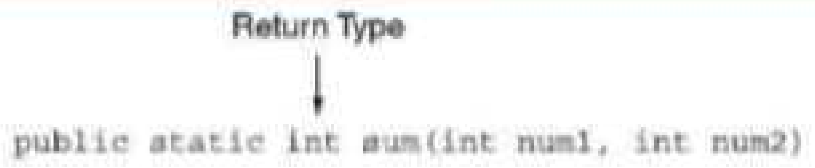
method header. Recall that a `void` method, which does not return a value, uses the key word `void` as its return type in the method header. A value-returning method will use `int`, `double`, `boolean`, or any other valid data type in its header. Here is an example of a method that returns an `int` value:

```
public static int sum(int num1, int num2)
{
   int result;

   result = num1 + num2;
   return result;
}
```

The name of this method is `sum`. Notice in the method header that the return type is `int`, as shown in Figure 5-14.

Figure 5-14 Return type in the method header



```
public static int sum(int num1, int num2)
```

This code defines a method named `sum` that accepts two `int` arguments. The arguments are passed into the parameter variables `num1` and `num2`. Inside the method, a local variable, `result`, is declared. The parameter variables `num1` and `num2` are added, and their sum is assigned to the `result` variable. The last statement in the method is as follows:

```
return result;
```

This is a `return` statement. You must have a `return` statement in a value-returning method. It causes the method to end execution and it returns a value to the statement that called the method. In a value-returning method, the general format of the `return` statement is as follows:

```
return Expression;
```

`Expression` is the value to be returned. It can be any expression that has a value, such as a variable, literal, or mathematical expression. In this case, the `sum` method returns the value in the `result` variable. However, we could have eliminated the `result` variable and returned the expression `num1 + num2`, as shown in the following code:

```
public static int sum(int num1, int num2)
{
   return num1 + num2;
}
```

NOTE: The `return` statement's expression must be of the same data type as the return type specified in the method header, or compatible with it. Otherwise, a compiler error will occur. Java will automatically widen the value of the `return` expression, if necessary, but it will not automatically narrow it.

Calling a Value-Returning Method

The program in Code Listing 5-9 shows an example of how to call the sum method. Notice that the documentation comments for the sum method have a new tag, @return. This tag will be explained later.

Code Listing 5-9 (ValueReturn.java)

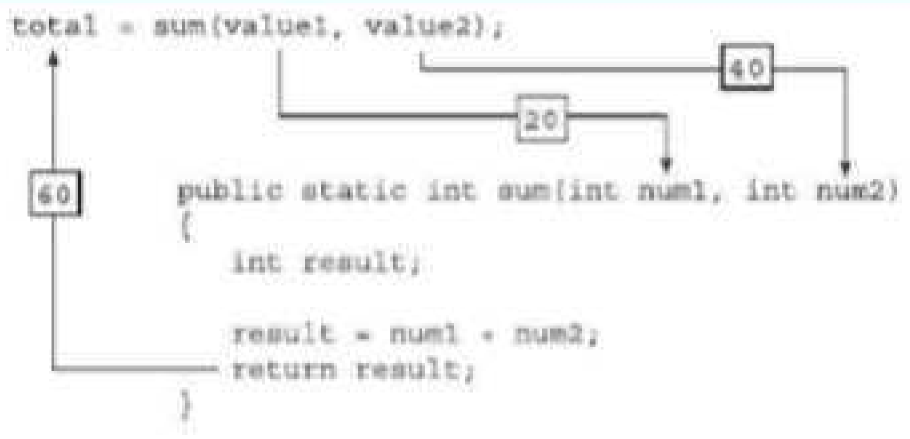
```java
/**
   This program demonstrates a value-returning method.
*/

public class ValueReturn
{
   public static void main(String[] args)
   {
      int total, value1 = 20, value2 = 40;

      // Call the sum method, passing the contents of
      // value1 and value2 as arguments. Assign the
      // return value to the total variable.
      total = sum(value1, value2);

      // Display the contents of all these variables.
      System.out.println("The sum of " + value1 +
                         " and " + value2 + " is " +
                         total);
   }

   /**
      The sum method returns the sum of its two parameters.
      @param num1 The first number to be added.
      @param num2 The second number to be added.
      @return The sum of num1 and num2.
   */

   public static int sum(int num1, int num2)
   {
      int result; // result is a local variable

      // Assign the value of num1 + num2 to result.
      result = num1 + num2;

      // Return the value in the result variable.
      return result;
   }
}
```

Program Output

```
The sum of 20 and 40 is 60
```

The statement in line 14 calls the `sum` method, passing `value1` and `value2` as arguments. It assigns the value returned by the `sum` method to the `total` variable. In this case, the method will return 60. Figure 5-15 shows how the arguments are passed into the method and how a value is passed back from the method.

Figure 5-15 Arguments passed to `sum` and a value returned


total = sum(value1, value2);
40
20
60
public static int sum(int num1, int num2)
{
int result;
result = num1 + num2;
return result;
}

When you call a value-returning method, you usually want to do something meaningful with the value it returns. The `ValueReturn.java` program shows a method's return value being assigned to a variable. This is commonly how return values are used, but you can do many other things with them. For example, the following code shows a math expression that uses a call to the `sum` method:

```
int x = 10, y = 15;
double average;
average = sum(x, y) / 2.0;
```

In the last statement, the `sum` method is called with `x` and `y` as its arguments. The method's return value, which is 25, is divided by 2.0. The result, 12.5, is assigned to `average`. Here is another example:

```
int x = 10, y = 15;
System.out.println("The sum is " + sum(x, y));
```

This code sends the `sum` method's return value to `System.out.println`, so it can be displayed on the screen. The message "The sum is 25" will be displayed.

Remember, a value-returning method returns a value of a specific data type. You can use the method's return value anywhere that you can use a regular value of the same data type. This means that anywhere an `int` value can be used, a call to an `int` value-returning method can be used. Likewise, anywhere a `double` value can be used, a call to a `double` value-returning method can be used. The same is true for all other data types.

Using the `@return` Tag in Documentation Comments

When writing the documentation comments for a value-returning method, you can provide a description of the return value by using a `@return` tag. When the `javadoc` utility sees a `@return` tag inside a method's documentation comments, it knows that a description of the method's return value appears next.

The general format of a `@return` tag comment is as follows:

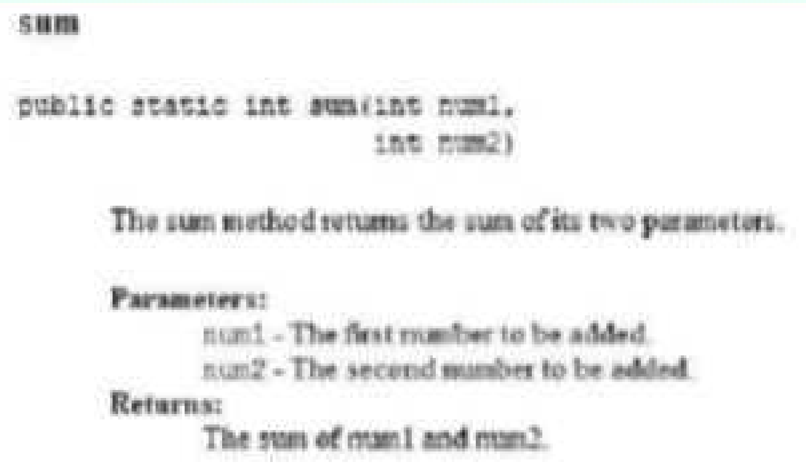
```
@return Description
```

`Description` is a description of the return value. Remember the following points about `@return` tag comments:

- The `@return` tag in a method's documentation comment must appear after the general description of the method.
- The description can span several lines. It ends at the end of the documentation comment (the `*/` symbol), or at the beginning of another tag.

When a method's documentation comments contain a `@return` tag, the `javadoc` utility will create a Returns section in the method's documentation. This is where the description of the method's return value will be listed. Figure 5-16 shows the documentation generated by `javadoc` for the `sum` method in the *ValueReturn.java* file.

Figure 5-16 Documentation for the `sum` method in `ValueReturn.java`



sum

```
public static int sum(int num1,
                      int num2)
```

The sum method returns the sum of its two parameters.

Parameters:
`num1` - The first number to be added.
`num2` - The second number to be added.
Returns:
The sum of num1 and num2.

In the Spotlight:

Using Methods

Your friend Michael runs a catering company. Some of the ingredients that his recipes require are measured in cups. When he goes to the grocery store to buy those ingredients, however, they are sold only by the fluid ounce. He has asked you to write a simple program that converts cups to fluid ounces.

You design the following algorithm:

1. *Get the number of cups from the user.*
2. *Convert the number of cups to fluid ounces.*
3. *Display the result.*

This algorithm lists the top level of tasks that the program needs to perform, and becomes the basis of the class's `main` method. The class will also have the following methods:

- `getCups`—This method will prompt the user to enter the number of cups, and then return that value as a `double`.

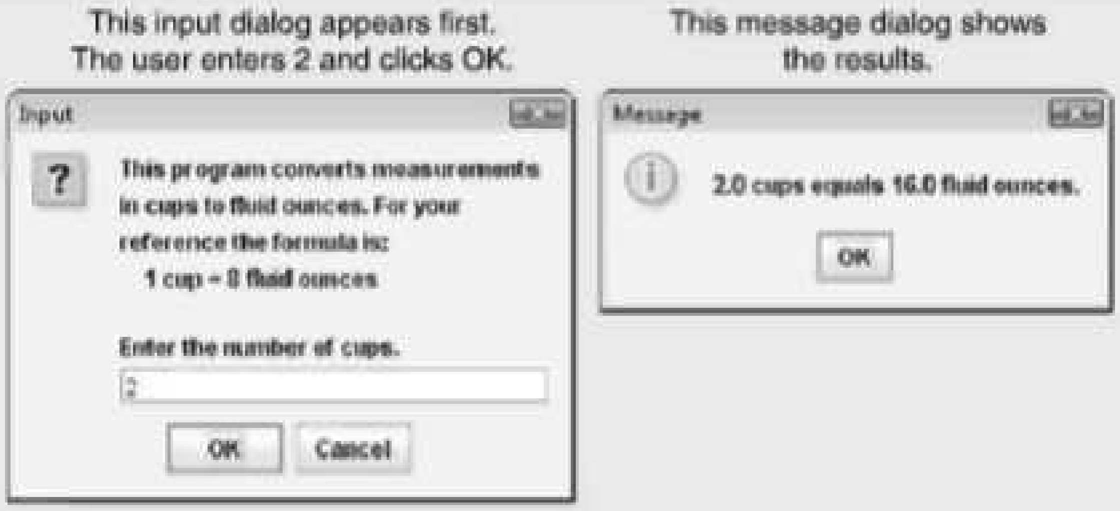
- `cupsToOunces`—This method will accept the number of cups as an argument and then return the equivalent number of fluid ounces as a `double`.
- `displayResults`—This method displays a message indicating the results of the conversion.

Code Listing 5-10 shows the program. Figure 5-17 shows interaction with the program during execution.

Code Listing 5-10 (`CupConverter.java`)

```java
import javax.swing.JOptionPane;

/**
   This program converts cups to fluid ounces.
*/

public class CupConverter
{
   public static void main(String[] args)
   {
      double cups;   // To hold the number of cups
      double ounces; // To hold the number of ounces

      // Get the number of cups.
      cups = getCups();

      // Convert the cups to fluid ounces.
      ounces = cupsToOunces(cups);

      // Display the results.
      displayResults(cups, ounces);
      System.exit(0);
   }

   /**
      The getCups method prompts the user to enter a number
      of cups.
      @return The number of cups entered by the user.
   */

   public static double getCups()
   {
      String input;     // To hold input.
      double numCups;   // To hold cups.

      // Get the number of cups from the user.
      input = JOptionPane.showInputDialog(
            "This program converts measurements\n" +
            "in cups to fluid ounces. For your\n" +
            "reference the formula is:\n" +
            "     1 cup = 8 fluid ounces\n\n" +
            "Enter the number of cups.");

      // Convert the input to a double.
      numCups = Double.parseDouble(input);

      // Return the number of cups.
      return numCups;
   }

   /**
      The cupsToOunces method converts a number of
      cups to fluid ounces, using the formula
      1 cup = 8 fluid ounces.
      @param numCups The number of cups to convert.
      @return The number of ounces.
   */

   public static double cupsToOunces(double numCups)
   {
      return numCups * 8.0;
   }

   /**
      The displayResults method displays a message showing
      the results of the conversion.
      @param cups A number of cups.
      @param ounces A number of ounces.
   */

   public static void displayResults(double cups, double ounces)
   {
      // Display the number of ounces.
      JOptionPane.showMessageDialog(null,
               cups + " cups equals " +
               ounces + " fluid ounces.");
   }
}
```

Figure 5-17 Interaction with the `CupConverter` program


This input dialog appears first.
The user enters 2 and clicks OK.
This message dialog shows
the results.
Input
OK
Cancel
Message
OK

This program converts measurements in cups to fluid ounces. For your reference the formula is:
1 cup = 8 fluid ounces

Enter the number of cups.
2

2.0 cups equals 16.0 fluid ounces.

Returning a `boolean` Value

Frequently there is a need for a method that tests an argument and returns a `true` or `false` value indicating whether or not a condition exists. Such a method would return a `boolean` value. For example, the following method accepts an argument and returns `true` if the argument is within the range of 1 through 100, or `false` otherwise:

```
public static boolean isValid(int number)
{
   boolean status;

   if (number >= 1 && number <= 100)
      status = true;
   else
      status = false;
   return status;
}
```

The following code shows an `if-else` statement that uses a call to the method:

```
int value = 20;
if (isValid(value))
   System.out.println("The value is within range.");
else
   System.out.println("The value is out of range.");
```

When this code executes, the message "The value is within range." will be displayed.

Returning a Reference to an Object

A value-returning method can also return a reference to a non-primitive type, such as a `String` object. The program in Code Listing 5-11 shows such an example.

Code Listing 5-11 `(ReturnString.java)`

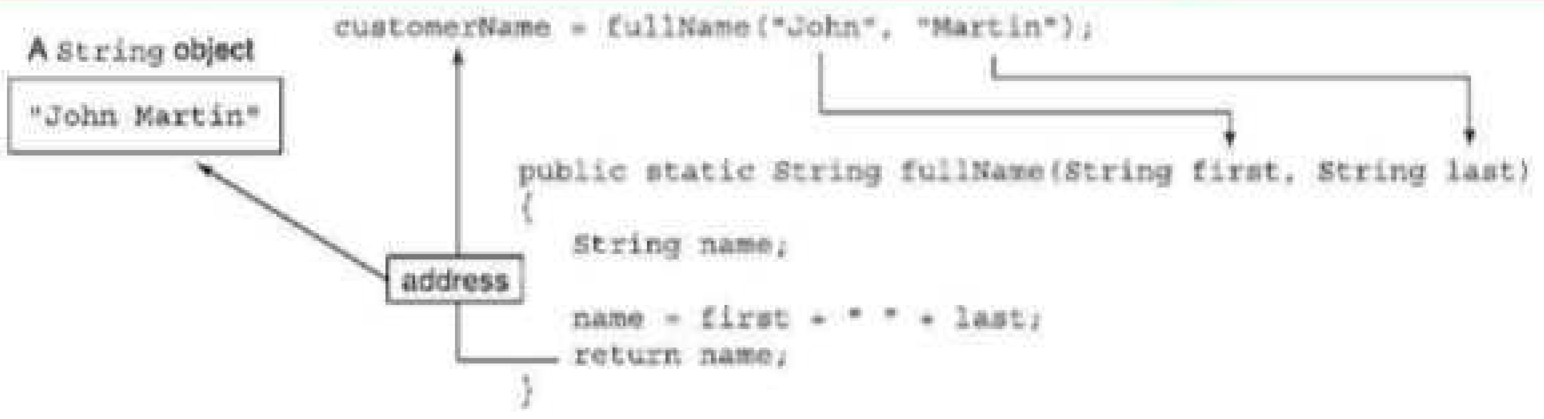
```
 1  /**
 2     This program demonstrates a method that
 3     returns a reference to a String object.
 4  */
 5
 6  public class ReturnString
 7  {
 8     public static void main(String[] args)
 9     {
10        String customerName;
11
12        customerName = fullName("John", "Martin");
13        System.out.println(customerName);
14     }
15
16     /**
17        The fullName method accepts two String arguments
18        containing a first and last name. It concatenates
19        them into a single String object.
20        @param first The first name.
21        @param last The last name.
22        @return A reference to a String object containing
23                the first and last names.
24     */
25
26     public static String fullName(String first, String last)
27     {
28        String name;
29
30        name = first + " " + last;
31        return name;
32     }
33  }
```

Program Output

```
John Martin
```

Line 12 calls the `fullName` method, passing "John" and "Martin" as arguments. The method returns a reference to a `String` object containing "John Martin". The reference is assigned to the `customerName` variable. This is illustrated in Figure 5-18.

Figure 5-18 The `fullName` method returning a reference to a `String` object



```
customerName = fullName("John", "Martin");

public static String fullName(String first, String last)
{
   String name;

   name = first + " " + last;
   return name;
}
```

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

5.11 Look at the following method header. What type of value does the method return?

```
public static double getValue(int a, float b, String c)
```

5.12 Write the header for a method named `days`. The method should return an `int` and have three `int` parameter variables: `years`, `months`, and `weeks`.

5.13 Write the header for a method named `distance`. The method should return a `double` and have two `double` parameter variables: `rate` and `time`.

5.14 Write the header for a method named `lightYears`. The method should return a `long` and have one `long` parameter variable: `miles`.

5.5 Problem Solving with Methods

CONCEPT: A large, complex problem can be solved a piece at a time by methods.

At the beginning of this chapter we introduced the idea of using methods to "divide and conquer" a problem. Often the best way to solve a complex problem is to break it down into smaller problems, and then solve the smaller problems. The process of breaking down a problem into smaller pieces is called *functional decomposition.*

In functional decomposition, instead of writing one long method that contains all of the statements necessary to solve a problem, small methods are written, which each solve a specific part of the problem. These small methods can then be executed in the desired order to solve the problem.

Let's look at an example. The program in Code Listing 5-12 reads 30 days of sales amounts from a file, and then displays the total sales and average daily sales. Here's a brief pseudocode model of the algorithm:

```
Ask the user to enter the name of the file.
Get the total of the sales amounts in the file.
Calculate the average daily sales.
Display the total and average daily sales.
```

The file `MonthlySales.txt`, in this chapter's source code (available at www.pearsonhighered.com/gaddis), is used to test the program. Figure 5-19 shows interaction with the program during execution.

Code Listing 5-12 (SalesReport.java)

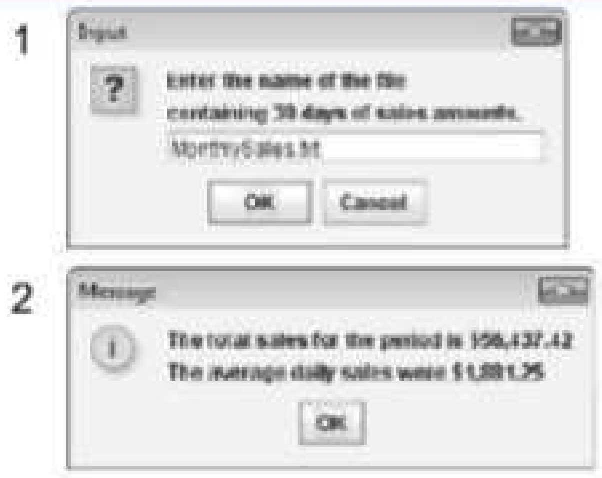
```
import java.util.Scanner;           // For the Scanner class
import java.io.*;                   // For file I/O classes
import java.text.DecimalFormat;     // For the DecimalFormat class
import javax.swing.JOptionPane;     // For the JOptionPane class

/**
   This program opens a file containing the sales
   amounts for 30 days. It calculates and displays
   the total sales and average daily sales.
*/

public class SalesReport
{
   public static void main(String[] args) throws IOException
   {
      final int NUM_DAYS = 30;         // Number of days of sales
      String filename;                 // The name of the file to open
      double totalSales;               // Total sales for period
      double averageSales;             // Average daily sales

      // Get the name of the file.
      filename = getFileName();

      // Get the total sales from the file.
      totalSales = getTotalSales(filename);

      // Calculate the average.
      averageSales = totalSales / NUM_DAYS;

      // Display the total and average.
      displayResults(totalSales, averageSales);

      System.exit(0);
   }

   /**
      The getFileName method prompts the user to enter
      the name of the file to open.
      @return A reference to a String object containing
               the name of the file.
   */

   public static String getFileName()
   {
      String file;                      // To hold the file name

      // Prompt the user to enter a file name.
      file = JOptionPane.showInputDialog("Enter " +
                           "the name of the file\n" +
                           "containing 30 days of " +
                           "sales amounts.");

      // Return the name.
      return file;
   }

   /**
      The getTotalSales method opens a file and
      reads the daily sales amounts, accumulating
      the total. The total is returned.
      @param filename The name of the file to open.
      @return The total of the sales amounts.
   */

   public static double getTotalSales(String filename)
                                      throws IOException
   {
      double total = 0.0;               // Accumulator
      double sales;                     // A daily sales amount

      // Open the file.
      File file = new File(filename);
      Scanner inputFile = new Scanner(file);

      // This loop processes the lines read from the file,
      // until the end of the file is encountered.
      while (inputFile.hasNext())
      {
         // Read a double from the file.
         sales = inputFile.nextDouble();

         // Add sales to the value already in total.
         total += sales;
      }

      // Close the file.
      inputFile.close();

      // Return the total sales.
      return total;
   }

   /**
```

```
      The displayResults method displays the total and
      average daily sales.
      @param total The total sales.
      @param avg The average daily sales.
   */

   public static void displayResults(double total, double avg)
   {
      // Create a DecimalFormat object capable of formatting
      // a dollar amount.
      DecimalFormat dollar = new DecimalFormat("#,###.00");

      // Display the total and average sales.
      JOptionPane.showMessageDialog(null, "The total sales for " +
                          "the period is $" + dollar.format(total) +
                          "\nThe average daily sales were $" +
                          dollar.format(avg));
   }
}
```

Instead of writing the entire program in the `main` method, the algorithm was broken down into the following methods:

- `getFileName`—This method displays an input dialog box asking the user to enter the name of the file containing 30 days of sales amounts. The method returns a reference to a `String` object containing the name entered by the user.
- `getTotalSales`—This method accepts the name of a file as an argument. The file is opened, the sales amounts are read from it, and the total of the sales amounts is accumulated. The method returns the total as a `double`.
- `displayResults`—This method accepts as arguments the total sales and the average daily sales. It displays a message dialog box indicating these values.

Figure 5-19 Interaction with the `SalesReport.java` program



Enter the name of the file
containing 30 days of sales amounts.
MonthlySales.txt

The total sales for the period is $56,437.42
The average daily sales were $1,881.25

Calling Methods That Throw Exceptions

One last thing about the *SalesReport.java* program should be discussed. Notice that the `main` method header (in line 14) and the `getTotalSales` method header (in lines 65 through 66) both have a `throws IOException` clause. The `getTotalSales` method has the clause because it uses a `Scanner` object to open a file. As you know from Chapter 4, any method that uses a `Scanner` object to open a file should have a throws `IOException` clause in its header. Let's quickly review why this is so.

When a `Scanner` object has a problem opening a file, it throws an exception known as `IOException`. Java requires that either (a) the exception is handled in the method that caused it to occur, or (b) the method terminates and throws the exception again. For now you must write your methods to throw the exception again because you will not learn how to handle exceptions until Chapter 11. By writing a `throws IOException` clause in a method's header, you are telling the compiler that the method does not handle the exception. Instead, it throws the exception again.

That explains why the `getTotalSales` method has the throws `IOException` clause, but it doesn't explain why the `main` method has one. After all, `main` doesn't use a `Scanner` object to perform any file operations. The reason `main` has to have the clause is because `main` calls the `getTotalSales` method. If the `Scanner` object in `getTotalSales` throws an `IOException`, the `getTotalSales` method terminates and throws the `IOException` again. That means that `main` must either handle the exception, or terminate and throw it once again. When the `main` method throws the exception, the JVM displays an error message on the screen.

TIP: Until you learn how to handle exceptions in Chapter 11, just remember this when writing programs that throw exceptions: If a method calls another method that has a `throws` clause in its header, then the calling method should have the same `throws` clause.

5.6 Common Errors to Avoid

- **Putting a semicolon at the end of a method header.** Method headers are never terminated with a semicolon.
- **Writing modifiers or return types in a method call statement.** Method modifiers and return types are written in method headers, but never in method calls.
- **Forgetting to write the empty parentheses in a call to a method that accepts no arguments.** You must always write the parentheses in a method call statement, even if the method doesn't accept arguments.
- **Forgetting to pass arguments to methods that require them.** If a method has parameter variables, you must provide arguments when calling the method.
- **Passing an argument of a data type that cannot be automatically converted to the data type of the parameter variable.** Java will automatically perform a widening conversion if the argument's data type is ranked lower than the parameter variable's data type. But Java will not automatically convert an argument to a lower-ranking data type.
- **Attempting to access a parameter variable with code outside the method where the variable is declared.** A parameter variable is visible only within the method it is declared in.

- **Not writing the data type of each parameter variable in a method header.** Each parameter variable declaration inside the parentheses of a method header must include the variable's data type.
- **Changing the contents of a method's parameter variable and expecting the argument that was passed into the parameter to change as well.** Method arguments are passed by value, which means that a copy of the argument is passed into a parameter variable. Changes to the parameter variable have no effect on the argument.
- **Using a variable to receive a method's return value when the variable's data type is incompatible with the data type of the return value.** A variable that receives a method's return value must be of a data type that is compatible with the data type of the return value.
- **Not writing a `return` statement in a value-returning method.** If a method's return type is anything other than `void`, it should return a value.
- **Not writing a required `throws` clause in a method that calls another method.** Any method that calls a method with a `throws` clause in its header must either handle the potential exception or have the same `throws` clause. You will learn how to handle exceptions in Chapter 11.

Review Questions and Exercises

Multiple Choice and True/False

1. This type of method does not return a value.
   a. `null`
   b. `void`
   c. empty
   d. anonymous
2. This appears at the beginning of a method definition.
   a. semicolon
   b. parentheses
   c. body
   d. header
3. The body of a method is enclosed in ________.
   a. curly braces `{ }`
   b. square brackets `[ ]`
   c. parentheses `( )`
   d. quotation marks `""`
4. A method header can contain ________.
   a. method modifiers
   b. the method return type
   c. the method name
   d. a list of parameter declarations
   e. all of these
   f. none of these

5. A value that is passed into a method when it is called is known as a(n) ________.
   a. parameter
   b. argument
   c. signal
   d. return value
6. A variable that receives a value that is passed into a method is known as a(n) ________.
   a. parameter
   b. argument
   c. signal
   d. return value
7. This `javadoc` tag is used to document a parameter variable.
   a. `@parameter`
   b. `@param`
   c. `@paramvar`
   d. `@arg`
8. This statement causes a method to end and sends a value back to the statement that called the method.
   a. `end`
   b. `send`
   c. `exit`
   d. `return`
9. This `javadoc` tag is used to document a method's return value.
   a. `@methodreturn`
   b. `@ret`
   c. `@return`
   d. `@returnval`
10. **True or False:** You terminate a method header with a semicolon.
11. **True or False:** When passing an argument to a method, Java will automatically perform a widening conversion (convert the argument to a higher-ranking data type), if necessary.
12. **True or False:** When passing an argument to a method, Java will automatically perform a narrowing conversion (convert the argument to a lower-ranking data type), if necessary.
13. **True or False:** A parameter variable's scope is the entire program that contains the method in which the parameter is declared.
14. **True or False:** When code in a method changes the value of a parameter, it also changes the value of the argument that was passed into the parameter.
15. **True or False:** When an object, such as a `String`, is passed as an argument, it is actually a reference to the object that is passed.

16. **True or False:** The contents of a `String` object cannot be changed.
17. **True or False:** When passing multiple arguments to a method, the order in which the arguments are passed is not important.
18. **True or False:** No two methods in the same program can have a local variable with the same name.
19. **True or False:** It is possible for one method to access a local variable that is declared in another method.
20. **True or False:** You must have a `return` statement in a value-returning method.

Find the Error

1. Find the error in the following method definition:

```
// This method has an error!
public static void sayHello();
{
   System.out.println("Hello");
}
```

2. Look at the following method header:

```
public static void showValue(int x)
```

The following code has a call to the `showValue` method. Find the error.

```
int x = 8;
showValue(int x);    // Error!
```

3. Find the error in the following method definition:

```
// This method has an error!
public static double timesTwo(double num)
{
   double result = num * 2;
}
```

4. Find the error in the following method definition:

```
// This method has an error!
public static int half(double num)
{
   double result = num / 2.0;
   return result;
}
```

Algorithm Workbench

1. Examine the following method header, and then write an example call to the method:

```
public static void doSomething(int x)
```

2. Here is the code for the `displayValue` method, shown earlier in this chapter:

```
public static void displayValue(int num)
{
   System.out.println("The value is " + num);
}
```

For each of the following code segments, indicate whether it will successfully compile or cause an error:

```
a. displayValue(100);
b. displayValue(6.0);
c. short s = 5;
   displayValue(s);
d. long num = 1;
   displayValue(num);
e. displayValue(6.2f);
f. displayValue((int) 7.5);
```

3. Look at the following method header:

```
public static void myMethod(int a, int b, int c)
```

Now look at the following call to `myMethod`:

```
myMethod(3, 2, 1);
```

When this call executes, what value will be stored in `a`? What value will be stored in `b`? What value will be stored in `c`?

4. What will the following program display?

```
public class ChangeParam
{
   public static void main(String[] args)
   {
      int x = 1;
      double y = 3.4;
      System.out.println(x + " " + y);
      changeUs(x, y);
      System.out.println(x + " " + y);
   }

   public static void changeUs(int a, double b)
   {
      a = 0;
      b = 0.0;
      System.out.println(a + " " + b);
   }
}
```

5. A program contains the following method definition:

   ```
   public static int cube(int num)
   {
        return num * num * num;
   }
   ```

   Write a statement that passes the value 4 to this method and assigns its return value to a variable named `result`.

6. A program contains the following method:

   ```
   public static void display(int arg1, double arg2, char arg3)
   {
        System.out.println("The values are " + arg1 + ", " +
                           arg2 + ", and " + arg3);
   }
   ```

   Write a statement that calls this method and passes the following variables as arguments:

   ```
   char initial = 'T';
   int age = 25;
   double income = 50000.00;
   ```

7. Write a method named `timesTen`. The method should accept a `double` argument, and return a `double` value that is ten times the value of the argument.
8. Write a method named `square` that accepts an integer argument and returns the square of that argument.
9. Write a method named `getName` that prompts the user to enter his or her first name, and then returns the user's input.
10. Write a method named `quartersToDollars`. The method should accept an `int` argument that is a number of quarters, and return the equivalent number of dollars as a `double`. For example, if you pass 4 as an argument, the method should return 1.0; and if you pass 7 as an argument, the method should return 1.75.

Short Answer

1. What is the "divide and conquer" approach to problem solving?
2. What is the difference between a void method and a value-returning method?
3. What is the difference between an argument and a parameter variable?
4. Where do you declare a parameter variable?
5. Explain what is meant by the phrase "pass by value."
6. Why do local variables lose their values between calls to the method in which they are declared?

Programming Challenges

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

1. `showChar` Method

Write a method named `showChar`. The method should accept two arguments: a reference to a `String` object and an integer. The integer argument is a character position within the `String`, with the first character being at position 0. When the method executes, it should display the character at that character position. Here is an example of a call to the method:

```
showChar("New York", 2);
```

In this call, the method will display the character `w` because it is in position 2. Demonstrate the method in a complete program.

2. Retail Price Calculator

VideoNote The Retail Price Calculator Problem

Write a program that asks the user to enter an item's wholesale cost and its markup percentage. It should then display the item's retail price. For example:

- If an item's wholesale cost is 5.00 and its markup percentage is 100 percent, then the item's retail price is 10.00.
- If an item's wholesale cost is 5.00 and its markup percentage is 50 percent, then the item's retail price is 7.50.

The program should have a method named `calculateRetail` that receives the wholesale cost and the markup percentage as arguments, and returns the retail price of the item.

3. Rectangle Area—Complete the Program

If you have downloaded the book's source code from www.pearsonhighered.com/gaddis, you will find a partially written program named `AreaRectangle.java` in this chapter's source code folder. Your job is to complete the program. When it is complete, the program will ask the user to enter the width and length of a rectangle, and then display the rectangle's area. The program calls the following methods, which have not been written:

- `getLength`—This method should ask the user to enter the rectangle's length, and then return that value as a `double`.
- `getWidth`—This method should ask the user to enter the rectangle's width, and then return that value as a `double`.
- `getArea`—This method should accept the rectangle's length and width as arguments, and return the rectangle's area. The area is calculated by multiplying the length by the width.
- `displayData`—This method should accept the rectangle's length, width, and area as arguments, and display them in an appropriate message on the screen.

4. Paint Job Estimator

A painting company has determined that for every 115 square feet of wall space, one gallon of paint and eight hours of labor will be required. The company charges $18.00 per hour for labor. Write a program that allows the user to enter the number of rooms to be painted and the price of the paint per gallon. It should also ask for the square feet of wall space in each room. The program should have methods that return the following data:

- The number of gallons of paint required
- The hours of labor required

- The cost of the paint
- The labor charges
- The total cost of the paint job

Then it should display the data on the screen.

5. Falling Distance

When an object is falling because of gravity, the following formula can be used to determine the distance the object falls in a specific time period:

$$d = \frac{1}{2} g t^2$$

The variables in the formula are as follows: d is the distance in meters, g is 9.8, and t is the amount of time, in seconds, that the object has been falling.

Write a method named `fallingDistance` that accepts an object's falling time (in seconds) as an argument. The method should return the distance, in meters, that the object has fallen during that time interval. Demonstrate the method by calling it in a loop that passes the values 1 through 10 as arguments, and displays the return value.

6. Celsius Temperature Table

The formula for converting a temperature from Fahrenheit to Celsius is

$$C = \frac{5}{9}(F - 32)$$

where F is the Fahrenheit temperature and C is the Celsius temperature. Write a method named `celsius` that accepts a Fahrenheit temperature as an argument. The method should return the temperature, converted to Celsius. Demonstrate the method by calling it in a loop that displays a table of the Fahrenheit temperatures 0 through 20 and their Celsius equivalents.

7. Test Average and Grade

Write a program that asks the user to enter five test scores. The program should display a letter grade for each score and the average test score. Write the following methods in the program:

- `calcAverage`—This method should accept five test scores as arguments and return the average of the scores.
- `determineGrade`—This method should accept a test score as an argument and return a letter grade for the score, based on the following grading scale:

Score	Letter Grade
90–100	A
80–89	B
70–79	C
60–69	D
Below 60	F

8. Conversion Program

Write a program that asks the user to enter a distance in meters. The program will then present the following menu of selections:

1. Convert to kilometers
2. Convert to inches
3. Convert to feet
4. Quit the program

The program will convert the distance to kilometers, inches, or feet, depending on the user's selection. Here are the specific requirements:

- Write a `void` method named `showKilometers`, which accepts the number of meters as an argument. The method should display the argument converted to kilometers. Convert the meters to kilometers using the following formula:

  `kilometers = meters * 0.001`
- Write a `void` method named `showInches`, which accepts the number of meters as an argument. The method should display the argument converted to inches. Convert the meters to inches using the following formula:

  `inches = meters * 39.37`
- Write a `void` method named `showFeet`, which accepts the number of meters as an argument. The method should display the argument converted to feet. Convert the meters to feet using the following formula:

  `feet = meters * 3.281`
- Write a `void` method named `menu` that displays the menu of selections. This method should not accept any arguments.
- The program should continue to display the menu until the user enters 4 to quit the program.
- The program should not accept negative numbers for the distance in meters.
- If the user selects an invalid choice from the menu, the program should display an error message.

Here is an example session with the program, using console input. The user's input is shown in bold.

```
Enter a distance in meters: 500 [Enter]
1. Convert to kilometers
2. Convert to inches
3. Convert to feet
4. Quit the program

Enter your choice: 1 [Enter]
500 meters is 0.5 kilometers.

1. Convert to kilometers
2. Convert to inches
3. Convert to feet
4. Quit the program
```

```
Enter your choice: 3 [Enter]
500 meters is 1640.5 feet.

1. Convert to kilometers
2. Convert to inches
3. Convert to feet
4. Quit the program

Enter your choice: 4 [Enter]
Bye!
```

9. Distance Traveled Modification

The distance a vehicle travels can be calculated as follows:

Distance = *Speed* * *Time*

Write a method named `distance` that accepts a vehicle's speed and time as arguments, and returns the distance the vehicle has traveled. Modify the "Distance Traveled" program you wrote in Chapter 4 (Programming Challenge 2) to use the method.

10. Stock Profit

The profit from the sale of a stock can be calculated as follows:

$Profit = ((NS \times SP) - SC) - ((NS \times PP) + PC)$

where NS is the number of shares, PP is the purchase price per share, PC is the purchase commission paid, SP is the sale price per share, and SC is the sale commission paid. If the calculation yields a positive value, then the sale of the stock resulted in a profit. If the calculation yields a negative number, then the sale resulted in a loss.

Write a method that accepts as arguments the number of shares, the purchase price per share, the purchase commission paid, the sale price per share, and the sale commission paid. The method should return the profit (or loss) from the sale of stock. Demonstrate the method in a program that asks the user to enter the necessary data and displays the amount of the profit or loss.

11. Multiple Stock Sales

Use the method that you wrote for Programming Challenge 10 (Stock Profit) in a program that calculates the total profit or loss from the sale of multiple stocks. The program should ask the user for the number of stock sales, and the necessary data for each stock sale. It should accumulate the profit or loss for each stock sale and then display the total.

12. Kinetic Energy

In physics, an object that is in motion is said to have kinetic energy. The following formula can be used to determine a moving object's kinetic energy:

$KE = \frac{1}{2} mv^2$

The variables in the formula are as follows: KE is the kinetic energy, m is the object's mass in kilograms, and v is the object's velocity, in meters per second.

Write a method named `kineticEnergy` that accepts an object's mass (in kilograms) and velocity (in meters per second) as arguments. The method should return the amount of kinetic energy that the object has. Demonstrate the method by calling it in a program that asks the user to enter values for mass and velocity.

13. `isPrime` Method

A prime number is a number that is evenly divisible only by itself and 1. For example, the number 5 is prime because it can be evenly divided only by 1 and 5. The number 6, however, is not prime because it can be divided evenly by 1, 2, 3, and 6.

Write a method named `isPrime`, which takes an integer as an argument and returns `true` if the argument is a prime number, or `false` otherwise. Demonstrate the method in a complete program.

TIP: Recall that the `%` operator divides one number by another, and returns the remainder of the division. In an expression such as `num1 % num2`, the `%` operator will return 0 if `num1` is evenly divisible by `num2`.

14. Prime Number List

Use the `isPrime` method that you wrote in Programming Challenge 13 in a program that stores a list of all the prime numbers from 1 through 100 in a file.

15. Even/Odd Counter

You can use the following logic to determine whether a number is even or odd:

```
if ((number % 2) == 0)
{
   // The number is even.
}
else
{
   // The number is odd.
}
```

Write a program with a method named `isEven` that accepts an `int` argument. The method should return `true` if the argument is even, or `false` otherwise. The program's `main` method should use a loop to generate 100 random integers. It should use the `isEven` method to determine whether each random number is even, or odd. When the loop is finished, the program should display the number of even numbers that were generated, and the number of odd numbers.

16. Present Value

Suppose you want to deposit a certain amount of money into a savings account, and then leave it alone to draw interest for the next 10 years. At the end of 10 years, you would like to have $10,000 in the account. How much do you need to deposit today to make that happen? You can use the following formula, which is known as the present value formula, to find out:

$$P = \frac{F}{(1 + r)^n}$$

The terms in the formula are as follows:

- P is the **present value**, or the amount that you need to deposit today.
- F is the **future value** that you want in the account. (In this case, F is $10,000.)
- r is the **annual interest rate**.
- n is the **number of years** that you plan to let the money sit in the account.

Write a method named `presentValue` that performs this calculation. The method should accept the future value, annual interest rate, and number of years as arguments. It should return the present value, which is the amount that you need to deposit today. Demonstrate the method in a program that lets the user experiment with different values for the formula's terms.

17. Rock, Paper, Scissors Game

Write a program that lets the user play the game of Rock, Paper, Scissors against the computer. The program should work as follows.

1. When the program begins, a random number in the range of 1 through 3 is generated. If the number is 1, then the computer has chosen rock. If the number is 2, then the computer has chosen paper. If the number is 3, then the computer has chosen scissors. (Don't display the computer's choice yet.)
2. The user enters his or her choice of "rock", "paper", or "scissors" at the keyboard. (You can use a menu if you prefer.)
3. The computer's choice is displayed.
4. A winner is selected according to the following rules:
   - If one player chooses rock and the other player chooses scissors, then rock wins. (The rock smashes the scissors.)
   - If one player chooses scissors and the other player chooses paper, then scissors wins. (Scissors cuts paper.)
   - If one player chooses paper and the other player chooses rock, then paper wins. (Paper wraps rock.)
   - If both players make the same choice, the game must be played again to determine the winner.

Be sure to divide the program into methods that perform each major task.

18. ESP Game

Write a program that tests your ESP (extrasensory perception). The program should randomly select the name of a color from the following list of words:

Red, Green, Blue, Orange, Yellow

To select a word, the program can generate a random number. For example, if the number is 0, the selected word is *Red*; if the number is 1, the selected word is *Green*; and so forth.

Next, the program should ask the user to enter the color that the computer has selected. After the user has entered his or her guess, the program should display the name of the randomly selected color. The program should repeat this 10 times and then display the number of times the user correctly guessed the selected color. Be sure to modularize the program into methods that perform each major task.

CHAPTER

6 A First Look at Classes

TOPICS

6.1 Objects and Classes

CONCEPT: An object is a software component that exists in memory, and serves a specific purpose in a program. An object is created from a class that contains code describing the object.

If you have ever driven a car, you know that a car consists of a lot of components. It has a steering wheel, an accelerator pedal, a brake pedal, a gear shifter, a speedometer, and numerous other devices that the driver interacts with. There are also a lot of components under the hood, such as the engine, the battery, the radiator, and so forth. So, a car is not just one single object, but rather a collection of objects that work together.

This same notion applies to computer programming as well. Most programming languages in use today are object-oriented. With an object-oriented language, such as Java, you create programs that are made of objects. In programming, however, an object isn't a physical device, like a steering wheel or a brake pedal; it's a software component that exists in the computer's memory and performs a specific task. In software, an object has two general capabilities:

- An object can store data. The data stored in an object are commonly called *fields*.
- An object can perform operations. The operations that an object can perform are called *methods*.

Objects are very important in Java. Here are some examples of objects that you have previously learned about:

- If you need to read input from the keyboard, or from a file, you can use a `Scanner` object.
- If you need to generate random numbers, you can use a `Random` object.
- If you need to write output to a file, you can use a `PrintWriter` object.

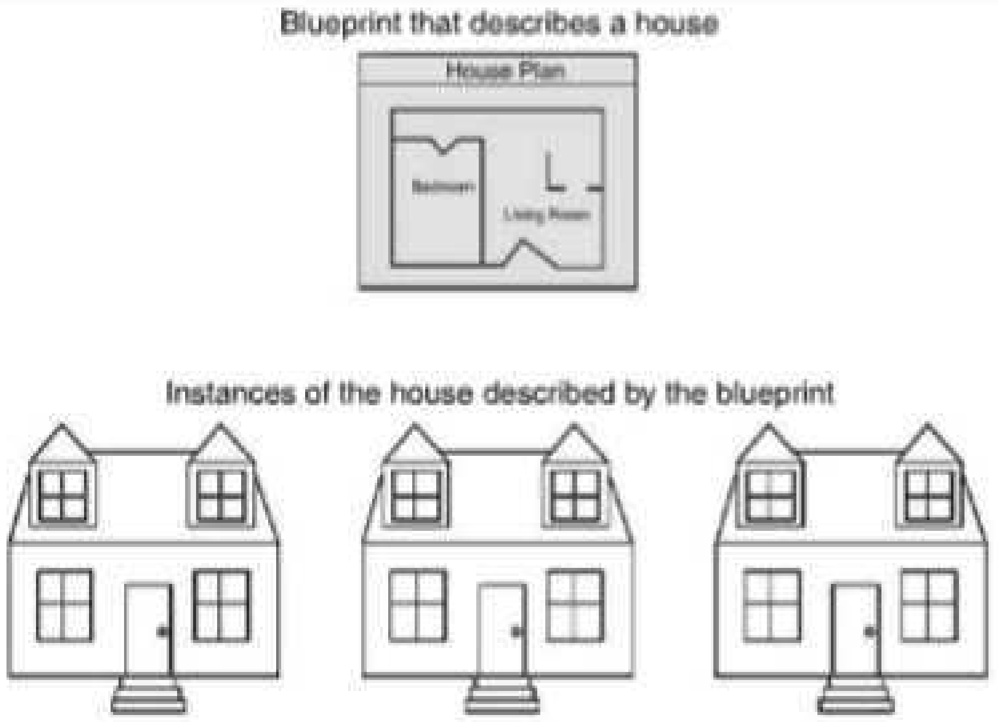
When a program needs the services of a particular type of object, it creates that object in memory, and then calls that object's methods as necessary.

Classes: Where Objects Come From

Objects are very useful, but they don't just magically appear in your program. Before a specific type of object can be used by a program, that object has to be created in memory. And, before an object can be created in memory, you must have a class for the object.

A *class* is code that describes a particular type of object. It specifies the data that an object can hold (the object's fields), and the actions that an object can perform (the object's methods). You can think of a class as a code "blueprint" that can be used to create a particular type of object. It serves a purpose similar to that of the blueprint for a house. The blueprint itself is not a house, but rather a detailed description of a house. When we use the blueprint to build an actual house, we could say we are building an instance of the house described by the blueprint. If we so desire, we can build several identical houses from the same blueprint. Each house is a separate instance of the house described by the blueprint. This idea is illustrated in Figure 6-1.

Figure 6-1 A blueprint and houses built from the blueprint


Blueprint that describes a house
House Plan
Bedroom
Living Room
Instances of the house described by the blueprint

So, a class is not an object, but a description of an object. When a program is running, it can use the class to create, in memory, as many objects of a specific type as needed. Each object that is created from a class is called an *instance* of the class.

NOTE: Up to this chapter, you have used classes for a different purpose: as containers for a program's methods. All of the Java programs that you have written so far have had a class containing a `main` method, and possibly other methods. In this chapter you will learn how to write classes from which objects can be created.

Classes in the Java API

So far, the objects that you have used in your programs are created from classes in the Java API. For example, each time you create a `Scanner` object, you are creating an instance of a class named `Scanner`, which is in the Java API. Likewise, when you create a `Random` object, you are creating an instance of a class named `Random`, which is in the Java API. The same is true for `PrintWriter` objects. When you need to write data to a file, you create an instance of the `PrintWriter` class, which is in the Java API. Look at Code Listing 6-1, a program that uses all of these types of objects.

Code Listing 6-1 `(ObjectDemo.java)`

```
import java.util.Scanner; // Needed for the Scanner class
import java.util.Random;  // Needed for the Random class
import java.io.*;         // Needed for file I/O classes

/*
  This program writes random numbers to a file.
*/

public class ObjectDemo
{
   public static void main(String[] args) throws IOException
   {
      int maxNumbers;    // Max number of random numbers
      int number;        // To hold a random number

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Create a Random object to generate random numbers.
      Random rand = new Random();

      // Create a PrintWriter object to open the file.
      PrintWriter outputFile = new PrintWriter("numbers.txt");

      // Get the number of random numbers to write.
      System.out.print("How many random numbers should I write? ");
      maxNumbers = keyboard.nextInt();

29         // Write the random numbers to the file.
30         for (int count = 0; count < maxNumbers; count++)
31         {
32            // Generate a random integer.
33            number = rand.nextInt();
34
35            // Write the random integer to the file.
36            outputFile.println(number);
37         }
38
39         // Close the file.
40         outputFile.close();
41         System.out.println("Done");
42      }
43   }
```

Program Output with Example Input Shown in Bold

```
How many random numbers should I write? **10 [Enter]**
Done
```

In a nutshell, this program writes a specified number of random numbers to a file named `numbers.txt`. When the program runs, it asks the user for the number of random numbers to write. It then writes that many numbers to the file. To do its job, it creates three objects:

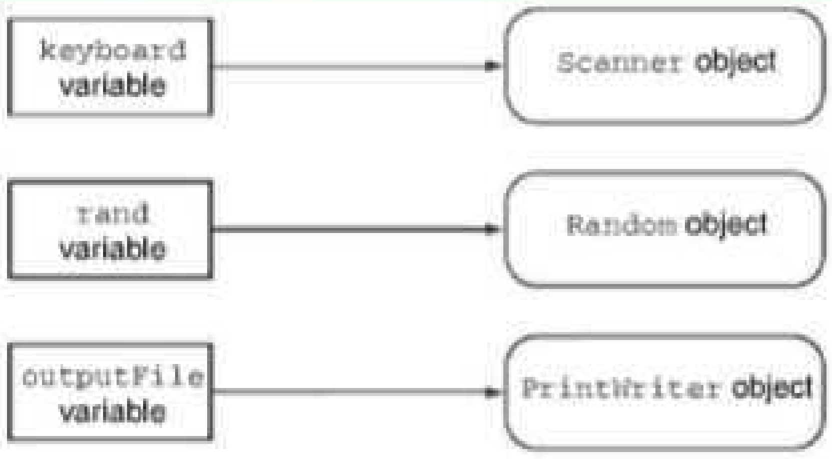
- In line 17 it creates an instance of the `Scanner` class, and assigns the object's address to a variable named `keyboard`. The object will be used to read keyboard input.
- In line 20 it creates an instance of the `Random` class, and assigns the object's address to a variable named `rand`. The object will be used to generate random numbers.
- In line 23 it creates an instance of the `PrintWriter` class, and assigns the object's address to a variable named `outputFile`. The object will be used to write output to the `numbers.txt` file.

Figure 6-2 illustrates the three objects that the program creates. As the program runs, it uses these objects to accomplish certain tasks. For example:

- In line 27 the `Scanner` object's `nextInt` method is called to read the user's input (which is the number of random numbers to generate). The value that is returned from the method is assigned to the `maxNumbers` variable.
- In line 33 the `Random` object's `nextInt` method is called to get a random integer. The value that is returned from the method is assigned to the `number` variable.
- In line 36 the `PrintWriter` object's `println` method is called to write the value of the `number` variable to the file.
- In line 40 the `PrintWriter` object's `close` method is called to close the file.

This simple example demonstrates how most programs work. A program typically creates the various objects that it needs to complete its job. Each object has a set of methods that can be called, causing the object to perform an operation. When the program needs an object to do something, it calls the appropriate method.

Figure 6-2 Objects created by the `ObjectDemo` program



NOTE: The `import` statements that appear in lines 1 through 3 of Code Listing 6-1 make the `Scanner`, `Random`, and `PrintWriter` classes available to the program. You will learn more about how the Java API is organized, and why you need these `import` statements later in this chapter.

Primitive Variables vs. Objects

Chapter 2 introduced you to the Java primitive data types: `byte`, `short`, `int`, `long`, `char`, `float`, `double`, and `boolean`. By now you have seen many programs that use both primitive data types and objects. In fact, the program in Code Listing 6-1 uses two primitive variables (`maxNumbers` and `number`, both `int` variables), as well as a `Scanner` object, a `Random` object, and a `PrintWriter` object.

You've probably noticed that the steps required to create an object differ from the steps required to create a primitive variable. For example, to create an `int` variable, you simply need a declaration such as the following:

```
int wholeNumber;
```

But, to create an object, you have to write some extra code. For example, the following statement creates a `Random` object:

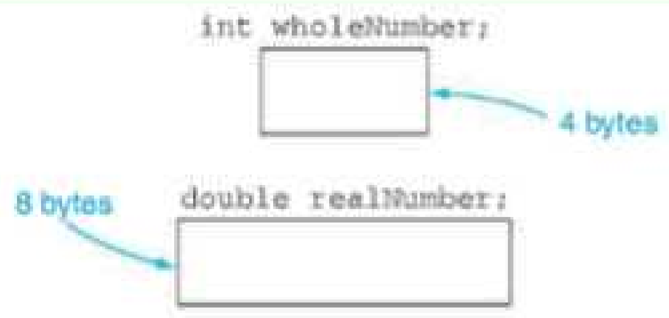
```
Random rand = new Rand();
```

Primitive variables, such as `int`s, `double`s, and so forth, are simply storage locations in the computer's memory. A primitive data type is called "primitive" because a variable created with a primitive data type has no built-in capabilities other than storing a value. When you declare a primitive variable, the compiler sets aside, or allocates, a chunk of memory that is big enough for that variable. For example, look at the following variable declarations:

```
int wholeNumber;
double realNumber;
```

Recall from Chapter 2 that an `int` uses 4 bytes of memory and a `double` uses 8 bytes of memory. These declaration statements will cause memory to be allocated as shown in Figure 6-3.

Figure 6-3 Memory allocated

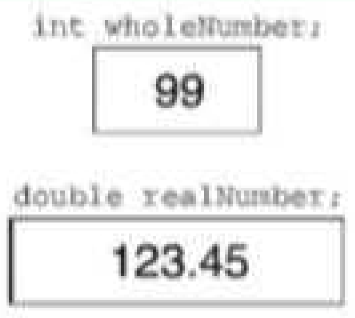


The memory that is allocated for a primitive variable is the actual location that will hold any value that is assigned to that variable. For example, suppose that we use the following statements to assign values to the variables shown in Figure 6-3:

```
wholeNumber = 99;
realNumber = 123.45;
```

Figure 6-4 shows how the assigned values are stored in each variable's memory location.

Figure 6-4 Values assigned to the variables



As you can see from these illustrations, primitive variables are very straightforward. When you are working with a primitive variable, you are using a storage location that holds a piece of data.

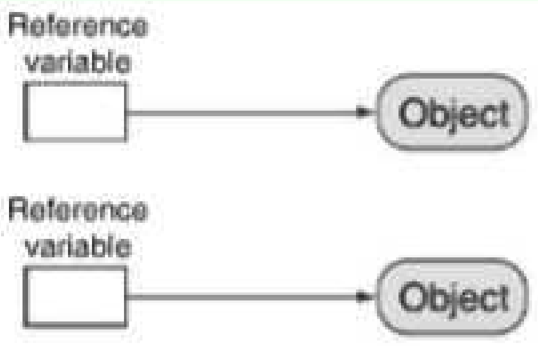
This is different from the way that objects work. When you are working with an object, you are typically using two things:

- The object itself, which must be created in memory
- A reference variable that refers to the object

The object that is created in memory holds data of some sort and performs operations of some sort. (Exactly what the data and operations are depends on what kind of object it is.) In order to work with the object in code, you need some way to refer to the object. That's where the reference variable comes in. The reference variable doesn't hold an actual piece of data that your program will work with. Instead, it holds the object's memory address. We say that the variable references the object. When you want to work with the object, you use the variable that references it.

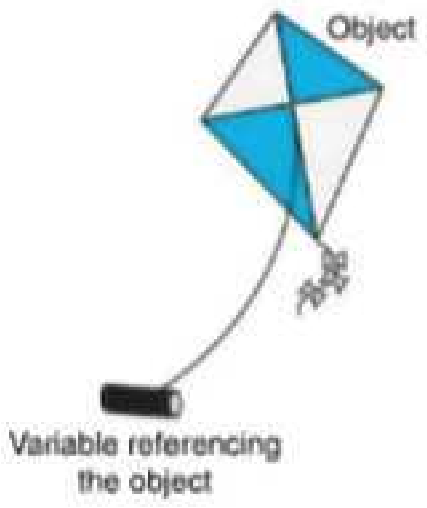
Reference variables, also known as class type variables, can be used only to reference objects. Figure 6-5 illustrates two objects that have been created in memory, each referenced by a variable.

Figure 6-5 Two objects referenced by variables



To understand how reference variables and objects work together, think about flying a kite. In order to fly a kite, you need a spool of string attached to it. When the kite is airborne, you use the spool of string to hold on to the kite and control it. This is similar to the relationship between an object and the variable that references the object. As shown in Figure 6-6, the object is like the kite, and the variable that references the object is like the spool of string.

Figure 6-6 The kite and string metaphor



Creating an object typically requires the following two steps:

1. You declare a reference variable.
2. You create the object in memory, and assign its memory address to the reference variable.

After you have performed these steps, you can use the reference variable to work with the object. Once again, here is the familiar example of how you create an object from the `Random` class:

```
Random rand = new Random();
```

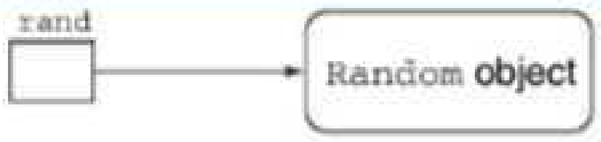
Let's look at the different parts of this statement:

- The first part of the statement, appearing on the left side of the `=` operator, reads `Random rand`. This declares a variable named `rand`, which can be used to reference an object of the `Random` type.
- The second part of the statement, appearing on the right side of the `=` operator, reads `new Random()`. The `new` operator creates an object in memory, and returns that object's memory address. So, the expression `new Random()` creates an object from the `Random` class, and returns that object's memory address.

- The = operator assigns the memory address that was returned from the new operator to the rand variable.

After this statement executes, the rand variable will reference a Random object, as shown in Figure 6-7. The rand variable can then be used to perform operations with the object, such as generating random numbers.

Figure 6-7 The rand variable references a Random object



Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

6.1 What does an object use its fields for?

6.2 What are an object's methods?

6.3 How is a class like a blueprint?

6.4 You have programs that create Scanner, Random, and PrintWriter objects. Where are the Scanner, Random, and PrintWriter classes?

6.5 What does the new operator do?

6.6 What values do reference variables hold?

6.7 How is the relationship between an object and a reference variable similar to a kite and a spool of string?

6.2 Writing a Simple Class, Step by Step

CONCEPT: You can write your own classes to create the objects that you need in a program. We will go through the process of writing a class in a step-by-step fashion.

The Java API provides many prewritten classes, ready for use in your programs. Sometimes, however, you will wish you had an object to perform a specific task, and no such class will exist in the Java API. This is not a problem, because you can write your own classes with the specific fields and methods that you need for any situation.

In this section we will write a class named Rectangle. Each object that is created from the Rectangle class will be able to hold data about a rectangle. Specifically, a Rectangle object will have the following fields:

- length. The length field will hold the rectangle's length.
- width. The width field will hold the rectangle's width.

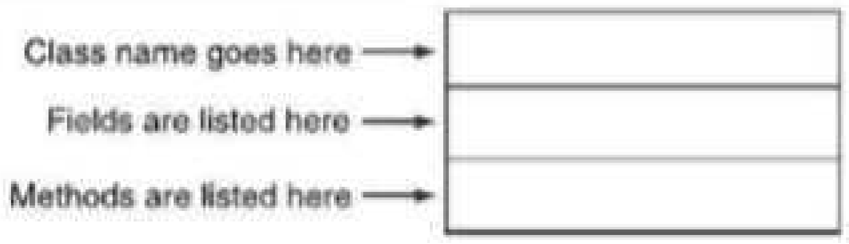
The Rectangle class will also have the following methods:

- setLength. The setLength method will store a value in an object's length field.
- setWidth. The setWidth method will store a value in an object's width field.

- `getLength`. The `getLength` method will return the value in an object's `length` field.
- `getWidth`. The `getWidth` method will return the value in an object's `width` field.
- `getArea`. The `getArea` method will return the area of the rectangle, which is the result of an object's length multiplied by its width.

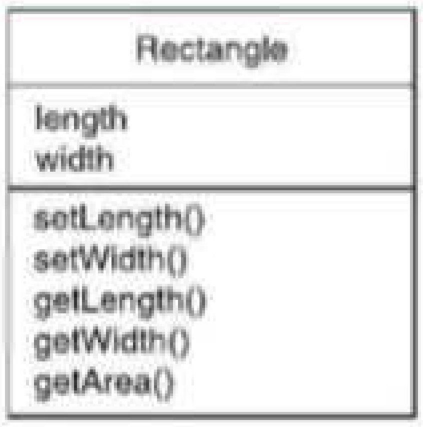
When designing a class it is often helpful to draw a UML diagram. UML stands for Unified Modeling Language. It provides a set of standard diagrams for graphically depicting object-oriented systems. Figure 6-8 shows the general layout of a UML diagram for a class. Notice that the diagram is a box that is divided into three sections. The top section is where you write the name of the class. The middle section holds a list of the class's fields. The bottom section holds a list of the class's methods.

Figure 6-8 General layout of a UML diagram for a class



Following this layout, Figure 6-9 shows a UML diagram for our `Rectangle` class. Throughout this book we frequently use UML diagrams to illustrate classes.

Figure 6-9 UML diagram for the `Rectangle` class



Writing the Code for a Class

Now that we have identified the fields and methods that we want the `Rectangle` class to have, let's write the Java code. First, we use an editor to create a new file named *Rectangle.java*. In the *Rectangle.java* file we will start by writing a general class "skeleton" as follows:

VideoNote
Writing Classes and Creating Objects

```
public class Rectangle
{
}
```

The key word `public`, which appears in the first line, is an access specifier. An access specifier indicates how the class may be accessed. The `public` access specifier indicates that the class will be publicly available to code that is written outside the *Rectangle.java* file. Almost all of the classes that we write in this book are `public`.

Following the access specifier is the key word `class`, followed by `Rectangle`, which is the name of the class. On the next line an opening brace appears, which is followed by a closing brace. The contents of the class, which are the fields and methods, will be written inside these braces. The general format of a class definition is as follows:

```
AccessSpecifier class Name
{
    Members
}
```

In general terms, the fields and methods that belong to a class are referred to as the class's *members*.

Writing the Code for the Class Fields

Let's continue writing our `Rectangle` class by filling in the code for some of its members. First we will write the code for the class's two fields, `length` and `width`. We will use variables of the `double` data type for the fields. The new lines of code are shown in bold, as follows:

```
public class Rectangle
{
    private double length;
    private double width;
}
```

These two lines of code that we have added declare the variables `length` and `width`. Notice that both declarations begin with the key word `private`, preceding the data type. The key word `private` is an access specifier. It indicates that these variables may not be accessed by statements outside the class.

By using the `private` access modifier, a class can hide its data from code outside the class. When a class's fields are hidden from outside code, the data is protected from accidental corruption. It is a common practice to make all of a class's fields `private` and to provide access to those fields through methods only. In other words, a class usually has `private` fields, and `public` methods that access those fields. Table 6-1 summarizes the difference between the `private` and `public` access specifiers.

Table 6-1 Summary of the `private` and `public` access specifiers for class members

Access Specifier	Description
`private`	When the `private` access specifier is applied to a class member, the member cannot be accessed by code outside the class. The member can be accessed only by methods that are members of the same class.
`public`	When the `public` access specifier is applied to a class member, the member can be accessed by code inside the class or outside.

Writing the `setLength` Method

Now we will begin writing the class methods. We will start with the `setLength` method. This method will allow code outside the class to store a value in the `length` field.

Code Listing 6-2 shows the `Rectangle` class at this stage of its development. The `setLength` method is in lines 17 through 20. (This file is in the source code folder *Chapter 06\Rectangle Class Phase 1*.)

Code Listing 6-2 **(`Rectangle.java`)**

```
 1  /**
 2     Rectangle class, phase 1
 3     Under construction!
 4  */
 5
 6  public class Rectangle
 7  {
 8     private double length;
 9     private double width;
10
11     /**
12        The setLength method stores a value in the
13        length field.
14        @param len The value to store in length.
15     */
16
17     public void setLength(double len)
18     {
19        length = len;
20     }
21  }
```

In lines 11 through 15, we write a block comment that gives a brief description of the method. It's important always to write comments that describe a class's methods so that in the future, anyone reading the code will understand it. The definition of the method appears in lines 17 through 20. Here is the method header:

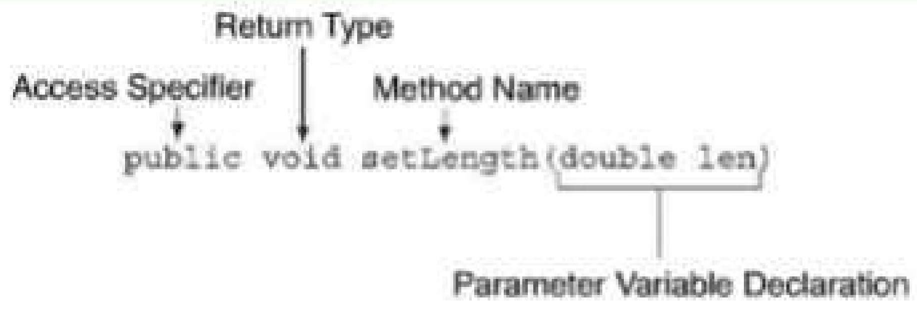
```
public void setLength(double len)
```

The method header looks very much like any other method header that you learned to write in Chapter 5. Let's look at the parts as follows:

- `public`. The key word `public` is an access specifier. It indicates that the method may be called by statements outside the class.
- `void`. This is the method's return type. The key word `void` indicates that the method returns no data to the statement that called it.
- `setLength`. This is the name of the method.
- `(double len)`. This is the declaration of a parameter variable of the `double` data type, named `len`.

Figure 6-10 labels each part of the header for the `setLength` method.

Figure 6-10 Header for the `setLength` method


Return Type
Access Specifier
Method Name
public void setLength(double len)
Parameter Variable Declaration

Notice that the word `static` does not appear in the method header. When a method is designed to work on an instance of a class, it is referred to as an *instance method*, and you do not write the word `static` in the header. Because this method will store a value in the `length` field of an instance of the `Rectangle` class, it is an instance method. We will discuss this in greater detail later.

After the header, the body of the method appears inside a set of braces:

```
{
    length = len;
}
```

The body of this method has only one statement, which assigns the value of `len` to the `length` field. When the method executes, the `len` parameter variable will hold the value of an argument that is passed to the method. That value is assigned to the `length` field.

Before adding the other methods to the class, it might help if we demonstrate how the `setLength` method works. First, notice that the `Rectangle` class does not have a `main` method. This class is not a complete program, but is a blueprint that `Rectangle` objects may be created from. Other programs will use the `Rectangle` class to create objects. The programs that create and use these objects will have their own `main` methods. We can demonstrate the class's `setLength` method by saving the current contents of the *Rectangle.java* file and then creating the program shown in Code Listing 6-3.

Code Listing 6-3 (`LengthDemo.java`)

```
/**
   This program demonstrates the Rectangle class's
   setLength method.
*/

public class LengthDemo
{
   public static void main(String[] args)
   {
      // Create a Rectangle object and assign its
      // address to the box variable.
12         Rectangle box = new Rectangle();
13
14         // Indicate what we are doing.
15         System.out.println("Sending the value 10.0 " +
16                            "to the setLength method.");
17
18         // Call the box object's setLength method.
19         box.setLength(10.0);
20
21         // Indicate we are done.
22         System.out.println("Done.");
23     }
24 }
```

Program Output

```
Sending the value 10.0 to the setLength method.
Done.
```

The program in Code Listing 6-3 must be saved as *LengthDemo.java* in the same folder or directory as the file *Rectangle.java*. The following command can then be used with the Sun JDK to compile the program:

```
javac LengthDemo.java
```

When the compiler reads the source code for `LengthDemo.java` and sees that a class named `Rectangle` is being used, it looks in the current folder or directory for the file *Rectangle.class*. That file does not exist, however, because we have not yet compiled *Rectangle.java*. So, the compiler searches for the file `Rectangle.java` and compiles it. This creates the file *Rectangle.class*, which makes the `Rectangle` class available. The compiler then finishes compiling `LengthDemo.java`. The resulting *LengthDemo.class* file may be executed with the following command:

```
java LengthDemo
```

The output of the program is shown at the bottom of Code Listing 6-3.

Let's look at each statement in this program's `main` method. First, the program uses the following statement, in line 12, to create a `Rectangle` object and associate it with a variable:

```
Rectangle box = new Rectangle();
```

Let's dissect the statement into two parts. The first part of the statement,

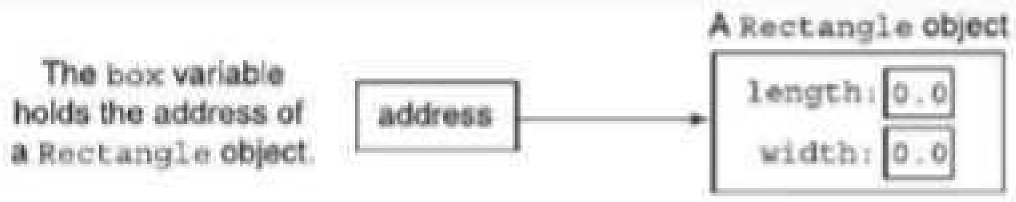
```
Rectangle box
```

declares a variable named `box`. The data type of the variable is `Rectangle`. (Because the word `Rectangle` is not the name of a primitive data type, Java assumes it to be the name of a class.) Recall that a variable of a class type is a reference variable, and it holds the memory address of an object. When a reference variable holds an object's memory address, it is said that the variable references the object. So, the variable `box` will be used to reference a `Rectangle` object. The second part of the statement is as follows:

```
= new Rectangle();
```

This part of the statement uses the key word `new`, which creates an object in memory. After the word `new`, the name of a class followed by a set of parentheses appears. This specifies the class that the object should be created from. In this case, an object of the `Rectangle` class is created. The memory address of the object is then assigned (by the = operator) to the variable `box`. After the statement executes, the variable `box` will reference the object that was created in memory. This is illustrated in Figure 6-11.

Figure 6-11 The box variable references a `Rectangle` class object



Notice that Figure 6-11 shows the `Rectangle` object's `length` and `width` fields set to 0. All of a class's numeric fields are initialized to 0 by default.

TIP: The parentheses in this statement are required. It would be an error to write the statement as follows:

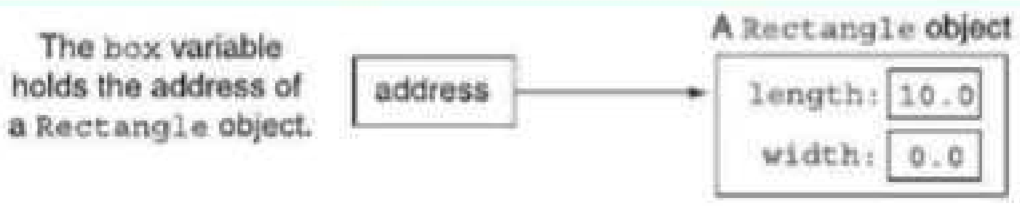
```
Rectangle box = new Rectangle; // ERROR!!
```

The statement in lines 15 and 16 uses the `System.out.println` method to display a message on the screen. The next statement, in line 19, calls the `box` object's `setLength` method as follows:

```
box.setLength(10.0);
```

This statement passes the argument 10.0 to the `setLength` method. When the method executes, the value 10.0 is copied into the `len` parameter variable. The method assigns the value of `len` to the `length` field and then terminates. Figure 6-12 shows the state of the `box` object after the method executes.

Figure 6-12 The state of the box object after the `setLength` method executes



Writing the `setWidth` Method

Now that we've seen how the `setLength` method works, let's add the `setWidth` method to the `Rectangle` class. The `setWidth` method is similar to `setLength`. It accepts an argument, which is assigned to the `width` field. Code Listing 6-4 shows the updated `Rectangle` class. The `setWidth` method is in lines 28 through 31. (This file is stored in the source code folder *Chapter 06\Rectangle Class Phase 2*.)

Code Listing 6-4 (`Rectangle.java`)

```
 1  /**
 2     Rectangle class, phase 2
 3     Under construction!
 4  */
 5
 6  public class Rectangle
 7  {
 8     private double length;
 9     private double width;
10
11     /**
12        The setLength method stores a value in the
13        length field.
14        @param len The value to store in length.
15     */
16
17     public void setLength(double len)
18     {
19        length = len;
20     }
21
22     /**
23        The setWidth method stores a value in the
24        width field.
25        @param w The value to store in width.
26     */
27
28     public void setWidth(double w)
29     {
30        width = w;
31     }
32  }
```

The setWidth method has a parameter variable named w, which is assigned to the width field. For example, assume that box references a Rectangle object and the following statement is executed:

```
box.setWidth(20.0);
```

After this statement executes, the box object's width field will be set to 20.0.

Writing the getLength and getWidth Methods

Because the length and width fields are private, we wrote the setLength and setWidth methods to allow code outside the Rectangle class to store values in the fields. We must also write methods that allow code outside the class to get the values that are stored in these fields. That's what the getLength and getWidth methods will do. The getLength method will return the value stored in the length field, and the getWidth method will return the value stored in the width field.

Here is the code for the getLength method:

```
public double getLength()
{
    return length;
}
```

Assume that size is a double variable and that box references a Rectangle object, and the following statement is executed:

```
size = box.getLength();
```

This statement assigns the value that is returned from the getLength method to the size variable. After this statement executes, the size variable will contain the same value as the box object's length field.

The getWidth method is similar to getLength. The code for the method follows:

```
public double getWidth()
{
     return width;
}
```

This method returns the value that is stored in the width field. For example, assume that size is a double variable and that box references a Rectangle object, and the following statement is executed:

```
size = box.getWidth();
```

This statement assigns the value that is returned from the getWidth method to the size variable. After this statement executes, the size variable will contain the same value as the box object's width field.

Code Listing 6-5 shows the Rectangle class with all of the members we have discussed so far. The code for the getLength and getWidth methods is shown in lines 33 through 53. (This file is stored in the source code folder *Chapter 06\Rectangle Class Phase 3*.)

Code Listing 6-5 **(Rectangle.java)**

```
/**
   Rectangle class, phase 3
   Under construction!
*/

public class Rectangle
{
   private double length;
   private double width;

   /**
      The setLength method stores a value in the
      length field.
      @param len The value to store in length.
   */

   public void setLength(double len)
   {
      length = len;
   }

   /**
      The setWidth method stores a value in the
      width field.
      @param w The value to store in width.
   */

   public void setWidth(double w)
   {
      width = w;
   }

   /**
      The getLength method returns a Rectangle
      object's length.
      @return The value in the length field.
   */

   public double getLength()
   {
      return length;
   }

   /**
      The getWidth method returns a Rectangle
      object's width.
      @return The value in the width field.
   */

   public double getWidth()
   {
      return width;
   }
}
```

Before continuing we should demonstrate how these methods work. Look at the program in Code Listing 6-6. (This file is also stored in the source code folder *Chapter 06\Rectangle Class Phase 3.*)

Code Listing 6-6 (`LengthWidthDemo.java`)

```
/**
   This program demonstrates the Rectangle class's
   setLength, setWidth, getLength, and getWidth methods.
*/

public class LengthWidthDemo
{
   public static void main(String[] args)
   {
      // Create a Rectangle object.
      Rectangle box = new Rectangle();

      // Call the object's setLength method, passing 10.0
      // as an argument.
      box.setLength(10.0);

      // Call the object's setWidth method, passing 20.0
      // as an argument.
      box.setWidth(20.0);

      // Display the object's length and width.
      System.out.println("The box's length is " +
                         box.getLength());
      System.out.println("The box's width is " +
                         box.getWidth());
   }
}
```

Program Output

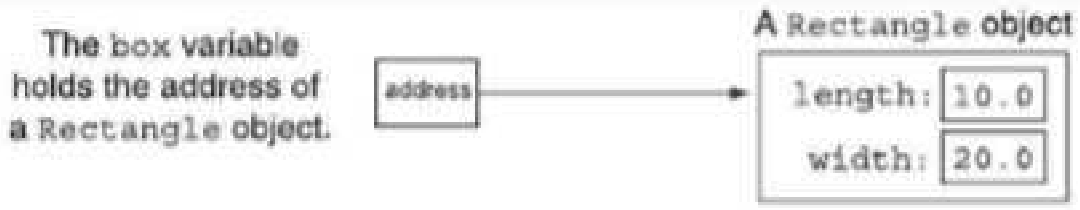
```
The box's length is 10.0
The box's width is 20.0
```

Let's take a closer look at the program. In line 11 this program creates a `Rectangle` object, which is referenced by the `box` variable. Then the following statements execute in lines 15 and 19:

```
box.setLength(10.0);
box.setWidth(20.0);
```

After these statements execute, the `box` object's `length` field is set to 10.0 and its `width` field is set to 20.0. The state of the object is shown in Figure 6-13.

Figure 6-13 State of the box object



Next, the following statement in lines 22 and 23 executes as follows:

```
System.out.println("The box's length is " +
                   box.getLength());
```

This statement calls the `box.getLength()` method, which returns the value 10.0. The following message is displayed on the screen:

```
The box's length is 10.0
```

Then the following statement executes in lines 24 and 25:

```
System.out.println("The box's width is " +
                   box.getWidth());
```

This statement calls the `box.getWidth()` method, which returns the value 20.0. The following message is displayed on the screen:

```
The box's width is 20.0
```

Writing the `getArea` Method

The last method we will write for the `Rectangle` class is `getArea`. This method returns the area of a rectangle, which is its length multiplied by its width. Here is the code for the `getArea` method:

```
public double getArea()
{
    return length * width;
}
```

This method returns the result of the mathematical expression `length * width`. For example, assume that `area` is a `double` variable and that `box` references a `Rectangle` object, and the following code is executed:

```
box.setLength(10.0);
box.setWidth(20.0);
area = box.getArea();
```

The last statement assigns the value that is returned from the `getArea` method to the `area` variable. After this statement executes, the `area` variable will contain the value 200.0.

Code Listing 6-7 shows the `Rectangle` class with all of the members we have discussed so far. The `getArea` method appears in lines 61 through 64. (This file is stored in the source code folder *Chapter 06\Rectangle Class Phase 4*.)

Code Listing 6-7 **(Rectangle.java)**

```
 1  /**
 2     Rectangle class, phase 4
 3     Under construction!
 4  */
 5
 6  public class Rectangle
 7  {
 8     private double length;
 9     private double width;
10
11     /**
12        The setLength method stores a value in the
13        length field.
14        @param len The value to store in length.
15     */
16
17     public void setLength(double len)
18     {
19        length = len;
20     }
21
22     /**
23        The setWidth method stores a value in the
24        width field.
25        @param w The value to store in width.
26     */
27
28     public void setWidth(double w)
29     {
30        width = w;
31     }
32
33     /**
34        The getLength method returns a Rectangle
35        object's length.
36        @return The value in the length field.
37     */
38
```

```
   public double getLength()
   {
      return length;
   }

   /**
      The getWidth method returns a Rectangle
      object's width.
      @return The value in the width field.
   */

   public double getWidth()
   {
      return width;
   }

   /**
      The getArea method returns a Rectangle
      object's area.
      @return The product of length times width.
   */

   public double getArea()
   {
      return length * width;
   }
}
```

The program in Code Listing 6-8 demonstrates all the methods of the `Rectangle` class, including `getArea`. (This file is also stored in the source code folder *Chapter 06\Rectangle Class Phase 4*.)

Code Listing 6-8 (`RectangleDemo.java`)

```
/**
   This program demonstrates the Rectangle class's
   setLength, setWidth, getLength, getWidth, and
   getArea methods.
*/

public class RectangleDemo
{
   public static void main(String[] args)
   {
      // Create a Rectangle object.
      Rectangle box = new Rectangle();

```

```
        // Set length to 10.0 and width to 20.0.
        box.setLength(10.0);
        box.setWidth(20.0);

        // Display the length.
        System.out.println("The box's length is " +
                           box.getLength());

        // Display the width.
        System.out.println("The box's width is " +
                           box.getWidth());

        // Display the area.
        System.out.println("The box's area is " +
                           box.getArea());
    }
}
```

Program Output

```
The box's length is 10.0
The box's width is 20.0
The box's area is 200.0
```

Accessor and Mutator Methods

As mentioned earlier, it is a common practice to make all of a class's fields private and to provide public methods for accessing and changing those fields. This ensures that the object owning those fields is in control of all changes being made to them. A method that gets a value from a class's field but does not change it is known as an *accessor method*. A method that stores a value in a field or changes the value of a field in some other way is known as a *mutator method*. In the `Rectangle` class, the methods `getLength` and `getWidth` are accessors, and the methods `setLength` and `setWidth` are mutators.

NOTE: Mutator methods are sometimes called "setters" and accessor methods are sometimes called "getters."

The Importance of Data Hiding

Data hiding is an important concept in object-oriented programming. An object hides its internal data from code that is outside the class that the object is an instance of. Only the class's methods may directly access and make changes to the object's internal data. You hide an object's internal data by making the class's fields `private`, and making the methods that access those fields `public`.

As a beginning student, you might be wondering why you would want to hide the data that is inside the classes you create. As you learn to program, you will be the user of your own classes, so it might seem that you are putting forth a great effort to hide data from yourself. If you write software in industry, however, the classes that you create will be used as components in large software systems, and programmers other than yourself will be using your classes. By hiding a class's data, and allowing it to be accessed only through the class's methods, you can better ensure that the class will operate as you intended it to.

Avoiding Stale Data

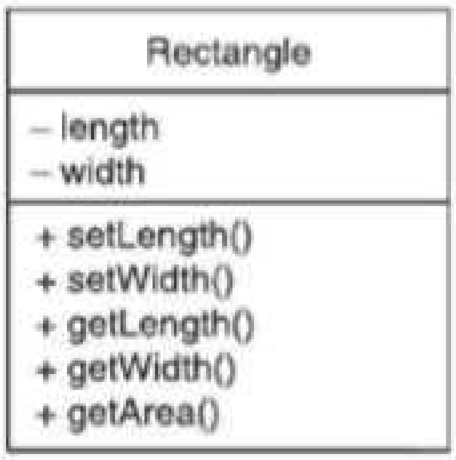
In the `Rectangle` class, the `getLength` and `getWidth` methods return the values stored in fields, but the `getArea` method returns the result of a calculation. You might be wondering why the area of the rectangle is not stored in a field, like the length and the width. The area is not stored in a field because it could potentially become stale. When the value of an item is dependent on other data and that item is not updated when the other data is changed, it is said that the item has become *stale*. If the area of the rectangle were stored in a field, the value of the field would become incorrect as soon as either the `length` or `width` field changed.

When designing a class, you should take care not to store in a field calculated data that can potentially become stale. Instead, provide a method that returns the result of the calculation.

Showing Access Specification in UML Diagrams

In Figure 6-9 we presented a UML diagram for the `Rectangle` class. The diagram listed all of the members of the class but did not indicate which members were private and which were public. In a UML diagram, you have the option to place a - character before a member name to indicate that it is private, or a + character to indicate that it is public. Figure 6-14 shows the UML diagram modified to include this notation.

Figure 6-14 UML diagram for the `Rectangle` class



Data Type and Parameter Notation in UML Diagrams

The Unified Modeling Language also provides notation that you may use to indicate the data types of fields, methods, and parameter variables. To indicate the data type of a field, place a colon followed by the name of the data type after the name of the field. For example, the `length` field in the `Rectangle` class is a `double`. It could be listed in the UML diagram as follows:

```
- length : double
```

The return type of a method can be listed in the same manner: After the method's name, place a colon followed by the return type. The `Rectangle` class's `getLength` method returns a `double`, so it could be listed in the UML diagram as follows:

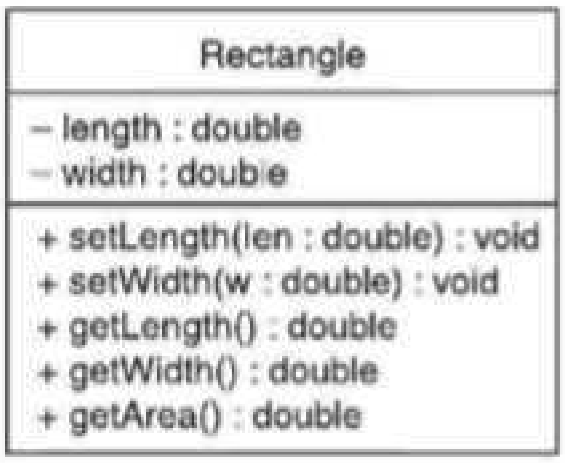
```
+ getLength() : double
```

Parameter variables and their data types may be listed inside a method's parentheses. For example, the `Rectangle` class's `setLength` method has a `double` parameter named `len`, so it could be listed in the UML diagram as follows:

```
+ setLength(len : double) : void
```

Figure 6-15 shows a UML diagram for the `Rectangle` class with parameter and data type notation.

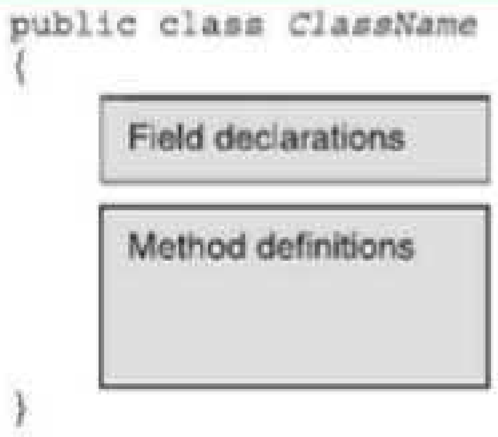
Figure 6-15 UML diagram for the `Rectangle` class with parameter and data type notation



Layout of Class Members

Notice that in the `Rectangle` class, the field variables are declared first and then the methods are defined. You are not required to write field declarations before the method definitions. In fact, some programmers prefer to write the definitions for the public methods first and write the declarations for the private fields last. Regardless of which style you use, you should be consistent. In this book we always write the field declarations first, followed by the method definitions. Figure 6-16 shows this layout.

Figure 6-16 Typical layout of class members



Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

6.8 You hear someone make the following comment: "A blueprint is a design for a house. A carpenter can use the blueprint to build the house. If the carpenter wishes, he or she can build several identical houses from the same blueprint." Think of this as a metaphor for classes and objects. Does the blueprint represent a class, or does it represent an object?

6.9 In this chapter we used the metaphor of a kite attached to a spool of string to describe the relationship between an object and a reference variable. In this metaphor, does the kite represent an object, or a reference variable?

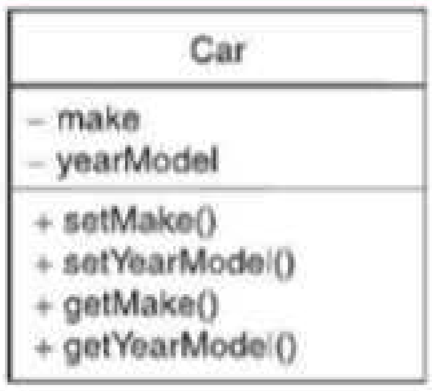
6.10 When a variable is said to reference an object, what is actually stored in the variable?

6.11 A string literal, such as `"Joe"`, causes what type of object to be created?

6.12 Look at the UML diagram in Figure 6-17 and answer the following questions:
   a) What is the name of the class?
   b) What are the fields?
   c) What are the methods?
   d) What are the private members?
   e) What are the public members?

6.13 Assume that limo is a variable that references an instance of the class shown in Figure 6-17. Write a statement that calls `setMake` and passes the argument `"Cadillac"`.

Figure 6-17 UML diagram



6.14 What does the key word `new` do?

6.15 What is an accessor? What is a mutator?

6.16 What is a stale data item?

6.3 Instance Fields and Methods

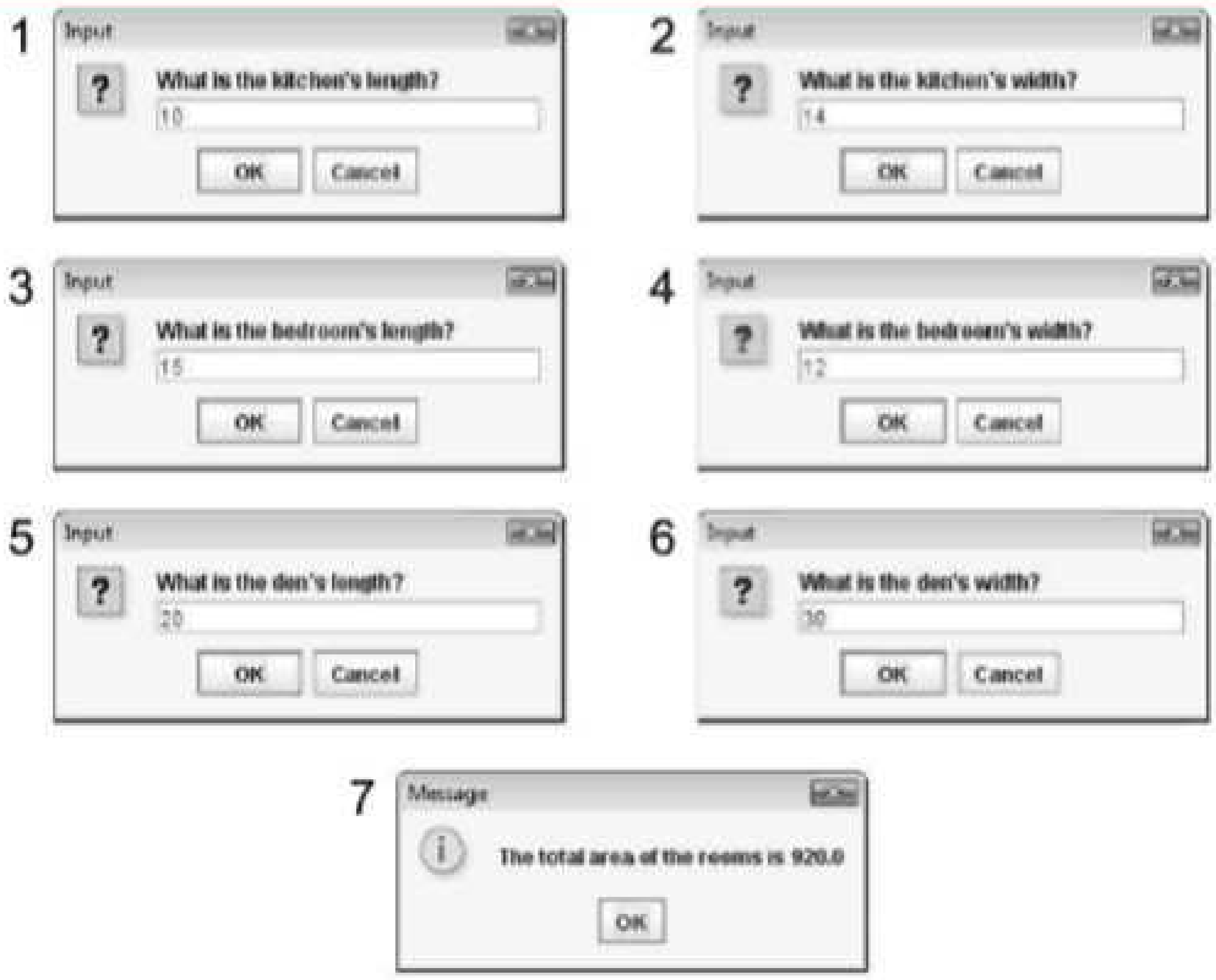
CONCEPT: Each instance of a class has its own set of fields, which are known as instance fields. You can create several instances of a class and store different values in each instance's fields. The methods that operate on an instance of a class are known as instance methods.

The program in Code Listing 6-8 creates one instance of the `Rectangle` class. It is possible to create many instances of the same class, each with its own data. For example, the `RoomAreas.java` program in Code Listing 6-9 creates three instances of the `Rectangle` class, referenced by the variables `kitchen`, `bedroom`, and `den`. Figure 6-18 shows example interaction with the program. (The file in Code Listing 6-9 is stored in the source code folder *Chapter 06\Rectangle Class Phase 4*.)

Code Listing 6-9 (RoomAreas.java)

```java
import javax.swing.JOptionPane;

/**
   This program creates three instances of the
   Rectangle class.
*/

public class RoomAreas
{
   public static void main(String[] args)
   {
      double number;        // To hold a number
      double totalArea;     // The total area
      String input;         // To hold user input

      // Create three Rectangle objects.
      Rectangle kitchen = new Rectangle();
      Rectangle bedroom = new Rectangle();
      Rectangle den = new Rectangle();

      // Get and store the dimensions of the kitchen.
      input = JOptionPane.showInputDialog("What is the " +
                                          "kitchen's length?");
      number = Double.parseDouble(input);
      kitchen.setLength(number);
      input = JOptionPane.showInputDialog("What is the " +
                                          "kitchen's width?");
      number = Double.parseDouble(input);
      kitchen.setWidth(number);

      // Get and store the dimensions of the bedroom.
      input = JOptionPane.showInputDialog("What is the " +
                                          "bedroom's length?");
      number = Double.parseDouble(input);
      bedroom.setLength(number);
      input = JOptionPane.showInputDialog("What is the " +
                                          "bedroom's width?");
      number = Double.parseDouble(input);
      bedroom.setWidth(number);

      // Get and store the dimensions of the den.
      input = JOptionPane.showInputDialog("What is the " +
                                          "den's length?");
      number = Double.parseDouble(input);
      den.setLength(number);
      input = JOptionPane.showInputDialog("What is the " +
                                          "den's width?");
      number = Double.parseDouble(input);
      den.setWidth(number);

      // Calculate the total area of the rooms.
      totalArea = kitchen.getArea() + bedroom.getArea()
                  + den.getArea();

      // Display the total area of the rooms.
      JOptionPane.showMessageDialog(null, "The total area " +
                                 "of the rooms is " + totalArea);

      System.exit(0);
   }
}
```

Figure 6-18 Interaction with the `RoomAreas.java` program



What is the kitchen's length?
10

What is the kitchen's width?
14

What is the bedroom's length?
15

What is the bedroom's width?
12

What is the den's length?
20

What is the den's width?
30

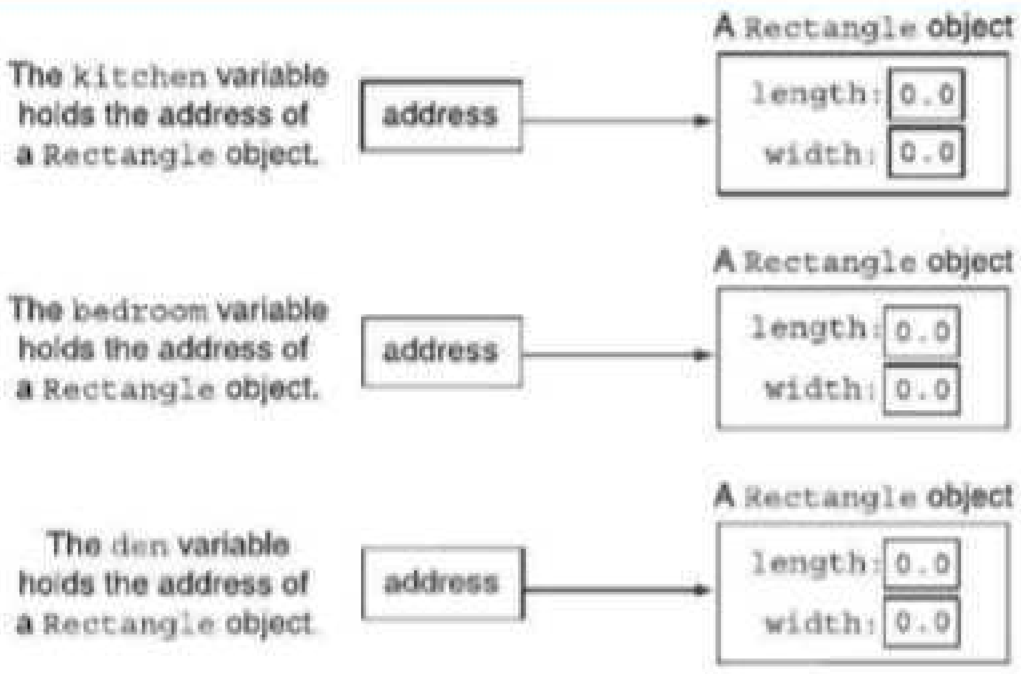
The total area of the rooms is 920.0

In lines 17, 18, and 19, the following code creates three objects, each an instance of the `Rectangle` class:

```
Rectangle kitchen = new Rectangle();
Rectangle bedroom = new Rectangle();
Rectangle den = new Rectangle();
```

Figure 6-19 illustrates how the `kitchen`, `bedroom`, and `den` variables reference the objects.

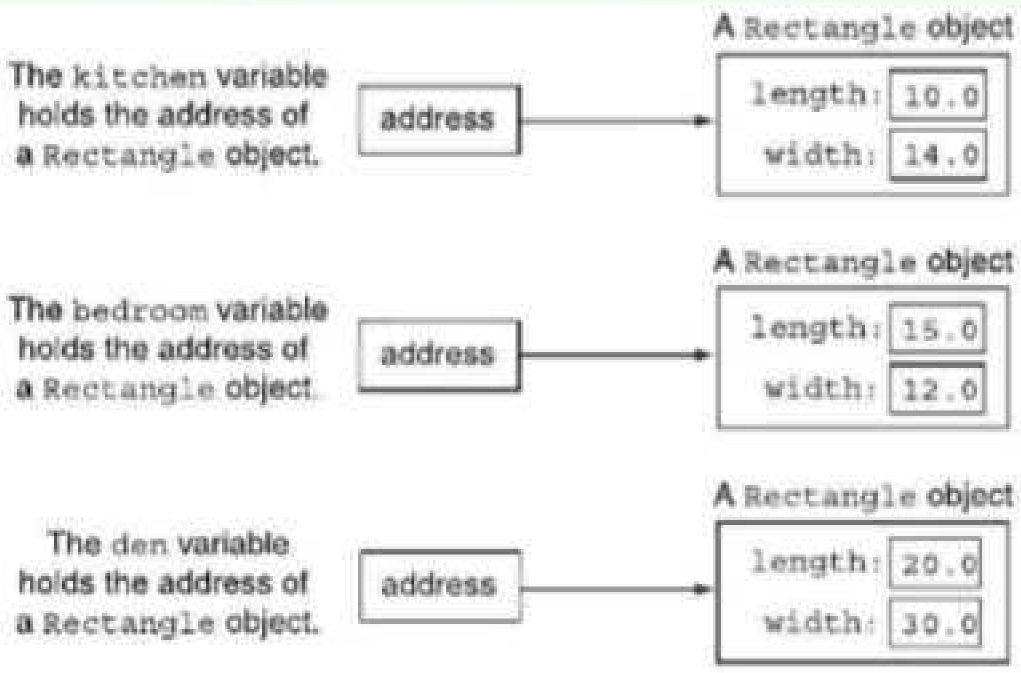
Figure 6-19 The `kitchen`, `bedroom`, and `den` variables reference `Rectangle` objects



A Rectangle object
The kitchen variable holds the address of a Rectangle object.
address
length: 0.0
width: 0.0
A Rectangle object
The bedroom variable holds the address of a Rectangle object.
address
length: 0.0
width: 0.0
A Rectangle object
The den variable holds the address of a Rectangle object.
address
length: 0.0
width: 0.0

In the example session with the program, the user enters 10 and 14 as the length and width of the kitchen, 15 and 12 as the length and width of the bedroom, and 20 and 30 as the length and width of the den. Figure 6-20 shows the states of the objects after these values are stored in them.

Figure 6-20 States of the objects after data has been stored in them



A Rectangle object
The kitchen variable holds the address of a Rectangle object.
address
length: 10.0
width: 14.0
A Rectangle object
The bedroom variable holds the address of a Rectangle object.
address
length: 15.0
width: 12.0
A Rectangle object
The den variable holds the address of a Rectangle object.
address
length: 20.0
width: 30.0

Notice from Figure 6-20 that each instance of the `Rectangle` class has its own `length` and `width` variables. For this reason, the variables are known as *instance variables*, or *instance fields*. Every instance of a class has its own set of instance fields and can store its own values in those fields.

The methods that operate on an instance of a class are known as *instance methods*. All of the methods in the `Rectangle` class are instance methods because they perform operations on specific instances of the class. For example, look at the following statement in line 25 of the `RoomAreas.java` program:

```
kitchen.setLength(number);
```

This statement calls the `setLength` method, which stores a value in the `kitchen` object's `length` field. Now look at the following statement in line 35:

```
bedroom.setLength(number);
```

This statement also calls the `setLength` method, but this time it stores a value in the `bedroom` object's `length` field. Likewise, the following statement in line 45 calls the `setLength` method to store a value in the `den` object's `length` field:

```
den.setLength(number);
```

The `setLength` method stores a value in a specific instance of the `Rectangle` class. This is true of all of the methods that are members of the `Rectangle` class.

NOTE: As previously mentioned, instance methods do not have the key word `static` in their headers.

Checkpoint

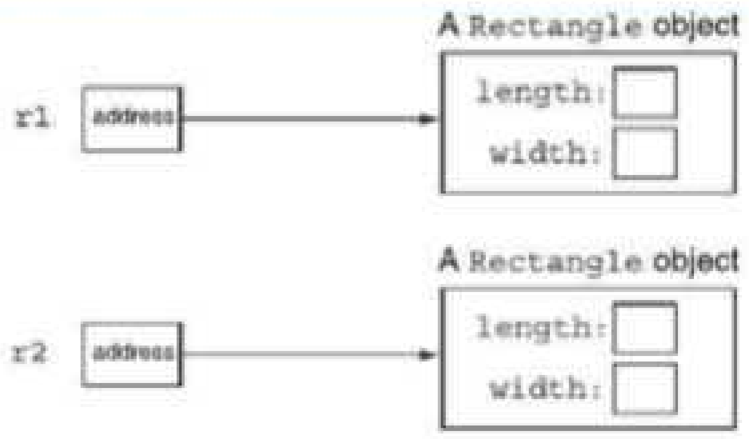
MyProgrammingLab™ *www.myprogramminglab.com*

6.17 Assume that `r1` and `r2` are variables that reference `Rectangle` objects, and the following statements are executed:

```
r1.setLength(5.0);
r2.setLength(10.0);
r1.setWidth(20.0);
r2.setWidth(15.0);
```

Fill in the boxes in Figure 6-21 that represent each object's length and width fields.

Figure 6-21 Fill in the boxes for each field



6.4 Constructors

CONCEPT: A constructor is a method that is automatically called when an object is created.

VideoNote Initializing an Object with a Constructor

A constructor is a method that is automatically called when an instance of a class is created. Constructors normally perform initialization or setup operations, such as storing initial values in instance fields. They are called "constructors" because they help construct an object.

A constructor method has the same name as the class. For example, Code Listing 6-10 shows the first few lines of a new version of the `Rectangle` class. In this version of the class, a constructor has been added. (This file is stored in the source code folder *Chapter 06\Rectangle Class Phase 5*.)

Code Listing 6-10 (`Rectangle.java`)

```
/**
   Rectangle class, phase 5
*/

public class Rectangle
{
   private double length;
   private double width;

   /**
      Constructor
      @param len The length of the rectangle.
      @param w The width of the rectangle.
   */

   public Rectangle(double len, double w)
   {
      length = len;
      width = w;
   }
```

. . . The remainder of the class has not changed, and is not shown.

This constructor accepts two arguments, which are passed into the `len` and `w` parameter variables. The parameter variables are then assigned to the `length` and `width` fields.

Notice that the constructor's header doesn't specify a return type—not even `void`. This is because constructors are not executed by explicit method calls and cannot return a value.

The method header for a constructor takes the following general format:

```
AccessSpecifier ClassName(Parameters...)
```

Here is an example statement that declares the variable `box`, creates a `Rectangle` object, and passes the values 7.0 and 14.0 to the constructor.

```
Rectangle box = new Rectangle(7.0, 14.0);
```

After this statement executes, `box` will reference a `Rectangle` object whose `length` field is set to 7 and whose `width` field is set to 14. The program in Code Listing 6-11 demonstrates the `Rectangle` class constructor. (This file is also stored in the source code folder *Chapter 06\ Rectangle Class Phase 5.*)

Code Listing 6-11 **(ConstructorDemo.java)**

```
/**
   This program demonstrates the Rectangle class's
   constructor.
*/

public class ConstructorDemo
{
   public static void main(String[] args)
   {
      // Create a Rectangle object, passing 5.0 and
      // 15.0 as arguments to the constructor.
      Rectangle box = new Rectangle(5.0, 15.0);

      // Display the length.
      System.out.println("The box's length is " +
                         box.getLength());

      // Display the width.
      System.out.println("The box's width is " +
                         box.getWidth());

      // Display the area.
      System.out.println("The box's area is " +
                         box.getArea());
   }
}
```

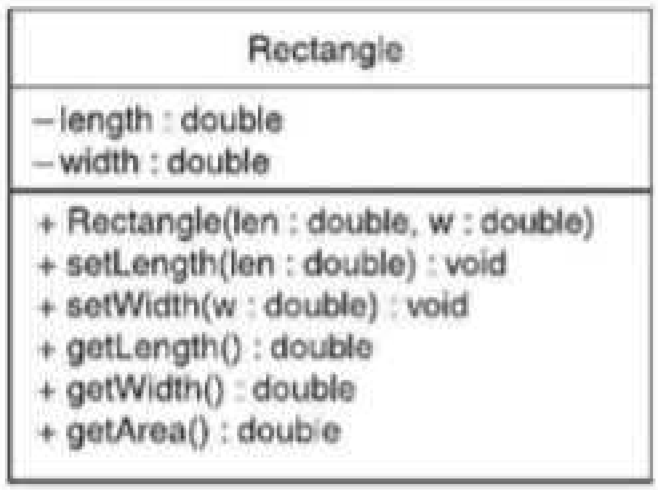
Program Output

```
The box's length is 5.0
The box's width is 15.0
The box's area is 75.0
```

Showing Constructors in a UML Diagram

There is more than one accepted way of showing a class's constructor in a UML diagram. In this book, we simply show a constructor just as any other method, except we list no return type. Figure 6-22 shows a UML diagram for the `Rectangle` class with the constructor listed.

Figure 6-22 UML diagram for the `Rectangle` class showing the constructor



Uninitialized Local Reference Variables

The program in Code Listing 6-11 initializes the `box` variable with the address of a `Rectangle` object. Reference variables can also be declared without being initialized, as in the following statement:

```
Rectangle box;
```

Note that this statement does not create a `Rectangle` object. It only declares a variable named `box` that can be used to reference a `Rectangle` object. Because the `box` variable does not yet hold an object's address, it is an *uninitialized reference variable*.

After declaring the reference variable, the following statement can be used to assign it the address of an object. This statement creates a `Rectangle` object, passes the values 7.0 and 14.0 to its constructor, and assigns the object's address to the `box` variable:

```
box = new Rectangle(7.0, 14.0);
```

Once this statement executes, the `box` variable will reference a `Rectangle` object.

You need to be careful when using uninitialized reference variables. Recall from Chapter 5 that local variables *must* be initialized or assigned a value before they can be used. This is also true for local reference variables. A local reference variable must reference an object before it can be used. Otherwise a compiler error will occur.

The Default Constructor

When an object is created, its constructor is *always* called. But what if we do not write a constructor in the object's class? If you do not write a constructor in a class, Java automatically provides one when the class is compiled. The constructor that Java provides is known as the *default constructor*. The default constructor doesn't accept arguments. It sets all of the object's numeric fields to 0 and `boolean` fields to `false`. If the object has any fields that are reference variables, the default constructor sets them to the special value `null`, which means that they do not reference anything.

The *only* time that Java provides a default constructor is when you do not write your own constructor for a class. For example, at the beginning of this chapter we developed the `Rectangle` class without writing a constructor for it. When we compiled the class, the compiler generated a default constructor that set both the `length` and `width` fields to 0.0. Assume that the following code uses that version of the class to create a `Rectangle` object:

```
// We wrote no constructor for the Rectangle class.
Rectangle r = new Rectangle();  // Calls the default constructor
```

When we created `Rectangle` objects using that version of the class, we did not pass any arguments to the default constructor because the default constructor doesn't accept arguments.

Later we added our own constructor to the class. The constructor that we added accepts arguments for the `length` and `width` fields. When we compiled the class at that point, Java did not provide a default constructor. The constructor that we added became the only constructor that the class has. When we create `Rectangle` objects with that version of the class, we *must* pass the length and width arguments to the constructor. Using that version of the class, the following statement would cause an error because we have not provided arguments for the constructor:

```
// Now we wrote our own constructor for the Rectangle class.
Rectangle box = new Rectangle(); // Error! Must now pass arguments.
```

Because we have added our own constructor, which requires two arguments, the class no longer has a default constructor.

Writing Your Own No-Arg Constructor

A constructor that does not accept arguments is known as a *no-arg constructor*. The default constructor doesn't accept arguments, so it is considered a no-arg constructor. In addition, you can write your own no-arg constructor. For example, suppose we wrote the following constructor for the `Rectangle` class:

```
public Rectangle()
{
    length = 1.0;
    width = 1.0;
}
```

If we were using this constructor in our `Rectangle` class, we would not pass any arguments when creating a `Rectangle` object. The following code shows an example. After this code executes, the `Rectangle` object's `length` and `width` fields would both be set to 1.0.

```
// Now we have written our own no-arg constructor.
Rectangle r = new Rectangle();  // Calls the no-arg constructor
```

The `String` Class Constructor

Earlier in this chapter (in Section 6.1) we discussed the difference between creating a primitive variable and creating an object. You create primitive variables with simple declaration statements, and you create objects with the `new` operator. There is one class, however, that can be instantiated without the `new` operator: the `String` class.

Because string operations are so common, Java allows you to create `String` objects in the same way that you create primitive variables. Here is an example:

```
String name = "Joe Mahoney";
```

This statement creates a `String` object in memory, initialized with the string literal `"Joe Mahoney"`. The object is referenced by the `name` variable. If you wish, you can use the `new` operator to create a `String` object, and initialize the object by passing a string literal to the constructor, as shown here:

```
String name = new String("Joe Mahoney");
```

NOTE: `String` objects are a special case in Java. Because they are so commonly used, Java provides numerous shortcut operations with `String` objects that are not possible with objects of other types. In addition to creating a `String` object without using the `new` operator, you can use the `=` operator to assign values to `String` objects, the `+` operator to concatenate strings, and so forth. Chapter 9 discusses several of the `String` class methods.

In the Spotlight:

Creating the `CellPhone` Class

Wireless Solutions, Inc., is a business that sells cell phones and wireless service. You are a programmer in the company's information technology (IT) department, and your team is designing a program to manage all of the cell phones that are in inventory. You have been asked to design a class that represents a cell phone. The data that should be kept as fields in the class are as follows:

- The name of the phone's manufacturer will be assigned to the `manufact` field.
- The phone's model number will be assigned to the `model` field.
- The phone's retail price will be assigned to the `retailPrice` field.

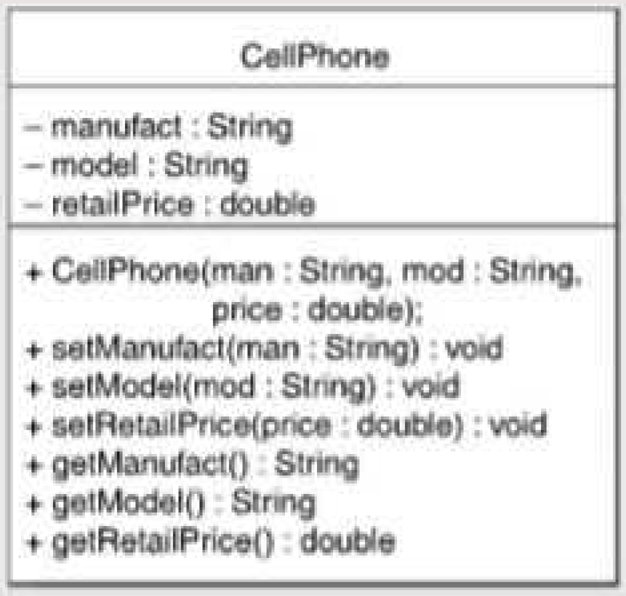
The class will also have the following methods:

- A constructor that accepts arguments for the manufacturer, model number, and retail price.
- A `setManufact` method that accepts an argument for the manufacturer. This method will allow us to change the value of the `manufact` field after the object has been created, if necessary.
- A `setModel` method that accepts an argument for the model. This method will allow us to change the value of the `model` field after the object has been created, if necessary.
- A `setRetailPrice` method that accepts an argument for the retail price. This method will allow us to change the value of the `retailPrice` field after the object has been created, if necessary.

- A `getManufact` method that returns the phone's manufacturer.
- A `getModel` method that returns the phone's model number.
- A `getRetailPrice` method that returns the phone's retail price.

Figure 6-23 shows a UML diagram for the class. Code Listing 6-12 shows the class definition.

Figure 6-23 UML diagram for the `CellPhone` class


CellPhone
– manufact : String
– model : String
– retailPrice : double
+ CellPhone(man : String, mod : String, price : double);
+ setManufact(man : String) : void
+ setModel(mod : String) : void
+ setRetailPrice(price : double) : void
+ getManufact() : String
+ getModel() : String
+ getRetailPrice() : double

Code Listing 6-12 (`CellPhone.java`)

```
/**
  The CellPhone class holds data about a cell phone.
*/

public class CellPhone
{
  // Fields
  private String manufact;       // Manufacturer
  private String model;          // Model
  private double retailPrice;    // Retail price

  /**
     Constructor
     @param man The phone's manufacturer.
     @param mod The phone's model number.
     @param price The phone's retail price.
  */

  public CellPhone(String man, String mod, double price)
  {
     manufact = man;
     model = mod;
     retailPrice = price;
  }
```

```

   /**
      The setManufact method sets the phone's
      manufacturer name.
      @param man The phone's manufacturer.
   */

   public void setManufact(String man)
   {
      manufact = man;
   }

   /**
      The setModel method sets the phone's
      model number.
      @param mod The phone's model number.
   */

   public void setMod(String mod)
   {
      model = mod;
   }

   /**
      The setRetailPrice method sets the phone's
      retail price.
      @param price The phone's retail price.
   */

   public void setRetailPrice(double price)
   {
      retailPrice = price;
   }

   /**
      getManufact method
      @return The name of the phone's manufacturer.
   */

   public String getManufact()
   {
      return manufact;
   }

   /**
      getModel method
      @return The phone's model number.
   */

   public String getModel()
   {
      return model;
   }

   /**
      getretailPrice method
      @return The phone's retail price.
   */

   public double getRetailPrice()
   {
      return retailPrice;
   }
}
```

The `CellPhone` class will be used by several programs that your team is developing. To perform a simple test of the class, you write the program shown in Code Listing 6-13. This is a simple program that prompts the user for the phone's manufacturer, model number, and retail price. An instance of the `CellPhone` class is created and the data is assigned to its attributes.

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

```
import java.util.Scanner;

/**
  This program runs a simple test
  of the CellPhone class.
*/

public class CellPhoneTest
{
  public static void main(String[] args)
  {
     String testMan;    // To hold a manufacturer
     String testMod;    // To hold a model number
     double testPrice;  // To hold a price

     // Create a Scanner object for keyboard input.
     Scanner keyboard = new Scanner(System.in);

     // Get the manufacturer name.
     System.out.print("Enter the manufacturer: ");
     testMan = keyboard.nextLine();

     // Get the model number.
     System.out.print("Enter the model number: ");
     testMod = keyboard.nextLine();
```

```

        // Get the retail price.
        System.out.print("Enter the retail price: ");
        testPrice = keyboard.nextDouble();

        // Create an instance of the CellPhone class,
        // passing the data that was entered as arguments
        // to the constructor.
        CellPhone phone = new CellPhone(testMan, testMod, testPrice);

        // Get the data from the phone and display it.
        System.out.println();
        System.out.println("Here is the data that you provided:");
        System.out.println("Manufacturer: " + phone.getManufact());
        System.out.println("Model number: " + phone.getModel());
        System.out.println("Retail price: " + phone.getRetailPrice());
    }
}
```

Program Output with Example Input Shown in Bold

```
Enter the manufacturer: Acme Electronics [Enter]
Enter the model number: M1000 [Enter]
Enter the retail price: 199.99 [Enter]

Here is the data that you provided:
Manufacturer: Acme Electronics
Model number: M1000
Retail price: $199.99
```

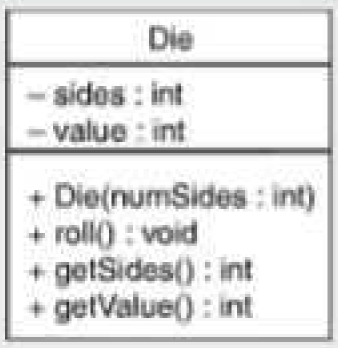
In the Spotlight:

Simulating Dice with Objects

Dice traditionally have six sides, representing the values 1 through 6. Some games, however, use specialized dice that have a different number of sides. For example, the fantasy role-playing game *Dungeons and Dragons®* uses dice with four, six, eight, ten, twelve, and twenty sides.

Suppose you are writing a program that needs to roll simulated dice with various numbers of sides. A simple approach would be to write a `Die` class with a constructor that accepts the number of sides as an argument. The class would also have appropriate methods for rolling the die, and getting the die's value. Figure 6-24 shows the UML diagram for such a class, and Code Listing 6-14 shows the code.

Figure 6-24 UML diagram for the `Die` class



Code Listing 6-14 (`Die.java`)

```
import java.util.Random;

/**
   The Die class simulates a six-sided die.
*/

public class Die
{
   private int sides;   // Number of sides
   private int value;   // The die's value

   /**
      The constructor performs an initial
      roll of the die.
      @param numSides The number of sides for this die.
   */

   public Die(int numSides)
   {
      sides = numSides;
      roll();
   }

   /**
      The roll method simulates the rolling of
      the die.
   */

   public void roll()
   {
      // Create a Random object.
      Random rand = new Random();

      // Get a random value for the die.
      value = rand.nextInt(sides) + 1;
   }
37
38    /**
39       getSides method
40       @return The number of sides for this die.
41    */
42
43    public int getSides()
44    {
45       return sides;
46    }
47
48    /**
49       getValue method
50       @return The value of the die.
51    */
52
53    public int getValue()
54    {
55       return value;
56    }
57 }
```

Let's take a closer look at the code for the class:

Lines 9 and 10: These statements declare two `int` fields. The `sides` field will hold the number of sides that the die has, and the `value` field will hold the value of the die once it has been rolled.

Lines 18–22: This is the constructor. Notice that the constructor has a parameter for the number of sides. The parameter is assigned to the `sides` field in line 20. Line 21 calls the `roll` method, which simulates the rolling of the die.

Lines 29–36: This is the `roll` method, which simulates the rolling of the die. In line 32 a `Random` object is created, and it is referenced by the `rand` variable. Line 35 uses the `Random` object to get a random number that is in the appropriate range for this particular die. For example, if the `sides` field is set to 6, the expression `rand.nextInt(sides) + 1` will return a random integer in the range of 1 through 6. The random number is assigned to the `value` field.

Lines 43–46: This is the `getSides` method, an accessor that returns the `sides` field.

Lines 53–56: This is the `getValue` method, an accessor that returns the `value` field.

The program in Code Listing 6-15 demonstrates the class. It creates two instances of the `Die` class: one with six sides, and the other with twelve sides. It then simulates five rolls of the dice.

Code Listing 6-15 (`DiceDemo.java`)

```
/**
   This program simulates the rolling of dice.
*/
```

```

public class DiceDemo
{
   public static void main(String[] args)
   {
      final int DIE1_SIDES = 6;   // Number of sides for die #1
      final int DIE2_SIDES = 12;  // Number of sides for die #2
      final int MAX_ROLLS = 5;    // Number of times to roll

      // Create two instances of the Die class.
      Die die1 = new Die(DIE1_SIDES);
      Die die2 = new Die(DIE2_SIDES);

      // Display the initial state of the dice.
      System.out.println("This simulates the rolling of a " +
                         DIE1_SIDES + " sided die and a " +
                         DIE2_SIDES + " sided die.");

      System.out.println("Initial value of the dice:");
      System.out.println(die1.getValue() + " " + die2.getValue());

      // Roll the dice five times.
      System.out.println("Rolling the dice " + MAX_ROLLS + " times.");

      for (int i = 0; i < MAX_ROLLS; i++)
      {
         // Roll the dice.
         die1.roll();
         die2.roll();

         // Display the values of the dice.
         System.out.println(die1.getValue() + " " + die2.getValue());
      }
   }
}
```

Program Output

```
This simulates the rolling of a 6 sided die and a 12 sided die.
Initial value of the dice:
2 7
Rolling the dice 5 times.
3 5
5 2
2 1
4 1
5 9
```

Let's take a closer look at the program:

Lines 9–11:	These statements declare three constants. `DIE1_SIDES` is the number of sides for the first die (6), `DIE2_SIDES` is the number of sides for the second die (12), and `MAX_ROLLS` is the number of times to roll the die (5).
Lines 14–15:	These statements create two instances of the `Die` class. Notice that `DIE1_SIDES`, which is 6, is passed to the constructor in line 14, and `DIE2_SIDES`, which is 12, is passed to the constructor in line 15. As a result, `die1` will reference a `Die` object with six sides, and `die2` will reference a `Die` object with twelve sides.
Lines 23:	This statement displays the initial value of both `Die` objects. (Recall that the `Die` class constructor performs an initial roll of the die.)
Lines 28–36:	This `for` loop iterates five times. Each time the loop iterates, line 31 calls the `die1` object's `roll` method, and line 32 calls the `die2` object's `roll` method. Line 35 displays the value of both dice.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

6.18 How is a constructor named?

6.19 What is a constructor's return type?

6.20 Assume that the following is a constructor, which appears in a class:

```
ClassAct(int number)
{
    item = number;
}
```

a) What is the name of the class that this constructor appears in?

b) Write a statement that creates an object from the class and passes the value 25 as an argument to the constructor.

6.5 Passing Objects as Arguments

CONCEPT: When an object is passed as an argument to a method, the object's address is passed into the method's parameter variable. As a result, the parameter references the object.

When you are developing applications that work with objects, you will often need to write methods that accept objects as arguments. For example, suppose that a program is using the `Die` class that was previously shown in Code Listing 6-14. The following code shows a method named `showDieSides` that accepts a `Die` object as an argument:

```
void showDieSides(Die d)
{
    System.out.println("This die has " + d.getSides() +
                       " sides.");
}
```

The following code sample shows how we might create a `Die` object, and then pass it as an argument to the `showDieSides` method:

```
Die myDie = new Die(6);
showDieSides(myDie)
```

When you pass an object as an argument, the thing that is passed into the parameter variable is the object's memory address. As a result, the parameter variable references the object, and the method has access to the object.

The program shown in Code Listing 6-16 gives a complete demonstration. It creates two `Die` objects: one with six sides, and the other with twenty sides. It passes each object to a method named `rollDie` that displays the die's sides, rolls the die, and displays the die's value.

Code Listing 6-16 **(DieArgument.java)**

```
/**
   This program rolls a 6-sided die and
   a 20-sided die.
*/

public class DieArgument
{
   public static void main(String[] args)
   {
      final int SIX_SIDES = 6;
      final int TWENTY_SIDES = 20;

      // Create a 6-sided die.
      Die sixDie = new Die(SIX_SIDES);

      // Create a 20-sided die.
      Die twentyDie = new Die(TWENTY_SIDES);

      // Roll the dice.
      rollDie(sixDie);
      rollDie(twentyDie);
   }

   /**
      This method simulates a die roll, displaying
      the die's number of sides and value.
      @param d The Die object to roll.
   */

   public static void rollDie(Die d)
   {
      // Display the number of sides.
```

```
         System.out.println("Rolling a " + d.getSides() +
                            " sided die.");

         // Roll the die.
         d.roll();

         // Display the die's value.
         System.out.println("The die's value: " + d.getValue());
      }
   }
```

Program Output

```
Rolling a 6 sided die.
The die's value: 3
Rolling a 20 sided die.
The die's value: 19
```

In the Spotlight:

Simulating the Game of Cho-Han

Cho-Han is a traditional Japanese gambling game in which a dealer uses a cup to roll two six-sided dice. The cup is placed upside down on a table so that the value of the dice is concealed. Players then wager on whether the sum of the dice values is even (Cho) or odd (Han). The winner or winners take all of the wagers, or the house takes them if there are no winners.

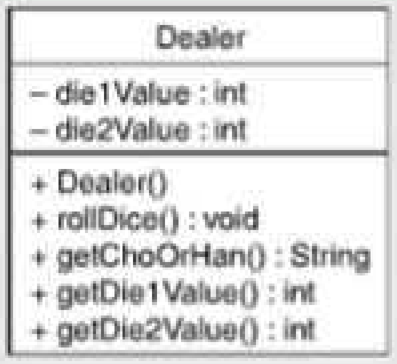
We will develop a program that simulates a simplified variation of the game. The simulated game will have a dealer and two players. The players will not wager money, but will simply guess whether the sum of the dice values is even (Cho) or odd (Han). One point will be awarded to the player, or players, correctly guessing the outcome. The game will play for five rounds, and the player with the most points is the grand winner.

In the program, we will use the `Die` class that was introduced in Code Listing 6-14. We will create two instances of the class to represent two six-sided dice. In addition to the `Die` class, we will write the following classes:

- `Dealer` class: We will create an instance of this class to represent the dealer. It will have the ability to roll the dice, report the value of the dice, and report whether the total dice value is Cho or Han.
- `Player` class: We will create two instances of this class to represent the players. Instances of the `Player` class can store the player's name, make a guess between Cho and Han, and be awarded points.

First, let's look at the `Dealer` class. Figure 6-25 shows a UML diagram for the class, and Code Listing 6-17 shows the code.

Figure 6-25 UML diagram for the `Dealer` class



Code Listing 6-17 **(Dealer.java)**

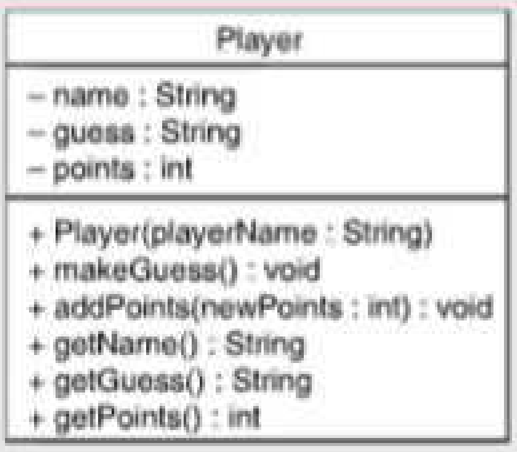
```
/**
   Dealer class for the game of Cho-Han
*/

public class Dealer
{
   private int die1Value;  // The value of die #1
   private int die2Value;  // The value of die #2

   /**
      Constructor
   */

   public Dealer()
   {
      die1Value = 0;
      die2Value = 0;
   }

   /**
      The rollDice method rolls the dice and saves
      their values.
   */

   public void rollDice()
   {
      final int SIDES = 6; // Number of sides for the dice

      // Create the two dice. (This also rolls them.)
      Die die1 = new Die(SIDES);
      Die die2 = new Die(SIDES);

      // Record their values.
      die1Value = die1.getValue();
      die2Value = die2.getValue();
   }

   /**
      The getChoOrHan method returns the result of
      the dice roll, Cho or Han.
      @return Either "Cho (even)" or "Han (odd)"
   */

   public String getChoOrHan()
   {
      String result; // To hold the result

      // Get the sum of the dice.
      int sum = die1Value + die2Value;

      // Determine even or odd.
      if (sum % 2 == 0)
         result = "Cho (even)";
      else
         result = "Han (odd)";

      // Return the result.
      return result;
   }

   /**
      The getDie1Value method returns the value of
      die #1.
      @return The die1Value field
   */

   public int getDie1Value()
   {
      return die1Value;
   }

   /**
      The getDie2Value method returns the value of
      die #2.
      @return The die2Value field
   */

   public int getDie2Value()
   {
      return die2Value;
   }
}
```

Let's take a closer look at the code for the `Dealer` class:

- Lines 7 and 8 declare the fields `die1Value` and `die2Value`. These fields will hold the value of the two dice after they have been rolled.
- The constructor, in lines 14 through 18, initializes the `die1Value` and `die2Value` fields to 0.
- The `rollDice` method, in lines 25 through 36, simulates the rolling of the dice. Lines 30 and 31 create two `Die` objects. Recall that the `Die` class constructor performs an initial roll of the die, so there is no need to call the `Die` objects' `roll` method. Lines 34 and 35 save the value of the dice in the `die1Value` and `die2Value` fields.
- The `getChoOrHan` method, in lines 44 through 59, returns a string indicating whether the sum of the dice is Cho (even) or Han (odd).
- The `getDie1Value` method, in lines 67 through 70, returns the value of the first die (stored in the `die1Value` field).
- The `getDie2Value` method, in lines 78 through 81, returns the value of the second die (stored in the `die2Value` field).

Now let's look at the `Player` class. Figure 6-26 shows a UML diagram for the class, and Code Listing 6-18 shows the code.

Figure 6-26 UML diagram for the `Player` class



Code Listing 6-18 `(Player.java)`

```
import java.util.Random;

/**
   Player class for the game of Cho-Han
*/

public class Player
{
   private String name;     // The player's name
   private String guess;    // The player's guess
   private int points;      // The player's points

   /**
      Constructor
      @param playerName The player's name.
   */

   public Player(String playerName)
   {
      name = playerName;
      guess = "";
      points = 0;
   }

   /**
      The makeGuess method causes the player to guess
      either "Cho (even)" or "Han (odd)".
   */

   public void makeGuess()
   {
      // Create a Random object.
      Random rand = new Random();

      // Get a random number, either 0 or 1.
      int guessNumber = rand.nextInt(2);

      // Convert the random number to a guess of
      // either "Cho (even)" or "Han (odd)".
      if (guessNumber == 0)
         guess = "Cho (even)";
      else
         guess = "Han (odd)";
   }

   /**
      The addPoints method adds a specified number of
      points to the player's current balance.
      @newPoints The points to add.
   */

   public void addPoints(int newPoints)
   {
      points += newPoints;
   }

   /**
      The getName method returns the player's name.
      @return The value of the name field.
   */

```

```
62     public String getName()
63     {
64        return name;
65     }
66
67     /**
68        The getGuess method returns the player's guess.
69        @return The value of the guess field.
70     */
71
72     public String getGuess()
73     {
74        return guess;
75     }
76
77     /**
78        The getPoints method returns the player's points
79        @return The value of the points field.
80     */
81
82     public int getPoints()
83     {
84        return points;
85     }
86  }
```

Here's a summary of the code for the `Player` class:

- Lines 9 through 11 declare the fields `name`, `guess`, and `points`. These fields will hold the player's name, the player's guess, and the number of points the player has earned.
- The constructor, in lines 18 through 23, accepts an argument for the player's name, which is assigned to the `name` field. The `guess` field is assigned an empty string, and the `points` field is set to 0.
- The `makeGuess` method, in lines 30 through 44, causes the player to make a guess. The method generates a random number that is either a 0 or a 1. The `if` statement that begins at line 40 assigns the string "Cho (even)" to the `guess` field if the random number is 0, or it assigns the string "Han (odd)" to the `guess` field if the random number is 1.
- The `addPoints` method, in lines 52 through 55, adds the number of points specified by the argument to the player's `point` field.
- The `getName` method, in lines 62 through 65, returns the player's name.
- The `getGuess` method, in lines 72 through 75, returns the player's guess.
- The `getPoints` method, in lines 82 through 85, returns the player's points.

Code Listing 6-19 shows the program that uses these classes to simulate the game. The `main` method simulates five rounds of the game, displaying the results of each round, and then displays the overall game results.

Code Listing 6-19 (ChoHan.java)

```
import java.util.Scanner;

public class ChoHan
{
   public static void main(String[] args)
   {
      final int MAX_ROUNDS = 5;  // Number of rounds
      String player1Name;        // First player's name
      String player2Name;        // Second player's name

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the players' names.
      System.out.print("Enter the first player's name: ");
      player1Name = keyboard.nextLine();
      System.out.print("Enter the second player's name: ");
      player2Name = keyboard.nextLine();

      // Create the dealer.
      Dealer dealer = new Dealer();

      // Create the two players.
      Player player1 = new Player(player1Name);
      Player player2 = new Player(player2Name);

      // Play the rounds.
      for (int round = 0; round < MAX_ROUNDS; round++)
      {
         System.out.println("----------------------------");
         System.out.printf("Now playing round %d.\n", round + 1);

         // Roll the dice.
         dealer.rollDice();

         // The players make their guesses.
         player1.makeGuess();
         player2.makeGuess();

         // Determine the winner of this round.
         roundResults(dealer, player1, player2);
      }

      // Display the grand winner.
      displayGrandWinner(player1, player2);
   }

   /**
      The roundResults method determines the results of
      the current round.
      @param dealer The Dealer object
      @param player1 Player #1 object
      @param player2 Player #2 object
   */

   public static void roundResults(Dealer dealer, Player player1,
                                   Player player2)
   {
      // Show the dice values.
      System.out.printf("The dealer rolled %d and %d.\n",
                        dealer.getDie1Value(), dealer.getDie2Value());
      System.out.printf("Result: %s\n", dealer.getChoOrHan());

      // Check each player's guess and award points.
      checkGuess(player1, dealer);
      checkGuess(player2, dealer);
   }

   /**
      The checkGuess method checks a player's guess against
      the dealer's result.
      @param player The Player object to check.
      @param dealer The Dealer object.
   */

   public static void checkGuess(Player player, Dealer dealer)
   {
      final int POINTS_TO_ADD = 1; // Points to award winner
      String guess = player.getGuess();           // Player's guess
      String choHanResult = dealer.getChoOrHan(); // Cho or Han

      // Display the player's guess.
      System.out.printf("The player %s guessed %s.\n",
                        player.getName(), player.getGuess());

      // Award points if the player guessed correctly.
      if (guess.equalsIgnoreCase(choHanResult))
      {
         player.addPoints(POINTS_TO_ADD);
         System.out.printf("Awarding %d point(s) to %s.\n",
                           POINTS_TO_ADD, player.getName());
      }
   }

   /**
      The displayGrandWinner method displays the game's grand winner.
      @param player1 Player #1
      @param player2 Player #2
   */

   public static void displayGrandWinner(Player player1, Player player2)
   {
      System.out.println("----------------------------");
      System.out.println("Game over. Here are the results:");
      System.out.printf("%s: %d points.\n", player1.getName(),
                        player1.getPoints());
      System.out.printf("%s: %d points.\n", player2.getName(),
                        player2.getPoints());

      if (player1.getPoints() > player2.getPoints())
         System.out.println(player1.getName() + " is the grand winner!");
      else if (player2.getPoints() > player1.getPoints())
         System.out.println(player2.getName() + " is the grand winner!");
      else
         System.out.println("Both players are tied!");
   }
}
```

Program Output with Example Input Shown in Bold

```
Enter the first player's name: Chelsea [Enter]
Enter the second player's name: Chris [Enter]
----------------------------
Now playing round 1.
The dealer rolled 3 and 6.
Result: Han (odd)
The player Chelsea guessed Han (odd).
Awarding 1 point(s) to Chelsea.
The player Chris guessed Han (odd).
Awarding 1 point(s) to Chris.
----------------------------
Now playing round 2.
The dealer rolled 4 and 5.
Result: Han (odd)
The player Chelsea guessed Cho (even).
The player Chris guessed Cho (even).
----------------------------
Now playing round 3.
The dealer rolled 5 and 6.
Result: Han (odd)
The player Chelsea guessed Cho (even).
The player Chris guessed Han (odd).
Awarding 1 point(s) to Chris.
```

```
----------------------------
Now playing round 4.
The dealer rolled 1 and 6.
Result: Han (odd)
The player Chelsea guessed Cho (even).
The player Chris guessed Cho (even).
----------------------------
Now playing round 5.
The dealer rolled 6 and 6.
Result: Cho (even)
The player Chelsea guessed Han (odd).
The player Chris guessed Cho (even).
Awarding 1 point(s) to Chris.
----------------------------
Game over. Here are the results:
Chelsea: 1 points.
Chris: 3 points.
Chris is the grand winner!
```

Let's look at the code. Here is a summary of the `main` method:

- Lines 7 through 9 make the following declarations: `MAX_ROUNDS`—the number of rounds to play, `player1Name`—to hold the name of player #1, and `player2Name`—to hold the name of player #2.
- Lines 15 through 18 prompt the user to enter the players' names.
- Line 21 creates an instance of the `Dealer` class. The object represents the dealer, and is referenced by the `dealer` variable.
- Line 24 creates an instance of the `Player` class. The object represents player #1, and is referenced by the `player1` variable. Notice that `player1Name` is passed as an argument to the constructor.
- Line 25 creates another instance of the `Player` class. The object represents player #2, and is referenced by the `player2` variable. Notice that `player2Name` is passed as an argument to the constructor.
- The `for` loop that begins in line 28 iterates five times, causing the simulation of five rounds of the game. The loop performs the following actions:
  - Line 34 causes the dealer to roll the dice.
  - Line 37 causes player #1 to make a guess (Cho or Han).
  - Line 38 causes player #2 to make a guess (Cho or Han).
  - Line 41 passes the `dealer`, `player1`, and `player2` objects to the `roundResults` method. The method displays the results of this round.
- Line 45 passes the `player1` and `player2` objects to the `displayGrandWinner` method, which displays the grand winner of the game.

The `roundResults` method, which displays the results of a round, appears in lines 56 through 67. Here is a summary of the method:

- The method accepts references to the `dealer`, `player1`, and `player2` objects as arguments.
- The statement in lines 60 and 61 displays the value of the two dice.

- Line 62 calls the `dealer` object's `getChoOrHan` method to display the results, Cho or Han.
- Line 65 calls the `checkGuess` method, passing the `player1` and `dealer` objects as arguments. The `checkGuess` method compares a player's guess to the dealer's result (Cho or Han), and awards points to the player, if the guess is correct.
- Line 66 calls the `checkGuess` method, passing the `player2` and `dealer` objects as arguments.

The `checkGuess` method, which compares a player's guess to the dealer's result, awarding points to the player for a correct guess, appears in lines 76 through 93. Here is a summary of the method:

- The method accepts references to a `Player` object and the `Dealer` object as arguments.
- Line 78 declares the constant `POINTS_TO_ADD`, set to the value 1, which is the number of points to add to the player's balance if the player's guess is correct.
- Line 79 assigns the player's guess to the `String` object `guess`.
- Line 80 assigns the dealer's results (Cho or Han) to the `String` object `choHanResult`.
- The statement in lines 83 and 84 displays the player's name and guess.
- The `if` statement in line 87 compares the player's guess to the dealer's result. If they match, then the player guessed correctly, and line 89 awards points to the player.

The `displayGrandWinner` method, which displays the grand winner of the game, appears in lines 101 through 116. Here is a summary of the method:

- The method accepts references to the `player1` and `player2` objects.
- The statements in lines 105 through 108 display both players' names and points.
- The `if-else-if` statement that begins in line 110 determines which of the two players has the highest score, and displays that player's name as the grand winner. If both players have the same score, a tie is declared.

6.6 Overloading Methods and Constructors

CONCEPT: Two or more methods in a class may have the same name as long as their parameter lists are different. This also applies to constructors.

Method overloading is an important part of object-oriented programming. When a method is *overloaded*, it means that multiple methods in the same class have the same name, but use different types of parameters. Method overloading is important because sometimes you need several different ways to perform the same operation. For example, suppose a class has the following two methods:

```
public int add(int num1, int num2)
{
   int sum = num1 + num2;
   return sum;
}
```

```
public String add(String str1, String str2)
{
   String combined = str1 + str2;
   return combined;
}
```

Both of these methods are named add. They both take two arguments, which are added together. The first one accepts two int arguments and returns their sum. The second accepts two String references and returns a reference to a String that is a concatenation of the two arguments. When we write a call to the add method, the compiler must determine which one of the overloaded methods we intended to call.

The process of matching a method call with the correct method is known as *binding*. When an overloaded method is being called, Java uses the method's name and parameter list to determine which method to bind the call to. If two int arguments are passed to the add method, the version of the method with two int parameters is called. Likewise, when two String arguments are passed to add, the version with two String parameters is called.

Java uses a method's signature to distinguish it from other methods of the same name. A method's *signature* consists of the method's name and the data types of the method's parameters, in the order that they appear. For example, here are the signatures of the add methods that were previously shown:

```
add(int, int)
add(String, String)
```

Note that the method's return type is *not* part of the signature. For this reason, the following add method cannot be added to the same class with the previous ones:

```
public int add(String str1, String str2)
{
   int sum = Integer.parstInt(str1) + Integer.parseInt(str2);
   return sum;
}
```

Because the return type is not part of the signature, this method's signature is the same as that of the other add method that takes two String arguments. For this reason, an error message will be issued when a class containing all of these methods is compiled.

Constructors can also be overloaded, which means that a class can have more than one constructor. The rules for overloading constructors are the same for overloading other methods: Each version of the constructor must have a different parameter list. As long as each constructor has a unique signature, the compiler can tell them apart. For example, the Rectangle class that we discussed earlier could have the following two constructors:

```
public Rectangle()
{
   length = 0.0;
   width = 0.0;
}
```

```
public Rectangle(double len, double w)
{
   length = len;
   width = w;
}
```

The first constructor shown here accepts no arguments, and assigns 0.0 to the `length` and `width` fields. The second constructor accepts two arguments, which are assigned to the `length` and `width` fields. The following code shows an example of how each constructor is called:

```
Rectangle box1 = new Rectangle();
Rectangle box2 = new Rectangle(5.0, 10.0);
```

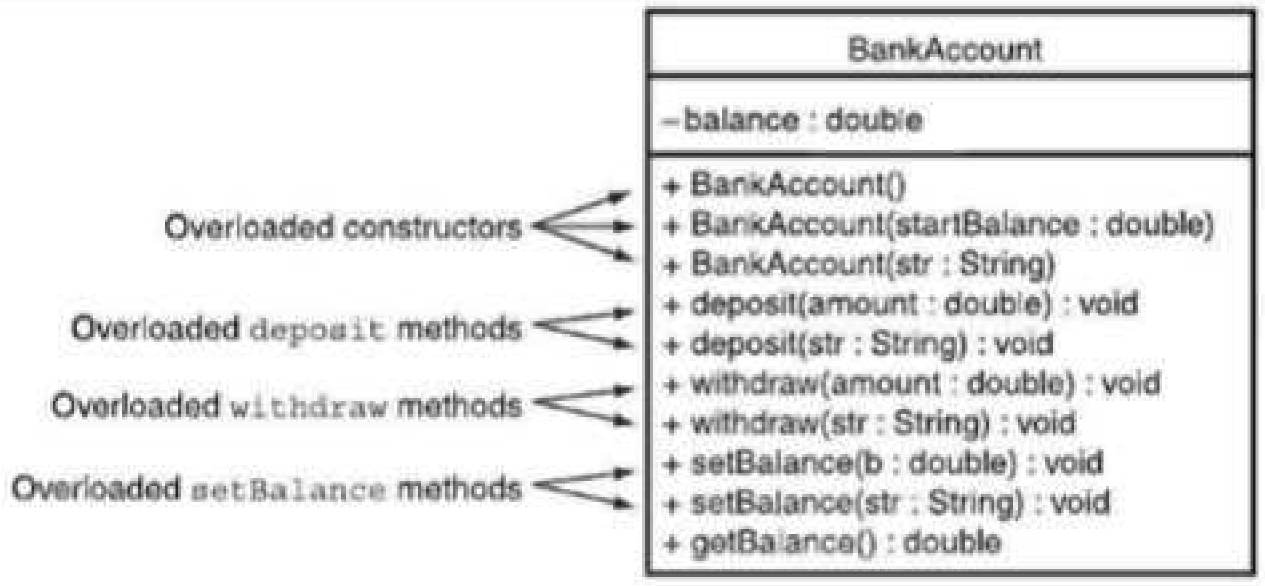
The first statement creates a `Rectangle` object, referenced by the `box1` variable, and executes the no-arg constructor. Its `length` and `width` fields will be set to 0.0. The second statement creates another `Rectangle` object, referenced by the `box2` variable, and executes the second constructor. Its `length` and `width` fields will be set to 5.0 and 10.0, respectively.

Recall that Java provides a default constructor only when you do not write any constructors for a class. If a class has a constructor that accepts arguments, but it does not have a no-arg constructor, you cannot create an instance of the class without passing arguments to the constructor. Therefore, any time you write a constructor for a class, and that constructor accepts arguments, you should also write a no-arg constructor if you want to be able to create instances of the class without passing arguments to the constructor.

The `BankAccount` Class

Now we will look at the `BankAccount` class. Objects that are created from this class will simulate bank accounts, allowing us to have a starting balance, make deposits, make withdrawals, and get the current balance. A UML diagram for the `BankAccount` class is shown in Figure 6-27. In the figure, the overloaded constructors and overloaded methods are pointed out. Note that the extra annotation is not part of the UML diagram. It is there to draw attention to the items that are overloaded.

Figure 6-27 UML diagram for the `BankAccount` class


BankAccount
−balance : double
+ BankAccount()
+ BankAccount(startBalance : double)
+ BankAccount(str : String)
+ deposit(amount : double) : void
+ deposit(str : String) : void
+ withdraw(amount : double) : void
+ withdraw(str : String) : void
+ setBalance(b : double) : void
+ setBalance(str : String) : void
+ getBalance() : double
Overloaded constructors
Overloaded deposit methods
Overloaded withdraw methods
Overloaded setBalance methods

As you can see from the diagram, the class has three overloaded constructors. Also, the class has two overloaded methods named `deposit`, two overloaded methods named `withdraw`, and two overloaded methods named `setBalance`. The last method, `getBalance`, is not overloaded. Code Listing 6-20 shows the code for the class.

Code Listing 6-20 (BankAccount.java)

```java
/**
   The BankAccount class simulates a bank account.
*/

public class BankAccount
{
   private double balance;      // Account balance

   /**
      This constructor sets the starting balance
      at 0.0.
   */

   public BankAccount()
   {
      balance = 0.0;
   }

   /**
      This constructor sets the starting balance
      to the value passed as an argument.
      @param startBalance The starting balance.
   */

   public BankAccount(double startBalance)
   {
      balance = startBalance;
   }

   /**
      This constructor sets the starting balance
      to the value in the String argument.
      @param str The starting balance, as a String.
   */

   public BankAccount(String str)
   {
      balance = Double.parseDouble(str);
   }

   /**
      The deposit method makes a deposit into
      the account.
      @param amount The amount to add to the
                    balance field.
   */

   public void deposit(double amount)
   {
      balance += amount;
   }

   /**
      The deposit method makes a deposit into
      the account.
      @param str The amount to add to the
                 balance field, as a String.
   */

   public void deposit(String str)
   {
      balance += Double.parseDouble(str);
   }

   /**
      The withdraw method withdraws an amount
      from the account.
      @param amount The amount to subtract from
                    the balance field.
   */

   public void withdraw(double amount)
   {
      balance -= amount;
   }

   /**
      The withdraw method withdraws an amount
      from the account.
      @param str The amount to subtract from
                 the balance field, as a String.
   */

   public void withdraw(String str)
   {
      balance -= Double.parseDouble(str);
   }

   /**
      The setBalance method sets the account balance.
      @param b The value to store in the balance field.
   */

   public void setBalance(double b)
   {
      balance = b;
   }

   /**
      The setBalance method sets the account balance.
      @param str The value, as a String, to store in
                 the balance field.
   */

   public void setBalance(String str)
   {
      balance = Double.parseDouble(str);
    }

   /**
      The getBalance method returns the
      account balance.
      @return The value in the balance field.
   */

   public double getBalance()
   {
      return balance;
   }
}
```

The class has one field, `balance`, which is a `double`. This field holds an account's current balance. Here is a summary of the class's overloaded constructors:

- The first constructor is a no-arg constructor. It sets the `balance` field to 0.0. If we wish to execute this constructor when we create an instance of the class, we simply pass no constructor arguments. Here is an example:

  ```
  BankAccount account = new BankAccount();
  ```

- The second constructor has a `double` parameter variable, `startBalance`, which is assigned to the `balance` field. If we wish to execute this constructor when we create an instance of the class, we pass a `double` value as a constructor argument. Here is an example:

  ```
  BankAccount account = new BankAccount(1000.0);
  ```