INTERVAL_SELECTION` to the component's `setSelectionMode` method. In single interval selection mode, single or multiple items can be selected. An interval is a set of contiguous items. (See Figure 13-5 to see an example of an interval.)

To select an interval of items, the user selects the first item in the interval by clicking on it, and then selects the last item in the interval by holding down the Shift key while clicking on it. All of the items that appear in the list from the first item through the last item are selected.

In single interval selection mode, the `getSelectedValue` method returns the first item in the selected interval. The `getSelectedIndex` method returns the index of the first item in the selected interval. To get the entire selected interval, use the `getSelectedValues` method. This method returns an array of objects. The array will hold the items in the selected interval. You can also use the `getSelectedIndices` method, which returns an array of `int` values. The values in the array will be the indices of all the selected items in the list.

Multiple Interval Selection Mode

You put a `JList` component in multiple interval selection mode by passing the constant `ListSelectionModel.MULTIPLE_INTERVAL_SELECTION` to the component's `setSelectionMode` method. In multiple interval selection mode, multiple items can be selected and the items do not have to be in the same interval. (See Figure 13-5 for an example.)

In multiple interval selection mode, the user can select single items or intervals. When the user holds down the Ctrl key while clicking on an item, it selects the item without deselecting any items that are currently selected. This allows the user to select multiple items that are not in an interval.

In multiple interval selection mode, the `getSelectedValue` method returns the first selected item. The `getSelectedIndex` method returns the index of the first selected item. The `getSelectedValues` method returns an array of objects containing the items that are selected. The `getSelectedIndices` method returns an `int` array containing the indices of all the selected items in the list.

The `MultipleIntervalSelection` class, shown in Code Listing 13-3, demonstrates a `JList` component used in multiple interval selection mode. The `main` method creates an instance of the class that displays the window shown on the left in Figure 13-10. When the user selects items from the top `JList` component and then clicks the Get Selections button, the selected items appear in the bottom `JList` component.

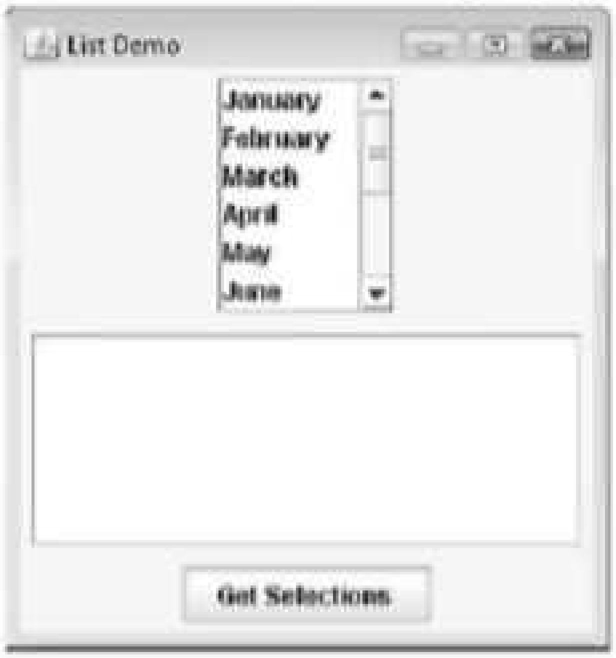
Code Listing 13-3 **(`MultipleIntervalSelection.java`)**

```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
   This class demonstrates the List Component in
   multiple interval selection mode.
*/

public class MultipleIntervalSelection extends JFrame
{
   private JPanel monthPanel;           // To hold components
   private JPanel selectedMonthPanel;   // To hold components
   private JPanel buttonPanel;          // To hold the button

   private JList monthList;             // To hold months
   private JList selectedMonthList;     // Selected months

   private JScrollPane scrollPanel;     // Scroll pane - first list
   private JScrollPane scrollPane2;     // Scroll pane - second list

   private JButton button;              // A button

   // The following array holds the values that
   // will be displayed in the monthList list component.
   private String[] months = { "January", "February",
             "March", "April", "May", "June", "July",
             "August", "September", "October", "November",
             "December" };

   /**
      Constructor
   */

   public MultipleIntervalSelection()
   {
      // Set the title.
      setTitle("List Demo");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Add a BorderLayout manager.
      setLayout(new BorderLayout());

      // Build the panels.
      buildMonthPanel();
      buildSelectedMonthsPanel();
      buildButtonPanel();

      // Add the panels to the content pane.
      add(monthPanel, BorderLayout.NORTH);
      add(selectedMonthPanel,BorderLayout.CENTER);
      add(buttonPanel, BorderLayout.SOUTH);

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildMonthPanel method adds a list containing the
      names of the months to a panel.
   */

   private void buildMonthPanel()
   {
      // Create a panel to hold the list.
      monthPanel = new JPanel();

      // Create the list.
      monthList = new JList(months);

      // Set the selection mode to multiple
      // interval selection.
      monthList.setSelectionMode(
        ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);

      // Set the number of visible rows to 6.
      monthList.setVisibleRowCount(6);

      // Add the list to a scroll pane.
      scrollPane1 = new JScrollPane(monthList);

      // Add the scroll pane to the panel.
      monthPanel.add(scrollPane1);
   }

   /**
      The buildSelectedMonthsPanel method adds a list
      to a panel. This will hold the selected months.
   */

   private void buildSelectedMonthsPanel()
   {
      // Create a panel to hold the list.
      selectedMonthPanel = new JPanel();

      // Create the list.
      selectedMonthList = new JList();

      // Set the number of visible rows to 6.
      selectedMonthList.setVisibleRowCount(6);

      // Add the list to a scroll pane.
      scrollPane2 =
               new JScrollPane(selectedMonthList);

      // Add the scroll pane to the panel.
      selectedMonthPanel.add(scrollPane2);
   }

   /**
      The buildButtonPanel method adds a
      button to a panel.
   */

   private void buildButtonPanel()
   {
      // Create a panel to hold the list.
      buttonPanel = new JPanel();

      // Create the button.
      button = new JButton("Get Selections");

      // Add an action listener to the button.
      button.addActionListener(new ButtonListener());

      // Add the button to the panel.
      buttonPanel.add(button);
   }

   /**
      Private inner class that handles the event when
      the user clicks the button.
   */

   private class ButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         // Get the selected values.
         Object[] selections =
                         monthList.getSelectedValues();

         // Store the selected items in selectedMonthList.
         selectedMonthList.setListData(selections);
      }
   }

   /**
      The main method creates an instance of the
      MultipleIntervalSelection class which causes it
      to display its window.
   */

   public static void main(String[] args)
```

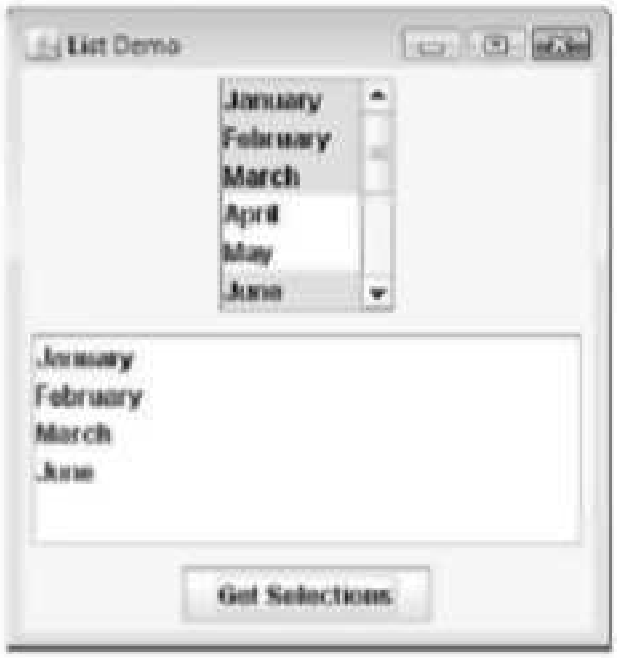
```
    {
        new MultipleIntervalSelection();
    }
}
```

Figure 13-10 The window displayed by the `MultipleIntervalSelection` class

This is the window as it is intially displayed.



This is the window after the user has selected some items from the top list and clicked the Get Selections button.



13.4 Combo Boxes

CONCEPT: A combo box allows the user to select an item from a drop-down list.

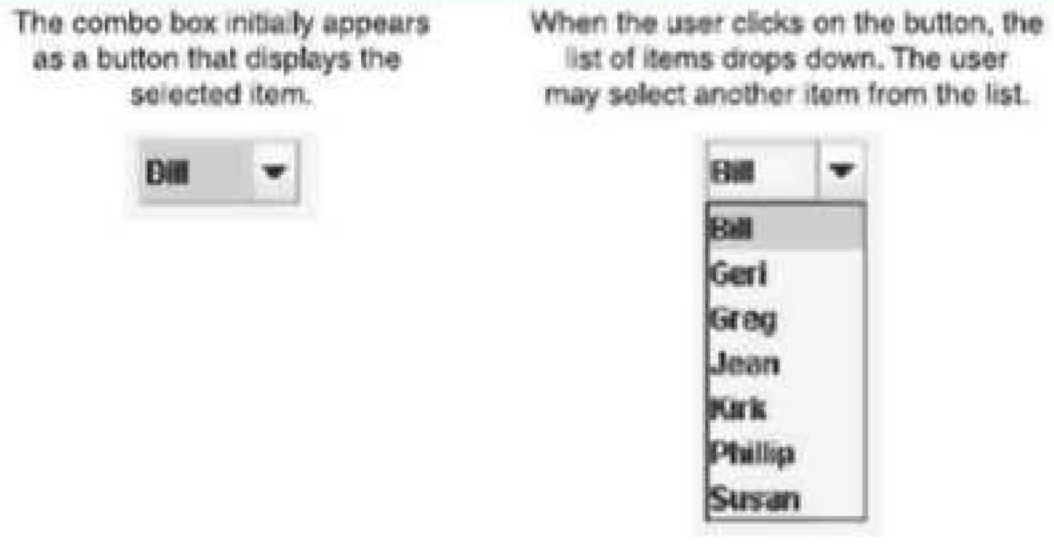
VideoNote The `JComboBox` Component

A combo box presents a list of items that the user may select from. Unlike a list component, a combo box presents its items in a drop-down list. You use the `JComboBox` class, which is in the `javax.swing` package, to create a combo box. You pass an array of objects that are to be displayed as the items in the drop-down list to the constructor. Here is an example:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
JComboBox nameBox = new JComboBox(names);
```

When displayed, the combo box created by this code will initially appear as the button shown on the left in Figure 13-11. The button displays the item that is currently selected. Notice that the first item in the list is automatically selected when the combo box is first displayed. When the user clicks the button, the drop-down list appears and the user may select another item.

Figure 13-11 A combo box


The combo box initially appears as a button that displays the selected item.
When the user clicks on the button, the list of items drops down. The user may select another item from the list.
Bill
Bill
Geri
Greg
Jean
Kirk
Phillip
Susan

As you can see, a combo box is a combination of two components. In the case of the combo box shown in Figure 13-11, it is the combination of a button and a list. This is where the name "combo box" comes from.

Responding to Combo Box Events

When an item in a `JComboBox` object is selected, it generates an action event. As with `JButton` components, you handle action events with an action event listener class, which must have an `actionPerformed` method. When the user selects an item in a combo box, the combo box executes its action event listener's `actionPerformed` method, passing an `ActionEvent` object as an argument.

Retrieving the Selected Item

There are two methods in the `JComboBox` class that you can use to determine which item in a combo box is currently selected: `getSelectedItem` and `getSelectedIndex`. The `getSelectedItem` method returns a reference to the item that is currently selected. For example, assume that `nameBox` references the `JComboBox` component shown earlier in Figure 13-11. The following code retrieves a reference to the name that is currently selected and assigns it to the `selectedName` variable:

```
String selectedName;
selectedName = (String) nameBox.getSelectedItem();
```

Note that the return value of the `getSelectedItem` method is an `Object` reference. In this code we had to cast the return value to the `String` type in order to store it in the `selectedName` variable.

The `getSelectedIndex` method returns the index of the selected item. As with `JList` components, the items that are stored in a combo box are numbered with indices that start at 0. You can use the index of the selected item to retrieve the item from an array. For example, assume that the following code was used to build the `nameBox` component shown in Figure 13-11:

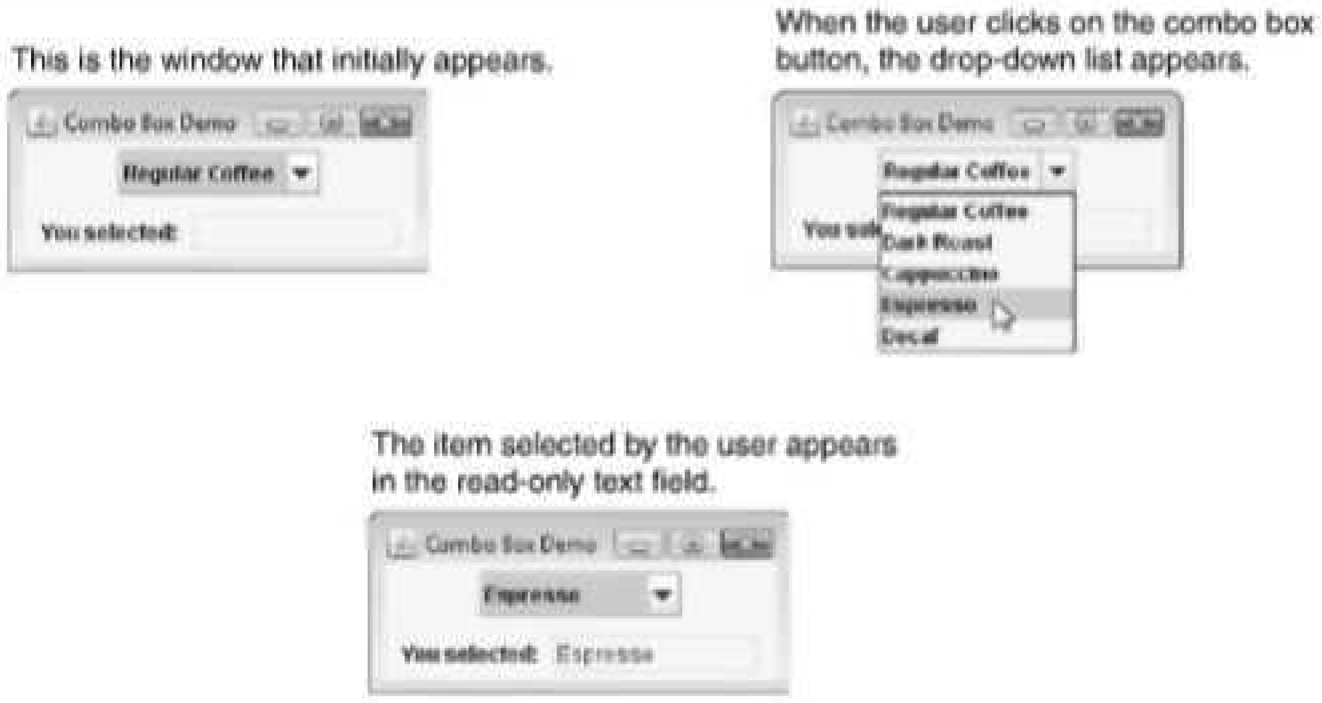
```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
JComboBox nameBox = new JComboBox(names);
```

Because the `names` array holds the values displayed in the `namesBox` component, the following code could be used to determine the selected item:

```
int index;
String selectedName;
index = nameList.getSelectedIndex();
selectedName = names[index];
```

The `ComboBoxWindow` class shown in Code Listing 13-4 demonstrates a combo box. It uses a `JComboBox` component with an action listener. When an item is selected from the combo box, it is displayed in a read-only text field. The `main` method creates an instance of the class, which initially displays the window shown at the top left of Figure 13-12. When the user clicks the combo box button, the drop-down list appears as shown in the top right of the figure. After the user selects Espresso from the list, the window appears as shown at the bottom of the figure.

Figure 13-12 The window displayed by the `ComboBoxWindow` class


This is the window that initially appears.

When the user clicks on the combo box button, the drop-down list appears.

The item selected by the user appears in the read-only text field.

Code Listing 13-4 `(ComboBoxWindow.java)`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
   This class demonstrates a combo box.
*/

public class ComboBoxWindow extends JFrame
{
   private JPanel coffeePanel;           // To hold components
   private JPanel selectedCoffeePanel;   // To hold components
   private JComboBox coffeeBox;          // A list of coffees
   private JLabel label;                 // Displays a message
   private JTextField selectedCoffee;    // Selected coffee

   // The following array holds the values that will
   // be displayed in the coffeeBox combo box.
   private String[] coffee = { "Regular Coffee",
                               "Dark Roast", "Cappuccino",
                               "Espresso", "Decaf"};

   /**
      Constructor
   */

   public ComboBoxWindow()
   {
      // Set the title.
      setTitle("Combo Box Demo");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create a BorderLayout manager.
      setLayout(new BorderLayout());

      // Build the panels.
      buildCoffeePanel();
      buildSelectedCoffeePanel();

      // Add the panels to the content pane.
      add(coffeePanel, BorderLayout.CENTER);
      add(selectedCoffeePanel, BorderLayout.SOUTH);

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildCoffeePanel method adds a combo box
      with the types of coffee to a panel.
   */

   private void buildCoffeePanel()
   {
      // Create a panel to hold the combo box.
      coffeePanel = new JPanel();

      // Create the combo box.
      coffeeBox = new JComboBox(coffee);

      // Register an action listener.
      coffeeBox.addActionListener(new ComboBoxListener());

      // Add the combo box to the panel.
      coffeePanel.add(coffeeBox);
   }

   /**
      The buildSelectedCoffeePanel method adds a
      read-only text field to a panel.
   */

   private void buildSelectedCoffeePanel()
   {
      // Create a panel to hold the components.
      selectedCoffeePanel = new JPanel();

      // Create the label.
      label = new JLabel("You selected: ");

      // Create the uneditable text field.
      selectedCoffee = new JTextField(10);
      selectedCoffee.setEditable(false);

      // Add the label and text field to the panel.
      selectedCoffeePanel.add(label);
      selectedCoffeePanel.add(selectedCoffee);
   }

   /**
      Private inner class that handles the event when
      the user selects an item from the combo box.
   */

   private class ComboBoxListener
                        implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         // Get the selected coffee.
         String selection =
                  (String) coffeeBox.getSelectedItem();

```

```
            // Display the selected coffee in the text field.
            selectedCoffee.setText(selection);
         }
      }

      /**
         The main method creates an instance of the
         ComboBoxWindow class, which causes it to display
         its window.
      */

      public static void main(String[] args)
      {
         new ComboBoxWindow();
      }
   }
```

Editable Combo Boxes

There are two types of combo boxes: uneditable and editable. The default type of combo box is uneditable. An uneditable combo box combines a button with a list and allows the user to select items from its list only. This is the type of combo box used in the previous examples.

An editable combo box combines a text field and a list. In addition to selecting items from the list, the user may also type input into the text field. You make a combo box editable by calling the component's `setEditable` method, passing `true` as the argument. Here is an example:

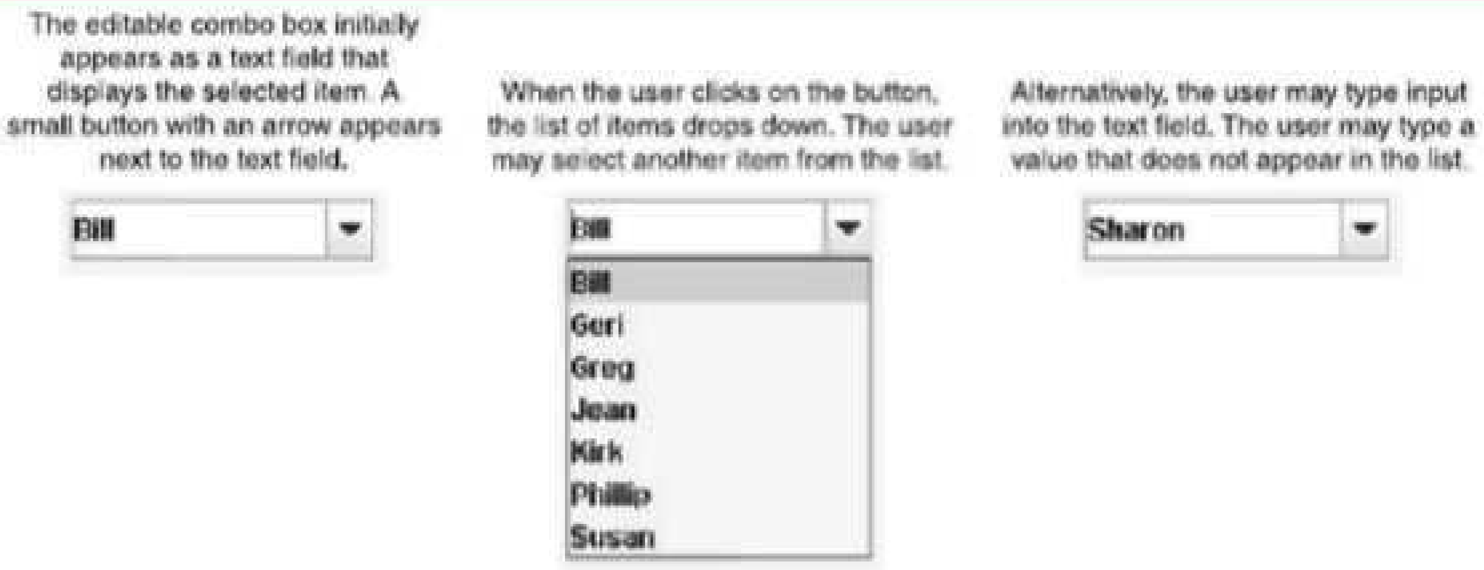
```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
JComboBox nameBox = new JComboBox(names);
nameBox.setEditable(true);
```

When displayed, the combo box created by this code initially appears as shown on the left of Figure 13-13. An editable combo box appears as a text field with a small button displaying an arrow joining it. The text field displays the item that is currently selected. When the user clicks the button, the drop-down list appears, as shown in the center of the figure. The user may select an item from the list. Alternatively, the user may type a value into the text field, as shown on the right of the figure. The user is not restricted to the values that appear in the list, and may type any input into the text field.

You can use the `getSelectedItem` method to retrieve a reference to the item that is currently selected. This method returns the item that appears in the combo box's text field, so it may or may not be an item that appears in the combo box's list.

The `getSelectedIndex` method returns the index of the selected item. However, if the user has entered a value in the text field that does not appear in the list, this method will return −1.

Figure 13-13 An editable combo box


The editable combo box initially appears as a text field that displays the selected item. A small button with an arrow appears next to the text field.
When the user clicks on the button, the list of items drops down. The user may select another item from the list.
Alternatively, the user may type input into the text field. The user may type a value that does not appear in the list.
Bill
Bill
Bill
Geri
Greg
Jean
Kirk
Phillip
Susan
Sharon

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

13.1 How do you make a text field read-only? In code, how do you store text in a text field?

13.2 What is the index of the first item stored in a `JList` or a `JComboBox` component? If one of these components holds 12 items, what is the index of the 12th item?

13.3 How do you retrieve the selected item from a `JList` component? How do you get the index of the selected item?

13.4 How do you cause a scroll bar to be displayed with a `JList` component?

13.5 How do you retrieve the selected item from a `JComboBox` component? How do you get the index of the selected item?

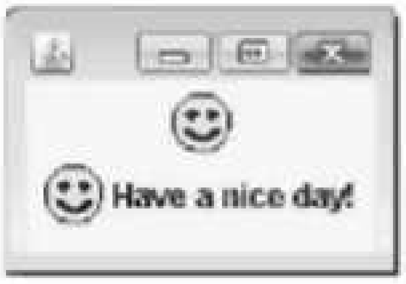
13.6 What is the difference between an uneditable and an editable combo box? Which of these is a combo box by default?

13.5 Displaying Images in Labels and Buttons

CONCEPT: Images may be displayed in labels and buttons. You use the `ImageIcon` class to get an image from a file.

In addition to displaying text in a label, you can also display an image. For example, Figure 13-14 shows a window with two labels. The top label displays a smiley face image and no text. The bottom label displays a smiley face image and text.

Figure 13-14 Labels displaying an image icon



To display an image, first you create an instance of the `ImageIcon` class, which can read the contents of an image file. The `ImageIcon` class is part of the `javax.swing` package. The constructor accepts a `String` argument that is the name of an image file. The supported file types are JPEG, GIF, and PNG. The name can also contain path information. Here is an example:

```
ImageIcon image = new ImageIcon("Smiley.gif");
```

This statement creates an `ImageIcon` object that reads the contents of the file *Smiley.gif*. Because no path was given, it is assumed that the file is in the current directory or folder. Here is an example that uses a path:

```
ImageIcon image = new ImageIcon("C:\\Chapter 13\\Images\\Smiley.gif");
```

Next, you can display the image in a label by passing the `ImageIcon` object as an argument to the `JLabel` constructor. Here is the general format of the constructor:

```
JLabel(Icon image)
```

The argument passed to the image parameter can be an `ImageIcon` object or any object that implements the `Icon` interface. Here is an example:

```
ImageIcon image = new ImageIcon("Smiley.gif");
JLabel label = new JLabel(image);
```

This creates a label with an image, but no text. You can also create a label with both an image and text. An easy way to do this is to create the label with text, as usual, and then use the `JLabel` class's `setIcon` method to add an image to the label. The `setIcon` method accepts an `ImageIcon` object as its argument. Here is an example:

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

The text will be displayed to the right of the image. The `JLabel` class also has the following constructor:

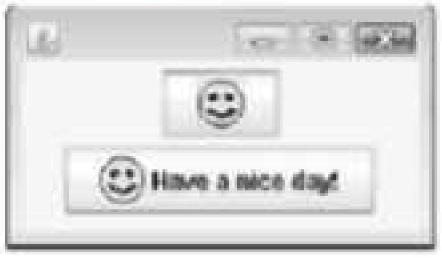
```
JLabel(String text, Icon image, int horizontalAlignment)
```

The first argument is the text to be displayed, the second argument is the image to be displayed, and the third argument is an `int` that specifies the horizontal alignment of the label contents. You should use the constants `SwingConstants.LEFT`, `SwingConstants.CENTER`, or `SwingConstants.RIGHT` to specify the horizontal alignment. Here is an example:

```
ImageIcon image = new ImageIcon("Smiley.gif");
JLabel label = new JLabel("Have a nice day!",
                          image,
                          SwingConstants.RIGHT);
```

You can also display images in buttons, as shown in Figure 13-15.

Figure 13-15 Buttons displaying an image icon



The process of creating a button with an image is similar to that of creating a label with an image. You use an `ImageIcon` object to read the image file, then pass the `ImageIcon` object as an argument to the `JButton` constructor. To create a button with an image and no text, pass only the `ImageIcon` object to the constructor. Here is an example:

```
// Create a button with an image, but no text.
ImageIcon image = new ImageIcon("Smiley.gif");
JButton button = new JButton(image);
```

To create a button with an image and text, pass a `String` and an `ImageIcon` object to the constructor. Here is an example:

```
// Create a button with an image and text.
ImageIcon image = new ImageIcon("Smiley.gif");
JButton button = new JButton("Have a nice day!", image);
```

To add an image to an existing button, pass an `ImageIcon` object to the button's `setIcon` method. Here is an example:

```
// Create a button with an image and text.
JButton button = new JButton("Have a nice day!");
ImageIcon image = new ImageIcon("Smiley.gif");
button.setIcon(image);
```

You are not limited to small graphical icons when placing images in labels or buttons. For example, the `MyCatImage` class in Code Listing 13-5 displays a digital photograph in a label when the user clicks a button. The `main` method creates an instance of the class, which displays the window shown at the left in Figure 13-16. When the user clicks the Get Image button, the window displays the image shown at the right in the figure.

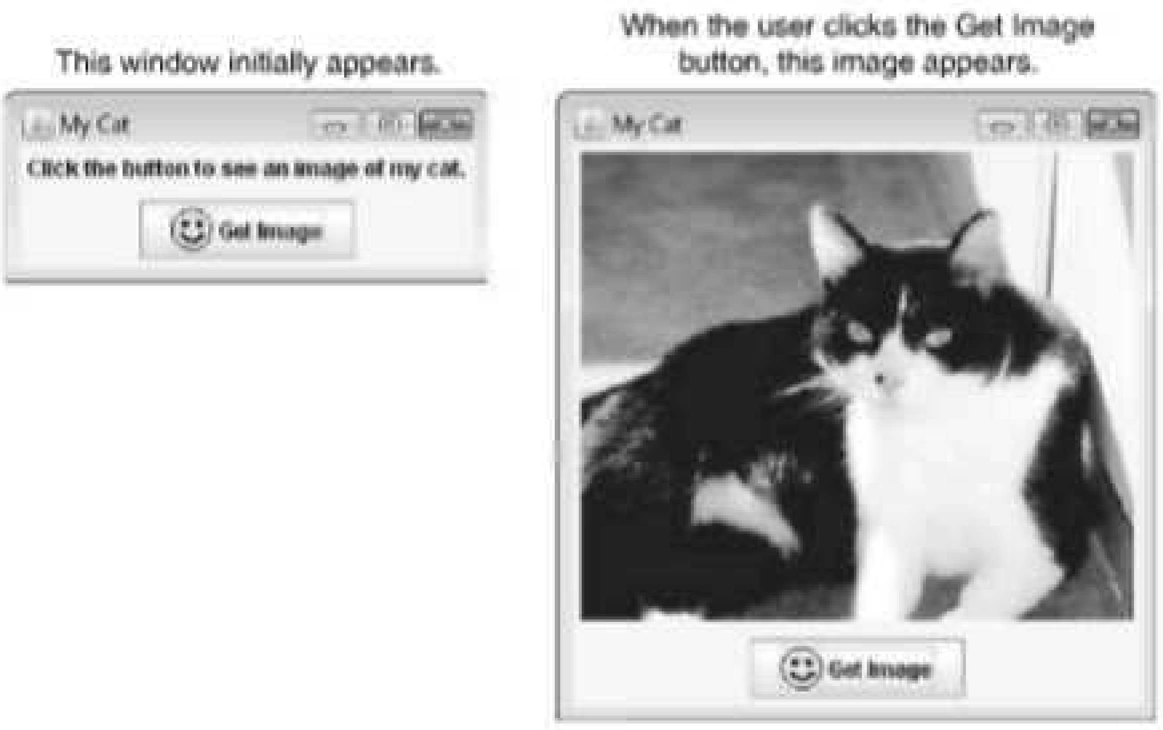
Code Listing 13-5 (`MyCatImage.java`)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
   This class demonstrates how to use an ImageIcon
   and a JLabel to display an image.
*/

```

```
public class MyCatImage extends JFrame
{
   private JPanel imagePanel;     // To hold the label
   private JPanel buttonPanel;    // To hold a button
   private JLabel imageLabel;     // To show an image
   private JButton button;        // To get an image


   /**
      Constructor
   */

   public MyCatImage()
   {
      // Set the title.
      setTitle("My Cat");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create a BorderLayout manager.
      setLayout(new BorderLayout());

      // Build the panels.
      buildImagePanel();
      buildButtonPanel();

      // Add the panels to the content pane.
      add(imagePanel, BorderLayout.CENTER);
      add(buttonPanel, BorderLayout.SOUTH);

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildImagePanel method adds a label to a panel.
   */

   private void buildImagePanel()
   {
      // Create a panel.
      imagePanel = new JPanel();

      // Create a label.
      imageLabel = new JLabel("Click the button to " +
                              "see an image of my cat.");

      // Add the label to the panel.
      imagePanel.add(imageLabel);
   }

   /**
      The buildButtonPanel method adds a button
      to a panel.
   */

   private void buildButtonPanel()
   {
      ImageIcon smileyImage;

      // Create a panel.
      buttonPanel = new JPanel();

      // Get the smiley face image.
      smileyImage = new ImageIcon("Smiley.gif");

      // Create a button.
      button = new JButton("Get Image");
      button.setIcon(smileyImage);

      // Register an action listener with the button.
      button.addActionListener(new ButtonListener());

      // Add the button to the panel.
      buttonPanel.add(button);
   }

   /**
      Private inner class that handles the event when
      the user clicks the button.
   */

   private class ButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         // Read the image file into an ImageIcon object.
         ImageIcon catImage = new ImageIcon("Cat.jpg");

         // Display the image in the label.
         imageLabel.setIcon(catImage);

         // Remove the text from the label.
         imageLabel.setText(null);
```

```
106
107             // Pack the frame again to accommodate the
108             // new size of the label.
109             pack();
110          }
111       }
112
113       /**
114          The main method creates an instance of the
115          MyCatImage class, which causes it to display
116          its window.
117       */
118       public static void main(String[] args)
119       {
120          new MyCatImage();
121       }
122 }
```

Figure 13-16 Window displayed by the `MyCatImage` class


This window initially appears.
When the user clicks the Get Image button, this image appears.
My Cat
Click the button to see an image of my cat.
Get Image
My Cat
Get Image

Let's take a closer look at the `MyCatImage` class. After some initial setup, the constructor calls the `buildImagePanel` method in line 34. Inside the `buildImagePanel` method, line 53 creates a `JPanel` component, referenced by the `imagePanel` variable, and then lines 56 and 57 create a `JLabel` component, referenced by the `imageLabel` variable. This is the label that will display the image when the user clicks the button. The last statement in the method, in line 60, adds the `imageLabel` component to the `imagePanel` panel.

Back in the constructor, line 35 calls the `buildButtonPanel` method, which creates the Get Image button and adds it to a panel. An instance of the `ButtonListener` inner class is also registered as the button's action listener. Let's look at the `ButtonListener` class's `actionPerformed` method. This method is executed when the user clicks the Get Image button. First, in line 99, an `ImageIcon` object is created from the file *Cat.jpg*. This file is in the same directory as the class. Next, in line 102, the image is stored in the `imageLabel` component. In line 105 the text that is currently displayed in the label is removed by passing `null` to the `imageLabel` component's `setText` method. The last statement, in line 109, calls the `JFrame` class's `pack` method. When the image was loaded into the `JLabel` component, the component resized itself to accommodate its new contents. The `JFrame` that encloses the window does not automatically resize itself, so we must call the `pack` method. This forces the `JFrame` to resize itself.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

13.7 How do you store an image in a `JLabel` component? How do you store both an image and text in a `JLabel` component?

13.8 How do you store an image in a `JButton` component? How do you store both an image and text in a `JButton` component?

13.9 What method do you use to store an image in an existing `JLabel` or `JButton` component?

13.6 Mnemonics and Tool Tips

CONCEPT: A mnemonic is a key that you press while holding down the Alt key to interact with a component. A tool tip is text that is displayed in a small box when the user holds the mouse cursor over a component.

Mnemonics

A mnemonic is a key on the keyboard that you press in combination with the Alt key to access a component such as a button quickly. These are sometimes referred to as shortcut keys, or hot keys. When you assign a mnemonic to a button, the user can click the button by holding down the Alt key and pressing the mnemonic key. Although users can interact with components with either the mouse or their mnemonic keys, those who are quick with the keyboard usually prefer to use mnemonic keys instead of the mouse.

You assign a mnemonic to a component through the component's `setMnemonic` method, which is inherited from the `AbstractButton` class. The method's general format is as follows:

```
void setMnemonic(int key)
```

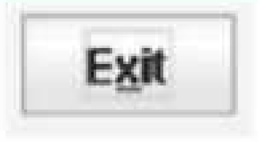
The argument that you pass to the method is an integer code that represents the key you wish to assign as a mnemonic. The `KeyEvent` class, which is in the `java.awt.event` package, has predefined constants that you can use. These constants take the form `KeyEvent.VK_x`, where `x` is a key on the keyboard. For example, to assign the A key as a mnemonic, you would use `KeyEvent.VK_A`. (The letters VK in the constants stand for "virtual key".) Here is an example of code that creates a button with the text "Exit" and assigns the X key as the mnemonic:

```
JButton exitButton = new JButton("Exit");
exitButton.setMnemonic(KeyEvent.VK_X);
```

The user may click this button by pressing a +X on the keyboard. (This means holding down the Alt key and pressing X.)

If the letter chosen as the mnemonic is in the component's text, the first occurrence of that letter will appear underlined when the component is displayed. For example, the button created with the previous code has the text "Exit". Because X was chosen as the mnemonic, the letter x will appear underlined, as shown in Figure 13-17.

Figure 13-17 Button with mnemonic X



If the mnemonic is a letter that does not appear in the component's text, then no letter will appear underlined.

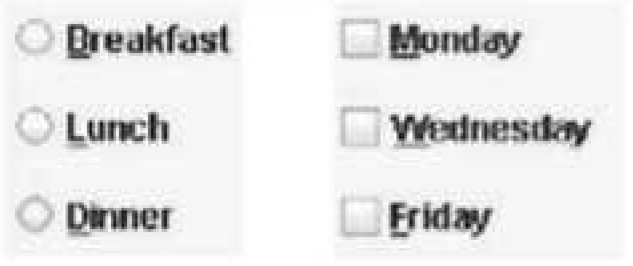
NOTE: The `KeyEvent` class also has constants for symbols. For example, the constant for the `!` symbol is `VK_EXCLAMATION_MARK`, and the constant for the `&` symbol is `VK_AMPERSAND`. See the Java API documentation for the `KeyEvent` class for a list of all the constants.

You can also assign mnemonics to radio buttons and check boxes, as shown in the following code:

```
//Create three radio buttons and assign mnemonics.
JRadioButton rb1 = new JRadioButton("Breakfast");
rb1.setMnemonic(KeyEvent.VK_B);
JRadioButton rb2 = new JRadioButton("Lunch");
rb2.setMnemonic(KeyEvent.VK_L);
JRadioButton rb3 = new JRadioButton("Dinner");
rb3.setMnemonic(KeyEvent.VK_D);

// Create three check boxes and assign mnemonics.
JCheckBox cb1 = new JCheckBox("Monday");
cb1.setMnemonic(KeyEvent.VK_M);
JCheckBox cb2 = new JCheckBox("Wednesday");
cb2.setMnemonic(KeyEvent.VK_W);
JCheckBox cb3 = new JCheckBox("Friday");
cb3.setMnemonic(KeyEvent.VK_F);
```

This code will create the components shown in Figure 13-18.

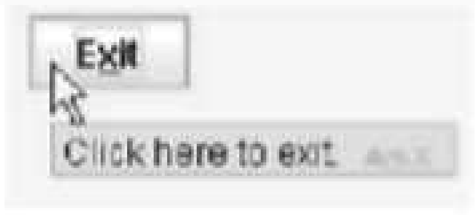
Figure 13-18 Radio buttons and check boxes with mnemonics assigned



Tool Tips

A tool tip is text that is displayed in a small box when the user holds the mouse cursor over a component. The box usually gives a short description of what the component does. Most GUI applications use tool tips as a way of providing immediate and concise help to the user. For example, Figure 13-19 shows a button with its tool tip displayed.

Figure 13-19 Button with tool tip displayed



You assign a tool tip to a component with the `setToolTipText` method, which is inherited from the `JComponent` class. Here is the method's general format:

```
void setToolTipText(String text)
```

The `String` that is passed as an argument is the text that will be displayed in the component's tool tip. For example, the following code creates the Exit button shown in Figure 13-19 and its associated tool tip:

```
JButton exitButton = new JButton("Exit");
exitButton.setToolTipText("Click here to exit.");
```

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

13.10 What is a mnemonic? How do you assign a mnemonic to a component?

13.11 What is a tool tip? How do you assign a tool tip to a component?

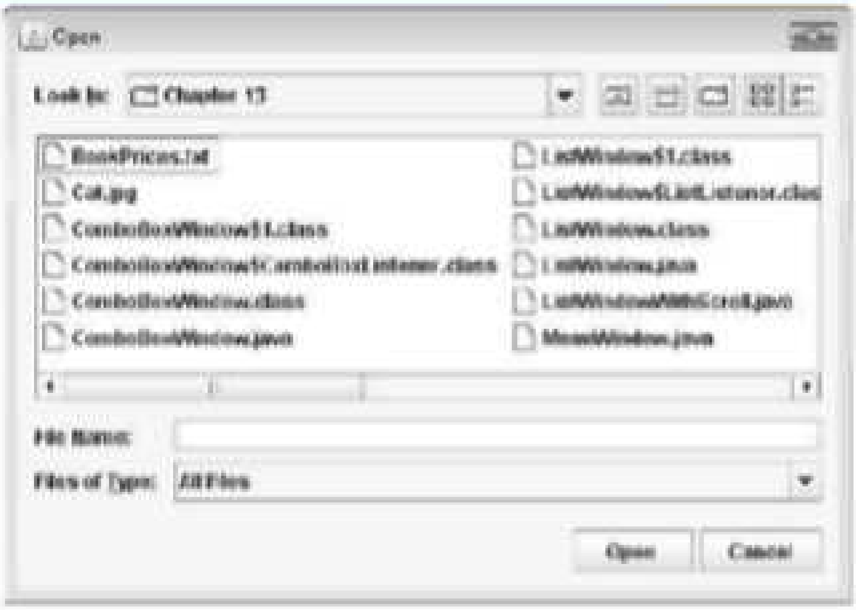
13.7 File Choosers and Color Choosers

CONCEPT: Java provides components that equip your applications with standard dialog boxes for opening files, saving files, and selecting colors.

File Choosers

A file chooser is a specialized dialog box that allows the user to browse for a file and select it. Figure 13-20 shows an example of a file chooser dialog box.

Figure 13-20 A file chooser dialog box for opening a file



You create an instance of the `JFileChooser` class, which is part of the `javax.swing` package, to display a file chooser dialog box. The class has several constructors. We will focus on two of them, which have the following general formats:

```
JFileChooser()
JFileChooser(String path)
```

The first constructor shown takes no arguments. This constructor uses the default directory as the starting point for all of its dialog boxes. If you are using Windows, this will probably be the "My Documents" folder under your account. If you are using UNIX, this will be your login directory. The second constructor takes a `String` argument containing a valid path. This path will be the starting point for the object's dialog boxes.

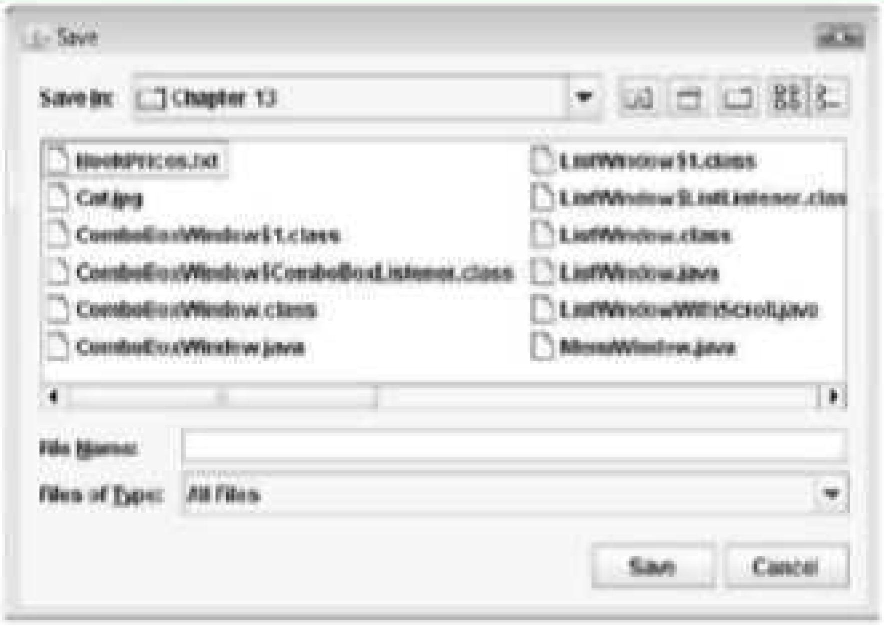
A `JFileChooser` object can display two types of predefined dialog boxes: an open file dialog box and a save file dialog box. Figure 13-20 shows an example of an open file dialog box. It lets the user browse for an existing file to open. A save file dialog box, as shown in Figure 13-21, is employed when the user needs to browse to a location to save a file. Both of these dialog boxes appear the same, except the open file dialog box displays "Open" in its title bar, and the save file dialog box displays "Save." Also, the open file dialog box has an Open button, and the save file dialog box has a Save button. There is no difference in the way they operate.

Displaying a File Chooser Dialog Box

To display an open file dialog box, use the `showOpenDialog` method. The method's general format is as follows:

```
int showOpenDialog(Component parent)
```

Figure 13-21 A save file dialog box



The argument can be either `null` or a reference to a component. If you pass `null`, the dialog box is normally centered in the screen. If you pass a reference to a component, such as `JFrame`, the dialog box is displayed over the component.

To display a save file dialog box, use the `showSaveDialog` method. The method's general format is as follows:

```
int showSaveDialog(Component parent)
```

Once again, the argument can be either `null` or a reference to a component. Both the `showOpenDialog` and `showSaveDialog` methods return an integer that indicates the action taken by the user to close the dialog box. You can compare the return value to one of the following constants:

- **`JFileChooser.CANCEL_OPTION`**. This return value indicates that the user clicked the Cancel button.
- **`JFileChooser.APPROVE_OPTION`**. This return value indicates that the user clicked the Open or Save button.
- **`JFileChooser.ERROR_OPTION`**. This return value indicates that an error occurred, or the user clicked the standard close button on the window to dismiss it.

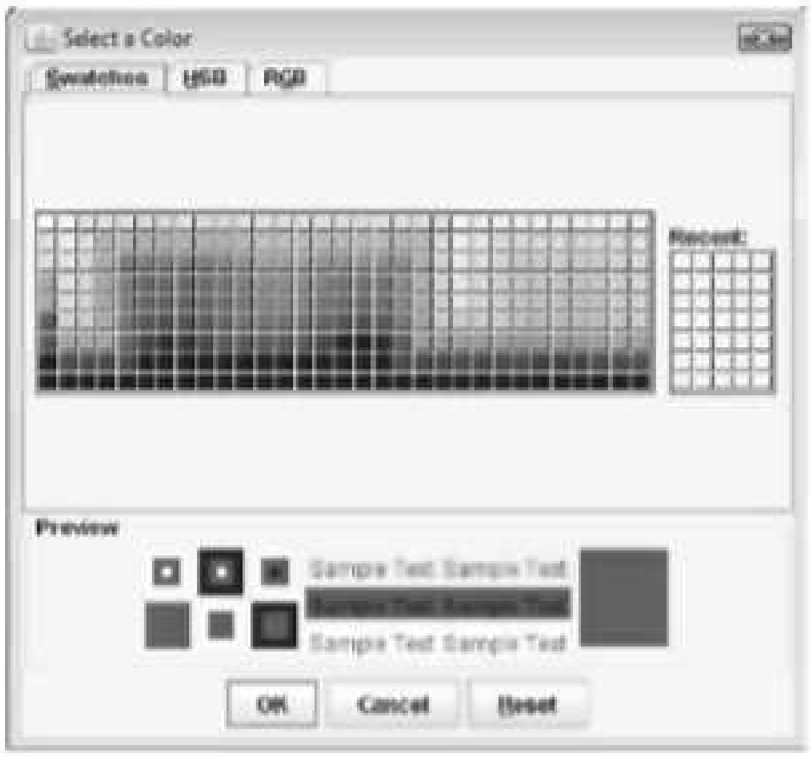
If the user selected a file, you can use the `getSelectedFile` method to determine the file that was selected. The `getSelectedFile` method returns a `File` object, which contains data about the selected file. The `File` class is part of the `java.io` package. You can use the `File` object's `getPath` method to get the path and file name as a `String`. Here is an example:

```
JFileChooser fileChooser = new JFileChooser();
int status = fileChooser.showOpenDialog(null);
if (status == JFileChooser.APPROVE_OPTION)
{
   File selectedFile = fileChooser.getSelectedFile();
   String filename = selectedFile.getPath();
   JOptionPane.showMessageDialog(null, "You selected " + filename);
}
```

Color Choosers

A color chooser is a specialized dialog box that allows the user to select a color from a predefined palette of colors. Figure 13-22 shows an example of a color chooser. By clicking the HSB tab you can select a color by specifying its hue, saturation, and brightness. By clicking the RGB tab you can select a color by specifying its red, green, and blue components.

Figure 13-22 A color chooser dialog box



You use the `JColorChooser` class, which is part of the `javax.swing` package, to display a color chooser dialog box. You do not create an instance of the class, however. It has a static method named `showDialog`, with the following general format:

```
Color showDialog(Component parent, String title, Color initial)
```

The first argument can be either `null` or a reference to a component. If you pass `null`, the dialog box is normally centered in the screen. If you pass a reference to a component, such as `JFrame`, the dialog box is displayed over the component. The second argument is text that is displayed in the dialog box's title bar. The third argument indicates the color that appears initially selected in the dialog box. This method returns the color selected by the user. The following code is an example. This code allows the user to select a color, and then that color is assigned as a panel's background color.

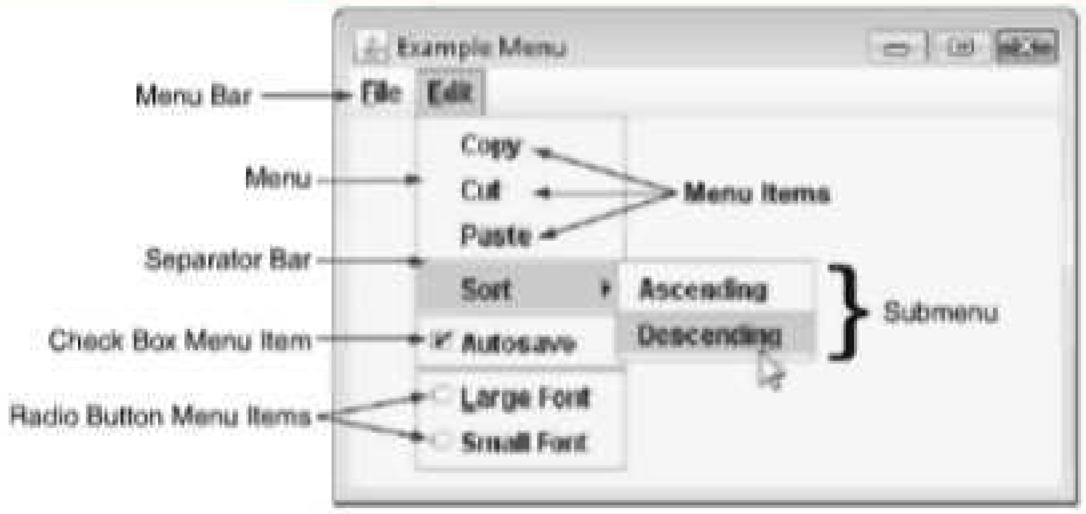
```
JPanel panel = new JPanel();
Color selectedColor;
selectedColor = JColorChooser.showDialog(null,
                "Select a Background Color", Color.BLUE);
panel.setBackground(selectedColor);
```

13.8 Menus

CONCEPT: Java provides classes for creating systems of drop-down menus. Menus can contain menu items, checked menu items, radio button menu items, and other menus.

In the GUI applications you have studied so far, the user initiates actions by clicking components such as buttons. When an application has several operations for the user to choose from, a menu system is more commonly used than buttons. A menu system is a collection of commands organized in one or more drop-down menus. Before learning how to construct a menu system, you must learn about the basic items that are found in a typical menu system. Look at the example menu system in Figure 13-23.

Figure 13-23 Example menu system



The menu system in the figure consists of the following items:

- **Menu Bar.** At the top of the window, just below the title bar, is a menu bar. The menu bar lists the names of one or more menus. The menu bar in Figure 13-23 shows the names of two menus: File and Edit.
- **Menu.** A menu is a drop-down list of menu items. The user may activate a menu by clicking on its name on the menu bar. In the figure, the Edit menu has been activated.
- **Menu Item.** A menu item can be selected by the user. When a menu item is selected, some type of action is usually performed.
- **Check box menu item.** A check box menu item appears with a small box beside it. The item may be selected or deselected. When it is selected, a check mark appears in the box. When it is deselected, the box appears empty. Check box menu items are normally used to turn an option on or off. The user toggles the state of a check box menu item each time he or she selects it.
- **Radio button menu item.** A radio button menu item may be selected or deselected. A small circle appears beside it that is filled in when the item is selected and empty when the item is deselected. Like a check box menu item, a radio button menu item can be used to turn an option on or off. When a set of radio button menu items are grouped

with a ButtonGroup object, only one of them can be selected at a time. When the user selects a radio button menu item, the one that was previously selected is deselected.

- **Submenu.** A menu within a menu is called a submenu. Some of the commands on a menu are actually the names of submenus. You can tell when a command is the name of a submenu because a small right arrow appears to its right. Activating the name of a submenu causes the submenu to appear. For example, in Figure 13-23, clicking on the Sort command causes a submenu to appear.
- **Separator bar.** A separator bar is a horizontal bar that is used to separate groups of items on a menu. Separator bars are only used as a visual aid and cannot be selected by the user.

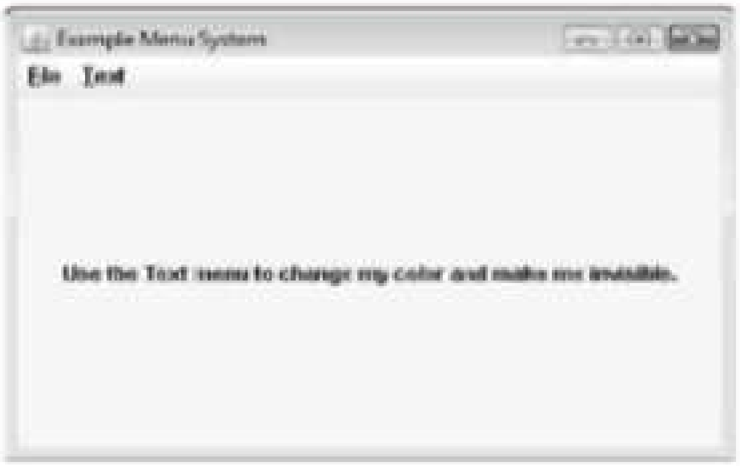
A menu system is constructed with the following classes:

- **`JMenuItem`.** Use this class to create a regular menu item. A `JMenuItem` component generates an action event when the user selects it.
- **`JCheckBoxMenuItem`.** Use this class to create a check box menu item. The class's `isSelected` method returns `true` if the item is selected, or `false` otherwise. A `JCheckBoxMenuItem` component generates an action event when the user selects it.
- **`JRadioButtonMenuItem`.** Use this class to create a radio button menu item. `JRadioButtonMenuItem` components can be grouped in a `ButtonGroup` object so that only one of them can be selected at a time. The class's `isSelected` method returns `true` if the item is selected, or `false` otherwise. A `JRadioButtonMenuItem` component generates an action event when the user selects it.
- **`JMenu`.** Use this class to create a menu. A `JMenu` component can contain `JMenuItem`, `JCheckBoxMenuItem`, and `JRadioButton` components, as well as other `JMenu` components. A submenu is a `JMenu` component that is inside another `JMenu` component.
- **`JMenuBar`.** Use this class to create a menu bar. A `JMenuBar` object can contain `JMenu` components.

All of these classes are in the `javax.swing` package. A menu system is a `JMenuBar` component that contains one or more `JMenu` components. Each `JMenu` component can contain `JMenuItem`, `JRadioButtonMenuItem`, and `JCheckBoxMenuItem` components, as well as other `JMenu` components. The classes contain all of the code necessary to operate the menu system.

To see an example of an application that uses a menu system, we look at the `MenuWindow` class shown in Code Listing 13-6. The class displays the window shown in Figure 13-24.

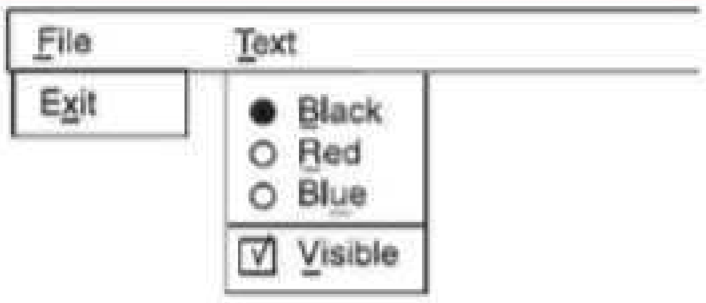
Figure 13-24 Window displayed by the `MenuWindow` class



Use the Text menu to change my color and make me invisible.

The class demonstrates how a label appears in different colors. Notice that the window has a menu bar with two menus: File and Text. Figure 13-25 shows a sketch of the menu system. When the user opens the Text menu, he or she can select a color using the radio button menu items and the label will change to the selected color. The Text menu also contains a Visible item, which is a check box menu item. When this item is selected (checked), the label is visible. When this item is deselected (unchecked), the label is invisible.

Figure 13-25 Sketch of the `MenuWindow` class's menu system



Code Listing 13-6 **(MenuWindow.java)**

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
   The MenuWindow class demonstrates a menu system.
*/

public class MenuWindow extends JFrame
{
   private JLabel messageLabel;               // Displays a message
   private final int LABEL_WIDTH = 400;       // Label's width
   private final int LABEL_HEIGHT = 200;      // Label's height

   // The following will reference menu components.
   private JMenuBar menuBar;                  // The menu bar
   private JMenu fileMenu;                    // The File menu
   private JMenu textMenu;                    // The Text menu
   private JMenuItem exitItem;                // To exit
   private JRadioButtonMenuItem blackItem;    // Makes text black
   private JRadioButtonMenuItem redItem;      // Makes text red
   private JRadioButtonMenuItem blueItem;     // Makes text blue
   private JCheckBoxMenuItem visibleItem;     // Toggle visibility

   /**
      Constructor
   */

   public MenuWindow()
   {
      // Set the title.
      setTitle("Example Menu System");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create the messageLabel label.
      messageLabel = new JLabel("Use the Text menu to " +
                 "change my color and make me invisible.",
                 SwingConstants.CENTER);

      // Set the label's preferred size.
      messageLabel.setPreferredSize(
              new Dimension(LABEL_WIDTH, LABEL_HEIGHT));

      // Set the label's foreground color.
      messageLabel.setForeground(Color.BLACK);

      // Add the label to the content pane.
      add(messageLabel);

      // Build the menu bar.
      buildMenuBar();

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildMenuBar method builds the menu bar.
   */

   private void buildMenuBar()
   {
      // Create the menu bar.
      menuBar = new JMenuBar();

      // Create the file and text menus.
      buildFileMenu();
      buildTextMenu();

      // Add the file and text menus to the menu bar.
      menuBar.add(fileMenu);
      menuBar.add(textMenu);

      // Set the window's menu bar.
      setJMenuBar(menuBar);
   }

   /**
      The buildFileMenu method builds the File menu
      and returns a reference to its JMenu object.
   */

   private void buildFileMenu()
   {
      // Create an Exit menu item.
      exitItem = new JMenuItem("Exit");
      exitItem.setMnemonic(KeyEvent.VK_X);
      exitItem.addActionListener(new ExitListener());

      // Create a JMenu object for the File menu.
      fileMenu = new JMenu("File");
      fileMenu.setMnemonic(KeyEvent.VK_F);

      // Add the Exit menu item to the File menu.
      fileMenu.add(exitItem);
   }

   /**
      The buildTextMenu method builds the Text menu
      and returns a reference to its JMenu object.
   */

   private void buildTextMenu()
   {
      // Create the radio button menu items to change
      // the color of the text. Add an action listener
      // to each one.
      blackItem = new JRadioButtonMenuItem("Black", true);
      blackItem.setMnemonic(KeyEvent.VK_B);
      blackItem.addActionListener(new ColorListener());

      redItem = new JRadioButtonMenuItem("Red");
      redItem.setMnemonic(KeyEvent.VK_R);
      redItem.addActionListener(new ColorListener());

      blueItem = new JRadioButtonMenuItem("Blue");
      blueItem.setMnemonic(KeyEvent.VK_U);
      blueItem.addActionListener(new ColorListener());

      // Create a button group for the radio button items.
      ButtonGroup group = new ButtonGroup();
```