```
      double sum = 0.0; // Accumulator, initialized to 0

      // Open the file for reading.
      File file = new File("Numbers.txt");
      Scanner inputFile = new Scanner(file);

      // Read all of the values from the file
      // and calculate their total.
      while (inputFile.hasNext())
      {
         // Read a value from the file.
         double number = inputFile.nextDouble();

         // Add the number to sum.
         sum = sum + number;
      }

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

      // Display the sum of the numbers.
      System.out.println("The sum of the numbers in " +
                         "Numbers.txt is " + sum);
   }
}
```

Program Output

```
The sum of the numbers in Numbers.txt is 41.4
```

Review

Let's quickly review the steps necessary when writing a program that reads data from a file:

1. You will need the import `java.util.Scanner;` statement in the top section of your program, so you can use the `Scanner` class. You will also need the `import java.io.*;` statement in the top section of your program. This is required by the `File` class.
2. Because we have not yet learned how to respond to exceptions, any method that uses a `Scanner` object to open a file must have a `throws IOException` clause in its header.
3. You create a `File` object and pass the name of the file as a string to the constructor.
4. You create a `Scanner` object and pass a reference to the `File` object as an argument to the constructor.
5. You use the `Scanner` class's `nextLine` method to read a line from the file. The method returns the line of data as a string. To read primitive values, use methods such as `nextInt`, `nextDouble`, and so forth.

6. Call the `Scanner` class's `hasNext` method to determine whether there is more data to read from the file. If the method returns `true`, then there is more data to read. If the method returns `false`, you have reached the end of the file.
7. When finished writing to the file, you use the `Scanner` class's `close` method to close the file.

Checking for a File's Existence

It's usually a good idea to make sure that a file exists before you try to open it for input. If you attempt to open a file for input, and the file does not exist, the program will throw an exception and halt. For example, the program you saw in Code Listing 4-20 will throw an exception at line 17 if the file *Numbers.txt* does not exist. Here is an example of the error message that will be displayed when this happens:

```
Exception in thread "main" java.io.FileNotFoundException: Numbers.txt (The
system cannot find the file specified)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:106)
        at java.util.Scanner.<init>(Scanner.java:636)
        at FileSum.main(FileSum.java:17)
```

Rather than allowing the exception to be thrown and permitting this cryptic error message to be displayed, your program can check for the file's existence before it attempts to open the file. If the file does not exist, the program can display a more user-friendly error message and gracefully shut down.

After you create a `File` object representing the file that you want to open, you can use the `File` class's `exists` method to determine whether the file exists. The method returns `true` if the file exists, or `false` if the file does not exist. Code Listing 4-21 shows how to use the method. This is a modification of the *FileSum* program in Code Listing 4-20. This version of the program checks for the existence of the file *Numbers.txt* before it attempts to open it.

Code Listing 4-21 (`FileSum2.java`)

```
 1  import java.util.Scanner;
 2  import java.io.*;
 3
 4  /**
 5     This version of the program confirms that the
 6     Numbers.txt file exists before opening it.
 7  */
 8
 9  public class FileSum2
10  {
11     public static void main(String[] args) throws IOException
12     {
13        double sum = 0.0;     // Accumulator, initialized to 0
14
15          // Make sure the file exists.
16          File file = new File("Numbers.txt");
17          if (!file.exists())
18          {
19             System.out.println("The file Numbers.txt is not found.");
20             System.exit(0);
21          }
22
23          // Open the file for reading.
24          Scanner inputFile = new Scanner(file);
25
26          // Read all of the values from the file
27          // and calculate their total.
28          while (inputFile.hasNext())
29          {
30             // Read a value from the file.
31             double number = inputFile.nextDouble();
32
33             // Add the number to sum.
34             sum = sum + number;
35          }
36
37          // Close the file.
38          inputFile.close();
39
40          // Display the sum of the numbers.
41          System.out.println("The sum of the numbers in " +
42                             "Numbers.txt is " + sum);
43       }
44    }
```

Program Output (Assuming Numbers.txt Does Not Exist)

```
The file Numbers.txt is not found.
```

In line 16 the program creates a `File` object to represent the *Numbers.txt* file. In line 17, the `if` statement calls the `file.exists()` method. Notice the use of the ! operator. If the method returns `false`, indicating that the file does not exist, the code in lines 19 and 20 executes. Line 19 displays an error message, and line 20 calls the `System.exit(0)` method to shut the program down.

The previous example shows you how to make sure that a file exists before trying to open it for input. But, when you are opening a file for output, sometimes you want to make sure the file does *not* exist. When you use a `PrintWriter` object to open a file, the file will be erased if it already exists. If you do not want to erase the existing file, you have to check for its existence before creating the `PrintWriter` object. Code Listing 4-22 shows you how to use the `File` class's `exists` method in this type of situation. This is a modification of the program you saw in Code Listing 4-17.

Code Listing 4-22 **(FileWriteDemo2.java)**

```
import java.util.Scanner;      // Needed for Scanner class
import java.io.*;              // Needed for File and IOException

/**
   This program writes data to a file. It makes sure the
   specified file does not exist before opening it.
*/

public class FileWriteDemo2
{
   public static void main(String[] args) throws IOException
   {
      String filename;           // Filename
      String friendName;         // Friend's name
      int numFriends;            // Number of friends

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

      // Get the number of friends.
      System.out.print("How many friends do you have? ");
      numFriends = keyboard.nextInt();

      // Consume the remaining newline character.
      keyboard.nextLine();

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

      // Make sure the file does not exist.
      File file = new File(filename);
      if (file.exists())
      {
         System.out.println("The file " + filename +
                            " already exists.");
         System.exit(0);
      }

      // Open the file.
      PrintWriter outputFile = new PrintWriter(file);

      // Get data and write it to the file.
      for (int i = 1; i <= numFriends; i++)
      {
         // Get the name of a friend.
47             System.out.print("Enter the name of friend " +
48                              "number " + i + ": ");
49             friendName = keyboard.nextLine();
50
51             // Write the name to the file.
52             outputFile.println(friendName);
53          }
54
55          // Close the file.
56          outputFile.close();
57          System.out.println("Data written to the file.");
58       }
59    }
```

Program Output with Example Input Shown in Bold

```
How many friends do you have? 2 [Enter]
Enter the filename: MyFriends.txt [Enter]
The file MyFriends.txt already exists.
```

Line 32 creates a `File` object representing the file. The `if` statement in line 33 calls the `file.exists()` method. If the method returns `true`, then the file exists. In this case the code in lines 35 through 37 executes. This code displays an error message and shuts the program down. If the file does not exist, the rest of the program executes.

Notice that in line 41 we pass a reference to the `File` object to the `PrintWriter` constructor. In previous programs that created an instance `PrintWriter`, we passed a filename to the constructor. If you have a reference to a `File` object that represents the file you wish to open, as we do in this program, you have the option of passing it to the `PrintWriter` constructor.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

4.16 What is the difference between an input file and an output file?

4.17 What `import` statement will you need in a program that performs file operations?

4.18 What class do you use to write data to a file?

4.19 Write code that does the following: opens a file named *MyName.txt*, writes your first name to the file, and then closes the file.

4.20 What classes do you use to read data from a file?

4.21 Write code that does the following: opens a file named *MyName.txt*, reads the first line from the file and displays it, and then closes the file.

4.22 You are opening an existing file for output. How do you open the file without erasing it, and at the same time make sure that new data that is written to the file is appended to the end of the file's existing data?

4.23 What clause must you write in the header of a method that performs a file operation?

4.11 Generating Random Numbers with the Random Class

CONCEPT: Random numbers are used in a variety of applications. Java provides the `Random` class that you can use to generate random numbers.

Random numbers are useful for lots of different programming tasks. The following are just a few examples.

- Random numbers are commonly used in games. For example, computer games that let the player roll dice use random numbers to represent the values of the dice. Programs that show cards being drawn from a shuffled deck use random numbers to represent the face values of the cards.
- Random numbers are useful in simulation programs. In some simulations, the computer must randomly decide how a person, animal, insect, or other living being will behave. Formulas can be constructed in which a random number is used to determine various actions and events that take place in the program.
- Random numbers are useful in statistical programs that must randomly select data for analysis.
- Random numbers are commonly used in computer security to encrypt sensitive data.

The Java API provides a class named `Random` that you can use to generate random numbers. The class is part of the `java.util` package, so any program that uses it will need an `import` statement such as:

```
import java.util.Random;
```

You create an object from the `Random` class with a statement such as this:

```
Random randomNumbers = new Random();
```

This statement does the following:

- It declares a variable named `randomNumbers`. The data type is the `Random` class.
- The expression `new Random()` creates an instance of the `Random` class.
- The equal sign assigns the address of the `Random` class to the `randomNumbers` variable.

After this statement executes, the `randomNumbers` variable will reference a `Random` object. Once you have created a `Random` object, you can call its `nextInt` method to get a random integer number. The following code shows an example:

```
// Declare an int variable.
int number;

// Create a Random object.
Random randomNumbers = new Random();

// Get a random integer and assign it to number.
number = randomNumbers.nextInt();
```

After this code executes, the `number` variable will contain a random integer. If you call the `nextInt` method with no arguments, as shown in this example, the returned integer is somewhere between -2,147,483,648 and +2,147,483,647. Alternatively, you can pass an argument that specifies an upper limit to the generated number's range. In the following statement, the value assigned to `number` is somewhere between 0 and 99:

```
number = randomNumbers.nextInt(100);
```

You can add or subtract a value to shift the numeric range upward or downward. In the following statement, we call the `nextInt` method to get a random number in the range of 0 through 9, and then we add 1 to it. So, the number assigned to `number` will be somewhere in the range of 1 through 10:

```
number = randomNumbers.nextInt(10) + 1;
```

The following statement shows another example. It assigns a random integer to `number` between -50 and +49:

```
number = randomNumbers.nextInt(100) - 50
```

The `Random` class has other methods for generating random numbers, and Table 4-1 summarizes several of them.

Table 4-1 Some of the `Random` class's methods

Method	Description
`nextDouble()`	Returns the next random number as a `double`. The number will be within the range of `0.0` through `1.0`.
`nextFloat()`	Returns the next random number as a `float`. The number will be within the range of `0.0` through `1.0`.
`nextInt()`	Returns the next random number as an `int`. The number will be within the range of an `int`, which is -2,147,483,648 to +2,147,483,648.
`nextInt(int n)`	This method accepts an integer argument, *n*. It returns a random number as an `int`. The number will be within the range of 0 through *n*.
`nextLong()`	Returns the next random number as a `long`. The number will be within the range of a `long`, which is -9,223,372,036,854,775,808 to +9,223,372,036,854,775,808.

The program in Code Listing 4-23 demonstrates using the `Random` class.

Code Listing 4-23 **(MathTutor.java)**

```
import java.util.Scanner;   // Needed for the Scanner class
import java.util.Random;    // Needed for the Random class

/**
   This program demonstrates the Random class.
*/

public class MathTutor
```

```
{
   public static void main(String[] args)
   {
      int number1;          // A number
      int number2;          // Another number
      int sum;              // The sum of the numbers
      int userAnswer;       // The user's answer

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

      // Create a Random class object.
      Random randomNumbers = new Random();

      // Get two random numbers.
      number1 = randomNumbers.nextInt(100);
      number2 = randomNumbers.nextInt(100);

      // Display an addition problem.
      System.out.println("What is the answer to the " +
                         "following problem?");
      System.out.print(number1 + " + " +
                       number2 + " = ? ");

      // Calculate the answer.
      sum = number1 + number2;

      // Get the user's answer.
      userAnswer = keyboard.nextInt();

      // Display the user's results.
      if (userAnswer == sum)
         System.out.println("Correct!");
      else
      {
         System.out.println("Sorry, wrong answer. " +
                            "The correct answer is " +
                            sum);
      }
   }
}
```

Program Output with Example Input Shown in Bold

```
What is the answer to the following problem?
52 + 19 = ? 71 [Enter]
Correct!
```

Program Output with Example Input Shown in Bold

```
What is the answer to the following problem?
27 + 73 = ? 101 [Enter]
Sorry, wrong answer. The correct answer is 100
```

In the Spotlight:

Using Random Numbers

Dr. Kimura teaches an introductory statistics class, and has asked you to write a program that he can use in class to simulate the rolling of dice. The program should randomly generate two numbers in the range of 1 through 6 and display them. In your interview with Dr. Kimura, you learn that he would like to use the program to simulate several rolls of the dice, one after the other. Here is the pseudocode for the program:

```
While the user wants to roll the dice:
    Display a random number in the range of 1 through 6
    Display another random number in the range of 1 through 6
    Ask the user if he or she wants to roll the dice again
```

You will write a `while` loop that simulates one roll of the dice, and then asks the user whether another roll should be performed. As long as the user answers "y" for yes, the loop will repeat. Code Listing 4-24 shows the program.

Code Listing 4-24 (`RollDice.java`)

```java
import java.util.Scanner;
import java.util.Random;

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

public class RollDice
{
   public static void main(String[] args)
   {
      String again = "y";    // To control the loop
      int die1;              // To hold the value of die #1
      int die2;              // to hold the value of die #2

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

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

      // Simulate rolling the dice.
      while (again.equalsIgnoreCase("y"))
      {
         System.out.println("Rolling the dice ...");
         die1 = rand.nextInt(6) + 1;
         die2 = rand.nextInt(6) + 1;
         System.out.println("Their values are:");
         System.out.println(die1 + " " + die2);

         System.out.print("Roll them again (y = yes)? ");
         again = keyboard.nextLine();
      }
   }
}
```

Program Output with Example Input Shown in Bold

```
Rolling the dice ...
Their values are:
4 3
Roll them again (y = yes)? y [Enter]
Rolling the dice ...
Their values are:
2 6
Roll them again (y = yes)? y [Enter]
Rolling the dice ...
Their values are:
1 5
Roll them again (y = yes)? n [Enter]
```

In the Spotlight:

Using Random Numbers to Represent Other Values

Dr. Kimura was so happy with the dice rolling simulator that you wrote for him, he has asked you to write one more program. He would like a program that he can use to simulate ten coin tosses, one after the other. Each time the program simulates a coin toss, it should randomly display either "Heads" or "Tails".

You decide that you can simulate the tossing of a coin by randomly generating a number in the range of 0 through 1. You will write an `if` statement that displays "Tails" if the random number is 0, or "Heads" otherwise. Here is the pseudocode:

```
Repeat 10 times:
   If a random number in the range of 0 through 1 equals 0, then:
      Display "Tails"
   Else:
      Display "Heads"
```

Because the program should simulate 10 tosses of a coin, you decide to use a `for` loop. The program is shown in Code Listing 4-25.

Code Listing 4-25 (CoinToss.java)

```
import java.util.Random;

/**
   This program simulates 10 tosses of a coin.
*/

public class CoinToss
{
   public static void main(String[] args)
   {
      // Create a Random object to generate random numbers.
      Random rand = new Random();

      // Simulate the coin tosses.
      for (int count = 0; count < 10; count++)
      {
         if (rand.nextInt(2) == 0)
            System.out.println("Tails");
         else
            System.out.println("Heads");
      }
   }
}
```

Program Output

```
Tails
Tails
Heads
Tails
Heads
Heads
Heads
Tails
Heads
Tails
```

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

4.24 Assume `x` is an `int` variable, and `rand` references a `Random` object. What does the following statement do?

```
x = rand.nextInt();
```

4.25 Assume `x` is an `int` variable, and `rand` references a `Random` object. What does the following statement do?

```
x = rand.nextInt(100);
```

4.26 Assume `x` is an `int` variable, and `rand` references a `Random` object. What does the following statement do?

```
x = rand.nextInt(9) + 1;
```

4.27 Assume `x` is a `double` variable, and `rand` references a `Random` object. What does the following statement do?

```
x = rand.nextDouble();
```

4.12 Common Errors to Avoid

The following list describes several errors that are commonly committed when learning this chapter's topics.

- **Using the increment or decrement operator in the wrong mode.** When the increment or decrement operator is placed in front of (to the left of) its operand, it is used in prefix mode. When either of these operators is placed behind (to the right of) its operand, it is used in postfix mode.
- **Forgetting to enclose the `boolean` expression in a `while` loop or a `do-while` loop inside parentheses.**
- **Placing a semicolon at the end of a `while` or `for` loop's header.** When you write a semicolon at the end of a `while` or `for` loop's header, Java assumes that the conditionally executed statement is a `null` or empty statement. This usually results in an infinite loop.
- **Forgetting to write the semicolon at the end of the `do-while` loop.** The `do-while` loop must be terminated with a semicolon.
- **Forgetting to enclose multiple statements in the body of a loop in braces.** Normally a loop conditionally executes only one statement. To conditionally execute more than one statement, you must place the statements in braces.
- **Using commas instead of semicolons to separate the initialization, test, and update expressions in a `for` loop.**
- **Forgetting to write code in the body of a `while` or `do-while` loop that modifies the loop control variable.** If a `while` or `do-while` loop's `boolean` expression never becomes `false`, the loop will repeat indefinitely. You must have code in the body of the loop that modifies the loop control variable so that the `boolean` expression will at some point become `false`.
- **Using a sentinel value that can also be a valid data value.** Remember, a sentinel is a special value that cannot be mistaken as a member of a list of data items and signals that there are no more data items from the list to be processed. If you choose as a sentinel a value that might also appear in the list, the loop will prematurely terminate if it encounters the value in the list.
- **Forgetting to initialize an accumulator to zero.** In order for an accumulator to keep a correct running total, it must be initialized to zero before any values are added to it.

Review Questions and Exercises

Multiple Choice and True/False

1. What will the `println` statement in the following program segment display?

   ```
   int x = 5;
   System.out.println(x++);
   ```

   a. 5
   b. 6
   c. 0
   d. None of these

2. What will the `println` statement in the following program segment display?

   ```
   int x = 5;
   System.out.println(++x);
   ```

   a. 5
   b. 6
   c. 0
   d. None of these

3. In the expression `number++`, the `++` operator is in what mode?
   a. prefix
   b. pretest
   c. postfix
   d. posttest

4. What is each repetition of a loop known as?
   a. cycle
   b. revolution
   c. orbit
   d. iteration

5. This is a variable that controls the number of iterations performed by a loop.
   a. loop control variable
   b. accumulator
   c. iteration register variable
   d. repetition meter

6. The `while` loop is this type of loop.
   a. pretest
   b. posttest
   c. prefix
   d. postfix

7. The `do-while` loop is this type of loop.
   a. pretest
   b. posttest
   c. prefix
   d. postfix

8. The `for` loop is this type of loop.
   a. pretest
   b. posttest
   c. prefix
   d. postfix
9. This type of loop has no way of ending and repeats until the program is interrupted.
   a. indeterminate
   b. interminable
   c. infinite
   d. timeless
10. This type of loop always executes at least once.
    a. `while`
    b. `do-while`
    c. `for`
    d. any of these
11. This expression is executed by the `for` loop only once, regardless of the number of iterations.
    a. initialization expression
    b. test expression
    c. update expression
    d. pre-increment expression
12. This is a variable that keeps a running total.
    a. sentinel
    b. sum
    c. total
    d. accumulator
13. This is a special value that signals when there are no more items from a list of items to be processed. This value cannot be mistaken as an item from the list.
    a. sentinel
    b. flag
    c. signal
    d. accumulator
14. To open a file for writing, you use the following class.
    a. `PrintWriter`
    b. `FileOpen`
    c. `OutputFile`
    d. `FileReader`
15. To open a file for reading, you use the following classes.
    a. `File` and `Writer`
    b. `File` and `Output`
    c. `File` and `Input`
    d. `File` and `Scanner`

16. When a program is finished using a file, it should do this.
    a. erase the file
    b. close the file
    c. throw an exception
    d. reset the read position
17. This class allows you to use the `print` and `println` methods to write data to a file.
    a. `File`
    b. `FileReader`
    c. `OutputFile`
    d. `PrintWriter`
18. This class allows you to read a line from a file.
    a. `FileWriter`
    b. `Scanner`
    c. `InputFile`
    d. `FileReader`
19. **True or False:** The `while` loop is a pretest loop.
20. **True or False:** The `do-while` loop is a pretest loop.
21. **True or False:** The `for` loop is a posttest loop.
22. **True or False:** It is not necessary to initialize accumulator variables.
23. **True or False:** One limitation of the `for` loop is that only one variable may be initialized in the initialization expression.
24. **True or False:** A variable may be defined in the initialization expression of the `for` loop.
25. **True or False:** In a nested loop, the inner loop goes through all of its iterations for every iteration of the outer loop.
26. **True or False:** To calculate the total number of iterations of a nested loop, add the number of iterations of all the loops.

Find the Error

Find the errors in the following code:

1.
```
// This code contains ERRORS!
// It adds two numbers entered by the user.
int num1, num2;
String input;
char again;

Scanner keyboard = new Scanner(System.in);
while (again == 'y' || again == 'Y')
   System.out.print("Enter a number: ");
   num1 = keyboard.nextInt();
   System.out.print("Enter another number: ";
   num2 = keyboard.nextInt();
   System.out.println("Their sum is "+ (num1 + num2));
   System.out.println("Do you want to do this again? ");
   keyboard.nextLine();  // Consume remaining newline
   input = keyboard.nextLine();
   again = input.charAt(0);
```

2.
```
// This code contains ERRORS!
int count = 1, total;
while (count <= 100)
   total += count;
System.out.print("The sum of the numbers 1 - 100 is ");
System.out.println(total);
```

3.
```
// This code contains ERRORS!
int choice, num1, num2;
Scanner keyboard = new Scanner(System.in);
do
{
   System.out.print("Enter a number: ");
   num1 = keyboard.nextInt();
   System.out.print("Enter another number: ");
   num2 = keyboard.nextInt();
   System.out.println("Their sum is " + (num1 + num2));
   System.out.println("Do you want to do this again? ");
   System.out.print("1 = yes, 0 = no ");
   choice = keyboard.nextInt();
} while (choice = 1)
```

4.
```
// This code contains ERRORS!
// Print the numbers 1 through 10.
for (int count = 1, count <= 10, count++;)
{
      System.out.println(count);
      count++;
}
```

Algorithm Workbench

1. Write a `while` loop that lets the user enter a number. The number should be multiplied by 10, and the result stored in the variable `product`. The loop should iterate as long as `product` contains a value less than 100.
2. Write a `do-while` loop that asks the user to enter two numbers. The numbers should be added and the sum displayed. The loop should ask the user whether he or she wishes to perform the operation again. If so, the loop should repeat; otherwise it should terminate.
3. Write a `for` loop that displays the following set of numbers:
```
0, 10, 20, 30, 40, 50 ... 1000
```

4. Write a loop that asks the user to enter a number. The loop should iterate 10 times and keep a running total of the numbers entered.
5. Write a `for` loop that calculates the total of the following series of numbers:

$$\frac{1}{30}+\frac{2}{29}+\frac{3}{28}+\dots\frac{30}{1}$$

6. Write a nested loop that displays 10 rows of `'#'` characters. There should be 15 `'#'` characters in each row.
7. Convert the `while` loop in the following code to a `do-while` loop:

```
Scanner keyboard = new Scanner(System.in);
int x = 1;
while (x > 0)
{
   System.out.print("Enter a number: ");
   x = keyboard.nextInt();
}
```

8. Convert the `do-while` loop in the following code to a `while` loop:

```
Scanner keyboard = new Scanner(System.in);
String input;
char sure;
do
{
   System.out.print("Are you sure you want to quit? ");
   input = keyboard.next();
   sure = input.charAt(0);
} while (sure != 'Y' && sure != 'N');
```

9. Convert the following `while` loop to a `for` loop:

```
int count = 0;
while (count < 50)
{
   System.out.println("count is " + count);
   count++;
}
```

10. Convert the following `for` loop to a `while` loop:

```
for (int x = 50; x > 0; x--)
{
   System.out.println(x + " seconds to go.");
}
```

11. Write an input validation loop that asks the user to enter a number in the range of 1 through 4.
12. Write an input validation loop that asks the user to enter the word "yes" or "no".

13. Write nested loops to draw this pattern:

```
*******
******
*****
****
***
**
*
```

14. Write nested loops to draw this pattern:

```
##
# #
#  #
#   #
#    #
#     #
```

15. Complete the following program so it displays a random integer in the range of 1 through 10.

```
// Write the necessary import statement(s) here.
public class ReviewQuestion15
{
   public static void main(String[] args)
   {
      // Write the necessary code here.
   }
}
```

16. Complete the following program so it performs the following actions 10 times:
    - Generates a random number that is either 0 or 1.
    - Displays either the word "Yes" or the word "No" depending on the random number that was generated.

```
// Write the necessary import statement(s) here.
public class ReviewQuestion16
{
   public static void main(String[] args)
   {
      // Write the necessary code here.
   }
}
```

17. Write code that does the following: opens a file named *NumberList.txt*, uses a loop to write the numbers 1 through 100 to the file, and then closes the file.
18. Write code that does the following: opens the *NumberList.txt* file that was created by the code in Question 17, reads all of the numbers from the file and displays them, and then closes the file.
19. Modify the code you wrote in Question 18 so it adds all of the numbers read from the file and displays their total.
20. Write code that opens a file named *NumberList.txt* for writing, but does not erase the file's contents if it already exists.

Short Answer

1. Briefly describe the difference between the prefix and postfix modes used by the increment and decrement operators.
2. Why should you indent the statements in the body of a loop?
3. Describe the difference between pretest loops and posttest loops.
4. Why are the statements in the body of a loop called conditionally executed statements?
5. Describe the difference between the `while` loop and the `do-while` loop.
6. Which loop should you use in situations where you want the loop to repeat until the `boolean` expression is `false`, and the loop should not execute if the test expression is `false` to begin with?
7. Which loop should you use in situations where you want the loop to repeat until the `boolean` expression is `false`, but the loop should execute at least once?
8. Which loop should you use when you know the number of required iterations?
9. Why is it critical that accumulator variables are properly initialized?
10. What is an infinite loop? Write the code for an infinite loop.
11. Describe a programming problem that would require the use of an accumulator.
12. What does it mean to let the user control a loop?
13. What is the advantage of using a sentinel?
14. Why must the value chosen for use as a sentinel be carefully selected?
15. Describe a programming problem requiring the use of nested loops.
16. How does a file buffer increase a program's performance?
17. Why should a program close a file when it's finished using it?
18. What is a file's read position? Where is the read position when a file is first opened for reading?
19. When writing data to a file, what is the difference between the `print` and the `println` methods?
20. What does the `Scanner` class's `hasNext` method return when the end of the file has been reached?
21. What is a potential error that can occur when a file is opened for reading?
22. What does it mean to append data to a file?
23. How do you open a file so that new data will be written to the end of the file's existing data?

Programming Challenges

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

1. Sum of Numbers

Write a program that asks the user for a positive nonzero integer value. The program should use a loop to get the sum of all the integers from 1 up to the number entered. For example, if the user enters 50, the loop will find the sum of 1, 2, 3, 4, . . . 50.

2. Distance Traveled

The distance a vehicle travels can be calculated as follows:

*Distance = Speed * Time*

For example, if a train travels 40 miles-per-hour for three hours, the distance traveled is 120 miles. Write a program that asks for the speed of a vehicle (in miles-per-hour) and the number of hours it has traveled. It should use a loop to display the distance a vehicle has traveled for each hour of a time period specified by the user. For example, if a vehicle is traveling at 40 mph for a three-hour time period, it should display a report similar to the one that follows:

```
Hour     Distance Traveled
--------------------------
1              40
2              80
3             120
```

Input Validation: Do not accept a negative number for speed and do not accept any value less than 1 for time traveled.

3. Distance File

Modify the program you wrote for Programming Challenge 2 (Distance Traveled) so it writes the report to a file instead of the screen. Open the file in Notepad or another text editor to confirm the output.

4. Pennies for Pay

VideoNote The Pennies for Pay Problem

Write a program that calculates the amount a person would earn over a period of time if his or her salary is one penny the first day, two pennies the second day, and continues to double each day. The program should display a table showing the salary for each day, and then show the total pay at the end of the period. The output should be displayed in a dollar amount, not the number of pennies.

Input Validation: Do not accept a number less than 1 for the number of days worked.

5. Letter Counter

Write a program that asks the user to enter a string, and then asks the user to enter a character. The program should count and display the number of times that the specified character appears in the string.

6. File Letter Counter

Write a program that asks the user to enter the name of a file, and then asks the user to enter a character. The program should count and display the number of times that the specified character appears in the file. Use Notepad or another text editor to create a simple file that can be used to test the program.

7. Hotel Occupancy

A hotel's occupancy rate is calculated as follows:

Occupancy rate = Number of rooms occupied ÷ Total number of rooms

Write a program that calculates the occupancy rate for each floor of a hotel. The program should start by asking for the number of floors in the hotel. A loop should then iterate once for each floor. During each iteration, the loop should ask the user for the number of rooms on the floor and the number of them that are occupied. After all the iterations, the program should display the number of rooms the hotel has, the number of them that are occupied, the number that are vacant, and the occupancy rate for the hotel.

Input Validation: Do not accept a value less than 1 for the number of floors. Do not accept a number less than 10 for the number of rooms on a floor.

8. Average Rainfall

Write a program that uses nested loops to collect data and calculate the average rainfall over a period of years. First the program should ask for the number of years. The outer loop will iterate once for each year. The inner loop will iterate 12 times, once for each month. Each iteration of the inner loop will ask the user for the inches of rainfall for that month. After all iterations, the program should display the number of months, the total inches of rainfall, and the average rainfall per month for the entire period.

Input Validation: Do not accept a number less than 1 for the number of years. Do not accept negative numbers for the monthly rainfall.

9. Population

Write a program that will predict the size of a population of organisms. The program should ask for the starting number of organisms, their average daily population increase (as a percentage), and the number of days they will multiply. For example, a population might begin with two organisms, have an average daily increase of 50 percent, and will be allowed to multiply for seven days. The program should use a loop to display the size of the population for each day.

Input Validation: Do not accept a number less than 2 for the starting size of the population. Do not accept a negative number for average daily population increase. Do not accept a number less than 1 for the number of days they will multiply.

10. Largest and Smallest

Write a program with a loop that lets the user enter a series of integers. The user should enter –99 to signal the end of the series. After all the numbers have been entered, the program should display the largest and smallest numbers entered.

11. Celsius to Fahrenheit Table

Write a program that displays a table of the Celsius temperatures 0 through 20 and their Fahrenheit equivalents. The formula for converting a temperature from Celsius to Fahrenheit is

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

where *F* is the Fahrenheit temperature and *C* is the Celsius temperature. Your program must use a loop to display the table.

12. Bar Chart

Write a program that asks the user to enter today's sales for five stores. The program should display a bar chart comparing each store's sales. Create each bar in the bar chart by displaying a row of asterisks. Each asterisk should represent $100 of sales. Here is an example of the program's output:

```
Enter today's sales for store 1: 1000 [Enter]
Enter today's sales for store 2: 1200 [Enter]
Enter today's sales for store 3: 1800 [Enter]
Enter today's sales for store 4: 800 [Enter]
Enter today's sales for store 5: 1900 [Enter]

SALES BAR CHART
Store 1: **********
Store 2: ************
Store 3: ******************
Store 4: ********
Store 5: *******************
```

13. File Head Display

Write a program that asks the user for the name of a file. The program should display only the first five lines of the file's contents. If the file contains fewer than five lines, it should display the file's entire contents.

14. Line Numbers

Write a program that asks the user for the name of a file. The program should display the contents of the file with each line preceded with a line number followed by a colon. The line numbering should start at 1.

15. Uppercase File Converter

Write a program that asks the user for the names of two files. The first file should be opened for reading and the second file should be opened for writing. The program should read the contents of the first file, change all characters to uppercase, and store the results in the second file. The second file will be a copy of the first file, except that all the characters will be uppercase. Use Notepad or another text editor to create a simple file that can be used to test the program.

16. Budget Analysis

Write a program that asks the user to enter the amount that he or she has budgeted for a month. A loop should then prompt the user to enter each of his or her expenses for the month, and keep a running total. When the loop finishes, the program should display the amount that the user is over or under budget.

17. Random Number Guessing Game

Write a program that generates a random number and asks the user to guess what the number is. If the user's guess is higher than the random number, the program should display "Too high, try again." If the user's guess is lower than the random number, the program should display "Too low, try again." The program should use a loop that repeats until the user correctly guesses the random number.

18. Random Number Guessing Game Enhancement

Enhance the program that you wrote for Programming Challenge 17 so it keeps a count of the number of guesses that the user makes. When the user correctly guesses the random number, the program should display the number of guesses.

19. Square Display

Write a program that asks the user for a positive integer no greater than 15. The program should then display a square on the screen using the character 'X'. The number entered by the user will be the length of each side of the square. For example, if the user enters 5, the program should display the following:

```
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
```

If the user enters 8, the program should display the following:

```
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
```

20. Dice Game

Write a program that plays a simple dice game between the computer and the user. When the program runs, a loop should repeat 10 times. Each iteration of the loop should do the following:

- Generate a random integer in the range of 1 through 6. This is the value of the computer's die.
- Generate another random integer in the range of 1 through 6. This is the value of the user's die.
- The die with the highest value wins. (In case of a tie, there is no winner for that particular roll of the dice.)

As the loop iterates, the program should keep count of the number of times the computer wins, and the number of times that the user wins. After the loop performs all of its iterations, the program should display who was the grand winner, the computer or the user.

21. Slot Machine Simulation

A slot machine is a gambling device that the user inserts money into and then pulls a lever (or presses a button). The slot machine then displays a set of random images. If two or more of the images match, the user wins an amount of money that the slot machine dispenses back to the user.

Create a program that simulates a slot machine. When the program runs, it should do the following:

- Asks the user to enter the amount of money he or she wants to enter into the slot machine.
- Instead of displaying images, the program will randomly select a word from the following list:

  Cherries, Oranges, Plums, Bells, Melons, Bars

  To select a word, the program can generate a random number in the range of 0 through 5. If the number is 0, the selected word is *Cherries*; if the number is 1, the selected word is *Oranges*; and so forth. The program should randomly select a word from this list three times and display all three of the words.
- If none of the randomly selected words match, the program will inform the user that he or she has won $0. If two of the words match, the program will inform the user that he or she has won two times the amount entered. If three of the words match, the program will inform the user that he or she has won three times the amount entered.
- The program will ask whether the user wants to play again. If so, these steps are repeated. If not, the program displays the total amount of money entered into the slot machine and the total amount won.Decision Structures

CHAPTER

5 Methods

TOPICS

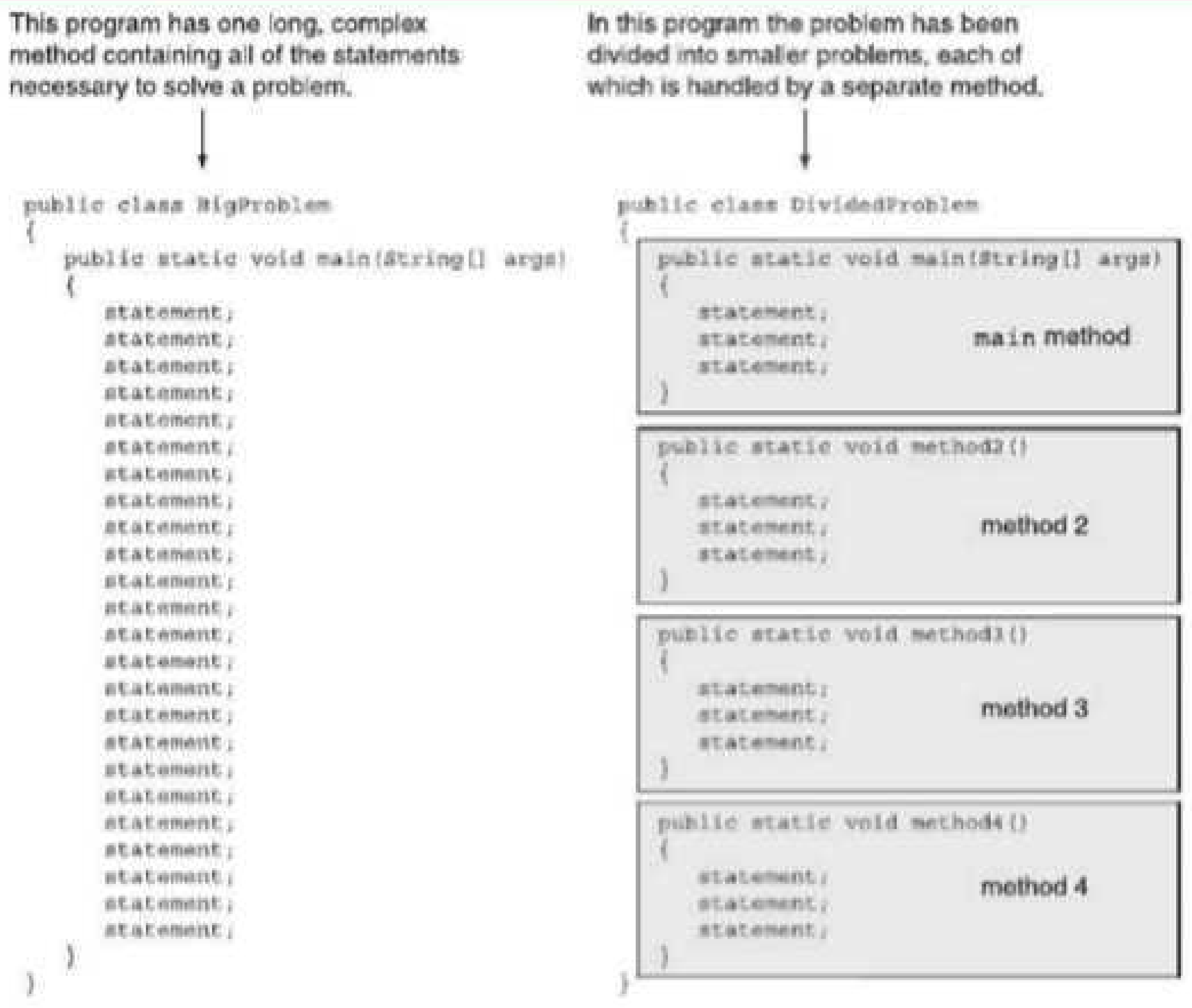
5.1 Introduction to Methods

CONCEPT: Methods can be used to break a complex program into small, manageable pieces. A `void` method simply executes a group of statements and then terminates. A value-returning method returns a value to the statement that called it.

In a general sense, a method is a collection of statements that performs a specific task. So far you have experienced methods in two ways: (1) You have created a method named `main` in every program you've written, and (2) you have executed predefined methods from the Java API, such as `System.out.println`, `Integer.parseInt`, and `Math.pow`. In this chapter you will learn how to create your own methods, other than `main`, that can be executed just as you execute the API methods.

Methods are commonly used to break a problem into small, manageable pieces. Instead of writing one long method that contains all of the statements necessary to solve a problem, several small methods that each solve a specific part of the problem can be written. These small methods can then be executed in the desired order to solve the problem. This approach is sometimes called *divide and conquer* because a large problem is divided into several smaller problems that are easily solved. Figure 5-1 illustrates this idea by comparing two programs: one that uses a long, complex method containing all of the statements necessary to solve a problem, and another that divides a problem into smaller problems, each of which is handled by a separate method.

Figure 5-1 Using methods to divide and conquer a problem


This program has one long, complex method containing all of the statements necessary to solve a problem.
In this program the problem has been divided into smaller problems, each of which is handled by a separate method.
main method
method 2
method 3
method 4

```
public class BigProblem
{
   public static void main(String[] args)
   {
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
   }
}
```

```
public class DividedProblem
{
   public static void main(String[] args)
   {
      statement;
      statement;
      statement;
   }

   public static void method2()
   {
      statement;
      statement;
      statement;
   }

   public static void method3()
   {
      statement;
      statement;
      statement;
   }

   public static void method4()
   {
      statement;
      statement;
      statement;
   }
}
```

Another reason to write methods is that they simplify programs. If a specific task is performed in several places in a program, a method can be written once to perform that task, and then be executed any time it is needed. This benefit of using methods is known as *code reuse* because you are writing the code to perform a task once and then reusing it each time you need to perform the task.

First, we will look at the general ways in which methods operate. At the end of the chapter we will discuss in greater detail how methods can be used in problem solving.

`void` Methods and Value-Returning Methods

In this chapter you will learn about two general categories of methods: `void` methods and value-returning methods. A *`void` method* is one that simply performs a task and then terminates. `System.out.println` is an example of a `void` method. For example, look at the following code:

```
int number = 7;
System.out.println(number);
number = 0;
```

The statement in line 1 declares the `number` variable and initializes it with the value 7. The statement in line 2 calls the `System.out.println` method, passing `number` as an argument. The method does its job, which is to display a value on the screen, and then terminates. The code then resumes at line 3.

A *value-returning method* not only performs a task but also sends a value back to the code that called it. The `Random` class's `nextInt` method is an example of a value-returning method. For example, look at the following code:

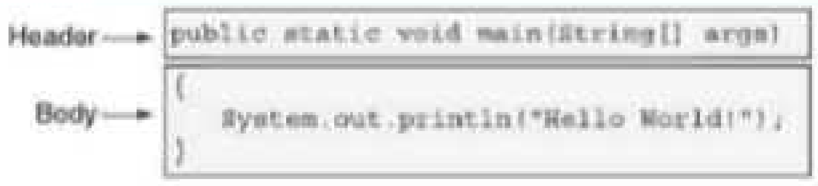
```
1    int number;
2    Random rand = new Random();
3    number = rand.nextInt();
```

The statement in line 1 declares the `number` variable. Line 2 creates a `Random` object and assigns its address to a variable named `rand`. Line 3 is an assignment statement, which assigns a value to the `number` variable. Notice that on the right side of the = operator is a call to the `rand.nextInt` method. The method executes, and then returns a value. The value that is returned from the method is assigned to the `number` variable.

Defining a `void` Method

To create a method you must write its *definition*, which consists of two general parts: a header and a body. You learned about both of these in Chapter 2, but let's briefly review. The *method header*, which appears at the beginning of a method definition, lists several important things about the method, including the method's name. The *method body* is a collection of statements that are performed when the method is executed. These statements are enclosed inside a set of curly braces. Figure 5-2 points out the header and body of a `main` method.

Figure 5-2 The header and body of a `main` method



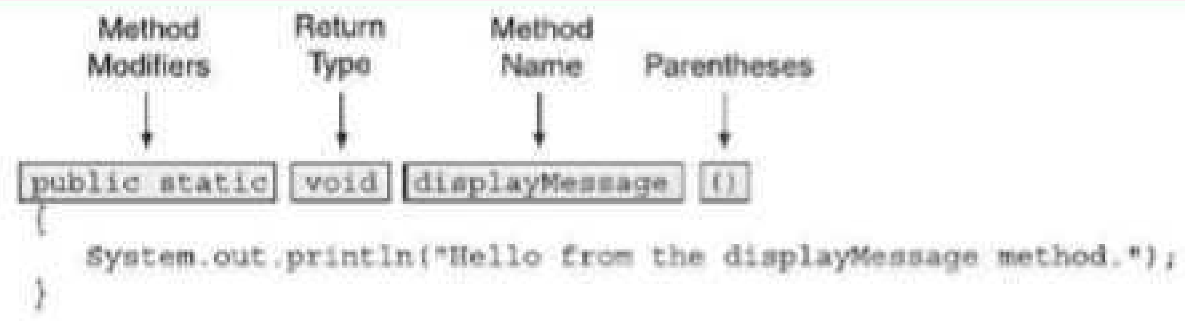
```
public static void main(String[] args)
{
    System.out.println("Hello World!");
}
```

As you already know, every complete Java program must have a `main` method. Java programs can have other methods as well. Here is an example of a simple method that displays a message on the screen:

```
public static void displayMessage()
{
   System.out.println("Hello from the displayMessage method.");
}
```

This method has a header and a body. Figure 5-3 shows the different parts of the method header.

Figure 5-3 Parts of the method header



```
public static void displayMessage ()
{
   System.out.println("Hello from the displayMessage method.");
}
```

Let's take a closer look at the parts identified in the figure as follows:

- **Method modifiers**—The key words `public` and `static` are modifiers. You don't need to be too concerned with these modifiers now, but if your curiosity is getting the best of you, here's a brief explanation: The word `public` means that the method is publicly available to code outside the class. The word `static` means that the method belongs to the class, not a specific object. You will learn more about these modifiers in later chapters. For this chapter, every method that we write will begin with `public static`.
- **Return type**—Recall our previous discussion of `void` and value-returning methods. When the key word `void` appears here, it means that the method is a `void` method, and does not return a value. As you will see later, a value-returning method lists a data type here.
- **Method name**—You should give each method a descriptive name. In general, the same rules that apply to variable names also apply to method names. This method is named `displayMessage`, so we can easily guess what the method does: It displays a message.
- **Parentheses**—In the header, the method name is always followed by a set of parentheses. As you will learn later in this chapter, methods can be capable of receiving arguments. When this is the case, a list of one or more variable declarations will appear inside the parentheses. The method in this example does not receive any arguments, so the parentheses are empty.

NOTE: The method header is never terminated with a semicolon.

Calling a Method

A method executes when it is called. The `main` method is automatically called when a program starts, but other methods are executed by method call statements. When a method is called, the JVM branches to that method and executes the statements in its body. Here is an example of a method call statement that calls the `displayMessage` method we previously examined:

```
displayMessage();
```

The statement is simply the name of the method followed by a set of parentheses. Because it is a complete statement, it is terminated with a semicolon.

TIP: Notice that the method modifiers and the `void` return type are not written in the method call statement. They are written only in the method header.

The program in Code Listing 5-1 demonstrates.

Code Listing 5-1 **(`SimpleMethod.java`)**

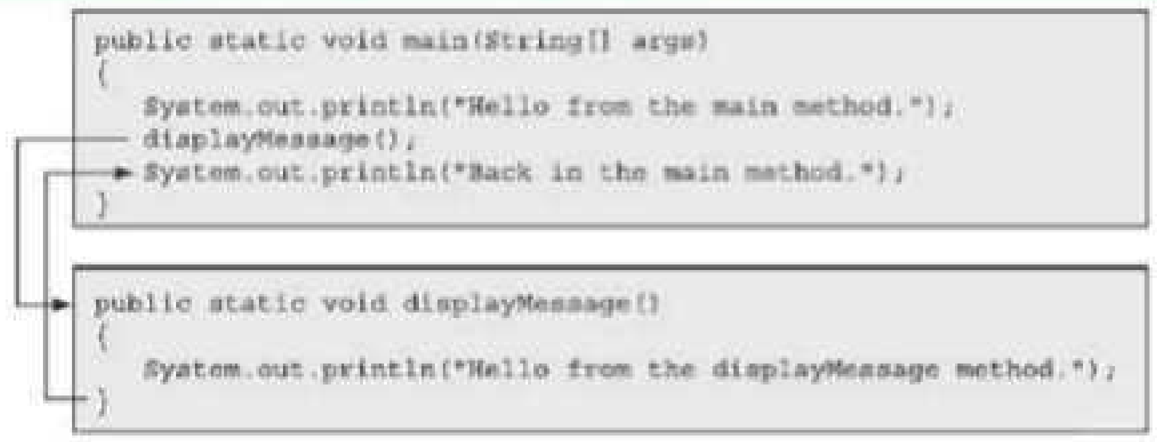
```
 1  /**
 2     This program defines and calls a simple method.
 3  */
 4
 5  public class SimpleMethod
 6  {
 7     public static void main(String[] args)
 8     {
 9        System.out.println("Hello from the main method.");
10        displayMessage();
11        System.out.println("Back in the main method.");
12     }
13
14     /**
15        The displayMessage method displays a greeting.
16     */
17
18     public static void displayMessage()
19     {
20        System.out.println("Hello from the displayMessage method.");
21     }
22  }
```

Program Output

```
Hello from the main method.
Hello from the displayMessage method.
Back in the main method.
```

Notice how the program flows. It starts, of course, in the `main` method. When the call to the `displayMessage` method in line 10 is encountered, the JVM branches to that method and performs the statement in its body (at line 20). Once the `displayMessage` method has finished executing, the JVM branches back to the `main` method and resumes at line 11 with the statement that follows the method call. This is illustrated in Figure 5-4.

Figure 5-4 Branching in the `SimpleMethod.java` program



```
public static void main(String[] args)
{
   System.out.println("Hello from the main method.");
   displayMessage();
   System.out.println("Back in the main method.");
}

public static void displayMessage()
{
   System.out.println("Hello from the displayMessage method.");
}
```

Method call statements may be used in control structures like loops, `if` statements, and `switch` statements. The program in Code Listing 5-2 places the `displayMessage` method call inside a loop.

Code Listing 5-2 (`LoopCall.java`)

```
/**
   This program defines and calls a simple method.
*/

public class LoopCall
{
   public static void main(String[] args)
   {
      System.out.println("Hello from the main method.");
      for (int i = 0; i < 5; i++)
         displayMessage();
      System.out.println("Back in the main method.");
   }

   /**
      The displayMessage method displays a greeting.
   */

   public static void displayMessage()
   {
      System.out.println("Hello from the displayMessage method.");
   }
}
```

Program Output

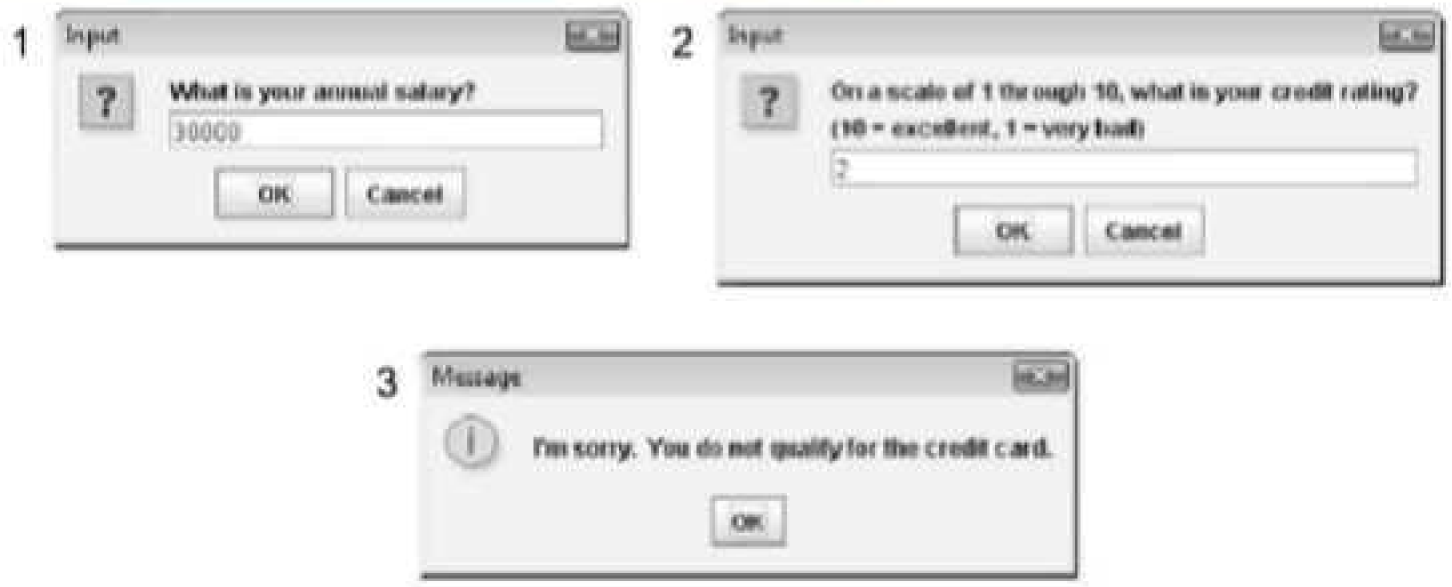
```
Hello from the main method.
Hello from the displayMessage method.
Hello from the displayMessage method.
Hello from the displayMessage method.
Hello from the displayMessage method.
Hello from the displayMessage method.
Back in the main method.
```

The program in Code Listing 5-3 shows another example. It asks the user to enter his or her annual salary and credit rating. The program then determines whether the user qualifies for a credit card. One of two void methods, `qualify` or `noQualify`, is called to display a message. Figures 5-5 and 5-6 show example interactions with the program.

Code Listing 5-3 (`CreditCard.java`)

```
import javax.swing.JOptionPane; 2

/**
   This program uses two void methods.
*/

public class CreditCard
{
   public static void main(String[] args)
   {
      double salary;        // Annual salary
      int creditRating;     // Credit rating
      String input;         // To hold the user's input

      // Get the user's annual salary.
      input = JOptionPane.showInputDialog("What is " +
                                 "your annual salary?");
      salary = Double.parseDouble(input);

      // Get the user's credit rating (1 through 10).
      input = JOptionPane.showInputDialog("On a scale of " +
            "1 through 10, what is your credit rating?\n" +
            "(10 = excellent, 1 = very bad)");
      creditRating = Integer.parseInt(input);

      // Determine whether the user qualifies.
      if (salary >= 20000 && creditRating >= 7)
         qualify();
      else
         noQualify();

      System.exit(0);
   }

   /**
      The qualify method informs the user that he
      or she qualifies for the credit card.
   */

   public static void qualify()
   {
      JOptionPane.showMessageDialog(null, "Congratulations! " +
                                  "You qualify for the credit card!");
   }

   /**
      The noQualify method informs the user that he
      or she does not qualify for the credit card.
   */

   public static void noQualify()
   {
      JOptionPane.showMessageDialog(null, "I'm sorry. You " +
                             "do not qualify for the credit card.");
   }
}
```

Figure 5-5 Interaction with the `CreditCard.java` program



1 Input

What is your annual salary?

30000

OK Cancel

2 Input

On a scale of 1 through 10, what is your credit rating?
(10 = excellent, 1 = very bad)

2

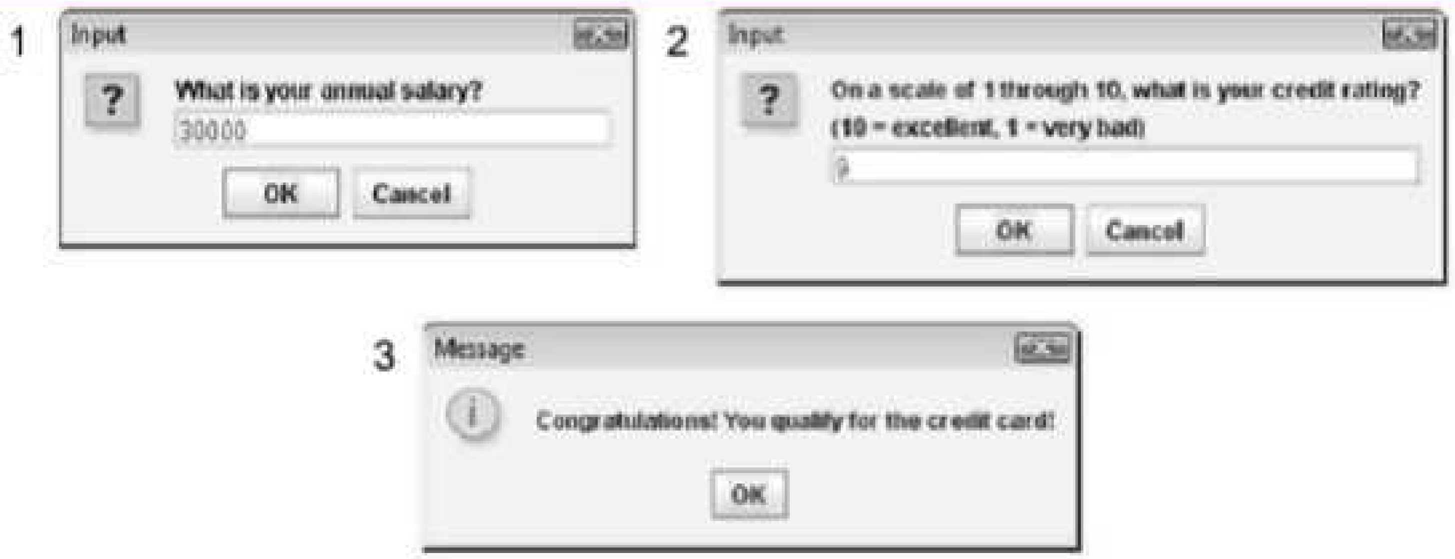
OK Cancel

3 Message

I'm sorry. You do not qualify for the credit card.

OK

Figure 5-6 Interaction with the `CreditCard.java` program



1 Input

What is your annual salary?

30000

OK Cancel

2 Input

On a scale of 1 through 10, what is your credit rating?
(10 = excellent, 1 = very bad)

8

OK Cancel

3 Message

Congratulations! You qualify for the credit card!

OK

Hierarchical Method Calls

Methods can also be called in a hierarchical, or layered fashion. In other words, method A can call method B, which can then call method C. When method C finishes, the JVM returns to method B. When method B finishes, the JVM returns to method A. The program in Code Listing 5-4 demonstrates this with three methods: `main`, `deep`, and `deeper`. The `main` method calls the `deep` method, which then calls the `deeper` method.

Code Listing 5-4 `(DeepAndDeeper.java)`

```java
/**
   This program demonstrates hierarchical method calls.
*/

public class DeepAndDeeper
{
   public static void main(String[] args)
   {
      System.out.println("I am starting in main.");
      deep();
      System.out.println("Now I am back in main.");
   }

   /**
      The deep method displays a message and then calls
      the deeper method.
   */

   public static void deep()
   {
      System.out.println("I am now in deep.");
      deeper();
      System.out.println("Now I am back in deep.");
   }

   /**
      The deeper method simply displays a message.
   */

   public static void deeper()
   {
      System.out.println("I am now in deeper.");
   }
}
```

Program Output

```
I am starting in main.
I am now in deep.
```

```
I am now in deeper.
Now I am back in deep.
Now I am back in main.
```

Using Documentation Comments with Methods

You should always document a method by writing comments that appear just before the method's definition. The comments should provide a brief explanation of the method's purpose. Notice that the programs we've looked at in this chapter use documentation comments. Recall from Chapter 2 that documentation comments begin with `/**` and end with `*/`. These types of comments can be read and processed by a program named `javadoc`, which produces attractive HTML documentation. As we progress through this chapter, you will learn more about documentation comments and how they can be used with methods.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

5.1 What is the difference between a `void` method and a value-returning method?

5.2 Is the following line of code a method header or a method call?

```
calcTotal();
```

5.3 Is the following line of code a method header or a method call?

```
public static void calcTotal()
```

5.4 What message will the following program display if the user enters 5? What if the user enters 10? What if the user enters 100?

```
import javax.swing.JOptionPane;
public class Checkpoint
{
   public static void main(String[] args)
   {
      String input;
      int number;

      input = JOptionPane.showInputDialog("Enter a number.");
      number = Integer.parseInt(input);

      if (number < 10)
         method1();
      else
         method2();

      System.exit(0);
   }

   public static void method1()
   {
      JOptionPane.showMessageDialog(null, "Able was I.");
   }
```

```
      public static void method2()
      {
         JOptionPane.showMessageDialog(null, "I saw Elba.");
      }
   }
```

5.5 Write a void method that displays your full name. The method should be named myName.

5.2 Passing Arguments to a Method

CONCEPT: A method may be written so it accepts arguments. Data can then be passed into the method when it is called.

VideoNote Passing Arguments to a Method

Values that are sent into a method are called *arguments*. You're already familiar with how to use arguments in a method call. For example, look at the following statement:

```
System.out.println("Hello");
```

This statement calls the System.out.println method and passes "Hello" as an argument. Here is another example:

```
number = Integer.parseInt(str);
```

This statement calls the Integer.parseInt method and passes the contents of the str variable as an argument. By using parameter variables, you can design your own methods that accept data this way. A *parameter variable*, sometimes simply referred to as a *parameter*, is a special variable that holds a value being passed into a method. Here is the definition of a method that uses a parameter:

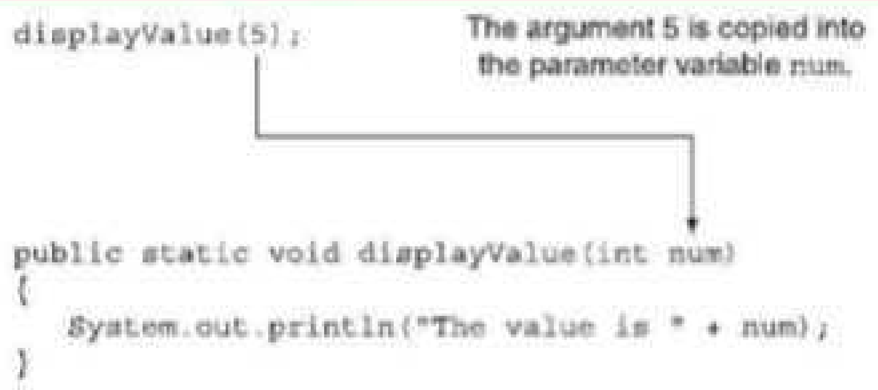
```
public static void displayValue(int num)
{
   System.out.println("The value is " + num);
}
```

Notice the integer variable declaration that appears inside the parentheses (int num). This is the declaration of a parameter variable, which enables the displayValue method to accept an integer value as an argument. Here is an example of a call to the displayValue method, passing 5 as an argument:

```
displayValue(5);
```

This statement executes the displayValue method. The argument that is listed inside the parentheses is copied into the method's parameter variable, num. This is illustrated in Figure 5-7.

Figure 5-7 Passing 5 to the `displayValue` method



```
displayValue(5);
```

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

Inside the `displayValue` method, the variable `num` will contain the value of whatever argument was passed into it. If we pass *5* as the argument, the method will display as follows:

```
The value is 5
```

You may also pass the contents of variables and the values of expressions as arguments. For example, the following statements call the `displayValue` method with various arguments passed:

```
displayValue(x);
displayValue(x * 4);
displayValue(Integer.parseInt("700"));
```

The first statement is simple. It passes the value in the variable `x` as the argument to the `displayValue` method. The second statement is also simple, but it does a little more work: It passes the result of the expression `x * 4` as the argument to the `displayValue` method. The third statement does even more work. It passes the value returned from the `Integer.parseInt` method as the argument to the `displayValue` method. (The `Integer.parseInt` method is called first, and its return value is passed to the `displayValue` method.) The program in Code Listing 5-5 demonstrates these method calls.

Code Listing 5-5 **(PassArg.java)**

```
/**
   This program demonstrates a method with a parameter.
*/

public class PassArg
{
   public static void main(String[] args)
   {
      int x = 10;

      System.out.println("I am passing values to displayValue.");
      displayValue(5);                          // Pass 5
      displayValue(x);                          // Pass 10
      displayValue(x * 4);                      // Pass 40
      displayValue(Integer.parseInt("700"));    // Pass 700
      System.out.println("Now I am back in main.");
   }

   /**
      The displayValue method displays the value
      of its integer parameter.
   */

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

Program Output

```
I am passing values to displayValue.
The value is 5
The value is 10
The value is 40
The value is 700
Now I am back in main.
```

WARNING! When passing a variable as an argument, simply write the variable name inside the parentheses of the method call. Do not write the data type of the argument variable in the method call. For example, the following statement will cause an error:

```
   displayValue(int x);     // Error!
```

The method call should appear as follows:

```
   displayValue(x);         // Correct
```

NOTE: In this text, the values that are passed into a method are called arguments, and the variables that receive those values are called parameters. There are several variations of these terms in use. In some circles these terms are switched in meaning. Also, some call the arguments *actual parameters* and call the parameters *formal parameters*. Others use the terms *actual argument* and *formal argument*. Regardless of which set of terms you use, it is important to be consistent.

Argument and Parameter Data Type Compatibility

When you pass an argument to a method, be sure that the argument's data type is compatible with the parameter variable's data type. Java will automatically perform a widening conversion if the argument's data type is ranked lower than the parameter variable's data type. For example, the `displayValue` method has an `int` parameter variable. Both of the following code segments will work because the `short` and `byte` arguments are automatically converted to an `int`:

```
short s = 1;
displayValue(s);      // Converts short to int

byte b = 2;
displayValue(b);      // Converts byte to int
```

However, Java will not automatically convert an argument to a lower-ranking data type. This means that a `long`, `float`, or `double` value cannot be passed to a method that has an `int` parameter variable. For example, the following code will cause a compiler error:

```
double d = 1.0;
displayValue(d);   // Error! Can't convert double to int.
```

TIP: You can use a cast operator to convert a value manually to a lower-ranking data type. For example, the following code will compile:

```
double d = 1.0;
displayValue((int)d);   // This will work.
```

Parameter Variable Scope

Recall from Chapter 2 that a variable's scope is the part of the program where the variable may be accessed by its name. A variable is visible only to statements inside the variable's scope. A parameter variable's scope is the method in which the parameter is declared. No statement outside the method can access the parameter variable by its name.

Passing Multiple Arguments

Often it is useful to pass more than one argument to a method. Here is a method that accepts two arguments:

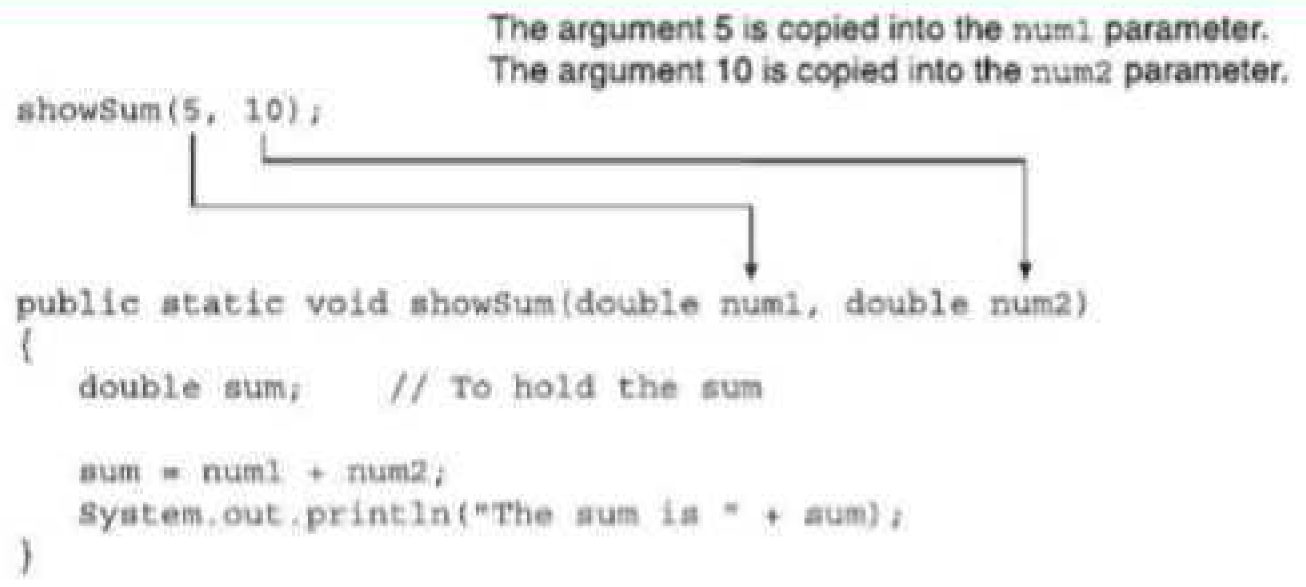
```
public static void showSum(double num1, double num2)
{
   double sum;    // To hold the sum

   sum = num1 + num2;
   System.out.println("The sum is " + sum);
}
```

Notice that two parameter variables, `num1` and `num2`, are declared inside the parentheses in the method header. This is often referred to as a *parameter list*. Also notice that a comma separates the declarations. Here is an example of a statement that calls the method:

```
showSum(5, 10);
```

This statement passes the arguments 5 and 10 into the method. The arguments are passed into the parameter variables in the order that they appear in the method call. In other words, the first argument is passed into the first parameter variable, the second argument is passed into the second parameter variable, and so forth. So, this statement causes 5 to be passed into the num1 parameter and 10 to be passed into the num2 parameter. This is illustrated in Figure 5-8.

Figure 5-8 Multiple arguments passed into multiple parameters


The argument 5 is copied into the num1 parameter.
The argument 10 is copied into the num2 parameter.

```
showSum(5, 10);

public static void showSum(double num1, double num2)
{
   double sum;     // To hold the sum

   sum = num1 + num2;
   System.out.println("The sum is " + sum);
}
```

Suppose we were to reverse the order in which the arguments are listed in the method call, as shown here:

```
showSum(10, 5);
```

This would cause 10 to be passed into the num1 parameter and 5 to be passed into the num2 parameter. The following code segment shows one more example. This time we are passing variables as arguments.

```
double value1 = 2.5;
double value2 = 3.5;
showSum(value1, value2);
```

When the showSum methods executes as a result of this code, the num1 parameter will contain 2.5 and the num2 parameter will contain 3.5.

WARNING! Each parameter variable in a parameter list must have a data type listed before its name. For example, a compiler error would occur if the parameter list for the showSum method were defined as shown in the following header:

```
public static void showSum(double num1, num2) // Error!
```

A data type for both the num1 and num2 parameter variables must be listed, as shown here:

```
public static void showSum(double num1, double num2)
```

See the program `TwoArgs.java` in this chapter's source code folder for a complete program that demonstrates the `showSum` method. You can download the book's source code from www.pearsonhighered.com/gaddis.

Arguments Are Passed by Value

In Java, all arguments of the primitive data types are *passed by value*, which means that only a copy of an argument's value is passed into a parameter variable. A method's parameter variables are separate and distinct from the arguments that are listed inside the parentheses of a method call. If a parameter variable is changed inside a method, it has no effect on the original argument. For example, look at the program in Code Listing 5-6.

Code Listing 5-6 (`PassByValue.java`)

```
/**
   This program demonstrates that only a copy of an argument
   is passed into a method.
*/

public class PassByValue
{
   public static void main(String[] args)
   {
      int number = 99; // number starts with 99

      // Display the value in number.
      System.out.println("number is " + number);

      // Call changeMe, passing the value in number
      // as an argument.
      changeMe(number);

      // Display the value in number again.
      System.out.println("number is " + number);
   }

   /**
      The changeMe method accepts an argument and then
      changes the value of the parameter.
   */

   public static void changeMe(int myValue)
   {
      System.out.println("I am changing the value.");

      // Change the myValue parameter variable to 0.
      myValue = 0;

      // Display the value in myValue.
      System.out.println("Now the value is " + myValue);
   }
}
```

Program Output

```
number is 99
I am changing the value.
Now the value is 0
number is 99
```

Even though the parameter variable `myValue` is changed in the `changeMe` method, the argument `number` is not modified. The `myValue` variable contains only a copy of the `number` variable.

Passing Object References to a Method

So far you've seen examples of methods that accept primitive values as arguments. You can also write methods that accept references to objects as arguments. For example, look at the following method:

```
public static void showLength(String str)
{
   System.out.println(str + " is " + str.length() +
                      " characters long.");
}
```

This method accepts a `String` object reference as its argument, and displays a message showing the number of characters in the object. The following code shows an example of how to call the method:

```
String name = "Warren";
showLength(name);
```

When this code executes, the `showLength` method will display the following:

```
Warren is 6 characters long.
```

When an object, such as a `String`, is passed as an argument, it is actually a reference to the object that is passed. In this example code, the `name` variable is a `String` reference variable. It is passed as an argument to the `showLength` method. The `showLength` method has a parameter variable, `str`, which is also a `String` reference variable, that receives the argument.

Recall that a reference variable holds the memory address of an object. When the `showLength` method is called, the address that is stored in `name` is passed into the `str` parameter variable. This is illustrated in Figure 5-9. This means that when the `showLength` method is executing, both `name` and `str` reference the same object. This is illustrated in Figure 5-10.

Figure 5-9 Passing a reference as an argument

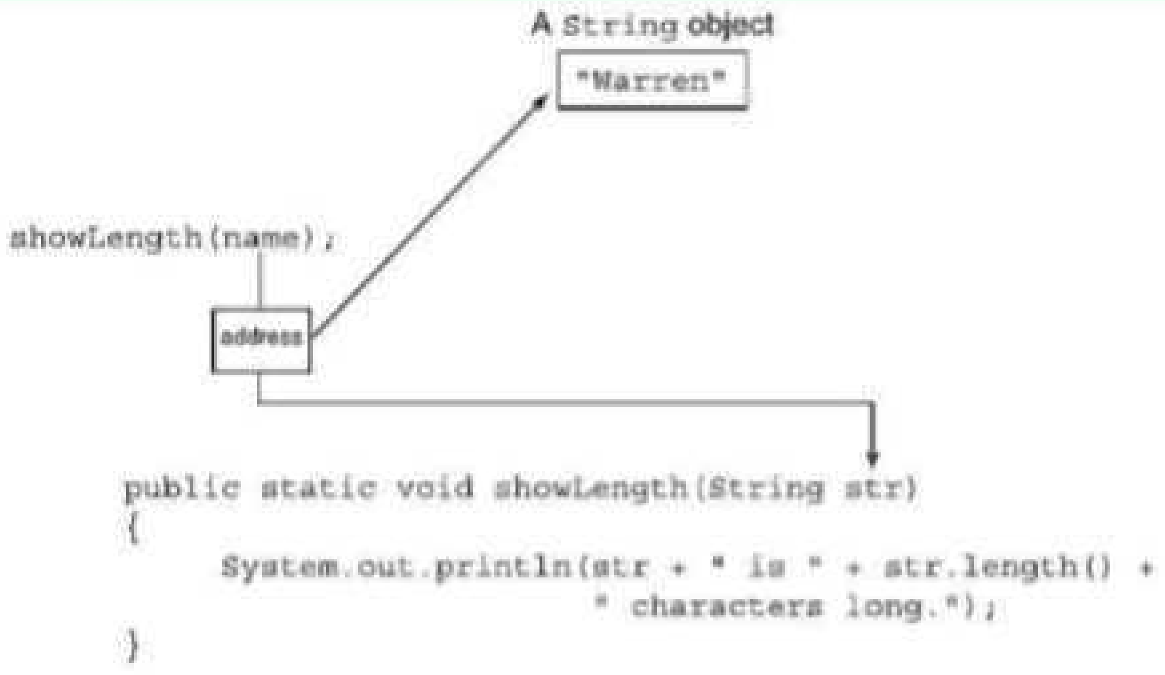
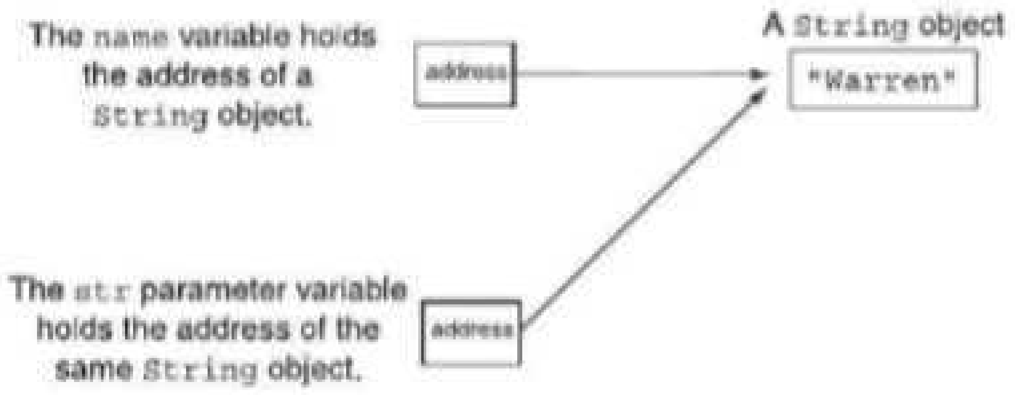

A String object
"Warren"
showLength(name);
address
public static void showLength(String str)
{
System.out.println(str + " is " + str.length() +
" characters long.");
}

Figure 5-10 Both `name` and `str` reference the same object


The name variable holds the address of a String object.
address
A String object
"Warren"
The str parameter variable holds the address of the same String object.
address

This might lead you to the conclusion that a method can change the contents of any `String` object that has been passed to it as an argument. After all, the parameter variable references the same object as the argument. However, `String` objects in Java are *immutable*, which means that they cannot be changed. For example, look at the program in Code Listing 5-7. It passes a `String` object to a method, which appears to change the object. In reality, the object is not changed.

Code Listing 5-7 **(PassString.java)**

```
/**
   This program demonstrates that String arguments
   cannot be changed.
*/

public class PassString
{
   public static void main(String[] args)
```

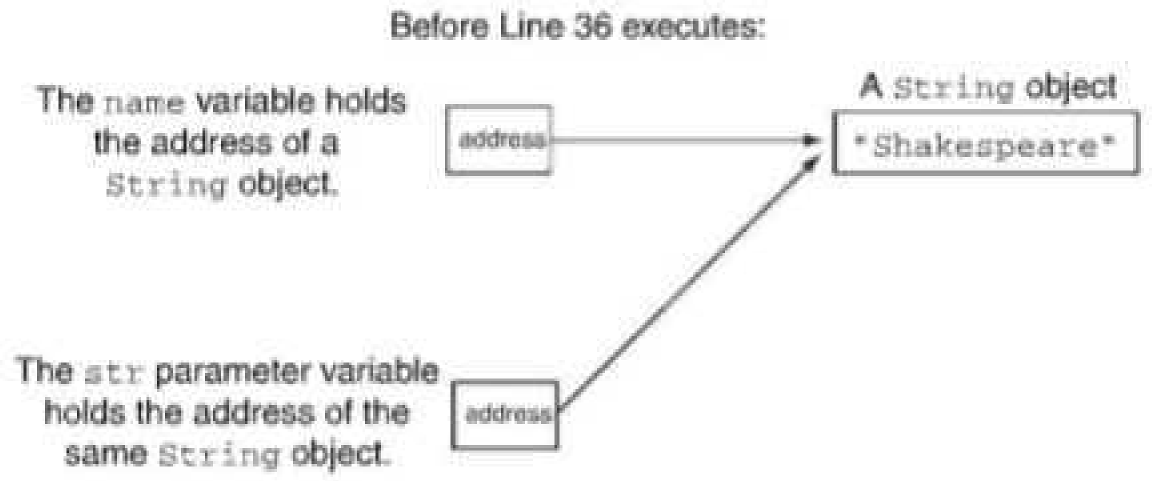
```
 9      {
10         // Create a String object containing "Shakespeare".
11         // The name variable references the object.
12         String name = "Shakespeare";
13
14         // Display the String referenced by the name variable.
15         System.out.println("In main, the name is " +
16                            name);
17
18         // Call the changeName method, passing the
19         // contents of the name variable as an argument.
20         changeName(name);
21
22         // Display the String referenced by the name variable.
23         System.out.println("Back in main, the name is " +
24                            name);
25      }
26
27      /**
28         The changeName method accepts a String as its argument
29         and assigns the str parameter to a new String.
30      */
31
32      public static void changeName(String str)
33      {
34         // Create a String object containing "Dickens".
35         // Assign its address to the str parameter variable.
36         str = "Dickens";
37
38         // Display the String referenced by str.
39         System.out.println("In changeName, the name " +
40                            "is now " + str);
41      }
42   }
```

Program Output

```
In main, the name is Shakespeare
In changeName, the name is now Dickens
Back in main, the name is Shakespeare
```

Let's take a closer look at this program. After line 12 executes, the `name` variable references a `String` object containing "Shakespeare". In line 20 the `changeName` method is called and the `name` variable is passed as an argument. This passes the address of the `String` object into the `str` parameter variable. At this point, both `name` and `str` reference the same object, as shown in Figure 5-11.

Figure 5-11 Before line 36 executes, both `name` and `str` reference the same object

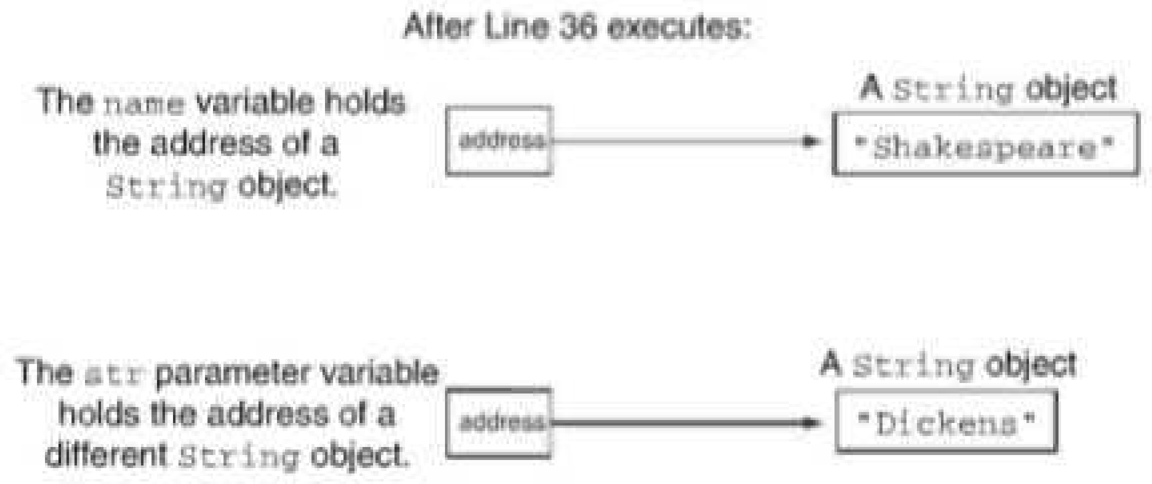

Before Line 36 executes:
The name variable holds the address of a String object.
address
A String object
"Shakespeare"
The str parameter variable holds the address of the same String object.
address

In the `changeName` method, line 36 executes as follows:

```
str = "Dickens";
```

At first, you might think that this statement changes the `String` object's contents to "Dickens". What actually happens is that a new `String` object containing "Dickens" is created and its address is stored in the `str` variable. After this statement executes, the `name` variable and the `str` parameter variable reference different objects. This is shown in Figure 5-12.

Figure 5-12 After line 36 executes, `name` and `str` reference different objects


After Line 36 executes:
The name variable holds the address of a String object.
address
A String object
"Shakespeare"
The str parameter variable holds the address of a different String object.
address
A String object
"Dickens"

In Chapter 9 we will discuss the immutability of `String` objects in greater detail. Until then, just remember the following point: `String` objects cannot be changed. Any time you use the `=` operator to assign a string literal to a `String` reference variable, a new `String` object is created in memory.

Using the `@param` Tag in Documentation Comments

When writing the documentation comments for a method, you can provide a description of each parameter by using a `@param` tag. When the `javadoc` utility sees a `@param` tag inside a method's documentation comments, it knows that the documentation for a parameter variable appears next. The file *TwoArgs2.java*, in this chapter's source code (available at www.pearsonhighered.com/gaddis), has the following method, which uses `@param` tags in its documentation comments:

```
/**
   The showSum method displays the sum of two numbers.
   @param num1 The first number.
   @param num2 The second number.
*/

  public static void showSum(double num1, double num2)
  {
     double sum;     // To hold the sum

     sum = num1 + num2;
     System.out.println("The sum is " + sum);
  }
```

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

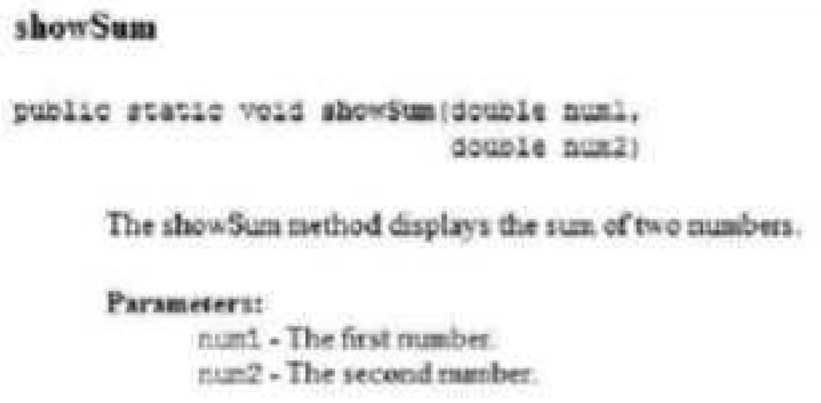
```
@param parameterName Description
```

In the general format, `parameterName` is the name of the parameter and `Description` is a description of the parameter. Remember the following points about `@param` tag comments:

- All `@param` tags 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 one or more `@param` tags, the `javadoc` utility will create a Parameters section in the method's documentation. This is where the descriptions of the method's parameters will be listed. Figure 5-13 shows the documentation generated by `javadoc` for the `showSum` method in the *TwoArgs2.java* file.

Figure 5-13 Documentation for the `showSum` method in *TwoArgs2.java*



showSum

```
public static void showSum(double num1,
                           double num2)
```

The showSum method displays the sum of two numbers.

Parameters:
num1 - The first number.
num2 - The second number.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

5.6 What is the difference between an argument and a parameter?

5.7 Look at the following method header:

```
public static void myMethod(int num)
```

Which of the following calls to the method will cause a compiler error?

a) `myMethod(7);`

b) `myMethod(6.2);`

c)
```
long x = 99;
myMethod(x);
```

d)
```
short s = 2;
myMethod(s);
```

5.8 Suppose a method named `showValues` accepts two `int` arguments. Which of the following method headers is written correctly?

a) `public static void showValues()`

b) `public static void showValues(int num1, num2)`

c) `public static void showValues(num1, num2)`

d) `public static void showValues(int num1, int num2)`

5.9 In Java, method arguments are passed by value. What does this mean?

5.10 What will the following program display?

```
public class Checkpoint
{
   public static void main(String[] args)
   {
      int num1 = 99;
      double num2 = 1.5;

      System.out.println(num1 + " " + num2);
      myMethod(num1, num2);
      System.out.println(num1 + " " + num2);
   }

   public static void myMethod(int i, double d)
   {
      System.out.println(i + " " + d);
      i = 0;
      d = 0.0;
      System.out.println(i + " " + d);
   }
}
```

5.3 More about Local Variables

CONCEPT: A local variable is declared inside a method and is not accessible to statements outside the method. Different methods can have local variables with the same names because the methods cannot see each other's local variables.

In Chapter 2 we introduced the concept of local variables, which are variables that are declared inside a method. They are called *local* because they are local to the method in which they are declared. Statements outside a method cannot access that method's local variables.

Because a method's local variables are hidden from other methods, the other methods may have their own local variables with the same name. For example, look at the program in Code Listing 5-8. In addition to the `main` method, this program has two other methods: `texas` and `california`. These two methods each have a local variable named `birds`.

Code Listing 5-8 (`LocalVars.java`)

```
/**
   This program demonstrates that two methods may have
   local variables with the same name.
*/

public class LocalVars
{
   public static void main(String[] args)
   {
      texas();
      california();
   }

   /**
      The texas method has a local variable named birds.
   */

   public static void texas()
   {
      int birds = 5000;

      System.out.println("In texas there are " +
                         birds + " birds.");
   }

   /**
      The california method also has a local variable named birds.
   */
```

```
    public static void california()
    {
       int birds = 3500;

       System.out.println("In california there are " +
                          birds + " birds.");
    }
}
```

Program Output

```
In texas there are 5000 birds.
In california there are 3500 birds.
```

Although there are two variables named `birds`, the program can see only one of them at a time because they are in different methods. When the `texas` method is executing, the `birds` variable declared inside `texas` is visible. When the `california` method is executing, the `birds` variable declared inside `california` is visible.

Local Variable Lifetime

A method's local variables exist only while the method is executing. This is known as the *lifetime* of a local variable. When the method begins, its local variables and its parameter variables are created in memory, and when the method ends, the local variables and parameter variables are destroyed. This means that any value stored in a local variable is lost between calls to the method in which the variable is declared.

Initializing Local Variables with Parameter Values

It is possible to use a parameter variable to initialize a local variable. Sometimes this simplifies the code in a method. For example, recall the following `showSum` method we discussed earlier:

```
public static void showSum(double num1, double num2)
{
   double sum;    // To hold the sum

   sum = num1 + num2;
   System.out.println("The sum is " + sum);
}
```

In the body of the method, the `sum` variable is declared and then a separate assignment statement assigns `num1 + num2` to `sum`. We can combine these statements into one, as shown in the following modified version of the method.

```
public static void showSum(double num1, double num2)
{
   double sum = num1 + num2;
   System.out.println("The sum is " + sum);
}
```

Because the scope of a parameter variable is the entire method in which it is declared, we can use parameter variables to initialize local variables.

WARNING! Local variables are not automatically initialized with a default value. They must be given a value before they can be used. If you attempt to use a local variable before it has been given a value, a compiler error will result. For example, look at the following method:

```
public static void myMethod()
{
    int x;
    System.out.println(x);    //Error! x has no value.
}
```

This code will cause a compiler error because the variable `x` has not been given a value, and it is being used as an argument to the `System.out.println` method.

5.4 Returning a Value from a Method

CONCEPT: A method may send a value back to the statement that called the method.

You've seen that data may be passed into a method by way of parameter variables. Data may also be returned from a method, back to the statement that called it. Methods that return a value are appropriately known as *value-returning methods*.

VideoNote Returning a Value from a Method

You are already experienced at using value-returning methods. For instance, you have used the wrapper class parse methods, such as `Integer.parseInt`. Here is an example:

```
int num;
num = Integer.parseInt("700");
```

The second line in this code calls the `Integer.parseInt` method, passing "700" as the argument. The method returns the integer value 700, which is assigned to the `num` variable by the `=` operator. You have also seen the `Math.pow` method, which returns a value. Here is an example:

```
double x;
x = Math.pow(4.0, 2.0);
```

The second line in this code calls the `Math.pow` method, passing 4.0 and 2.0 as arguments. The method calculates the value of 4.0 raised to the power of 2.0 and returns that value. The value, which is 16.0, is assigned to the `x` variable by the `=` operator.

In this section we will discuss how you can write your own value-returning methods.

Defining a Value-Returning Method

When you are writing a value-returning method, you must decide what type of value the method will return. This is because you must specify the data type of the return value in the