CHAPTER 10

Additional Topics

After studying this chapter, you will be able to:

- Create a simple programmer-defined class
- Create a simple graphical user interface (GUI)

This chapter covers topics included in Chapters 10, 11, and 12 in *Programming Logic and Design, Eighth Edition*, by Joyce Farrell.

A Programmer-Defined Class

You should do the exercises and labs in this section after you have finished Chapters 10 and 11 in *Programming Logic and Design, Eighth Edition.* You should also take a moment to review the object-oriented terminology (class, attribute, and method) presented in Chapter 1 of this book and in Chapter 10 of *Programming Logic and Design.*

You have been using prewritten classes, objects, and methods throughout this book. For example, you have used the `showInputDialog()` method that belongs to the `JOptionPane` class to display an input dialog box, and you have used the `parseInt()` method that belongs to the `Integer` class. In this section, you learn how to create your own class that includes attributes and methods of your choice. In programming terminology, a class created by the programmer is referred to as a **programmer-defined class**.

To review, procedural programming focuses on declaring data and defining methods separate from the data and then calling those methods to operate on the data. This is the style of programming you have been using in Chapters 1 through 9 of this book. Object-oriented programming is different from procedural programming. Object-oriented programming focuses on an application's data and the methods you need to manipulate that data. The data and methods are **encapsulated**, or contained within, a class. Objects are created as an instance of a class. The program tells an object to perform tasks by passing messages to it. Such a message consists of an instruction to execute one of the class's methods. The class method then manipulates the data (which is part of the object itself).

Creating a Programmer-Defined Class

In Chapter 10 of *Programming Logic and Design,* you studied pseudocode for the `Employee` class. This pseudocode is shown in Figure 10-1. The Java code that implements the `Employee` class is shown in Figure 10-2.

```
class Employee
   string lastName
   num hourlyWage
   num weeklyPay

   void setLastName(string name)
      lastName = name
   return

   void setHourlyWage(num wage)
      hourlyWage = wage
      calculateWeeklyPay()
   return

   string getLastName()
   return lastName

   num getHourlyWage()
   return hourlyWage

   num getWeeklyPay()
   return weeklyPay

   void calculateWeeklyPay()
      num WORK_WEEK_HOURS = 40
      weeklyPay = hourlyWage * WORK_WEEK_HOURS
   return
endClass
```

Figure 10-1 Pseudocode for `Employee` class

```
 1 // Employee class
 2 public class Employee
 3 {
 4    private String lastName;
 5    private double hourlyWage;
 6    private double weeklyPay;
 7
 8    public void setLastName(String name)
 9    {
10       lastName = name;
11       return;
12    }
13
14    public void setHourlyWage(double wage)
15    {
16       hourlyWage = wage;
17       calculateWeeklyPay();
18       return;
19    }
20
21    public String getLastName()
22    {
23       return lastName;
24    }
25
26    public double getHourlyWage()
27    {
28       return hourlyWage;
29    }
30
31    public double getWeeklyPay()
32    {
33       return weeklyPay;
34    }
35
36    private void calculateWeeklyPay()
37    {
38       final int WORK_WEEK_HOURS = 40;
39       weeklyPay = hourlyWage * WORK_WEEK_HOURS;
40       return;
41    }
42 } // End Employee class
```

Figure 10-2 `Employee` class implemented in Java

Looking at the pseudocode in Figure 10-1, you see that you begin creating a class by specifying that it is a class. In the Java code in Figure 10-2, line 1 is a comment. This is followed by the class declaration for the `Employee` class on line 2. The class declaration begins with the keyword, `public`, which allows this class to be used in programs, followed by the keyword, `class`, which specifies that what follows is a Java class. The opening curly brace on line 3 and the closing curly brace on line 42 mark the beginning and the end of the class.

Adding Attributes to a Class

The next step is to define the attributes (data) that are included in the `Employee` class. As shown on lines 2, 3, and 4 of the pseudocode in Figure 10-1, there are three attributes in this pseudocode class, `string lastName`, `num hourlyWage`, and `num weeklyPay`.

Lines 4, 5, and 6 in Figure 10-2 include these attributes in the Java version of the `Employee` class. Notice in the Java code that `hourlyWage` and `weeklyPay` are defined using the `double` data type, and `lastName` is defined as a `String`. Also, notice that all three attributes are `private`. As explained in *Programming Logic and Design,* this means the data cannot be accessed by any method that is not part of the class. Programs that use the `Employee` class must use the methods that are part of the class to access private data.

Adding Methods to a Class

The next step is to add methods to the `Employee` class. The pseudocode versions of these methods, shown on lines 6 through 27 in Figure 10-1, are nonstatic methods. As you learned in Chapter 10 of *Programming Logic and Design,* **nonstatic methods** are methods that are meant to be used with an object created from a class. In other words, to use these methods, you must create an object of the `Employee` class first and then use that object to invoke (or call) the method.

The code shown in Figure 10-2 shows how to include methods in the `Employee` class using Java. This discussion begins with the set methods. You learned in *Programming Logic and Design* that **set methods** are those whose purpose is to set the values of attributes (data fields) within the class. There are three data fields in the `Employee` class, but you will only add two set methods, `setLastName()` and `setHourlyWage()`. You will not add a `setWeeklyPay()` method, because the `weeklyPay` data field will be set by the `setHourlyWage()` method. The `setHourlyWage()` method uses another method, `calculateWeeklyPay()`, to accomplish this.

The two set methods, `setLastName()` shown on lines 8 through 12 in Figure 10-2, and `setHourlyWage()` shown on lines 14 through 19, are declared using the keyword `public`. This means that programs may use these methods to gain access to the private data. The `calculateWeeklyPay()` method, shown on lines 36 through 41 in Figure 10-2, is `private`, which means it must be called from within another method that already belongs to the class. In the `Employee` class, the `calculateWeeklyPay()` method is called from the `setHourlyWage()` method (line 17), which ensures that the class retains full control over when and how the `calculateWeeklyPay()` method is used.

The `setLastName()` method (lines 8 through 12) accepts one argument, `String name`, that is assigned to the private attribute, `lastName`. This sets the value of `lastName`. The `setLastName()` method is a `void` method—that is, it returns nothing.

The `setHourlyWage()` method (lines 14 through 19) accepts one argument, `double wage`, that is assigned to the private attribute, `hourlyWage`. This sets the value of `hourlyWage`. Next, it calls the `private` method, `calculateWeeklyPay()`. The `calculateWeeklyPay()` method does not accept arguments. Within the method, on line 38, a constant, `final int`

WORK_WEEK_HOURS, is declared and initialized with the value 40. The calculateWeeklyPay() method then calculates weekly pay (line 39) by multiplying the private attribute, hourlyWage, by WORK_WEEK_HOURS. The result is assigned to the private attribute, weeklyPay. The setHourlyWage() method and the calculateWeeklyPay() method are void methods, which means they return nothing.

The final step in creating the Employee class is adding the get methods. **Get methods** are methods that return a value to the program using the class. The pseudocode in Figure 10-1 includes three get methods, getLastName() on lines 15 and 16, getHourlyWage() on lines 18 and 19, and getWeeklyPay() on lines 21 and 22. Lines 21 through 34 in Figure 10-2 illustrate the Java version of the get methods in the Employee class.

The three get methods are public methods and accept no arguments. The getLastName() method, shown on lines 21 through 24, returns a String, which is the value of the private attribute, lastName. The getHourlyWage() method, shown on lines 26 through 29, returns a double, which is the value of the private attribute, hourlyWage, and the getWeeklyPay() method, shown on lines 31 through 34, also returns a double, which is the value of the private attribute, weeklyPay.

The Employee class is now complete and may be used in a Java program. The Employee class does not contain a main() method because it is not an application but rather a class that an application may now use to instantiate objects.

The completed Employee class is included in the student files provided for this book in a file named Employee.java.

Figure 10-3 illustrates a program named Employee Wages that uses the `Employee` class.

```
 1 // This program uses the programmer-defined Employee class.
 2
 3 public class EmployeeWages
 4 {
 5    public static void main(String args[])
 6    {
 7       final double LOW = 9.00;
 8       final double HIGH = 14.65;
 9       // Instantiate an Employee object
10       Employee myGardener = new Employee();
11
12       // Use the get and set methods
13       myGardener.setLastName("Greene");
14       myGardener.setHourlyWage(LOW);
15       System.out.println("My gardener makes " +
16              myGardener.getWeeklyPay() + " per week.");
17
18       // Use the get and set methods
19       myGardener.setHourlyWage(HIGH);
20       System.out.println("My gardener makes " +
21              myGardener.getWeeklyPay() + " per week.");
22       System.exit(0);
23    }
24 }
```

Figure 10-3 Employee Wages program that uses the `Employee` class

As shown in Figure 10-3, the Employee Wages program begins with a comment on line 1, followed by the creation of a class named `EmployeeWages` on line 3. This class contains a `main()` method that begins on line 5. A `main()` method must be written in this class because it is an application. As in other programs you have seen throughout this book, the `main()` method header includes the keyword `static`. As you learned in Chapter 10 of *Programming Logic and Design,* **static methods** are those for which no object needs to exist. This means that you do not need to create an `EmployeeWages` object in order to call the `main()` method. On lines 7 and 8 within the `main()` method, two constants, `LOW` and `HIGH`, are declared and initialized. Next, on line 10, an `Employee` object (an instance of the `Employee` class) is created with the following statement:

```
Employee myGardener = new Employee();
```

In Java, a statement that creates a new object consists of the class name followed by the object's name. In the preceding example, the class is `Employee`, and the name of the object is `myGardener`. Next comes the assignment operator, followed by the `new` keyword and the name of a constructor you want to use to create the object.

You used the `new` keyword to instantiate `FileReader` and `FileWriter` objects in Chapter 7 of this book.

As you learned in *Programming Logic and Design,* a **constructor** is a method that creates an object. You also learned that you can use a prewritten **default constructor**, which is a constructor that expects no arguments and is created automatically by the compiler for every class you write. The `Employee()` constructor used in the Employee Wages program is an example of a prewritten default constructor.

Constructors always have the same name as the class and are always written with no return value—not even `void`.

You can also write your own constructors. You will learn more about additional constructors in future Java courses.

Once the `myGardener` object is created, you can use `myGardener` to invoke the set methods to set the value of `lastName` to `"Greene"` and the `hourlyWage` to `LOW`. The syntax used is shown in the following code sample.

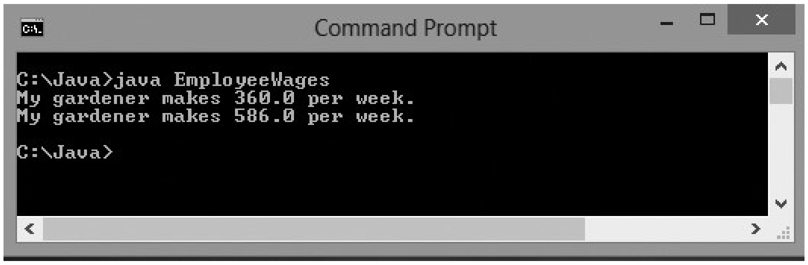
```
myGardener.setLastName("Greene");
myGardener.setHourlyWage(LOW);
```

This is the syntax used to invoke a method with an instance (an object) of a class.

Notice the syntax, ***`objectName.methodName`***, in which the name of the object is separated from the name of the method by a dot, which is actually a period.

On lines 15 and 16 in Figure 10-3, the program then prints `"My gardener makes "` (a string constant) followed by the return value of `myGardener.getWeeklyPay()`, followed by the string constant `" per week."`. Here, the `myGardener` object is used again—this time to invoke the `getWeeklyPay()` method.

On line 19, `myGardener` invokes the set method, `setHourlyWage()`, to set a new value for `hourlyWage`. This time `hourlyWage` is set to `HIGH`. The program then prints (lines 20 and 21) `"My gardener makes "` (a string constant) followed by the return value of `myGardener.getWeeklyPay()`, followed by the string constant `" per week."`. The `System.exit(0);` statement on line 22 ends the program. The output from this program is shown in Figure 10-4.



```
C:\Java>java EmployeeWages
My gardener makes 360.0 per week.
My gardener makes 586.0 per week.

C:\Java>
```

Figure 10-4 Output from the Employee Wages program

You will find the completed program in a file named `EmployeeWages.java` included with the student files for this book.

Exercise 10-1: Creating a Programmer-Defined Class in Java

In this exercise, you use what you have learned about creating and using a programmer defined class. Study the following code, and then answer Questions 1–4.

```
class Circle
{
   private double radius;   // Radius of this circle
   final double PI = 3.14159;
   public void setRadius(double rad)
   {
      radius = rad;
   }
   public double getRadius()
   {
      return radius;
   }
   public double calculateCircumference()
   {
      return (2 * PI * radius);
   }
   public double calculateArea()
   {
      return(PI * radius * radius);
   }
} // End of Circle class
```

In this exercise, assume that a `Circle` object named `oneCircle` has been created in a program that uses the `Circle` class, and `radius` is given a value as shown in the following code:

```
Circle oneCircle = new Circle();
oneCircle.setRadius(4.5);
```

1. What is the output when the following line of Java code executes?

```
System.out.println("The circumference is : " +
       oneCircle.calculateCircumference());
```

2. Is the following a legal Java statement? Why or why not?

   `System.out.println("The area is : " + calculateArea());`

   ---

3. Consider the following Java code. What is the value stored in the `oneCircle` object's attribute named `radius`?

   `oneCircle.setRadius(10.0);`

   ---

4. Write the Java code that will assign the circumference of `oneCircle` to a `double` variable named `circumference1`.

Lab 10-1: Creating a Programmer-Defined Class in Java

In this lab, you will create a programmer-defined class and then use it in a Java program. The program should create two `Rectangle` objects and find their area and perimeter. Use the `Circle` class that you worked with in Exercise 10-1 as a guide.

1. Open the class file named `Rectangle.java` using Notepad or the text editor of your choice.
2. In the `Rectangle` class, create two private attributes named `length` and `width`. Both `length` and `width` should be data type `double`.
3. Write `public` set methods to set the values for `length` and `width`.
4. Write `public` get methods to retrieve the values for `length` and `width`.
5. Write a `public calculateArea()` method and a `public calculatePerimeter()` method to calculate and return the area of the rectangle and the perimeter of the rectangle.
6. Save this class file, `Rectangle.java`, in a directory of your choice, and then open the file named `MyRectangleClassProgram.java`.
7. In the `MyRectangleClassProgram` class, create two `Rectangle` objects named `rectangle1` and `rectangle2` using the default constructor as you saw in `EmployeeWages.java`.
8. Set the length of `rectangle1` to `10.0` and the width to `5.0`. Set the length of `rectangle2` to `7.0` and the width to `3.0`.
9. Print the value of `rectangle1`'s perimeter and area, and then print the value of `rectangle2`'s perimeter and area.
10. Save `MyRectangleClassProgram.java` in the same directory as `Rectangle.java`.
11. Compile the source code file `MyRectangleClassProgram.java`.
12. Execute the program.
13. Record the output below.

Creating a Graphical User Interface (GUI)

You should do the exercises and labs in this section after you have finished Chapter 12 in *Programming Logic and Design, Eighth Edition,* which discusses creating a **graphical user interface** (GUI). To review briefly, a GUI allows users to interact with programs by using a mouse to point, drag, or click. GUI programs are referred to as **event-driven** or **event-based** because this type of program responds to user-initiated events, such as a mouse click. Within a GUI program, an **event listener** waits for an event to occur and then responds to it. An event listener is actually a method that contains Java code that executes when a particular event occurs. For example, when a user of a GUI program clicks a button, an event occurs. In response to the event, the event listener (a method) that is written as part of the GUI program executes.

To create full-blown, event-driven programs that make use of a graphical user interface, you need to learn more about Java than is included in this book. In this section, you will learn to use just a few of the many graphical user interface components that are included in the Java Standard Edition Development Kit, such as a button, a label, and a frame. You will also learn to write event listeners that respond to specific user actions, such as clicking.

The Java program shown in Figure 10-5 creates the graphical user interface shown in Figure 10-6. This GUI is made up of a frame, a panel, some buttons, and some labels. When the program executes, the user can click buttons to change the color of a button or the background color of the panel. You will learn about buttons, labels, frames, and panels in the following sections.

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

class GuiDemo
{
   JPanel panel;

   GuiDemo()
   {
      JLabel redLabel = new JLabel("Click to change color");
      JLabel blueLabel =
             new JLabel("Click to change color");
      JLabel backLabel =
             new JLabel("Click to change background color");

```

Figure 10-5 Java program that uses a graphical user interface (GUI) *(continues)*

(continued)

```
      final JButton redButton = new JButton("Red");
      redButton.addActionListener(new ActionListener(){
         public void actionPerformed(ActionEvent e){
            redButton.setBackground(Color.RED);
         }
      });
      final JButton blueButton = new JButton("Blue");
      blueButton.addActionListener(new ActionListener(){
         public void actionPerformed(ActionEvent e){
            blueButton.setBackground(Color.BLUE);
         }
      });
      final JButton backButton = new JButton("Background");
      backButton.addActionListener(new ActionListener(){
         public void actionPerformed(ActionEvent e){
            panel.setBackground(Color.GREEN);
         }
      });

      panel = new JPanel();

      panel.add(redLabel);
      panel.add(redButton);

      panel.add(blueLabel);
      panel.add(blueButton);

      panel.add(backLabel);
      panel.add(backButton);
   }

   public static void main(String args[])
   {
      GuiDemo demo = new GuiDemo();
      JFrame frame = new JFrame("GUI Demo");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.setContentPane(demo.panel);
      frame.pack();
      frame.setVisible(true);
   }
}
```

Figure 10-5 Java program that uses a graphical user interface (GUI)

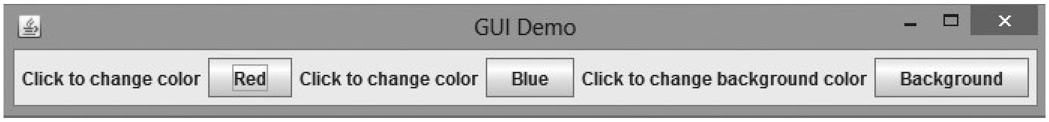


Figure 10-6 Graphical user interface created by `GuiDemo.java`

In Figure 10-5, lines 1, 2, and 3 are `import` statements that import packages. You learned about `import` statements in Chapter 2 of this book. Remember that a **package** is a group of related classes. The classes used in this program are part of the packages named `javax.swing`, `java.awt`, and `java.awt.event`. When you **import** a class, a program then has access to the methods that are part of that class. The `javax.swing` package contains components such as the `JButton` class. The `java.awt` package contains component classes as well as other graphics classes, such as the `Color` class. The `java.awt.event` package contains classes that you can use to write event listeners that respond to events.

Line 5 in Figure 10-5 begins the class named `GuiDemo`. The first statement (line 7) in the `GuiDemo` class uses the `JPanel` class to create a reference to a `JPanel` object named `panel`. The reference is not itself a `JPanel` object, but merely a location in memory where the address of an actual `JPanel` object will be stored later in the program. A **JPanel** is a Java component that is considered a container. In Java, a **container** is a component that is used to hold or organize other components. In this program, the `JPanel` is used to hold buttons and labels.

Writing a Constructor

Lines 9 through 46 of Figure 10-5 include a method named `GuiDemo()`. You know this method is a constructor because it has the same name as the class. This constructor expects no arguments and will execute when a `GuiDemo` object is created. Within the `GuiDemo()` constructor (lines 11 through 15), you create three `JLabel` objects named `redLabel`, `blueLabel`, and `backLabel` as:

```
JLabel redLabel = new JLabel("Click to change color");
JLabel blueLabel =
       new JLabel("Click to change color");
JLabel backLabel =
       new JLabel("Click to change background color");
```

In Java, a `JLabel` is used to display a single line of read-only text. **Read-only** means that the user cannot change the text that is displayed. In this example, the interface displays two instances of read-only text: `"Click to change color"` and `"Click to change background color"`.

The next section of code is rather complicated:

```
final JButton redButton = new JButton("Red");
redButton.addActionListener(new ActionListener(){
     public void actionPerformed(ActionEvent e){
          redButton.setBackground(Color.RED);
     }
});
final JButton blueButton = new JButton("Blue");
blueButton.addActionListener(new ActionListener(){
     public void actionPerformed(ActionEvent e){
          blueButton.setBackground(Color.BLUE);
     }
});
```

```
final JButton backButton = new JButton("Background");
backButton.addActionListener(new ActionListener(){
     public void actionPerformed(ActionEvent e){
          panel.setBackground(Color.GREEN);
     }
});
```

This code (lines 17 through 34 in Figure 10-5) creates `JButton` objects and attaches event listener methods to the `JButtons`. The following three lines of code (lines 17, 23, and 29) create three `JButton` objects known as push buttons. When a user clicks (pushes) a `JButton`, an event occurs that causes something to happen in the program. In this program, clicking the `redButton` causes it to turn red, clicking the `blueButton` causes it to turn blue, and clicking the `backButton` causes the background color of the `JPanel` to turn green. The string constants within the parentheses cause the text `"Red"`, `"Blue"`, or `"Background"` to be displayed on the `JButtons`.

In Java, local variables, such as `JButtons`, must be declared `final` to be used in an anonymous inner class, which is discussed next.

```
final JButton redButton = new JButton("Red");
final JButton blueButton = new JButton("Blue");
final JButton backButton = new JButton("Background");
```

Next, you examine the event handlers. The following code (lines 18 through 22) adds an event handler to the `JButton` named `redButton`:

```
redButton.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e){
      redButton.setBackground(Color.RED);
   }
});
```

The `redButton` object invokes the `addActionListener()` method (line 18) and passes a new `ActionListener` object as a parameter. In Java, `JButton` objects generate `Action Events` when they are clicked and require an event listener to handle the `Action Event`. The event listener for `Action Events` is called an `ActionListener`. Therefore, the `redButton` requires an `ActionListener`. To add the `ActionListener` to the `redButton`, you need to create a new `ActionListener` object. You accomplish this by creating an **anonymous inner class**, which is a class that does not have a name and that is nested within another class.

The program has access to the `addActionListener()` method and `ActionEvent` objects because you imported the `java.awt.event` package.

Within this inner class, you need to write one method, `public void actionPerformed(ActionEvent e)`. The `actionPerformed()` method must be written to accept one parameter, an `ActionEvent` object, which, in this program, is named `e`.

This method contains code that instructs the program what action to take when the user clicks the `redButton`. The code required for this program (line 20) is shown in the following example:

```
redButton.setBackground(Color.RED);
```

The `redButton` object invokes the `setBackground(Color.RED)` method and passes `Color.RED` as an argument. This method changes the color of the `redButton` object to the color passed to it. In this case, the color is red.

You have access to the `setBackground()` method because it is contained in the `JButton` class, which is part of the `javax.swing` package you imported.

Lines 23 through 28 add an `ActionListener` to the `JButton` named `blueButton` to change its color to `Color.BLUE` when it is clicked. Similarly, lines 29 through 34 add an `ActionListener` to the `JButton` named `backButton` to change the color of the `JPanel` named `panel` to `Color.GREEN` when it is clicked.

You have access to the `Color` class because you imported the `java.awt` package. Several attributes are defined in the `Color` class, including `RED`, `BLUE`, and `GREEN`.

Line 36 creates a `JPanel` object and assigns its reference (memory address) to `panel`. Lines 38 through 45 use `panel` (the `JPanel` object) to invoke the `add()` method. The `add()` method is used to add the `JLabels` and `JButtons` to the `JPanel` container. You are now finished writing the `GuiDemo()` constructor.

Writing the `main()` Method

As shown in Figure 10-5, the `main()` method is included in the `GuiDemo` class.

The first line of code (line 50) in the `main()` method, `GuiDemo demo = new GuiDemo();`, is responsible for creating a new `GuiDemo` object named `demo`. This line causes the `GuiDemo()` constructor to be called. As you saw previously, the `GuiDemo()` constructor creates the graphical user interface by adding `JLabels` and `JButtons` to a `JPanel`. It also assigns `ActionListeners` to the `JButtons`.

Remember, the `main()` method is the first method called when a program executes.

The next step is to create a `JFrame` object named `frame` (line 51), as follows:

```
JFrame frame = new JFrame("GUI Demo");
```

A JFrame is a Window that can have a border, a title bar, and a menu bar. In this example, the string constant "GUI Demo" (in parentheses) is specified as the title for the JFrame title bar.

The next line of code (line 52),

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

is a shortcut technique for adding an event handler to a JFrame. It causes the JFrame window to close when the user clicks the X button on the title bar.

The syntax for accessing panel (the name of the JPanel) is demo.panel because panel is a member of the GuiDemo object named demo.

The last three lines in the main() method (lines 54, 55, and 56) look like this:

```
frame.setContentPane(demo.panel);
frame.pack();
frame.setVisible(true);
```

The **ContentPane** is the container to which you add Components such as JButtons and JLabels. In this case, you want to use JPanel as a ContentPane. To specify this, you pass the name of the JPanel (in this case, demo.panel) to the setContentPane() method.

The method named pack() causes the JFrame to be sized to fit the size of the Components that have been added to it. The method named setVisible() allows users to see the JFrame if it receives true as an argument. Passing false to the setVisible() method keeps the JFrame from being seen.

You might wonder why you would want to create a JFrame that you cannot see. Many Java programs consist of multiple JFrames that are displayed to the user at different times.

The Gui Demo program is now complete. The program is stored in a file named GuiDemo.java along with the other student files for this book. You should compile the program and then execute it to see the JButtons or the JPanel change color when you click the buttons.

Exercise 10-2: Creating a Graphical User Interface in Java

In this exercise, you use what you have learned about creating a graphical user interface to answer Questions 1–4.

1. Write the Java statement that creates a JPanel named bookStorePanel.

   ____

2. Write the Java statement that creates a JButton named saveButton. The JButton should include the text "Save".

   ____

3. Write the Java statement that adds saveButton to the JPanel named bookStorePanel.

   ____

4. Write the Java statement that changes the color of saveButton to green.

   ____

Lab 10-2: Creating a Graphical User Interface in Java

In this lab, you create a graphical user interface in a partially completed Java program. The program should create two JButtons. Display the text Yes on one of the JButtons, and display the text No on the other JButton. You should also create three JLabels. Display the text Do you like GUI programming? Vote Yes or No. on one of the JLabels. Display the text Click here to vote Yes on another JLabel, and display the text Click here to vote No on the third JLabel. Also, add event handlers that cause the background color of the JPanel to change to yellow if a user votes yes and to red if a user votes no. Use the GuiDemo class discussed in this section as a guide.

1. Open the file named JavaQuiz.java using Notepad or the text editor of your choice.
2. Create the three JLabels named labelYes, labelNo, and labelQuestion with the text described above.
3. Create two JButtons named buttonYes and buttonNo with the text described above.
4. Create a JPanel named myPanel.
5. Save the file, JavaQuiz.java, in a directory of your choice.
6. Compile the file JavaQuiz.java.
7. Execute the program.