CHAPTER 23

A First Look at GUI Applications with Swing

TOPICS

NOTE: This chapter discusses GUI development using the Swing classes. Oracle has announced that JavaFX is replacing Swing as the standard GUI library for Java. Swing will remain part of the Java API for the foreseeable future, however, so we are providing this chapter for you to use as you make the transition from Swing to JavaFX. To learn about JavaFX, see Chapters 12, 13, and 14.

23.1 Introduction

CONCEPT: In Java, you use the Java Foundation Classes (JFC) to create a graphical user interface for your application. Within the JFC you use the Abstract Windowing Toolkit (AWT) or Swing classes to create a graphical user interface.

In this chapter, we discuss the basics of creating a Java application with a *graphical user interface* or *GUI* (pronounced "gooey"). A GUI is a graphical window or a system of graphical windows that is presented by an application for interaction with the user. In addition to accepting input from the keyboard, GUIs typically accept input from a mouse as well.

A window in a GUI commonly consists of several *components* that present data to the user and/or allow interaction with the application. Some of the common GUI components are buttons, labels, text fields, check boxes, and radio buttons. Figure 23-1 shows an example of a window with a variety of components. Table 23-1 describes the components that appear in the window.

Figure 23-1 Various GUI components (Oracle Corporate Counsel)

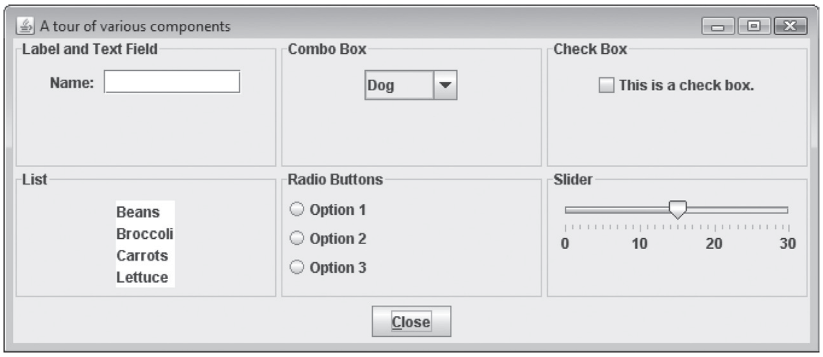


Table 23-1 Some GUI components

Component	Description
Label	An area that can display text.
Text field	An area in which the user may type a single line of input from the keyboard.
Combo box	A component that displays a drop-down list of items from which the user may select. A combo box also provides a text field in which the user may type input. It is called a combo box because it is the combination of a list and a text field.
Check box	A component that has a box that may be checked or unchecked.
List	A list from which the user may select an item.
Radio button	A component that can be either selected or deselected. Radio buttons usually appear in groups and allow the user to select one of several options.
Slider	A component that allows the user to select a value by moving a slider along a track.
Button	A button that can cause an action to occur when it is clicked.

The JFC, AWT, and Swing

Java programmers use the *Java Foundation Classes (JFC)* to create GUI applications. The JFC consists of several sets of classes, many of which are beyond the scope of this book. The two sets of JFC classes that we focus on are the AWT and Swing classes. First, we discuss the differences between them.

Java has been equipped, since its earliest version, with a set of classes for drawing graphics and creating GUIs. These classes are part of the *Abstract Windowing Toolkit (AWT)*. The AWT allows programmers to create applications and applets that interact with the user via windows and other GUI components.

Programmers are limited in what they can do with the AWT classes, however. This is because the AWT classes do not actually draw user interface components on the screen. Instead, the AWT classes communicate with another layer of software, known as the *peer classes*, which directs the underlying operating system to draw its own built-in components. Each version of Java that is developed for a particular operating system has its own set of peer classes. Although this means that Java programs have a look that is consistent with other applications on the same system, it also leads to some problems.

One problem is that not all operating systems offer the same set of GUI components. For example, one operating system might provide a sophisticated slider bar component that is not found on any other platform. Other operating systems might have their own unique components as well. In order for the AWT to retain its portability, it has to offer only those components that are common to all the operating systems that support Java.

Another problem is in the behavior of components across various operating systems. A component on one operating system might have slightly different behavior than the same component on a different operating system. In addition, the peer classes for some operating systems reportedly have bugs. As a result, programmers cannot be completely sure how their AWT programs will behave on different operating systems until they test each one.

A third problem is that programmers cannot easily customize the AWT components. Because these components rely on the appearance and behavior of the underlying operating system components, there is little that can be done by the programmer to change their properties.

To remedy these problems, Swing was introduced with the release of Java 2. *Swing* is a library of classes that do not replace the AWT, but provide an improved alternative for creating GUI applications and applets. Very few of the Swing classes rely on an underlying system of peer classes. Instead, Swing draws most of its own components on the screen. This means that Swing components can have a consistent look and predictable behavior on any operating system.

NOTE: Swing applications can have the look of a specific operating system. The programmer may choose from a variety of "look and feel" themes.

Swing components can also be easily customized. The Swing library provides many sophisticated components that are not found in the AWT. In this chapter and in Chapter 24, we primarily use Swing to develop GUI applications. In Chapter 25, we use AWT to develop applets.

NOTE: AWT components are commonly called heavyweight components because they are coupled with their underlying peer classes. Very few of the Swing components are coupled with peer classes, so they are referred to as lightweight components.

Event-Driven Programming

Programs that operate in a GUI environment must be *event-driven*. An *event* is an action that takes place within a program, such as the clicking of a button. Part of writing a GUI application is creating event listeners. An *event listener* is an object that automatically executes one of its methods when a specific event occurs. If you wish for an application to perform an operation when a particular event occurs, you must create an event listener object that responds when that event takes place.

The `javax.swing` and `java.awt` Packages

In this chapter, we use the Swing classes for all of the graphical components that we create in our GUIs. The Swing classes are part of the `javax.swing` package. (Take note of the letter `x` that appears after the word `java`.) The following `import` statement will be used in every applicaton:

```
import javax.swing.*;
```

We also use some of the AWT classes to determine when events, such as the clicking of a mouse, take place in our applications. The AWT classes are part of the `java.awt` package. (Note that there is no `x` after `java` in this package name.) Programs that use the AWT classes will have the following `import` statement:

```
import java.awt.*;
```

23.2 Creating Windows

CONCEPT: You can use Swing classes to create windows containing various GUI components.

VideoNote
Creating a Simple GUI Application

The `JOptionPane` dialog boxes that you learned about in Chapter 2 allow you to easily display messages and gather input. If an application is to provide a full graphical user interface, however, much more is needed. Often, applications need one or more windows with various components that allow the user to enter and/or select data and interact with the application. For example, the window that is displayed in Figure 23-1 has several different components within it.

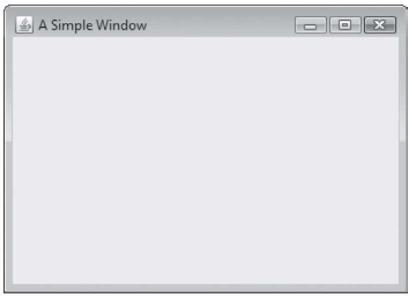
A window is a component, but because a window contains other components, it is more appropriately considered a container. A *container* is simply a component that holds other components. In GUI terminology, a container that can be displayed as a window is known as a *frame*. A frame appears as a basic window that has a border around it, a title bar, and a set of buttons for minimizing, maximizing, and closing the window. In a Swing application, you create a frame object from the `JFrame` class.

There are a number of steps involved in creating a window, so let's look at an example. The program in Code Listing 23-1 displays the window shown in Figure 23-2.

Code Listing 23-1 (`ShowWindow.java`)

```java
import javax.swing.*;   // Needed for Swing classes

/**
   This program displays a simple window with a title. The
   application exits when the user clicks the close button.
*/

public class ShowWindow
{
   public static void main(String[] args)
   {
      final int WINDOW_WIDTH = 350;   // Window width in pixels
      final int WINDOW_HEIGHT = 250;  // Window height in pixels

      // Create a window.
      JFrame window = new JFrame();

      // Set the title.
      window.setTitle("A Simple Window");

      // Set the size of the window.
      window.setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

      // Specify what happens when the close button is clicked.
      window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

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

Figure 23-2 Window displayed by `ShowWindow.java` (Oracle Corporate Counsel)



The window shown in Figure 23-2 was produced on a system running Microsoft Windows. Notice that the window has a border and a title bar with "A Simple Window" displayed in it. In addition, it has the standard Microsoft Windows buttons in the upper-right corner: a minimize button, a maximize button, and a close button. These standard features are sometimes referred to as *decorations*. If you run this program, you will see the window displayed on your screen. When you click on the close button, the window disappears and the program terminates.

Let's take a closer look at the code. First, notice that the following `import` statement is used in line 1:

```
import javax.swing.*;   // Needed for Swing classes
```

Any program that uses a Swing class, such as `JFrame`, must have this `import` statement. In lines 12 and 13 the two constants `WINDOW_WIDTH` and `WINDOW_HEIGHT` are declared as follows:

```
final int WINDOW_WIDTH = 350;   // Window width in pixels
final int WINDOW_HEIGHT = 250;  // Window height in pixels
```

We use these constants later in the program to set the size of the window. The window's size is measured in pixels. A *pixel* is one of the small dots that make up a screen display; the resolution of your monitor is measured in pixels. For example, if your monitor's resolution is 1024 by 768, that means the width of your screen is 1024 pixels, and the height of your screen is 768 pixels.

Next, we create an instance of the `JFrame` class with the following statement in line 16:

```
JFrame window = new JFrame();
```

This statement creates a `JFrame` object in memory and assigns its address to the `window` variable. This statement does not display the window on the screen, however. A `JFrame` is initially invisible.

In line 19 we call the `JFrame` object's `setTitle` method as follows:

```
window.setTitle("A Simple Window");
```

The string that is passed as an argument to `setTitle` will appear in the window's title bar when it is displayed. In line 22 we call the `JFrame` object's `setSize` method to set the window's size as follows:

```
window.setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
```

The two arguments passed to `setSize` specify the window's width and height in pixels. In this program we pass the constants `WINDOW_WIDTH` and `WINDOW_HEIGHT`, which we declared earlier, to set the size of the window to 350 pixels by 250 pixels.

In line 25 we specify the action that we wish to take place when the user clicks on the close button, which appears in the upper-right corner of the window as follows:

```
window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

There are a number of actions that can take place when the user clicks on the close button. The `setDefaultCloseOperation` method takes an `int` argument, which specifies the action. In this statement, we pass the constant `JFrame.EXIT_ON_CLOSE`, which causes the application to end with a System.exit method call. If we had passed JFrame.HIDE_ON_CLOSE, the window would be hidden from view, but the application would not end. The default action is JFrame.HIDE_ON_CLOSE.

Last, in line 28, we use the following code to display the window:

```
window.setVisible(true);
```

The setVisible method takes a boolean argument. If the argument is true, the window is made visible. If the argument is false, the window is hidden.

Using Inheritance to Extend the JFrame Class

The program in Code Listing 23-1 performs a very simple operation: It creates an instance of the JFrame class and displays it. Most of the time, your GUI applications will be much more involved than this. As you progress through this chapter, you will add numerous components and capabilities to the windows that you create.

Instead of simply creating an instance of the JFrame class, as shown in Code Listing 23-1, a more common technique is to use inheritance to create a new class that extends the JFrame class.

IF YOU'VE SKIPPED AHEAD TO THIS CHAPTER: This chapter is written so that you can skip ahead to it any time after Chapter 6. Reading about inheritance and interfaces in Chapter 10 would be helpful; but if you have not read that material yet, the following summarizes what you need to know for this chapter.

When a new class *extends* an existing class, it inherits many of the existing class's members just as if they were part of the new class. For example, you saw how the program in Code Listing 23-1 created a JFrame object and then called four of its methods: setTitle, setSize, setDefaultCloseOperation, and setVisible. These methods are all members of the JFrame class. If you create a new class that extends the JFrame class, the new class will automatically inherit these methods. Then these methods can be called from an instance of the new class just as if they were written into its declaration. You can add your own custom code to the new class, making it a specialized, or extended, version of the JFrame class. Programs can then create instances of your new specialized class instead of the more generic JFrame class.

Let's look at the SimpleWindow class in Code Listing 23-2. This is an example of a class that extends the JFrame class.

Code Listing 23-2 (SimpleWindow.java)

```
import javax.swing.*;   // Needed for Swing classes

/**
   This class extends the JFrame class. Its constructor displays
   a simple window with a title. The application exits when the
```

```
 6     user clicks the close button.
 7  */
 8
 9  public class SimpleWindow extends JFrame
10  {
11     /**
12        Constructor
13     */
14
15     public SimpleWindow()
16     {
17        final int WINDOW_WIDTH = 350;   // Window width in pixels
18        final int WINDOW_HEIGHT = 250;  // Window height in pixels
19
20        // Set this window's title.
21        setTitle("A Simple Window");
22
23        // Set the size of this window.
24        setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
25
26        // Specify what happens when the close button is clicked.
27        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28
29        // Display the window.
30        setVisible(true);
31     }
32  }
```

Notice the class header in line 9 as follows:

```
public class SimpleWindow extends JFrame
```

The words `extends JFrame` indicate that the `SimpleWindow` class extends the `JFrame` class. This means that the `SimpleWindow` class inherits members of the `JFrame` class, such as the `setTitle`, `setSize`, `setDefaultCloseOperation`, and `setVisible` methods, just as if they were written into the `SimpleWindow` class declaration. Now look at the constructor. In lines 17 and 18 we declare the `WINDOW_WIDTH` and `WINDOW_HEIGHT` constants, which will be used to establish the size of the window as follows:

```
final int WINDOW_WIDTH = 350;   // Window width in pixels
final int WINDOW_HEIGHT = 250;  // Window height in pixels
```

In line 21 we call the `setTitle` method to set the text for the window's title bar as follows:

```
setTitle("A Simple Window");
```

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 `JFrame` class, and we can call it just as if it were written into the `SimpleWindow` class declaration.

The rest of the constructor calls the `setSize`, `setDefaultCloseOperation`, and `setVisible` methods. All that is necessary to display the window is to create an instance of the `SimpleWindow` class, as shown in the program in Code Listing 23-3. When this program runs, the window that was previously shown in Figure 23-2 is displayed. Remember, the `SimpleWindow` class is an extended version of the `JFrame` class. When we create an instance of the `SimpleWindow` class, we are really creating an instance of the `JFrame` class, with some customized code added to its constructor.

Code Listing 23-3 (`SimpleWindowDemo.java`)

```
/**
   This program creates an instance of the
   SimpleWindow class.
*/

public class SimpleWindowDemo
{
   public static void main(String[] args)
   {
      SimpleWindow myWindow = new SimpleWindow();
   }
}
```

Equipping GUI Classes with a `main` Method

You know that a Java application always starts execution with a static method named `main`. The previous example consists of two separate files:

- `SimpleWindow.java`: This file contains the `SimpleWindow` class, which defines a GUI window.
- `SimpleWindowDemo.java`: This file contains a static `main` method that creates an object of the GUI window class, thus displaying it.

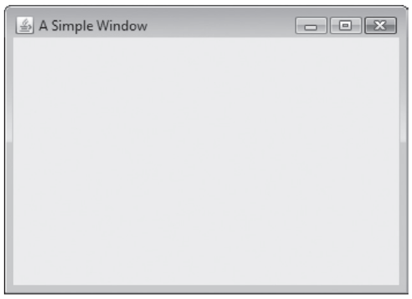
The purpose of the `SimpleWindowDemo.java` file is simply to create an instance of the `SimpleWindow` class. It is possible to eliminate the second file, `SimpleWindowDemo.java`, by writing the static `main` method directly into the `SimpleWindow.java` file. The `EmbeddedMain` class in Code Listing 23-4 shows an example.

Code Listing 23-4 (`EmbeddedMain.java`)

```
import javax.swing.*; // Needed for Swing classes

/**
   This class defines a GUI window and has its own
   main method.
*/

public class EmbeddedMain extends JFrame
{
   final int WINDOW_WIDTH = 350;   // Window width in pixels
   final int WINDOW_HEIGHT = 250;  // Window height in pixels
12
13    /**
14       Constructor
15    */
16
17    public EmbeddedMain()
18    {
19       // Set this window's title.
20       setTitle("A Simple Window");
21
22       // Set the size of this window.
23       setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
24
25       // Specify what happens when the close button is clicked.
26       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27
28       // Display the window.
29       setVisible(true);
30    }
31
32    /**
33       The main method creates an instance of the EmbeddedMain
34       class, which causes it to display its window.
35    */
36
37    public static void main(String[] args)
38    {
39       EmbeddedMain em = new EmbeddedMain();
40    }
41 }
```

The `EmbeddedMain` class contains its own static `main` method (in lines 37 through 40), which creates an instance of the class. Notice that the `main` method has exactly the same header as any other static `main` method that we have written. We can compile the *EmbeddedMain.java* file and then run the resulting *.class* file. When we do, we see the window shown in Figure 23-3.

Figure 23-3 Window displayed by the `EmbeddedMain` class (Oracle Corporate Counsel)



Notice that in line 39 the `main` method declares a variable named `em` to reference the instance of the class. Once the instance is created, however, the variable is not used again. Because we do not need the variable, we can instantiate the class *anonymously* as shown here:

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

In this version of the method, an instance of the `EmbeddedMain` class is created in memory, but its address is not assigned to any reference variable.

Adding Components to a Window

Swing provides numerous GUI components that can be added to a window. Three fundamental components are the label, the text field, and the button. These are summarized in Table 23-2.

Table 23-2 Label, text field, and button controls

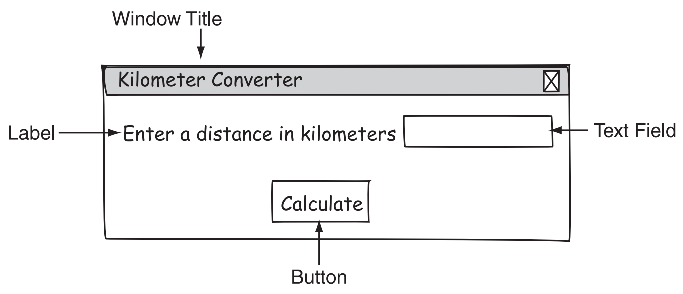
Component	Swing Class	Description
Label	`JLabel`	An area that can display text
Text field	`JTextField`	An area in which the user may type a single line of input from the keyboard
Button	`JButton`	A button that can cause an action to occur when it is clicked

In Swing, labels are created with the `JLabel` class, text fields are created with the `JTextField` class, and buttons are created with the `JButton` class. To demonstrate these components, we will build a simple GUI application: The Kilometer Converter. This application will present a window in which the user will be able to enter a distance in kilometers, and then click a button to see that distance converted to miles. The conversion formula is as follows:

$$Miles = Kilometers \times 0.6214$$

When designing a GUI application, it is usually helpful to draw a sketch showing the window you are creating. Figure 23-4 shows a sketch of what the Kilometer Converter application's window will look like. As you can see from the sketch, the window will have a label, a text field, and a button. When the user clicks the button, the distance in miles will be displayed in a separate `JOptionPane` dialog box.

Figure 23-4 Sketch of the Kilometer Converter window (Oracle Corporate Counsel)

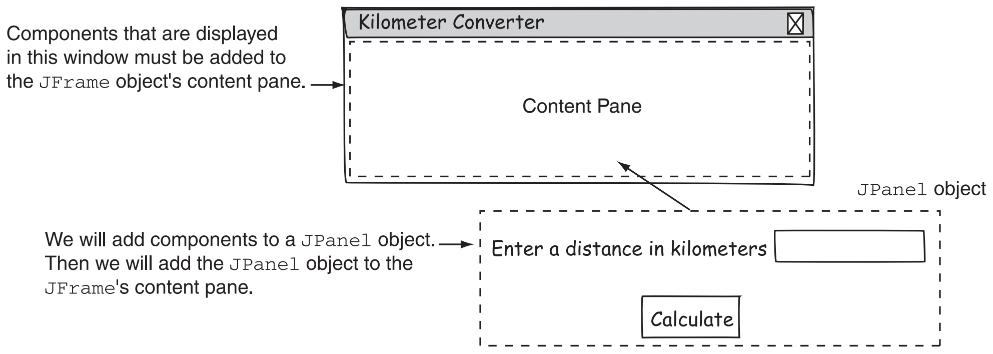


Content Panes and Panels

Before we start writing code, you should be familiar with content panes and panels. A *content pane* is a container that is part of every `JFrame` object. You cannot see the content pane and it does not have a border, but any component that is to be displayed in a `JFrame` must be added to its content pane.

A *panel* is also a container that can hold GUI components. Unlike `JFrame` objects, panels cannot be displayed by themselves; however, they are commonly used to hold and organize collections of related components. With Swing, you create panels with the `JPanel` class. In our Kilometer Converter application, we will create a panel to hold the label, text field, and button. Then we will add the panel to the `JFrame` object's content pane. This is illustrated in Figure 23-5.

Figure 23-5 A panel is added to the content pane (Oracle Corporate Counsel)


Components that are displayed in this window must be added to the JFrame object's content pane.
Kilometer Converter
Content Pane
JPanel object
We will add components to a JPanel object. Then we will add the JPanel object to the JFrame's content pane.
Enter a distance in kilometers
Calculate

Code Listing 23-5 shows the initial code for the `KiloConverter` class. We will be adding to this code as we develop the application. This version of the class is stored in the source code folder *Chapter 23\KiloConverter Phase 1*.

Code Listing 23-5 (`KiloConverter.java`)

```
import javax.swing.*;

/**
   The KiloConverter class displays a JFrame that
   lets the user enter a distance in kilometers. When
   the Calculate button is clicked, a dialog box is
   displayed with the distance converted to miles.
*/

public class KiloConverter extends JFrame
{
   private JPanel panel;                 // To reference a panel
   private JLabel messageLabel;          // To reference a label
   private JTextField kiloTextField;     // To reference a text field
   private JButton calcButton;           // To reference a button
   private final int WINDOW_WIDTH = 310; // Window width
   private final int WINDOW_HEIGHT = 100;// Window height

   /**
      Constructor
   */

   public KiloConverter()
   {
      // Set the window title.
      setTitle("Kilometer Converter");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

      // Specify what happens when the close button is clicked.
      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");
54
55          // Create a text field 10 characters wide.
56          kiloTextField = new JTextField(10);
57
58          // Create a button with the caption "Calculate".
59          calcButton = new JButton("Calculate");
60
61          // Create a JPanel object and let the panel
62          // field reference it.
63          panel = new JPanel();
64
65          // Add the label, text field, and button
66          // components to the panel.
67          panel.add(messageLabel);
68          panel.add(kiloTextField);
69          panel.add(calcButton);
70       }
71
72       /**
73          main method
74       */
75
76       public static void main(String[] args)
77       {
78          new KiloConverter();
79       }
80    }
```

Let's take a closer look at this class. First, notice in line 10 that the `KiloConverter` class extends the `JFrame` class as follows:

```
public class KiloConverter extends JFrame
```

Next, in lines 12 through 17, notice in the following that the class declares a number of fields, and according to good class design principles, the fields are private:

```
private JPanel panel;                   // To reference a panel
private JLabel messageLabel;            // To reference a label
private JTextField kiloTextField;       // To reference a text field
private JButton calcButton;             // To reference a button
private final int WINDOW_WIDTH = 310;   // Window width
private final int WINDOW_HEIGHT = 100;  // Window height
```

The statement in line 12 declares a `JPanel` reference variable named `panel`, which we will use to reference the panel that will hold the other components. The `messageLabel` variable, declared in line 13, will reference a `JLabel` object that displays a message instructing the user to enter a distance in kilometers. The `kiloTextField` variable, declared in line 14, will reference a `JTextField` object that will hold a value typed by the user. The `calcButton` variable, declared in line 15, will reference a `JButton` object that will calculate and display the kilometers converted to miles when clicked. The `WINDOW_WIDTH` and `WINDOW_HEIGHT` fields, declared in lines 16 and 17, are constants that hold the width and height of the window.

Now let's look at the constructor. In line 26 the `setTitle` method, which was inherited from the `JFrame` class, is called to set the text for the window's title bar. Next, in line 29, the inherited `setSize` method is called to establish the size of the window. In line 32, the inherited `setDefaultCloseOperation` method is called to establish the action that should occur when the window's close button is clicked.

Line 35 calls the `buildPanel` method. The `buildPanel` method is defined in this class, in lines 49 through 70. The purpose of the `buildPanel` method is to create a label, a text field, and a button, and then add those components to a panel. Let's look at the method.

First, look at the method header in line 49 and notice that it is declared private. When a method is private, only other methods in the same class can call it. This method is not meant to be called by code outside the class, so it is declared private. In lines 52 and 53, the method uses the following statement to create a `JLabel` object and assign its address to the `message` field:

```
messageLabel = new JLabel("Enter a distance " +
                          "in kilometers");
```

The string that is passed to the `JLabel` constructor is the text that will be displayed in the label. The following statement appears in line 56. It creates a `JTextField` object, and assigns its address to the `kiloTextField` field:

```
kiloTextField = new JTextField(10);
```

The argument that is passed to the `JTextField` constructor is the width of the text field in columns. One column is enough space to hold the letter "m," which is the widest letter in the alphabet.

The following statement appears in line 59; it creates a `JButton` object, and assigns its address to the `calcButton` field:

```
calcButton = new JButton("Calculate");
```

The string that is passed as an argument to the `JButton` constructor is the text that will be displayed on the button.

Next, in line 63, the method uses the following statement to create a `JPanel` object and assign its address to the `panel` field, which is a private field in the class:

```
panel = new JPanel();
```

A `JPanel` object is used to hold other components. You add a component to a `JPanel` object with the `add` method. The following code, in lines 67 through 69, adds the objects referenced by the `messageLabel`, `kiloTextField`, and `calcButton` variables to the `JPanel` object:

```
panel.add(messageLabel);
panel.add(kiloTextField);
panel.add(calcButton);
```

At this point, the panel is fully constructed in memory. The `buildPanel` method ends, and control returns to the class constructor. Here's the next statement in the constructor, which appears in line 38:

```
add(panel);
```

This statement calls the `add` method, which was inherited from the `JFrame` class. The purpose of the `add` method is to add an object to the content pane. This statement adds the object referenced by `panel` to the content pane.

The constructor's last statement, in line 41, calls the inherited `setVisible` method to display the window on the screen as follows:

```
setVisible(true);
```

The class has a static `main` method, which appears in lines 76 through 79. Line 78 creates an instance of the `KiloConverter` class. When this program is executed, the window shown in Figure 23-6 is displayed on the screen.

Figure 23-6 Kilometer Converter window (Oracle Corporate Counsel)

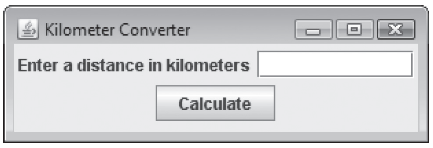
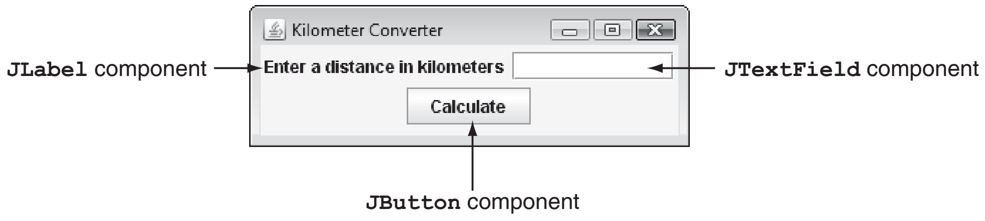


Figure 23-7 shows the window again, this time pointing out each of the components.

Although you can type input into the text field, the application does nothing when you click the Calculate button because we have not written an event handler that will execute when the button is clicked. That's the next step.

Figure 23-7 Components in the Kilometer Converter window (Oracle Corporate Counsel)



NOTE: Recall that the size of the window in the `KiloConverter` class is set to 310 pixels wide by 100 pixels high. This is set with the `WINDOW_WIDTH` and `WINDOW_HEIGHT` constants. Figures 23-6 and 23-7 show the window as it appears on a system set at a video resolution of 1024 by 768 pixels. If your video resolution is lower, the window might not appear exactly as shown in the figures. If this is the case, you can increase the values of the `WINDOW_WIDTH` and `WINDOW_HEIGHT` constants and recompile the program. This is true for other applications in this chapter as well.

Handling Events with Action Listeners

VideoNote
Handling Events

An *event* is an action that takes place within a program, such as the clicking of a button. When an event takes place, the component that is responsible for the event creates an event object in memory. The *event object* contains information about the event. The component that generated the event object is known as the *event source*. For example, when the user clicks a button, the `JButton` component generates an event object. The `JButton` component that generated the event object is the event source.

But what happens to the event object once it is generated by a source component? It is possible that the source component is connected to one or more event listeners. An *event listener* is an object that responds to events. If the source component is connected to an event listener, then the event object is automatically passed, as an argument, to a specific method in the event listener. The method then performs any actions that it was programmed to perform in response to the event. This process is sometimes referred to as *event firing*.

When you are writing a GUI application, it is your responsibility to write the classes for the event listeners that your application needs. For example, if you write an application with a `JButton` component, an event will be generated each time the user clicks the button. Therefore, you should write an event listener class that can handle the event. In your application you would create an instance of the event listener class and connect it to the `JButton` component. Before looking at a specific example, we must discuss two important topics that arise when writing event listeners: private inner classes and interfaces.

Writing Event Listener Classes as Private Inner Classes

Java allows you to write a class definition inside of another class definition. A class that is defined inside of another class is known as an *inner class*. Figure 23-8 illustrates a class definition inside of another class definition.

Figure 23-8 A class with an inner class

```
public class Outer
{
    Fields and methods of the Outer
    class appear here.

    private class Inner
    {
        Fields and methods of the Inner
        class appear here.
    }
}
```

When an inner class is private, as shown in the figure, it is accessible only to code in the class that contains it. For example, the `Inner` class shown in the figure would be accessible only to methods that belong to the `Outer` class. Code outside the `Outer` class would not be able to access the `Inner` class. A common technique for writing an event listener class is to write it as a private inner class, inside the class that creates the GUI. Although this is not the only way to write event listener classes, it is the approach we take in this book.

Event Listeners Must Implement an Interface

There is a special requirement that all event listener classes must meet: They must *implement an interface*.

We discussed interfaces in detail in Chapter 10, but in case you haven't read that material, you can think of an interface as something like a class, containing one or more method headers. Interfaces do not have actual methods, however, only their headers. When you write a class that implements an interface, you are agreeing that the class will have all of the methods that are specified in the interface.

Java provides numerous interfaces that you can use with event listener classes. There are several different types of events that can occur within a GUI application, and the specific interface that you use depends on the type of event you want to handle. `JButton` components generate *action events*, and an event listener class that can handle action events is also known as an *action listener* class. When you write an action listener class for a `JButton` component, it must implement an interface known as `ActionListener`. In case you are curious, this is what the code for the `ActionListener` interface looks like:

```
public interface ActionListener
{
   public void actionPerformed(ActionEvent e);
}
```

As you can see, the `ActionListener` interface contains the header for only one method: `actionPerformed`. Notice that the method has `public` access, is `void`, and has a parameter of the `ActionEvent` type. When you write a class that implements this interface, it *must* have a method named `actionPerformed`, with a header exactly like the one in the interface.

NOTE: The *ActionListener* interface, as well as other event listener interfaces, is in the *java.awt.event* package. We will use the following *import* statement in order to use those interfaces:

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

You use the `implements` key word in a class header to indicate that it implements an interface. Here is an example of a class named `MyButtonListener` that implements the `ActionListener` interface:

```
private class MyButtonListener implements ActionListener
{
   public void actionPerformed(ActionEvent e)
   {
      Write code here to handle the event.
   }
}
```

Remember, when you write a class that implements an interface, you are "promising" that the class will have the methods specified in the interface. Notice that this class lives up to its promise. It has a method named `actionPerformed`, with a header that matches the `actionPerformed` header in the `ActionListener` interface exactly.

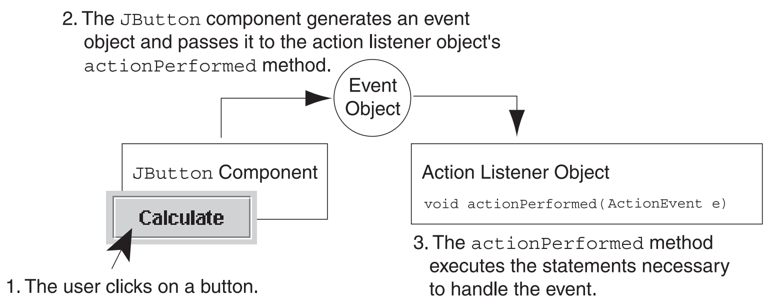
NOTE: In your action listener class, the only part of the `actionPerformed` method header that does not have to match that which is shown in the `ActionListener` interface exactly is the name of the parameter variable. Instead of using the name `e`, you can use any legal variable name that you wish.

Registering an Event Listener Object

Once you have written an event listener class, you can create an object of that class, and then connect the object with a GUI component. The process of connecting an event listener object to a GUI component is known as *registering* the event listener.

When a `JButton` component generates an event, it automatically executes the `actionPerformed` method of the event listener object that is registered with it, passing the event object as an argument. This is illustrated in Figure 23-9.

Figure 23-9 A `JButton` component firing an action event (Oracle Corporate Counsel)


2. The JButton component generates an event object and passes it to the action listener object's actionPerformed method.
Event Object
JButton Component
Calculate
Action Listener Object
void actionPerformed(ActionEvent e)
3. The actionPerformed method executes the statements necessary to handle the event.
1. The user clicks on a button.

Writing an Event Listener for the `KiloConverter` Class

Now that we've gone over the basics of event listeners, let's continue to develop the `KiloConverter` class. Code Listing 23-6 shows the class with an action listener added to it. This version of the class is stored in the source code folder *Chapter 23\KiloConverter Phase 2*. The action listener is a private inner class named `CalcButtonListener`. The new code is shown in bold.

Code Listing 23-6 (`KiloConverter.java`)

```
import javax.swing.*;     // Needed for Swing classes
import java.awt.event.*; // Needed for ActionListener Interface

/**
   The KiloConverter class displays a JFrame that
   lets the user enter a distance in kilometers. When
   the Calculate button is clicked, a dialog box is
   displayed with the distance converted to miles.
```

```
*/

public class KiloConverter extends JFrame
{
   private JPanel panel;             // To reference a panel
   private JLabel messageLabel;      // To reference a label
   private JTextField kiloTextField; // To reference a text field
   private JButton calcButton;       // To reference a button
   private final int WINDOW_WIDTH = 310;  // Window width
   private final int WINDOW_HEIGHT = 100; // Window height

   /**
      Constructor
   */

   public KiloConverter()
   {
      // Set the window title.
      setTitle("Kilometer Converter");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

      // Specify what happens when the close button is clicked.
      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();

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

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

```
105        }
106     }
107
108     /**
109        main method
110     */
111
112     public static void main(String[] args)
113     {
114        new KiloConverter();
115     }
116  }
```

First, notice that we've added the `import java.awt.event.*;` statement in line 2. This is necessary for our program to use the `ActionListener` interface. Next, look at the following code in line 81:

```
private class CalcButtonListener implements ActionListener
```

This is the header for an inner class that we will use to create event listener objects. The name of this class is `CalcButtonListener` and it implements the `ActionListener` interface. We could have named the class anything we wanted to, but because it will handle the `JButton` component's action events, it must implement the `ActionListener` interface. The class has one method, `actionPerformed`, which is required by the `ActionListener` interface. The header for the `actionPerformed` method appears in line 89 as follows:

```
public void actionPerformed(ActionEvent e)
```

This method will be executed when the user clicks the `JButton` component. It has one parameter, `e`, which is an `ActionEvent` object. This parameter receives the event object that is passed to the method when it is called. Although we do not actually use the `e` parameter in this method, we still have to list it inside the method header's parentheses because it is required by the `ActionListener` interface.

The `actionPerformed` method declares a constant for the conversion factor in line 91, and two local variables in lines 92 and 93: `input`, a reference to a `String` object; and `miles`, a `double`. The following statement appears in line 97:

```
input = kiloTextField.getText();
```

All `JTextField` objects have a `getText` method that returns the text contained in the text field. This will be any value entered into the text field by the user. The value is returned as a string. So, this statement retrieves any value entered by the user into the text field and assigns it to `input`.

The following statement appears in line 100:

```
miles = Double.parseDouble(input) * CONVERSION;
```

This statement converts the value in `input` to a `double`, and then multiplies it by the constant `CONVERSION`, which is set to 0.6214. This will convert the number of kilometers entered by the user to miles. The result is stored in the `miles` variable. The method's last statement, in lines 103 and 104, uses `JOptionPane` to display a dialog box showing the distance converted to miles as follows:

```
JOptionPane.showMessageDialog(null, input +
          " kilometers is " + miles + " miles.");
```

Writing an action listener class is only part of the process of handling a `JButton` component's action events. We must also create an object from the class and then register the object with the `JButton` component. When we register the action listener object with the `JButton` component, we are creating a connection between the two objects.

`JButton` components have a method named `addActionListener`, which is used for registering action event listeners. In line 63, which is in the `buildPanel` method, the following statement creates a `CalcButtonListener` object and registers that object with the `calcButton` object:

```
calcButton.addActionListener(new CalcButtonListener());
```

You pass the address of an action listener object as the argument to the `addActionListener` method. This statement uses the expression `new CalcButtonListener()` to create an instance of the `CalcButtonListener` class. The address of that instance is then passed to the `addActionListener` method. Now, when the user clicks the Calculate button, the `CalcButtonListener` object's `actionPerformed` method will be executed.

TIP: Instead of the one statement in line 63, we could have written the following two statements:

```
CalcButtonListener listener = new CalcButtonListener();
calcButton.addActionListener(listener);
```

The first statement shown here declares a `CalcButtonListener` variable named `listener`, creates a new `CalcButtonListener` object, and assigns the object's address to the `listener` variable. The second statement passes the address in listener to the `addActionListener` method. These two statements accomplish the same thing as the one statement in line 63, but they declare a variable, `listener`, that we will not use again in the program. A better way is to use the one statement that appears in line 63 as follows:

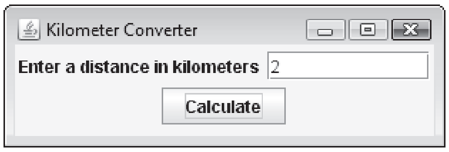
```
calcButton.addActionListener(new CalcButtonListener());
```

Recall that the `new` key word creates an object and returns the object's address. This statement uses the `new` key word to create a `CalcButtonListener` object, and passes the object's address directly to the `addActionListener` method. Because we do not need to refer to the object again in the program, we do not assign the object's address to a variable. It is known as an *anonymous object*.

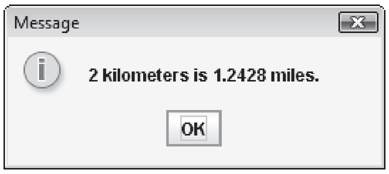
When this program is executed, the first window shown in Figure 23-10 is displayed on the screen. If the user enters 2 in the text field and clicks the Calculate button, the second window shown in the figure (a dialog box) appears. To exit the application, the user clicks the OK button on the dialog box, and then clicks the close button in the upper-right corner of the main window.

Figure 23-10 Windows displayed by the KiloConverter class (Oracle Corporate Counsel)

This window appears first. The user enters 2 in the text field and then clicks the Calculate button.



This dialog box appears next.



2 kilometers is 1.2428 miles.

Background and Foreground Colors

Many of the Swing component classes have methods named `setBackground` and `setForeground`. You call these methods to change a component's color. The background color is the color of the component itself, and the foreground color is the color of text that might be displayed on the component.

The argument that you pass to the `setBackground` and `setForeground` methods is a color code. Table 23-3 lists several predefined constants that you can use for colors. To use these constants, you must have the `import java.awt.*;` statement in your code.

Table 23-3 Color constants (Oracle Corporate Counsel)

`Color.BLACK`	`Color.BLUE`
`Color.CYAN`	`Color.DARK_GRAY`
`Color.GRAY`	`Color.GREEN`
`Color.LIGHT_GRAY`	`Color.MAGENTA`
`Color.ORANGE`	`Color.PINK`
`Color.RED`	`Color.WHITE`
`Color.YELLOW`	

For example, the following code creates a button with the text "OK" displayed on it. The `setBackground` and `setForeground` methods are called to make the button blue and the text yellow.

```
JButton okButton = new JButton("OK");
okButton.setBackground(Color.BLUE);
okButton.setForeground(Color.YELLOW);
```

The `ColorWindow` class in Code Listing 23-7 displays a window with a label and three buttons. When the user clicks a button, it changes the background color of the panel that contains the components and the foreground color of the label.

Code Listing 23-7 (ColorWindow.java)

```java
import javax.swing.*;    // Needed for Swing classes
import java.awt.*;       // Needed for Color class
import java.awt.event.*; // Needed for event listener interface

/**
   This class demonstrates how to set the background color of
   a panel and the foreground color of a label.
*/

public class ColorWindow extends JFrame
{
   private JLabel messageLabel;    // To display a message
   private JButton redButton;      // Changes color to red
   private JButton blueButton;     // Changes color to blue
   private JButton yellowButton;   // Changes color to yellow
   private JPanel panel;           // A panel to hold components
   private final int WINDOW_WIDTH = 200;  // Window width
   private final int WINDOW_HEIGHT = 125; // Window height

   /**
      Constructor
   */

   public ColorWindow()
   {
      // Set the title bar text.
      setTitle("Colors");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

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

      // Create a label.
      messageLabel = new JLabel("Click a button to " +
                                "select a color.");

      // Create the three buttons.
      redButton = new JButton("Red");
      blueButton = new JButton("Blue");
      yellowButton = new JButton("Yellow");

      // Register an event listener with all 3 buttons.
      redButton.addActionListener(new RedButtonListener());
      blueButton.addActionListener(new BlueButtonListener());
      yellowButton.addActionListener(new YellowButtonListener());

      // Create a panel and add the components to it.
      panel = new JPanel();
      panel.add(messageLabel);
      panel.add(redButton);
      panel.add(blueButton);
      panel.add(yellowButton);

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

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

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

   private class RedButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         // Set the panel's background to red.
         panel.setBackground(Color.RED);

         // Set the label's text to blue.
         messageLabel.setForeground(Color.BLUE);
      }
   }

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

   private class BlueButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         // Set the panel's background to blue.
         panel.setBackground(Color.BLUE);
```

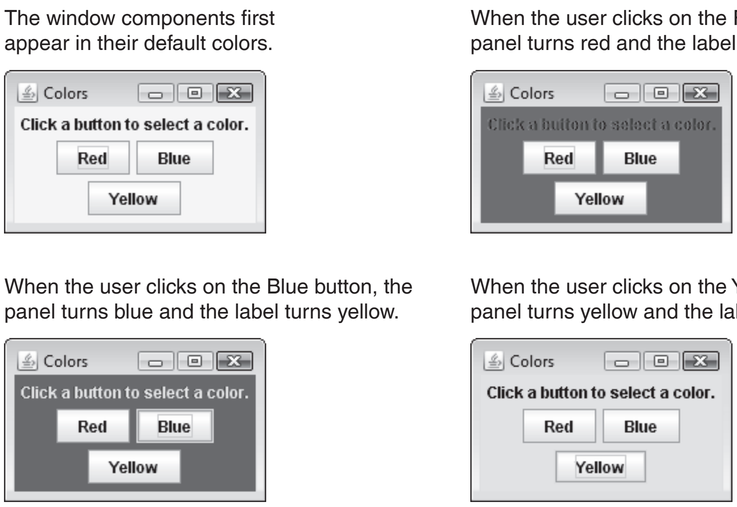
```
 91 
 92          // Set the label's text to yellow.
 93          messageLabel.setForeground(Color.YELLOW);
 94       }
 95    }
 96 
 97    /**
 98       Private inner class that handles the event when
 99       the user clicks the Yellow button.
100    */
101 
102    private class YellowButtonListener implements ActionListener
103    {
104       public void actionPerformed(ActionEvent e)
105       {
106          // Set the panel's background to yellow.
107          panel.setBackground(Color.YELLOW);
108 
109          // Set the label's text to black.
110          messageLabel.setForeground(Color.BLACK);
111       }
112    }
113 
114    /**
115       main method
116    */
117 
118    public static void main(String[] args)
119    {
120       new ColorWindow();
121    }
122 }
```

Notice that this class has three action listener classes, one for each button. The action listener classes are `RedButtonListener`, `BlueButtonListener`, and `YellowButtonListener`. The following statements, in lines 45 through 47, register instances of these classes with the appropriate button components:

```
redButton.addActionListener(new RedButtonListener());
blueButton.addActionListener(new BlueButtonListener());
yellowButton.addActionListener(new YellowButtonListener());
```

When you run the program, the window shown in Figure 23-11 appears.

Figure 23-11 The window produced by the `ColorWindow` class (Oracle Corporate Counsel)


The window components first appear in their default colors.

When the user clicks on the Red button, the panel turns red and the label turns blue.

When the user clicks on the Blue button, the panel turns blue and the label turns yellow.

When the user clicks on the Yellow button, the panel turns yellow and the label turns black.

Changing the Background Color of a `JFrame` Object's Content Pane

Recall that a `JFrame` object has a content pane, which is a container for all the components that are added to the `JFrame`. When you add a component to a `JFrame` object, you are actually adding it to the object's content pane. In the example shown in this section, we added a label and some buttons to a panel, and then added the panel to the `JFrame` object's content pane. When we changed the background color, we changed the background color of the panel. In this example, the color of the content pane does not matter because it is completely filled up by the panel. The color of the panel covers up the color of the content pane.

In some cases, where you have not filled up the `JFrame` object's content pane with a panel, you might want to change the background color of the content pane. If you wish to change the background color of a `JFrame` object's content pane, you must call the content pane's `setBackground` method, not the `JFrame` object's `setBackground` method. For example, in a class that extends the `JFrame` class, the following statement can be used to change the content pane's background to blue:

```
getContentPane().setBackground(Color.BLUE);
```

In this statement, the `getContentPane` method is called to get a reference to the `JFrame` object's content pane. This reference is then used to call the content pane's `setBackground` method. As a result, the content pane's background color will change to blue.

The `ActionEvent` Object

The action listener's `actionPerformed` method has a parameter variable named `e` that is declared as follows:

```
ActionEvent e
```

`ActionEvent` is a class that is defined in the Java API. When an action event occurs, an object of the `ActionEvent` class is created, the action listener's `actionPerformed` method is called, and a reference to the `ActionEvent` object is passed into the `e` parameter variable. So, when the `actionPerformed` method executes, the `e` parameter references the event object that was generated in response to the event.

Earlier it was mentioned that the event object contains information about the event. If you wish, you can retrieve certain information about the event by calling one of the event object's methods. Two of the `ActionEvent` methods are listed in Table 23-4.

Table 23-4 ActionEvent methods

Method Name	Description
`getActionCommand()`	Returns the action command for this event as a `String`
`getSource()`	Returns a reference to the object that generated this event

The `getActionCommand` Method

The first method listed in Table 23-4, `getActionCommand`, returns the *action command* that is associated with the event. When a `JButton` component generates an event, the action command is the text that appears on the button. The `getActionCommand` returns this text as a `String`. You can use the `getActionCommand` method to determine which button was clicked when several buttons share the same action listener class.

To demonstrate, look at the `EventObjectWindow` class in Code Listing 23-8. It produces a window with three buttons. The buttons have the text "Button 1", "Button 2", and "Button 3". The action listener class displays the contents of the event object's action command when any of these buttons are clicked.

Code Listing 23-8 (`EventObjectWindow.java`)

```java
import javax.swing.*;    // Needed for Swing classes
import java.awt.event.*; // Needed for event listener interface

/**
    This class demonstrates how to retrieve the action command
    from an event object.
*/

public class EventObject extends JFrame
{
   private JButton button1;    // Button 1
   private JButton button2;    // Button 2
   private JButton button3;    // Button 3
   private JPanel panel;       // A panel to hold components
   private final int WINDOW_WIDTH = 300; // Window width
   private final int WINDOW_HEIGHT = 70; // Window height

   /**
         Constructor
   */

   public EventObject()
   {
      // Set the title bar text.
      setTitle("Event Object Demonstration");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

      // Specify what happens when the close button is clicked.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create the three buttons.
      button1 = new JButton("Button 1");
      button2 = new JButton("Button 2");
      button3 = new JButton("Button 3");

      // Register an event listener with all 3 buttons.
      button1.addActionListener(new ButtonListener());
      button2.addActionListener(new ButtonListener());
      button3.addActionListener(new ButtonListener());

      // Create a panel and add the buttons to it.
      panel = new JPanel();
      panel.add(button1);
      panel.add(button2);
      panel.add(button3);

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

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

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

   private class ButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
```

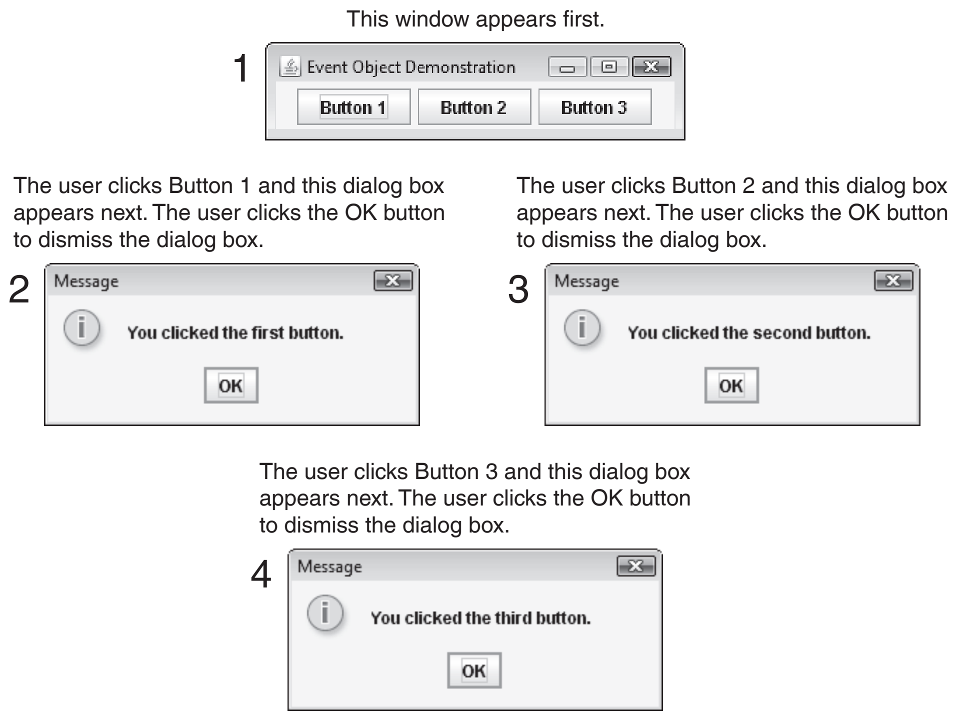
```
64       {
65          // Get the action command.
66          String actionCommand = e.getActionCommand();
67
68          // Determine which button was clicked and display
69          // a message.
70          if (actionCommand.equals("Button 1"))
71          {
72             JOptionPane.showMessageDialog(null, "You clicked " +
73                                           "the first button.");
74          }
75          else if (actionCommand.equals("Button 2"))
76          {
77             JOptionPane.showMessageDialog(null, "You clicked " +
78                                           "the second button.");
79          }
80          else if (actionCommand.equals("Button 3"))
81          {
82             JOptionPane.showMessageDialog(null, "You clicked " +
83                                           "the third button.");
84          }
85       }
86    }
87
88    /**
89       main method
90    */
91
92    public static void main(String[] args)
93    {
94       new EventObject();
95    }
96 }
```

Previously you saw the `ColorWindow` class, in Code Listing 23-7, which had three buttons and three different action listener classes. The `EventObjectWindow` class also has three buttons, but only one action listener class. In lines 39 through 41, we create and register three separate instances of the class with the three buttons as follows:

```
button1.addActionListener(new ButtonListener());
button2.addActionListener(new ButtonListener());
button3.addActionListener(new ButtonListener());
```

Figure 23-12 shows the output of the application when the user clicks Button 1, Button 2, and Button 3.

Figure 23-12 Output of `EventObjectWindow` class (Oracle Corporate Counsel)


This window appears first.

1

The user clicks Button 1 and this dialog box appears next. The user clicks the OK button to dismiss the dialog box.

2

You clicked the first button.

The user clicks Button 2 and this dialog box appears next. The user clicks the OK button to dismiss the dialog box.

3

You clicked the second button.

The user clicks Button 3 and this dialog box appears next. The user clicks the OK button to dismiss the dialog box.

4

You clicked the third button.

TIP: The text that is displayed on a button is the default action command. You can change the action command by calling the `JButton` class's `setActionCommand` method. For example, assuming that `myButton` references a `JButton` component, the following statement would change the component's action command to "The button was clicked":

```
myButton.setActionCommand("The button was clicked");
```

NOTE: Changing a `JButton` component's action command does not change the text that is displayed on the button. For a demonstration of how to change the action command, see the *ActionCommand.java* file in this chapter's source code folder.

The `getSource` Method

The second `ActionEvent` method listed in Table 23-4, `getSource`, returns a reference to the component that is the source of the event. As with the `getActionCommand` method, if you have several buttons and use objects of the same action listener class to respond to their events, you can use the `getSource` method to determine which button was clicked. For example, the `ButtonListener` class's `actionPerformed` method in Code Listing 23-8 could have been written as follows, to achieve the same result:

```
public void actionPerformed(ActionEvent e)
{
   // Determine which button was clicked and display
   // a message.
```

```
         if (e.getSource() == button1)
         {
            JOptionPane.showMessageDialog(null, "You clicked " +
                                          "the first button.");
         }
         else if (e.getSource() == button2)
         {
            JOptionPane.showMessageDialog(null, "You clicked " +
                                          "the second button.");
         }
         else if (e.getSource() == button3)
         {
            JOptionPane.showMessageDialog(null, "You clicked " +
                                          "the third button.");
         }
      }
```

See the *EventObjectWindow2.java* file in this chapter's source code folder for a demonstration of this code.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

23.1 What is a frame? How do you create a frame with Swing?

23.2 How do you set a frame's size?

23.3 How do you display a frame on the screen?

23.4 What is a content pane?

23.5 What is the difference between a frame and a panel?

23.6 What is an event listener?

23.7 If you are writing an event listener class for a `JButton` component, what interface must the class implement? What method must the class have? When is this method executed?

23.8 How do you register an event listener with a `JButton` component?

23.9 How do you change the background color of a component? How do you change the color of text displayed by a label or a button?

23.3 Layout Managers

CONCEPT: A layout manager is an object that governs the positions and sizes of components in a container. The layout manager automatically repositions and, in some cases, resizes the components when the container is resized.

An important part of designing a GUI application is determining the layout of the components that are displayed in the application's windows. The term *layout* refers to the positioning and sizing of components. In Java, you do not normally specify the exact location of a component within a window. Instead, you let a layout manager control the positions of components for you. A *layout manager* is an object that has its own rules about how components are to be positioned and sized, and it makes adjustments when necessary. For example, when the user resizes a window, the layout manager determines where the components should be moved to.

In order to use a layout manager with a group of components, you must place the components in a container, and then create a layout manager object. The layout manager object and the container work together. In this chapter we discuss the three layout managers described in Table 23-5. To use any of these classes, your code should have the following `import` statement: `import java.awt.*;`

Table 23-5 Layout managers

Layout Manager	Description
`FlowLayout`	Arranges components in rows; this is the default layout manager for `JPanel` objects
`BorderLayout`	Arranges components in five regions: north, south, east, west, and center; this is the default layout manager for a `JFrame` object's content pane
`GridLayout`	Arranges components in a grid with rows and columns

Adding a Layout Manager to a Container

You add a layout manager to a container, such as a content pane or a panel, by calling the `setLayout` method and passing a reference to a layout manager object as the argument. For example, the following code creates a `JPanel` object, then sets a `BorderLayout` object as its layout manager:

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

Likewise, the following code might appear in the constructor of a class that extends the `JFrame` class. It sets a `FlowLayout` object as the layout manager for the content pane:

```
setLayout(new FlowLayout());
```

Once you establish a layout manager for a container, the layout manager governs the positions and sizes of the components that are added to the container.

The `FlowLayout` Manager

The `FlowLayout` manager arranges components in rows. This is the default layout manager for `JPanel` objects. Here are some rules that the `FlowLayout` manager follows:

- You can add multiple components to a container that uses a `FlowLayout` manager.
- When you add components to a container that uses a `FlowLayout` manager, the components appear horizontally, from left to right, in the order that they were added to the component.
- When there is no more room in a row but more components are added, the new components "flow" to the next row.

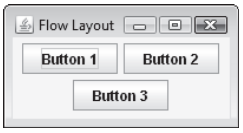
For example, the `FlowWindow` class shown in Code Listing 23-9 extends `JFrame`. This class creates a 200 pixel wide by 105 pixel high window. In the constructor, the `setLayout` method is called to give the content pane a `FlowLayout` manager. Then, three buttons are created and added to the content pane. The `main` method creates an instance of the `FlowWindow` class, which displays the window.

Code Listing 23-9 (`FlowWindow.java`)

```
import javax.swing.*; // Needed for Swing classes
import java.awt.*;    // Needed for FlowLayout class

/**
   This class demonstrates how to use a FlowLayout manager
   with the content pane.
*/

public class FlowWindow extends JFrame
{
   private final int WINDOW_WIDTH = 200;  // Window width
   private final int WINDOW_HEIGHT = 105; // Window height

   /**
      Constructor
   */

   public FlowWindow()
   {
      // Set the title bar text.
      setTitle("Flow Layout");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

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

      // Add a FlowLayout manager to the content pane.
      setLayout(new FlowLayout());

      // Create three buttons.
      JButton button1 = new JButton("Button 1");
      JButton button2 = new JButton("Button 2");
      JButton button3 = new JButton("Button 3");

      // Add the three buttons to the content pane.
      add(button1);
      add(button2);
      add(button3);

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

   /**
      The main method creates an instance of the FlowWindow
      class, causing it to display its window.
   */

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

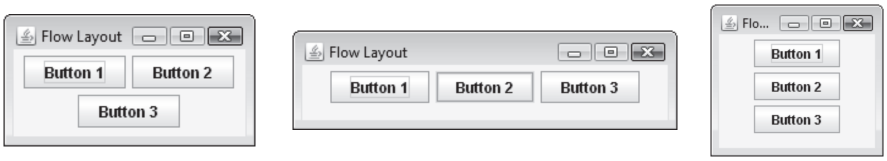
Figure 23-13 shows the window that is displayed by this class. Notice that the buttons appear from left to right in the order they were added to the content pane. Because there is only enough room for the first two buttons in the first row, the third button is positioned in the second row. By default, the content of each row is centered and there is a five pixel gap between the components.

Figure 23-13 The window displayed by the `FlowWindow` class (Oracle Corporate Counsel)



If the user resizes the window, the layout manager repositions the components according to its rules. Figure 23-14 shows the appearance of the window in three different sizes.

Figure 23-14 The arrangements of the buttons after resizing



Adjusting the `FlowLayout` Alignment

The `FlowLayout` manager allows you to align components in the center of each row or along the left or right edge of each row. An overloaded constructor allows you to pass one of the following constants as an argument to set an alignment: `FlowLayout.CENTER`, `FlowLayout.LEFT`, or `FlowLayout.RIGHT`. Here is an example that sets left alignment:

```
setLayout(new FlowLayout(FlowLayout.LEFT));
```

Figure 23-15 shows examples of windows that use a `FlowLayout` manager with left, center, and right alignment.

Adjusting the `FlowLayout` Component Gaps

By default, the `FlowLayout` manager inserts a gap of five pixels between components, both horizontally and vertically. You can adjust this gap by passing values for the horizontal and vertical gaps as arguments to an overloaded `FlowLayout` constructor. The constructor has the following format:

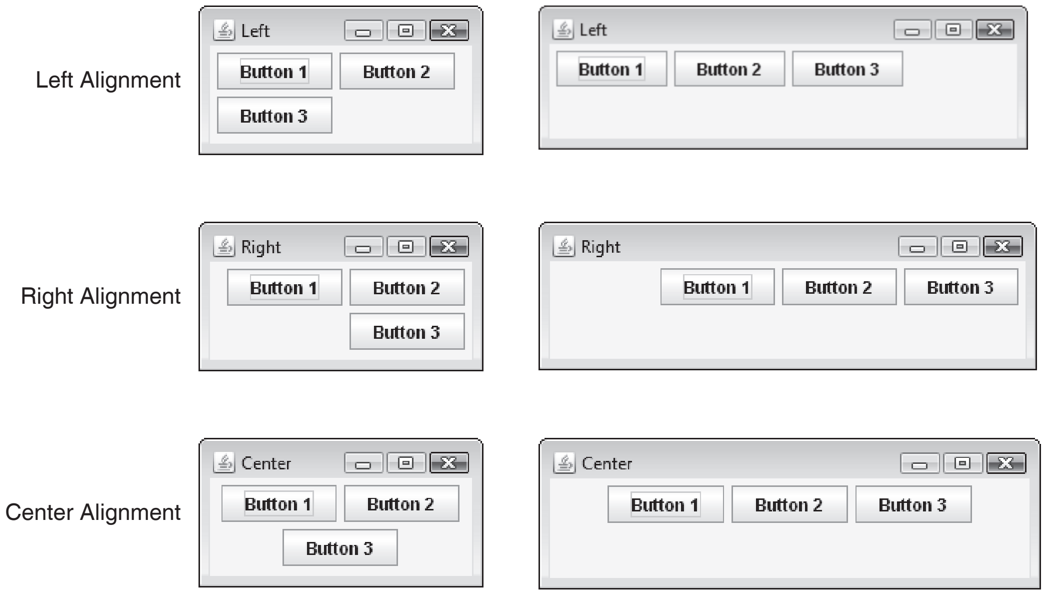
```
FlowLayout(int alignment, int horizontalGap, int verticalGap)
```

You pass one of the alignment constants discussed in the previous section to the `alignment` parameter. The `horizontalGap` parameter is the number of pixels to separate components horizontally, and the `verticalGap` parameter is the number of pixels to separate components vertically. Here is an example of the constructor call:

```
setLayout(new FlowLayout(FlowLayout.LEFT, 10, 7));
```

This statement causes components to be left aligned with a horizontal gap of 10 pixels and a vertical gap of seven pixels.

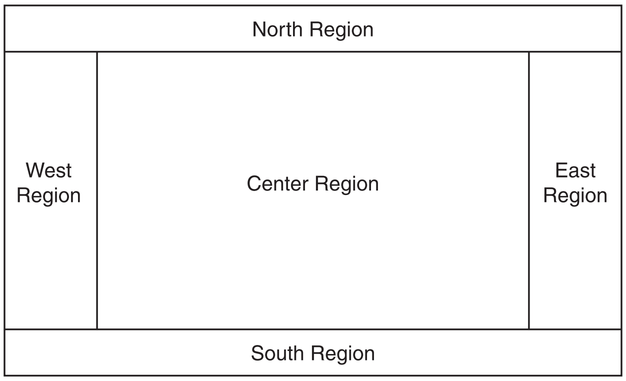
Figure 23-15 Left, center, and right alignment (Oracle Corporate Counsel)



The `BorderLayout` Manager

The `BorderLayout` manager divides a container into five regions. The regions are known as north, south, east, west, and center. The arrangement of these regions is shown in Figure 23-16.

Figure 23-16 The regions of a BorderLayout manager (Oracle Corporate Counsel)



When a component is placed into a container that is managed by a BorderLayout manager, the component must be placed into one of these five regions. Only one component at a time may be placed into a region. When adding a component to the container, you specify the region by passing one of the following constants as a second argument to the container's add method: BorderLayout.NORTH, BorderLayout.SOUTH, BorderLayout.EAST, BorderLayout.WEST, or BorderLayout.CENTER.

For example, look at the following code:

```
JPanel panel = new JPanel();
JButton button = new JButton("Click Me");
panel.setLayout(new BorderLayout());
panel.add(button, BorderLayout.NORTH);
```

The first statement creates a JPanel object, referenced by the panel variable. The second statement creates a JButton object, referenced by the button variable. The third statement sets the JPanel object's layout manager to a BorderLayout object. The fourth statement adds the JButton object to the JPanel object's north region.

If you do not pass a second argument to the add method, the component will be added to the center region. Here are some rules that the BorderLayout manager follows:

- Each region can hold only one component at a time.
- When a component is added to a region, the component is stretched so it fills up the entire region.

Look at the BorderWindow class shown in Code Listing 23-10, which extends JFrame. This class creates a 400 pixel wide by 300 pixel high window. In the constructor, the setLayout method is called to give the content pane a BorderLayout manager. Then, five buttons are created and each is added to a different region.

Code Listing 23-10 (BorderWindow.java)

```
import javax.swing.*; // Needed for Swing classes
import java.awt.*;    // Needed for BorderLayout class
```

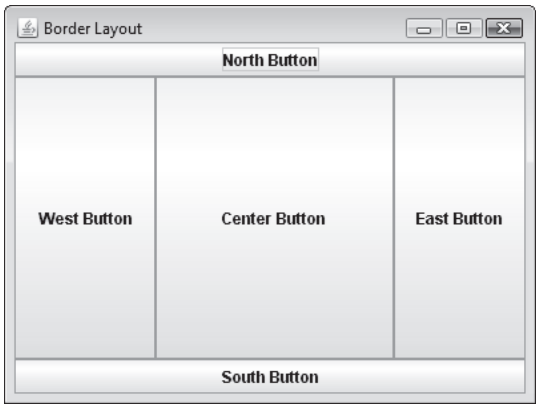
```

/**
   This class demonstrates the BorderLayout manager.
*/

public class BorderWindow extends JFrame
{
   private final int WINDOW_WIDTH = 400;   // Window width
   private final int WINDOW_HEIGHT = 300;  // Window height

   /**
      Constructor
   */

   public BorderWindow()
   {
      // Set the title bar text.
      setTitle("Border Layout");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

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

      // Add a BorderLayout manager to the content pane.
      setLayout(new BorderLayout());

      // Create five buttons.
      JButton button1 = new JButton("North Button");
      JButton button2 = new JButton("South Button");
      JButton button3 = new JButton("East Button");
      JButton button4 = new JButton("West Button");
      JButton button5 = new JButton("Center Button");

      // Add the five buttons to the content pane.
      add(button1, BorderLayout.NORTH);
      add(button2, BorderLayout.SOUTH);
      add(button3, BorderLayout.EAST);
      add(button4, BorderLayout.WEST);
      add(button5, BorderLayout.CENTER);

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

   /**
```

```
      The main method creates an instance of the BorderWindow
      class, causing it to display its window.
   */

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

NOTE: A `JFrame` object's content pane is automatically given a `BorderLayout` manager. We have explicitly added it in Code Listing 23-10 so it is clear that we are using a `BorderLayout` manager.

Figure 23-17 shows the window that is displayed. Normally the size of a button is just large enough to accommodate the text that is displayed on the button. Notice that the buttons displayed in this window did not retain their normal size. Instead, they were stretched to fill all of the space in their regions. If the user resizes the window, the sizes of the components will be changed as well. This is shown in Figure 23-18.

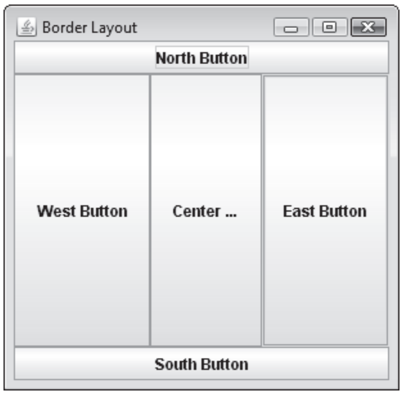
Figure 23-17 The window displayed by the `BorderWindow` class (Oracle Corporate Counsel)



Here are the rules that govern how a `BorderLayout` manager resizes components:

- A component that is placed in the north or south regions may be resized horizontally so it fills up the entire region.
- A component that is placed in the east or west regions may be resized vertically so it fills up the entire region.

Figure 23-18 The window resized (Oracle Corporate Counsel)



- A component that is placed in the center region may be resized both horizontally and vertically so it fills up the entire region.

TIP: You do not have to place a component in every region of a border layout. To achieve the desired positioning, you might want to place components in only a few of the layout regions. In Chapter 24, you will see examples of applications that do this.

By default there is no gap between the regions. You can use an overloaded version of the `BorderLayout` constructor to specify horizontal and vertical gaps, however. Here is the constructor's format:

```
BorderLayout(int horizontalGap, int verticalGap)
```

The `horizontalGap` parameter is the number of pixels to separate the regions horizontally, and the `verticalGap` parameter is the number of pixels to separate the regions vertically. Here is an example of the constructor call:

```
setLayout(new BorderLayout(5, 10));
```

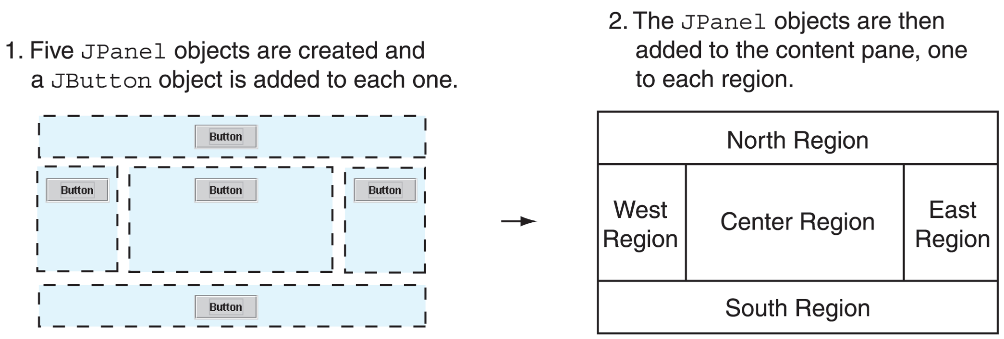
This statement causes the regions to appear with a horizontal gap of five pixels and a vertical gap of 10 pixels.

Nesting Panels Inside a Container's Regions

You might think that the `BorderLayout` manager is limiting because it allows only one component per region, and the components that are placed in its regions are automatically resized to fill up any extra space. These limitations are easy to overcome, however, by adding components to panels and then nesting the panels inside the regions.

For example, suppose we wish to modify the `BorderWindow` class in Code Listing 23-10 so the buttons retain their original size. We can accomplish this by placing each button in a separate `JPanel` object and then adding the `JPanel` objects to the content pane's five regions. This is illustrated in Figure 23-19. As a result, the `BorderLayout` manager resizes the `JPanel` objects to fill up the space in the regions, not the buttons contained within the `JPanel` objects.

Figure 23-19 Nesting JPanel objects inside each region

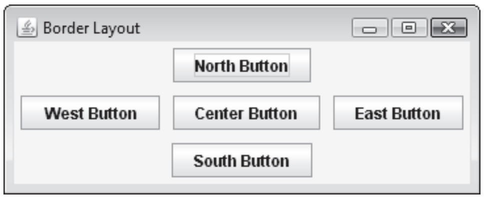

1. Five JPanel objects are created and a JButton object is added to each one.
Button
Button
Button
Button
Button
2. The JPanel objects are then added to the content pane, one to each region.
North Region
West Region
Center Region
East Region
South Region

The BorderPanelWindow class in Code Listing 23-11 demonstrates this technique. This class also introduces a new way of sizing windows. Notice that the constructor does not explicitly set the size of the window with the setSize method. Instead, it calls the pack method just before calling the setVisible method. The pack method, which is inherited from JFrame, automatically sizes the window to accommodate the components contained within it. Figure 23-20 shows the window that the class displays.

Code Listing 23-11 (BorderPanelWindow.java)

```
import java.awt.*;      // Needed for BorderLayout class
import javax.swing.*;   // Needed for Swing classes

/**
   This class demonstrates how JPanels can be nested
   inside each region of a content pane governed by
   a BorderLayout manager.
*/

public class BorderPanelWindow extends JFrame
{
   /**
      Constructor
   */

   public BorderPanelWindow()
   {
      // Set the title bar text.
      setTitle("Border Layout");

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

      // Add a BorderLayout manager to the content pane.
      setLayout(new BorderLayout());

      // Create five panels.
      JPanel panel1 = new JPanel();
      JPanel panel2 = new JPanel();
      JPanel panel3 = new JPanel();
      JPanel panel4 = new JPanel();
      JPanel panel5 = new JPanel();

      // Create five buttons.
      JButton button1 = new JButton("North Button");
      JButton button2 = new JButton("South Button");
      JButton button3 = new JButton("East Button");
      JButton button4 = new JButton("West Button");
      JButton button5 = new JButton("Center Button");

      // Add the buttons to the panels.
      panel1.add(button1);
      panel2.add(button2);
      panel3.add(button3);
      panel4.add(button4);
      panel5.add(button5);

      // Add the five panels to the content pane.
      add(panel1, BorderLayout.NORTH);
      add(panel2, BorderLayout.SOUTH);
      add(panel3, BorderLayout.EAST);
      add(panel4, BorderLayout.WEST);
      add(panel5, BorderLayout.CENTER);

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

   /**
      The main method creates an instance of the
      BorderPanelWindow class, causing it to display
      its window.
   */

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

Figure 23-20 Window displayed by the `BorderPanelWindow` class (Oracle Corporate Counsel)

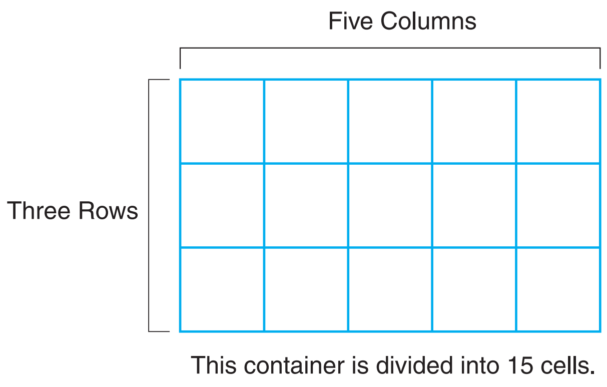


NOTE: There are multiple layout managers at work in the `BorderPanelWindow` class. The content pane uses a `BorderLayout` manager, and each of the `JPanel` objects use a `FlowLayout` manager.

The `GridLayout` Manager

The `GridLayout` manager creates a grid with rows and columns, much like a spreadsheet. As a result, the container that is managed by a `GridLayout` object is divided into equally sized cells. Figure 23-21 illustrates a container with three rows and five columns. This means that the container is divided into 15 cells.

Figure 23-21 The `GridLayout` manager divides a container into cells



Here are some rules that the `GridLayout` manager follows:

- Each cell can hold only one component.
- All of the cells are the same size. This is the size of the largest component placed within the layout.
- A component that is placed in a cell is automatically resized to fill up any extra space.

You pass the number of rows and columns that a container should have as arguments to the `GridLayout` constructor. Here is the general format of the constructor:

```
GridLayout(int rows, int columns)
```

Here is an example of the constructor call:

```
setLayout(new GridLayout(2, 3));
```

This statement gives the container two rows and three columns, for a total of six cells. You can pass 0 as an argument for the rows or the columns, but not both. Passing 0 for both arguments will cause an error.

When adding components to a container that is governed by the `GridLayout` manager, you cannot specify a cell. Instead, the components are assigned to cells in the order they are added. The first component added to the container is assigned to the first cell, which is in the upper-left corner. As other components are added, they are assigned to the remaining cells in the first row, from left to right. When the first row is filled up, components are assigned to the cells in the second row, and so forth.

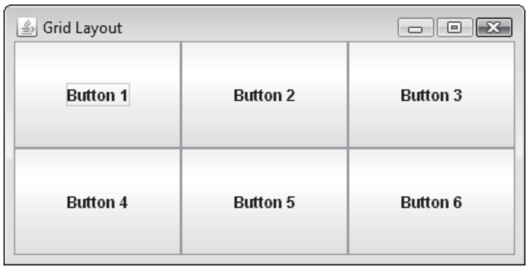
The `GridWindow` class shown in Code Listing 23-12 demonstrates. It creates a 400 pixel wide by 200 pixel high window, governed by a `GridLayout` manager. The content pane is divided into two rows and three columns, and a button is added to each cell. Figure 23-22 shows the window displayed by the class.

Code Listing 23-12 (`GridWindow.java`)

```
import javax.swing.*; // Needed for Swing classes
import java.awt.*;    // Needed for GridLayout class

/**
   This class demonstrates the GridLayout manager.
*/

public class GridWindow extends JFrame
{
   private final int WINDOW_WIDTH = 400;   // Window width
   private final int WINDOW_HEIGHT = 200;  // Window height

   /**
      Constructor
   */

   public GridWindow()
   {
      // Set the title bar text.
      setTitle("Grid Layout");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

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

      // Add a GridLayout manager to the content pane.
      setLayout(new GridLayout(2, 3));

      // Create six buttons.
      JButton button1 = new JButton("Button 1");
      JButton button2 = new JButton("Button 2");
      JButton button3 = new JButton("Button 3");
      JButton button4 = new JButton("Button 4");
      JButton button5 = new JButton("Button 5");
      JButton button6 = new JButton("Button 6");

      // Add the six buttons to the content pane.
      add(button1);  // Goes into row 1, column 1
      add(button2);  // Goes into row 1, column 2
      add(button3);  // Goes into row 1, column 3
      add(button4);  // Goes into row 2, column 1
      add(button5);  // Goes into row 2, column 2
      add(button6);  // Goes into row 2, column 3

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

   /**
      The main method creates an instance of the GridWindow
      class, causing it to display its window.
   */

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

As previously mentioned, the `GridLayout` manager limits each cell to only one component and resizes components to fill up all of the space in a cell. To get around these limitations you can nest panels inside the cells and add other components to the panels. For example, the `GridPanelWindow` class shown in Code Listing 23-13 is a modification of the `GridWindow` class. It creates six panels and adds a button and a label to each panel. These panels are then added to the content pane's cells. Figure 23-23 shows the window displayed by this class.

Figure 23-22 Window displayed by the `GridWindow` class (Oracle Corporate Counsel)

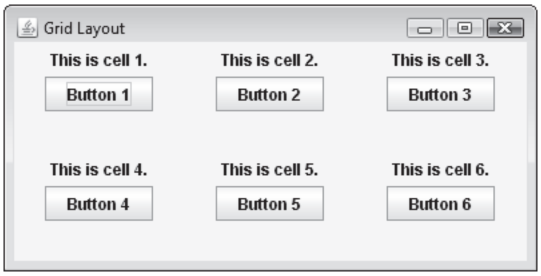


Code Listing 23-13 (`GridPanelWindow.java`)

```java
import javax.swing.*; // Needed for Swing classes
import java.awt.*;    // Needed for GridLayout class

/**
   This class demonstrates how panels may be added to
   the cells created by a GridLayout manager.
*/

public class GridPanelWindow extends JFrame
{
   private final int WINDOW_WIDTH = 400;   // Window width
   private final int WINDOW_HEIGHT = 200;  // Window height

   /**
      Constructor
   */

   public GridPanelWindow()
   {
      // Set the title bar text.
      setTitle("Grid Layout");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

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

      // Add a GridLayout manager to the content pane.
      setLayout(new GridLayout(2, 3));

      // Create six buttons.
      JButton button1 = new JButton("Button 1");
      JButton button2 = new JButton("Button 2");
      JButton button3 = new JButton("Button 3");
      JButton button4 = new JButton("Button 4");
      JButton button5 = new JButton("Button 5");
      JButton button6 = new JButton("Button 6");

      // Create six labels.
      JLabel label1 = new JLabel("This is cell 1.");
      JLabel label2 = new JLabel("This is cell 2.");
      JLabel label3 = new JLabel("This is cell 3.");
      JLabel label4 = new JLabel("This is cell 4.");
      JLabel label5 = new JLabel("This is cell 5.");
      JLabel label6 = new JLabel("This is cell 6.");

      // Create six panels.
      JPanel panel1 = new JPanel();
      JPanel panel2 = new JPanel();
      JPanel panel3 = new JPanel();
      JPanel panel4 = new JPanel();
      JPanel panel5 = new JPanel();
      JPanel panel6 = new JPanel();

      // Add the labels to the panels.
      panel1.add(label1);
      panel2.add(label2);
      panel3.add(label3);
      panel4.add(label4);
      panel5.add(label5);
      panel6.add(label6);

      // Add the buttons to the panels.
      panel1.add(button1);
      panel2.add(button2);
      panel3.add(button3);
      panel4.add(button4);
      panel5.add(button5);
      panel6.add(button6);

      // Add the panels to the content pane.
      add(panel1);  // Goes into row 1, column 1
      add(panel2);  // Goes into row 1, column 2
      add(panel3);  // Goes into row 1, column 3
      add(panel4);  // Goes into row 2, column 1
      add(panel5);  // Goes into row 2, column 2
      add(panel6);  // Goes into row 2, column 3

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

```
   }

   /**
      The main method creates an instance of the
      GridPanelWindow class, displaying its window.
   */

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

Figure 23-23 Window displayed by the `GridPanelWindow` class (Oracle Corporate Counsel)



Because we have containers nested inside the content pane, there are multiple layout managers at work in the `GridPanelWindow` class. The content pane uses a `GridLayout` manager, and each of the `JPanel` objects uses a `FlowLayout` manager.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

23.10 How do you add a layout manager to a container?

23.11 Which layout manager divides a container into regions known as north, south, east, west, and center?

23.12 Which layout manager arranges components in a row, from left to right, in the order they were added to the container?

23.13 Which layout manager arranges components in rows and columns?

23.14 How many components can you have at one time in a `BorderLayout` region? In a `GridLayout` cell?

23.15 How do you prevent the `BorderLayout` manager from resizing a component that has been placed in its region?

23.16 How can you cause a content pane to be automatically sized to accommodate the components contained within it?

23.17 What is the default layout manager for a `JFrame` object's content pane? For a `JPanel` object?

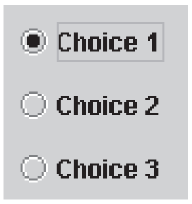
23.4 Radio Buttons and Check Boxes

CONCEPT: Radio buttons normally appear in groups of two or more and allow the user to select one of several possible options. Check boxes, which may appear alone or in groups, allow the user to make yes/no or on/off selections.

Radio Buttons

Radio buttons are useful when you want the user to select one choice from several possible options. Figure 23-24 shows a group of radio buttons.

Figure 23-24 Radio buttons (Oracle Corporate Counsel)



A radio button may be selected or deselected. Each radio button has a small circle that appears filled in when the radio button is selected and appears empty when the radio button is deselected. You use the `JRadioButton` class to create radio buttons. Here are the general formats of two `JRadioButton` constructors:

```
JRadioButton(String text)
JRadioButton(String text, boolean selected)
```

The first constructor shown creates a deselected radio button. The argument passed to the `text` parameter is the string that is displayed next to the radio button. For example, the following statement creates a radio button with the text "Choice 1" displayed next to it. The radio button initially appears deselected.

```
JRadioButton radio1 = new JRadioButton("Choice 1");
```

The second constructor takes an additional `boolean` argument, which is passed to the `selected` parameter. If `true` is passed as the `selected` argument, the radio button initially appears selected. If `false` is passed, the radio button initially appears deselected. For example, the following statement creates a radio button with the text "Choice 1" displayed next to it. The radio button initially appears selected.

```
JRadioButton radio1 = new JRadioButton("Choice 1", true);
```

Radio buttons are normally grouped together. When a set of radio buttons are grouped together, only one of the radio buttons in the group may be selected at any time. Clicking a radio button selects it and automatically deselects any other radio button in the same group. Because only one radio button in a group can be selected at any given time, the buttons are said to be *mutually exclusive*.

NOTE: The name "radio button" refers to the old car radios that had push buttons for selecting stations. Only one of the buttons could be pushed in at a time. When you pushed a button in, it automatically popped out any other button that was pushed in.

Grouping with the `ButtonGroup` class

Once you have created the `JRadioButton` objects that you wish to appear in a group, you must create an instance of the `ButtonGroup` class, and then add the `JRadioButton` objects to it. The `ButtonGroup` object creates the mutually exclusive relationship among the radio buttons that it contains. The following code shows an example:

```
// Create three radio buttons.
JRadioButton radio1 = new JRadioButton("Choice 1", true);
JRadioButton radio2 = new JRadioButton("Choice 2");
JRadioButton radio3 = new JRadioButton("Choice 3");

// Create a ButtonGroup object.
ButtonGroup group = new ButtonGroup();

// Add the radio buttons to the ButtonGroup object.
group.add(radio1);
group.add(radio2);
group.add(radio3);
```

Although you add radio buttons to a `ButtonGroup` object, `ButtonGroup` objects are not containers like `JPanel` objects, or content frames. The function of a `ButtonGroup` object is to deselect all the other radio buttons when one of them is selected. If you wish to add the radio buttons to a panel or a content frame, you must add them individually, as shown here:

```
// Add the radio buttons to the JPanel referenced by panel.
panel.add(radio1);
panel.add(radio2);
panel.add(radio3);
```

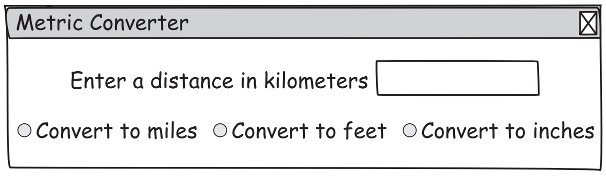
Responding to Radio Button Events

Just like `JButton` objects, `JRadioButton` objects generate an action event when they are clicked. To respond to a radio button action event, you must write an action listener class and then register an instance of that class with the `JRadioButton` object. To demonstrate, we will look at the `MetricConverter` class, which is similar to the `KiloConverter` class shown earlier. The `MetricConverter` class presents a window in which the user can enter a distance in kilometers, and then click radio buttons to see that distance converted to miles, feet, or inches. The conversion formulas are as follows:

$Miles = Kilometers \times 0.6214$
$Feet = Kilometers \times 3281.0$
$Inches = Kilometers \times 39370.0$

Figure 23-25 shows a sketch of what the window will look like. As you can see from the sketch, the window will have a label, a text field, and three radio buttons. When the user clicks on one of the radio buttons, the distance will be converted to the selected units and displayed in a separate `JOptionPane` dialog box.

Figure 23-25 Metric Converter window (Oracle Corporate Counsel)



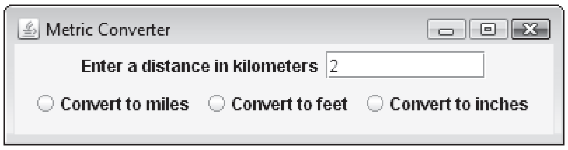
The `MetricConverter` class is shown in Code Listing 23-14. The class initially displays the window shown at the top of Figure 23-26. The figure also shows the dialog boxes that are displayed when the user clicks any of the radio buttons.

Code Listing 23-14 `(MetricConverter.java)`

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

/**
   The MetricConverter class lets the user enter a
   distance in kilometers. Radio buttons can be selected to
   convert the kilometers to miles, feet, or inches.
*/

public class MetricConverter extends JFrame
{
   private JPanel panel;                      // A holding panel
   private JLabel messageLabel;               // A message to the user
   private JTextField kiloTextField;          // To hold user input
   private JRadioButton milesButton;          // To convert to miles
   private JRadioButton feetButton;           // To convert to feet
   private JRadioButton inchesButton;         // To convert to inches
   private ButtonGroup radioButtonGroup;      // To group radio buttons
   private final int WINDOW_WIDTH = 400;      // Window width
   private final int WINDOW_HEIGHT = 100;     // Window height

   /**
      Constructor
   */

   public MetricConverter()
   {
      // Set the title.
      setTitle("Metric Converter");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

      // Specify an action for the close button.
      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, text field, and
      and three buttons to a panel.
   */

   private void buildPanel()
   {
      // Create the label, text field, and radio buttons.
      messageLabel = new JLabel("Enter a distance in kilometers");
      kiloTextField = new JTextField(10);
      milesButton = new JRadioButton("Convert to miles");
      feetButton = new JRadioButton("Convert to feet");
      inchesButton = new JRadioButton("Convert to inches");

      // Group the radio buttons.
      radioButtonGroup = new ButtonGroup();
      radioButtonGroup.add(milesButton);
      radioButtonGroup.add(feetButton);
      radioButtonGroup.add(inchesButton);

      // Add action listeners to the radio buttons.
      milesButton.addActionListener(new RadioButtonListener());
      feetButton.addActionListener(new RadioButtonListener());
      inchesButton.addActionListener(new RadioButtonListener());

      // Create a panel and add the components to it.
      panel = new JPanel();
      panel.add(messageLabel);
      panel.add(kiloTextField);
      panel.add(milesButton);
      panel.add(feetButton);
      panel.add(inchesButton);
   }

   /**
      Private inner class that handles the event when
      the user clicks one of the radio buttons.
   */

   private class RadioButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         String input;           // To hold the user's input
         String convertTo = "";  // The units we're converting to
         double result = 0.0;    // To hold the conversion

         // Get the kilometers entered.
         input = kiloTextField.getText();

         // Determine which radio button was clicked.
         if (e.getSource() == milesButton)
         {
            // Convert to miles.
            convertTo = " miles.";
            result = Double.parseDouble(input) * 0.6214;
         }
         else if (e.getSource() == feetButton)
         {
            // Convert to feet.
            convertTo = " feet.";
            result = Double.parseDouble(input) * 3281.0;
         }
         else if (e.getSource() == inchesButton)
         {
            // Convert to inches.
            convertTo = " inches.";
            result = Double.parseDouble(input) * 39370.0;
         }

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

   /**
      The main method creates an instance of the
      MetricConverter class, displaying its window.
   */

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

Figure 23-26 Window and dialog boxes displayed by the `MetricConverter` class (Oracle Corporate Counsel)

This window appears first. The user enters 2 into the text field.

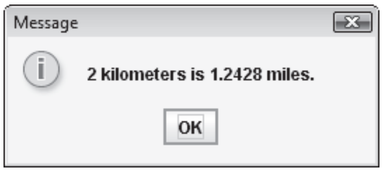


Metric Converter

Enter a distance in kilometers 2

○ Convert to miles ○ Convert to feet ○ Convert to inches

This dialog box appears when the user clicks the "Convert to miles" radio button.

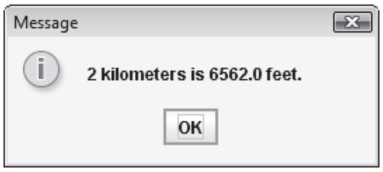


Message

2 kilometers is 1.2428 miles.

OK

This dialog box appears when the user clicks the "Convert to feet" radio button.

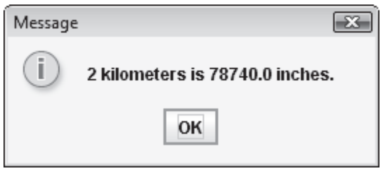


Message

2 kilometers is 6562.0 feet.

OK

This dialog box appears when the user clicks the "Convert to inches" radio button.



Message

2 kilometers is 78740.0 inches.

OK

Determining in Code Whether a Radio Button Is Selected

In many applications you will merely want to know whether a radio button is selected. The `JRadioButton` class's `isSelected` method returns a `boolean` value indicating whether the radio button is selected. If the radio button is selected, the method returns `true`. Otherwise, it returns `false`. In the following code, the `radio` variable references a radio button. The `if` statement calls the `isSelected` method to determine whether the radio button is selected.

```
if (radio.isSelected())
{
   // Code here executes if the radio
   // button is selected.
}
```

Selecting a Radio Button in Code

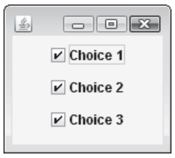
It is also possible to select a radio button in code with the `JRadioButton` class's `doClick` method. When the method is called, the radio button is selected just as if the user had clicked on it. As a result, an action event is generated. In the following statement, the `radio` variable references a radio button. When this statement executes, the radio button will be selected.

```
radio.doClick();
```

Check Boxes

A *check box* appears as a small box with a label appearing next to it. The window shown in Figure 23-27 has three check boxes.

Figure 23-27 Check boxes (Oracle Corporate Counsel)



Like radio buttons, check boxes may be selected or deselected at run time. When a check box is selected, a small check mark appears inside the box. Although check boxes are often displayed in groups, they are not usually grouped in a `ButtonGroup` like radio buttons. This is because check boxes are not normally used to make mutually exclusive selections. Instead, the user is allowed to select any or all of the check boxes that are displayed in a group.

You create a check box with the `JCheckBox` class. Here are the general formats of two `JCheckBox` constructors:

```
JCheckBox(String text)
JCheckBox(String text, boolean selected)
```

The first constructor shown creates a deselected check box. The argument passed to the `text` parameter is the string that is displayed next to the check box. For example, the following statement creates a check box with the text "Macaroni" displayed next to it. The check box initially appears deselected.

```
JCheckBox check1 = new JCheckBox("Macaroni");
```

The second constructor takes an additional `boolean` argument, which is passed to the `selected` parameter. If `true` is passed as the `selected` argument, the radio check box initially appears selected. If `false` is passed, the check box initially appears deselected. For example, the following statement creates a check box with the text "Macaroni" displayed next to it. The radio check box initially appears selected.

```
JCheckBox check1 = new JCheckBox("Macaroni", true);
```

Responding to Check Box Events

When a `JCheckBox` object is selected or deselected, it generates an *item event*. You handle item events in a manner similar to the way you handle the action events that are generated by `JButton` and `JRadioButton` objects. First, you write an *item listener* class, which must meet the following requirements:

- It must implement the `ItemListener` interface.
- It must have a method named `itemStateChanged` with the following header:

```
    public void itemStateChanged(ItemEvent e)
```

NOTE: When implementing the `ItemListener` interface, your code must have the following import statement: `import java.awt.event.*;`

Once you have written an item listener class, you create an object of that class, and then register the item listener object with the `JCheckBox` component. When a `JCheckBox` component generates an event, it automatically executes the `itemStateChanged` method of the item listener object that is registered to it, passing the event object as an argument.

Determining in Code Whether a Check Box Is Selected

As with `JRadioButton`, you use the `isSelected` method to determine whether a `JCheckBox` component is selected. The method returns a `boolean` value. If the check box is selected, the method returns `true`. Otherwise, it returns `false`. In the following code, the `checkBox` variable references a `JCheckBox` component. The `if` statement calls the `isSelected` method to determine whether the check box is selected.

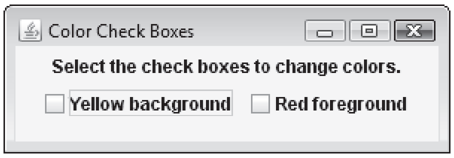
```
if (checkBox.isSelected())
{
   // Code here executes if the check
   // box is selected.
}
```

The `ColorCheckBoxWindow` class, shown in Code Listing 23-15, demonstrates how check boxes are used. It displays the window shown in Figure 23-28. When the "Yellow background" check box is selected, the background color of the content pane, the label, and the check boxes turns yellow. When this check box is deselected, the background colors go back to light gray. When the "Red foreground" check box is selected, the color of the text displayed in the label and the check boxes turns red. When this check box is deselected, the foreground colors go back to black.

Code Listing 23-15 (ColorCheckBoxWindow.java)

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

/**
   The ColorCheckBoxWindow class demonstrates how check boxes
   can be used.
*/

public class ColorCheckBoxWindow extends JFrame
{
   private JLabel messageLabel;        // A message to the user
   private JCheckBox yellowCheckBox;   // To select yellow background
   private JCheckBox redCheckBox;      // To select red foreground
   private final int WINDOW_WIDTH = 300;     // Window width
   private final int WINDOW_HEIGHT = 100;    // Window height

   /**
      Constructor
   */

   public ColorCheckBoxWindow()
   {
      // Set the text for the title bar.
      setTitle("Color Check Boxes");

      // Set the size of the window.
      setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

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

      // Create a label.
      messageLabel = new JLabel("Select the check " +
                                "boxes to change colors.");

      // Create the check boxes.
      yellowCheckBox = new JCheckBox("Yellow background");
      redCheckBox = new JCheckBox("Red foreground");

      // Add an item listener to the check boxes.
      yellowCheckBox.addItemListener(new CheckBoxListener());
      redCheckBox.addItemListener(new CheckBoxListener());

      // Add a FlowLayout manager to the content pane.
      setLayout(new FlowLayout());

      // Add the label and check boxes to the content pane.
      add(messageLabel);
      add(yellowCheckBox);
      add(redCheckBox);

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

   /**
      Private inner class that handles the event when
      the user clicks one of the check boxes.
   */

   private class CheckBoxListener implements ItemListener
   {
      public void itemStateChanged(ItemEvent e)
      {
         // Determine which check box was clicked.
         if (e.getSource() == yellowCheckBox)
         {
            // Is the yellow check box selected? If so, we
            // want to set the background color to yellow.
            if (yellowCheckBox.isSelected())
            {
               // The yellow check box was selected. Set
               // the background color for the content
               // pane and the two check boxes to yellow.
               getContentPane().setBackground(Color.YELLOW);
               yellowCheckBox.setBackground(Color.YELLOW);
               redCheckBox.setBackground(Color.YELLOW);
            }
            else
            {
               // The yellow check box was deselected. Set
               // the background color for the content
               // pane and the two check boxes to light gray.
               getContentPane().setBackground(Color.LIGHT_GRAY);
               yellowCheckBox.setBackground(Color.LIGHT_GRAY);
               redCheckBox.setBackground(Color.LIGHT_GRAY);
            }
         }
         else if (e.getSource() == redCheckBox)
         {
            // Is the red check box selected? If so, we want
            // to set the foreground color to red.
            if (redCheckBox.isSelected())
            {
               // The red check box was selected. Set the
               // foreground color for the label and the
               // two check boxes to red.
               messageLabel.setForeground(Color.RED);
               yellowCheckBox.setForeground(Color.RED);
               redCheckBox.setForeground(Color.RED);
            }
            else
            {
               // The red check box was deselected. Set the
               // foreground color for the label and the
               // two check boxes to black.
               messageLabel.setForeground(Color.BLACK);
               yellowCheckBox.setForeground(Color.BLACK);
               redCheckBox.setForeground(Color.BLACK);
            }
         }
      }
   }

   /**
      The main method creates an instance of the
      ColorCheckBoxWindow class, displaying its window.
   */

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

Figure 23-28 Window displayed by the `ColorCheckBoxWindow` class (Oracle Corporate Counsel)



Selecting a Check Box in Code

As with radio buttons, it is possible to select check boxes in code with the `JCheckBox` class's `doClick` method. When the method is called, the radio check box is selected just as if the user had clicked on it. As a result, an item event is generated. In the following statement, the checkBox variable references a JCheckBox object. When this statement executes, the check box will be selected.

```
checkBox.doClick();
```

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

23.18 You want the user to be able to select only one item from a group of items. Which type of component would you use for the items, radio buttons or check boxes?

23.19 You want the user to be able to select any number of items from a group of items. Which type of component would you use for the items, radio buttons or check boxes?

23.20 What is the purpose of a ButtonGroup object?

23.21 Do you normally add radio buttons, check boxes, or both to a ButtonGroup object?

23.22 What type of event does a radio button generate when the user clicks on it?

23.23 What type of event does a check box generate when the user clicks on it?

23.24 How do you determine in code whether a radio button is selected?

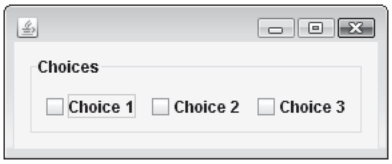
23.25 How do you determine in code whether a check box is selected?

23.5 Borders

CONCEPT: A component can appear with several different styles of borders around it. A Border object specifies the details of a border. You use the BorderFactory class to create Border objects.

Sometimes it is helpful to place a border around a component or a group of components on a panel. You can give windows a more organized look by grouping related components inside borders. For example, Figure 23-29 shows a group of check boxes that are enclosed in a border. In addition, notice that the border has a title.

Figure 23-29 A group of check boxes with a titled border



JPanel components have a method named setBorder, which is used to add a border to the panel. The setBorder method accepts a Border object as its argument. A Border object contains detailed information describing the appearance of a border.

Rather than creating Border objects yourself, you should use the BorderFactory class to create them for you. The BorderFactory class has methods that return various types of borders. Table 23-6 describes borders that can be created with the `BorderFactory` class. The table also lists the `BorderFactory` methods that can be called to create the borders. Note that there are several overloaded versions of each method.

NOTE: If you use the `BorderFactory` class in your code, you should have the following `import` statement: `import javax.swing.*;`

Table 23-6 Borders produced by the BorderFactory class

Border	BorderFactory Method	Description
Compound border	`createCompoundBorder`	A border that has two parts: an inside edge and an outside edge. The inside and outside edges can be any of the other borders.
Empty border	`createEmptyBorder`	A border that contains only empty space.
Etched border	`createEtchedBorder`	A border with a 3-D appearance that looks "etched" into the background.
Line border	`createLineBorder`	A border that appears as a line.
Lowered bevel border	`createLoweredBevelBorder`	A border that looks like beveled edges. It has a 3-D appearance that gives the illusion of being sunken into the surrounding background.
Matte border	`createMatteBorder`	A line border that can have edges of different thicknesses.
Raised bevel border	`createRaisedBevelBorder`	A border that looks like beveled edges. It has a 3-D appearance that gives the illusion of being raised above the surrounding background.
Titled border	`createTitledBorder`	An etched border with a title.

In this chapter, we will concentrate on empty borders, line borders, and titled borders.

Empty Borders

An empty border is simply empty space around the edges of a component. To create an empty border, call the `BorderFactory` class's `createEmtpyBorder` method. Here is the method's general format:

```
BorderFactory.createEmptyBorder(int top, int left,
                                int bottom, int right);
```

The arguments passed into `top`, `left`, `bottom`, and `right` specify in pixels the size of the border's top, left, bottom, and right edges. 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.createEmptyBorder(5, 5, 5, 5));
```

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*

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

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

23.6 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

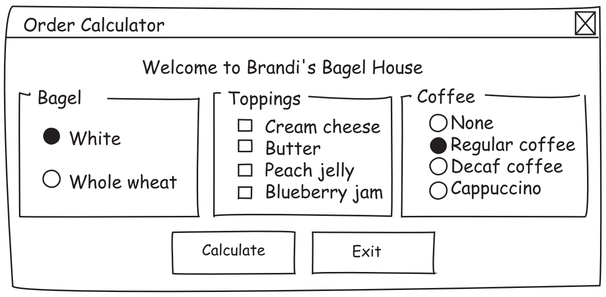
Brandi's Bagel House has a bagel and coffee delivery service for the businesses in her neighborhood. Customers may call in and order white and whole wheat bagels with a variety of toppings. In addition, customers may order three different types of coffee. (Delivery for coffee alone is not available, 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 23-30 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 23-30 Sketch of the Order Calculator window (Oracle Corporate Counsel)



- **`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 23-16 shows the class, which extends `JPanel`.

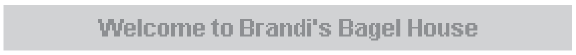
Code Listing 23-16 (`GreetingPanel.java`)

```
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 23-31 shows how the component will appear when it is placed in the window’s north region.

Figure 23-31 Appearance of the `GreetingPanel` component


Welcome to Brandi's Bagel House

The `BagelPanel` Class

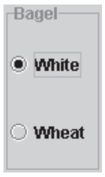
The `BagelPanel` class holds the radio buttons for the types of bagels. Notice that this panel uses a `GridLayout` manager with two rows and one column. Code Listing 23-17 shows the class, which is extended from `JPanel`.

Code Listing 23-17 (`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 23-32 shows how the component appears when it is placed in the window's west region.

Figure 23-32 Appearance of the `BagelPanel` component (Oracle Corporate Counsel)



The `ToppingPanel` Class

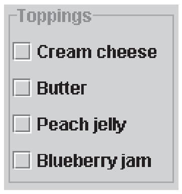
The `ToppingPanel` class holds the check boxes for the available toppings. Code Listing 23-18 shows the class, which is also extended from `JPanel`.

Code Listing 23-18 (`ToppingPanel.java`)

```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 23-33 shows how the component appears when it is placed in the window's center region.

Figure 23-33 Appearance of the `ToppingPanel` component (Oracle Corporate Counsel)



The `CoffeePanel` Class

The `CoffeePanel` class holds the radio buttons for the available coffee selections. Code Listing 23-19 shows the class, which extends `JPanel`.

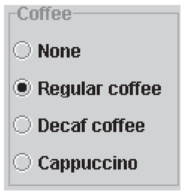
Code Listing 23-19 (`CoffeePanel.java`)

```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 23-34 shows how the component appears when it is placed in the window's east region.

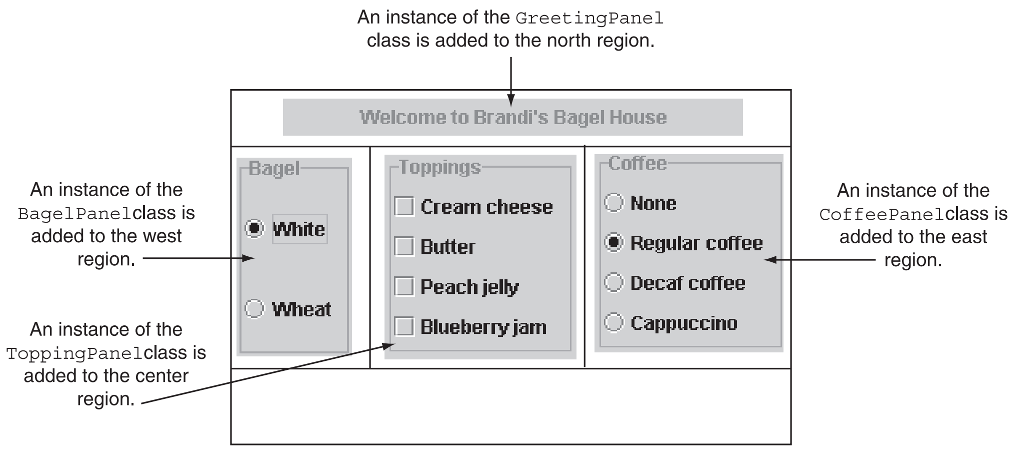
Figure 23-34 Appearance of the `CoffeePanel` component



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 23-35 shows how instances of the `GreetingPanel`, `BagelPanel`, `ToppingPanel`, and `CoffeePanel` classes are placed in the content pane.

Figure 23-35 Placement of the custom panels


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

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 23-20.

Code Listing 23-20 (`OrderCalculatorGUI.java`)

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

/**
   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;

         // Display the charges.
         JOptionPane.showMessageDialog(null,
            String.format("Subtotal: $%,.2f\n" +
                          "Tax: $%,.2f\n" +
                          "Total: $%,.2f",
                          subtotal, tax, 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 23-36 appears. Figure 23-37 shows the `JOptionPane` dialog box that is displayed when the user selects a wheat bagel with butter, cream cheese, and decaf coffee.

Figure 23-36 The Order Calculator window (Oracle Corporate Counsel)

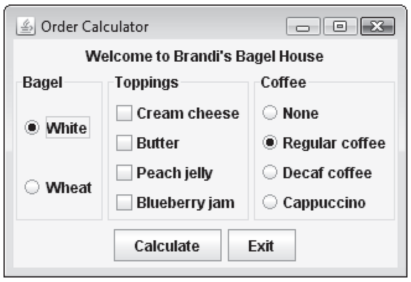
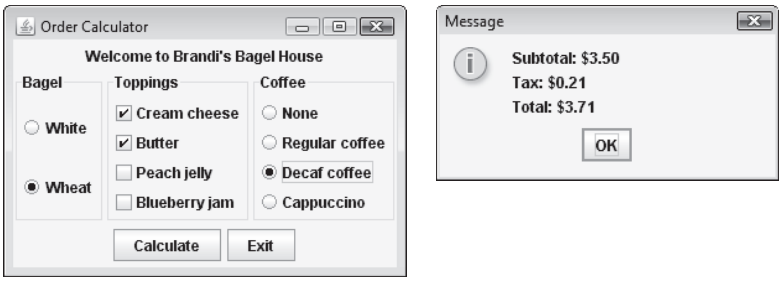


Figure 23-37 The subtotal, tax, and total displayed (Oracle Corporate Counsel)



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

23.7 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:

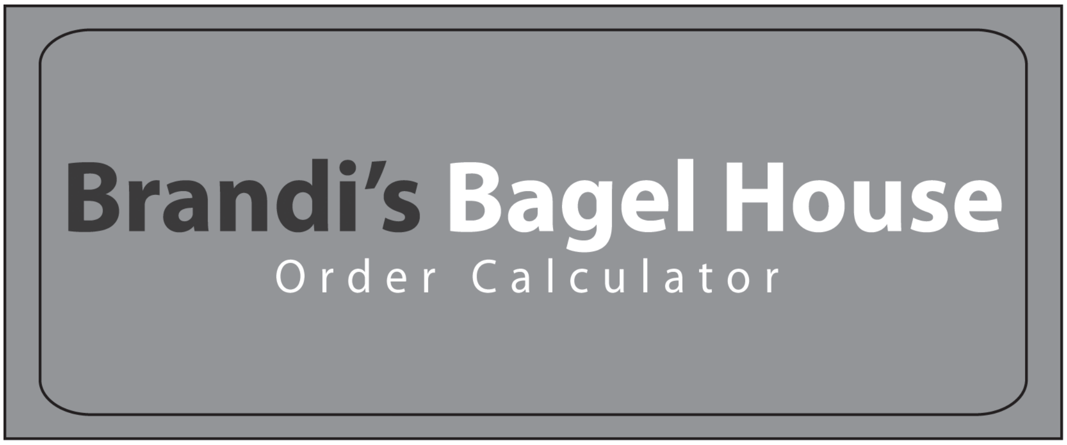
```
java -splash:GraphicFileName ClassFileName
```

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 23-38, 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 23-38 Splash screen for the *Brandi's Bagel House* application (Oracle Corporate Counsel)



23.8 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 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 23-21 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 23\KiloConverter Phase 3*.)

Code Listing 23-21 **(`KiloConverter.java`)**

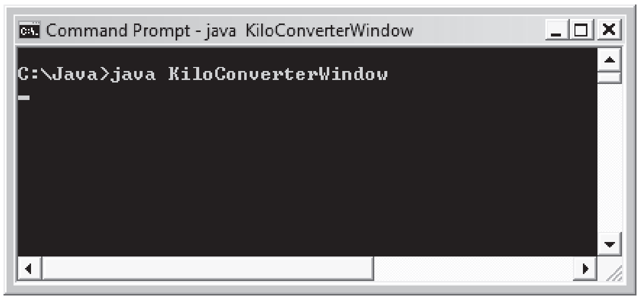
```
 1 import javax.swing.*;     // Needed for Swing classes
 2 import java.awt.event.*;  // Needed for ActionListener Interface
 3
 4 /**
 5    The KiloConverter class displays a JFrame that
 6    lets the user enter a distance in kilometers. When
 7    the Calculate button is clicked, a dialog box 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();
   }
}
```

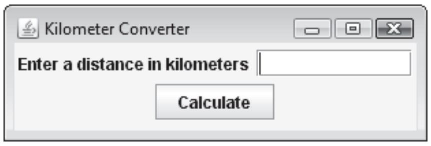
Let's take a closer look. In lines 101 and 102, a message is displayed to the console showing the value that was read from the text field. In lines 103 and 104, another message is displayed showing the value after it is converted to a `double`. Then, in line 115, a message is displayed indicating that the application is ready for its next input. Figure 23-39 shows an example session with the application on a computer running Microsoft Windows. Both the console window and the application windows are shown.

Figure 23-39 Messages displayed to the console during the application's execution
(Oracle Corporate Counsel)

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

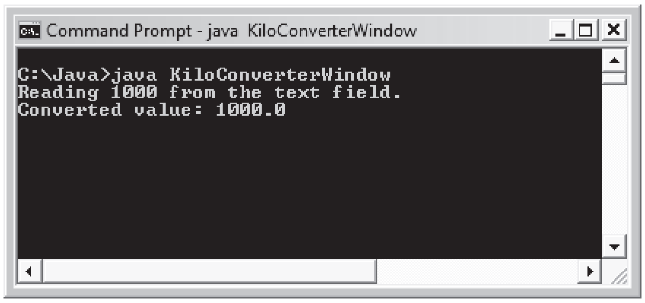


```
C:\Java>java KiloConverterWindow
_
```

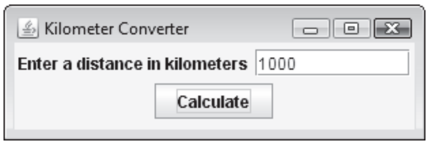


Enter a distance in kilometers

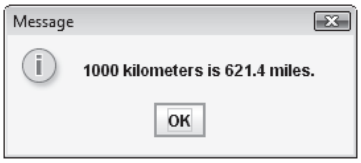
2. The user types a value into the text field and clicks the Calculate button. Debugging messages appear in the console window, and a message dialog appears showing the value converted to miles.



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

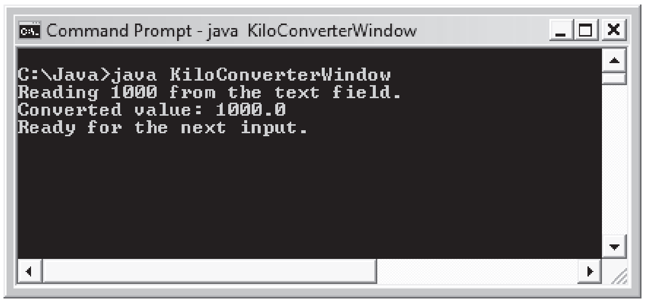


Enter a distance in kilometers 1000

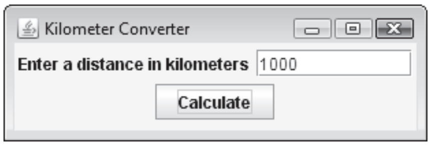


1000 kilometers is 621.4 miles.

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.
```



Enter a distance in kilometers 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.

23.9 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.