CHAPTER

6

Graphical User Interface (GUI) and Object-Oriented Design (OOD)

IN THIS CHAPTER, YOU WILL:

- Learn about basic GUI components
- Explore how the GUI components **`JFrame`**, **`JLabel`**, **`JTextField`**, and **`JButton`** work
- Become familiar with the concept of event-driven programming
- Discover events and event handlers
- Explore object-oriented design
- Learn how to identify objects, classes, and members of a class
- Learn about wrapper classes
- Become familiar with the autoboxing and auto-unboxing of primitive data types

Java is equipped with many powerful, yet easy-to-use graphical user interface (GUI) components, such as the input and output dialog boxes you learned about in Chapter 3. You can use these to make your programs attractive and user-friendly. The first half of this chapter introduces you to some basic Java GUI components. Chapter 12 covers GUI in some details.

In Chapter 1, you were introduced to the object-oriented design (OOD) problem-solving methodology. The second half of this chapter outlines a general approach to solving problems using OOD, and provides several examples to clarify this problem-solving methodology.

Graphical User Interface (GUI) Components

In Chapter 3, you learned how to use input and output dialog boxes to input data into a program and show the output of a program. Before introducing the various GUI components, we will use input and output dialog boxes to write a program to determine the area and perimeter of a rectangle. We will then discuss how to use additional GUI components to create a different graphical user interface to determine the area and perimeter of a rectangle.

The program in Example 6-1 prompts the user to input the length and width of a rectangle and then displays its area and perimeter. We will use the method `showInputDialog` to create an input dialog box and the method `showMessageDialog` to create an output dialog box. Recall that these methods are contained in the `class` `JOptionPane` and this class is contained in the package `javax.swing`.

EXAMPLE 6-1

```
// This Java Program determines the area and
// perimeter of a rectangle.

import javax.swing.JOptionPane;

public class Rectangle
{
    public static void main(String[] args)
    {
        double width, length, area, perimeter;                    //Line 1

        String lengthStr, widthStr, outputStr;                    //Line 2

        lengthStr =
              JOptionPane.showInputDialog("Enter the length: "); //Line 3
        length = Double.parseDouble(lengthStr);                   //Line 4
```

```
        widthStr =
               JOptionPane.showInputDialog("Enter the width: "); //Line 5
        width = Double.parseDouble(widthStr);                    //Line 6

        area = length * width;                                   //Line 7
        perimeter = 2 * (length + width);                        //Line 8

        outputStr = "Length: " + length + "\n" +
                    "Width: " + width + "\n" +
                    "Area: " + area + " square units\n" +
                    "Perimeter: " + perimeter + " units\n";      //Line 9

        JOptionPane.showMessageDialog(null, outputStr,
                              "Rectangle",
                              JOptionPane.INFORMATION_MESSAGE);  //Line 10

        System.exit(0);                                          //Line 11
    }
}
```

Sample Run: (Figure 6-1 shows the sample run.)

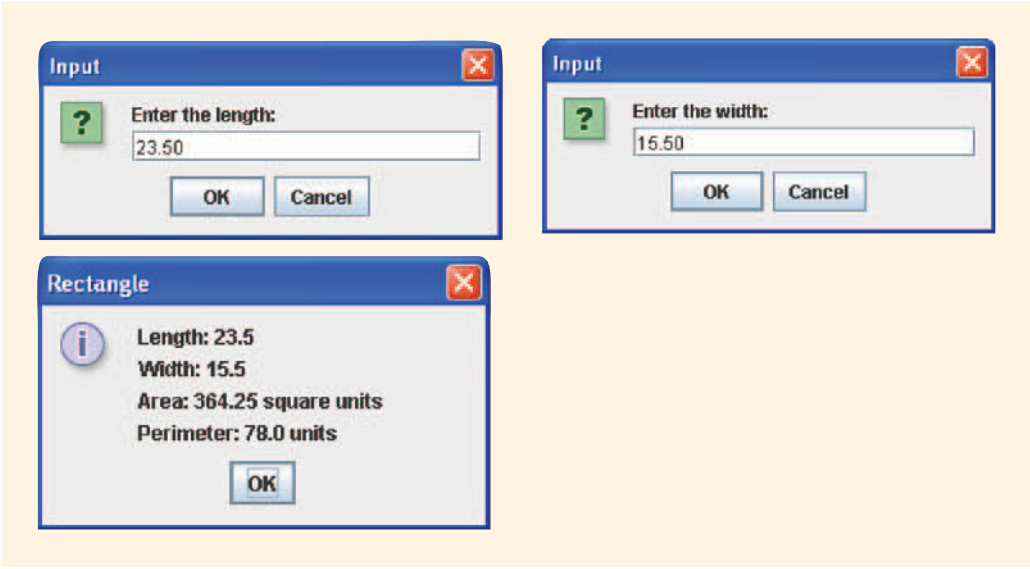

FIGURE 6-1 Sample run for `Rectangle`

The program in Example 6-1 works as follows: The statements in Lines 1 and 2 declare various variables to manipulate the data. The statement in Line 3 displays the first dialog box of the sample run and prompts the user to enter the length of the rectangle. The

entered length is assigned as a string to `lengthStr`. The statement in Line 4 retrieves the length and stores it in the variable `length`.

The statement in Line 5 displays the second dialog box of the sample run and prompts the user to enter the width of the rectangle. The entered width is assigned as a string to `widthStr`. The statement in Line 6 retrieves the width and stores it in the variable `width`.

The statement in Line 7 determines the area, and the statement in Line 8 determines the perimeter of the rectangle. The statement in Line 9 creates the string containing the desired output and assigns it to `outputStr`. The statement in Line 10 uses the output dialog box to display the desired output, which is shown in the third dialog box of the sample run. Finally, the statement in Line 11 terminates the program.

The program in Example 6-1 uses input and output dialog boxes to accomplish its job. When you run this program, you see only one dialog box at a time.

However, suppose that you want the program to display all the input and output in one dialog box, as shown in Figure 6-2.

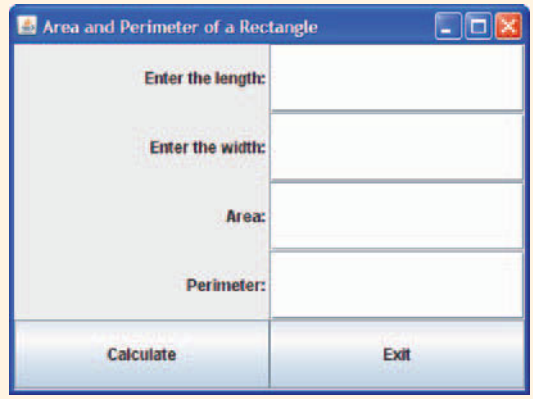

FIGURE 6-2 GUI to find the area and perimeter of a rectangle

In Java terminology, such a dialog box is called a **graphical user interface** (GUI), or simply a **user interface**. In this GUI, the user enters the length and width in the top two white boxes. When the user clicks the `Calculate` button, the program displays the area and the perimeter in their respective locations. When the user clicks the `Exit` button, the program terminates.

In this interface, the user can:

- See the entire input and output simultaneously
- Input values for length and width, in any order of preference
- Input values that can be corrected after entering them and before clicking the `Calculate` button
- Enter another set of input values and click the `Calculate` button to obtain the area and perimeter of another rectangle

The interface shown in Figure 6-2 contains various Java GUI components that are labeled in Figure 6-3.

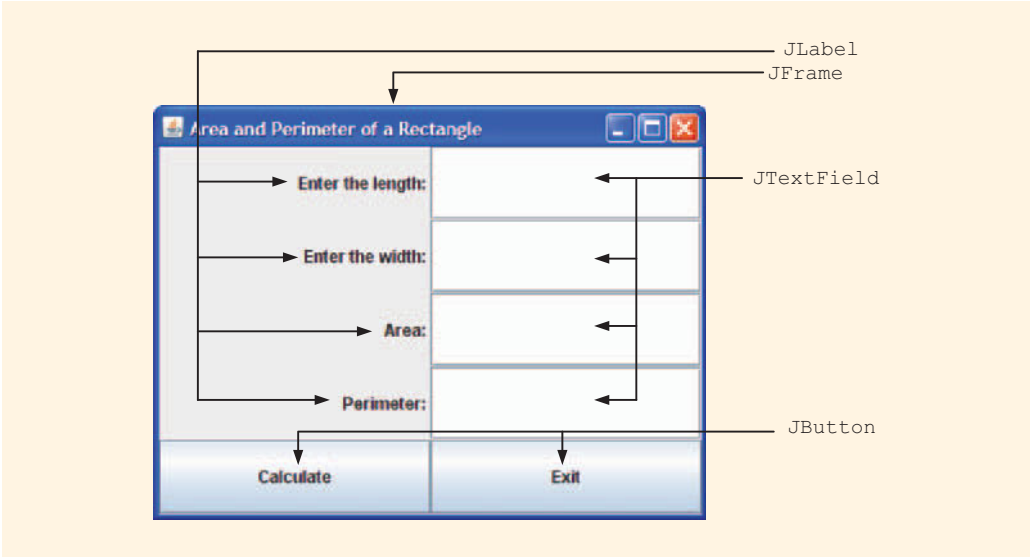

FIGURE 6-3 Java GUI components

As you can see in Figure 6-3, the white areas used to get the input and show the results are called `JTextFields`. The labels for these text fields, such as `Enter the length:`, are called `JLabels`; the buttons `Calculate` and `Exit` are each called a `JButton`. All these components are placed in a window, called `JFrame`.

Creating this type of user interface is not difficult. Java has done all the work; you merely need to learn how to use the tools provided by Java to create such an interface. For example, to create an interface like the one shown in Figures 6-2 and 6-3 that contains labels, text fields, buttons, and windows, you need to learn how to write the statements

that create these components. The next sections describe how to create the following GUI components:

- Windows
- Labels
- Text fields
- Buttons

GUI components, such as labels, are placed in an area called the **content pane** of the window. You can think of a content pane as the inner area of the window, below the title bar and inside the border. You will also learn how to place these GUI components in the content pane of a window.

In Figure 6-2, when you click the `Calculate` button, the program displays the area and perimeter of the rectangle you have specified. This means that clicking the `Calculate` button causes the program to execute the code to calculate the area and perimeter and then display the results. When the `Calculate` button is clicked, we say that an **event** has occurred. The Java system is very prompt in listening for the events generated by a program and then reacting to those events. This chapter will describe how to write the code that needs to be executed when a particular event occurs, such as when a button is clicked. So, in addition to creating windows, labels, text fields, and buttons, you will learn:

- How to access the content pane
- How to create event listeners
- How to process or handle events

We begin by describing how to create a window.

Creating a Window

GUI components such as windows and labels are, in fact, objects. Recall that an object is an instance of a particular class. Therefore, these components (objects) are instances of a particular class type. `JFrame` is a `class` and the GUI component `window` can be created by using a `JFrame` object. Various attributes are associated with a window. For example:

- Every window has a title.
- Every window has width and height.

JFrame

The `class` `JFrame` provides various methods to control the attributes of a window. For example, it has methods to set the window title and methods to specify the height and width of the window. Table 6-1 describes some of the methods provided by the `class` `JFrame`.

TABLE 6-1 Some Methods Provided by the `class JFrame`

Method / Description / Example

`public JFrame()`
`  //This is used when an object of type JFrame is`
`  //instantiated and the window is created without any title.`
`  //Example: JFrame myWindow = new JFrame();`
`  //  myWindow is a window with no title`

`public JFrame(String s)`
`  //This is used when an object of type JFrame is`
`  //instantiated and the title specified by the string s.`
`  //Example: JFrame myWindow = new JFrame("Rectangle");`
`  //  myWindow is a window with the title Rectangle`

`public void setSize(int w, int h)`
`  //Method to set the size of the window.`
`  //Example: The statement`
`  //  myWindow.setSize(400, 300);`
`  //  sets the width of the window to 400 pixels and`
`  //  the height to 300 pixels.`

`public void setTitle(String s)`
`  //Method to set the title of the window.`
`  //Example: myWindow.setTitle("Rectangle");`
`  //  sets the title of the window to Rectangle.`

`public void setVisible(boolean b)`
`  //Method to display the window in the program. If the value of b is`
`  //true, the window will be displayed on the screen.`
`  //Example: myWindow.setVisible(true);`
`  //  After this statement executes, the window will be shown`
`  //  during program execution.`

`public void setDefaultCloseOperation(int operation)`
`  //Method to determine the action to be taken when the user clicks`
`  //on the window closing button, ×, to close the window.`
`  //Choices for the parameter operation are the named constants –`
`  //EXIT_ON_CLOSE, HIDE_ON_CLOSE, DISPOSE_ON_CLOSE, and`
`  //DO_NOTHING_ON_CLOSE. The named constant EXIT_ON_CLOSE is defined`
`  //in the class JFrame. The last three named constants are defined in`
`  //javax.swing.WindowConstants.`
`  //Example: The statement`
`  //  setDefaultCloseOperation(EXIT_ON_CLOSE);`
`  //sets the default close option of the window closing to close the`
`  //window and terminate the program when the user clicks the`
`  //window closing button, ×.`

TABLE 6-1 Some Methods Provided by the `class JFrame` (continued)

Method / Description / Example
`public void addWindowListener(WindowEvent e)` `  //Method to register a window listener object to a JFrame.`

NOTE The `class JFrame` also contains methods to set the color of a window. Chapter 12 describes these methods.

There are two ways to make an application program create a window. The first way is to declare an object of type `JFrame`, instantiate the object, and then use various methods to manipulate the window. In this case, the object created can use the various applicable methods of the class.

The second way is to create the class containing the application program by *extending* the definition of the `class JFrame`; that is, the class containing the application program is built "on top of" the `class JFrame`. In Java, this way of creating a class uses the mechanism of **inheritance**. Inheritance means that a new class can be derived from or based on an already existing class. The new class "inherits" features such as methods from the existing class, which saves a lot of time for programmers. For example, we could define a new `class RectangleProgram` that would extend the definition of `JFrame`. The class `RectangleProgram` would be able to use the variables and methods from `JFrame`, and also add some functionality of its own (such as the ability to calculate the area and perimeter of a rectangle).

When you use inheritance, the class containing your application program will have more than one method. In addition to the method `main`, you will have at least one other method that will be used to create a window object containing the required GUI components (such as labels and text fields). This additional method is a special type of method called a **constructor**. A constructor is a method of a class that is automatically executed when an object of the class is created. Typically, a constructor is used to initialize an object. The name of the constructor is always the same as the name of the class. For example, the constructor for the `class RectangleProgram` would be named `RectangleProgram`.

NOTE Chapter 10 discusses the principles of inheritance in detail. Constructors are covered in detail in Chapter 8.

Because inheritance is an important concept in programming languages such as Java, we will use the second way of creating a window. We will extend the definition of the

class JFrame by using the modifier extends. For example, the definition of the class RectangleProgram, containing the application program to calculate the area and perimeter of a rectangle, is as follows:

```
public class RectangleProgram extends JFrame
{
   public RectangleProgram()       //constructor
   {
       //Necessary code
   }

   public static void main(String[] args)
   {
       //Code for the method main
   }
}
```

In Java, extends is a reserved word. The remainder of this section describes the necessary code to create a window.

An important property of inheritance is that the class (called a **subclass**) that extends the definition of an existing class (called a **superclass**) inherits all the properties of the superclass. For example, all public methods of the superclass can be *directly* accessed in the subclass. In our example, the class RectangleProgram is a subclass of the class JFrame, so it can access the public methods of the class JFrame. Therefore, to set the title of the window to Area and Perimeter of a Rectangle, you use the method setTitle of the class JFrame as follows:

```
setTitle("Area and Perimeter of a Rectangle");          //Line 1
```

Similarly, the statement:

```
setSize(400, 300);                                      //Line 2
```

sets the window's width to 400 pixels and its height to 300 pixels. (A **pixel** is the smallest unit of space on your screen. The term pixel stands for *picture element*.) Note that since the pixel size depends on the current monitor setting, it is impossible to predict the exact width and height of a window in centimeters or inches.

Next, to display the window, you must invoke the method setVisible. The following statement accomplishes this:

```
setVisible(true);                                       //Line 3
```

To terminate the application program when the user closes the window, use the following statement (as described in Table 6-1):

```
setDefaultCloseOperation(EXIT_ON_CLOSE);                //Line 4
```

The statements in Lines 1, 2, 3, and 4 will be placed in the constructor (that is, in the method whose heading is public RectangleProgram()). Thus, you can write the constructor as follows:

```
public RectangleProgram()
{
    setTitle("Area and Perimeter of a Rectangle");
    setSize(400, 300);
    setVisible(true);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
}
```

You could create a window by using an object of type `JFrame`. However, for our program, if we do so, then the window created will not have a title or the required size unless we specify the necessary statements similar to the ones in the preceding code. Because `RectangleProgram` is also a `class`, we can create objects of type `RectangleProgram`. Because the `class RectangleProgram extends` the definition of the `class JFrame`, it inherits the properties of the `class JFrame`. If we create an object of type `RectangleProgram`, not only do we create a window, but the created window will also have a title and a specific size, and the window will be displayed when the program executes.

Consider the following statement:

```
RectangleProgram rectObject = new RectangleProgram();  //Line 5
```

This statement creates the object `rectObject` of type `RectangleProgram`.

The statement in Line 5 causes the window shown in Figure 6-4 to appear on the screen.

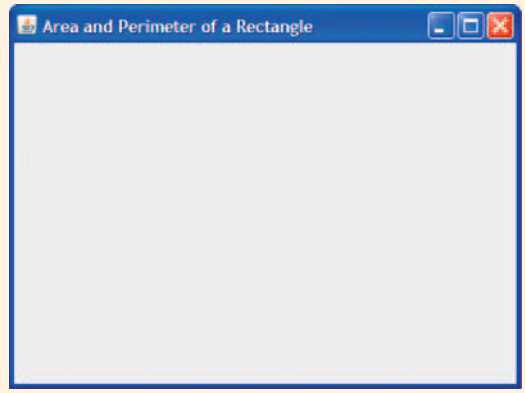

FIGURE 6-4 Window with the title `Area and Perimeter of a Rectangle`

You can close the window in Figure 6-4 by clicking the "close" button, the button containing the ×, in the upper-right corner. The window in Figure 6-4 is empty because we have not yet created labels, text fields, and so on.

The program to create the window shown in Figure 6-4 uses the `class JFrame`; this class is contained in the package `javax.swing`. Therefore, the program must include either of the following two statements:

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

or:

```
import javax.swing.JFrame;
```

After making the minor changes in the statements described in this section, the program to create the window shown in Figure 6-4 is as follows:

```
//Java program to create a window.

import javax.swing.*;

public class RectangleProgramOne extends JFrame
{
    private static final int WIDTH = 400;
    private static final int HEIGHT = 300;

    public RectangleProgramOne()
    {
        setTitle("Area and Perimeter of a Rectangle");
        setSize(WIDTH, HEIGHT);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args)
    {
        RectangleProgramOne rectProg = new RectangleProgramOne();
    }
}
```

Notice that the named constants `WIDTH` and `HEIGHT` are declared with the modifier `private`. This is because we want these named constants to be used only within the `class` `RectangleProgram`. In general, if a named constant, variable, or method is to be used only within the specified `class`, then it is declared with the modifier `private`. Also, note that `private` is a reserved word in Java. (Chapter 8 discusses the modifier `private` in detail.)

(Note that in the preceding program we have changed the name of the `class` to `RectangleProgramOne`. This is because we have not yet added all the GUI components to the program. After adding labels, we will call it `class` `RectangleProgramTwo`, and so on. After adding all the necessary GUI components, we will call it `class` `RectangleProgram`. The Web site, *www.course.com*, and the CD accompanying this book contain all these programs.)

Let's review the important points introduced in this section:

- The preceding program has exactly one class: `RectangleProgramOne`.
- The `class` `RectangleProgramOne` contains the constructor `RectangleProgramOne` and the `main` method.

- You created the new class RectangleProgramOne by extending the existing class, JFrame. Therefore, JFrame is the superclass of RectangleProgramOne, and RectangleProgramOne is a subclass of JFrame.
- Whenever there is a superclass–subclass relationship, the subclass inherits all the data members and methods of the superclass. The methods setTitle, setSize, setVisible, and setDefaultCloseOperation are methods of the class JFrame, and these methods can be inherited by its subclasses.

The next few sections describe how to create GUI labels, text fields, and buttons, which can all be placed in the content pane of a window. Before you can place GUI components in the content pane, you must learn how to access the content pane.

Getting Access to the Content Pane

If you can visualize JFrame as a window, think of the content pane as the inner area of the window (below the title bar and inside the border). The class JFrame has the method getContentPane that you can use to access the content pane of the window. However, the class JFrame does not have the necessary tools to manage the components of the content pane. The components of the content pane are managed by declaring a reference variable of the Container type and then using the method getContentPane, as shown next.

Consider the following statements:

```
Container pane;                          //Line 1
pane = getContentPane();                 //Line 2
```

The statement in Line 1 declares pane to be a reference variable of the Container type. The statement in Line 2 gets the content pane of the window as a container, that is, the reference variable pane now points to the content pane. You can now access the content pane to add GUI components to it by using the reference variable pane.

The statements in Lines 1 and 2 can be combined into one statement:

```
Container pane = getContentPane();     //Line 3
```

If you look back at Figure 6-2, you will see that the labels, text fields, and buttons are arranged in five rows and two columns. To control the placement of GUI components in the content pane, you set the layout of the content pane. The layout used in Figure 6-2 is called the grid layout. The class Container provides the method setLayout, as described in Table 6-2, to set the layout of the content pane. To add components such as labels and text fields to the content pane, you use the method add of the class Container, which is also described in Table 6-2.

TABLE 6-2 Some Methods of the `class Container`

Method / Description
`public void add(Object obj)` `  //Method to add an object to the pane.`
`public void setLayout(Object obj)` `  //Method to set the layout of the pane.`

The `class Container` is contained in the package `java.awt`. To use this `class` in your program, you need to include one of the following statements:

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

or:

```
import java.awt.Container;
```

As noted earlier, the method `setLayout` is used to set the layout of the content pane, `pane`. To set the layout of the container to a grid, you use the `class GridLayout`. Consider the following statement:

```
pane.setLayout(new GridLayout(5, 2));
```

This statement creates an object belonging to the `class GridLayout` and assigns that object as the layout of the content pane, `pane`, by invoking the `setLayout` method. Moreover, this statement sets the layout of the content pane, `pane`, to five rows and two columns. This allows you to add 10 components arranged in five rows and two columns.

Note that the `GridLayout` manager arranges GUI components in a matrix formation with the number of rows and columns defined by the constructor and that the components are placed left to right, starting with the first row. For example, in the statement `pane.setLayout(new GridLayout(5, 2));`, the expression `new GridLayout(5, 2)`, invokes the constructor of the `class GridLayout` and sets the number of rows to 5 and the number of columns to 2. Also, in this chapter, we only discuss the `GridLayout` manager; additional layout managers are discussed in Chapter 12. Layout managers allow you to manage GUI components in a content pane.

If you do not specify a layout, Java uses a default layout. If you specify a layout, you must set the layout before adding any components. Once the layout is set, you can use the method `add` to add the components to the pane; this process is described in the next section.

JLabel

Now you will learn how to create labels and add them to the pane. We assume the following statements:

```
Container pane = getContentPane();
pane.setLayout(new GridLayout(4, 1));
```

Labels are objects of a particular **class** type. The Java **class** that you use to create labels is **JLabel**. Therefore, to create labels, you instantiate objects of type **JLabel**. The **class** **JLabel** is contained in the package **javax.swing**.

Just like a window, various attributes are associated with a label. For example, every label has a title, width, and height. The **class** **JLabel** contains various methods to control the display of labels. Table 6-3 describes some of the methods provided by the **class** **JLabel**.

TABLE 6-3 Some Methods Provided by the **class** JLabel

Method / Description/ Example

`public JLabel(String str)`
`  //Constructor to create a label with left-aligned text specified`
`  //by str.`
`  //Example: JLabel lengthL;`
`  //          lengthL = new JLabel("Enter the length:")`
`  //    Creates the label lengthL with the title Enter the length:`

`public JLabel(String str, int align)`
`  //Constructor to create a label with the text specified by str.`
`  //    The value of align can be any one of the following:`
`  //    SwingConstants.LEFT, SwingConstants.RIGHT,`
`  //    SwingConstants.CENTER`
`  //Example:`
`  //  JLabel lengthL;`
`  //  lengthL = new JLabel("Enter the length:",`
`  //                       SwingConstants.RIGHT);`
`  //  The label lengthL is right aligned.`

`public JLabel(String t, Icon icon, int align)`
`  //Constructs a JLabel with both text and an icon.`
`  //The icon is to the left of the text.`

`public JLabel(Icon icon)` `  //Constructs a JLabel with an icon.`

NOTE In Table 6-3, **SwingConstants.LEFT**, **SwingConstants.RIGHT**, and **SwingConstants.CENTER** are constants defined in the **class** **SwingConstants**. They specify whether to set the string describing the label as left-justified, right-justified, or centered.

Consider the statements:

```
JLabel lengthL;
lengthL = new JLabel("Enter the length:", SwingConstants.RIGHT);
```

After these statements execute, the label in Figure 6-5 is created.

FIGURE 6-5 JLabel with the text `Enter the length:`

Now consider the following statements:

```
private JLabel lengthL, widthL, areaL, perimeterL;          //Line 1

lengthL =
    new JLabel("Enter the length: ", SwingConstants.RIGHT); //Line 2

widthL =
    new JLabel("Enter the width: ", SwingConstants.RIGHT);  //Line 3

areaL = new JLabel("Area: ", SwingConstants.RIGHT);         //Line 4

perimeterL =
    new JLabel("Perimeter: ", SwingConstants.RIGHT);        //Line 5
```

The statement in Line 1 declares four reference variables, `lengthL`, `widthL`, `areaL`, and `perimeterL`, of the `JLabel` type. The statement in Line 2 instantiates the object `lengthL`, assigns it the title `Enter the length:`, and sets the title alignment to right-justified. The statements in Lines 3 through 5 instantiate the objects `widthL`, `areaL`, and `perimeterL` with appropriate titles and text alignment.

Next, we add these labels to the `pane` declared at the beginning of this section. The following statements accomplish this. (Recall from the preceding section that we use the method `add` to add components to a `pane`.)

```
pane.add(lengthL);
pane.add(widthL);
pane.add(areaL);
pane.add(perimeterL);
```

Because we have specified a grid layout for the `pane` with four rows and one column, the label `lengthL` is added to the first row, the label `widthL` is added to the second row, and so on.

Now that you know how to add the components to the `pane`, you can put together the program to create these labels. `RectangleProgramTwo` builds on the `RectangleProgramOne` of the preceding section and, like `RectangleProgramOne`, is a subclass of `JFrame`.

```
//Java program to create a window and place four labels

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

public class RectangleProgramTwo extends JFrame
{
   private static final int WIDTH = 400;
   private static final int HEIGHT = 300;

   private JLabel lengthL, widthL, areaL, perimeterL;

   public RectangleProgramTwo()
   {
       setTitle("Area and Perimeter of a Rectangle");

       lengthL =
          new JLabel("Enter the length: ", SwingConstants.RIGHT);
       widthL =
          new JLabel("Enter the width: ", SwingConstants.RIGHT);
       areaL = new JLabel("Area: ", SwingConstants.RIGHT);
       perimeterL =
          new JLabel("Perimeter: ", SwingConstants.RIGHT);

       Container pane = getContentPane();
       pane.setLayout(new GridLayout(4, 1));

       pane.add(lengthL);
       pane.add(widthL);
       pane.add(areaL);
       pane.add(perimeterL);

       setSize(WIDTH, HEIGHT);
       setVisible(true);
       setDefaultCloseOperation(EXIT_ON_CLOSE);
   }

   public static void main(String[] args)
   {
       RectangleProgramTwo rectObject = new RectangleProgramTwo();
   }
}
```

Sample Run: (Figure 6-6 shows a sample run.)

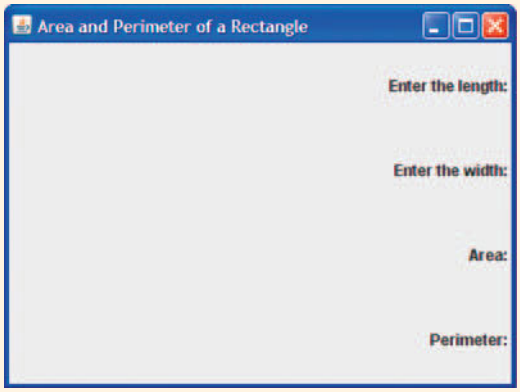

FIGURE 6-6 Sample run for `RectangleProgramTwo`

Now you are ready to create and place the text fields and buttons. The techniques for creating and placing components, such as `JTextField` and `JButton`, in a container are similar to the ones used for `JLabel`, and are described in the next two sections.

JTextField

As you may recall, text fields are objects belonging to the `class` `JTextField`. Therefore, you can create a text field by declaring a reference variable of type `JTextField` followed by an instantiation of the object.

Table 6-4 describes some of the methods of the `class` `JTextField`.

TABLE 6-4 Some Methods of the `class` `JTextField`

Method / Description
`public JTextField(int columns)` `  //Constructor to set the size of the text field.`

`public JTextField(String str)`
`  //Constructor to initialize the object with the text specified`
`  //by str.`

TABLE 6-4 Some Methods of the `class` `JTextField` (continued)

Method / Description

`public JTextField(String str, int columns)`
`  //Constructor to initialize the object with the text specified`
`  //by str and to set the size of the text field.`

`public void setText(String str)`
`  //Method to set the text of the text field to the string specified`
`  //by str.`

`public String getText()` `  //Method to return the text contained in the text field.`

`public void setEditable(boolean b)`
`  //If the value of the boolean variable b is false, the user cannot`
`  //type in the text field.`
`  //In this case, the text field is used as a tool to display`
`  //the result.`

`public void addActionListener(ActionListener obj)` `  //Method to register a listener object to a JTextField.`

Consider the following statements:

```
private JTextField lengthTF, widthTF, areaTF,
                   perimeterTF;                  //Line 1

lengthTF = new JTextField(10);                   //Line 2
widthTF = new JTextField(10);                    //Line 3
areaTF = new JTextField(10);                     //Line 4
perimeterTF = new JTextField(10);                //Line 5
```

The statement in Line 1 declares four reference variables, `lengthTF`, `widthTF`, `areaTF`, and `perimeterTF`, of type `JTextField`. The statement in Line 2 instantiates the object `lengthTF` and sets the width of this text field to 10 characters. That is, this text field can display no more than 10 characters. The meaning of the other statements is similar.

Placing these objects involves using the `add` method of the `class` `Container` as described in the previous section. The following statements add these components to the container:

```
pane.add(lengthTF);
pane.add(widthTF);
pane.add(areaTF);
pane.add(perimeterTF);
```

The container `pane` now would contain eight objects—four labels and four text fields. We want to place the object `lengthTF` adjacent to the label `lengthL` in the same row, and use similar placements for the other objects. So we need to expand the grid layout to four rows and two columns. The following statements create the required grid layout and the necessary objects:

```
pane.setLayout(new GridLayout(4, 2));
pane.add(lengthL);
pane.add(lengthTF);
pane.add(widthL);
pane.add(widthTF);
pane.add(areaL);
pane.add(areaTF);
pane.add(perimeterL);
pane.add(perimeterTF);
```


The following program, `RectangleProgramThree`, summarizes our discussion so far:

```
//Java program to create a window
//and place four labels and four text fields

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

public class RectangleProgramThree extends JFrame
{
   private static final int WIDTH = 400;
   private static final int HEIGHT = 300;

   private JLabel lengthL, widthL, areaL, perimeterL;
   private JTextField lengthTF, widthTF, areaTF,
                      perimeterTF;

   public RectangleProgramThree()
   {
      setTitle("Area and Perimeter of a Rectangle");

      lengthL =
          new JLabel("Enter the length: ", SwingConstants.RIGHT);
      widthL =
          new JLabel("Enter the width: ", SwingConstants.RIGHT);
      areaL =
          new JLabel("Area: ", SwingConstants.RIGHT);
      perimeterL =
          new JLabel("Perimeter: ", SwingConstants.RIGHT);
```

```
        lengthTF = new JTextField(10);
        widthTF = new JTextField(10);
        areaTF = new JTextField(10);
        perimeterTF = new JTextField(10);

        Container pane = getContentPane();
        pane.setLayout(new GridLayout(4, 2));

        pane.add(lengthL);
        pane.add(lengthTF);
        pane.add(widthL);
        pane.add(widthTF);
        pane.add(areaL);
        pane.add(areaTF);
        pane.add(perimeterL);
        pane.add(perimeterTF);

        setSize(WIDTH, HEIGHT);
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public static void main(String[] args)
    {
        RectangleProgramThree rectObject =
                            new RectangleProgramThree();
    }
}
```

Sample Run: (Figure 6-7 shows the sample run.)

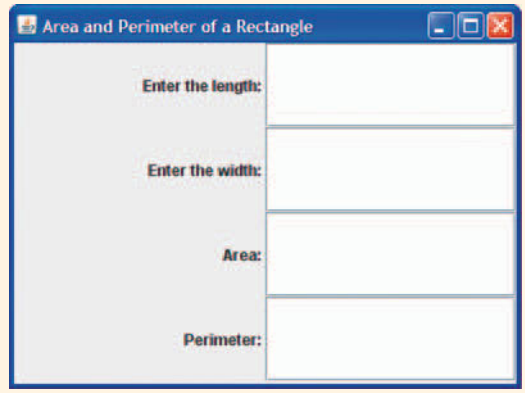

FIGURE 6-7 Sample run for `RectangleProgramThree`

To complete the design of the user interface, we will discuss how to create buttons.

JButton

To create a button, Java provides the class JButton. Thus, to create objects belonging to the class JButton, we use a technique similar to the one we used to create instances of JLabel and JTextField. Table 6-5 shows some methods of the class JButton.

TABLE 6-5 Commonly Used Methods of the class JButton

Method / Description

`public JButton(Icon ic)`
`  //Constructor to initialize the button object with the icon`
`  //specified by ic.`

`public JButton(String str)`
`  //Constructor to initialize the button object to the text specified`
`  //by str.`

`public JButton(String str, Icon ic)`
`  //Constructor to initialize the button object to the text specified`
`  //by str and the icon specified by ic.`

`public void setText(String str)` `  //Method to set the text of the button to the string specified by str.`
`public String getText()` `  //Method to return the text contained in the button.`
`public void addActionListener(ActionListener obj)` `  //Method to register a listener object to the button object.`

The following three lines will create two buttons, Calculate and Exit, shown earlier in Figure 6-2:

```
JButton calculateB, exitB;                //Line 1

calculateB = new JButton("Calculate");    //Line 2
exitB = new JButton("Exit");              //Line 3
```

The statement in Line 1 declares calculateB and exitB to be reference variables of type JButton. The statement in Line 2 instantiates the button object calculateB and sets the text for the button to the string Calculate. Similarly, the statement in Line 3 instantiates the button object exitB and sets the text for exitB to the string Exit.

The buttons `calculateB` and `exitB` can be placed into the container `pane` by using the method `add`. The following statements add these buttons to the `pane`:

```
pane.add(calculateB);
pane.add(exitB);
```

Now you have two more objects in the container, so you need to modify the `GridLayout` to accommodate five rows and two columns, and then add all the components. The following statements create the required grid layout and add the labels, text fields, and buttons to the container `pane`:

```
pane.setLayout(new GridLayout(5,2));//specify the layout

pane.add(lengthL);                  //add the label lengthL
pane.add(lengthTF);                 //add the text field lengthTF
pane.add(widthL);                   //add the label widthL
pane.add(widthTF);                  //add the text field widthTF
pane.add(areaL);                    //add the label areaL
pane.add(areaTF);                   //add the text field areaTF
pane.add(perimeterL);               //add the label perimeterL
pane.add(perimeterTF);              //add the text field perimeterTF
pane.add(calculateB);               //add the button calculateB
pane.add(exitB);                    //add the button exitB
```

Notice that the preceding `add` statements place the components from left to right and from top to bottom.

HANDLING AN EVENT

You have now learned how to create a window, how to create a container, and how to create labels, text fields, and buttons.

Now that you can create a button, such as `calculateB`, you need to specify how such a button should behave when you click it. For example, when you click the button `calculateB`, you want the program to calculate the area and perimeter of the rectangle and display these values in their respective text fields. Similarly, when you click the button `exitB`, the program should terminate.

Clicking a `JButton` creates an event, known as an **action event**, which sends a message to another object, known as an **action listener**. When the listener receives the message, it performs some action. Sending a message or an event to a listener object simply means that some method in the listener object is invoked with the event as the argument. This invocation happens automatically; you will not see the code corresponding to the method invocation. However, you must specify two things:

- For each `JButton`, you must specify the corresponding listener object. In Java, this is known as **registering** the listener.
- You must define the methods that will be invoked when the event is sent to the listener. Normally, you will write these methods and you will never write the code for invocation.

Java provides various classes to handle different kinds of events. The action event is handled by the `class` `ActionListener`, which contains only the method `actionPerformed`. In the method `actionPerformed`, you include the code that you want the system to execute when an action event is generated.

The `class` `ActionListener` that handles the action event is a special type of class, called an `interface`. In Java, `interface` is a reserved word. Roughly speaking, an `interface` is a class that contains only the method headings, and each method heading is terminated with a semicolon. For example, the definition of the `interface` `ActionListener` containing the method `actionPerformed` is:


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

Because the method `actionPerformed` does not contain a body, Java does not allow you to instantiate an object of type `ActionListener`. So how do you register an action listener with the object `calculateB`?

One way is as follows (there are other ways not discussed here): Because you cannot instantiate an object of type `ActionListener`, first you need to create a class on top of `ActionListener` so that the required object can be instantiated. The class created must provide the necessary code for the method `actionPerformed`. You will create the `class` `CalculateButtonHandler` to handle the event generated by clicking the button `calculateB`.

The `class` `CalculateButtonHandler` is created on top of the `interface` `ActionListener`. The definition of the `class` `CalculateButtonHandler` is:

```
private class CalculateButtonHandler implements
                                        ActionListener  //Line 1
{
   public void actionPerformed(ActionEvent e)          //Line 2
   {
      //The code for calculating the area and the perimeter
      //and displaying these quantities goes here
   }
}
```

Notice the following:

- The `class` `CalculateButtonHandler` starts with the modifier `private`. This is because you want this class to be used only within your `RectangleProgram`.
- This class uses another modifier, `implements`. This is how you build classes on top of classes that are interfaces. Notice that you have not yet provided the code for the method `actionPerformed`. You will do that shortly.

In Java, `implements` is a reserved word.

Next, we illustrate how to create a listener object of type `CalculateButtonHandler`. Consider the following statements:

```
CalculateButtonHandler cbHandler;

cbHandler = new CalculateButtonHandler();  //instantiate the object
```

As described, these statements create the listener object. Having created a listener, you next must associate (or in Java terminology, register) this handler with the corresponding `JButton`. The following line of code registers `cbHandler` as the listener object of `calculateB`:

```
calculateB.addActionListener(cbHandler);
```

The complete definition of the `class` `CalculateButtonHandler`, including the code for the method `actionPerformed`, is:

```
private class CalculateButtonHandler implements
                                      ActionListener   //Line 1
{
   public void actionPerformed(ActionEvent e)         //Line 2
   {
      double width, length, area, perimeter;          //Line 3

      length
           = Double.parseDouble(lengthTF.getText());  //Line 4
      width
           = Double.parseDouble(widthTF.getText());   //Line 5
      area = length * width;                          //Line 6
      perimeter = 2 * (length + width);               //Line 7

      areaTF.setText("" + area);                      //Line 8
      perimeterTF.setText("" + perimeter);            //Line 9
   }
}
```

In the preceding program segment, Line 1 declares the `class` `CalculateButtonHandler` and makes it an action listener by including the phrase `implements` `ActionListener`. Note that all of this code is just a new class definition.

This class has one method; Line 2 is the first statement of that method. Let us look at the statement in Line 4:

```
length = Double.parseDouble(lengthTF.getText());
```

The length of the rectangle is stored in the text field `lengthTF`. We use the method `getText` to retrieve the string from this text field, specifying the length. Now the value of the expression `lengthTF.getText()` is the length, but it is in a string form. So we need to use the method `parseDouble` to convert the length string into an equivalent decimal number. The length is then stored in the variable `length`. The statement in Line 5 works similarly for the width.

The statements in Lines 6 and 7 compute the area and the perimeter, respectively. The statement in Line 8 uses the method `setText` of the `class` `JTextField` to display the area. Because `setText` requires that the argument be a string, you need to convert the value of the variable `area` into a string. The easiest way to do this is to concatenate the value of `area` to an empty string. Similar conventions apply for the statement in Line 9.

It follows that the method `actionPerformed` displays the area and perimeter in the corresponding `JTextFields`.

Before creating an action listener for the `JButton exitB`, let us summarize what we've done so far to create and register an action event listener:

1. Created a class that implements the `interface` `ActionListener`. For example, for the `JButton calculateB` we created the `class` `CalculateButtonHandler`.
2. Provided the definition of the method `actionPerformed` within the class that you created in Step 1. The method `actionPerformed` contains the code that the program executes when a specific event is generated. For example, when you click the `JButton calculateB`, the program should calculate and display the area and perimeter of the rectangle.
3. Created and instantiated an object of the class type created in Step 1. For example, for the `JButton calculateB` we created the object `cbHandler`.
4. Registered the event handler created in Step 3 with the object that generates an action event using the method `addActionListener`. For example, for `JButton calculateB` the following statement registers the object `cbHandler` to listen and register the action event:

   ```
   calculateB.addActionListener(cbHandler);
   ```

We can now repeat these four steps to create and register the action listener with the `JButton exitB`.

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

The following statements create the action listener object for the button `exitB`:

```
ExitButtonHandler ebHandler;

ebHandler = new ExitButtonHandler();
exitB.addActionListener(ebHandler);
```

The `interface ActionListener` is contained in the package `java.awt.event`. Therefore, to use this interface to handle events, your program must include the statement:

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

or:

```
import java.awt.event.ActionListener;
```

The complete program to calculate the perimeter and area of a rectangle is:

```
//Given the length and width of a rectangle, this Java
//program determines its area and perimeter.

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

public class RectangleProgram extends JFrame
{
   private JLabel lengthL, widthL, areaL, perimeterL;

   private JTextField lengthTF, widthTF, areaTF, perimeterTF;

   private JButton calculateB, exitB;

   private CalculateButtonHandler cbHandler;
   private ExitButtonHandler ebHandler;

   private static final int WIDTH = 400;
   private static final int HEIGHT = 300;

   public RectangleProgram()
   {
            //Create the four labels
      lengthL = new JLabel("Enter the length: ",
                                  SwingConstants.RIGHT);
```

```
        widthL = new JLabel("Enter the width: ",
                                    SwingConstants.RIGHT);
        areaL = new JLabel("Area: ", SwingConstants.RIGHT);
        perimeterL = new JLabel("Perimeter: ",
                                    SwingConstants.RIGHT);

                //Create the four text fields
        lengthTF = new JTextField(10);
        widthTF = new JTextField(10);
        areaTF = new JTextField(10);
        perimeterTF = new JTextField(10);

                //Create Calculate Button
        calculateB = new JButton("Calculate");
        cbHandler = new CalculateButtonHandler();
        calculateB.addActionListener(cbHandler);

                //Create Exit Button
        exitB = new JButton("Exit");
        ebHandler = new ExitButtonHandler();
        exitB.addActionListener(ebHandler);

                //Set the title of the window
        setTitle("Area and Perimeter of a Rectangle");

                //Get the container
        Container pane = getContentPane();

                //Set the layout
        pane.setLayout(new GridLayout(5, 2));

                //Place the components in the pane
        pane.add(lengthL);
        pane.add(lengthTF);
        pane.add(widthL);
        pane.add(widthTF);
        pane.add(areaL);
        pane.add(areaTF);
        pane.add(perimeterL);
        pane.add(perimeterTF);
        pane.add(calculateB);
        pane.add(exitB);

                //Set the size of the window and display it
        setSize(WIDTH, HEIGHT);
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
```

```
   private class CalculateButtonHandler implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         double width, length, area, perimeter;

         length = Double.parseDouble(lengthTF.getText());
         width = Double.parseDouble(widthTF.getText());
         area = length * width;
         perimeter = 2 * (length + width);

         areaTF.setText("" + area);
         perimeterTF.setText("" + perimeter);
      }
   }

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

   public static void main(String[] args)
   {
       RectangleProgram rectObject = new RectangleProgram();
   }
}
```

Sample Run: (Figure 6-8 shows the sample run.)

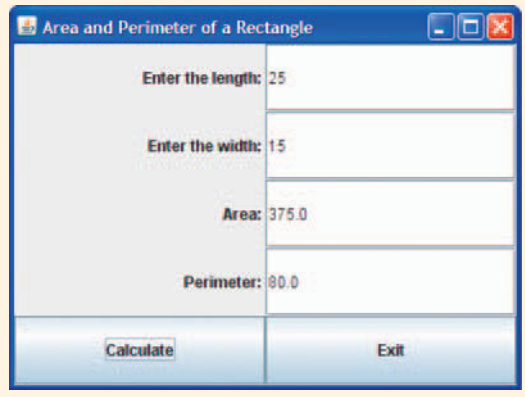

FIGURE 6-8 Sample run for the final `RectangleProgram`

PROGRAMMING EXAMPLE: Temperature Conversion

Write a program that creates the GUI shown in Figure 6-9, to convert the temperature from Fahrenheit to Celsius and from Celsius to Fahrenheit.

FIGURE 6-9 GUI for the temperature conversion program

When the user enters the temperature in the text field adjacent to the label `Temp in Celsius` and presses the `Enter` key, the program displays the equivalent temperature in the text field adjacent to the label `Temp in Fahrenheit`. Similarly, when the user enters the temperature in Fahrenheit and presses the `Enter` key, the program displays the equivalent temperature in Celsius.

Input: Temperature in Fahrenheit or Celsius

Output: Temperature in Celsius if the input is Fahrenheit; the temperature in Fahrenheit if the input is Celsius

PROBLEM ANALYSIS, GUI DESIGN, AND ALGORITHM DESIGN

Suppose that the variable `celsius` represents the temperature in Celsius and the variable `fahrenheit` represents the temperature in Fahrenheit. If the user enters the temperature in Fahrenheit, the formula for calculating the equivalent temperature in Celsius is:

```
celsius = (5.0 / 9.0) * (fahrenheit - 32)
```

For example, if `fahrenheit` is `98.6`, then:

```
celsius = 5.0 / 9.0 * (98.6 - 32) = 37.00
```

Similarly, if the user enters the temperature in Celsius, then the formula for calculating the equivalent temperature in Fahrenheit is:

```
fahrenheit = 9.0 / 5.0 * celsius + 32
```

For example, if `celsius` is `20`, then:

```
fahrenheit = 9.0 / 5.0 * 20 + 32 = 68.0
```

The GUI in Figure 6-9 contains a window, a container, two labels, and two text fields. The labels and text fields are placed in the container of the window. As we did

in the rectangle program earlier in this chapter, we can create the window by making the application extend the `class` `JFrame`. To get access to the container, we will use a reference variable of the `Container` type. To create labels, we use objects of type `JLabel`; to create text fields, we use objects of type `JTextField`. Suppose that we have the following declarations:

```
JLabel celsiusLabel;         //label Celsius
JLabel fahrenheitLabel;      //label Fahrenheit

JTextField celsiusTF;        //text field Celsius
JTextField fahrenheitTF;     //text field Fahrenheit
```

When the user enters the temperature in the text field `celsiusTF` and presses the `Enter` key, we want the program to show the equivalent temperature in the text field `fahrenheitTF` and vice versa.

Recall that when you click a `JButton`, it generates an action event. Moreover, the action event is handled by the method `actionPerformed` of the `interface` `ActionListener`. Similarly, when you press the `Enter` key in a text field, it generates an action event. Therefore, we can register an action event listener with the text fields `celsiusTF` and `fahrenheitTF` to take the appropriate action.

Based on this analysis and the GUI shown in Figure 6-9, you can design an event-driven algorithm as follows:

1. Have a listener in each text field.
2. Register an event handler with each text field.
3. Let each event handler registered with a text field do the following:
   a. Get the data from the text field once the user presses `Enter`.
   b. Apply the corresponding formula to perform the conversion.
   c. Set the value of the other text field.

This process of adding an event listener and then registering the event listener to a text field is similar to the process we used to register an event listener to a `JButton` earlier in the chapter. (This process will be described later in this programming example.)

VARIABLES, OBJECTS, AND NAMED CONSTANTS

The input to the program is either the temperature in Celsius or the temperature in Fahrenheit. If the input is a value for Celsius, then the program calculates the equivalent temperature in Fahrenheit. Similarly, if the input is a value for Fahrenheit, then the program calculates the equivalent temperature in Celsius. Therefore, the program needs the following variables:

```
double celsius;     //variable to hold Celsius
double fahrenheit;  //variable to hold Fahrenheit
```

Notice that these variables are needed in each event handler.

The formulas to convert the temperature from Fahrenheit to Celsius and vice versa use the special values `32`, `9.0/5.0`, and `5.0/9.0`, which we will declare as named constants as follows:

```
private static final double FTOC = 5.0 / 9.0;
private static final double CTOF = 9.0 / 5.0;
private static final int OFFSET = 32;
```

As in the GUI, you need two labels—one to label the text field corresponding to the Celsius value and another to label the text field corresponding to the Fahrenheit value. Therefore, the following statements are needed:

```
private JLabel celsiusLabel;         //label Celsius
private JLabel fahrenheitLabel;      //label Fahrenheit

celsiusLabel = new JLabel("Temp in Celsius: ",
                    SwingConstants.RIGHT);//object instantiation

fahrenheitLabel = new JLabel("Temp in Fahrenheit: ",
                    SwingConstants.RIGHT);//object instantiation
```

You also need two `JTextField` objects. The necessary Java code is:

```
private JTextField celsiusTF;         //text field Celsius
private JTextField fahrenheitTF;      //text field Fahrenheit

celsiusTF = new JTextField(7);        //object instantiation
fahrenheitTF = new JTextField(7);     //object instantiation
```

Now you need a window to display the labels and the text fields. Because a window is an object of type `JFrame`, the class containing the application program that we create will extend the definition of the `class` `JFrame`. We will set the width of the window to 500 pixels and the height to 50 pixels. We'll call the class containing the application program `TempConversion`. The application will look like this:

```
//Java program to convert the temperature from Celsius to
//Fahrenheit and vice versa.

import javax.swing.*;

public class TempConversion extends JFrame
{
    private static final int WIDTH = 500;
    private static final int HEIGHT = 50;

    private static final double FTOC = 5.0 / 9.0;
    private static final double CTOF = 9.0 / 5.0;
    private static final int OFFSET = 32;
```

```
    public TempConversion()
    {
        setTitle("Temperature Conversion");
        setSize(WIDTH, HEIGHT);
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public static void main(String[] args)
    {
        TempConversion tempConv = new TempConversion();
    }
}
```

Now you need to access the container content pane to place the GUI components and set the required layout of the pane. Therefore, as before, you need the following statements:

```
Container c = getContentPane();     //get the container

c.setLayout(new GridLayout(1, 4)); //create a new layout

c.add(celsiusLabel);               //add the label celsiusLabel
                                   //to the container
c.add(celsiusTF);                  //add the text field celsiusTF
                                   //to the container
c.add(fahrenheitLabel);            //add the label fahrenheitLabel
                                   //to the container
c.add(fahrenheitTF);               //add the text field fahrenheitTF
                                   //to the container
```

You want your program to respond to the events generated by `JTextFields`. Just as when you click a `JButton` an action event is generated, when you press `Enter` in a `JTextField`, it generates an action event. Therefore, to register event listeners with `JTextFields`, we use the four steps outlined in the section Handling an Event earlier in this chapter: (1) create a class that implements the `interface ActionListener`; (2) provide the definition of the method `actionPerformed` within the class that you created in Step 1; (3) create and instantiate an object of the class created in Step 1; and (4) register the event handler created in Step 3 with the object that generates an action event using the method `addActionListener`.

Next, we create and register an action listener with the `JTextField celsiusTF`.

First we create the `class CelsHandler`, implementing the `interface ActionListener`. Then, we provide the definition of the method `actionPerformed` of the `class CelsHandler`. When the user enters the temperature in the `JTextField celsiusTF` and presses `Enter`, the program needs to calculate and display the equivalent temperature in the `JTextField`

fahrenheitTF. The necessary code is placed within the body of the method actionPerformed.

We now describe the steps of the method actionPerformed. The temperature in Celsius is contained in the JTextField celsiusTF. We use the method getText of the class JTextField to retrieve the temperature in celsiusTF. However, the value returned by the method getText is in string form, so we use the method parseDouble of the class Double to convert the numeric string into a decimal value. It follows that we need a variable of type double, say, celsius, to store the temperature in Celsius. We accomplish this with the following statement:

```
celsius = Double.parseDouble(celsiusTF.getText());
```

We also need a variable of type double, say, fahrenheit, to store the equivalent temperature in Fahrenheit. Because we want to display the temperature to two decimal places, we use the method format of the class String.

We can now write the definition of the class CelsHandler as follows:

```
private class CelsHandler implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        double celsius, fahrenheit;

        celsius =
              Double.parseDouble(celsiusTF.getText());

        fahrenheit = celsius * CTOF + OFFSET;

        fahrenheitTF.setText(String.format("%.2f",
                                           fahrenheit));
    }
}
```

We can now create an object of type CelsHandler as follows:

```
private CelsHandler celsiusHandler;
celsiusHandler = new CelsHandler();
```

Having created a listener, you must associate this handler with the corresponding JTextField celsiusTF. The following code does this:

```
celsiusTF.addActionListener(celsiusHandler);
```

Similarly, we can create and register an action listener with the text field `fahrenheitTF`. The necessary code is:

```
private class FahrHandler implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        double celsius, fahrenheit;

        fahrenheit =
            Double.parseDouble(fahrenheitTF.getText());

        celsius = (fahrenheit - OFFSET) * FTOC;

        celsiusTF.setText(String.format("%.2f",
                                        celsius));
    }
}

private FahrHandler fahrenheitHandler;

fahrenheitHandler = new FahrHandler(); //instantiate the object

fahrenheitTF.addActionListener(fahrenheitHandler);
                                        //add the action listener
```

Now that we have created the necessary GUI components and the programming code, we can put everything together to create the complete program.

PUTTING IT TOGETHER

You can start with the window creation program and then add all the components, handlers, and classes developed. You also need the necessary `import` statements. In this case, they are:

```
import java.awt.*;          //for the class Container
import java.awt.event.*;    //for events
import javax.swing.*;       //for JLabel and JTextField
```

Thus, you have the following Java program:

```
//*************************************************************
//Author: D.S. Malik
//
//Java program to convert the temperature between Celsius and
//Fahrenheit.
//*************************************************************

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

```
public class TempConversion extends JFrame
{
    private JLabel celsiusLabel;
    private JLabel fahrenheitLabel;

    private JTextField celsiusTF;
    private JTextField fahrenheitTF;

    private CelsHandler celsiusHandler;
    private FahrHandler fahrenheitHandler;

    private static final int WIDTH = 500;
    private static final int HEIGHT = 50;
    private static final double FTOC = 5.0 / 9.0;
    private static final double CTOF = 9.0 / 5.0;
    private static final int OFFSET = 32;

    public TempConversion()
    {
        setTitle("Temperature Conversion");
        Container c = getContentPane();
        c.setLayout(new GridLayout(1, 4));

        celsiusLabel = new JLabel("Temp in Celsius: ",
                                  SwingConstants.RIGHT);
        fahrenheitLabel = new JLabel("Temp in Fahrenheit: ",
                                     SwingConstants.RIGHT);

        celsiusTF = new JTextField(7);
        fahrenheitTF = new JTextField(7);

        c.add(celsiusLabel);
        c.add(celsiusTF);
        c.add(fahrenheitLabel);
        c.add(fahrenheitTF);

        celsiusHandler = new CelsHandler();
        fahrenheitHandler = new FahrHandler();

        celsiusTF.addActionListener(celsiusHandler);
        fahrenheitTF.addActionListener(fahrenheitHandler);

        setSize(WIDTH, HEIGHT);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }
```

```
    private class CelsHandler implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            double celsius, fahrenheit;

            celsius =
                  Double.parseDouble(celsiusTF.getText());

            fahrenheit = celsius * CTOF + OFFSET;

            fahrenheitTF.setText(String.format("%.2f",
                                               fahrenheit));
        }
    }

    private class FahrHandler implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            double celsius, fahrenheit;

            fahrenheit =
                Double.parseDouble(fahrenheitTF.getText());

            celsius = (fahrenheit - OFFSET) * FTOC;

            celsiusTF.setText(String.format("%.2f",
                                            celsius));
        }
    }

    public static void main(String[] args)
    {
        TempConversion tempConv = new TempConversion();
    }
}
```

Sample Run: (Figure 6-10 shows the display after the user typed `98.60` in the text field `Temp in Fahrenheit` and pressed `Enter`.)

FIGURE 6-10 Sample run for `TempConversion`

Object-Oriented Design

Chapter 3 discussed the `class String` in detail. Using the `class String`, you can create various `String` objects. Moreover, using the methods of the `class String`, you can manipulate the string stored in a `String` object. Recall that `String` objects are instances of the `class String`. Similarly, a Java program that uses GUI components also uses various objects. For example, in the first part of this chapter, you used the `JFrame`, `JLabel`, `JTextField`, and `JButton` objects. Labels are instances of the `class JLabel`, buttons are instances of the `class JButton`, and so on. In general, an object is an instance of a particular class.

In this section, we delve a little deeper into the general concept of objects and how they are used in object-oriented design (OOD). OOD is a major field of study in its own right. Most colleges and universities offer courses on this topic. This section by no means presents an in-depth treatise on OOD. Rather, we review its general concepts and give a simplified methodology for using the OOD approach to problem solving.

Since Chapter 2, you have used `String` objects. Moreover, in the first part of this chapter, you used objects belonging to various classes, such as `JFrame`, `JLabel`, `JTextField`, `JButton`, and `String`. In fact, in your daily life you use objects such as a VCR, CD player, and so on without realizing how they might be conceptualized as objects or classes. For example, regarding a VCR, note the following facts:

- To use a VCR, you do not need to know how the VCR is made. You do not need to know the internal parts of a VCR or how they work. These are hidden from you.
- To use a VCR, you do need to know the functions of various buttons and how to use them.
- Once you know how to use a VCR, you can use it either as a stand-alone device or you can combine it with other devices to create an entertainment system.
- You cannot modify the functions of a VCR. The Record button will always function as a Record button.

Any Java objects, such as `String` objects, that you have encountered also have the properties mentioned above. You can use the objects and their methods, but you don't need to know how they work.

The aim of OOD is to build software from components called classes so that if someone wants to use a class, all they need to know is the various methods provided by that class.

Recall that in OOD, an object combines data and operations on that data in a single unit, a feature called **encapsulation**. In OOD, we first identify the object, then identify the relevant data, and then identify the operations needed to manipulate the object.

For example, the relevant data for a `String` object is the actual string and the length of the string, that is, the number of characters in the string. Every `String` object must have

memory space to store the relevant data, that is, the string and its length. Next, we must identify the type of operations performed on a string. Some of the operations on a string might be to replace a particular character of a string, extract part of a string, change a string from uppercase to lowercase, and so on. The `class String` provides the necessary operations to be performed on a string.

As another example of how an object contains both data and operations on that data, consider objects of type `JButton`. Because every button has a label, which is a string, every button must have memory space to store its label. Some of the operations on a button that you have encountered are to set the label of the button and to add a listener object to a button. Other operations that can be performed on a button are to set its size and location. These operations are the methods of a class. Thus, the `class JButton` provides the methods to set a button's size and location.

A Simplified OOD Methodology

Now that you have an overview of objects and the essential components of OOD, you may be eager to learn how to solve a particular problem using OOD methodology. The best way to learn is by practice. A simplified OOD methodology can be expressed as follows:

1. Write down a detailed description of the problem.
2. Identify all the (relevant) nouns and verbs.
3. From the list of nouns, select the objects. Identify the data components of each object.
4. From the list of verbs, select the operations.

In item 3, after identifying the objects or classes, usually you will realize that several objects function in the same way. That is, they have the same data components and same operations. In other words, they will lead to the construction of the same class.

Remember that objects are nothing but instances of a particular class. Therefore, to create objects you have to learn how to create classes. In other words, to create objects you first need to create classes; to know what type of classes to create, you need to know what an object stores and what operations are needed to manipulate an object's data. You can see that objects and classes are closely related. Because an object consists of data and operations on the data in a single unit, in Java we use the mechanism of classes to combine data and its operations in a single unit. In OOD methodology, we therefore identify classes, data members of classes, and operations. In Java, data members are also known as **fields**.

The remainder of this section gives various examples to illustrate how objects, data components of objects, and operations on data are identified. In these examples, nouns (objects) are in bold type, and verbs (operations) are in italics.

EXAMPLE 6-2

Consider the problem presented in Example 6-1. In simple terms, the problem can be stated as follows:

"Write a **program** to *input* the **length** and **width** of a **rectangle** and *calculate* and *print* the **perimeter** and **area** of the **rectangle**."

Step 1: Identify all the (relevant) nouns.

- Length
- Width
- Perimeter
- Area
- Rectangle

Step 2: Identify the class(es).

Considering all five nouns, it is clear that:

- Length is the length of a rectangle.
- Width is the width of a rectangle.
- Perimeter is the perimeter of a rectangle.
- Area is the area of a rectangle.

Notice that four of the five nouns are related to the fifth one, namely, `rectangle`. Therefore, choose `Rectangle` as a class. From the `class Rectangle`, you can instantiate rectangles of various dimensions. The `class Rectangle` can be graphically represented as in Figure 6-11.

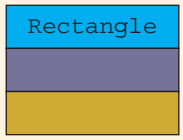

FIGURE 6-11 `class Rectangle`

Step 3: Identify the data members for each of the classes.

In this step, you evaluate the remaining nouns and determine the information that is essential to fully describing each class. Therefore, consider each noun—length, width, perimeter, and area—and ask: "Is each of these nouns essential for describing the rectangle?"

- Perimeter is not needed, because it can be computed from length and width. Perimeter is not a data member.
- Area is not needed, because it can be computed from length and width. Area is not a data member.
- Length is required. Length is a data member.
- Width is required. Width is a data member.

Having made these choices, the `class Rectangle` can be represented with data members, as shown in Figure 6-12.

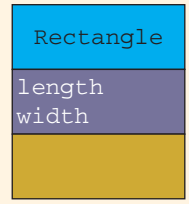

FIGURE 6-12 `class Rectangle` with data members

Step 4: Identify the operations for each of the classes.

Many operations for a class or an object can be determined by looking at the list of verbs. Let us consider the verbs *input*, *calculate*, and *print*. The possible operations on a rectangle object are *input* the length and width, *calculate* the perimeter and area, and *print* the perimeter and area. In this step, we focus on the functionalities of the class(es) involved. By carefully reading the problem statement, you may conclude that you need at least the following operations:

- `setLength`: Set the length of the rectangle.
- `setWidth`: Set the width of the rectangle.
- `computePerimeter`: Calculate the perimeter of the rectangle.
- `computeArea`: Calculate the area of the rectangle.
- `print`: Print the perimeter and area of the rectangle.

It is customary to include operations to retrieve the values of the data members of an object. Therefore, you also need the following operations:

- `getLength`: Retrieve the length of the rectangle.
- `getWidth`: Retrieve the width of the rectangle.

Figure 6-13 shows the `class Rectangle` with data members and operations.

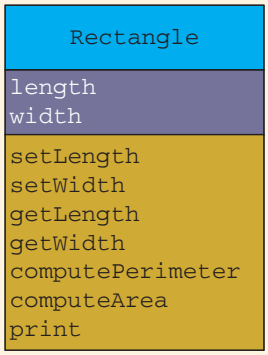

FIGURE 6-13 `class Rectangle` with data members and operations

With these steps completed, you can design an algorithm for each operation of an object (class) and implement each algorithm in Java.

NOTE The diagram in Figure 6-13 is a form of a diagram known as the *class unified modeling language (UML) diagram*. After introducing a few more terms used in a class UML diagram, we formally introduce the class UML diagram in Chapter 8, when we discuss classes in general.

EXAMPLE 6-3

Consider the following problem:

A **place** to buy **candy** is from a **candy machine**. A new candy machine is purchased for the **cafeteria**, but it is not working properly. The candy machine has four **dispensers** to hold and release **items** sold by the candy machine as well as a **cash register**. The machine sells four products—**candies**, **chips**, **gum**, and **cookies**—each stored in a separate dispenser. You have been asked to write a program for this candy machine so that it can be put into operation.

The program should do the following:

- *Show* the **customer** the different **products** sold by the **candy machine**.
- Let the **customer** *make* the selection.
- *Show* the **customer** the **cost of the item** selected.
- *Accept* the **money** from the **customer**.
- *Return* the **change**.
- *Release* the **item**, that is, *make* the sale.

The OOD solution to this problem proceeds as follows:

Step 1: Identify all the nouns.

Place, **candy**, **candy machine**, **cafeteria**, **dispenser**, **items**, **cash register**, **chips**, **gum**, **cookies**, **customer**, **products**, **cost** (of the item), **money**, and **change**.

In this description of the problem, products stand for items such as candy, chips, gum, and cookies. In fact, the actual product in the machine is not that important. What is important is to note that there are four dispensers, each capable of dispensing one product. Further, there is one cash register. Thus, the candy machine consists of four dispensers and one cash register. Graphically, this can be represented as in Figure 6-14.

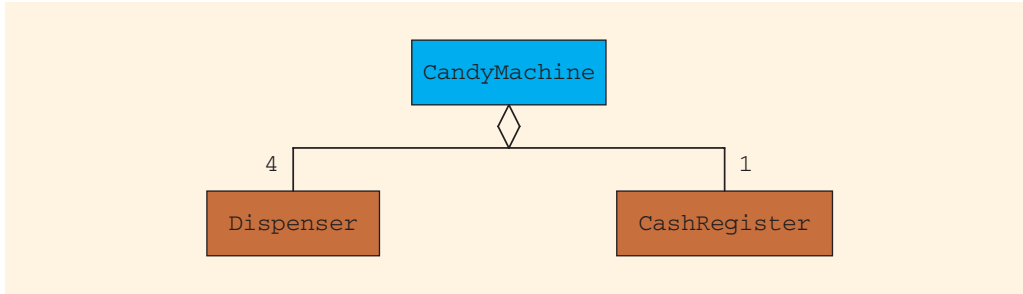

FIGURE 6-14 Candy Machine and its components

In Figure 6-14, the number 4 on top of the box `Dispenser` indicates that there are four dispensers in the candy machine. Similarly, the number 1 on top of the box `CashRegister` indicates that the candy machine has one cash register.

Step 2: Identify the class(es).

You can see that the program you are about to write is supposed to deal with dispensers and cash registers. That is, the main objects are four dispensers and a cash register. Because all the dispensers are of the same type, you need to create a class, say, `Dispenser`, to create the dispensers. Similarly, you need to create a class, say, `CashRegister`, to create a cash register. We will create the `class CandyMachine` containing the four dispensers, a cash register, and the application program.

Step 3: Identify the data members for each of the class(es).

Dispenser To make the sale, at least one item must be in the dispenser and the customer must know the cost of the product. Therefore, the data members of a dispenser are:

- Product cost
- Number of items in the dispenser

Cash Register The cash register accepts money and returns change. Therefore, the cash register has only one data member, which we call `cashOnHand`.

Candy Machine The `class CandyMachine` has four dispensers and a cash register. You can name the four dispensers by the items they store. Therefore, the candy machine has five data members—four dispensers and a cash register.

Step 4: Identify the operations for each of the objects (classes).

The relevant verbs are *show* (selection), *make* (selection), *show* (cost), *accept* (money), *return* (change), and *make* (sale).

The verbs *show* (selection) and *make* (selection) relate to the candy machine. The verbs *show* (cost) and *make* (sale) relate to the dispenser. Similarly, the verbs *accept* (money) and *return* (change) relate to the cash register.

Dispenser The verb *show* (cost) applies to either printing or retrieving the value of the data member `cost`. The verb *make* (sale) applies to reducing the number of items in the dispenser by 1. Of course, the dispenser has to be nonempty. You must also provide an operation to set the cost and the number of items in the dispenser. Thus, the operations for a dispenser object are:

- `getCount`: Retrieve the number of items in the dispenser.
- `getProductCost`: Retrieve the cost of the item.
- `makeSale`: Reduce the number of items in the dispenser by 1.
- `setCost`: Set the cost of the product.
- `setNumberOfItems`: Set the number of items in the dispenser.

Cash Register The verb *accept* (money) applies to updating the money in the cash register by adding the money deposited by the customer. Similarly, the verb *return* (change) applies to reducing the money in the cash register by returning the overpaid amount (by the customer) to the customer. You also need to (initially) set the money in the cash register and retrieve the money from the cash register. Thus, the possible operations on a cash register are:

- `acceptAmount`: Update the amount in the cash register.
- `returnChange`: Return the change.
- `getCashOnHand`: Retrieve the amount in the cash register.
- `setCashOnHand`: Set the amount in the cash register.

Candy Machine The verbs *show* (selection) and *make* (selection) apply to the candy machine. Thus, the two possible operations are:

- `showSelection`: Show the number of products sold by the candy machine.
- `makeSelection`: Allow the customer to select the product.

The result of the OOD for this problem is shown in Figure 6-15.

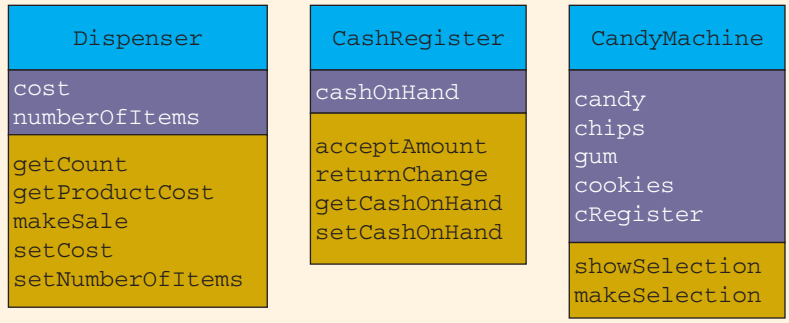

FIGURE 6-15 Classes `Dispenser`, `CashRegister`, `CandyMachine`, and their members

Implementing Classes and Operations

From the preceding examples, it is clear that once the relevant classes, data members of each class, and relevant operations for each class are identified, the next step is to implement these things in Java. Because objects are nothing but instances of classes, we need to learn how to implement classes in Java. Implementing data members, that is, fields, of classes is simple because you need variables to store the data.

What about operations? In Java, we write algorithms to implement operations. Because there is usually more than one operation on an object, each algorithm is implemented with the help of Java's methods. In Chapter 3, we briefly introduced methods and described some predefined methods. However, Java does not provide all the methods that you will ever need. Therefore, to learn how to design and implement classes, you first must learn how to construct and implement your own methods. Because methods are an essential part of Java (or any programming language), Chapter 7 is devoted to teaching you how to create methods.

Primitive Data Types and the Wrapper Classes

Chapter 3 discussed how to use the method `parseInt` of the `class` `Integer` to convert an integer string into an integer. Moreover, you learned that the `class` `Integer` is called a **wrapper** class, or simply a wrapper. It is used to wrap `int` values into `Integer` objects so that `int` values can be regarded as objects. Similarly, the `class` `Long` is used to wrap `long` values into `Long` objects, the `class` `Double` is used

to wrap `double` values into `Double` objects, and the `class` `Float` is used to wrap `float` values into `Float` objects. In fact, Java provides a wrapper class corresponding to each primitive data type. For example, the wrapper class corresponding to the type `int` is `Integer`.

Next, we briefly discuss the `class` `Integer`. Table 6-6 describes some members of the `class` `Integer`.

TABLE 6-6 Some Members of the `class` `Integer`

Named Constants

```
public static final int MAX_VALUE = 2147483647;
```

```
public static final int MIN_VALUE = -2147483648;
```

Constructors

```
public Integer(int num)
  //Creates an object initialized to the value specified
  //by num.
```

```
public Integer(String str)
  //Creates an object initialized to the value specified
  //by the num contained in str.
```

Methods

```
int compareTo(Integer anotherInteger)
  //Compares two Integer objects numerically.
  //Returns the value 0 if the value of this Integer object is
  //equal to the value of anotherInteger object, a value less
  //than 0 if the value of this Integer is less than the value of
  //anotherInteger object, and a value greater than 0 if the value
  //of this Integer object is greater than the value of
  //anotherInteger object.
```

```
public int intValue()
  //Returns the value of the object as an int value.
```

```
public double doubleValue()
  //Returns the value of the object as a double value.
```

```
public boolean equals(Object obj)
  //Returns true if the value of this object is equal
  //to the value of the object specified by obj;
  //otherwise returns false.
```

TABLE 6-6 Some Members of the `class Integer` (continued)

```
public static int parseInt(String str)
  //Returns the value of the number contained in str.
```

```
public String toString()
  //Returns the int value, of the object, as a string.
```

```
public static String toString(int num)
  //Returns the value of num as a string.
```

```
public static Integer valueOf(String str)
  //Returns an Integer object initialized to the value
  //specified by str.
```

Consider the following statements:

```
Integer num;              //Line 1
num = new Integer(86)     //Line 2
```

The statement in Line 1 declares `num` to be a reference variable of type `Integer`. The statement in Line 2 creates an `Integer` object, stores the value `86` in it, and then stores the address of this object into `num`. (See Figure 6-16. Suppose that the address of the `Integer` object is `1350`.)

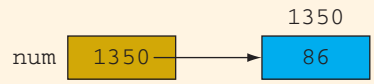

FIGURE 6-16 The reference variable `num` and the object it points to

As you can see, the `int` value `86` is wrapped into an `Integer` object. Just like the `class String`, the `class Integer` does not provide any method to change the value of an existing `Integer` object. That is, `Integer` objects are **immutable**. (In fact, wrapper class objects are immutable.)

As of Java 5.0, Java has simplified the wrapping and unwrapping of primitive type values, called the **autoboxing** and **auto-unboxing** of primitive types. For example, consider the following statements:

```
int x;                    //Line 3
Integer num;              //Line 4
```

The statement in Line 3 declares the int variable x; the statement in Line 4 declares num to be a reference variable of type Integer.

Consider the statement:

```
num = 25;                    //Line 5
```

For the most part, this statement is equivalent to the statement:

```
num = new Integer(25);       //Line 6
```

That is, after the execution of either of these statements, num refers to or points to an Integer object with value 25. The expression in Line 5 is referred to as *autoboxing* of int type.

NOTE

In reality, for the statement in Line 5, if an Integer object with value 25 already exists, then num would point to that object. On the other hand, if the statement in Line 6 executes, then an Integer object with value 25 will be created, even if such an object exists, and num would point to that object. In either case, num would point to an Integer object with value 25.

Now consider the statement:

```
x = num;                     //Line 7
```

This statement is equivalent to the statement:

```
x = num.intValue();          //Line 8
```

After the execution of either the statement in Line 7 or Line 8, the value of x is 25. The statement in Line 7 is referred to as *auto-unboxing* of int type.

NOTE

Autoboxing and -unboxing of primitive types are features of Java 5.0 and are *not* available in Java versions lower than 5.0.

Next, consider the following statement:

```
x = 2 * num;                 //Line 9
```

This statement first unboxes the value of the object num, which is 25, multiplies this value by 2, and then stores the value, which is 50, into x. This illustrates that unboxing also occurs in an expression.

To compare the values of two Integer objects, you can use the method compareTo, described in Table 6-6. If you want to compare the values of two Integer objects only for equality, then you can use the method equals.

NOTE

Suppose you have the following statements:

```
Integer num1 = 24;
Integer num2 = 35;
```

Now consider the following statements:

```
if (num1.equals(num2))
    System.out.println("The values of the "
                       + "objects num1 and num2 "
                       + "are the same.");
else
    System.out.println("The values of the "
                       + "objects num1 and num2 "
                       + "are not the same.");
```

The expression in the `if` statement determines if the value of the object `num1`, which is `24`, is the same as the value of the object `num2`, which is `35`. Next, consider the following statements:

```
if (num1 == num2)
    System.out.println("Both num1 and num2 "
                       + "point to the same "
                       + "object.");
else
    System.out.println("num1 and num2 "
                       + "do not point to the "
                       + "same object.");
```

It follows that when the operator == is used with reference variables of the `Integer` type, it compares whether the objects point to the same object. Therefore, if you want to compare the values of two `Integer` objects, then you should use the method `equals` of the `class` `Integer`. On the other hand, if you want to determine whether two reference variables of `Integer` type points to the same `Integer` object, then you should use the operator ==.

The preceding discussion of comparing Integer objects also applies to other wrapper classes' objects.

Autoboxing and -unboxing of primitive types is a new feature of Java and is available in Java 5.0 and higher versions. It automatically boxes and unboxes primitive type values into appropriate objects. For example, as explained above, `int` values can be automatically boxed and unboxed into `Integer` objects. Example 6-4 further illustrates autoboxing and auto-unboxing of `Integer` objects.

EXAMPLE 6-4

```
//Programm illustrating autoboxing and -unboxing
//of Integer objects.

public class IntegerClassExample
{
    public static void main(String[] args)
    {
        int   x, y;                                    //Line 1

        Integer num1, num2;                            //Line 2

        num1 = 8;      //Autobox 8                     //Line 3
        num2 = 16;     //Autobox 16                    //Line 4

        System.out.println("Line 5: num1 = " + num1
                           + ", num2 = " + num2);      //Line 5

        x = num1 + 4;                                  //Line 6

        System.out.println("Line 7: x = " + x);        //Line 7

        y = num1 + num2;                               //Line 8

        System.out.println("Line 9: y = " + y);        //Line 9

        System.out.println("Line 10: The value of "
                           + "2 * num1 + num2 = "
                           + (2 * num1 + num2));       //Line 10

        System.out.println("Line 11: The value of "
                           + "x * num2 - num1 = "
                           + (x * num2 - num1));       //Line 11

        System.out.println("Line 12: The value of "
                           + "num1 <= num2 is "
                           + (num1 <= num2));          //Line 12

        System.out.println("Line 13: The value of "
                           + "2 * num1 <= x is "
                           + (2 * num1 <= x));         //Line 13

        System.out.println("Line 14: The value of "
                           + "2 * num1 >= num2 is "
                           + (2 * num1 >= num2));      //Line 14
    }
}
```

Sample Run:

```
Line 5: num1 = 8, num2 = 16
Line 7: x = 12
```

```
Line 9: y = 24
Line 10: The value of 2 * num1 + num2 = 32
Line 11: The value of x * num2 - num1 = 184
Line 12: The value of num1 <= num2 is true
Line 13: The value of 2 * num1 <= x is false
Line 14: The value of 2 * num1 >= num2 is true
```

For the most part, the preceding sample run is self-explanatory. Let us look at some of the statements. The statement in Line 3 autoboxes the value `8` into an `Integer` object and stores the address of that object into the reference variable `num1`. The meaning of the statement in Line 4 is similar.

The statement in Line 6 unboxes the value of the object to which `num1` points, adds `4` to that value, and stores the result in `x`. Similarly, the statement in Line 8 unboxes the values of objects pointed to by `num1` and `num2`, adds the values, and stores the result in `y`.

The statement in Line 12 unboxes the values of the objects pointed by `num1` and `num2`, and then compares the values using the relational operator `<=`. (Note that we are not using the operator `==`, so autoboxing occurs here.)

The `class` `Double` also has methods similar to the methods shown in Table 6-6. The Web site, *www.course.com*, and the CD accompanying this book contain a program that illustrates the autoboxing and -unboxing of `double` values into `Double` objects. The program is named `DoubleClassExample.java`. However, to compare the values, for equality, of the wrapper classes objects, you should use the method `equals`. See the following example.

EXAMPLE 6-5

```
//Program illustrating how the operator == and the
//method equals works with Double objects.

public class DoubleClassMethodEquals
{
    public static void main(String[] args)
    {
        Double num1, num2;                                  //Line 1

        num1 = 2567.58;                                     //Line 2
        num2 = 2567.58;                                     //Line 3

        System.out.println("Line 4: num1 = " + num1
                           + ", num2 = " + num2);           //Line 4

        System.out.println("Line 5: The value of "
                           + "num1.equals(num2) is "
                           + num1.equals(num2));            //Line 5
```

```
        System.out.println("Line 6: The value of "
                           + "num1 == num2 is "
                           + (num1 == num2));          //Line 6
    }
}
```

Sample Run:

```
Line 4: num1 = 2567.58, num2 = 2567.58
Line 5: The value of num1.equals(num2) is true
Line 6: The value of num1 == num2 is false
```

In the preceding program, the statements in Lines 2 and 3 create two objects, each with the value `2567.58` and make `num1` and `num2`, respectively, point to these objects. The expression `num1.equals(num2)`, in Line 5, compares the values stored in the objects to which `num1` and `num2` point. Because both objects contain the same value, this expression evaluates to `true`; see the output of the statement in Line 5. On the other hand, the expression `num1 == num2`, in Line 6, determines whether `num1` and `num2` point to the same object.

NOTE

Note that the program in Example 6-5 also illustrates that when you create a `Double` object using the assignment operator without explicitly using the operator `new`, the system always creates a different `Double` object even if one with a given value already exists. For example, see the statements in Lines 2 and 3, and the output of the statement in Line 6.

QUICK REVIEW

1. GUI stands for graphical user interface.
2. Every GUI program requires a window.
3. Various components are added to the content pane of the window and not to the window itself.
4. You must create a layout before you can add a component to the content pane.
5. Pixel stands for picture element. Windows are measured in pixels of height and width.
6. `JFrame` is a class and the GUI component `window` can be created as an instance of `JFrame`.
7. `JLabel` is used to label other GUI components and to display information to the user.
8. A `JTextField` can be used for both input and output.
9. A `JButton` generates an event.
10. An event handler is a Java method that determines the action to be performed as the event happens.
11. When you click a button, an action event is created and sent to another object known as an action listener.

12. An action listener must have a method called `actionPerformed`.
13. A `class` is a collection of data members and methods associated with those data members.
14. OOD starts with a problem statement and tries to identify the classes required by identifying the nouns appearing in the problem statement.
15. Methods of a class are identified with the help of verbs appearing in the problem statement.
16. To wrap values of primitive data types into objects corresponding to each primitive type, Java provides a `class`, called a wrapper class. For example, to wrap an `int` value into an object, the corresponding wrapper `class` is `Integer`. Similarly, to wrap a `double` value into an object, the corresponding wrapper `class` is `Double`.
17. Java 5.0 simplifies the wrapping and unwrapping of primitive type values, called the autoboxing and auto-unboxing of primitive data types.
18. `Integer` objects are immutable. (In fact, wrapper classes' objects are immutable.)
19. To compare the values of two `Integer` objects, you can use the method `compareTo`. If you want to compare the values of two `Integer` objects only for equality, then you can use the method `equals`.

EXERCISES

1. Mark the following statements as true or false.
   a. Every window has a width and height.
   b. In Java, `JFrame` is a class.
   c. To display the window, you need not invoke a method such as `setVisible`.
   d. In Java, the reserved word `extends` allows you to create a new class from an existing one.
   e. The window you see displayed on your screen is a class.
   f. Labels are used to display the output of a program.
   g. Every GUI component you need has to be created and added to a container.
   h. In Java, `implements` is a keyword.
   i. Clicking a button is an example of an action event.
   j. In a problem statement, every verb is a possible class.
   k. In a problem statement, every noun is a possible method.
   l. To use an object, you must know how it is implemented.

2. Name some commonly used GUI components and their uses.
3. Name a GUI component that can be used for both input and output.
4. Name two input GUI components.
5. Why do you need labels in a GUI program?
6. Why would you prefer a GUI program over a non-GUI version?
7. What are the advantages of problem analysis, GUI design, and algorithm design over directly writing a program?
8. Modify the temperature conversion program to convert centimeters to inches, and vice versa.
9. Modify the program to compute the area and perimeter of a rectangle so that your new program will compute the sum and product of two numbers.
10. Fill in the blanks in each of the following:
    a. A(n) _____ places GUI components in a container.
    b. Clicking a button is a(n)_____.
    c. The method _____ is invoked when a button is pressed and a(n) _____ is registered to handle the event.
    d. _____operator is needed to instantiate an object.
    e. A class has two types of members: _____ and _____.
    f. To create a window, you extend the _____class.
    g. Every GUI program is a(n)_____ program.
    h. The method _____ gets the string in the `JTextField` and the method _____ changes the string displayed in a `JTextField`.
    i. If `Student` is a class and you create a new `class GradStudent` by extending `Student`, then `Student` is a(n) _____ and `GradStudent` is a(n) _____.
    j. Event and event listener classes are contained in the package _____.
    k. The unit of measure of length in a window is _____.
11. Write necessary statements to create the following GUI components:
    a. A `JLabel` with the text string `"Enter the number of courses"`
    b. A `JButton` with the text string `"Run"`
    c. A `JTextField` that can display 15 characters
    d. A window with the title `"Welcome Home!"`
    e. A window with a width of 200 pixels and a height of 400 pixels
    f. A `JTextField` that displays the string `"Apple tree"`

12. Correct the syntax errors in the following program and add any additional statements necessary to make the program work:

```
import javax.jswing.*;

public class ROne extends JFrame
{
    static private final int WIDTH = 400;
    static private final int HEIGHT = 300;

    public RectangleProgramOne()
    {
        setTitle("Welcome");
        setSize(WIDTH,HEIGHT);
        SetVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public static void main(String args[])
    {
        ROne r1 = r1();
    }
}
```

13. Correct the syntax errors in the following program:

```
public class RTwo extends JFrame
{
    public RTwoProgram()
    {
        private JLabel length, width, area;

        setTitle("Good day Area");

        length =  JLabel("Enter the length);
        width = JLabel("Enter the width);
        area = JLabel("Area: ");
        containerPane  = ContentPane();
        pane.setLayout(GridLayout(4,1));
        setSize(WIDTH,HEIGHT);
        setVisible();
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public static void main(String args[])
    {
        RTwoProgram R2 = new RTwoProgram();
    }
}
```

14. Consider a common VCR. What are the methods of a VCR?

15. What are the methods of an ATM?

16. Do an OOD analysis of the following problem: Write a program to input the dimensions of a cylinder, and calculate and print the surface area and volume.

17. Lead County Credit Union (LCCU) has recently upgraded its software systems to an OOD design. List at least five classes that you think should be included in this design. For each class, identify some of the data members and methods.
18. Your local public library wants to design new software to keep track of patrons, books, and lending activity. List at least three classes you think should be in the design. For each class, identify some data members and methods.
19. The Custom Consulting Company (CCC) places temporary computer professionals in companies that request such employees. CCC's business can be explained as follows:

    CCC keeps a list of professionals willing to work or currently working on a temporary assignment. A professional may have up to three qualifications, including programmer, senior programmer, analyst, tester, designer, and so on. A company always requests a professional with a single specific qualification. CCC keeps a list of all its clients (that is, a list of other companies) and their current needs. If CCC can find a match, a professional with the required qualification is assigned to a specific opening at one of CCC's clients.

    Identify at least five classes and, for each class, list possible data members and methods.

PROGRAMMING EXERCISES

1. Design a GUI program to find the weighted average of four test scores. The four test scores and their respective weights are given in the following format:

   ```
   testscore1 weight1
   ...
   ```

   For example, the sample data is as follows:

   ```
   75 0.20
   95 0.35
   85 0.15
   65 0.30
   ```

   The user is supposed to enter the data and press a Calculate button. The program must display the weighted average.
2. Write a GUI program that converts seconds to years, weeks, days, hours, and minutes. For this problem, assume 1 year is 365 days.
3. Design and implement a GUI program to compare two strings and display the larger one.

4. Write a GUI program to convert a character to a corresponding integer, and vice versa.
5. Write a GUI program to convert all letters in a string to uppercase letters. For example, `Alb34ert` will be converted to `ALB34ERT`.
6. Write a GUI program to convert all lowercase letters in a string to uppercase letters, and vice versa. For example, `Alb34eRt` will be converted to `aLB34ErT`.
7. Write a GUI program to compute the amount of a certificate of deposit on maturity. The sample data follows:

   ```
   Amount deposited: 80000.00
   Years: 15
   Interest rate: 7.75
   ```

   Hint: To solve this problem, compute $80000.00\ (1 + 7.75 / 100)^{15}$.
8. Write a GUI program that will accept three (integer) input values, say, `x`, `y`, and `z`, and then verify whether or not `x * x + y * y = z * z`.
9. Design and implement a GUI program to convert a positive number given in one base to another base. For this problem, assume that both bases are less than or equal to 10. Consider the sample data:

   ```
   number = 2010, base = 3, and new base = 4.
   ```

   In this case, first convert `2010` in base `3` into the equivalent number in base `10` as follows:

   $$2 * 3^3 + 0 * 3^2 + 1 * 3 + 0 = 54 + 0 + 3 + 0 = 57$$

   To convert `57` to base `4`, you need to find the remainders obtained by dividing by `4`, as shown in the following:

   ```
   57 % 4 = 1, quotient = 14
   14 % 4 = 2, quotient = 3
   3 % 4 = 3, quotient = 0.
   ```

   Therefore, `57` in base `4` is `321`.