```
 8  {
 9     public static void main(String[] args)
10    {
11       // Declare a variety of double variables.
12       double num1 = 127.899;
13       double num2 = 3465.148;
14       double num3 = 3.776;
15       double num4 = 264.821;
16       double num5 = 88.081;
17       double num6 = 1799.999;
18
19       // Display each variable in a field of
20       // 8 spaces with 2 decimal places.
21       System.out.printf("%8.2f\n", num1);
22       System.out.printf("%8.2f\n", num2);
23       System.out.printf("%8.2f\n", num3);
24       System.out.printf("%8.2f\n", num4);
25       System.out.printf("%8.2f\n", num5);
26       System.out.printf("%8.2f\n", num6);
27    }
28 }
```

Program Output

```
  127.90
 3465.15
    3.78
  264.82
   88.08
 1800.00
```

Flags

There are several optional flags that you can insert into a format specifier to cause a value to be formatted in a particular way. In this book, we will use flags for the following purposes:

- To display numbers with comma separators
- To pad numbers with leading zeros
- To left-justify numbers

If you use a flag in a format specifier, you must write the flag before the field width and the precision.

Comma Separators

Large numbers are easier to read if they are displayed with comma separators. You can format a number with comma separators by inserting a comma (,) flag into the format specifier. Here is an example:

```
double amount = 1234567.89;
System.out.printf("%,f\n", amount);
```

This code will produce the following output:

```
1,234,567.890000
```

Quite often, you will want to format a number with comma separators, and round the number to a specific number of decimal places. You can accomplish this by inserting a comma, followed by the precision value, into the %f format specifier, as shown in the following example:

```
double sales = 28756.89;
System.out.printf("Sales for the month are %,.2f\n", sales);
```

This code will produce the following output:

```
Sales for the month are 28,756.89
```

Code Listing 3-18 demonstrates how the comma separator and a precision of two decimal places can be used to format a number as a currency amount.

Code Listing 3-18 (CurrencyFormat.java)

```
1  /**
2     This program demonstrates how to use the System.out.printf
3     method to format a number as currency.
4  */
5
6  public class CurrencyFormat
7  {
8     public static void main(String[] args)
9     {
10       double monthlyPay = 5000.0;
11       double annualPay = monthlyPay * 12;
12       System.out.printf("Your annual pay is $%,.2f\n", annualPay);
13    }
14 }
```

Program Output

```
Your annual pay is $60,000.00
```

The following example displays a floating-point number with comma separators, in a field of 15 spaces, rounded to two decimal places:

```
double amount = 1234567.8901;
System.out.printf("%,15.2f\n", amount);
```

This code will produce the following output:

```
   1,234,567.89
```

The following example displays an int with a minimum field width of six characters:

```
int number = 200;
System.out.printf("The number is:%6d", number);
```

This code will display the following:

```
The number is:   200
```

The following example displays an `int` with comma separators, with a minimum field width of 10 characters:

```
int number = 20000;
System.out.printf("The number is:%,10d", number);
```

This code will display the following:

```
The number is:    20,000
```

Padding Numbers with Leading Zeros

Sometimes, when a number is shorter than the field in which it is displayed, you want to pad the number with leading zeros. If you insert a `0` flag into a format specifier, the resulting number will be padded with leading zeros, if it is shorter than the field width. The following code shows an example:

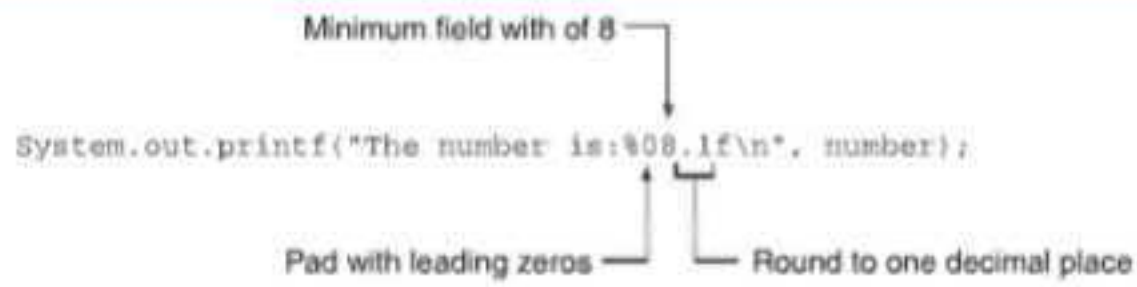
```
double number = 123.4;
System.out.printf("The number is:%08.1f\n", number);
```

This code will produce the following output:

```
The number is:000123.4
```

The diagram in Figure 3-29 shows the purpose of each part of the format specifier in the previous example.

Figure 3-29 Format specifier that pads with leading zeros



The following example displays an `int` padded with leading zeros, with a minimum field width of seven characters:

```
int number = 1234;
System.out.printf("The number is:%07d", number);
```

This code will display the following:

```
The number is:0001234
```

The program in Code Listing 3-19 shows another example. This program displays a variety of floating-point numbers with leading zeros, in a field of nine spaces, rounded to two decimal places.

Code Listing 3-19 (LeadingZeros.java)

```
1  /**
2     This program displays numbers padded with leading zeros.
3  */
4
5  public class LeadingZeros
6  {
7     public static void main(String[] args)
8     {
9        // Declare a variety of double variables.
10       double number1 = 1.234;
11       double number2 = 12.345;
12       double number3 = 123.456;
13
14       // Display each variable with leading
15       // zeros, in a field of 9 spaces, rounded
16       // to 2 decimal places.
17       System.out.printf("%09.2f\n", number1);
18       System.out.printf("%09.2f\n", number2);
19       System.out.printf("%09.2f\n", number3);
20    }
21 }
```

Program Output

```
000001.23
000012.35
000123.46
```

Left-Justifying Numbers

By default, when a number is shorter than the field in which it is displayed, the number is right-justified within that field. If you want a number to be left-justified within its field, you insert a minus sign (-) flag into the format specifier. Code Listing 3-20 shows an example.

Code Listing 3-20 (LeftJustified.java)

```
1  /**
2     This program displays a variety of
3     numbers left-justified in columns.
4  */
5
6  public class LeftJustified
7  {
8     public static void main(String[] args)
9     {
10       // Declare a variety of int variables.
```

```
11        int num1 = 123;
12        int num2 = 12;
13        int num3 = 45678;
14        int num4 = 456;
15        int num5 = 1234567;
16        int num6 = 1234;
17
18        // Display each variable left-justified
19        // in a field of 8 spaces.
20        System.out.printf("%-8d%-8d\n", num1, num2);
21        System.out.printf("%-8d%-8d\n", num3, num4);
22        System.out.printf("%-8d%-8d\n", num5, num6);
23     }
24  }
```

Program Output

```
123     12
45678   456
1234567 1234
```

Formatting String Arguments

If you wish to print a string argument, use the `%s` format specifier. Here is an example:

```
String name = "Ringo";
System.out.printf("Your name is %s\n", name);
```

This code produces the following output:

```
Your name is Ringo
```

You can also use a field width when printing strings. For example, look at the following code:

```
String name1 = "George";
String name2 = "Franklin";
String name3 = "Jay";
String name4 = "Ozzy";
String name5 = "Carmine";
String name6 = "Dee";
System.out.printf("%10s%10s\n", name1, name2);
System.out.printf("%10s%10s\n", name3, name4);
System.out.printf("%10s%10s\n", name5, name6);
```

The `%10s` format specifier prints a string in a field that is ten spaces wide. This code displays the values of the variables in a table with three rows and two columns. Each column has a width of ten spaces. Here is the output of the code:

```
    George  Franklin
       Jay      Ozzy
   Carmine       Dee
```

Notice that the strings are right-justified. You can use the minus flag (-) to left-justify a string within its field. The following code demonstrates:

```
String name1 = "George";
String name2 = "Franklin";
String name3 = "Jay";
String name4 = "Ozzy";
String name5 = "Carmine";
String name6 = "Dee";
System.out.printf("%-10s%-10s\n", name1, name2);
System.out.printf("%-10s%-10s\n", name3, name4);
System.out.printf("%-10s%-10s\n", name5, name6);
```

Here is the output of the code:

```
George    Franklin
Jay       Ozzy
Carmine   Dee
```

The following example shows how you can print arguments of different data types:

```
int hours = 40;
double pay = hours * 25;
String name = "Jay";
System.out.printf("Name: %s, Hours: %d, Pay: $%,.2f\n",
                  name, hours, pay);
```

In this example, we are displaying a `String`, an `int`, and a `double`. The code will produce the following output:

```
Name: Jay, Hours: 40, Pay: $1,000.00
```

NOTE: The format specifiers we have shown in this section are the basic ones. Java provides much more powerful format specifiers for more complex formatting needs. The API documentation gives an overview of them all.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

3.29 Assume the following variable declaration exists in a program:

```
double number = 1234567.456;
```

Write a statement that uses `System.out.printf` to display the value of the `number` variable formatted as:

```
1,234,567.46
```

3.30 Assume the following variable declaration exists in a program:

```
double number = 123.456;
```

Write a statement that uses `System.out.printf` to display the value of the `number` variable rounded to one decimal place, in a field that is 10 spaces wide. (Do not use comma separators.)

3.31 Assume the following variable declaration exists in a program:

```
double number = 123.456;
```

Write a statement that uses `System.out.printf` to display the value of the `number` variable padded with leading zeros, in a field that is eight spaces wide, rounded to one decimal place. (Do not use comma separators.)

3.32 Assume the following variable declaration exists in a program:

```
int number = 123456;
```

Write a statement that uses `System.out.printf` to display the value of the `number` variable in a field that is 10 spaces wide, with comma separators.

3.33 Assume the following variable declaration exists in a program:

```
double number = 123456.789;
```

Write a statement that uses `System.out.printf` to display the value of the `number` variable left-justified, with comma separators, in a field that is 20 spaces wide, rounded to two decimal places.

3.34 Assume the following declaration exists in a program:

```
String name = "James";
```

Write a statement that uses `System.out.printf` to display the value of `name` in a field that is 20 spaces wide.

3.11 Creating Objects with the `DecimalFormat` Class

CONCEPT: The `DecimalFormat` class can be used to format the appearance of floating-point numbers rounded to a specified number of decimal places. It is useful for formatting numbers that will be displayed in message dialogs.

In the previous section you learned how to format console output with the `System.out.printf` method. However, if you want to display formatted output in a graphical interface, such as a message dialog, you will need to use a different approach. In this section we will discuss the `DecimalFormat` class, which can be used to format numbers, regardless of whether they are displayed in the console window, or in a message dialog.

The `DecimalFormat` class is part of the Java API, but it is not automatically available to your programs. To use the `DecimalFormat` class you must have the following `import` statement at the top of your program:

```
import java.text.DecimalFormat;
```

This statement makes the class available to your program. Then, in the part of the program where you want to format a number, you create a `DecimalFormat` object. Here is an example:

```
DecimalFormat formatter = new DecimalFormat("#0.00");
```

Let's dissect the statement into two parts. The first part of the statement is as follows:

```
DecimalFormat formatter =
```

This declares a variable named `formatter`. The data type of the variable is `DecimalFormat`. Because the word `DecimalFormat` is not the name of a primitive data type, Java assumes it to be the name of a class. Recall from Chapter 2 that a variable of a class type is known as a reference

variable, and it is used to hold 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 `formatter` variable will be used to reference a `DecimalFormat` object. The = operator that appears next assigns the address of an object that is created by the second part of the statement as follows:

```
new DecimalFormat("#0.00");
```

This part of the statement uses the key word `new`, which creates an object in memory. After the word `new`, the name `DecimalFormat` appears, followed by some data enclosed in a set of parentheses. The name `DecimalFormat` specifies that an object of the `DecimalFormat` class should be created.

Now let's look at the data appearing inside the parentheses. When an object is created, a special method known as a *constructor* is automatically executed. The purpose of the constructor is to initialize the object's attributes with appropriate data and perform any necessary setup operations. In other words, it constructs the object. The data that appears inside the parentheses is an argument that is passed to the constructor. When you create a `DecimalFormat` object, you pass a string that contains a *formatting pattern* to the constructor. A formatting pattern consists of special characters specifying how numbers should be formatted. In this example the string `"#0.00"` is being passed to the constructor. This string will be assigned to one of the object's internal attributes. After the statement executes, the `formatter` variable will reference the object that was created in memory. This is illustrated in Figure 3-30.

Figure 3-30 The `formatter` variable references a `DecimalFormat` object

Each character in the formatting pattern corresponds with a position in a number. The first two characters, `#0`, correspond to the two digits before the decimal point, the period indicates the decimal point, and the characters `00` correspond to two digits after the decimal point. The `#` character specifies that a digit should be displayed in this position if it is present. If there is no digit in this position, no digit should be displayed. The `0` character also specifies that a digit should be displayed in this position if it is present. However, if there is no digit present in this position, a `0` should be displayed. The two zeros that appear after the decimal point indicate that numbers should be rounded to two decimal places.

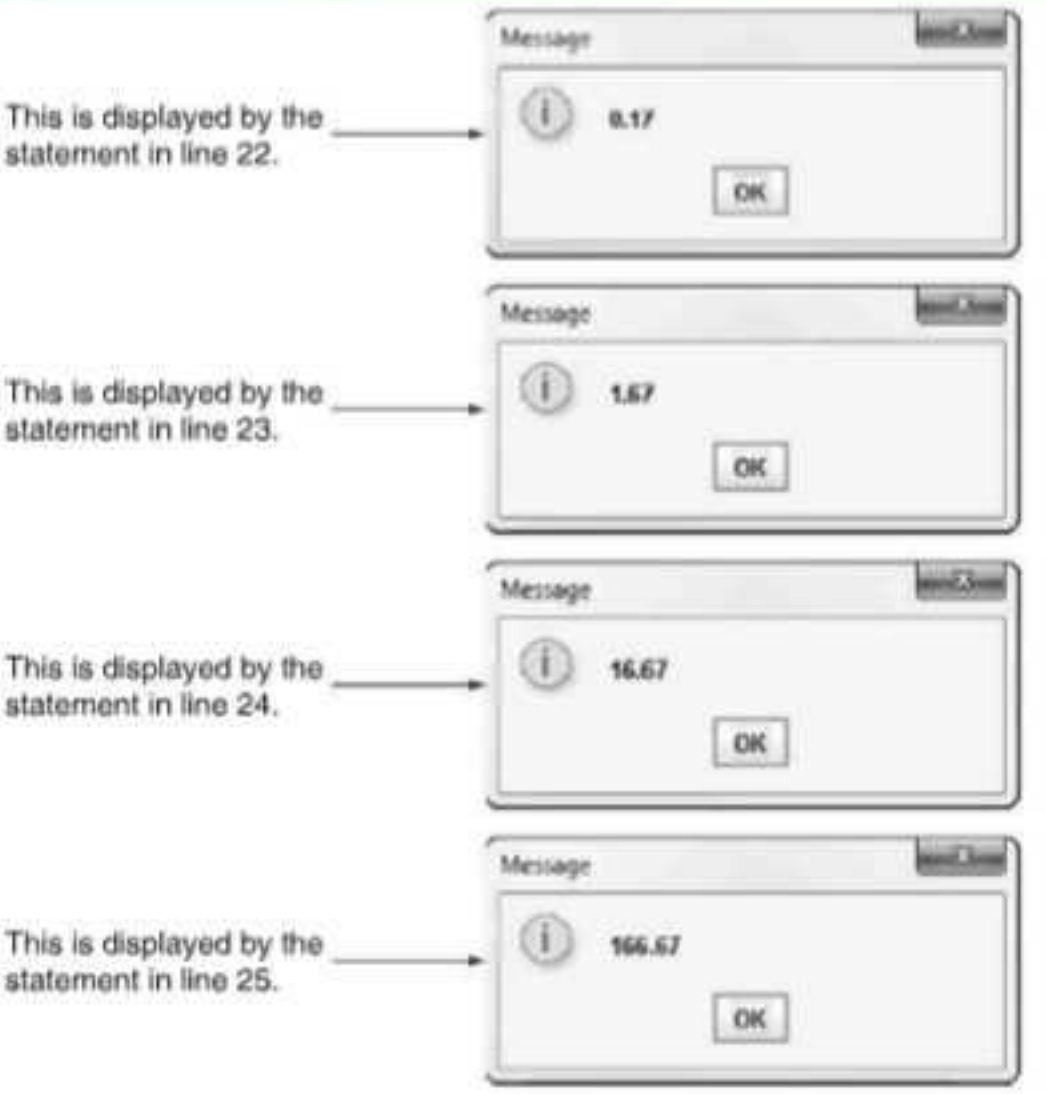
Once you have properly created a `DecimalFormat` object, you call its `format` method and pass the number you wish to format as an argument. (You can pass either a floating-point value or an integer value to the method.) The method returns a string containing the formatted number. For example, look at the program in Code Listing 3-21. The program's output is shown in Figure 3-31.

Code Listing 3-21 (Format1.java)

```
1  import javax.swing.JOptionPane;
2  import java.text.DecimalFormat;
3
4  /**
5     This program uses the DecimalFormat class to display
```

```
 6     formatted numbers in a message dialog.
 7  */
 8
 9  public class Format1
10  {
11     public static void main(String[] args)
12     {
13        double number1 = 0.166666666666667;
14        double number2 = 1.666666666666667;
15        double number3 = 16.666666666666667;
16        double number4 = 166.666666666666667;
17
18        // Create a DecimalFormat object.
19        DecimalFormat formatter = new DecimalFormat("#0.00");
20
21        // Display the formatted variable contents.
22        JOptionPane.showMessageDialog(null, formatter.format(number1));
23        JOptionPane.showMessageDialog(null, formatter.format(number2));
24        JOptionPane.showMessageDialog(null, formatter.format(number3));
25        JOptionPane.showMessageDialog(null, formatter.format(number4));
26     }
27  }
```

Figure 3-31 Output of Code Listing 3-21



Notice the subtle difference between the output of the # character and the 0 character in the formatting pattern:

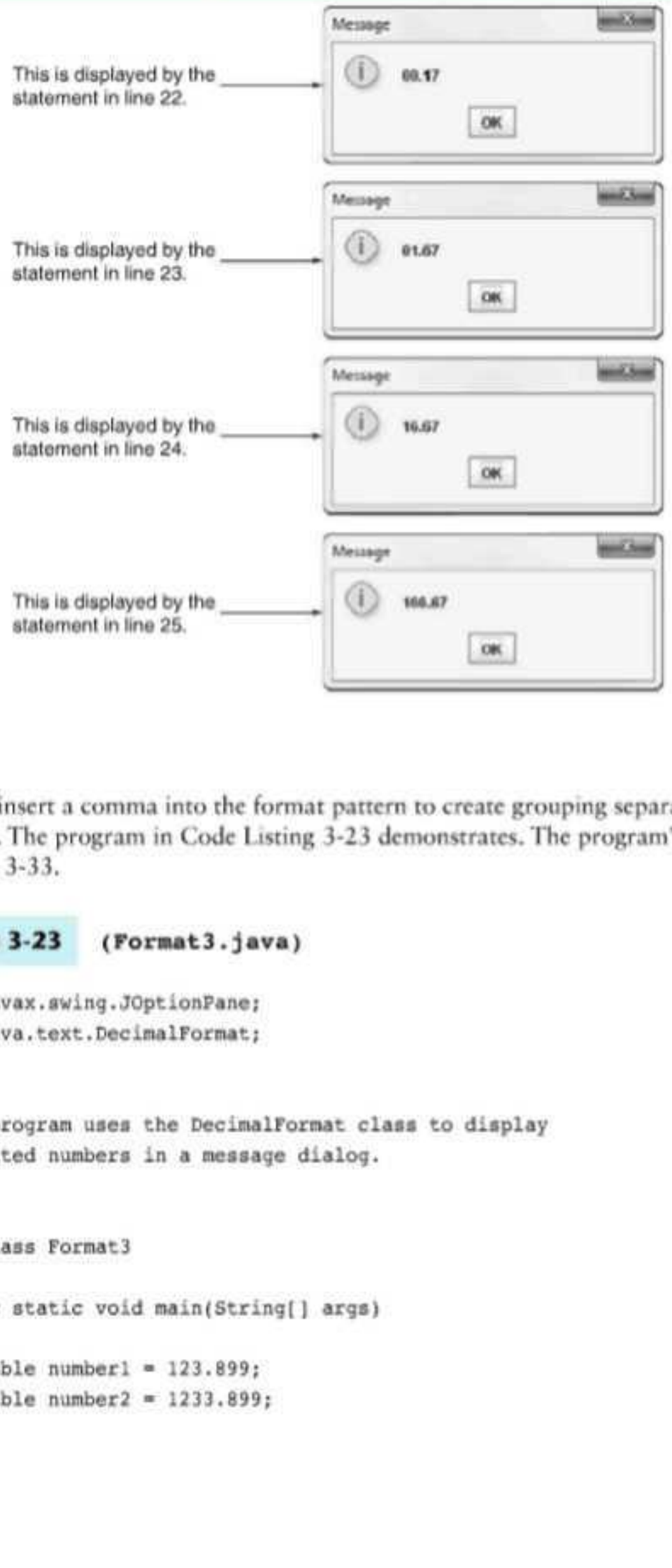
- If the number contains a digit in the position of a # character in the formatting pattern, the digit will be displayed. Otherwise, no digit will be displayed.
- If the number contains a digit in the position of a 0 character in the formatting pattern, the digit will be displayed. Otherwise, a 0 will be displayed.

For example, look at the program in Code Listing 3-22. This is the same program as shown in Code Listing 3-21, but using a different format pattern. The program's output is shown in Figure 3-32.

Code Listing 3-22 (Format2.java)

```
1  import javax.swing.JOptionPane;
2  import java.text.DecimalFormat;
3
4  /**
5     This program uses the DecimalFormat class to display
6     formatted numbers in a message dialog.
7  */
8
9  public class Format2
10 {
11    public static void main(String[] args)
12    {
13       double number1 = 0.166666666666667;
14       double number2 = 1.666666666666667;
15       double number3 = 16.666666666666667;
16       double number4 = 166.666666666666667;
17
18       // Create a DecimalFormat object.
19       DecimalFormat formatter = new DecimalFormat("00.00");
20
21       // Display the formatted variable contents.
22       JOptionPane.showMessageDialog(null, formatter.format(number1));
23       JOptionPane.showMessageDialog(null, formatter.format(number2));
24       JOptionPane.showMessageDialog(null, formatter.format(number3));
25       JOptionPane.showMessageDialog(null, formatter.format(number4));
26    }
27 }
```

Figure 3-32 Output of Code Listing 3-22



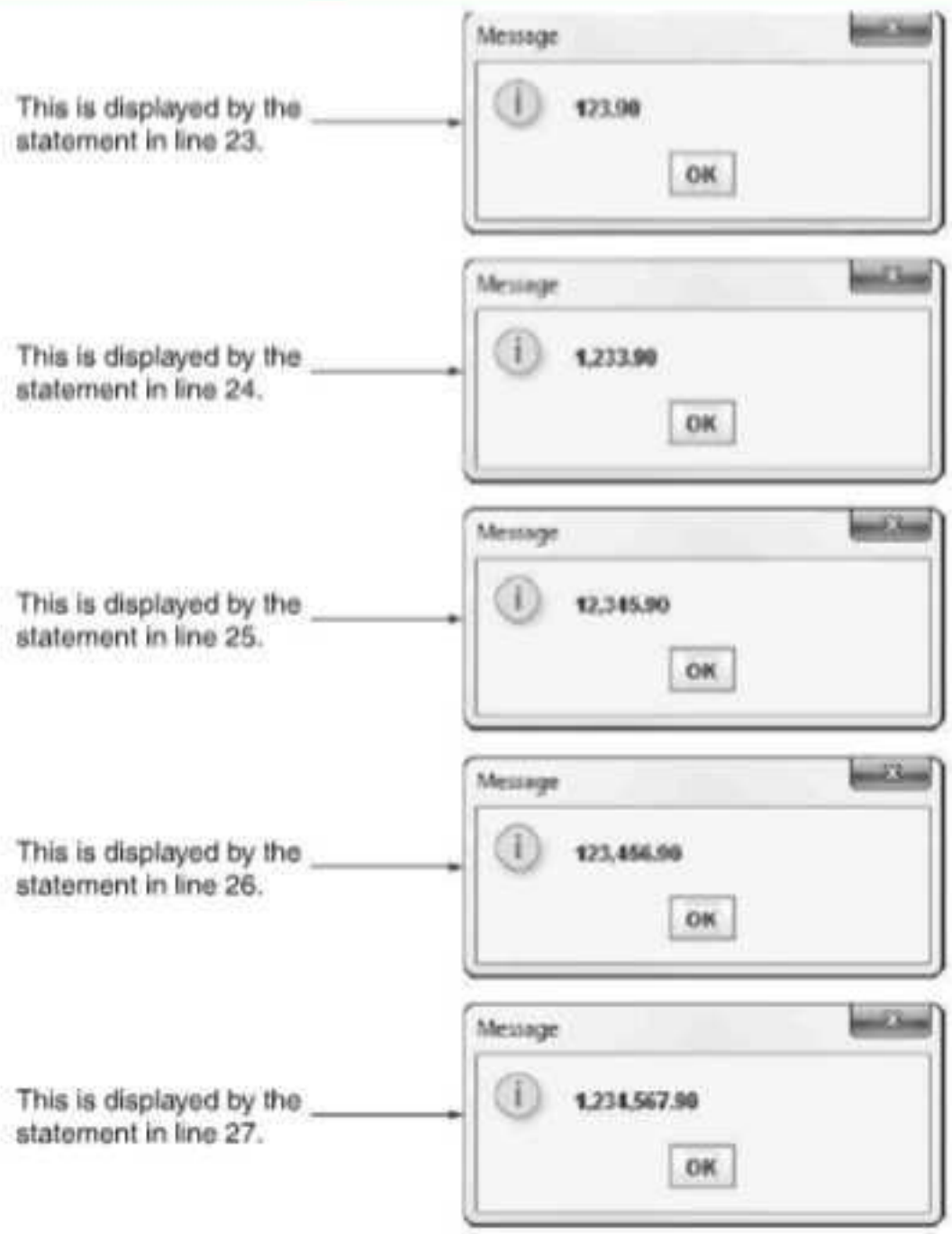
You can insert a comma into the format pattern to create grouping separators in formatted numbers. The program in Code Listing 3-23 demonstrates. The program's output is shown in Figure 3-33.

Code Listing 3-23 (Format3.java)

```
1  import javax.swing.JOptionPane;
2  import java.text.DecimalFormat;
3
4  /**
5     This program uses the DecimalFormat class to display
6     formatted numbers in a message dialog.
7  */
8
9  public class Format3
10 {
11    public static void main(String[] args)
12    {
13       double number1 = 123.899;
14       double number2 = 1233.899;
```

```
15          double number3 = 12345.899;
16          double number4 = 123456.899;
17          double number5 = 1234567.899;
18
19          // Create a DecimalFormat object.
20          DecimalFormat formatter = new DecimalFormat("#,##0.00");
21
22          // Display the formatted variable contents.
23          JOptionPane.showMessageDialog(null, formatter.format(number1));
24          JOptionPane.showMessageDialog(null, formatter.format(number2));
25          JOptionPane.showMessageDialog(null, formatter.format(number3));
26          JOptionPane.showMessageDialog(null, formatter.format(number4));
27          JOptionPane.showMessageDialog(null, formatter.format(number5));
28      }
29  }
```

Figure 3-33 Output of Code Listing 3-23

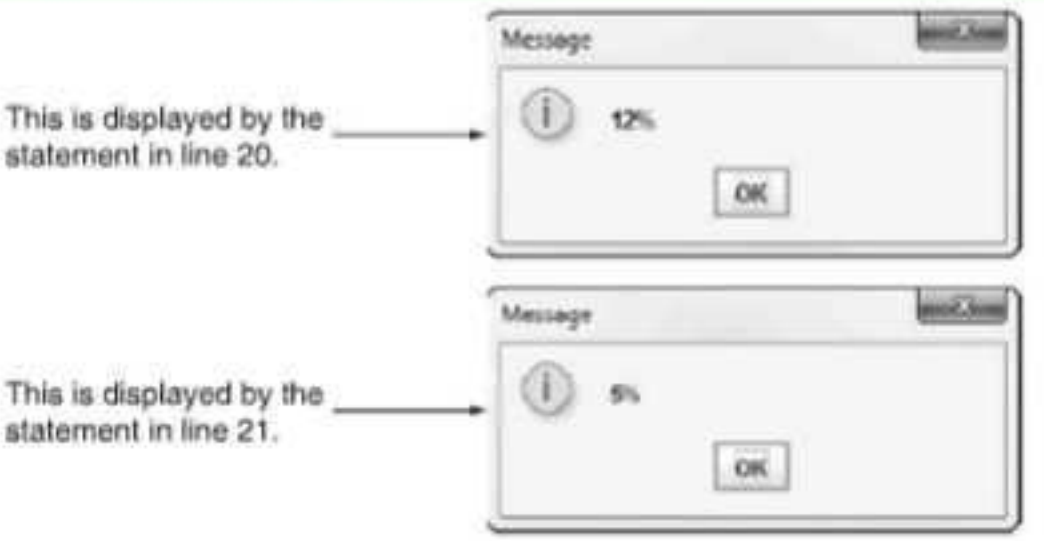


You can also format numbers as percentages by writing the % character at the last position in the format pattern. This causes a number to be multiplied by 100, and the % character is appended to its end. The program in Code Listing 3-24 demonstrates. The program's output is shown in Figure 3-34.

Code Listing 3-24 (Format4.java)

```
1  import javax.swing.JOptionPane;
2  import java.text.DecimalFormat;
3
4  /**
5     This program uses the DecimalFormat class to display
6     formatted numbers in a message dialog.
7  */
8
9  public class Format4
10 {
11    public static void main(String[] args)
12    {
13       double number1 = 0.12;
14       double number2 = 0.05;
15
16       // Create a DecimalFormat object.
17       DecimalFormat formatter = new DecimalFormat("#0%");
18
19       // Display the formatted variable contents.
20       JOptionPane.showMessageDialog(null, formatter.format(number1));
21       JOptionPane.showMessageDialog(null, formatter.format(number2));
22    }
23 }
```

Figure 3-34 Output of Code Listing 3-24



Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

3.35 Assume that the `double` variable `number` holds the value 459.6329. What format pattern would you use with a `DecimalFormat` object to display the number as 00459.633?

3.36 Assume that the `double` variable `number` holds the value 0.179. What format pattern would you use with a `DecimalFormat` object to display the number as .18?

3.37 Assume that the `double` variable `number` holds the value 7634869.1. What format pattern would you use with a `DecimalFormat` object to display the number as 7,634,869.10?

CASE STUDY: This book's companion Web site (www.pearsonhighered.com/gaddis) provides a valuable case study, Calculating Sales Commission, that demonstrates how sales commissions may be calculated using various commission rates. The commission rate is determined by the amount of sales.

3.12 Common Errors to Avoid

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

- **Using = instead of == to compare primitive values.** Remember, = is the assignment operator and == tests for equality.
- **Using == instead of the `equals` method to compare `String` objects.** You cannot use the == operator to compare the contents of a `String` object with another string. Instead you must use the `equals` or `compareTo` method.
- **Forgetting to enclose an `if` statement's `boolean` expression in parentheses.** Java requires that the `boolean` expression being tested by an `if` statement is enclosed in a set of parentheses. An error will result if you omit the parentheses or use any other grouping characters.
- **Writing a semicolon at the end of an `if` clause.** When you write a semicolon at the end of an `if` clause, Java assumes that the conditionally executed statement is a null or empty statement.
- **Forgetting to enclose multiple conditionally executed statements in braces.** Normally the `if` statement conditionally executes only one statement. To conditionally execute more than one statement, you must enclose them in braces.
- **Omitting the trailing `else` in an `if-else-if` statement.** This is not a syntax error, but can lead to logical errors. If you omit the trailing `else` from an `if-else-if` statement, no code will be executed if none of the statement's `boolean` expressions are `true`.
- **Not writing complete `boolean` expressions on both sides of a logical `&&` or `||` operator.** You must write a complete `boolean` expression on both sides of a logical `&&` or `||` operator. For example, the expression `x > 0 && < 10` is not valid because `< 10` is not a complete expression. The expression should be written as `x > 0 && x < 10`.
- **Trying to perform case-insensitive string comparisons with the `String` class's `equals` and `compareTo` methods.** To perform case-insensitive string comparisons, use the `String` class's `equalsIgnoreCase` and `compareToIgnoreCase` methods.
- **Using a `SwitchExpression` that is not an `int`, `short`, `byte`, or `char`.** The `switch` statement can only evaluate expressions that are of the `int`, `short`, `byte`, or `char` data types.

- **Using a `CaseExpression` that is not a literal or a `final` variable.** Because the compiler must determine the value of a `CaseExpression` at compile time, `CaseExpressions` must be either literal values or `final` variables.
- **Forgetting to write a colon at the end of a `case` statement.** A colon must appear after the `CaseExpression` in each `case` statement.
- **Forgetting to write a `break` statement in a `case` section.** This is not a syntax error, but it can lead to logical errors. The program does not branch out of a `switch` statement until it reaches a `break` statement or the end of the `switch` statement.
- **Forgetting to write a `default` section in a `switch` statement.** This is not a syntax error, but can lead to a logical error. If you omit the `default` section, no code will be executed if none of the `CaseExpressions` match the `SwitchExpression`.
- **Reversing the `?` and the `:` when using the conditional operator.** When using the conditional operator, the `?` character appears first in the conditional expression, then the `:` character.
- When formatting a number with `System.out.printf`, writing the flags, field width, and precision in an incorrect order.
- When writing a format specifier for the `System.out.printf` method, using the wrong type indicator (`%f` = floating-point, `%d` = integer, `%s` = string).
- **Forgetting to pass a formatting string to a `DecimalFormat` object's constructor.** The formatting string specifies how the object should format any numbers that are passed to the `format` method.

Review Questions and Exercises

Multiple Choice and True/False

1. The `if` statement is an example of a _____.
   a. sequence structure
   b. decision structure
   c. pathway structure
   d. class structure
2. This type of expression has a value of either `true` or `false`.
   a. `binary` expression
   b. `decision` expression
   c. `unconditional` expression
   d. `boolean` expression
3. `>`, `<`, and `==` are _____.
   a. relational operators
   b. logical operators
   c. conditional operators
   d. ternary operators
4. `&&`, `||`, and `!` are _____.
   a. relational operators
   b. logical operators
   c. conditional operators
   d. ternary operators

5. This is an empty statement that does nothing.
   a. missing statement
   b. virtual statement
   c. `null` statement
   d. conditional statement
6. To create a block of statements, you enclose the statements in these.
   a. parentheses`()`
   b. square brackets `[]`
   c. angled brackets `<>`
   d. braces `{}`
7. This is a `boolean` variable that signals when some condition exists in the program.
   a. flag
   b. signal
   c. sentinel
   d. siren
8. How does the character 'A' compare to the character 'B'?
   a. 'A' is greater than 'B'
   b. 'A' is less than 'B'
   c. 'A' is equal to 'B'
   d. You cannot compare characters
9. This is an `if` statement that appears inside another `if` statement.
   a. nested `if` statement
   b. tiered `if` statement
   c. dislodged `if` statement
   d. structured `if` statement
10. An `else` clause always goes with _____.
    a. the closest previous `if` clause that doesn't already have its own `else` clause
    b. the closest `if` clause
    c. the `if` clause that is randomly selected by the compiler
    d. none of these
11. When determining whether a number is inside a range, it's best to use this operator.
    a. `&&`
    b. `!`
    c. `||`
    d. `? :`
12. This determines whether two different `String` objects contain the same string.
    a. the `==` operator
    b. the `=` operator
    c. the `equals` method
    d. the `stringCompare` method
13. The conditional operator takes this many operands.
    a. one
    b. two
    c. three
    d. four

14. This section of a `switch` statement is branched to if none of the `case` expressions match the `switch` expression.
    a. `else`
    b. `default`
    c. `case`
    d. `otherwise`
15. You can use this method to display formatted output in a console window.
    a. `Format.out.println`
    b. `Console.format`
    c. `System.out.printf`
    d. `System.out.formatted`
16. **True or False:** The = operator and the == operator perform the same operation.
17. **True or False:** A conditionally executed statement should be indented one level from the `if` clause.
18. **True or False:** All lines in a conditionally executed block should be indented one level.
19. **True or False:** When an `if` statement is nested in the `if` clause of another statement, the only time the inner `if` statement is executed is when the `boolean` expression of the outer `if` statement is `true`.
20. **True or False:** When an `if` statement is nested in the `else` clause of another statement, the only time the inner `if` statement is executed is when the `boolean` expression of the outer `if` statement is `true`.
21. **True or False:** The scope of a variable is limited to the block in which it is defined.

Find the Error

Find the errors in the following code:

1.
```
// Warning! This code contains ERRORS!
if (x == 1);
   y = 2;
else if (x == 2);
   y = 3;
else if (x == 3);
   y = 4;
```

2.
```
// Warning! This code contains an ERROR!
if (average = 100)
   System.out.println("Perfect Average!");
```

3.
```
// Warning! This code contains ERRORS!
if (num2 == 0)
   System.out.println("Division by zero is not possible.");
   System.out.println("Please run the program again ");
   System.out.println("and enter a number besides zero.");
else
   Quotient = num1 / num2;
   System.out.print("The quotient of " + Num1);
   System.out.print(" divided by " + Num2 + " is ");
   System.out.println(Quotient);
```

4.
```
// Warning! This code contains ERRORS!
switch (score)
{
   case (score > 90):
      grade = 'A';
      break;
   case(score > 80):
      grade = 'b';
      break;
   case(score > 70):
      grade = 'C';
      break;
   case (score > 60):
      grade = 'D';
      break;
   default:
      grade = 'F';
}
```

5. The following statement should determine whether `x` is not greater than 20. What is wrong with it?
```
if (!x > 20)
```

6. The following statement should determine whether `count` is within the range of 0 through 100. What is wrong with it?
```
if (count >= 0 || count <= 100)
```

7. The following statement should determine whether `count` is outside the range of 0 through 100. What is wrong with it?
```
if (count < 0 && count > 100)
```

8. The following statement should assign 0 to `z` if `a` is less than 10; otherwise, it should assign 7 to `z`. What is wrong with it?
```
z = (a < 10) : 0 ? 7;
```

9. Assume that `partNumber` references a `String` object. The following `if` statement should perform a case-insensitive comparison. What is wrong with it?
```
if (partNumber.equals("BQ789W4"))
   available = true;
```

10. What is wrong with the following code?
```
double value = 12345.678;
System.out.printf("%.2d", value);
```

Algorithm Workbench

1. Write an `if` statement that assigns 100 to `x` when `y` is equal to 0.
2. Write an `if-else` statement that assigns 0 to `x` when `y` is equal to 10. Otherwise, it should assign 1 to `x`.

3. Using the following chart, write an `if-else-if` statement that assigns .10, .15, or .20 to `commission`, depending on the value in `sales`.

Sales	Commission Rate
Up to $10,000	10%
$10,000 to $15,000	15%
Over $15,000	20%

4. Write an `if` statement that sets the variable `hours` to 10 when the `boolean` flag variable `minimum` is equal to `true`.
5. Write nested `if` statements that perform the following tests: If `amount1` is greater than 10 and `amount2` is less than 100, display the greater of the two.
6. Write an `if` statement that prints the message "The number is valid" if the variable `grade` is within the range 0 through 100.
7. Write an `if` statement that prints the message "The number is valid" if the variable `temperature` is within the range –50 through 150.
8. Write an `if` statement that prints the message "The number is not valid" if the variable `hours` is outside the range 0 through 80.
9. Write an `if-else` statement that displays the `String` objects `title1` and `title2` in alphabetical order.
10. Convert the following `if-else-if` statement into a `switch` statement:

```
if (choice == 1)
{
   System.out.println("You selected 1.");
}
else if (choice == 2 || choice == 3)
{
   System.out.println("You selected 2 or 3.");
}
else if (choice == 4)
{
     System.out.println("You selected 4.");
}
else
{
     System.out.println("Select again please.");
}
```

11. Match the conditional expression with the `if-else` statement that performs the same operation.

a. `q = x < y ? a + b : x * 2;`
b. `q = x < y ? x * 2 : a + b;`
c. `q = x < y ? 0 : 1;`

____

```
if (x < y)
   q = 0;
else
   q = 1;
```

```
____    if (x < y)
           q = a + b;
        else
           q = x * 2;
____    if (x < y)
           q = x * 2;
        else
           q = a + b;
```

12. Assume the `double` variable `number` contains the value 12345.6789. Write a statement that uses `System.out.printf` to display the number as 12345.7.
13. Assume the `double` variable `number` contains the value 12345.6789. Write a statement that uses `System.out.printf` to display the number as 12,345.68.
14. Assume the `int` variable `number` contains the value 1234567. Write a statement that uses `System.out.printf` to display the number as 1,234,567.
15. Assume that the `double` variable `number` holds the value 0.0329. What format pattern would you use with the `DecimalFormat` class to display the number as 00000.033?
16. Assume that the `double` variable `number` holds the value 456198736.3382. What format pattern would you use with the `DecimalFormat` class to display the number as 456,198,736.34?

Short Answer

1. Explain what is meant by the phrase "conditionally executed."
2. Explain why a misplaced semicolon can cause an `if` statement to operate incorrectly.
3. Why is it good advice to indent all the statements inside a set of braces?
4. What happens when you compare two `String` objects with the `==` operator?
5. Explain the purpose of a flag variable. Of what data type should a flag variable be?
6. What risk does a programmer take when not placing a trailing `else` at the end of an `if-else-if` statement?
7. Briefly describe how the `&&` operator works.
8. Briefly describe how the `||` operator works.
9. Why are the relational operators called "relational"?
10. When does a constructor execute? What is its purpose?

Programming Challenges

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

1. Roman Numerals

Write a program that prompts the user to enter a number within the range of 1 through 10. The program should display the Roman numeral version of that number. If the number is outside the range of 1 through 10, the program should display an error message.

2. Magic Dates

The date June 10, 1960, is special because when we write it in the following format, the month times the day equals the year:

6/10/60

Write a program that asks the user to enter a month (in numeric form), a day, and a two-digit year. The program should then determine whether the month times the day is equal to the year. If so, it should display a message saying the date is magic. Otherwise, it should display a message saying the date is not magic.

3. Body Mass Index

Write a program that calculates and displays a person's body mass index (BMI). The BMI is often used to determine whether a person with a sedentary lifestyle is overweight or underweight for his or her height. A person's BMI is calculated with the following formula:

$$BMI = Weight \times 703 / Height^2$$

where *weight* is measured in pounds and *height* is measured in inches. The program should display a message indicating whether the person has optimal weight, is underweight, or is overweight. A sedentary person's weight is considered optimal if his or her BMI is between 18.5 and 25. If the BMI is less than 18.5, the person is considered underweight. If the BMI value is greater than 25, the person is considered overweight.

4. Test Scores and Grade

Write a program that has variables to hold three test scores. The program should ask the user to enter three test scores and then assign the values entered to the variables. The program should display the average of the test scores and the letter grade that is assigned for the test score average. Use the grading scheme in the following table:

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

5. Mass and Weight

Scientists measure an object's mass in kilograms and its weight in Newtons. If you know the amount of mass that an object has, you can calculate its weight, in Newtons, with the following formula:

$$Weight = Mass \times 9.8$$

Write a program that asks the user to enter an object's mass, and then calculate its weight. If the object weighs more than 1,000 Newtons, display a message indicating that it is too heavy. If the object weighs less than 10 Newtons, display a message indicating that the object is too light.

6. Time Calculator

The Time Calculator Problem

Write a program that asks the user to enter a number of seconds.

- There are 60 seconds in a minute. If the number of seconds entered by the user is greater than or equal to 60, the program should display the number of minutes in that many seconds.
- There are 3,600 seconds in an hour. If the number of seconds entered by the user is greater than or equal to 3,600, the program should display the number of hours in that many seconds.
- There are 86,400 seconds in a day. If the number of seconds entered by the user is greater than or equal to 86,400, the program should display the number of days in that many seconds.

7. Sorted Names

Write a program that asks the user to enter three names, and then displays the names sorted in ascending order. For example, if the user entered "Charlie", "Leslie", and "Andy", the program would display:

```
Andy
Charlie
Leslie
```

8. Software Sales

A software company sells a package that retails for $99. Quantity discounts are given according to the following table:

Quantity	Discount
10–19	20%
20–49	30%
50–99	40%
100 or more	50%

Write a program that asks the user to enter the number of packages purchased. The program should then display the amount of the discount (if any) and the total amount of the purchase after the discount.

9. Shipping Charges

The Fast Freight Shipping Company charges the following rates:

Weight of Package	Rate per 500 Miles Shipped
2 pounds or less	$1.10
Over 2 pounds but not more than 6 pounds	$2.20
Over 6 pounds but not more than 10 pounds	$3.70
Over 10 pounds	$3.80

The shipping charges per 500 miles are not prorated. For example, if a 2-pound package is shipped 550 miles, the charges would be $2.20. Write a program that asks the user to enter the weight of a package and then displays the shipping charges.

10. Fat Gram Calculator

Write a program that asks the user to enter the number of calories and fat grams in a food item. The program should display the percentage of the calories that come from fat. One gram of fat has 9 calories; therefore:

*Calories from fat = Fat grams * 9*

The percentage of calories from fat can be calculated as follows:

Calories from fat ÷ Total calories

If the calories from fat are less than 30 percent of the total calories of the food, it should also display a message indicating the food is low in fat.

NOTE: The number of calories from fat cannot be greater than the total number of calories in the food item. If the program determines that the number of calories from fat is greater than the number of calories in the food item, it should display an error message indicating that the input is invalid.

11. Running the Race

Write a program that asks for the names of three runners and the time, in minutes, it took each of them to finish a race. The program should display the names of the runners in the order that they finished.

12. The Speed of Sound

The following table shows the approximate speed of sound in air, water, and steel:

Medium	Speed
Air	1,100 feet per second
Water	4,900 feet per second
Steel	16,400 feet per second

Write a program that asks the user to enter "air", "water", or "steel", and the distance that a sound wave will travel in the medium. The program should then display the amount of time it will take. You can calculate the amount of time it takes sound to travel in air with the following formula:

Time = Distance / 1,100

You can calculate the amount of time it takes sound to travel in water with the following formula:

Time = Distance / 4,900

You can calculate the amount of time it takes sound to travel in steel with the following formula:

Time = *Distance* / 16,400

13. Internet Service Provider

An Internet service provider has three different subscription packages for its customers:

Package A:	For $9.95 per month 10 hours of access are provided. Additional hours are $2.00 per hour.
Package B:	For $13.95 per month 20 hours of access are provided. Additional hours are $1.00 per hour.
Package C:	For $19.95 per month unlimited access is provided.

Write a program that calculates a customer's monthly bill. It should ask the user to enter the letter of the package the customer has purchased (A, B, or C) and the number of hours that were used. It should then display the total charges.

14. Internet Service Provider, Part 2

Modify the program you wrote for Programming Challenge 13 so it also calculates and displays the amount of money Package A customers would save if they purchased Package B or C, and the amount of money Package B customers would save if they purchased Package C. If there would be no savings, no message should be printed.

15. Bank Charges

A bank charges a base fee of $10 per month, plus the following check fees for a commercial checking account:

$.10 each for less than 20 checks
$.08 each for 20–39 checks
$.06 each for 40–59 checks
$.04 each for 60 or more checks

Write a program that asks for the number of checks written for the month. The program should then calculate and display the bank's service fees for the month.

16. Book Club Points

Serendipity Booksellers has a book club that awards points to its customers based on the number of books purchased each month. The points are awarded as follows:

- If a customer purchases 0 books, he or she earns 0 points.
- If a customer purchases 1 book, he or she earns 5 points.
- If a customer purchases 2 books, he or she earns 15 points.
- If a customer purchases 3 books, he or she earns 30 points.
- If a customer purchases 4 or more books, he or she earns 60 points.

Write a program that asks the user to enter the number of books that he or she has purchased this month and then displays the number of points awarded.

CHAPTER

4 Loops and Files

TOPICS

4.1 The Increment and Decrement Operators

CONCEPT: ++ and -- are operators that add and subtract one from their operands.

To *increment* a value means to increase it by one, and to *decrement* a value means to decrease it by one. Both of the following statements increment the variable `number`:

```
number = number + 1;
number += 1;
```

And `number` is decremented in both of the following statements:

```
number = number - 1;
number -= 1;
```

Java provides a set of simple unary operators designed just for incrementing and decrementing variables. The increment operator is ++ and the decrement operator is --. The following statement uses the ++ operator to increment `number`:

```
number++;
```

And the following statement decrements `number`:

```
number--;
```

NOTE: The expression `number++` is pronounced "number plus plus," and `number--` is pronounced "number minus minus."

The program in Code Listing 4-1 demonstrates the ++ and -- operators.

Code Listing 4-1 (IncrementDecrement.java)

```
1  /**
2     This program demonstrates the ++ and -- operators.
3  */
4
5  public class IncrementDecrement
6  {
7     public static void main(String[] args)
8     {
9        int number = 4; // number starts out with 4
10
11       // Display the value in number.
12       System.out.println("number is " + number);
13       System.out.println("I will increment number.");
14
15       // Increment number.
16       number++;
17
18       // Display the value in number again.
19       System.out.println("Now, number is " + number);
20       System.out.println("I will decrement number.");
21
22       // Decrement number.
23       number--;
24
25       // Display the value in number once more.
26       System.out.println("Now, number is " + number);
27    }
28 }
```

Program Output

```
number is 4
I will increment number.
Now, number is 5
I will decrement number.
Now, number is 4
```

The statements in Code Listing 4-1 show the increment and decrement operators used in *postfix mode*, which means the operator is placed after the variable. The operators also work in *prefix mode*, where the operator is placed before the variable name as follows:

```
++number;

--number;
```

In both postfix and prefix mode, these operators add one to or subtract one from their operand. Code Listing 4-2 demonstrates this.

Code Listing 4-2 (Prefix.java)

```
1  /**
2     This program demonstrates the ++ and -- operators
3     in prefix mode.
4  */
5
6  public class Prefix
7  {
8     public static void main(String[] args)
9     {
10       int number = 4; // number starts out with 4
11
12       // Display the value in number.
13       System.out.println("number is " + number);
14       System.out.println("I will increment number.");
15
16       // Increment number.
17       ++number;
18
19       // Display the value in number again.
20       System.out.println("Now, number is " + number);
21       System.out.println("I will decrement number.");
22
23       // Decrement number.
24       --number;
25
26       // Display the value in number once again.
27       System.out.println("Now, number is " + number);
28    }
29 }
```

Program Output

```
number is 4
I will increment number.
Now, number is 5
I will decrement number.
Now, number is 4
```

The Difference between Postfix and Prefix Modes

In Code Listings 4-1 and 4-2, the statements `number++` and `++number` increment the variable `number`, while the statements `number--` and `--number` decrement the variable `number`. In these simple statements it doesn't matter whether the operator is used in postfix or prefix mode. The difference is important, however, when these operators are used in statements that do more than just increment or decrement. For example, look at the following code:

```
number = 4;
System.out.println(number++);
```

The statement that calls the `println` method does two things: (1) calls `println` to display the value of `number`, and (2) increments `number`. But which happens first? The `println` method will display a different value if `number` is incremented first than if `number` is incremented last. The answer depends upon the mode of the increment operator.

Postfix mode causes the increment to happen after the value of the variable is used in the expression. In the previously shown statement, the `println` method will display 4 and then `number` will be incremented to 5. Prefix mode, however, causes the increment to happen first. Here is an example:

```
number = 4;
System.out.println(++number);
```

In these statements, `number` is incremented to 5, then `println` will display the value in `number` (which is 5). For another example, look at the following code:

```
int x = 1, y;
y = x++;            // Postfix increment
```

The first statement declares the variable `x` (initialized with the value 1) and the variable `y`. The second statement does the following:

- It assigns the value of `x` to the variable `y`.
- The variable `x` is incremented.

The value that will be stored in `y` depends on when the increment takes place. Because the `++` operator is used in postfix mode, it acts after the assignment takes place. So, this code will store 1 in `y`. After the code has executed, `x` will contain 2. Let's look at the same code, but with the `++` operator used in prefix mode as follows:

```
int x = 1, y;
y = ++x;            // Prefix increment
```

The first statement declares the variable `x` (initialized with the value 1) and the variable `y`. In the second statement, the `++` operator is used in prefix mode, so it acts on the variable before the assignment takes place. So, this code will store 2 in `y`. After the code has executed, `x` will also contain 2.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

4.1 What will the following program segments display?

a)
```
x = 2;
y = x++;
System.out.println(y);
```

b)
```
x = 2;
System.out.println(x++);
```

c)
```
x = 2;
System.out.println(--x);
```

d)
```
x = 8;
y = x--;
System.out.println(y);
```

4.2 The while Loop

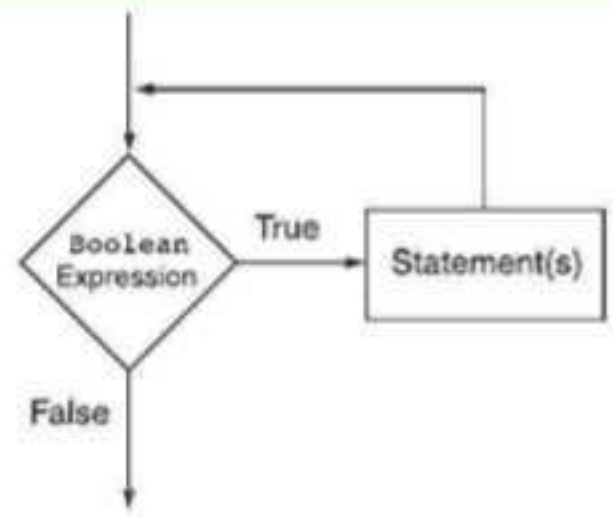
CONCEPT: A loop is part of a program that repeats.

In Chapter 3, you were introduced to the concept of control structures, which direct the flow of a program. A *loop* is a control structure that causes a statement or group of statements to repeat. Java has three looping control structures: the `while` loop, the `do-while` loop, and the `for` loop. The difference among each of these is how they control the repetition. In this section we will focus on the `while` loop.

VideoNote
The while Loop

The `while` loop has two important parts: (1) a `boolean` expression that is tested for a `true` or `false` value, and (2) a statement or block of statements that is repeated as long as the expression is `true`. Figure 4-1 shows the logic of a `while` loop.

Figure 4-1 Logic of a `while` loop



Here is the general format of the `while` loop:

```
while (BooleanExpression)
   Statement;
```

In the general format, *BooleanExpression* is any valid `boolean` expression, and *Statement* is any valid Java statement. The first line shown in the format is sometimes called the *loop header*. It consists of the key word `while` followed by the *BooleanExpression* enclosed in parentheses.

Here's how the loop works: The *BooleanExpression* is tested, and if it is `true`, the *Statement* is executed. Then, the *BooleanExpression* is tested again. If it is `true`, the *Statement* is executed. This cycle repeats until the *BooleanExpression* is `false`.

The statement that is repeated is known as the *body* of the loop. It is also considered a conditionally executed statement, because it is only executed under the condition that the *BooleanExpression* is `true`.

Notice there is no semicolon at the end of the loop header. Like the `if` statement, the `while` loop is not complete without the conditionally executed statement that follows it.

If you wish the `while` loop to repeat a block of statements, the format is as follows:

```
while (BooleanExpression)
{
   Statement;
   Statement;
   // Place as many statements here
   // as necessary.
}
```

The `while` loop works like an `if` statement that executes over and over. As long as the expression in the parentheses is `true`, the conditionally executed statement or block will repeat. The program in Code Listing 4-3 uses the `while` loop to print "Hello" five times.

Code Listing 4-3 (WhileLoop.java)

```
1  /**
2     This program demonstrates the while loop.
3  */
4
5  public class WhileLoop
6  {
7     public static void main(String[] args)
8     {
9        int number = 1;
10
11       while (number <= 5)
12       {
13          System.out.println("Hello");
14          number++;
15       }
```

```
16
17            System.out.println("That's all!");
18      }
19  }
```

Program Output

```
Hello
Hello
Hello
Hello
Hello
That's all!
```

Let's take a closer look at this program. An integer variable, `number`, is declared and initialized with the value 1. The `while` loop begins with the following statement:

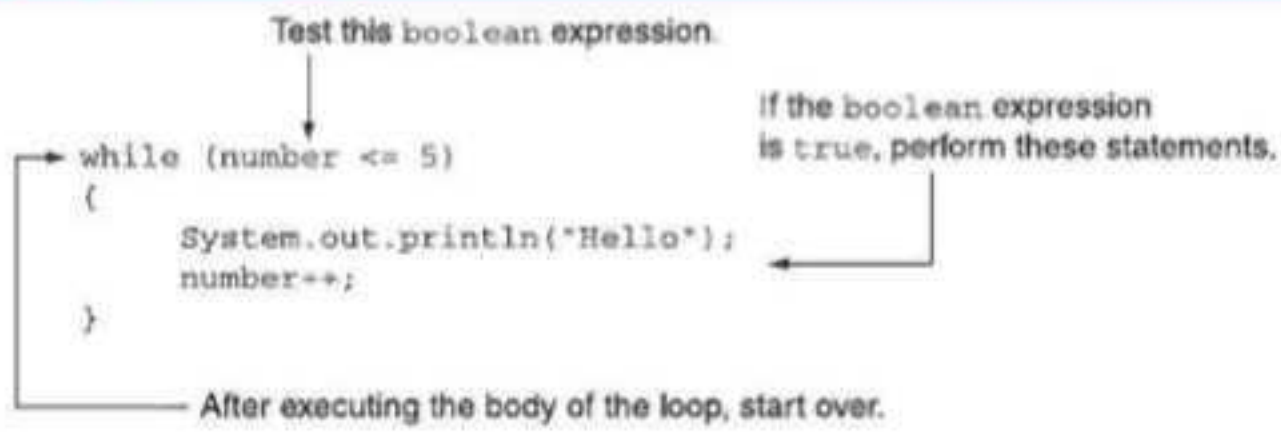
```
while (number <= 5)
```

This statement tests the variable `number` to determine whether it is less than or equal to 5. If it is, then the statements in the body of the loop are executed as follows:

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

The first statement in the body of the loop prints the word "Hello". The second statement uses the increment operator to add one to `number`. This is the last statement in the body of the loop, so after it executes, the loop starts over. It tests the `boolean` expression again, and if it is `true`, the statements in the body of the loop are executed. This cycle repeats until the `boolean` expression `number <= 5` is **`false`**, as illustrated in Figure 4-2.

Figure 4-2 The `while` Loop

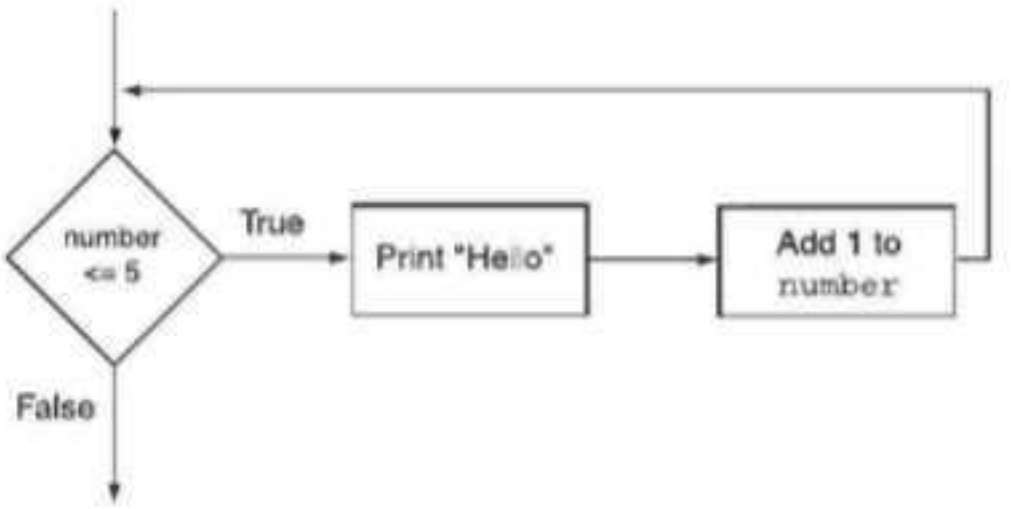


Each repetition of a loop is known as an *iteration*. This loop will perform five iterations because the variable `number` is initialized with the value 1, and it is incremented each time the body of the loop is executed. When the expression `number <= 5` is tested and found to be

`false`, the loop will terminate and the program will resume execution at the statement that immediately follows the loop. Figure 4-3 shows the logic of this loop.

In this example, the `number` variable is referred to as the *loop control variable* because it controls the number of times that the loop iterates.

Figure 4-3 Logic of the example `while` loop



The `while` Loop Is a Pretest Loop

The `while` loop is known as a *pretest* loop, which means it tests its expression before each iteration. Notice the variable declaration of `number` in Code Listing 4-3:

```
int number = 1;
```

The `number` variable is initialized with the value 1. If `number` had been initialized with a value that is greater than 5, as shown in the following program segment, the loop would never execute:

```
int number = 6;
while (number <= 5)
{
   System.out.println("Hello");
   number++;
}
```

An important characteristic of the `while` loop is that the loop will never iterate if the `boolean` expression is `false` to start with. If you want to be sure that a `while` loop executes the first time, you must initialize the relevant data in such a way that the `boolean` expression starts out as `true`.

Infinite Loops

In all but rare cases, loops must contain a way to terminate within themselves. This means that something inside the loop must eventually make the `boolean` expression `false`. The loop in Code Listing 4-3 stops when the expression `number <= 5` is `false`.

If a loop does not have a way of stopping, it is called an infinite loop. An *infinite loop* continues to repeat until the program is interrupted. Here is an example of an infinite loop:

```
int number = 1;
while (number <= 5)
{
   System.out.println("Hello");
}
```

This is an infinite loop because it does not contain a statement that changes the value of the number variable. Each time the boolean expression is tested, number will contain the value 1.

It's also possible to create an infinite loop by accidentally placing a semicolon after the first line of the while loop. Here is an example:

```
int number = 1;
while (number <= 5); // This semicolon is an ERROR!
{
   System.out.println("Hello");
   number++;
}
```

The semicolon at the end of the first line is assumed to be a null statement and disconnects the while statement from the block that comes after it. To the compiler, this loop looks like the following:

```
while (number <= 5);
```

This while loop will forever execute the null statement, which does nothing. The program will appear to have "gone into space" because there is nothing to display screen output or show activity.

Don't Forget the Braces with a Block of Statements

If you are using a block of statements, don't forget to enclose all of the statements in a set of braces. If the braces are accidentally left out, the while statement conditionally executes only the very next statement. For example, look at the following code:

```
int number = 1;
// This loop is missing its braces!
while (number <= 5)
   System.out.println("Hello");
   number++;
```

In this code the number++ statement is not in the body of the loop. Because the braces are missing, the while statement executes only the statement that immediately follows it. This loop will execute infinitely because there is no code in its body that changes the number variable.

Programming Style and the `while` Loop

It's possible to create loops that look like the following:

```
while (number <= 5) { System.out.println("Hello"); number++; }
```

Avoid this style of programming. The programming style you should use with the `while` loop is similar to that of the `if` statement as follows:

- If there is only one statement repeated by the loop, it should appear on the line after the `while` statement and be indented one additional level. The statement can optionally appear inside a set of braces.
- If the loop repeats a block, each line inside the braces should be indented.

This programming style should visually set the body of the loop apart from the surrounding code. In general, you'll find a similar style being used with the other types of loops presented in this chapter.

In the Spotlight:

Designing a Program with a `while` Loop

A project currently underway at Chemical Labs, Inc., requires that a substance be continually heated in a vat. A technician must check the substance's temperature every 15 minutes. If the substance's temperature does not exceed 102.5 degrees Celsius, then the technician does nothing. However, if the temperature is greater than 102.5 degrees Celsius, the technician must turn down the vat's thermostat, wait 5 minutes, and check the temperature again. The technician repeats these steps until the temperature does not exceed 102.5 degrees Celsius. The director of engineering has asked you to write a program that guides the technician through this process.

Here is the algorithm:

1. *Prompt the user to enter the substance's temperature.*
2. *Repeat the following steps as long as the temperature is greater than 102.5 degrees Celsius:*
   (a) Tell the technician to turn down the thermostat, wait 5 minutes, and check the temperature again.
   (b) Prompt the user to enter the substance's temperature.
3. *After the loop finishes, tell the technician that the temperature is acceptable and to check it again in 15 minutes.*

After reviewing this algorithm, you realize that Steps 2(a) and 2(b) should not be performed if the test condition (temperature is greater than 102.5) is false to begin with. The `while` loop will work well in this situation, because it will not execute even once if its condition is false. Code Listing 4-4 shows the program.

Code Listing 4-4 **(CheckTemperature.java)**

```
1  import java.util.Scanner;
2
```

```
3  /**
4     This program assists a technician in the process
5     of checking a substance's temperature.
6  */
7  public class CheckTemperature
8  {
9     public static void main(String[] args)
10    {
11       final double MAX_TEMP = 102.5;      // Maximum temperature
12       double temperature;                 // To hold the temperature
13
14       // Create a Scanner object for keyboard input.
15       Scanner keyboard = new Scanner(System.in);
16
17       // Get the current temperature.
18       System.out.print("Enter the substance's Celsius temperature: ");
19       temperature = keyboard.nextDouble();
20
21       // As long as necessary, instruct the technician
22       // to adjust the temperature.
23       while (temperature > MAX_TEMP)
24       {
25          System.out.println("The temperature is too high. Turn the");
26          System.out.println("thermostat down and wait 5 minutes.");
27          System.out.println("Then, take the Celsius temperature again");
28          System.out.print("and enter it here: ");
29          temperature = keyboard.nextDouble();
30       }
31
32       // Remind the technician to check the temperature
33       // again in 15 minutes.
34       System.out.println("The temperature is acceptable.");
35       System.out.println("Check it again in 15 minutes.");
36    }
37 }
```

Program Output with Example Input Shown in Bold

```
Enter the substance's Celsius temperature: 104.7 [Enter]
The temperature is too high. Turn the
thermostat down and wait 5 minutes.
Then, take the Celsius temperature again
and enter it here: 103.2 [Enter]
The temperature is too high. Turn the
thermostat down and wait 5 minutes.
Then, take the Celsius temperature again
and enter it here: 102.1 [Enter]
The temperature is acceptable.
Check it again in 15 minutes.
```

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

4.2 How many times will `"Hello World"` be printed in the following program segment?

```
int count = 10;
while (count < 1)
{
    System.out.println("Hello World");
    count++;
}
```

4.3 How many times will `"I love Java programming!"` be printed in the following program segment?

```
int count = 0;
while (count < 10)
   System.out.println("I love Java programming!");
```

4.3 Using the `while` Loop for Input Validation

CONCEPT: **The `while` loop can be used to create input routines that repeat until acceptable data is entered.**

Perhaps the most famous saying of the computer industry is "garbage in, garbage out." The integrity of a program's output is only as good as its input, so you should try to make sure garbage does not go into your programs. *Input validation* is the process of inspecting data given to a program by the user and determining whether it is valid. A good program should give clear instructions about the kind of input that is acceptable, and not assume the user has followed those instructions.

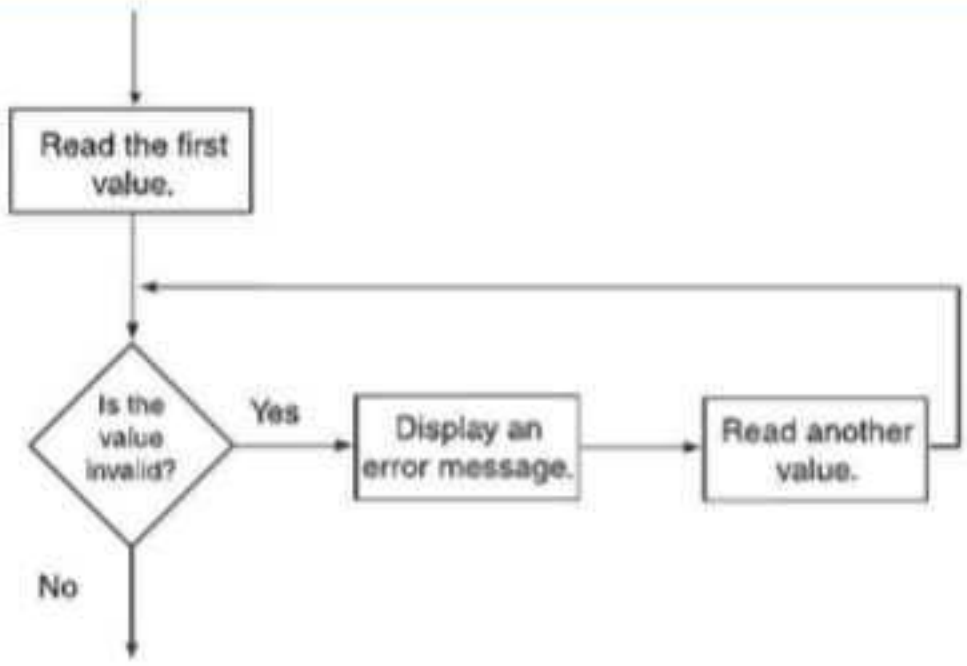
The `while` loop is especially useful for validating input. If an invalid value is entered, a loop can require that the user reenter it as many times as necessary. For example, the following loop asks for a number in the range of 1 through 100:

```
input = JOptionPane.showInputDialog("Enter a number " +
                    "in the range of 1 through 100.");
number = Integer.parseInt(input);
// Validate the input.
while (number < 1 || number > 100)
{
   input = JOptionPane.showInputDialog("Invalid input. " +
                       "Enter a number in the range of " +
                       "1 through 100.");
   number = Integer.parseInt(input);
}
```

This code first allows the user to enter a number. This takes place just before the loop. If the input is valid, the loop will not execute. If the input is invalid, however, the loop will display an error message and require the user to enter another number. The loop will continue to

execute until the user enters a valid number. The general logic of performing input validation is shown in Figure 4-4.

Figure 4-4 Input validation logic



The read operation that takes place just before the loop is called a *priming read*. It provides the first value for the loop to test. Subsequent values are obtained by the loop.

The program in Code Listing 4-5 calculates the number of soccer teams a youth league may create, based on a given number of players and a maximum number of players per team. The program uses `while` loops (in lines 28 through 36 and lines 44 through 49) to validate the user's input. Figure 4-5 shows an example of interaction with the program.

Code Listing 4-5 (SoccerTeams.java)

```
1  import javax.swing.JOptionPane;
2
3  /**
4     This program calculates the number of soccer teams
5     that a youth league may create from the number of
6     available players. Input validation is demonstrated
7     with while loops.
8  */
9
10 public class SoccerTeams
11 {
12    public static void main(String[] args)
13    {
14       final int MIN_PLAYERS = 9;     // Minimum players per team
15       final int MAX_PLAYERS = 15;    // Maximum players per team
16       int players;                   // Number of available players
17       int teamSize;                  // Number of players per team
18       int teams;                     // Number of teams
```

```
19        int leftOver;                  // Number of leftover players
20        String input;                  // To hold the user input
21
22        // Get the number of players per team.
23        input = JOptionPane.showInputDialog("Enter the number of " +
24                                            "players per team.");
25        teamSize = Integer.parseInt(input);
26
27        // Validate the number entered.
28        while (teamSize < MIN_PLAYERS || teamSize > MAX_PLAYERS)
29        {
30           input = JOptionPane.showInputDialog("The number must " +
31                                       "be at least " + MIN_PLAYERS +
32                                       " and no more than " +
33                                       MAX_PLAYERS + ".\n Enter " +
34                                       "the number of players.");
35           teamSize = Integer.parseInt(input);
36        }
37
38        // Get the number of available players.
39        input = JOptionPane.showInputDialog("Enter the available " +
40                                            "number of players.");
41        players = Integer.parseInt(input);
42
43        // Validate the number entered.
44        while (players < 0)
45        {
46           input = JOptionPane.showInputDialog("Enter 0 or " +
47                                               "greater.");
48           players = Integer.parseInt(input);
49        }
50
51        // Calculate the number of teams.
52        teams = players / teamSize;
53
54        // Calculate the number of leftover players.
55        leftOver = players % teamSize;
56
57        // Display the results.
58        JOptionPane.showMessageDialog(null, "There will be " +
59                                      teams + " teams with " +
60                                      leftOver +
61                                      " players left over.");
62        System.exit(0);
63     }
64  }
```

Figure 4-5 Interaction with the SoccerTeams program

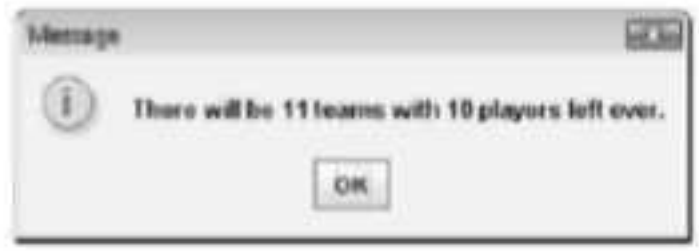
This input dialog box appears first. The user enters 4 (an invalid value) and clicks the OK button.

This input dialog box appears next. The user enters 12 and clicks the OK button.

This input dialog box appears next. The user enters –142 (an invalid value) and clicks the OK button.

This input dialog box appears next. The user enters 142 and clicks the OK button.

This message dialog box appears next.



Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

4.4 Write an input validation loop that asks the user to enter a number in the range of 10 through 24.

4.5 Write an input validation loop that asks the user to enter 'Y', 'y', 'N', or 'n'.

4.6 Write an input validation loop that asks the user to enter "Yes" or "No".

4.4 The do-while Loop

CONCEPT: The `do-while` loop is a posttest loop, which means its `boolean` expression is tested after each iteration.

The `do-while` loop looks something like an inverted `while` loop. Here is the `do-while` loop's format when the body of the loop contains only a single statement:

```
do
    Statement;
while (BooleanExpression);
```

Here is the format of the `do-while` loop when the body of the loop contains multiple statements:

```
do
{
    Statement;
    Statement;
    // Place as many statements here as necessary.
} while (BooleanExpression);
```

NOTE: The `do-while` loop must be terminated with a semicolon.

The `do-while` loop is a *posttest* loop. This means it does not test its `boolean` expression until it has completed an iteration. As a result, the `do-while` loop always performs at least one iteration, even if the `boolean` expression is `false` to begin with. This differs from the behavior of a `while` loop, which you will recall is a pretest loop. For example, in the following `while` loop the `println` statement will not execute at all:

```
int x = 1;
while (x < 0)
   System.out.println(x);
```

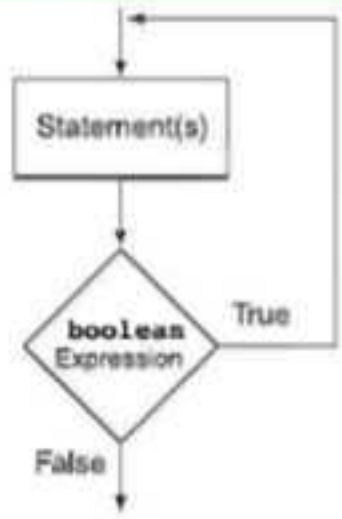
But the `println` statement in the following `do-while` loop will execute once because the `do-while` loop does not evaluate the expression `x < 0` until the end of the iteration:

```
int x = 1;
do
   System.out.println(x);
while (x < 0);
```

Figure 4-6 illustrates the logic of the `do-while` loop.

You should use the `do-while` loop when you want to make sure the loop executes at least once. For example, the program in Code Listing 4-6 averages a series of three test scores for a student. After the average is displayed, it asks the user whether he or she wants to average another set of test scores. The program repeats as long as the user enters `Y` for yes.

Figure 4-6 Logic of the do-while loop



Code Listing 4-6 (TestAverage1.java)

```
1  import java.util.Scanner; // Needed for the Scanner class
2
3  /**
4     This program demonstrates a user controlled loop.
5  */
6
7  public class TestAverage1
8  {
9     public static void main(String[] args)
10    {
11       int score1, score2, score3;      // Three test scores
12       double average;                  // Average test score
13       char repeat;                     // To hold 'y' or 'n'
14       String input;                    // To hold input
15
16       System.out.println("This program calculates the " +
17                          "average of three test scores.");
18
19       // Create a Scanner object for keyboard input.
20       Scanner keyboard = new Scanner(System.in);
21
22       // Get as many sets of test scores as the user wants.
23       do
24       {
25          // Get the first test score in this set.
26          System.out.print("Enter score #1: ");
27          score1 = keyboard.nextInt();
28
29          // Get the second test score in this set.
30          System.out.print("Enter score #2: ");
31          score2 = keyboard.nextInt();
32
```

```
33            // Get the third test score in this set.
34            System.out.print("Enter score #3: ");
35            score3 = keyboard.nextInt();
36
37            // Consume the remaining newline.
38            keyboard.nextLine();
39
40            // Calculate and print the average test score.
41            average = (score1 + score2 + score3) / 3.0;
42            System.out.println("The average is " + average);
43            System.out.println(); // Prints a blank line
44
45            // Does the user want to average another set?
46            System.out.println("Would you like to average " +
47                               "another set of test scores?");
48            System.out.print("Enter Y for yes or N for no: ");
49            input = keyboard.nextLine();    // Read a line.
50            repeat = input.charAt(0);       // Get the first char.
51
52         } while (repeat == 'Y' || repeat == 'y');
53      }
54   }
```

Program Output with Example Input Shown in Bold

```
This program calculates the average of three test scores.
Enter score #1: 89 [Enter]
Enter score #2: 90 [Enter]
Enter score #3: 97 [Enter]
The average is 92.0

Would you like to average another set of test scores?
Enter Y for yes or N for no: y [Enter]
Enter score #1: 78 [Enter]
Enter score #2: 65 [Enter]
Enter score #3: 88 [Enter]
The average is 77.0

Would you like to average another set of test scores?
Enter Y for yes or N for no: n [Enter]
```

When this program was written, the programmer had no way of knowing the number of times the loop would iterate. This is because the loop asks the user whether he or she wants to repeat the process. This type of loop is known as a *user controlled loop*, because it allows the user to decide the number of iterations.

4.5 The for Loop

CONCEPT: The for loop is ideal for performing a known number of iterations.

In general, there are two categories of loops: conditional loops and count-controlled loops. A *conditional loop* executes as long as a particular condition exists. For example, an input validation loop executes as long as the input value is invalid. When you write a conditional loop, you have no way of knowing the number of times it will iterate.

Sometimes you do know the exact number of iterations that a loop must perform. A loop that repeats a specific number of times is known as a *count-controlled loop*. For example, if a loop asks the user to enter the sales amounts for each month in the year, it will iterate 12 times. In essence, the loop counts to 12 and asks the user to enter a sales amount each time it makes a count.

A count-controlled loop must possess three elements:

1. It must initialize a control variable to a starting value.
2. It must test the control variable by comparing it to a maximum value. When the control variable reaches its maximum value, the loop terminates.
3. It must update the control variable during each iteration. This is usually done by incrementing the variable.

In Java, the for loop is ideal for writing count-controlled loops. It is specifically designed to initialize, test, and update a loop control variable. Here is the format of the for loop when used to repeat a single statement:

```
for (Initialization; Test; Update)
     Statement;
```

The format of the for loop when used to repeat a block is as follows:

```
for (Initialization; Test; Update)
{
   Statement;
   Statement;
   // Place as many statements here as necessary.
}
```

The first line of the for loop is known as the *loop header*. After the key word for, there are three expressions inside the parentheses, separated by semicolons. (Notice there is not a semicolon after the third expression.) The first expression is the *initialization expression*. It is normally used to initialize a control variable to its starting value. This is the first action performed by the loop, and it is done only once. The second expression is the *test expression*. This is a boolean expression that controls the execution of the loop. As long as this expression is true, the body of the for loop will repeat. The for loop is a pretest loop, so it evaluates the test expression before each iteration. The third expression is the *update expression*. It executes at the end of each iteration. Typically, this is a statement that increments the loop's control variable.

Here is an example of a simple `for` loop that prints "Hello" five times:

```
for (count = 1; count <= 5; count++)
   System.out.println("Hello");
```

In this loop, the initialization expression is `count = 1`, the test expression is `count <= 5`, and the update expression is `count++`. The body of the loop has one statement, which is the `println` statement. Figure 4-7 illustrates the sequence of events that takes place during the loop's execution. Notice that Steps 2 through 4 are repeated as long as the test expression is `true`.

Figure 4-7 Sequence of events in the `for` loop

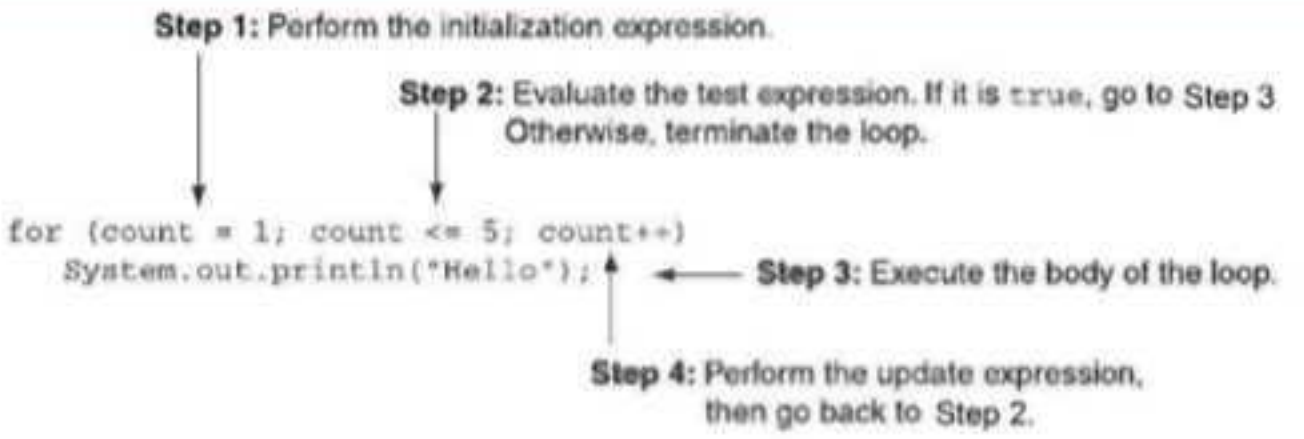
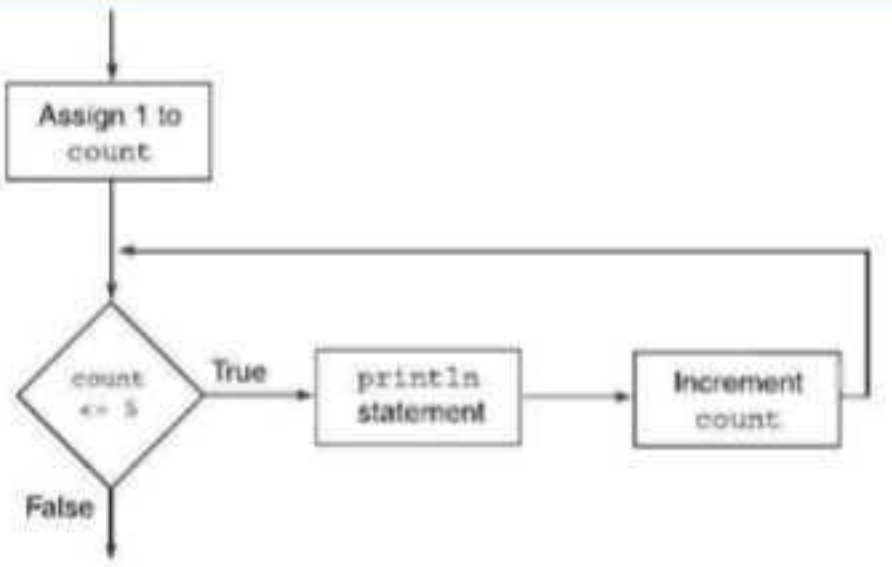


Figure 4-8 shows the loop's logic in the form of a flowchart.

Figure 4-8 Logic of the `for` loop



Notice how the control variable, `count`, is used to control the number of times that the loop iterates. During the execution of the loop, this variable takes on the values 1 through 5, and when the test expression `count <= 5` is `false`, the loop terminates. Because this variable keeps a count of the number of iterations, it is often called a *counter variable*.

Also notice that in this example the `count` variable is used only in the loop header, to control the number of loop iterations. It is not used for any other purpose. It is also possible to use the control variable within the body of the loop. For example, look at the following code:

```
for (number = 1; number <= 10; number++)
   System.out.print(number + " ");
```

The control variable in this loop is `number`. In addition to controlling the number of iterations, it is also used in the body of the loop. This loop will produce the following output:

```
1 2 3 4 5 6 7 8 9 10
```

As you can see, the loop displays the contents of the `number` variable during each iteration. The program in Code Listing 4-7 shows another example of a `for` loop that uses its control variable within the body of the loop. This program displays a table showing the numbers 1 through 10 and their squares.

Code Listing 4-7 (Squares.java)

```
1  /**
2     This program demonstrates the for loop.
3  */
4
5  public class Squares
6  {
7     public static void main(String[] args)
8     {
9        int number; // Loop control variable
10
11       System.out.println("Number Number Squared");
12       System.out.println("-----------------------");
13
14       for (number = 1; number <= 10; number++)
15       {
16          System.out.println(number + "\t\t" +
17                             number * number);
18       }
19    }
20 }
```

Program Output

```
Number      Number Squared
-------------------------------------
1                   1
2                   4
3                   9
4                   16
5                   25
6                   36
7                   49
8                   64
9                   81
10                  100
```

Figure 4-9 illustrates the sequence of events performed by this `for` loop.

Figure 4-9 Sequence of events with the `for` loop in Code Listing 4-7

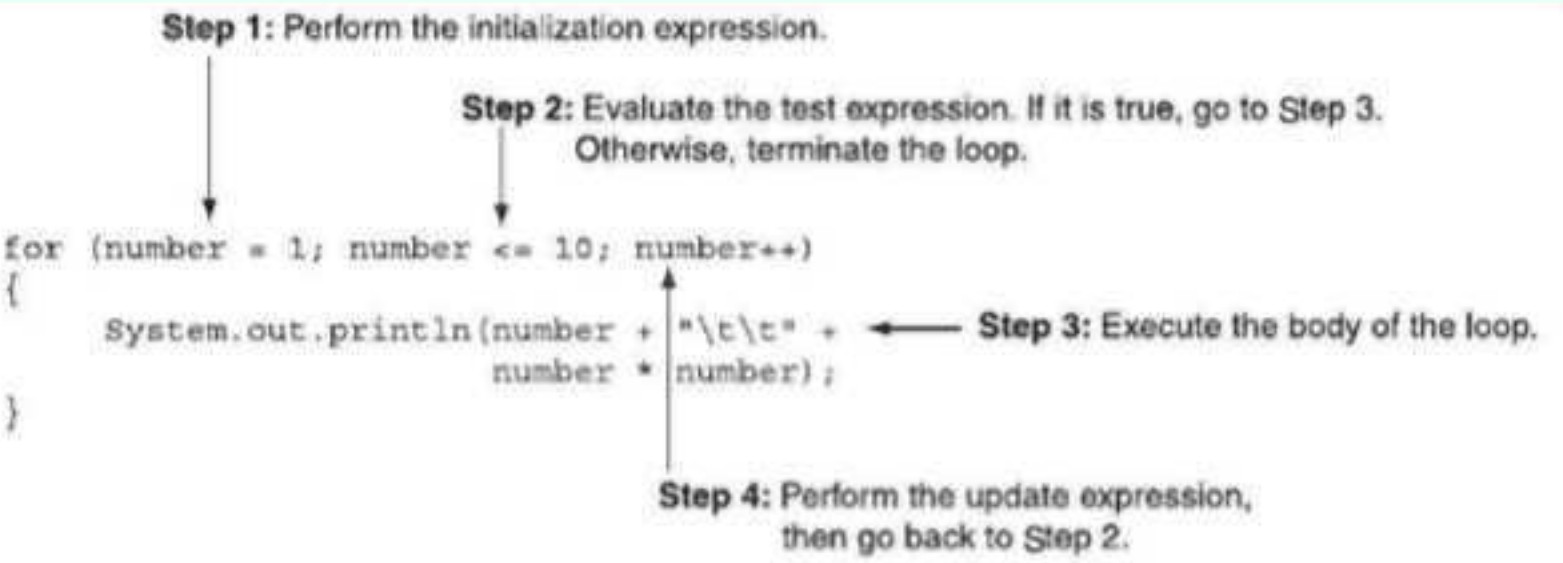
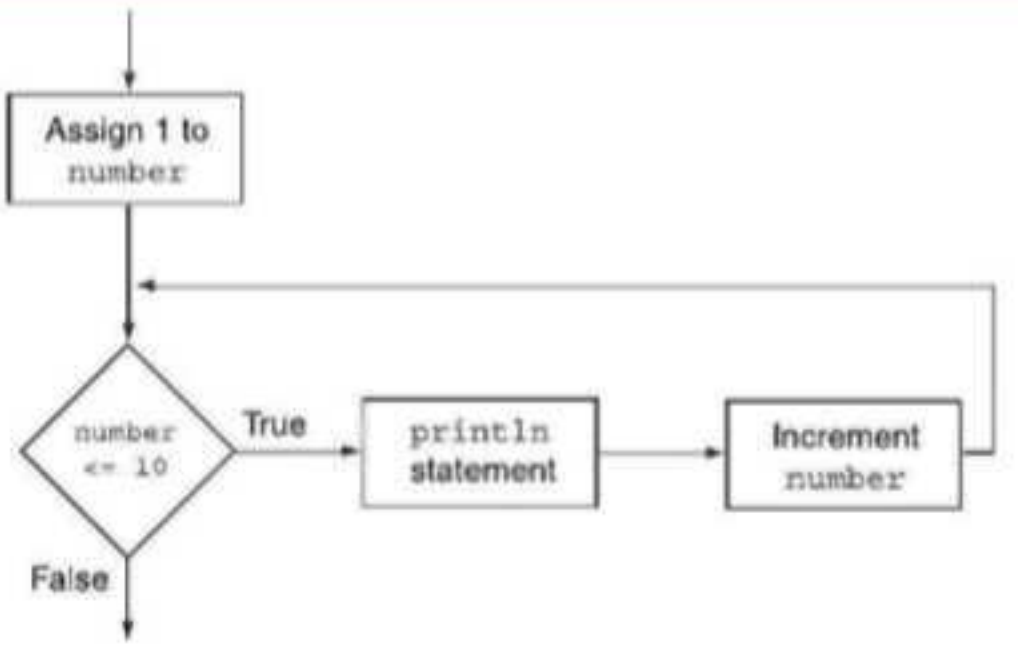


Figure 4-10 shows the logic of the loop.

Figure 4-10 Logic of the `for` loop in Code Listing 4-7



The `for` Loop Is a Pretest Loop

Because the `for` loop tests its `boolean` expression before it performs an iteration, it is a pretest loop. It is possible to write a `for` loop in such a way that it will never iterate. Here is an example:

```
for (count = 11; count <= 10; count++)
   System.out.println("Hello");
```

Because the variable `count` is initialized to a value that makes the `boolean` expression `false` from the beginning, this loop terminates as soon as it begins.

Avoid Modifying the Control Variable in the Body of the for Loop

Be careful not to place a statement that modifies the control variable in the body of the for loop. All modifications of the control variable should take place in the update expression, which is automatically executed at the end of each iteration. If a statement in the body of the loop also modifies the control variable, the loop probably will not terminate when you expect it to. The following loop, for example, increments x twice for each iteration:

```
for (x = 1; x <= 10; x++)
{
   System.out.println(x);
   x++; // Wrong!
}
```

Other Forms of the Update Expression

You are not limited to using increment statements in the update expression. Here is a loop that displays all the even numbers from 2 through 100 by adding 2 to its counter:

```
for (number = 2; number <= 100; number += 2)
   System.out.println(number);
```

And here is a loop that counts backward from 10 to 0:

```
for (number = 10; number >= 0; number--)
   System.out.println(number);
```

Declaring a Variable in the for Loop's Initialization Expression

Not only may the control variable be initialized in the initialization expression, but also it may be declared there. The following code shows an example. The following is a modified version of the loop in Code Listing 4-7:

```
for (int number = 1; number <= 10; number++)
{
   System.out.println(number + "\t\t" +
                      number * number);
}
```

In this loop, the number variable is both declared and initialized in the initialization expression. If the control variable is used only in the loop, it makes sense to declare it in the loop header. This makes the variable's purpose clearer.

When a variable is declared in the initialization expression of a for loop, the scope of the variable is limited to the loop. This means you cannot access the variable in statements outside the loop. For example, the following program segment will not compile because the last println statement cannot access the variable count:

```
for (int count = 1; count <= 10; count++)
   System.out.println(count);
System.out.println("count is now " + count); // ERROR!
```

Creating a User Controlled `for` Loop

Sometimes you want the user to determine the maximum value of the control variable in a `for` loop, and therefore determine the number of times the loop iterates. For example, look at the program in Code Listing 4-8. It is a modification of Code Listing 4-7. Instead of displaying the numbers 1 through 10 and their squares, this program allows the user to enter the maximum value to display.

Code Listing 4-8 (UserSquares.java)

```
1  import java.util.Scanner;      // Needed for the Scanner class
2
3  /**
4     This program demonstrates a user controlled for loop.
5  */
6
7  public class UserSquares
8  {
9     public static void main(String[] args)
10    {
11       int number;      // Loop control variable
12       int maxValue;    // Maximum value to display
13
14       System.out.println("I will display a table of " +
15                          "numbers and their squares.");
16
17       // Create a Scanner object for keyboard input.
18       Scanner keyboard = new Scanner(System.in);
19
20       // Get the maximum value to display.
21       System.out.print("How high should I go? ");
22       maxValue = keyboard.nextInt();
23
24       // Display the table.
25       System.out.println("Number Number Squared");
26       System.out.println("-----------------------");
27       for (number = 1; number <= maxValue; number++)
28       {
29          System.out.println(number + "\t\t" +
30                             number * number);
31       }
32    }
33 }
```

Program Output with Example Input Shown in Bold

```
I will display a table of numbers and their squares.
How high should I go? 7 [Enter]
Number      Number Squared
-------------------------
1                   1
2                   4
3                   9
4                   16
5                   25
6                   36
7                   49
```

In lines 21 and 22, which are before the loop, this program asks the user to enter the highest value to display. This value is stored in the `maxValue` variable as follows:

```
System.out.print("How high should I go? ");
maxValue = keyboard.nextInt();
```

In line 27, the for loop's test expression uses the value in the `maxValue` variable as the upper limit for the control variable as follows:

```
for (number = 1; number <= maxValue; number++)
```

In this loop, the `number` variable takes on the values 1 through `maxValue`, and then the loop terminates.

Using Multiple Statements in the Initialization and Update Expressions

It is possible to execute more than one statement in the initialization expression and the update expression. When using multiple statements in either of these expressions, simply separate the statements with commas. For example, look at the loop in the following code, which has two statements in the initialization expression:

```
int x, y;
for (x = 1, y = 1; x <= 5; x++)
{
   System.out.println(x + " plus " + y +
                      " equals " + (x + y));
}
```

This loop's initialization expression is as follows:

```
x = 1, y = 1
```

This initializes two variables, x and y. The output produced by this loop is as follows:

```
1 plus 1 equals 2
2 plus 1 equals 3
3 plus 1 equals 4
```

```
4 plus 1 equals 5
5 plus 1 equals 6
```

We can further modify the loop to execute two statements in the update expression. Here is an example:

```
int x, y;
for (x = 1, y = 1; x <= 5; x++, y++)
{
   System.out.println(x + " plus " + y +
                      " equals " + (x + y));
}
```

The loop's update expression is as follows:

```
x++, y++
```

This update expression increments both the x and y variables. The output produced by this loop is as follows:

```
1 plus 1 equals 2
2 plus 2 equals 4
3 plus 3 equals 6
4 plus 4 equals 8
5 plus 5 equals 10
```

Connecting multiple statements with commas works well in the initialization and update expressions, but don't try to connect multiple `boolean` expressions this way in the test expression. If you wish to combine multiple `boolean` expressions in the test expression, you must use the `&&` or `||` operators.

In the Spotlight:

Designing a Count-Controlled `for` Loop

Your friend Amanda just inherited a European sports car from her uncle. Amanda lives in the United States, and she is afraid she will get a speeding ticket because the car's speedometer indicates kilometers per hour (KPH). She has asked you to write a program that displays a table of speeds in kilometers per hour with their values converted to miles per hour (MPH). The formula for converting KPH to MPH is

$$MPH = KPH * 0.6214$$

In the formula, *MPH* is the speed in miles per hour and *KPH* is the speed in kilometers per hour.

The table that your program displays should show speeds from 60 kilometers per hour through 130 kilometers per hour, in increments of 10, along with their values converted to miles per hour. The table should look something like this:

KPH	MPH
60	37.3
70	43.5
80	49.7
etc.	
130	80.8

After thinking about this table of values, you decide that you will write a for loop that uses a counter variable to hold the KPH speeds. The counter's starting value will be 60, its ending value will be 130, and you will add 10 to the counter variable after each iteration. Inside the loop you will use the counter variable to calculate a speed in MPH. Code Listing 4-9 shows the code.

Code Listing 4-9 (SpeedConverter.java)

```
1  /**
2     This program displays a table of speeds in
3     kph converted to mph.
4  */
5
6  public class SpeedConverter
7  {
8     public static void main(String[] args)
9     {
10       // Constants
11       final int STARTING_KPH = 60;        // Starting speed
12       final int MAX_KPH = 130;            // Maximum speed
13       final int INCREMENT = 10;           // Speed increment
14
15       // Variables
16       int kph;        // To hold the speed in kph
17       double mph;     // To hold the speed in mph
18
19       // Display the table headings.
20       System.out.println("KPH\t\tMPH");
21       System.out.println("--------------------");
22
23       // Display the speeds.
24       for (kph = STARTING_KPH; kph <= MAX_KPH; kph += INCREMENT)
25       {
26           // Calculate the mph.
27           mph = kph * 0.6214;
28
29           // Display the speeds in kph and mph.
30           System.out.printf("%d\t\t%.1f\n", kph, mph);
31       }
32    }
33 }
```

Program Output

```
KPH      MPH
-----------------
60       37.3
70       43.5
80       49.7
90       55.9
100      62.1
110      68.4
120      74.6
130      80.8
```

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

4.7 Name the three expressions that appear inside the parentheses in the `for` loop's header.

4.8 You want to write a `for` loop that displays "I love to program" 50 times. Assume that you will use a control variable named `count`.
- a) What initialization expression will you use?
- b) What test expression will you use?
- c) What update expression will you use?
- d) Write the loop.

4.9 What will the following program segments display?

a)
```
for (int count = 0; count < 6; count++)
   System.out.println(count + count);
```

b)
```
for (int value = -5; value < 5; value++)
   System.out.println(value);
```

c)
```
int x;
for (x = 5; x <= 14; x += 3)
   System.out.println(x);
System.out.println(x);
```

4.10 Write a `for` loop that displays your name 10 times.

4.11 Write a `for` loop that displays all of the odd numbers, 1 through 49.

4.12 Write a `for` loop that displays every fifth number, zero through 100.

4.6 Running Totals and Sentinel Values

CONCEPT: **A running total is a sum of numbers that accumulates with each iteration of a loop. The variable used to keep the running total is called an accumulator. A sentinel is a value that signals when the end of a list of values has been reached.**

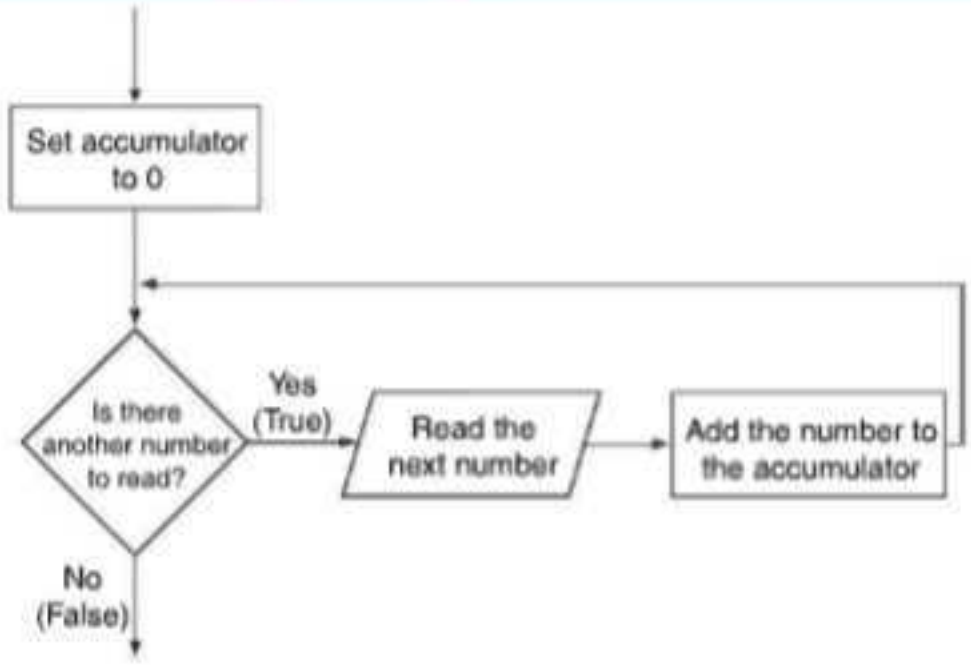
Many programming tasks require you to calculate the total of a series of numbers. For example, suppose you are writing a program that calculates a business's total sales for a week. The program would read the sales for each day as input and calculate the total of those numbers.

Programs that calculate the total of a series of numbers typically use two elements:

- A loop that reads each number in the series.
- A variable that accumulates the total of the numbers as they are read.

The variable used to accumulate the total of the numbers is called an *accumulator*. It is often said that the loop keeps a *running total* because it accumulates the total as it reads each number in the series. Figure 4-11 shows the general logic of a loop that calculates a running total.

Figure 4-11 Logic for calculating a running total



When the loop finishes, the accumulator will contain the total of the numbers that were read by the loop. Notice that the first step in the flowchart is to set the accumulator variable to 0. This is a critical step. Each time the loop reads a number, it adds it to the accumulator. If the accumulator starts with any value other than 0, it will not contain the correct total when the loop finishes.

Let's look at a program that calculates a running total. Code Listing 4-10 calculates a company's total sales over a period of time by taking daily sales figures as input and calculating a running total of them as they are gathered. Figure 4-12 shows an example of interaction with the program.

Code Listing 4-10 (TotalSales.java)

```
1  import java.text.DecimalFormat;
2  import javax.swing.JOptionPane;
3
4  /**
5     This program calculates a running total.
6  */
7
8  public class TotalSales
9  {
10    public static void main(String[] args)
11    {
12       int days;                  // The number of days
```

```
13        double sales;             // A day's sales figure
14        double totalSales;        // Accumulator
15        String input;             // To hold the user's input
16
17        // Create a DecimalFormat object to format output.
18        DecimalFormat dollar = new DecimalFormat("#,##0.00");
19
20        // Get the number of days.
21        input = JOptionPane.showInputDialog("For how many days " +
22                                            "do you have sales figures?");
23        days = Integer.parseInt(input);
24
25        // Set the accumulator to 0.
26        totalSales = 0.0;
27
28        // Get the sales figures and calculate a running total.
29        for (int count = 1; count <= days; count++)
30        {
31           input = JOptionPane.showInputDialog("Enter the sales " +
32                                               "for day " + count + ":");
33           sales = Double.parseDouble(input);
34           totalSales += sales; // Add sales to totalSales.
35        }
36
37        // Display the total sales.
38        JOptionPane.showMessageDialog(null, "The total sales are $" +
39                                      dollar.format(totalSales));
40
41        System.exit(0);
42     }
43  }
```

Figure 4-12 Interaction with the `TotalSales` program

Let's take a closer look at this program. In lines 20 through 23 the user is asked to enter the number of days for which he or she has sales figures. The number is read from an input dialog box and assigned to the `days` variable. Then, in line 26 the `totalSales` variable is assigned 0.0. In general programming terms, the `totalSales` variable is referred to as an accumulator. An *accumulator* is a variable that is initialized with a starting value, which is usually zero, and then accumulates a sum of numbers by having the numbers added to it. As you will see, it is critical that the accumulator is set to zero before values are added to it.

Next, the `for` loop in lines 29 through 35 executes. During each iteration of the loop, the user enters the amount of sales for a specific day, which are assigned to the `sales` variable. This is done in lines 31 through 33. Then, in line 34 the contents of `sales` is added to the existing value in the `totalSales` variable. (Note that line 34 does not assign `sales` to `totalSales`, but adds `sales` to `totalSales`. Put another way, this line increases `totalSales` by the amount in `sales`.)

Because `totalSales` was initially assigned 0.0, after the first iteration of the loop, `totalSales` will be set to the same value as `sales`. After each subsequent iteration, `totalSales` will be increased by the amount in `sales`. After the loop has finished, `totalSales` will contain the total of all the daily sales figures entered. Now it should be clear why we assigned 0.0 to `totalSales` before the loop executed. If `totalSales` started at any other value, the total would be incorrect.

Using a Sentinel Value

The program in Code Listing 4-10 requires the user to know in advance the number of days for which he or she has sales figures. Sometimes the user has a very long list of input values, and doesn't know the exact number of items. In other cases, the user might be entering values from several lists and it is impractical to require that every item in every list is counted.

A technique that can be used in these situations is to ask the user to enter a sentinel value at the end of the list. A *sentinel value* is a special value that cannot be mistaken as a member of the list, and signals that there are no more values to be entered. When the user enters the sentinel value, the loop terminates.

The program in Code Listing 4-11 shows an example. It calculates the total points earned by a soccer team over a series of games. It allows the user to enter the series of game points, and then -1 to signal the end of the list.

Code Listing 4-11 (SoccerPoints.java)

```
1  import java.util.Scanner;      // Needed for the Scanner class
2
3  /**
4     This program calculates the total number of points a
5     soccer team has earned over a series of games. The user
6     enters a series of point values, then -1 when finished.
7  */
8
9  public class SoccerPoints
```

```
10 {
11    public static void main(String[] args)
12    {
13       int points;                  // Game points
14       int totalPoints = 0;         // Accumulator initialized to 0
15
16       // Create a Scanner object for keyboard input.
17       Scanner keyboard = new Scanner(System.in);
18
19       // Display general instructions.
20       System.out.println("Enter the number of points your team");
21       System.out.println("has earned for each game this season.");
22       System.out.println("Enter -1 when finished.");
23       System.out.println();
24
25       // Get the first number of points.
26       System.out.print("Enter game points or -1 to end: ");
27       points = keyboard.nextInt();
28
29       // Accumulate the points until -1 is entered.
30       while (points != -1)
31       {
32          // Add points to totalPoints.
33          totalPoints += points;
34
35          // Get the next number of points.
36          System.out.print("Enter game points or -1 to end: ");
37          points = keyboard.nextInt();
38       }
39
40       // Display the total number of points.
41       System.out.println("The total points are " +
42                          totalPoints);
43    }
44 }
```

Program Output with Example Input Shown in Bold

```
Enter the number of points your team
has earned for each game this season.
Enter -1 when finished.

Enter game points or -1 to end: 7 [Enter]
Enter game points or -1 to end: 9 [Enter]
Enter game points or -1 to end: 4 [Enter]
Enter game points or -1 to end: 6 [Enter]
Enter game points or -1 to end: 8 [Enter]
Enter game points or -1 to end: -1 [Enter]
The total points are 34
```

The value -1 was chosen for the sentinel because it is not possible for a team to score negative points. Notice that this program performs a priming read to get the first value. This makes it possible for the loop to terminate immediately if the user enters -1 as the first value. Also note that the sentinel value is not included in the running total.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

4.13 Write a `for` loop that repeats seven times, asking the user to enter a number. The loop should also calculate the sum of the numbers entered.

4.14 In the following program segment, which variable is the loop control variable (also known as the counter variable) and which is the accumulator?

```
int a, x = 0, y = 0;
while (x < 10)
{
   a = x * 2;
   y += a;
   x++;
}
System.out.println("The sum is " + y);
```

4.15 Why should you be careful when choosing a sentinel value?

4.7 Nested Loops

CONCEPT: A loop that is inside another loop is called a nested loop.

Nested loops are necessary when a task performs a repetitive operation and that task itself must be repeated. A clock is a good example of something that works like a nested loop. The program in Code Listing 4-12 uses nested loops to simulate a clock.

Code Listing 4-12 (Clock.java)

```
1 /**
2    This program uses nested loops to simulate a clock.
3 */
4
5 public class Clock
6 {
7    public static void main(String[] args)
8    {
9       // Simulate the clock.
10      for (int hours = 1; hours <= 12; hours++)
11      {
12          for (int minutes = 0; minutes <= 59; minutes++)
13          {
14             for (int seconds = 0; seconds <= 59; seconds++)
```

```
15                {
16                   System.out.printf("%02d:%02d:%02d\n", hours, minutes, seconds);
17                }
18             }
19          }
20       }
21 }
```

Program Output

```
01:00:00
01:00:01
01:00:02
01:00:03

(The loop continues to count . . .)

12:59:57
12:59:58
12:59:59
```

The innermost loop (which begins at line 14) will iterate 60 times for each single iteration of the middle loop. The middle loop (which begins at line 12) will iterate 60 times for each single iteration of the outermost loop. When the outermost loop (which begins at line 10) has iterated 12 times, the middle loop will have iterated 720 times and the innermost loop will have iterated 43,200 times.

The simulated clock example brings up a few points about nested loops:

- An inner loop goes through all of its iterations for each iteration of an outer loop.
- Inner loops complete their iterations before outer loops do.
- To get the total number of iterations of a nested loop, multiply the number of iterations of all the loops.

The program in Code Listing 4-13 shows another example. It is a program that a teacher might use to get the average of each student's test scores. In line 22 the user enters the number of students, and in line 26 the user enters the number of test scores per student. The `for` loop that begins in line 29 iterates once for each student. The nested inner `for` loop, in lines 36 through 41, iterates once for each test score.

Code Listing 4-13 (TestAverage2.java)

```
1 import java.util.Scanner;
2
3 /**
4    This program demonstrates a nested loop.
5 */
6
7 public class TestAverage2
```

```
 8 {
 9    public static void main(String [] args)
10    {
11       int numStudents,     // Number of students
12           numTests,        // Number of tests per student
13           score,           // Test score
14           total;           // Accumulator for test scores
15       double average;      // Average test score
16
17       // Create a Scanner object for keyboard input.
18       Scanner keyboard = new Scanner(System.in);
19
20       // Get the number of students.
21       System.out.print("How many students do you have? ");
22       numStudents = keyboard.nextInt();
23
24       // Get the number of test scores per student.
25       System.out.print("How many test scores per student? ");
26       numTests = keyboard.nextInt();
27
28       // Process all the students.
29       for (int student = 1; student <= numStudents; student++)
30       {
31          total = 0; // Set the accumulator to zero.
32
33          // Get the test scores for a student.
34          System.out.println("Student number " + student);
35          System.out.println("---------------------");
36          for (int test = 1; test <= numTests; test++)
37          {
38             System.out.print("Enter score " + test + ": ");
39             score = keyboard.nextInt();
40             total += score; // Add score to total.
41          }
42
43          // Calculate and display the average.
44          average = total / numTests;
45          System.out.printf("The average for student %d is %.1f.\n\n",
46                            student, average);
47       }
48    }
49 }
```

Program Output with Example Input Shown in Bold

```
How many students do you have? 3 [Enter]
How many test scores per student? 3 [Enter]
```

```
Student number 1
-----------------------
Enter score 1: 100 [Enter]
Enter score 2: 95 [Enter]
Enter score 3: 90 [Enter]
The average for student number 1 is 95.0.

Student number 2
-----------------------
Enter score 1: 80 [Enter]
Enter score 2: 81 [Enter]
Enter score 3: 82 [Enter]
The average for student number 2 is 81.0.

Student number 3
-----------------------
Enter score 1: 75 [Enter]
Enter score 2: 85 [Enter]
Enter score 3: 80 [Enter]
The average for student number 3 is 80.0.
```

In the Spotlight:

Using Nested Loops to Print Patterns

One interesting way to learn about nested loops is to use them to display patterns on the screen. Let's look at a simple example. Suppose we want to print asterisks on the screen in the following rectangular pattern:

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

If you think of this pattern as having rows and columns, you can see that it has eight rows, and each row has six columns. The following code can be used to display one row of asterisks:

```
final int COLS = 6;
for (int col = 0; col < COLS; col++)
{
   System.out.print("*");
}
```

If we run this code in a program, it will produce the following output:

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

To complete the entire pattern, we need to execute this loop eight times. We can place the loop inside another loop that iterates eight times, as shown here:

```
1        final int COLS = 6;
2        final int ROWS = 8;
3        for (int row = 0; row < ROWS; row++)
4        {
5           for (int col = 0; col < COLS; col++)
6           {
7              System.out.print("*");
8           }
9           System.out.println();
10       }
```

The outer loop iterates eight times. Each time it iterates, the inner loop iterates six times. (Notice that in line 9, after each row has been printed, we call the `System.out.println()` method. We have to do that to advance the screen cursor to the next line at the end of each row. Without that statement, all the asterisks will be printed in one long row on the screen.)

We could easily write a program that prompts the user for the number of rows and columns, as shown in Code Listing 4-14.

Code Listing 4-14 **(RectangularPattern.java)**

```
1 import java.util.Scanner;
2
3 /**
4    This program displays a rectangular pattern
5    of asterisks.
6 */
7
8 public class RectangularPattern
9 {
10    public static void main(String[] args)
11    {
12       int rows, cols;
13
14       // Create a Scanner object for keyboard input.
15       Scanner keyboard = new Scanner(System.in);
16
17       // Get the number of rows and columns.
18       System.out.print("How many rows? ");
19       rows = keyboard.nextInt();
20       System.out.print("How many columns? ");
21       cols = keyboard.nextInt();
22
23       for (int r = 0; r < rows; r++)
24       {
25          for (int c = 0; c < cols; c++)
```

```
26          {
27             System.out.print("*");
28          }
29          System.out.println();
30       }
31    }
32 }
```

Program Output with Example Input Shown in Bold

```
How many rows? 5 [Enter]
How many columns? 10 [Enter]
**********
**********
**********
**********
**********
```

Let's look at another example. Suppose you want to print asterisks in a pattern that looks like the following triangle:

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

Once again, think of the pattern as being arranged in rows and columns. The pattern has a total of eight rows. In the first row, there is one column. In the second row, there are two columns. In the third row, there are three columns. This continues to the eighth row, which has eight columns. Code Listing 4-15 shows the program that produces this pattern.

Code Listing 4-15 (TrianglePattern.java)

```
1 import java.util.Scanner;
2
3 /**
4    This program displays a triangle pattern.
5 */
6
7 public class TrianglePattern
8 {
9    public static void main(String[] args)
10   {
11      final int BASE_SIZE = 8;
12
13      for (int r = 0; r < BASE_SIZE; r++)
```

```
14       {
15          for (int c = 0; c < (r + 1); c++)
16          {
17             System.out.print("*");
18          }
19          System.out.println();
20       }
21    }
22 }
```

Program Output

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

The outer loop (which begins in line 13) will iterate eight times. As the loop iterates, the variable `r` will be assigned the values 0 through 7.

For each iteration of the outer loop, the inner loop will iterate `r + 1` times. So,

- During the outer loop's first iteration, the variable `r` is assigned 0. The inner loop iterates one time, printing one asterisk.
- During the outer loop's second iteration, the variable `r` is assigned 1. The inner loop iterates two times, printing two asterisks.
- During the outer loop's third iteration, the variable `r` is assigned 2. The inner loop iterates three times, printing three asterisks.
- And so forth.

Let's look at another example. Suppose you want to display the following stair-step pattern:

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

The pattern has six rows. In general, we can describe each row as having some number of spaces followed by a # character. Here's a row-by-row description:

First row:	0 spaces followed by a # character.
Second row:	1 space followed by a # character.
Third row:	2 spaces followed by a # character.
Fourth row:	3 spaces followed by a # character.
Fifth row:	4 spaces followed by a # character.
Sixth row:	5 spaces followed by a # character.

To display this pattern, we can write code containing a pair of nested loops that work in the following manner:

- The outer loop will iterate six times. Each iteration will perform the following:
  - The inner loop will display the correct number of spaces, side by side.
  - Then, a # character will be displayed.

Code Listing 4-16 shows the Java code.

Code Listing 4-16 (StairStepPattern.java)

```
1 import java.util.Scanner;
2
3 /**
4    This program displays a stairstep pattern.
5 */
6
7 public class StairStepPattern
8 {
9    public static void main(String[] args)
10   {
11      final int NUM_STEPS = 6;
12
13      for (int r = 0; r < NUM_STEPS; r++)
14      {
15         for (int c = 0; c < r; c++)
16         {
17            System.out.print(" ");
18         }
19         System.out.println("#");
20      }
21   }
22 }
```

Program Output

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

The outer loop (which begins in line 13) will iterate six times. As the loop iterates, the variable `r` will be assigned the values 0 through 5.

For each iteration of the outer loop, the inner loop will iterate `r` times. So,

- During the outer loop's first iteration, the variable `r` is assigned 0. The inner loop will not execute at this time.

- During the outer loop's second iteration, the variable `r` is assigned 1. The inner loop iterates one time, printing one space.
- During the outer loop's third iteration, the variable `r` is assigned 2. The inner loop iterates two times, printing two spaces.
- And so forth.

4.8 The `break` and `continue` Statements (Optional)

CONCEPT: **The `break` statement causes a loop to terminate early. The `continue` statement causes a loop to stop its current iteration and begin the next one.**

The `break` statement, which was used with the `switch` statement in Chapter 3, can also be placed inside a loop. When it is encountered, the loop stops and the program jumps to the statement immediately following the loop. Although it is perfectly acceptable to use the `break` statement in a `switch` statement, it is considered taboo to use it in a loop. This is because it bypasses the normal condition that is required to terminate the loop, and it makes code difficult to understand and debug. For this reason, you should avoid using the `break` statement in a loop when possible.

The `continue` statement causes the current iteration of a loop to end immediately. When `continue` is encountered, all the statements in the body of the loop that appear after it are ignored, and the loop prepares for the next iteration. In a `while` loop, this means the program jumps to the `boolean` expression at the top of the loop. As usual, if the expression is still `true`, the next iteration begins. In a `do-while` loop, the program jumps to the `boolean` expression at the bottom of the loop, which determines whether the next iteration will begin. In a `for` loop, `continue` causes the update expression to be executed, and then the test expression is evaluated.

The `continue` statement should also be avoided. Like the `break` statement, it bypasses the loop's logic and makes the code difficult to understand and debug.

4.9 Deciding Which Loop to Use

CONCEPT: **Although most repetitive algorithms can be written with any of the three types of loops, each works best in different situations.**

Each of Java's three loops is ideal to use in different situations. The following is a short summary of when each loop should be used:

- **The `while` loop.** The `while` loop is a pretest loop. It is ideal in situations where you do not want the loop to iterate if the condition is `false` from the beginning. It is also ideal if you want to use a sentinel value to terminate the loop.
- **The `do-while` loop.** The `do-while` loop is a posttest loop. It is ideal in situations where you always want the loop to iterate at least once.
- **The `for` loop.** The `for` loop is a pretest loop that has built-in expressions for initializing, testing, and updating. These expressions make it very convenient to use a loop control variable as a counter. The `for` loop is ideal in situations where the exact number of iterations is known.

4.10 Introduction to File Input and Output

CONCEPT: **The Java API provides several classes that you can use for writing data to a file and reading data from a file. To write data to a file, you can use the `PrintWriter` class and, optionally, the `FileWriter` class. To read data from a file, you can use the `Scanner` class and the `File` class.**

The programs you have written so far require you to reenter data each time the program runs. This is because the data stored in variables and objects in RAM disappears once the program stops running. To retain data between the times it runs, a program must have a way of saving the data.

Data may be saved in a *file*, which is usually stored on a computer's disk. Once the data is saved in a file, it will remain there after the program stops running. The data can then be retrieved and used at a later time. In general, there are three steps that are taken when a file is used by a program:

1. The file must be *opened*. When the file is opened, a connection is created between the file and the program.
2. Data is then written to the file or read from the file.
3. When the program is finished using the file, the file must be *closed*.

In this section we will discuss how to write Java programs that write data to files and read data from files. The terms *input file* and *output file* are commonly used. An *input file* is a file that a program reads data from. It is called an input file because the data stored in it serves as input to the program. An *output file* is a file that a program writes data to. It is called an output file because the program stores output in the file.

In general, there are two types of files: text and binary. A *text file* contains data that has been encoded as text, using a scheme such as Unicode. Even if the file contains numbers, those numbers are stored in the file as a series of characters. As a result, the file may be opened and viewed in a text editor such as Notepad. A *binary file* contains data that has not been converted to text. As a consequence, you cannot view the contents of a binary file with a text editor. In this chapter, we will discuss how to work with text files. Binary files are discussed in Chapter 11.

The Java API provides a number of classes that you will use to work with files. To use these classes, you will place the following `import` statement near the top of your program:

```
import java.io.*;
```

Using the `PrintWriter` Class to Write Data to a File

To write data to a file you will create an instance of the `PrintWriter` class. The `PrintWriter` class allows you to open a file for writing. It also allows you to write data to the file using the same `print` and `println` methods that you have been using to display data on the screen. You pass the name of the file that you wish to open, as a string, to the `PrintWriter` class's constructor. For example, the following statement creates a `PrintWriter` object and passes the file name *StudentData.txt* to the constructor.

```
PrintWriter outputFile = new PrintWriter("StudentData.txt");
```

This statement will create an empty file named *StudentData.txt* and establish a connection between it and the `PrintWriter` object that is referenced by `outputFile`. The file will be created in the current directory or folder.

You may also pass a reference to a `String` object as an argument to the `PrintWriter` constructor. For example, in the following code the user specifies the name of the file.

```
String filename;
filename = JOptionPane.showInputDialog("Enter the filename.");
PrintWriter outputFile = new PrintWriter(filename);
```

WARNING! If the file that you are opening with the `PrintWriter` object already exists, it will be erased and an empty file by the same name will be created.

Once you have created an instance of the `PrintWriter` class and opened a file, you can write data to the file using the `print` and `println` methods. You already know how to use `print` and `println` with `System.out` to display data on the screen. They are used the same way with a `PrintWriter` object to write data to a file. For example, assuming that `outputFile` references a `PrintWriter` object, the following statement writes the string `"Jim"` to the file:

```
outputFile.println("Jim");
```

When the program is finished writing data to the file, it must close the file. To close the file use the `PrintWriter` class's `close` method. Here is an example of the method's use:

```
outputFile.close();
```

Your application should always close files when finished with them. This is because the system creates one or more buffers when a file is opened. A *buffer* is a small "holding section" of memory. When a program writes data to a file, that data is first written to the buffer. When the buffer is filled, all the information stored there is written to the file. This technique increases the system's performance because writing data to memory is faster than writing it to a disk. The `close` method writes any unsaved data remaining in the file buffer.

Once a file is closed, the connection between it and the `PrintWriter` object is removed. In order to perform further operations on the file, it must be opened again.

More about the `PrintWriter` Class's `println` Method

The `PrintWriter` class's `println` method writes a line of data to a file. For example, assume an application creates a file and writes three students' first names and their test scores to the file with the following code:

```
PrintWriter outputFile = new PrintWriter("StudentData.txt");
outputFile.println("Jim");
outputFile.println(95);
outputFile.println("Karen");
outputFile.println(98);
outputFile.println("Bob");
outputFile.println(82);
outputFile.close();
```

The `println` method writes data to the file and then writes a newline character immediately after the data. You can visualize the data written to the file in the following manner:

`Jim`*<newline>*`95`*<newline>*`Karen`*<newline>*`98`*<newline>*`Bob`*<newline>*`82`*<newline>*

The newline characters are represented here as *<newline>*. You do not actually see the newline characters, but when the file is opened in a text editor such as Notepad, its contents will appear as shown in Figure 4-13. As you can see from the figure, each newline character causes the data that follows it to be displayed on a new line.

Figure 4-13 File contents displayed in Notepad

In addition to separating the contents of a file into lines, the newline character also serves as a delimiter. A *delimiter* is an item that separates other items. When you write data to a file using the `println` method, newline characters will separate the individual items of data. Later you will see that the individual items of data in a file must be separated in order for them to be read from the file.

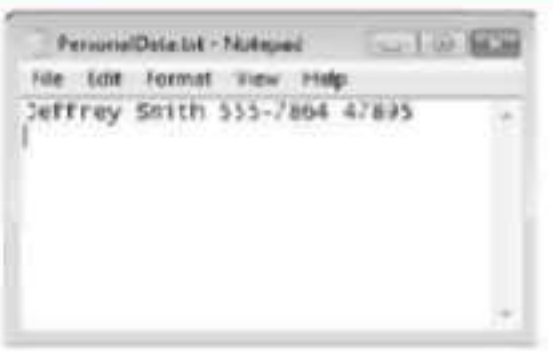
The `PrintWriter` Class's `print` Method

The `print` method is used to write an item of data to a file without writing the newline character. For example, look at the following code:

```
String name = "Jeffrey Smith";
String phone = "555-7864";
int idNumber = 47895;
PrintWriter outputFile = new PrintWriter("PersonalData.txt");
outputFile.print(name + " ");
outputFile.print(phone + " ");
outputFile.println(idNumber);
outputFile.close();
```

This code uses the `print` method to write the contents of the `name` object to the file, followed by a space (`" "`). Then it uses the `print` method to write the contents of the `phone` object to the file, followed by a space. Then it uses the `println` method to write the contents of the `idNumber` variable, followed by a newline character. Figure 4-14 shows the contents of the file displayed in Notepad.

Figure 4-14 Contents of file displayed in Notepad



Adding a `throws` Clause to the Method Header

When an unexpected event occurs in a Java program, it is said that the program throws an exception. For now, you can think of an *exception* as a signal indicating that the program cannot continue until the unexpected event has been dealt with. For example, suppose you create a `PrintWriter` object and pass the name of a file to its constructor. The `PrintWriter` object attempts to create the file, but unexpectedly, the disk is full and the file cannot be created. Obviously the program cannot continue until this situation has been dealt with, so an exception is thrown, which causes the program to suspend normal execution.

When an exception is thrown, the method that is executing must either deal with the exception or throw it again. If the `main` method throws an exception, the program halts and an error message is displayed. Because `PrintWriter` objects are capable of throwing exceptions, we must either write code that deals with the possible exceptions, or allow our methods to rethrow the exceptions when they occur. In Chapter 12 you will learn all about exceptions and how to respond to them, but for now, we will simply allow our methods to rethrow any exceptions that might occur.

To allow a method to rethrow an exception that has not been dealt with, you simply write a `throws` clause in the method header. The `throws` clause must indicate the type of exception that might be thrown. The following is an example:

```
public static void main(String[] args) throws IOException
```

This header indicates that the `main` method is capable of throwing an exception of the `IOException` type. This is the type of exception that `PrintWriter` objects are capable of throwing. So, any method that uses `PrintWriter` objects and does not respond to their exceptions must have this `throws` clause listed in its header.

In addition, any method that calls a method that uses a `PrintWriter` object should have a `throws IOException` clause in its header. For example, suppose the `main` method does not perform any file operations, but calls a method named `buildFile` that opens a file and writes data to it. Both the `buildFile` and `main` methods should have a `throws IOException` clause in their headers. Otherwise a compiler error will occur.

An Example Program

Let's look at an example program that writes data to a file. The program in Code Listing 4-17 writes the names of your friends to a file.

Code Listing 4-17 (`FileWriteDemo.java`)

```
1  import java.util.Scanner;       // Needed for Scanner class
2  import java.io.*;               // Needed for File I/O classes
3
4  /**
5    This program writes data to a file.
6  */
7
8  public class FileWriteDemo
9  {
```

```
10    public static void main(String[] args) throws IOException
11    {
12       String filename;       // File name
13       String friendName;     // Friend's name
14       int numFriends;        // Number of friends
15
16       // Create a Scanner object for keyboard input.
17       Scanner keyboard = new Scanner(System.in);
18
19       // Get the number of friends.
20       System.out.print("How many friends do you have? ");
21       numFriends = keyboard.nextInt();
22
23       // Consume the remaining newline character.
24       keyboard.nextLine();
25
26       // Get the filename.
27       System.out.print("Enter the filename: ");
28       filename = keyboard.nextLine();
29
30       // Open the file.
31       PrintWriter outputFile = new PrintWriter(filename);
32
33       // Get data and write it to the file.
34       for (int i = 1; i <= numFriends; i++)
35       {
36          // Get the name of a friend.
37          System.out.print("Enter the name of friend " +
38                           "number " + i + ": ");
39          friendName = keyboard.nextLine();
40
41          // Write the name to the file.
42          outputFile.println(friendName);
43       }
44
45       // Close the file.
46       outputFile.close();
47       System.out.println("Data written to the file.");
48    }
49 }
```

Program Output with Example Input Shown in Bold

```
How many friends do you have? 5 [Enter]
Enter the filename: MyFriends.txt [Enter]
Enter the name of friend number 1: Joe [Enter]
Enter the name of friend number 2: Rose [Enter]
Enter the name of friend number 3: Greg [Enter]
```

```
Enter the name of friend number 4: Kirk [Enter]
Enter the name of friend number 5: Renee [Enter]
Data written to the file.
```

The `import` statement in line 2 is necessary because this program uses the `PrintWriter` class. In addition, the `main` method header, in line 10, has a `throws IOException` clause because objects of the `PrintWriter` class can potentially throw an `IOException`.

This program asks the user to enter the number of friends he or she has (in lines 20 and 21), then a name for the file that will be created (in lines 27 and 28). The `filename` variable references the name of the file, and is used in the following statement, in line 31:

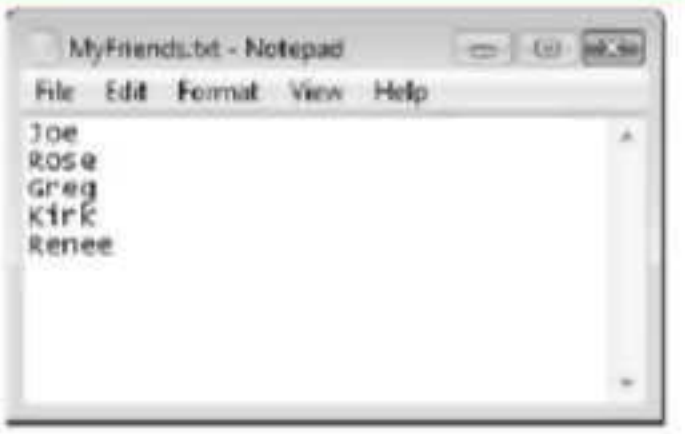
```
PrintWriter outputFile = new PrintWriter(filename);
```

This statement opens the file and creates a `PrintWriter` object that can be used to write data to the file. The `for` loop in lines 34 through 43 performs an iteration for each friend that the user has, each time asking for the name of a friend. The user's input is referenced by the `friendName` variable. Once the name is entered, it is written to the file with the following statement, which appears in line 42:

```
outputFile.println(friendName);
```

After the loop finishes, the file is closed in line 46. After the program is executed with the input shown in the example run, the file *MyFriends.txt* will be created. If we open the file in Notepad, we will see its contents as shown in Figure 4-15.

Figure 4-15 Contents of the file displayed in Notepad



Review

Before moving on, let's review the basic steps necessary when writing a program that writes data to a file:

1. You need the `import java.io.*;` statement in the top section of your program.
2. Because we have not yet learned how to respond to exceptions, any method that uses a `PrintWriter` object must have a `throws IOException` clause in its header.
3. You create a `PrintWriter` object and pass the name of the file as a string to the constructor.

4. You use the `PrintWriter` class's `print` and `println` methods to write data to the file.
5. When finished writing to the file, you use the `PrintWriter` class's `close` method to close the file.

Appending Data to a File

When you pass the name of a file to the `PrintWriter` constructor, and the file already exists, it will be erased and a new empty file with the same name will be created. Sometimes, however, you want to preserve an existing file and append new data to its current contents. Appending to a file means writing new data to the end of the data that already exists in the file.

To append data to an existing file, you first create an instance of the `FileWriter` class. You pass two arguments to the `FileWriter` constructor: a string containing the name of the file, and the `boolean` value `true`. Here is an example:

```
FileWriter fwriter = new FileWriter("MyFriends.txt", true);
```

This statement creates a `FileWriter` object and opens the file *MyFriends.txt* for writing. Any data written to the file will be appended to the file's existing contents. (If the file does not exist, it will be created.)

You still need to create a `PrintWriter` object so you can use the `print` and `println` methods to write data to the file. When you create the `PrintWriter` object, you pass a reference to the `FileWriter` object as an argument to the `PrintWriter` constructor. For example, look at the following code:

```
FileWriter fwriter = new FileWriter("MyFriends.txt", true);
PrintWriter outputFile = new PrintWriter(fwriter);
```

This creates a `PrintWriter` object that can be used to write data to the file *MyFriends.txt*. Any data that is written to the file will be appended to the file's existing contents. For example, assume the file *MyFriends.txt* exists and contains the following data:

```
Joe
Rose
Greg
Kirk
Renee
```

The following code opens the file and appends additional data to its existing contents:

```
FileWriter fwriter = new FileWriter("MyFriends.txt", true);
PrintWriter outputFile = new PrintWriter(fwriter);
outputFile.println("Bill");
outputFile.println("Steven");
outputFile.println("Sharon");
outputFile.close();
```

After this code executes, the *MyFriends.txt* file will contain the following data:

```
Joe
Rose
Greg
Kirk
Renee
Bill
Steven
Sharon
```

NOTE: The `FileWriter` class also throws an `IOException` if the file cannot be opened for any reason.

Specifying the File Location

When you open a file you may specify its path along with its filename. On a Windows computer, paths contain backslash characters. Remember that when a single backslash character appears in a string literal, it marks the beginning of an escape sequence such as `"\n"`. Two backslash characters in a string literal represent a single backslash. So, when you provide a path in a string literal, and the path contains backslash characters, you must use two backslash characters in the place of each single backslash character.

For example, the path `"E:\\Names.txt"` specifies that *Names.txt* is in the root folder of drive E:, and the path `"C:\\MyData\\Data.txt"` specifies that *Data.txt* is in the *\MyData* folder on drive C:. In the following statement, the file *Pricelist.txt* is created in the root folder of drive A:.

```
PrintWriter outputFile = new PrintWriter("A:\\PriceList.txt");
```

You only need to use double backslashes if the file's path is in a string literal. If your program asks the user to enter a path into a `String` object, which is then passed to the `PrintWriter` or `FileWriter` constructor, the user does not have to enter double backslashes.

TIP: Java allows you to substitute forward slashes for backslashes in a Windows path. For example, the path `"C:\\MyData\\Data.txt"` could be written as `"C:/MyData/Data.txt"`. This eliminates the need to use double backslashes.

On a UNIX or Linux computer, you can provide a path without any modifications. Here is an example:

```
PrintWriter outputFile = new PrintWriter("/home/rharrison/names.txt");
```

Reading Data from a File

In Chapter 2 you learned how to use the `Scanner` class to read input from the keyboard. To read keyboard input, recall that we create a `Scanner` object, passing `System.in` to the `Scanner` class constructor. Here is an example:

```
Scanner keyboard = new Scanner(System.in);
```

Recall that the `System.in` object represents the keyboard. Passing `System.in` as an argument to the `Scanner` constructor specifies that the keyboard is the `Scanner` object's source of input.

You can also use the `Scanner` class to read input from a file. Instead of passing `System.in` to the `Scanner` class constructor, you pass a reference to a `File` object. Here is an example:

```
File myFile = new File("Customers.txt");
Scanner inputFile = new Scanner(myFile);
```

The first statement creates an instance of the `File` class. The `File` class is in the Java API, and is used to represent a file. Notice that we have passed the string `"Customers.txt"` to the constructor. This creates a `File` object that represents the file *Customers.txt*.

In the second statement we pass a reference to this `File` object as an argument to the `Scanner` class constructor. This creates a `Scanner` object that uses the file *Customers.txt* as its source of input. You can then use the same `Scanner` class methods that you learned about in Chapter 2 to read items from the file. (See Table 2-17 for a list of commonly used methods.)

When you are finished reading from the file, you use the `Scanner` class's `close` method to close the file. For example, assuming the variable `inputFile` references a `Scanner` object, the following statement closes the file that is the object's source of input:

```
inputFile.close();
```

Reading Lines from a File with the `nextLine` Method

The `Scanner` class's `nextLine` method reads a line of input, and returns the line as a `String`. The program in Code Listing 4-18 demonstrates how the `nextLine` method can be used to read a line from a file. This program asks the user to enter a filename. It then displays the first line in the file on the screen.

Code Listing 4-18 (ReadFirstLine.java)

```
1  import java.util.Scanner;  // Needed for Scanner class
2  import java.io.*;          // Needed for File and IOException
3
4  /**
5     This program reads the first line from a file.
6  */
7
8  public class ReadFirstLine
9  {
10    public static void main(String[] args) throws IOException
11    {
12       // Create a Scanner object for keyboard input.
13       Scanner keyboard = new Scanner(System.in);
14
```

```
15        // Get the file name.
16        System.out.print("Enter the name of a file: ");
17        String filename = keyboard.nextLine();
18
19        // Open the file.
20        File file = new File(filename);
21        Scanner inputFile = new Scanner(file);
22
23        // Read the first line from the file.
24        String line = inputFile.nextLine();
25
26        // Display the line.
27        System.out.println("The first line in the file is:");
28        System.out.println(line);
29
30        // Close the file.
31        inputFile.close();
32     }
33  }
```

Program Output with Example Input Shown in Bold

```
Enter the name of a file: MyFriends.txt [Enter]
The first line in the file is:
Joe
```

This program gets the name of a file from the user in line 17. A `File` object is created in line 20 to represent the file, and a `Scanner` object is created in line 21 to read data from the file. Line 24 reads a line from the file. After this statement executes, the `line` variable references a `String` object holding the line that was read from the file. The line is displayed on the screen in line 28, and the file is closed in line 31.

It's worth pointing out that this program creates two separate `Scanner` objects. The `Scanner` object that is created in line 13 reads data from the keyboard, and the `Scanner` object that is created in line 21 reads data from a file.

When a file is opened for reading, a special value known as a *read position* is internally maintained for that file. A file's read position marks the location of the next item that will be read from the file. When a file is opened, its read position is set to the first item in the file. When the item is read, the read position is advanced to the next item in the file. As subsequent items are read, the internal read position advances through the file. For example, consider the file *Quotation.txt*, shown in Figure 4-16. As you can see from the figure, the file has three lines.

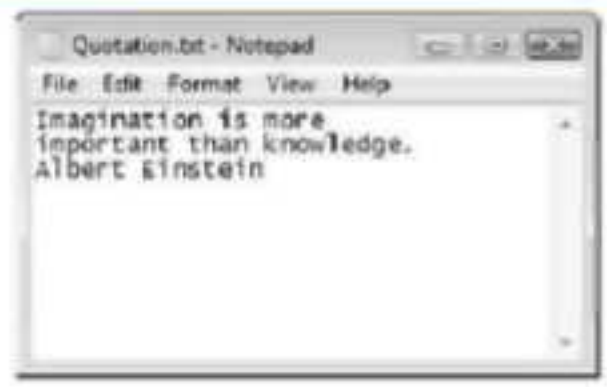
You can visualize that the data is stored in the file in the following manner:

```
Imagination is more<newline>important than knowledge.<newline>
Albert Einstein<newline>
```

Suppose a program opens the file with the following code:

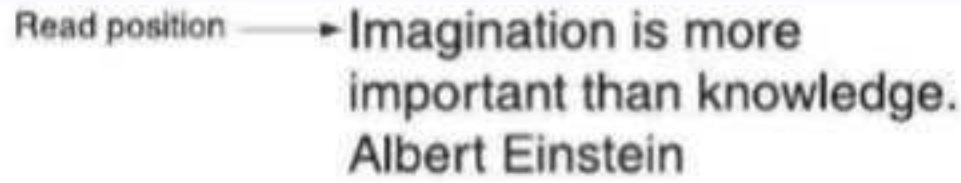
```
File file = new File("Quotation.txt");
Scanner inputFile = new Scanner(file);
```

Figure 4-16 File with three lines



When this code opens the file, its read position is at the beginning of the first line, as illustrated in Figure 4-17.

Figure 4-17 Initial read position

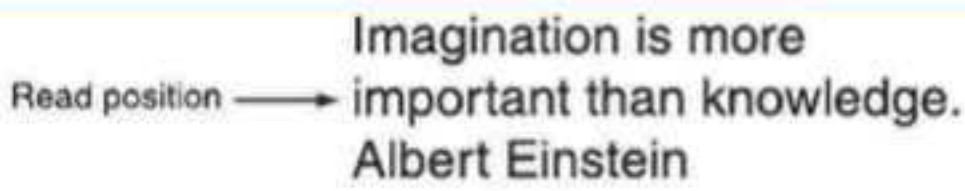


Now, suppose the program uses the following statement to read a line from the file:

```
String str = inputFile.nextLine();
```

This statement will read a line from the file, beginning at the current read position. After the statement executes, the object referenced by `str` will contain the string "Imagination is more". The file's read position will be advanced to the next line, as illustrated in Figure 4-18.

Figure 4-18 Read position after first line is read



If the `nextLine` method is called again, the second line will be read from the file and the file's read position will be advanced to the third line. After all the lines have been read, the read position will be at the end of the file.

NOTE: The string that is returned from the `nextLine` method will not contain the newline character.

Adding a `throws` Clause to the Method Header

When you pass a `File` object reference to the `Scanner` class constructor, the constructor will throw an exception of the `IOException` type if the specified file is not found. So, you will need to write a throws `IOException` clause in the header of any method that passes a `File` object reference to the `Scanner` class constructor.

Detecting the End of a File

Quite often a program must read the contents of a file without knowing the number of items that are stored in the file. For example, the *MyFriends.txt* file that was created by the program in Code Listing 4-17 can have any number of names stored in it. This is because the program asks the user for the number of friends that he or she has. If the user enters *5* for the number of friends, the program creates a file with five names in it. If the user enters 100, the program creates a file with 100 names in it.

The `Scanner` class has a method named `hasNext` that can be used to determine whether the file has more data that can be read. You call the `hasNext` method before you call any other methods to read from the file. If there is more data that can be read from the file, the `hasNext` method returns `true`. If the end of the file has been reached and there is no more data to read, the `hasNext` method returns `false`.

Code Listing 4-19 shows an example. The program reads the file containing the names of your friends, which was created by the program in Code Listing 4-17.

Code Listing 4-19 (`FileReadDemo.java`)

```
1  import java.util.Scanner;  // Needed for the Scanner class
2  import java.io.*;          // Needed for the File and IOException
3
4  /**
5     This program reads data from a file.
6  */
7
8  public class FileReadDemo
9  {
10    public static void main(String[] args) throws IOException
11    {
12       // Create a Scanner object for keyboard input.
13       Scanner keyboard = new Scanner(System.in);
14
15       // Get the filename.
16       System.out.print("Enter the filename: ");
17       String filename = keyboard.nextLine();
```

```
18
19        // Open the file.
20        File file = new File(filename);
21        Scanner inputFile = new Scanner(file);
22
23        // Read lines from the file until no more are left.
24        while (inputFile.hasNext())
25        {
26           // Read the next name.
27           String friendName = inputFile.nextLine();
28
29           // Display the last name read.
30           System.out.println(friendName);
31        }
32
33        // Close the file.
34        inputFile.close();
35     }
36 }
```

Program Output with Example Input Shown in Bold

```
Enter the filename: MyFriends.txt [Enter]
Joe
Rose
Greg
Kirk
Renee
```

The file is opened and a `Scanner` object to read it is created in line 21. The loop in lines 24 through 31 reads all of the lines from the file and displays them. In line 24 the loop calls the `Scanner` object's `hasNext` method. If the method returns `true`, then the file has more data to read. In that case, the next line is read from the file in line 27, and is displayed in line 30. The loop repeats until the `hasNext` method returns false in line 24. Figure 4-19 shows the logic of reading a file until the end is reached.

Reading Primitive Values from a File

Recall from Chapter 2 that the `Scanner` class provides methods for reading primitive values. These methods are named `nextByte`, `nextDouble`, `nextFloat`, `nextInt`, `nextLine`, `nextLong`, and `nextShort`. Table 2-17 gives more information on each of these methods, which can be used to read primitive values from a file.

The program in Code Listing 4-20 demonstrates how the `nextDouble` method can be used to read floating-point values from a file. The program reads the contents of a file named *Numbers.txt*. The contents of the *Numbers.txt* file are shown in Figure 4-20. As you can see, the file contains a series of floating-point numbers. The program reads all of the numbers from the file and calculates their total.

Figure 4-19 Logic of reading a file until the end is reached

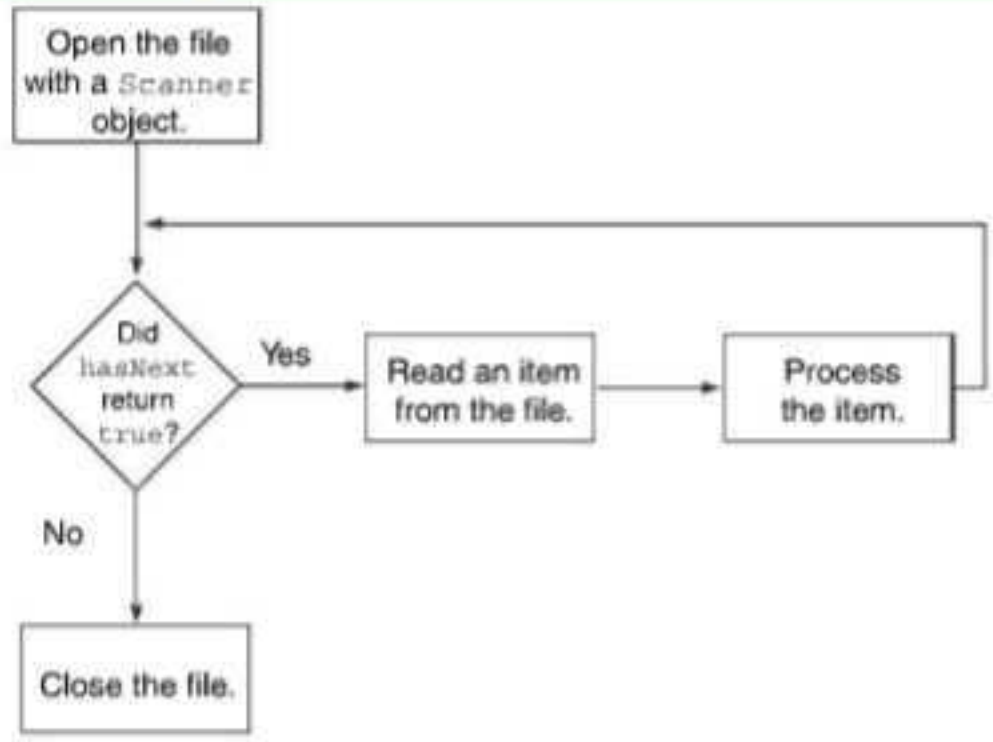


Figure 4-20 Contents of *Numbers.txt*

Code Listing 4-20 (FileSum.java)

```
1  import java.util.Scanner;
2  import java.io.*;
3
4  /**
5     This program reads a series of numbers from a file and
6     accumulates their sum.
7  */
8
9  public class FileSum
10 {
11    public static void main(String[] args) throws IOException
12    {
```