

2.13 Reading Keyboard Input

CONCEPT: Objects of the **Scanner** class can be used to read input from the keyboard.

Previously we discussed the `System.out` object, and how it refers to the standard output device. The Java API has another object, `System.in`, which refers to the standard input device. The *standard input device* is normally the keyboard. You can use the `System.in` object to read keystrokes that have been typed at the keyboard. However, using `System.in` is not as simple and straightforward as using `System.out` because the `System.in` object reads input only as byte values. This isn't very useful because programs normally require values of other data types as input. To work around this, you can use the `System.in` object in conjunction with an object of the `Scanner` class. The `Scanner` class is designed to read input from a source (such as `System.in`), and it provides methods that you can use to retrieve the input formatted as primitive values or strings.

First, you create a `Scanner` object and connect it to the `System.in` object. Here is an example of a statement that does just that:

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

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

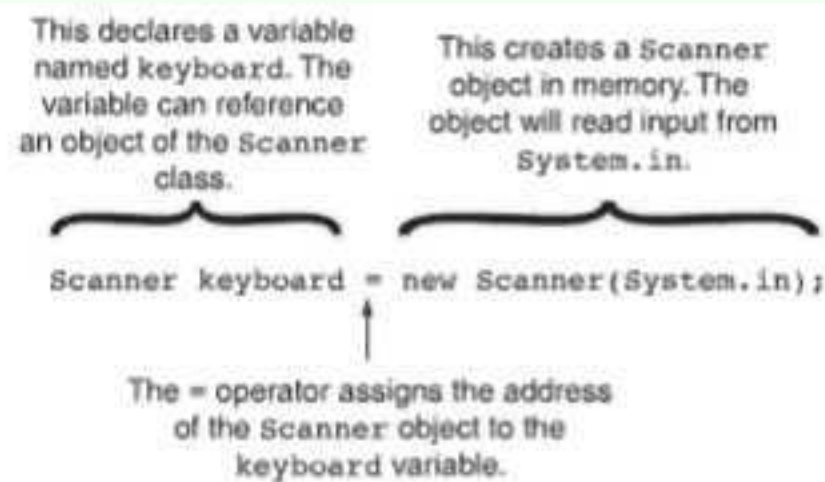
```
Scanner keyboard
```

declares a variable named `keyboard`. The data type of the variable is `Scanner`. Because `Scanner` is a class, the `keyboard` variable is a class type variable. Recall from our discussion on `String` objects that a class type variable holds the memory address of an object. Therefore, the `keyboard` variable will be used to hold the address of a `Scanner` object. The second part of the statement is as follows:

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

The first thing we see in this part of the statement is the assignment operator (`=`). The assignment operator will assign something to the `keyboard` variable. After the assignment operator we see the word `new`, which is a Java key word. The purpose of the `new` key word is to create an object in memory. The type of object that will be created is listed next. In this case, we see `Scanner(System.in)` listed after the `new` key word. This specifies that a `Scanner` object should be created, and it should be connected to the `System.in` object. The memory address of the object is assigned (by the `=` operator) to the variable `keyboard`. After the statement executes, the `keyboard` variable will reference the `Scanner` object that was created in memory.

Figure 2-12 points out the purpose of each part of this statement. Figure 2-13 illustrates how the `keyboard` variable references an object of the `Scanner` class.

Figure 2-12 The parts of the statement**Figure 2-13** The keyboard variable references a Scanner object

NOTE: In the preceding code, we chose keyboard as the variable name. There is nothing special about the name keyboard. We simply chose that name because we will use the variable to read input from the keyboard.

The scanner class has methods for reading strings, bytes, integers, long integers, short integers, floats, and doubles. For example, the following code uses an object of the Scanner class to read an int value from the keyboard and assign the value to the number variable.

```
int number;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter an integer value: ");
number = keyboard.nextInt();
```

The last statement shown here calls the scanner class's nextInt method. The nextInt method formats an input value as an int, and then returns that value. Therefore, this statement formats the input that was entered at the keyboard as an int, and then returns it. The value is assigned to the number variable.

Table 2-17 lists several of the scanner class's methods and describes their use.

Table 2-17 Some of the `Scanner` class methods

Method	Example and Description
<code>nextByte</code>	<p>Example Usage:</p> <pre>byte x; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a byte value: "); x = keyboard.nextByte();</pre> <p>Description: Returns input as a byte.</p>
<code>nextDouble</code>	<p>Example Usage:</p> <pre>double number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a double value: "); number = keyboard.nextDouble();</pre> <p>Description: Returns input as a double.</p>
<code>nextFloat</code>	<p>Example Usage:</p> <pre>float number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a float value: "); number = keyboard.nextFloat();</pre> <p>Description: Returns input as a float.</p>
<code>nextInt</code>	<p>Example Usage:</p> <pre>int number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter an integer value: "); number = keyboard.nextInt();</pre> <p>Description: Returns input as an int.</p>
<code>nextLine</code>	<p>Example Usage:</p> <pre>String name; Scanner keyboard = new Scanner(System.in); System.out.print("Enter your name: "); name = keyboard.nextLine();</pre> <p>Description: Returns input as a String.</p>
<code>nextLong</code>	<p>Example Usage:</p> <pre>long number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a long value: "); number = keyboard.nextLong();</pre> <p>Description: Returns input as a long.</p>
<code>nextShort</code>	<p>Example Usage:</p> <pre>short number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a short value: "); number = keyboard.nextShort();</pre> <p>Description: Returns input as a short.</p>

Using the import Statement

There is one last detail about the Scanner class that you must know before you will be ready to use it. The Scanner class is not automatically available to your Java programs. Any program that uses the Scanner class should have the following statement near the beginning of the file, before any class definition:

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

This statement tells the Java compiler where in the Java library to find the Scanner class, and makes it available to your program.

Code Listing 2-29 shows the Scanner class being used to read a String, an int, and a double.

Code Listing 2-29 (Payroll.java)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program demonstrates the Scanner class.
5  */
6
7 public class Payroll
8 {
9     public static void main(String[] args)
10    {
11        String name;           // To hold a name
12        int hours;             // Hours worked
13        double payRate;        // Hourly pay rate
14        double grossPay;       // Gross pay
15
16        // Create a Scanner object to read input.
17        Scanner keyboard = new Scanner(System.in);
18
19        // Get the user's name.
20        System.out.print("What is your name? ");
21        name = keyboard.nextLine();
22
23        // Get the number of hours worked this week.
24        System.out.print("How many hours did you work this week? ");
25        hours = keyboard.nextInt();
26
27        // Get the user's hourly pay rate.
28        System.out.print("What is your hourly pay rate? ");
29        payRate = keyboard.nextDouble();
30
31        // Calculate the gross pay.
32        grossPay = hours * payRate;
33    }
```



```

34         // Display the resulting information.
35         System.out.println("Hello, " + name);
36         System.out.println("Your gross pay is $" + grossPay);
37     }
38 }

```

Program Output with Example Input Shown in Bold

```

What is your name? Joe Mahoney [Enter]
How many hours did you work this week? 40 [Enter]
What is your hourly pay rate? 20 [Enter]
Hello, Joe Mahoney
Your gross pay is $800.0

```



NOTE: Notice that each `Scanner` class method that we used waits for the user to press the **Enter** key before it returns a value. When the **Enter** key is pressed, the cursor automatically moves to the next line for subsequent output operations.

Reading a Character

Sometimes you will want to read a single character from the keyboard. For example, your program might ask the user a yes/no question, and specify that he or she type Y for yes or N for no. The `Scanner` class does not have a method for reading a single character, however. The approach that we will use in this book for reading a character is to use the `Scanner` class's `nextLine` method to read a string from the keyboard, and then use the `String` class's `charAt` method to extract the first character of the string. This will be the character that the user entered at the keyboard. Here is an example:

```

String input; // To hold a line of input
char answer;  // To hold a single character

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

// Ask the user a question.
System.out.print("Are you having fun? (Y=yes, N=no) ");
input = keyboard.nextLine(); // Get a line of input.
answer = input.charAt(0);    // Get the first character.

```

The `input` variable references a `String` object. The last statement in this code calls the `String` class's `charAt` method to retrieve the character at position 0, which is the first character in the string. After this statement executes, the `answer` variable will hold the character that the user typed at the keyboard.

Mixing Calls to `nextLine` with Calls to Other `Scanner` Methods

When you call one of the `Scanner` class's methods to read a primitive value, such as `nextInt` or `nextDouble`, and then call the `nextLine` method to read a string, an annoying and hard-to-find problem can occur. For example, look at the program in Code Listing 2-30.

Code Listing 2-30 (InputProblem.java)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /*
4  This program has a problem reading input.
5  */
6
7 public class InputProblem
8 {
9     public static void main(String[] args)
10    {
11        String name;    // To hold the user's name
12        int age;         // To hold the user's age
13        double income;  // To hold the user's income
14
15        // Create a Scanner object to read input.
16        Scanner keyboard = new Scanner(System.in);
17
18        // Get the user's age.
19        System.out.print("What is your age? ");
20        age = keyboard.nextInt();
21
22        // Get the user's income
23        System.out.print("What is your annual income? ");
24        income = keyboard.nextDouble();
25
26        // Get the user's name.
27        System.out.print("What is your name? ");
28        name = keyboard.nextLine();
29
30        // Display the information back to the user.
31        System.out.println("Hello, " + name + ". Your age is " +
32                           age + " and your income is $" +
33                           income);
34    }
35 }
```

Program Output with Example Input Shown in Bold

```
What is your age? 24 [Enter]
What is your annual income? 50000.00 [Enter]
What is your name? Hello, . Your age is 24 and your income is $50000.0
```

Notice in the example output that the program first allows the user to enter his or her age. The statement in line 20 reads an `int` from the keyboard and stores the value in the `age` variable. Next, the user enters his or her income. The statement in line 24 reads a `double` from the keyboard and stores the value in the `income` variable. Then the user is asked to

enter his or her name, but it appears that the statement in line 28 is skipped. The name is never read from the keyboard. This happens because of a slight difference in behavior between the `nextLine` method and the other `Scanner` class methods.

When the user types keystrokes at the keyboard, those keystrokes are stored in an area of memory that is sometimes called the *keyboard buffer*. Pressing the `[Enter]` key causes a new-line character to be stored in the keyboard buffer. In the example running of the program in Code Listing 2-30, the user was asked to enter his or her age, and the statement in line 20 called the `nextInt` method to read an integer from the keyboard buffer. Notice that the user typed 24 and then pressed the `[Enter]` key. The `nextInt` method read the value 24 from the keyboard buffer, and then stopped when it encountered the newline character. So the value 24 was read from the keyboard buffer, but the newline character was not read. The newline character remained in the keyboard buffer.

Next, the user was asked to enter his or her annual income. The user typed 50000.00 and then pressed the `[Enter]` key. When the `nextDouble` method in line 24 executed, it first encountered the newline character that was left behind by the `nextInt` method. This does not cause a problem because the `nextDouble` method is designed to skip any leading newline characters it encounters. It skips over the initial newline, reads the value 50000.00 from the keyboard buffer, and stops reading when it encounters the next newline character. This newline character is then left in the keyboard buffer.

Next, the user is asked to enter his or her name. In line 28 the `nextLine` method is called. The `nextLine` method, however, is not designed to skip over an initial newline character. If a newline character is the first character that the `nextLine` method encounters, then nothing will be read. Because the `nextDouble` method, back in line 24, left a newline character in the keyboard buffer, the `nextLine` method will not read any input. Instead, it will immediately terminate and the user will not be given a chance to enter his or her name.

Although the details of this problem might seem confusing, the solution is easy. The program in Code Listing 2-31 is a modification of Code Listing 2-30, with the input problem fixed.

Code Listing 2-31 (CorrectedInputProblem.java)

```

1  import java.util.Scanner; // Needed for the Scanner class
2
3  /*
4   This program correctly reads numeric and string input.
5  */
6
7  public class CorrectedInputProblem
8  {
9      public static void main(String[] args)
10     {
11         String name; // To hold the user's name

```

```
12     int age;           // To hold the user's age
13     double income;    // To hold the user's income
14
15     // Create a Scanner object to read input.
16     Scanner keyboard = new Scanner(System.in);
17
18     // Get the user's age.
19     System.out.print("What is your age? ");
20     age = keyboard.nextInt();
21
22     // Get the user's income
23     System.out.print("What is your annual income? ");
24     income = keyboard.nextDouble();
25
26     // Consume the remaining newline.
27     keyboard.nextLine();
28
29     // Get the user's name.
30     System.out.print("What is your name? ");
31     name = keyboard.nextLine();
32
33     // Display the information back to the user.
34     System.out.println("Hello, " + name + ". Your age is " +
35                        age + " and your income is $" +
36                        income);
37 }
38 }
```

Program Output with Example Input Shown in Bold

```
What is your age? 24 [Enter]
What is your annual income? 50000.00 [Enter]
What is your name? Mary Simpson [Enter]
Hello, Mary Simpson. Your age is 24 and your income is $50000.0
```

Notice that after the user's income is read by the `nextDouble` method in line 24, the `nextLine` method is called in line 27. The purpose of this call is to consume, or remove, the newline character that remains in the keyboard buffer. Then, in line 31, the `nextLine` method is called again. This time it correctly reads the user's name.



NOTE: Notice that in line 27, where we consume the remaining newline character, we do not assign the method's return value to any variable. This is because we are simply calling the method to remove the newline character, and we do not need to keep the method's return value.

2.14 Dialog Boxes

CONCEPT: The `JOptionPane` class allows you to quickly display a dialog box, which is a small graphical window displaying a message or requesting input.

A *dialog box* is a small graphical window that displays a message to the user or requests input. You can quickly display dialog boxes with the `JOptionPane` class. In this section we will discuss the following types of dialog boxes and how you can display them using `JOptionPane`:

- **Message Dialog** A dialog box that displays a message; an OK button is also displayed
- **Input Dialog** A dialog box that prompts the user for input and provides a text field where input is typed; an OK button and a Cancel button are also displayed

Figure 2-14 shows an example of each type of dialog box.

Figure 2-14 A message dialog and an input dialog



The `JOptionPane` class is not automatically available to your Java programs. Any program that uses the `JOptionPane` class must have the following statement near the beginning of the file:

```
import javax.swing.JOptionPane;
```

This statement tells the compiler where to find the `JOptionPane` class and makes it available to your program.

Displaying Message Dialogs

The `showMessageDialog` method is used to display a message dialog. Here is a statement that calls the method:

```
JOptionPane.showMessageDialog(null, "Hello World");
```

The first argument is only important in programs that display other graphical windows. You will learn more about this in Chapter 12. Until then, we will always pass the key word `null` as the first argument. This causes the dialog box to be displayed in the center of the screen. The second argument is the message that we wish to display in the dialog box. This code will cause the dialog box in Figure 2-15 to appear. When the user clicks the OK button, the dialog box will close.

Figure 2-15 Message dialog



Displaying Input Dialogs

An input dialog is a quick and simple way to ask the user to enter data. You use the `JOptionPane` class's `showInputDialog` method to display an input dialog. The following code calls the method:

```
String name;  
name = JOptionPane.showInputDialog("Enter your name.");
```

The argument passed to the method is a message to display in the dialog box. This statement will cause the dialog box shown in Figure 2-16 to be displayed in the center of the screen. If the user clicks the OK button, `name` will reference the string value entered by the user into the text field. If the user clicks the Cancel button, `name` will reference the special value `null`.

Figure 2-16 Input dialog



An Example Program

The program in Code Listing 2-32 demonstrates how to use both types of dialog boxes. This program uses input dialogs to ask the user to enter his or her first, middle, and last names, and then displays a greeting with a message dialog. When this program executes, the dialog boxes shown in Figure 2-17 will be displayed, one at a time.

Code Listing 2-32 (NamesDialog.java)

```

1  import javax.swing.JOptionPane;
2
3  /**
4   * This program demonstrates using dialogs with
5   * JOptionPane.
6   */
7
8  public class NamesDialog
9  {
10     public static void main(String[] args)
11     {
12         String firstName; // The user's first name
13         String middleName; // The user's middle name
14         String lastName; // The user's last name
15
16         // Get the user's first name.
17         firstName =
18             JOptionPane.showInputDialog("What is " +
19                                     "your first name? ");
20
21         // Get the user's middle name.
22         middleName =
23             JOptionPane.showInputDialog("What is " +
24                                     "your middle name? ");
25
26         // Get the user's last name.
27         lastName =
28             JOptionPane.showInputDialog("What is " +
29                                     "your last name? ");
30
31         // Display a greeting
32         JOptionPane.showMessageDialog(null, "Hello " +
33                                     firstName + " " + middleName +
34                                     " " + lastName);
35         System.exit(0);
36     }
37 }

```

Notice the last statement in the `main` method:

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

This statement causes the program to end, and is required if you use the `JOptionPane` class to display dialog boxes. Unlike a console program, a program that uses `JOptionPane` does not automatically stop executing when the end of the `main` method is reached, because the `JOptionPane` class causes an additional task to run in the JVM. If the `System.exit` method

Figure 2-17 Dialog boxes displayed by the `NamesDialog` program

The first dialog box appears as shown here. The user types Joe and clicks OK.



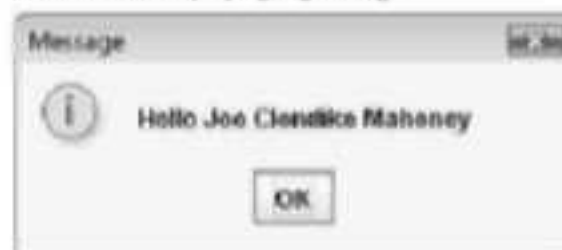
The second dialog box appears, as shown here. In this example the user types Clondike and clicks OK.



The third dialog box appears, as shown here. In this example the user types Mahoney and clicks OK.



The fourth dialog box appears, as shown here, displaying a greeting.



is not called, this task, also known as a *thread*, will continue to execute, even after the end of the `main` method has been reached.

The `System.exit` method requires an integer argument. This argument is an exit code that is passed back to the operating system. Although this code is usually ignored, it can be used outside the program to indicate whether the program ended successfully or as the result of a failure. The value 0 traditionally indicates that the program ended successfully.

Converting String Input to Numbers

Unlike the `Scanner` class, the `JOptionPane` class does not have different methods for reading values of different data types as input. The `showInputDialog` method always returns the

user's input as a `String`, even if the user enters numeric data. For example, if the user enters the number 72 into an input dialog, the `showInputDialog` method will return the string "72". This can be a problem if you wish to use the user's input in a math operation because, as you know, you cannot perform math on strings. In such a case, you must convert the input to a numeric value. To convert a string value to a numeric value, you use one of the methods listed in Table 2-18.

Table 2-18 Methods for converting strings to numbers

Method	Use This Method To ...	Example Code
<code>Byte.parseByte</code>	Convert a string to a byte.	<pre>byte num; num = Byte.parseByte(str);</pre>
<code>Double.parseDouble</code>	Convert a string to a double.	<pre>double num; num = Double.parseDouble(str);</pre>
<code>Float.parseFloat</code>	Convert a string to a float.	<pre>float num; num = Float.parseFloat(str);</pre>
<code>Integer.parseInt</code>	Convert a string to an int.	<pre>int num; num = Integer.parseInt(str);</pre>
<code>Long.parseLong</code>	Convert a string to a long.	<pre>long num; num = Long.parseLong(str);</pre>
<code>Short.parseShort</code>	Convert a string to a short.	<pre>short num; num = Short.parseShort(str);</pre>



NOTE: The methods in Table 2-18 are part of Java's wrapper classes, which you will learn more about in Chapter 9.

Here is an example of how you would use the `Integer.parseInt` method to convert the value returned from the `JOptionPane.showInputDialog` method to an `int`:

```
int number;
String str;
str = JOptionPane.showInputDialog("Enter a number.");
number = Integer.parseInt(str);
```

After this code executes, the `number` variable will hold the value entered by the user, converted to an `int`. Here is an example of how you would use the `Double.parseDouble` method to convert the user's input to a `double`:

```
double price;
String str;
str = JOptionPane.showInputDialog("Enter the retail price.");
price = Double.parseDouble(str);
```

After this code executes, the `price` variable will hold the value entered by the user, converted to a `double`. Code Listing 2-33 shows a complete program. This is a modification of the `Payroll.java` program in Code Listing 2-29. When this program executes, the dialog boxes shown in Figure 2-18 will be displayed, one at a time.

Code Listing 2-33 (PayrollDialog.java)

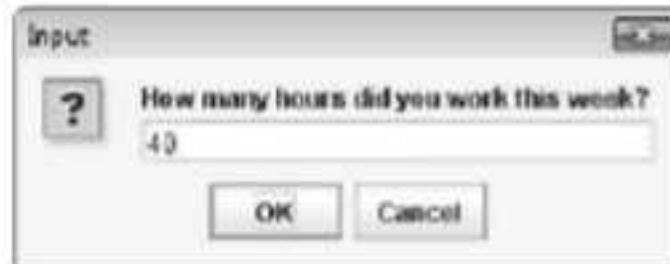
```
1 import javax.swing.JOptionPane;
2
3 /**
4  * This program demonstrates using dialogs with
5  * JOptionPane.
6  */
7
8 public class PayrollDialog
9 {
10     public static void main(String[] args)
11     {
12         String inputString;    // For reading input
13         String name;           // The user's name
14         int hours;             // The number of hours worked
15         double payRate;        // The user's hourly pay rate
16         double grossPay;       // The user's gross pay
17
18         // Get the user's name.
19         name = JOptionPane.showInputDialog("What is " +
20                                         "your name? ");
21
22         // Get the hours worked.
23         inputString =
24             JOptionPane.showInputDialog("How many hours " +
25                                         "did you work this week? ");
26
27         // Convert the input to an int.
28         hours = Integer.parseInt(inputString);
29
30         // Get the hourly pay rate.
31         inputString =
32             JOptionPane.showInputDialog("What is your " +
33                                         "hourly pay rate? ");
34
35         // Convert the input to a double.
36         payRate = Double.parseDouble(inputString);
37
38         // Calculate the gross pay.
39         grossPay = hours * payRate;
40
41         // Display the results.
42         JOptionPane.showMessageDialog(null, "Hello " +
43                                         name + ". Your gross pay is $" +
44                                         grossPay);
45
46         // End the program.
47         System.exit(0);
48     }
49 }
```


Figure 2-18 Dialog boxes displayed by `PayrollDialog.java`

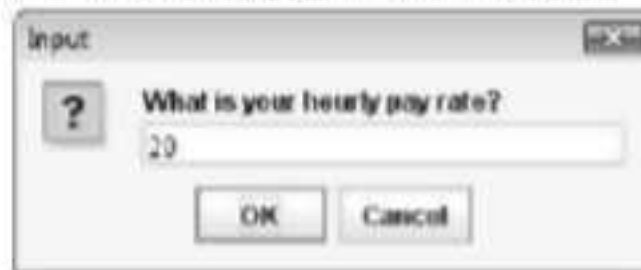
The first dialog box appears as shown here. The user enters his or her name and then clicks OK.



The second dialog box appears, as shown here. The user enters the number of hours worked and then clicks OK.



The third dialog box appears, as shown here. The user enters his or her hourly pay rate and then clicks OK.



The fourth dialog box appears, as shown here.



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

- 2.34 What is the purpose of the following types of dialog boxes?
- Message dialog
 - Input dialog
- 2.35 Write code that will display each of the dialog boxes shown in Figure 2-19.

Figure 2-19 Dialog boxes

- 2.36 Write code that displays an input dialog asking the user to enter his or her age. Convert the input value to an `int` and store it in an `int` variable named `age`.
- 2.37 What `import` statement do you write in a program that uses the `JOptionPane` class?

2.15 Common Errors to Avoid

- **Mismatched braces, quotation marks, or parentheses.** In this chapter you saw that the statements making up a class definition are enclosed in a set of braces. Also, you saw that the statements in a method are also enclosed in a set of braces. For every opening brace, there must be a closing brace in the proper location. The same is true of double-quotation marks that enclose string literals and single-quotation marks that enclose character literals. Also, in a statement that uses parentheses, such as a mathematical expression, you must have a closing parenthesis for every opening parenthesis.
- **Misspelling key words.** Java will not recognize a key word that has been misspelled.
- **Using capital letters in key words.** Remember that Java is a case-sensitive language, and all key words are written in lowercase. Using an uppercase letter in a key word is the same as misspelling the key word.
- **Using a key word as a variable name.** The key words are reserved for special uses; they cannot be used for any other purpose.
- **Using inconsistent spelling of variable names.** Each time you use a variable name, it must be spelled exactly as it appears in its declaration statement.
- **Using inconsistent case of letters in variable names.** Because Java is a case-sensitive language, it distinguishes between uppercase and lowercase letters. Java will not recognize a variable name that is not written exactly as it appears in its declaration statement.
- **Inserting a space in a variable name.** Spaces are not allowed in variable names. Instead of using a two-word name such as `gross pay`, use one word, such as `grossPay`.
- **Forgetting the semicolon at the end of a statement.** A semicolon appears at the end of each complete statement in Java.
- **Assigning a `double` literal to a `float` variable.** Java is a strongly typed language, which means that it only allows you to store values of compatible data types in variables. All floating-point literals are treated as `doubles`, and a `double` value is not compatible with a `float` variable. A floating-point literal must end with the letter `F` or `f` in order to be stored in a `float` variable.
- **Using commas or other currency symbols in numeric literals.** Numeric literals cannot contain commas or currency symbols, such as the dollar sign.
- **Unintentionally performing integer division.** When both operands of a division statement are integers, the statement will result in an integer. If there is a remainder, it will be discarded.
- **Forgetting to group parts of a mathematical expression.** If you use more than one operator in a mathematical expression, the expression will be evaluated according to the order of operations. If you wish to change the order in which the operators are used, you must use parentheses to group part of the expression.
- **Inserting a space in a combined assignment operator.** A space cannot appear between the two operators that make a combined assignment operator.
- **Using a variable to receive the result of a calculation when the variable's data type is incompatible with the data type of the result.** A variable that receives the result of a calculation must be of a data type that is compatible with the data type of the result.

- **Incorrectly terminating a multi-line comment or a documentation comment.** Multi-line comments and documentation comments are terminated by the `*/` characters. Forgetting to place these characters at a comment's desired ending point, or accidentally switching the `*` and the `/`, will cause the comment not to have an ending point.
- **Forgetting to use the correct `import` statement in a program that uses the `Scanner` class or the `JOptionPane` class.** In order for the `Scanner` class to be available to your program, you must have the `import java.util.Scanner;` statement near the top of your program file. In order for the `JOptionPane` class to be available to your program, you must have the `import javax.swing.JOptionPane;` statement near the top of the program file.
- **When using an input dialog to read numeric input, not converting the `showInputDialog` method's return value to a number.** The `showInputDialog` method always returns the user's input as a string. If the user enters a numeric value, it must be converted to a number before it can be used in a math statement.

Review Questions and Exercises

Multiple Choice and True/False

1. Every complete statement ends with a _____.
 - a. period
 - b. parenthesis
 - c. semicolon
 - d. ending brace
2. The following data


```
72
'A'
"Hello World"
2.8712
```

 are all examples of _____.
 - a. variables
 - b. literals
 - c. strings
 - d. none of these
3. A group of statements, such as the contents of a class or a method, are enclosed in _____.
 - a. braces `{ }`
 - b. parentheses `()`
 - c. brackets `[]`
 - d. any of these will do
4. Which of the following are *not* valid assignment statements? (Indicate all that apply.)
 - a. `total = 9;`
 - b. `72 = amount;`
 - c. `profit = 129`
 - d. `letter = 'W';`

5. Which of the following are not valid `println` statements? (Indicate all that apply.)
 - a. `System.out.println + "Hello World";`
 - b. `System.out.println("Have a nice day");`
 - c. `out.System.println(value);`
 - d. `println.out(Programming is great fun);`
6. The negation operator is _____.
 - a. unary
 - b. binary
 - c. ternary
 - d. none of these
7. This key word is used to declare a named constant.
 - a. `constant`
 - b. `namedConstant`
 - c. `final`
 - d. `concrete`
8. These characters mark the beginning of a multi-line comment.
 - a. `//`
 - b. `/*`
 - c. `*/`
 - d. `/**`
9. These characters mark the beginning of a single-line comment.
 - a. `//`
 - b. `/*`
 - c. `*/`
 - d. `/**`
10. These characters mark the beginning of a documentation comment.
 - a. `//`
 - b. `/*`
 - c. `*/`
 - d. `/**`
11. Which `Scanner` class method would you use to read a string as input?
 - a. `nextString`
 - b. `nextLine`
 - c. `readString`
 - d. `getline`
12. Which `Scanner` class method would you use to read a double as input?
 - a. `nextDouble`
 - b. `getDouble`
 - c. `readDouble`
 - d. None of these; you cannot read a double with the `Scanner` class
13. You can use this class to display dialog boxes.
 - a. `JOptionPane`
 - b. `BufferedReader`
 - c. `InputStreamReader`
 - d. `DialogBox`

14. When Java converts a lower-ranked value to a higher-ranked type, it is called a(n) _____.
 - a. 4-bit conversion
 - b. escalating conversion
 - c. widening conversion
 - d. narrowing conversion
15. This type of operator lets you manually convert a value, even if it means that a narrowing conversion will take place.
 - a. cast
 - b. binary
 - c. uploading
 - d. dot
16. True or False: A left brace in a Java program is always followed by a right brace later in the program.
17. True or False: A variable must be declared before it can be used.
18. True or False: Variable names may begin with a number.
19. True or False: You cannot change the value of a variable whose declaration uses the `final` key word.
20. True or False: Comments that begin with `//` can be processed by `javadoc`.
21. True or False: If one of an operator's operands is a `double`, and the other operand is an `int`, Java will automatically convert the value of the `double` to an `int`.

Predict the Output

What will the following code segments print on the screen?

1.

```
int freeze = 32, boil = 212;
freeze = 0;
boil = 100;
System.out.println(freeze + "\n" + boil + "\n");
```
2.

```
int x = 0, y = 2;
x = y * 4;
System.out.println(x + "\n" + y + "\n");
```
3.

```
System.out.print("I am the incredible");
System.out.print("computing\nmachine");
System.out.print("\nand I will\namaze\n");
System.out.println("you.");
```
4.

```
System.out.print("Be careful\n");
System.out.print("This might/n be a trick ");
System.out.println("question.");
```
5.

```
int a, x = 23;
a = x % 2;
System.out.println(x + "\n" + a);
```

Find the Error

There are a number of syntax errors in the following program. Locate as many as you can.

```

/* What's wrong with this program? */
public MyProgram
{
    public static void main(String[] args);
    }
    int a, b, c    \\ Three integers
    a = 3
    b = 4
    c = a + b
    System.out.println("The value of c is" + C);
}

```

Algorithm Workbench

1. Show how the double variables temp, weight, and age can be declared in one statement.
2. Show how the int variables months, days, and years may be declared in one statement, with months initialized to 2 and years initialized to 3.
3. Write assignment statements that perform the following operations with the variables a, b, and c.
 - a. Adds 2 to a and stores the result in b
 - b. Multiplies b times 4 and stores the result in a
 - c. Divides a by 3.14 and stores the result in b
 - d. Subtracts 8 from b and stores the result in a
 - e. Stores the character 'K' in c
 - f. Stores the Unicode code for 'B' in c
4. Assume the variables result, w, x, y, and z are all integers, and that w = 5, x = 4, y = 8, and z = 2. What value will be stored in result in each of the following statements?
 - a. result = x + y;
 - b. result = z * 2;
 - c. result = y / x;
 - d. result = y - z;
 - e. result = w % 2;
5. How would each of the following numbers be represented in E notation?
 - a. 3.287×10^6
 - b. -9.7865×10^{12}
 - c. 7.65491×10^{-3}
6. Modify the following program so it prints two blank lines between each line of text.

```

public class
{
    public static void main(String[] args)
    {
        System.out.print("Hearing in the distance");
        System.out.print("Two mandolins like creatures in the");
        System.out.print("dark");
        System.out.print("Creating the agony of ecstasy.");
        System.out.println("                - George Barker");
    }
}

```


7. What will the following code output?

```
int apples = 0, bananas = 2, pears = 10;
apples += 10;
bananas *= 10;
pears /= 10;
System.out.println(apples + " " +
                    bananas + " " +
                    pears);
```
8. What will the following code output?

```
double d = 12.9;
int i = (int)d;
System.out.println(i);
```
9. What will the following code output?

```
String message = "Have a great day!";
System.out.println(message.charAt(5));
```
10. What will the following code output?

```
String message = "Have a great day!";
System.out.println(message.toUpperCase());
System.out.println(message);
```
11. Convert the following pseudocode to Java code. Be sure to declare the appropriate variables.
Store 20 in the speed variable.
Store 10 in the time variable.
Multiply speed by time and store the result in the distance variable.
Display the contents of the distance variable.
12. Convert the following pseudocode to Java code. Be sure to declare the appropriate variables.
Store 172.5 in the force variable.
Store 27.5 in the area variable.
Divide area by force and store the result in the pressure variable.
Display the contents of the pressure variable.
13. Write the code to set up all the necessary objects for reading keyboard input. Then write code that asks the user to enter his or her desired annual income. Store the input in a double variable.
14. Write the code to display a dialog box that asks the user to enter his or her desired annual income. Store the input in a double variable.
15. A program has a float variable named total and a double variable named number. Write a statement that assigns number to total without causing an error when compiled.

Short Answer

1. Is the following comment a single-line style comment or a multi-line style comment?

```
/* This program was written by M. A. Codewriter */
```
2. Is the following comment a single-line style comment or a multi-line style comment?

```
// This program was written by M. A. Codewriter
```
3. Describe what the phrase “self-documenting program” means.

4. What is meant by “case-sensitive”? Why is it important for a programmer to know that Java is a case-sensitive language?
5. Briefly explain how the `print` and `println` methods are related to the `System` class and the `out` object.
6. What does a variable declaration tell the Java compiler about a variable?
7. Why are variable names like `x` not recommended?
8. What things must be considered when deciding on a data type to use for a variable?
9. Briefly describe the difference between variable assignment and variable initialization.
10. What is the difference between comments that start with the `//` characters and comments that start with the `/*` characters?
11. Briefly describe what programming style means. Why should your programming style be consistent?
12. Assume that a program uses the named constant `PI` to represent the value 3.14. The program uses the named constant in several statements. What is the advantage of using the named constant instead of the actual value 3.14 in each statement?
13. Assume the file *SalesAverage.java* is a Java source file that contains documentation comments. Assuming you are in the same folder or directory as the source code file, what command would you enter at the operating system command prompt to generate the HTML documentation files?
14. An expression adds a `byte` variable and a `short` variable. Of what data type will the result be?

Programming Challenges

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

1. Name, Age, and Annual Income

Write a program that declares the following:

- a `String` variable named `name`
- an `int` variable named `age`
- a `double` variable named `annualPay`

Store your age, name, and desired annual income as literals in these variables. The program should display these values on the screen in a manner similar to the following:

```
My name is Joe Mahoney, my age is 26 and  
I hope to earn $100000.0 per year.
```

2. Name and Initials

Write a program that has the following `String` variables: `firstName`, `middleName`, and `lastName`. Initialize these with your first, middle, and last names. The program should also have the following `char` variables: `firstInitial`, `middleInitial`, and `lastInitial`. Store your first, middle, and last initials in these variables. The program should display the contents of these variables on the screen.

3. Personal Information

Write a program that displays the following information, each on a separate line:

- Your name
- Your address, with city, state, and ZIP
- Your telephone number
- Your college major

Although these items should be displayed on separate output lines, use only a single `println` statement in your program.

4. Star Pattern

Write a program that displays the following pattern:

```

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

5. Sum of Two Numbers

Write a program that stores the integers 62 and 99 in variables, and stores their sum in a variable named `total`.

6. Sales Prediction

The East Coast sales division of a company generates 62 percent of total sales. Based on that percentage, write a program that will predict how much the East Coast division will generate if the company has \$4.6 million in sales this year. *Hint: Use the value 0.62 to represent 62 percent.*

7. Land Calculation

One acre of land is equivalent to 43,560 square feet. Write a program that calculates the number of acres in a tract of land with 389,767 square feet. *Hint: Divide the size of the tract of land by the size of an acre to get the number of acres.*

8. Sales Tax

Write a program that will ask the user to enter the amount of a purchase. The program should then compute the state and county sales tax. Assume the state sales tax is 4 percent and the county sales tax is 2 percent. The program should display the amount of the purchase, the state sales tax, the county sales tax, the total sales tax, and the total of the sale (which is the sum of the amount of purchase plus the total sales tax). *Hint: Use the value 0.02 to represent 2 percent, and 0.04 to represent 4 percent.*



VideoNote

The Miles-per-Gallon Problem

9. Miles-per-Gallon

A car's miles-per-gallon (MPG) can be calculated with the following formula:

$$\text{MPG} = \text{Miles driven} / \text{Gallons of gas used}$$

Write a program that asks the user for the number of miles driven and the gallons of gas used. It should calculate the car's miles-per-gallon and display the result on the screen.

10. Test Average

Write a program that asks the user to enter three test scores. The program should display each test score, as well as the average of the scores.

11. Circuit Board Profit

An electronics company sells circuit boards at a 40 percent profit. If you know the retail price of a circuit board, you can calculate its profit with the following formula:

$$\text{Profit} = \text{Retail price} \times 0.4$$

Write a program that asks the user for the retail price of a circuit board, calculates the amount of profit earned for that product, and displays the results on the screen.

12. String Manipulator

Write a program that asks the user to enter the name of his or her favorite city. Use a `String` variable to store the input. The program should display the following:

- The number of characters in the city name
- The name of the city in all uppercase letters
- The name of the city in all lowercase letters
- The first character in the name of the city

13. Restaurant Bill

Write a program that computes the tax and tip on a restaurant bill. The program should ask the user to enter the charge for the meal. The tax should be 6.75 percent of the meal charge. The tip should be 15 percent of the total after adding the tax. Display the meal charge, tax amount, tip amount, and total bill on the screen.

14. Stock Commission

Kathryn bought 600 shares of stock at a price of \$21.77 per share. She must pay her stock-broker a 2 percent commission for the transaction. Write a program that calculates and displays the following:

- The amount paid for the stock alone (without the commission)
- The amount of the commission
- The total amount paid (for the stock plus the commission)

15. Energy Drink Consumption

A soft drink company recently surveyed 12,467 of its customers and found that approximately 14 percent of those surveyed purchase one or more energy drinks per week. Of those customers who purchase energy drinks, approximately 64 percent of them prefer citrus-flavored energy drinks. Write a program that displays the following:

- The approximate number of customers in the survey who purchase one or more energy drinks per week
- The approximate number of customers in the survey who prefer citrus-flavored energy drinks

16. Word Game

Write a program that plays a word game with the user. The program should ask the user to enter the following:

- His or her name
- His or her age
- The name of a city
- The name of a college
- A profession
- A type of animal
- A pet's name

After the user has entered these items, the program should display the following story, inserting the user's input into the appropriate locations:

```
There once was a person named NAME who lived in CITY. At the age of AGE,  
NAME went to college at COLLEGE. NAME graduated and went to work as a  
PROFESSION. Then, NAME adopted a(n) ANIMAL named PETNAME. They both lived  
happily ever after!
```

17. Stock Transaction Program

Last month Joe purchased some stock in Acme Software, Inc. Here are the details of the purchase:

- The number of shares that Joe purchased was 1,000.
- When Joe purchased the stock, he paid \$32.87 per share.
- Joe paid his stockbroker a commission that amounted to 2% of the amount he paid for the stock.

Two weeks later Joe sold the stock. Here are the details of the sale:

- The number of shares that Joe sold was 1,000.
- He sold the stock for \$33.92 per share.
- He paid his stockbroker another commission that amounted to 2% of the amount he received for the stock.

Write a program that displays the following information:

- The amount of money Joe paid for the stock.
- The amount of commission Joe paid his broker when he bought the stock.
- The amount that Joe sold the stock for.
- The amount of commission Joe paid his broker when he sold the stock.
- Display the amount of profit that Joe made after selling his stock and paying the two commissions to his broker. (If the amount of profit that your program displays is a negative number, then Joe lost money on the transaction.)

TOPICS

- | | |
|---|---|
| 3.1 The <code>if</code> Statement | 3.8 The Conditional Operator (Optional) |
| 3.2 The <code>if-else</code> Statement | 3.9 The <code>switch</code> Statement |
| 3.3 Nested <code>if</code> Statements | 3.10 The <code>System.out.printf</code> Method |
| 3.4 The <code>if-else-if</code> Statement | 3.11 Creating Objects with the <code>DecimalFormat</code> Class |
| 3.5 Logical Operators | 3.12 Common Errors to Avoid |
| 3.6 Comparing String Objects | On the Web: Case Study—Calculating Sales Commission |
| 3.7 More about Variable Declaration and Scope | |

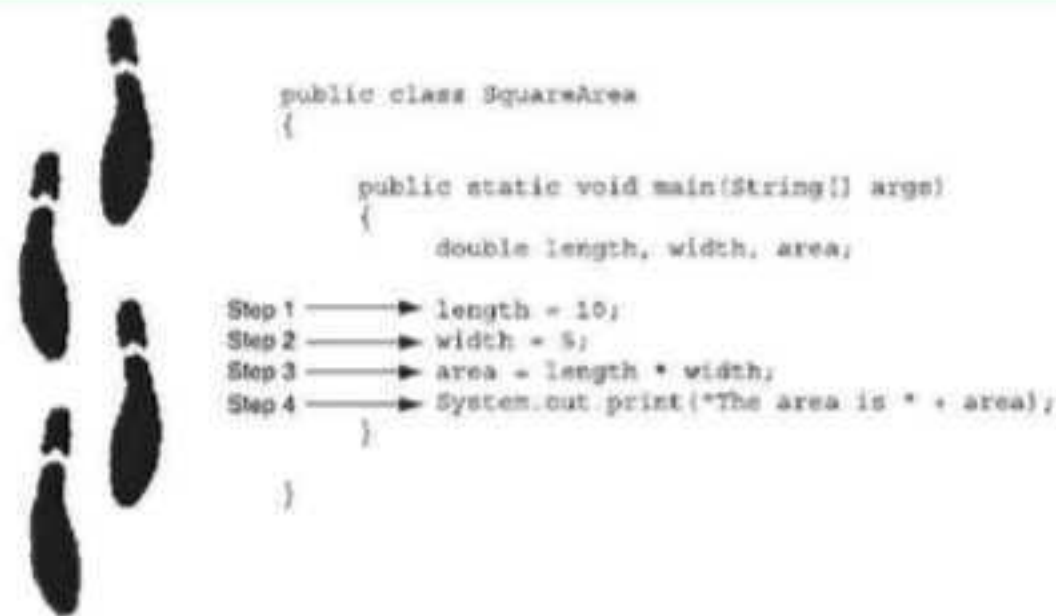
3.1 The `if` Statement

CONCEPT: The `if` statement is used to create a decision structure, which allows a program to have more than one path of execution. The `if` statement causes one or more statements to execute only when a **boolean** expression is **true**.



VideoNote
The `if`
Statement

In all the programs you have written so far, the statements are executed one after the other, in the order they appear. You might think of sequentially executed statements as the steps you take as you walk down a road. To complete the journey, you must start at the beginning and take each step, one after the other, until you reach your destination. This is illustrated in Figure 3-1.

Figure 3-1 Sequence structure

The type of code shown in Figure 3-1 is called a *sequence structure*, because the statements are executed in sequence, without branching off in another direction. Programs often need more than one path of execution, however. Many algorithms require a program to execute some statements only under certain circumstances. This can be accomplished with a *decision structure*.

In a decision structure's simplest form, a specific action is taken only when a condition exists. If the condition does not exist, the action is not performed. The flowchart in Figure 3-2 shows the logic of a decision structure. The diamond symbol represents a yes/no question or a true/false condition. If the answer to the question is yes (or if the condition is true), the program flow follows one path, which leads to an action being performed. If the answer to the question is no (or the condition is false), the program flow follows another path, which skips the action.

Figure 3-2 Simple decision structure logic

In the flowchart, the action “Wear a coat” is performed only when it is cold outside. If it is not cold outside, the action is skipped. The action is *conditionally executed* because it is performed only when a certain condition (cold outside) exists. Figure 3-3 shows a more elaborate flowchart, where three actions are taken only when it is cold outside.

Figure 3-3 Three-action decision structure logic

One way to code a decision structure in Java is with the `if` statement. Here is the general format of the `if` statement:

```

if (BooleanExpression)
    statement;
  
```

The `if` statement is simple in the way it works: The *BooleanExpression* that appears inside the parentheses must be a boolean expression. A *boolean expression* is one that is either true or false. If the boolean expression is true, the very next statement is executed. Otherwise, it is skipped. The statement is conditionally executed because it executes only under the condition that the expression in the parentheses is true.

Using Relational Operators to Form Conditions

Typically, the boolean expression that is tested by an `if` statement is formed with a relational operator. A *relational operator* determines whether a specific relationship exists between two values. For example, the greater than operator (`>`) determines whether one value is greater than another. The equal to operator (`==`) determines whether two values are equal. Table 3-1 lists all of the Java relational operators.

Table 3-1 Relational operators

Relational Operators (in Order of Precedence)	Meaning
<code>></code>	Greater than
<code><</code>	Less than
<code>>=</code>	Greater than or equal to
<code><=</code>	Less than or equal to
<code>==</code>	Equal to
<code>!=</code>	Not equal to

All of the relational operators are binary, which means they use two operands. Here is an example of an expression using the greater than operator:

```
length > width
```

This expression determines whether `length` is greater than `width`. If `length` is greater than `width`, the value of the expression is `true`. Otherwise, the value of the expression is `false`. Because the expression can be only `true` or `false`, it is a boolean expression. The following expression uses the less than operator to determine whether `length` is less than `width`:

```
length < width
```

Table 3-2 shows examples of several boolean expressions that compare the variables `x` and `y`.

Table 3-2 boolean expressions using relational operators

Expression	Meaning
<code>x > y</code>	Is <code>x</code> greater than <code>y</code> ?
<code>x < y</code>	Is <code>x</code> less than <code>y</code> ?
<code>x >= y</code>	Is <code>x</code> greater than or equal to <code>y</code> ?
<code>x <= y</code>	Is <code>x</code> less than or equal to <code>y</code> ?
<code>x == y</code>	Is <code>x</code> equal to <code>y</code> ?
<code>x != y</code>	Is <code>x</code> not equal to <code>y</code> ?

Two of the operators, `>=` and `<=`, test for more than one relationship. The `>=` operator determines whether the operand on its left is greater than or equal to the operand on its right. Assuming that `a` is 4, `b` is 6, and `c` is 4, both of the expressions `b >= a` and `a >= c` are `true`, but `a >= 5` is `false`. When using this operator, the `>` symbol must precede the `=` symbol, and there is no space between them. The `<=` operator determines whether the operand on its left is less than or equal to the operand on its right. Once again, assuming that `a` is 4, `b` is 6, and `c` is 4, both `a <= c` and `b <= 10` are `true`, but `b <= a` is `false`. When using this operator, the `<` symbol must precede the `=` symbol, and there is no space between them.

The `==` operator determines whether the operand on its left is equal to the operand on its right. If both operands have the same value, the expression is `true`. Assuming that `a` is 4, the expression `a == 4` is `true` and the expression `a == 2` is `false`. Notice the equality operator is two `=` symbols together. Don't confuse this operator with the assignment operator, which is one `=` symbol.

The `!=` operator is the not equal operator. It determines whether the operand on its left is not equal to the operand on its right, which is the opposite of the `==` operator. As before, assuming `a` is 4, `b` is 6, and `c` is 4, both `a != b` and `b != c` are `true` because `a` is not equal to `b` and `b` is not equal to `c`. However, `a != c` is `false` because `a` is equal to `c`.

Putting It All Together

Let's look at an example of the `if` statement:

```
if (sales > 50000)
    bonus = 500.0;
```

This statement uses the > operator to determine whether `sales` is greater than 50,000. If the expression `sales > 50000` is true, the variable `bonus` is assigned 500.0. If the expression is false, however, the assignment statement is skipped. The program in Code Listing 3-1 shows another example. The user enters three test scores and the program calculates their average. If the average is greater than 95, the program congratulates the user on obtaining a high score.

Code Listing 3-1 (AverageScore.java)

```
1 import javax.swing.JOptionPane; // Needed for JOptionPane
2
3 /**
4  * This program demonstrates the if statement.
5  */
6
7 public class AverageScore
8 {
9     public static void main(String[] args)
10    {
11        double score1;    // To hold score #1
12        double score2;    // To hold score #2
13        double score3;    // To hold score #3
14        double average;    // To hold the average score
15        String input;    // To hold the user's input
16
17        // Get the first test score.
18        input = JOptionPane.showInputDialog("Enter score #1:");
19        score1 = Double.parseDouble(input);
20
21        // Get the second score.
22        input = JOptionPane.showInputDialog("Enter score #2:");
23        score2 = Double.parseDouble(input);
24
25        // Get the third test score.
26        input = JOptionPane.showInputDialog("Enter score #3:");
27        score3 = Double.parseDouble(input);
28
29        // Calculate the average score.
30        average = (score1 + score2 + score3) / 3.0;
31
32        // Display the average score.
33        JOptionPane.showMessageDialog(null,
34                                     "The average is " + average);
35
36        // If the score was greater than 95, let the user know
37        // that's a great score.
38        if (average > 95)
```

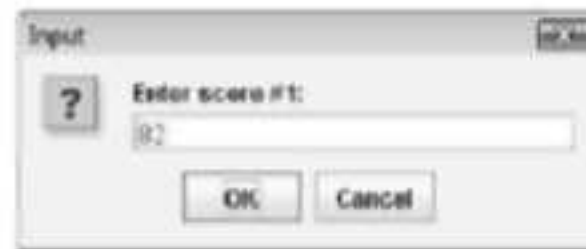


```
39         JOptionPane.showMessageDialog(null,  
40                                     "That's a great score!");  
41  
42     System.exit(0);  
43 }  
44 }
```

Figures 3-4 and 3-5 show examples of interaction with this program. In Figure 3-4 the average of the test scores is not greater than 95. In Figure 3-5 the average is greater than 95.

Figure 3-4 Interaction with the AverageScore program

This input dialog box appears first. The user enters 82 and then clicks on the OK button.



This input dialog box appears next. The user enters 76 and then clicks on the OK button.



This input dialog box appears next. The user enters 91 and then clicks on the OK button.



This message dialog box appears next. The average of the three test scores is displayed.

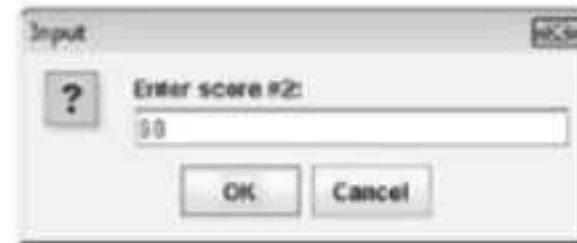


Figure 3-5 Interaction with the AverageScore program

This input dialog box appears first. The user enters 92 and then clicks on the OK button.



This input dialog box appears next. The user enters 98 and then clicks on the OK button.



This input dialog box appears next. The user enters 100 and then clicks on the OK button.



This message dialog box appears next. The average of the three test scores is displayed. The user clicks on the OK button.



This message dialog box appears next because the average is greater than 95.

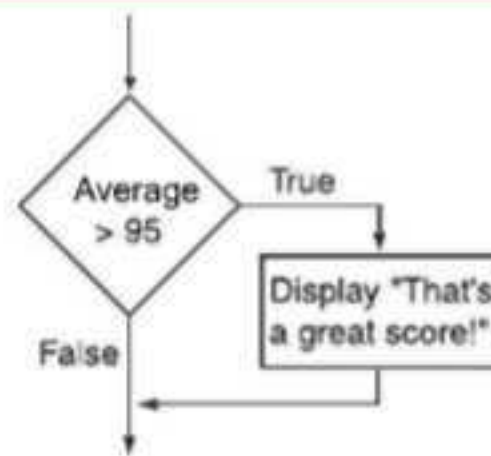


The code in lines 38 through 40 causes the congratulatory message to be printed:

```
if (average > 95)
    JOptionPane.showMessageDialog(null,
        "That's a great score!");
```

Figure 3-6 shows the logic of this if statement.

Table 3-3 shows other examples of if statements and their outcomes.

Figure 3-6 Logic of the `if` statement**Table 3-3** Other examples of `if` statements

Statement	Outcome
<code>if (hours > 40)</code> <code>overTime = true;</code>	If <code>hours</code> is greater than 40, assigns <code>true</code> to the boolean variable <code>overTime</code> .
<code>if (value < 32)</code> <code>System.out.println("Invalid number");</code>	If <code>value</code> is less than 32, displays the message "Invalid number".

Programming Style and the `if` Statement

Even though an `if` statement usually spans more than one line, it is really one long statement. For instance, the following `if` statements are identical except for the style in which they are written:

```

if (value > 32)
    System.out.println("Invalid number.");
if (value > 32) System.out.println("Invalid number.");
  
```

In both of these examples, the compiler considers the `if` statement and the conditionally executed statement as one unit, with a semicolon properly placed at the end. Indentations and spacing are for the human readers of a program, not the compiler. Here are two important style rules you should adopt for writing `if` statements:

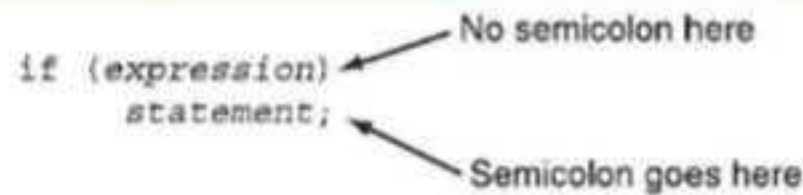
- The conditionally executed statement should appear on the line after the `if` statement.
- The conditionally executed statement should be indented one level from the `if` statement.

In most editors, each time you press the tab key, you are indenting one level. By indenting the conditionally executed statement, you are causing it to stand out visually. This is so you can tell at a glance what part of the program the `if` statement executes. This is a standard way of writing `if` statements and is the method you should use.

Be Careful with Semicolons

You do not put a semicolon after the `if (expression)` portion of an `if` statement, as illustrated in Figure 3-7. This is because the `if` statement isn't complete without its conditionally executed statement.

Figure 3-7 Do not prematurely terminate an `if` statement with a semicolon



If you prematurely terminate an `if` statement with a semicolon, the compiler will not display an error message, but will assume that you are placing a *null statement* there. The null statement, which is an empty statement that does nothing, will become the conditionally executed statement. The statement that you intended to be conditionally executed will be disconnected from the `if` statement and will always execute.

For example, look at the following code:

```
int x = 0, y = 10;

// The following if statement is prematurely
// terminated with a semicolon.
if (x > y);
    System.out.println(x + " is greater than " + y);
```

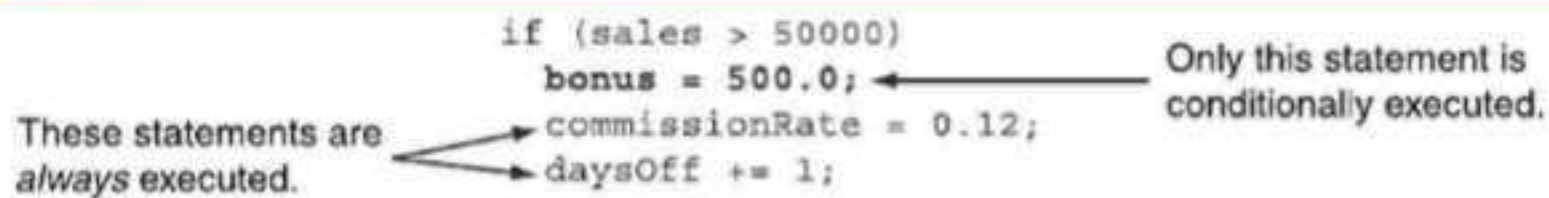
The `if` statement in this code is prematurely terminated with a semicolon. Because the `println` statement is not connected to the `if` statement, it will always execute.

Having Multiple Conditionally Executed Statements

The previous examples of the `if` statement conditionally execute a single statement. The `if` statement can also conditionally execute a group of statements, as long as they are enclosed in a set of braces. Enclosing a group of statements inside braces creates a *block* of statements. Here is an example:

```
if (sales > 50000)
{
    bonus = 500.0;
    commissionRate = 0.12;
    daysOff += 1;
}
```

If `sales` is greater than 50,000, this code will execute all three of the statements inside the braces, in the order they appear. If the braces are accidentally left out, however, the `if` statement conditionally executes only the very next statement. Figure 3-8 illustrates this.

Figure 3-8 An `if` statement missing its braces

Flags

A flag is a boolean variable that signals when some condition exists in the program. When the flag variable is set to `false`, it indicates the condition does not yet exist. When the flag variable is set to `true`, it means the condition does exist.

For example, suppose a program similar to the previous test averaging program has a boolean variable named `highScore`. The variable might be used to signal that a high score has been achieved by the following code:

```

if (average > 95)
    highScore = true;
  
```

Later, the same program might use code similar to the following to test the `highScore` variable, in order to determine whether a high score has been achieved:

```

if (highScore)
    System.out.println("That's a high score!");
  
```

You will find flag variables useful in many circumstances, and we will come back to them in future chapters.

Comparing Characters

You can use the relational operators to test character data as well as numbers. For example, assuming that `ch` is a `char` variable, the following code segment uses the `==` operator to compare it to the character 'A':

```

if (ch == 'A')
    System.out.println("The letter is A.");
  
```

The `!=` operator can also be used with characters to test for inequality. For example, the following statement determines whether the `char` variable `ch` is not equal to the letter 'A':

```

if (ch != 'A')
    System.out.println("Not the letter A.");
  
```

You can also use the `>`, `<`, `>=`, and `<=` operators to compare characters. Computers do not actually store characters, such as A, B, C, and so forth, in memory. Instead, they store numeric codes that represent the characters. Recall from Chapter 2 that Java uses Unicode, which is a set of numbers that represents all the letters of the alphabet (both lowercase and uppercase), the printable digits 0 through 9, punctuation symbols, and special characters. When a character is stored in memory, it is actually the Unicode number that is stored. When the computer is instructed to print the value on the screen, it displays the character that corresponds with the numeric code.



NOTE: Unicode is an international encoding system that is extensive enough to represent all the characters of all the world's alphabets.

In Unicode, letters are arranged in alphabetic order. Because 'A' comes before 'B', the numeric code for the character 'A' is less than the code for the character 'B'. (The code for 'A' is 65 and the code for 'B' is 66. Appendix B, available for download from this book's companion Web site, lists the codes for all of the printable English characters.) In the following `if` statement, the boolean expression `'A' < 'B'` is true:

```
if ('A' < 'B')
    System.out.println("A is less than B.");
```

In Unicode, the uppercase letters come before the lowercase letters, so the numeric code for 'A' (65) is less than the numeric code for 'a' (97). In addition, the space character (code 32) comes before all the alphabetic characters.



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

- 3.1 Write an `if` statement that assigns 0 to `x` when `y` is equal to 20.
- 3.2 Write an `if` statement that multiplies `payRate` by 1.5 if `hours` is greater than 40.
- 3.3 Write an `if` statement that assigns 0.2 to `commission` if `sales` is greater than or equal to 10000.
- 3.4 Write an `if` statement that sets the variable `fees` to 50 if the boolean variable `max` is true.
- 3.5 Write an `if` statement that assigns 20 to the variable `y` and assigns 40 to the variable `z` if the variable `x` is greater than 100.
- 3.6 Write an `if` statement that assigns 0 to the variable `b` and assigns 1 to the variable `c` if the variable `a` is less than 10.
- 3.7 Write an `if` statement that displays "Goodbye" if the variable `myCharacter` contains the character 'D'.

3.2

The if-else Statement

CONCEPT: The `if-else` statement will execute one group of statements if its boolean expression is true, or another group if its boolean expression is false.



VideoNote
The if-else
Statement

The `if-else` statement is an expansion of the `if` statement. Here is its format:

```
if (BooleanExpression)
    statement or block
else
    statement or block
```

Like the `if` statement, a boolean expression is evaluated. If the expression is true, a statement or block of statements is executed. If the expression is false, however, a separate group

of statements is executed. The program in Code Listing 3-2 uses the `if-else` statement to handle a classic programming problem: division by zero. In Java, a program crashes when it divides an integer by 0. When a floating-point value is divided by 0, the program doesn't crash. Instead, the special value `Infinity` is produced as the result of the division.

Code Listing 3-2 (Division.java)

```

1  import java.util.Scanner; // Needed for the Scanner class
2
3  /**
4   * This program demonstrates the if-else statement.
5   */
6
7  public class Division
8  {
9      public static void main(String[] args)
10     {
11         double number1, number2; // Division operands
12         double quotient;         // Result of division
13
14         // Create a Scanner object for keyboard input.
15         Scanner keyboard = new Scanner(System.in);
16
17         // Get the first number.
18         System.out.print("Enter a number: ");
19         number1 = keyboard.nextDouble();
20
21         // Get the second number.
22         System.out.print("Enter another number: ");
23         number2 = keyboard.nextDouble();
24
25         if (number2 == 0)
26         {
27             System.out.println("Division by zero is not possible.");
28             System.out.println("Please run the program again and ");
29             System.out.println("enter a number other than zero.");
30         }
31         else
32         {
33             quotient = number1 / number2;
34             System.out.print("The quotient of " + number1);
35             System.out.print(" divided by " + number2);
36             System.out.println(" is " + quotient);
37         }
38     }
39 }

```

Program Output with Example Input Shown in Bold

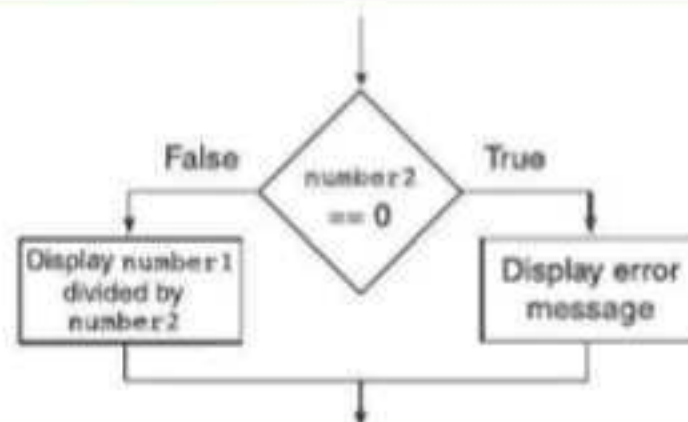
```
Enter a number: 10 [Enter]
Enter another number: 0 [Enter]
Division by zero is not possible.
Please run the program again and
enter a number other than zero.
```

Program Output with Example Input Shown in Bold

```
Enter a number: 10 [Enter]
Enter another number: 5 [Enter]
The quotient of 10 divided by 5 is 2.0
```

The value of `number2` is tested before the division is performed. If the user enters 0, the block of statements controlled by the `if` clause executes, displaying a message that indicates the program cannot perform division by zero. Otherwise, the `else` clause takes control, which divides `number1` by `number2` and displays the result. Figure 3-9 shows the logic of the `if-else` statement.

Figure 3-9 Logic of the `if-else` statement

**Checkpoint**

MyProgrammingLab™ www.myprogramminglab.com

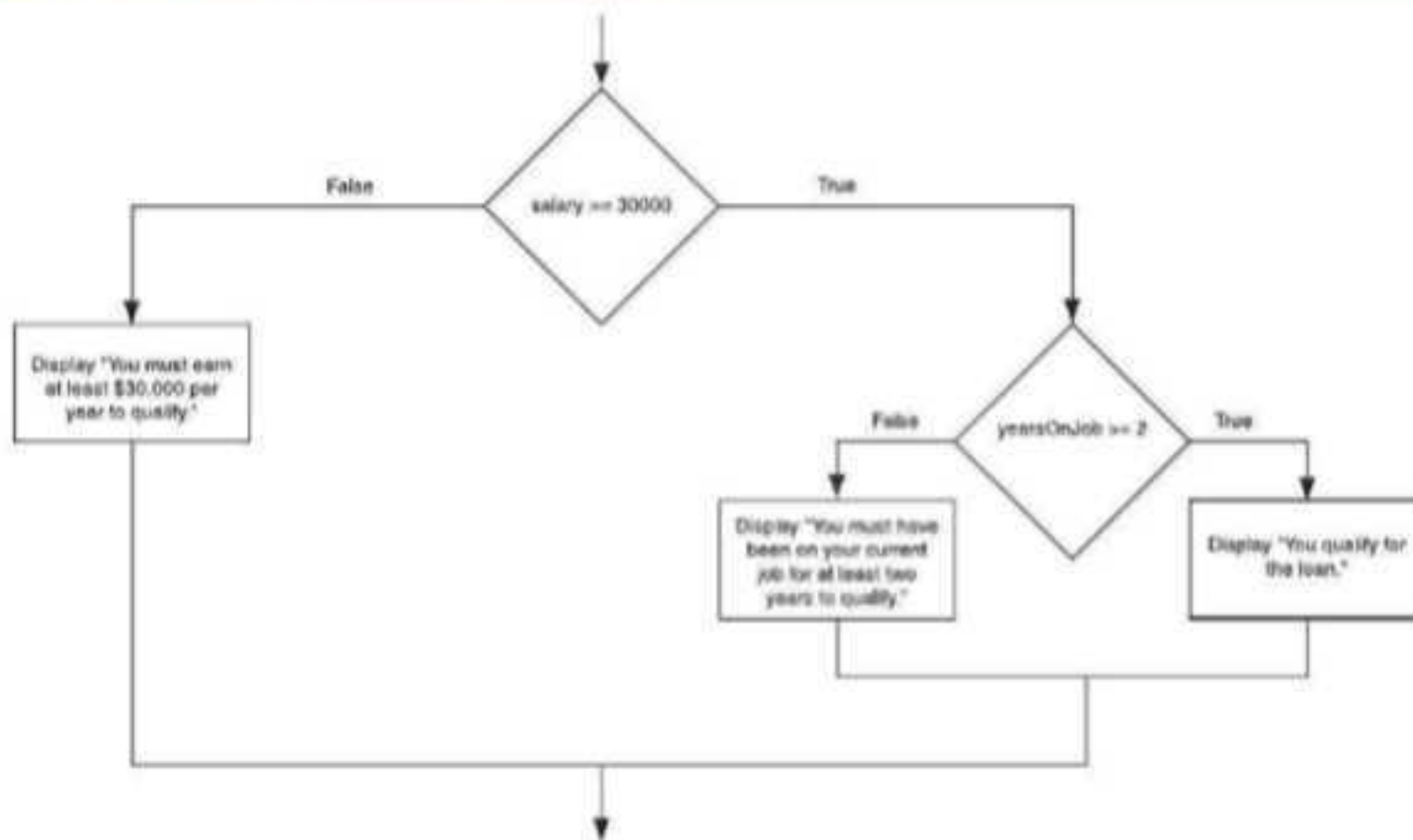
- 3.8 Write an `if-else` statement that assigns 20 to the variable `y` if the variable `x` is greater than 100. Otherwise, it should assign 0 to the variable `y`.
- 3.9 Write an `if-else` statement that assigns 1 to `x` when `y` is equal to 100. Otherwise, it should assign 0 to `x`.
- 3.10 Write an `if-else` statement that assigns 0.10 to `commission` unless `sales` is greater than or equal to 50000.0, in which case it assigns 0.2 to `commission`.
- 3.11 Write an `if-else` statement that assigns 0 to the variable `b` and assigns 1 to the variable `c` if the variable `a` is less than 10. Otherwise, it should assign -99 to the variable `b` and assign 0 to the variable `c`.

3.3 Nested if Statements

CONCEPT: To test more than one condition, an `if` statement can be nested inside another `if` statement.

Sometimes an `if` statement must be nested inside another `if` statement. For example, consider a banking program that determines whether a bank customer qualifies for a special, low interest rate on a loan. To qualify, two conditions must exist: (1) the customer's salary must be at least \$30,000, and (2) the customer must have held his or her current job for at least two years. Figure 3-10 shows a flowchart for an algorithm that could be used in such a program.

Figure 3-10 Logic of nested `if` statements



If we follow the flow of execution in the flowchart, we see that the expression `salary >= 30000` is tested. If this expression is false, there is no need to perform further tests; we know that the customer does not qualify for the special interest rate. If the expression is true, however, we need to test the second condition. This is done with a nested decision structure that tests the expression `yearsOnJob >= 2`. If this expression is true, then the customer qualifies for the special interest rate. If this expression is false, then the customer does not qualify. Code Listing 3-3 shows the complete program. Figures 3-11, 3-12, and 3-13 show what happens during three different sessions with the program.

Code Listing 3-3 (LoanQualifier.java)

```

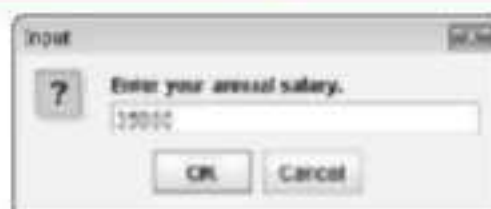
1 import javax.swing.JOptionPane; // Needed for JOptionPane class
2
3 /**

```

```
4  This program demonstrates a nested if statement.
5  */
6
7  public class LoanQualifier
8  {
9      public static void main(String[] args)
10     {
11         double salary;          // Annual salary
12         double yearsOnJob;      // Years at current job
13         String input;           // To hold string input
14
15         // Get the user's annual salary.
16         input = JOptionPane.showInputDialog("Enter your " +
17                                             "annual salary.");
18         salary = Double.parseDouble(input);
19
20         // Get the number of years at the current job.
21         input = JOptionPane.showInputDialog("Enter the number of " +
22                                             "years at your current job.");
23         yearsOnJob = Double.parseDouble(input);
24
25         // Determine whether the user qualifies for the loan.
26         if (salary >= 30000)
27         {
28             if (yearsOnJob >= 2)
29             {
30                 JOptionPane.showMessageDialog(null, "You qualify " +
31                                             "for the loan.");
32             }
33             else
34             {
35                 JOptionPane.showMessageDialog(null, "You must have " +
36                                             "been on your current job for at least " +
37                                             "two years to qualify.");
38             }
39         }
40         else
41         {
42             JOptionPane.showMessageDialog(null, "You must earn " +
43                                             "at least $30,000 per year to qualify.");
44         }
45
46         System.exit(0);
47     }
48 }
```


Figure 3-11 Interaction with the LoanQualifier program

This input dialog box appears first. The user enters 35000 and clicks on the OK button.

An input dialog box titled "Input" with a question mark icon. It contains the text "Enter your annual salary." and a text field with the value "35000". There are "OK" and "Cancel" buttons at the bottom.

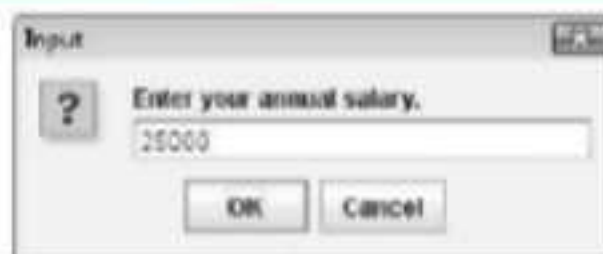
This input dialog box appears next. The user enters 1 and clicks on the OK button.

An input dialog box titled "Input" with a question mark icon. It contains the text "Enter the number of years at your current job." and a text field with the value "1". There are "OK" and "Cancel" buttons at the bottom.

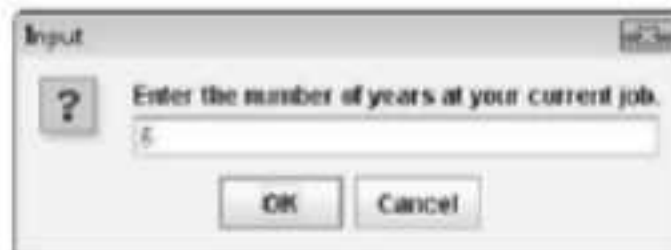
This message dialog box appears next.

A message dialog box titled "Message" with an information icon. It contains the text "You must have been on your current job for at least two years to qualify." and an "OK" button at the bottom.**Figure 3-12** Interaction with the LoanQualifier program

This input dialog box appears first. The user enters 25000 and clicks on the OK button.

An input dialog box titled "Input" with a question mark icon. It contains the text "Enter your annual salary." and a text field with the value "25000". There are "OK" and "Cancel" buttons at the bottom.

This input dialog box appears next. The user enters 5 and clicks on the OK button.

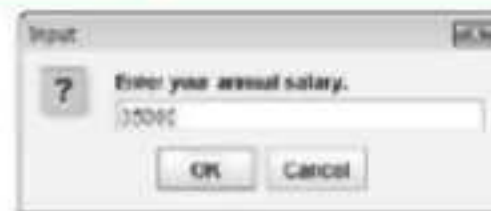
An input dialog box titled "Input" with a question mark icon. It contains the text "Enter the number of years at your current job." and a text field with the value "5". There are "OK" and "Cancel" buttons at the bottom.

This message dialog box appears next.

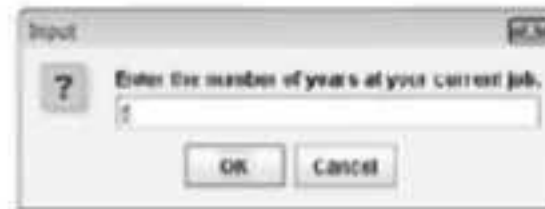
A message dialog box titled "Message" with an information icon. It contains the text "You must earn at least \$30,000 per year to qualify." and an "OK" button at the bottom.

Figure 3-13 Interaction with the LoanQualifier program

This input dialog box appears first. The user enters 35000 and clicks on the OK button.



This input dialog box appears next. The user enters 5 and clicks on the OK button.



This message dialog box appears next.

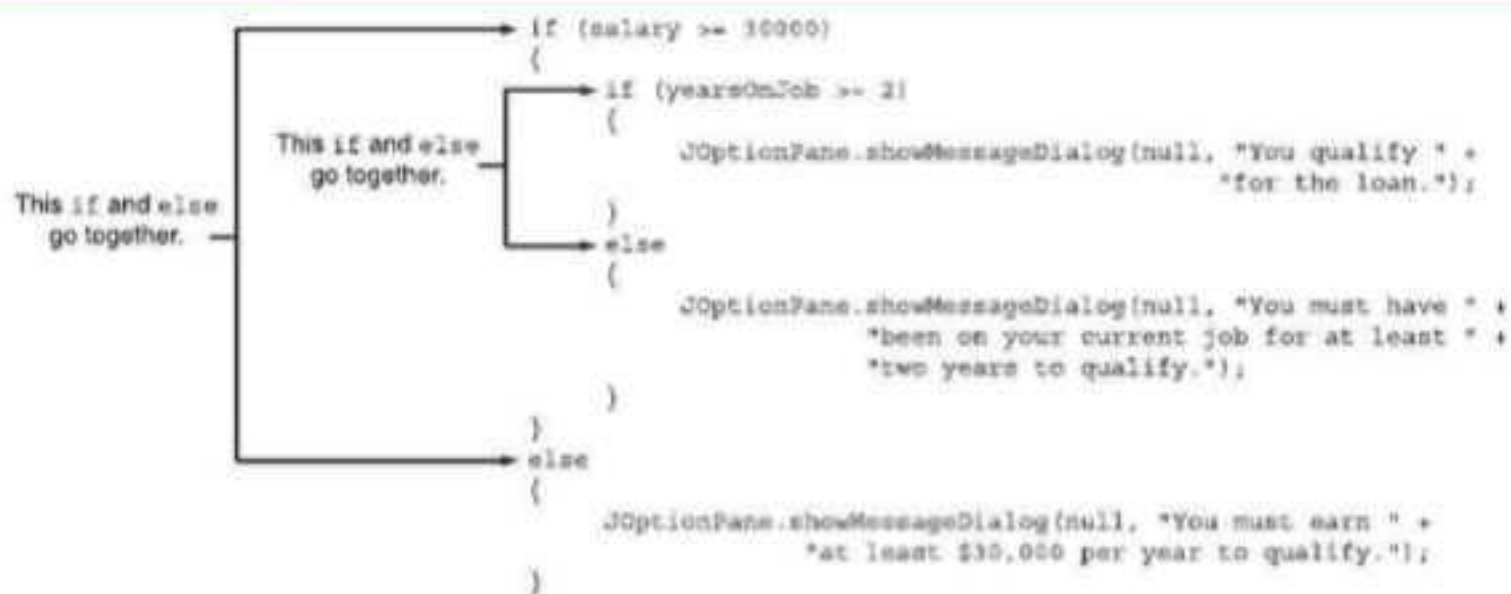


The first `if` statement (which begins in line 26) conditionally executes the second one (which begins in line 28). The only way the program will execute the second `if` statement is for the `salary` variable to contain a value that is greater than or equal to 30,000. When this is the case, the second `if` statement will test the `yearsOnJob` variable. If it contains a value that is greater than or equal to 2, a dialog box will be displayed informing the user that he or she qualifies for the loan.

It should be noted that the braces used in the `if` statements in this program are not required. They could have been written as follows:

```
if (salary >= 30000)
    if (yearsOnJob >= 2)
        JOptionPane.showMessageDialog(null, "You qualify " +
            "for the loan.");
    else
        JOptionPane.showMessageDialog(null, "You must have " +
            "been on your current job for at least " +
            "two years to qualify.");
else
    JOptionPane.showMessageDialog(null, "You must earn " +
        "at least $30,000 per year to qualify.");
```

Not only do the braces make the statements easier to read, but they also help in debugging code. When debugging a program with nested `if-else` statements, it's important to know which `if` clause each `else` clause belongs to. The rule for matching `else` clauses with `if` clauses is this: An `else` clause goes with the closest previous `if` clause that doesn't already have its own `else` clause. This is easy to see when the conditionally executed statements are enclosed in braces and are properly indented, as shown in Figure 3-14. Each `else` clause lines up with the `if` clause it belongs to. These visual cues are important because nested `if` statements can be very long and complex.

Figure 3-14 Alignment of `if` and `else` clauses

Testing a Series of Conditions

In the previous example, you saw how a program can use nested decision structures to test more than one condition. It is not uncommon for a program to have a series of conditions to test, and then perform an action depending on which condition is true. One way to accomplish this is to have a decision structure with numerous other decision structures nested inside it. For example, consider the program presented in the following *In the Spotlight* section.

In the Spotlight: Multiple Nested Decision Structures



Suppose one of your professors uses the following 10-point grading scale for exams:

Test Score	Grade
90 and above	A
80–89	B
70–79	C
60–69	D
Below 60	F

Your professor has asked you to write a program that will allow a student to enter a test score and then display the grade for that score. Here is the algorithm that you will use:

Ask the user to enter a test score.

Determine the grade in the following manner:

If the score is less than 60, then the grade is F.

Otherwise, if the score is less than 70, then the grade is D.

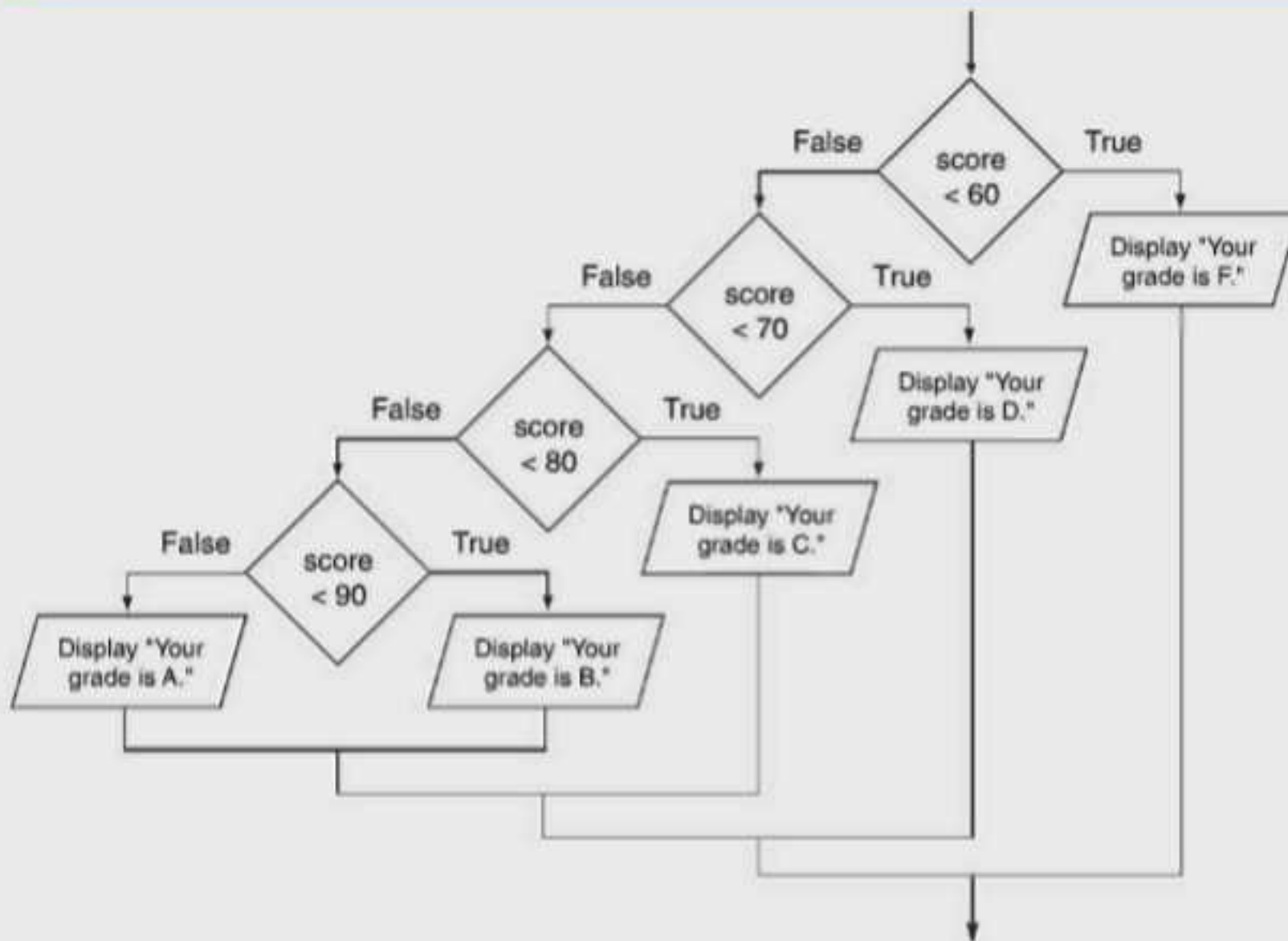
Otherwise, if the score is less than 80, then the grade is C.

Otherwise, if the score is less than 90, then the grade is B.

Otherwise, the grade is A.

You decide that the process of determining the grade will require several nested decision structures, as shown in Figure 3-15. Code Listing 3-4 shows the complete program. The code for the nested decision structures is in lines 23 through 51. Figures 3-16 and 3-17 show what happens in two different sessions with the program.

Figure 3-15 Nested decision structure to determine a grade



Code Listing 3-4 (NestedDecision.java)

```

1 import javax.swing.JOptionPane; // Needed for JOptionPane
2
3 /**
4  This program asks the user to enter a numeric test
5  score and displays a letter grade for the score. The
6  program uses nested decision structures
7  to determine the grade.
8  */
9
10 public class NestedDecision
11 {

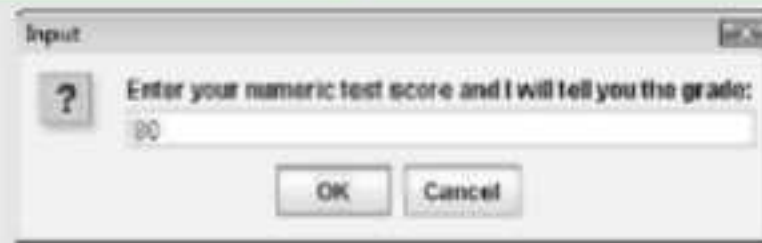
```



```
12 public static void main(String[] args)
13 {
14     int testScore;    // Numeric test score
15     String input;     // To hold the user's input
16
17     // Get the numeric test score.
18     input = JOptionPane.showInputDialog("Enter your numeric " +
19         "test score and I will tell you the grade: ");
20     testScore = Integer.parseInt(input);
21
22     // Display the grade.
23     if (testScore < 60)
24     {
25         JOptionPane.showMessageDialog(null, "Your grade is F.");
26     }
27     else
28     {
29         if (testScore < 70)
30         {
31             JOptionPane.showMessageDialog(null, "Your grade is D.");
32         }
33         else
34         {
35             if (testScore < 80)
36             {
37                 JOptionPane.showMessageDialog(null, "Your grade is C.");
38             }
39             else
40             {
41                 if (testScore < 90)
42                 {
43                     JOptionPane.showMessageDialog(null, "Your grade is B.");
44                 }
45                 else
46                 {
47                     JOptionPane.showMessageDialog(null, "Your grade is A.");
48                 }
49             }
50         }
51     }
52
53     System.exit(0);
54 }
55 }
```

Figure 3-16 Interaction with the NestedDecision program

This input dialog box appears first. The user enters 80 and then clicks the OK button.



Input dialog box titled "Input" with a question mark icon. The text says "Enter your numeric test score and I will tell you the grade:". The input field contains "80". There are "OK" and "Cancel" buttons at the bottom right.


This message dialog box appears next.



Message dialog box titled "Message" with an information icon. The text says "Your grade is B.". There is an "OK" button at the bottom right.

Figure 3-17 Interaction with the NestedDecision program

This input dialog box appears first. The user enters 72 and then clicks the OK button.



Input dialog box titled "Input" with a question mark icon. The text says "Enter your numeric test score and I will tell you the grade:". The input field contains "72". There are "OK" and "Cancel" buttons at the bottom right.

This message dialog box appears next.



Message dialog box titled "Message" with an information icon. The text says "Your grade is C.". There is an "OK" button at the bottom right.



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

- 3.12 Write nested if statements that perform the following test: If amount1 is greater than 10 and amount2 is less than 100, display the greater of the two.
- 3.13 Write code that tests the variable x to determine whether it is greater than 0. If x is greater than 0, the code should test the variable y to determine whether it is less than 20. If y is less than 20, the code should assign 1 to the variable z. If y is not less than 20, the code should assign 0 to the variable z.

3.4 The if-else-if Statement

CONCEPT: The if-else-if statement tests a series of conditions. It is often simpler to test a series of conditions with the if-else-if statement than with a set of nested if-else statements.

Even though Code Listing 3-4 is a simple example, the logic of the nested decision structure is fairly complex. You can alternatively test a series of conditions using the if-else-if



VideoNote

The if-else-if Statement

statement. The if-else-if statement makes certain types of nested decision logic simpler to write. Here is the general format of the if-else-if statement:

```

if (expression_1)
{
    statement
    statement
    etc.
}
else if (expression_2)
{
    statement
    statement
    etc.
}
Insert as many else if clauses as necessary
else
{
    statement
    statement
    etc.
}

```

If expression_1 is true these statements are executed, and the rest of the structure is ignored.

Otherwise, if expression_2 is true these statements are executed, and the rest of the structure is ignored.

These statements are executed if none of the expressions above are true.

When the statement executes, *expression_1* is tested. If *expression_1* is true, the block of statements that immediately follows is executed, and the rest of the structure is ignored. If *expression_1* is false, however, the program jumps to the next `else if` clause and tests *expression_2*. If it is true, the block of statements that immediately follows is executed, and then the rest of the structure is ignored. This process continues, from the top of the structure to the bottom, until one of the expressions is found to be true. If none of the expressions are true, the last `else` clause takes over and the block of statements immediately following it is executed.

The last `else` clause, which does not have an `if` statement following it, is referred to as the *trailing else*. The trailing `else` is optional, but in most cases you will use it.



NOTE: The general format shows braces surrounding each block of conditionally executed statements. As with other forms of the `if` statement, the braces are required only when more than one statement is conditionally executed.

Code Listing 3-5 shows an example of the if-else-if statement. This program is a modification of Code Listing 3-4, which appears in the previous *In the Spotlight* section. The output of this program is the same as Code Listing 3-4.

Code Listing 3-5 (TestResults.Java)

```
1 import javax.swing.JOptionPane;    // Needed for JOptionPane
2
3 /**
4  This program asks the user to enter a numeric test
5  score and displays a letter grade for the score. The
6  program uses an if-else-if statement to determine
7  the letter grade.
8  */
9
10 public class TestResults
11 {
12     public static void main(String[] args)
13     {
14         int testScore;    // Numeric test score
15         String input;     // To hold the user's input
16
17         // Get the numeric test score.
18         input = JOptionPane.showInputDialog("Enter your numeric " +
19             "test score and I will tell you the grade: ");
20         testScore = Integer.parseInt(input);
21
22         // Display the grade.
23         if (testScore < 60)
24             JOptionPane.showMessageDialog(null, "Your grade is F.");
25         else if (testScore < 70)
26             JOptionPane.showMessageDialog(null, "Your grade is D.");
27         else if (testScore < 80)
28             JOptionPane.showMessageDialog(null, "Your grade is C.");
29         else if (testScore < 90)
30             JOptionPane.showMessageDialog(null, "Your grade is B.");
31         else
32             JOptionPane.showMessageDialog(null, "Your grade is A.");
33
34         System.exit(0);
35     }
36 }
```

Let's analyze how the if-else-if statement in lines 23 through 32 works. First, the expression `testScore < 60` is tested in line 23:

```
→ if (testScore < 60)
    JOptionPane.showMessageDialog(null, "Your grade is F.");
else if (testScore < 70)
    JOptionPane.showMessageDialog(null, "Your grade is D.");
else if (testScore < 80)
```



```

        JOptionPane.showMessageDialog(null, "Your grade is C.");
    else if (testScore < 90)
        JOptionPane.showMessageDialog(null, "Your grade is B.");
    else
        JOptionPane.showMessageDialog(null, "Your grade is A.");

```

If `testScore` is less than 60, the message "Your grade is F." is displayed and the rest of the `if-else-if` statement is skipped. If `testScore` is not less than 60, the `else` clause in line 25 takes over and causes the next `if` statement to be executed:

```

    if (testScore < 60)
        JOptionPane.showMessageDialog(null, "Your grade is F.");
    → else if (testScore < 70)
        JOptionPane.showMessageDialog(null, "Your grade is D.");
    else if (testScore < 80)
        JOptionPane.showMessageDialog(null, "Your grade is C.");
    else if (testScore < 90)
        JOptionPane.showMessageDialog(null, "Your grade is B.");
    else
        JOptionPane.showMessageDialog(null, "Your grade is A.");

```

The first `if` statement handled all of the grades less than 60, so when this `if` statement executes, `testScore` will have a value of 60 or greater. If `testScore` is less than 70, the message "Your grade is D." is displayed and the rest of the `if-else-if` statement is skipped. This chain of events continues until one of the expressions is found to be true, or the last `else` clause at the end of the statement is encountered.

Notice the alignment and indentation that are used with the `if-else-if` statement: The starting `if` clause, the `else if` clauses, and the trailing `else` clause are all aligned, and the conditionally executed statements are indented.

Using the Trailing `else` to Catch Errors

The trailing `else` clause, which appears at the end of the `if-else-if` statement, is optional, but in many situations you will use it to catch errors. For example, Code Listing 3-5 will assign the grade 'A' to any test score that is 90 or greater. What if the highest possible test score is 100? We can modify the code as shown in Code Listing 3-6 so the trailing `else` clause catches any value greater than 100 and displays an error message. Figure 3-18 shows what happens when the user enters a test score that is greater than 100.

Code Listing 3-6 (TrailingElse.java)

```

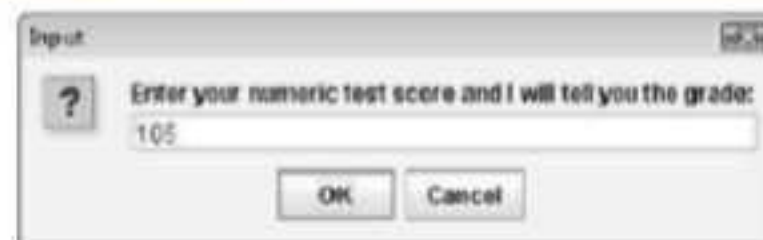
1 import javax.swing.JOptionPane; // Needed for JOptionPane
2
3 /**
4  This program asks the user to enter a numeric test
5  score and displays a letter grade for the score. The
6  program displays an error message if an invalid
7  numeric score is entered.
8 */
9

```

```
10 public class TrailingElse
11 {
12     public static void main(String[] args)
13     {
14         int testScore;    // Numeric test score
15         String input;     // To hold the user's input
16
17         // Get the numeric test score.
18         input = JOptionPane.showInputDialog("Enter your numeric " +
19             "test score and I will tell you the grade: ");
20         testScore = Integer.parseInt(input);
21
22         // Display the grade.
23         if (testScore < 60)
24             JOptionPane.showMessageDialog(null, "Your grade is F.");
25         else if (testScore < 70)
26             JOptionPane.showMessageDialog(null, "Your grade is D.");
27         else if (testScore < 80)
28             JOptionPane.showMessageDialog(null, "Your grade is C.");
29         else if (testScore < 90)
30             JOptionPane.showMessageDialog(null, "Your grade is B.");
31         else if (testScore <= 100)
32             JOptionPane.showMessageDialog(null, "Your grade is A.");
33         else
34             JOptionPane.showMessageDialog(null, "Invalid score.");
35
36         System.exit(0);
37     }
38 }
```

Figure 3-18 Interaction with the `NestedDecision` program

This input dialog box appears first. The user enters 105 and then clicks the OK button.



This message dialog box appears next.



The if-else-if Statement Compared to a Nested Decision Structure

You never have to use the if-else-if statement because its logic can be coded with nested if-else statements. However, a long series of nested if-else statements has two particular disadvantages when you are debugging code:

- The code can grow complex and become difficult to understand.

- Because indenting is important in nested statements, a long series of nested `if-else` statements can become too long to be displayed on the computer screen without horizontal scrolling. Also, long statements tend to “wrap around” when printed on paper, making the code even more difficult to read.

The logic of an `if-else-if` statement is usually easier to follow than that of a long series of nested `if-else` statements. And, because all of the clauses are aligned in an `if-else-if` statement, the lengths of the lines in the statement tend to be shorter.



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

3.14 What will the following program display?

```
public class CheckPoint
{
    public static void main(String[] args)
    {
        int funny = 7, serious = 15;
        funny = serious % 2;
        if (funny != 1)
        {
            funny = 0;
            serious = 0;
        }
        else if (funny == 2)
        {
            funny = 10;
            serious = 10;
        }
        else
        {
            funny = 1;
            serious = 1;
        }
        System.out.println(funny + " " + serious);
    }
}
```

3.15 The following program is used in a bookstore to determine how many discount coupons a customer gets. Complete the table that appears after the program.

```
import javax.swing.JOptionPane;
public class CheckPoint
{
    public static void main(String[] args)
    {
        int books, coupons;
        String input;
        input = JOptionPane.showInputDialog("How many books " +
                                           "are being purchased? ");
        books = Integer.parseInt(input);
```

```

    if (books < 1)
        coupons = 0;
    else if (books < 3)
        coupons = 1;
    else if (books < 5)
        coupons = 2;
    else
        coupons = 3;
    JOptionPane.showMessageDialog(null,
        "The number of coupons to give is " +
        coupons);
    System.exit(0);
}
}

```

If the customer purchases
this many books ...

1
2
3
4
5
10

this many coupons
are given.

3.5 Logical Operators

CONCEPT: Logical operators connect two or more relational expressions into one or reverse the logic of an expression.

Java provides two binary logical operators, `&&` and `||`, which are used to combine two boolean expressions into a single expression. It also provides the unary `!` operator, which reverses the truth of a boolean expression. Table 3-4 describes these logical operators.

Table 3-4 Logical operators

Operator	Meaning	Effect
<code>&&</code>	AND	Connects two boolean expressions into one. Both expressions must be true for the overall expression to be true.
<code> </code>	OR	Connects two boolean expressions into one. One or both expressions must be true for the overall expression to be true. It is only necessary for one to be true, and it does not matter which one.
<code>!</code>	NOT	The <code>!</code> operator reverses the truth of a boolean expression. If it is applied to an expression that is true, the operator returns false. If it is applied to an expression that is false, the operator returns true.

Table 3-5 shows examples of several boolean expressions that use logical operators.

Table 3-5 boolean expressions using logical operators

Expression	Meaning
<code>x > y && a < b</code>	Is <code>x</code> greater than <code>y</code> AND is <code>a</code> less than <code>b</code> ?
<code>x == y x == z</code>	Is <code>x</code> equal to <code>y</code> OR is <code>x</code> equal to <code>z</code> ?
<code>!(x > y)</code>	Is the expression <code>x > y</code> NOT true?

Let's take a close look at each of these operators.

The `&&` Operator

The `&&` operator is known as the logical AND operator. It takes two boolean expressions as operands and creates a boolean expression that is `true` only when both subexpressions are `true`. Here is an example of an `if` statement that uses the `&&` operator:

```
if (temperature < 20 && minutes > 12)
{
    System.out.println("The temperature is in the " +
        "danger zone.");
}
```

In this statement the two boolean expressions `temperature < 20` and `minutes > 12` are combined into a single expression. The message will be displayed only if `temperature` is less than 20 AND `minutes` is greater than 12. If either boolean expression is `false`, the entire expression is `false` and the message is not displayed.

Table 3-6 shows a truth table for the `&&` operator. The truth table lists expressions showing all the possible combinations of `true` and `false` connected with the `&&` operator. The resulting values of the expressions are also shown.

Table 3-6 Truth table for the `&&` operator

Expression	Value of the Expression
<code>true && false</code>	<code>false</code>
<code>false && true</code>	<code>false</code>
<code>false && false</code>	<code>false</code>
<code>true && true</code>	<code>true</code>

As the table shows, both sides of the `&&` operator must be `true` for the operator to return a `true` value.

The `&&` operator performs *short-circuit evaluation*. Here's how it works: If the expression on the left side of the `&&` operator is `false`, the expression on the right side will not be checked. Because the entire expression is `false` if only one of the subexpressions is `false`, it would waste CPU time to check the remaining expression. So, when the `&&` operator

finds that the expression on its left is `false`, it short-circuits and does not evaluate the expression on its right.

The `&&` operator can be used to simplify programs that otherwise would use nested `if` statements. The program in Code Listing 3-7 is a different version of the `LoanQualifier` program in Code Listing 3-3, written to use the `&&` operator. Figures 3-19 and 3-20 show the interaction during two different sessions with the program.

Code Listing 3-7 (LogicalAnd.java)

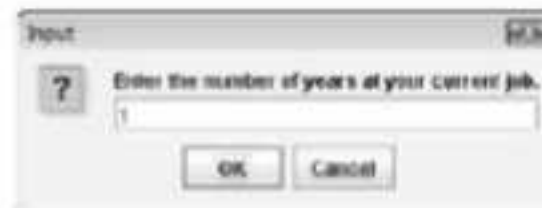
```
1 import javax.swing.JOptionPane; // Needed for JOptionPane class
2
3 /**
4  * This program demonstrates the logical && operator.
5  */
6
7 public class LogicalAnd
8 {
9     public static void main(String[] args)
10    {
11        double salary;        // Annual salary
12        double yearsOnJob;    // Years at current job
13        String input;         // To hold string input
14
15        // Get the user's annual salary.
16        input = JOptionPane.showInputDialog("Enter your " +
17                                           "annual salary.");
18        salary = Double.parseDouble(input);
19
20        // Get the number of years at the current job.
21        input = JOptionPane.showInputDialog("Enter the number of " +
22                                           "years at your current job.");
23        yearsOnJob = Double.parseDouble(input);
24
25        // Determine whether the user qualifies for the loan.
26        if (salary >= 30000 && yearsOnJob >= 2)
27        {
28            JOptionPane.showMessageDialog(null, "You qualify " +
29                                           "for the loan.");
30        }
31        else
32        {
33            JOptionPane.showMessageDialog(null, "You do not " +
34                                           "qualify for the loan.");
35        }
36
37        System.exit(0);
38    }
39 }
```


Figure 3-19 Interaction with the LogicalAnd program

This input dialog box appears first. The user enters 50000 and clicks on the OK button.


 An input dialog box titled "Input" with a question mark icon. The text inside says "Enter your annual salary." Below the text is a text field containing the number "50000". At the bottom are two buttons: "OK" and "Cancel".


This input dialog box appears next. The user enters 1 and clicks on the OK button.


 An input dialog box titled "Input" with a question mark icon. The text inside says "Enter the number of years at your current job." Below the text is a text field containing the number "1". At the bottom are two buttons: "OK" and "Cancel".

This message dialog box appears next.


 A message dialog box titled "Message" with an information icon. The text inside says "You do not qualify for the loan." At the bottom is an "OK" button.
Figure 3-20 Interaction with the LogicalAnd program

This input dialog box appears first. The user enters 50000 and clicks on the OK button.


 An input dialog box titled "Input" with a question mark icon. The text inside says "Enter your annual salary." Below the text is a text field containing the number "50000". At the bottom are two buttons: "OK" and "Cancel".

This input dialog box appears next. The user enters 4 and clicks on the OK button.


 An input dialog box titled "Input" with a question mark icon. The text inside says "Enter the number of years at your current job." Below the text is a text field containing the number "4". At the bottom are two buttons: "OK" and "Cancel".

This message dialog box appears next.


 A message dialog box titled "Message" with an information icon. The text inside says "You qualify for the loan." At the bottom is an "OK" button.

The message "You qualify for the loan." is displayed only when both the expressions `salary >= 30000` and `yearsOnJob >= 2` are true. If either of these expressions is false, the message "You do not qualify for the loan." is displayed.

You can also use logical operators with boolean variables. For example, assuming that `isValid` is a boolean variable, the following if statement determines whether `isValid` is true and `x` is greater than 90.

```
if (isValid && x > 90)
```

The || Operator

The || operator is known as the logical OR operator. It takes two boolean expressions as operands and creates a boolean expression that is true when either of the subexpressions is true. Here is an example of an if statement that uses the || operator:

```
if (temperature < 20 || temperature > 100)
{
    System.out.println("The temperature is in the " +
        "danger zone.");
}
```

The message will be displayed if temperature is less than 20 OR temperature is greater than 100. If either relational test is true, the entire expression is true.

Table 3-7 shows a truth table for the || operator.

All it takes for an OR expression to be true is for one side of the || operator to be true. It doesn't matter if the other side is false or true. Like the && operator, the || operator performs short-circuit evaluation. If the expression on the left side of the || operator is true, the expression on the right side will not be checked. Because it is necessary for only one of the expressions to be true, it would waste CPU time to check the remaining expression.

Table 3-7 Truth table for the || operator

Expression	Value
true false	true
false true	true
false false	false
true true	true

The program in Code Listing 3-8 is a different version of the previous program, shown in Code Listing 3-7. This version uses the || operator to determine whether salary >= 30000 is true OR yearsOnJob >= 2 is true. If either expression is true, then the person qualifies for the loan. Figure 3-21 shows example interaction with the program.

Code Listing 3-8 (LogicalOr.java)

```
1 import javax.swing.JOptionPane; // Needed for JOptionPane class
2
3 /**
4  * This program demonstrates the logical || operator.
5  */
6
7 public class LogicalOr
8 {
9     public static void main(String[] args)
10    {
```



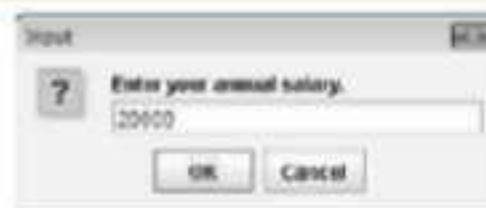
```

11     double salary;           // Annual salary
12     double yearsOnJob;      // Years at current job
13     String input;           // To hold string input
14
15     // Get the user's annual salary.
16     input = JOptionPane.showInputDialog("Enter your " +
17                                         "annual salary.");
18     salary = Double.parseDouble(input);
19
20     // Get the number of years at the current job.
21     input = JOptionPane.showInputDialog("Enter the number of " +
22                                         "years at your current job.");
23     yearsOnJob = Double.parseDouble(input);
24
25     // Determine whether the user qualifies for loan.
26     if (salary >= 30000 || yearsOnJob >= 2)
27     {
28         JOptionPane.showMessageDialog(null, "You qualify " +
29                                         "for the loan.");
30     }
31     else
32     {
33         JOptionPane.showMessageDialog(null, "You do not " +
34                                         "qualify for the loan.");
35     }
36
37     System.exit(0);
38 }
39 }

```

Figure 3-21 Interaction with the LogicalOr program

This input dialog box appears first. The user enters 20000 and clicks on the OK button.



This input dialog box appears next. The user enters 7 and clicks on the OK button.



This message dialog box appears next.



The ! Operator

The `!` operator performs a logical NOT operation. It is a unary operator that takes a boolean expression as its operand and reverses its logical value. In other words, if the expression is `true`, the `!` operator returns `false`, and if the expression is `false`, it returns `true`. Here is an `if` statement using the `!` operator:

```
if (!(temperature > 100))
    System.out.println("This is below the maximum temperature.");
```

First, the expression `(temperature > 100)` is tested and a value of either `true` or `false` is the result. Then the `!` operator is applied to that value. If the expression `(temperature > 100)` is `true`, the `!` operator returns `false`. If the expression `(temperature > 100)` is `false`, the `!` operator returns `true`. The previous code is equivalent to asking: “Is the temperature not greater than 100?”

Table 3-8 shows a truth table for the `!` operator.

Table 3-8 Truth table for the `!` operator

Expression	Value
<code>!true</code>	<code>false</code>
<code>!false</code>	<code>true</code>

The Precedence of Logical Operators

Like other operators, the logical operators have orders of precedence and associativity. Table 3-9 shows the precedence of the logical operators, from highest to lowest.

Table 3-9 Logical operators in order of precedence

<code>!</code>
<code>&&</code>
<code> </code>

The `!` operator has a higher precedence than many of Java’s other operators. You should always enclose its operand in parentheses unless you intend to apply it to a variable or a simple expression with no other operators. For example, consider the following expressions (assume `x` is an `int` variable with a value stored in it):

```
!(x > 2)
!x > 2
```

The first expression applies the `!` operator to the expression `x > 2`. It is asking “is `x` not greater than 2?” The second expression, however, attempts to apply the `!` operator to `x` only. It is asking “is the logical complement of `x` greater than 2?” Because the `!` operator can only be applied to boolean expressions, this statement would cause a compiler error.

The `&&` and `||` operators rank lower in precedence than the relational operators, so precedence problems are less likely to occur. If you are unsure, however, it doesn't hurt to use parentheses anyway.

`(a > b) && (x < y)` is the same as `a > b && x < y`
`(x == y) || (b > a)` is the same as `x == y || b > a`

The logical operators evaluate their expressions from left to right. In the following expression, `a < b` is evaluated before `y == z`.

`a < b || y == z`

In the following expression, `y == z` is evaluated first, however, because the `&&` operator has higher precedence than `||`.

`a < b || y == z && m > j`

This expression is equivalent to the following:

`(a < b) || ((y == z) && (m > j))`

Table 3-10 shows the precedence of all the operators we have discussed so far. This table includes the assignment, arithmetic, relational, and logical operators.

Table 3-10 Precedence of all operators discussed so far

Order of Precedence	Operators	Description
1	- (unary negation) !	Unary negation, logical NOT
2	* / %	Multiplication, division, modulus
3	+ -	Addition, subtraction
4	< > <= >=	Less than, greater than, less than or equal to, greater than or equal to
5	== !=	Equal to, not equal to
6	&&	Logical AND
7		Logical OR
8	= += -= *= /= %=	Assignment and combined assignment

Checking Numeric Ranges with Logical Operators

Sometimes you will need to write code that determines whether a numeric value is within a specific range of values or outside a specific range of values. When determining whether a number is inside a range, it's best to use the `&&` operator. For example, the following `if` statement checks the value in `x` to determine whether it is in the range of 20 through 40:

```
if (x >= 20 && x <= 40)
    System.out.println(x + " is in the acceptable range.");
```

The boolean expression in the `if` statement will be `true` only when `x` is greater than or equal to 20 AND less than or equal to 40. The value in `x` must be within the range of 20 through 40 for this expression to be `true`.

When determining whether a number is outside a range, it's best to use the `||` operator. The following statement determines whether `x` is outside the range of 20 through 40:

```
if (x < 20 || x > 40)
    System.out.println(x + " is outside the acceptable range.");
```

It's important not to get the logic of these logical operators confused. For example, the boolean expression in the following `if` statement would never test true:

```
if (x < 20 && x > 40)
    System.out.println(x + " is outside the acceptable range.");
```

Obviously, `x` cannot be less than 20 and at the same time be greater than 40.



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

- 3.16 The following truth table shows various combinations of the values `true` and `false` connected by a logical operator. Complete the table by circling `T` or `F` to indicate whether the result of such a combination is true or false.

Logical Expression	Result (true or false)	
<code>true && false</code>	<code>T</code>	<code>F</code>
<code>true && true</code>	<code>T</code>	<code>F</code>
<code>false && true</code>	<code>T</code>	<code>F</code>
<code>false && false</code>	<code>T</code>	<code>F</code>
<code>true false</code>	<code>T</code>	<code>F</code>
<code>true true</code>	<code>T</code>	<code>F</code>
<code>false true</code>	<code>T</code>	<code>F</code>
<code>false false</code>	<code>T</code>	<code>F</code>
<code>!true</code>	<code>T</code>	<code>F</code>
<code>!false</code>	<code>T</code>	<code>F</code>

- 3.17 Assume the variables `a = 2`, `b = 4`, and `c = 6`. Circle the `T` or `F` for each of the following conditions to indicate whether it is true or false.

<code>a == 4 b > 2</code>	<code>T</code>	<code>F</code>
<code>6 <= c && a > 3</code>	<code>T</code>	<code>F</code>
<code>1 != b && c != 3</code>	<code>T</code>	<code>F</code>
<code>a >= -1 a <= b</code>	<code>T</code>	<code>F</code>
<code>!(a > 2)</code>	<code>T</code>	<code>F</code>

- 3.18 Write an `if` statement that displays the message "The number is valid" if the variable `speed` is within the range 0 through 200.
- 3.19 Write an `if` statement that displays the message "The number is not valid" if the variable `speed` is outside the range 0 through 200.

3.6

Comparing String Objects

CONCEPT: You cannot use relational operators to compare `String` objects. Instead you must use a `String` method.

You saw in the preceding sections how numeric values can be compared using the relational operators. You should not use the relational operators to compare `String` objects, however.

Remember that a `String` object is referenced by a variable that contains the object's memory address. When you use a relational operator with the reference variable, the operator works on the memory address that the variable contains, not the contents of the `String` object. For example, suppose a program has the following declarations:

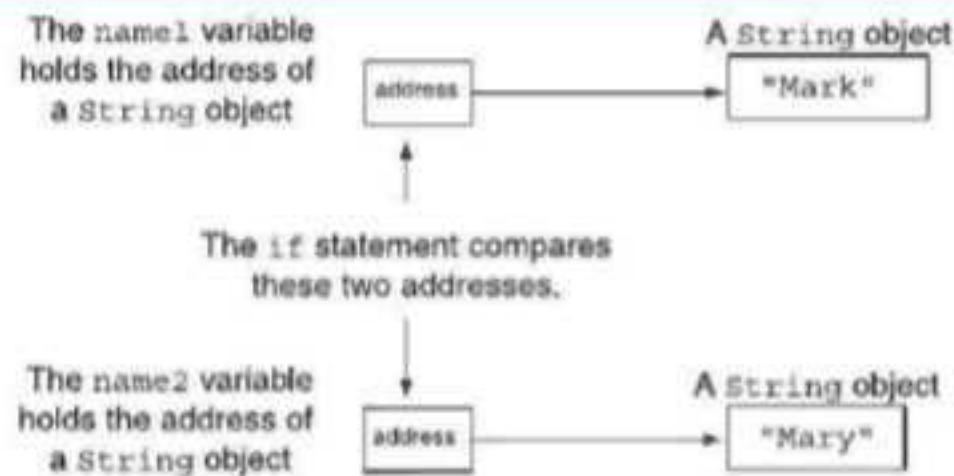
```
String name1 = "Mark";
String name2 = "Mary";
```

And later, the same program has the following `if` statement:

```
if (name1 == name2)
```

The expression `name1 == name2` will be `false`, but not because the strings "Mark" and "Mary" are different. The expression will be `false` because the variables `name1` and `name2` reference different objects. Figure 3-22 illustrates how the variables reference the `String` objects.

Figure 3-22 The `name1` and `name2` variables reference different `String` objects



To compare the contents of two `String` objects correctly, you should use the `String` class's `equals` method. The general form of the method is as follows:

```
StringReference1.equals(StringReference2)
```

StringReference1 is a variable that references a `String` object, and *StringReference2* is another variable that references a `String` object. The method returns `true` if the two strings are equal, or `false` if they are not equal. Here is an example:

```
if (name1.equals(name2))
```

Assuming that `name1` and `name2` reference `String` objects, the expression in the `if` statement will return `true` if they are the same, or `false` if they are not the same. The program in Code Listing 3-9 demonstrates.

Code Listing 3-9 (StringCompare.java)

```
1  /**
2   * This program correctly compares two String objects using
3   * the equals method.
```

```
4  */
5
6  public class StringCompare
7  {
8      public static void main(String[] args)
9      {
10         String name1 = "Mark",
11             name2 = "Mark",
12             name3 = "Mary";
13
14         // Compare "Mark" and "Mark"
15
16         if (name1.equals(name2))
17         {
18             System.out.println(name1 + " and " + name2 +
19                               " are the same.");
20         }
21         else
22         {
23             System.out.println(name1 + " and " + name2 +
24                               " are NOT the same.");
25         }
26
27         // Compare "Mark" and "Mary"
28
29         if (name1.equals(name3))
30         {
31             System.out.println(name1 + " and " + name3 +
32                               " are the same.");
33         }
34         else
35         {
36             System.out.println(name1 + " and " + name3 +
37                               " are NOT the same.");
38         }
39     }
40 }
```

Program Output

```
Mark and Mark are the same.
Mark and Mary are NOT the same.
```

You can also compare `String` objects to string literals. Simply pass the string literal as the argument to the `equals` method, as follows:

```
if (name1.equals("Mark"))
```


To determine whether two strings are not equal, simply apply the `!` operator to the `equals` method's return value. Here is an example:

```
if (!name1.equals("Mark"))
```

The boolean expression in this `if` statement performs a not-equal-to operation. It determines whether the object referenced by `name1` is not equal to "Mark".

The `String` class also provides the `compareTo` method, which can be used to determine whether one string is greater than, equal to, or less than another string. The general form of the method is as follows:

```
StringReference.compareTo(OtherString)
```

StringReference is a variable that references a `String` object, and *OtherString* is either another variable that references a `String` object or a string literal. The method returns an integer value that can be used in the following manner:

- If the method's return value is negative, the string referenced by *StringReference* (the calling object) is less than the *OtherString* argument.
- If the method's return value is 0, the two strings are equal.
- If the method's return value is positive, the string referenced by *StringReference* (the calling object) is greater than the *OtherString* argument.

For example, assume that `name1` and `name2` are variables that reference `String` objects. The following `if` statement uses the `compareTo` method to compare the strings:

```
if (name1.compareTo(name2) == 0)
    System.out.println("The names are the same.");
```

Also, the following expression compares the string referenced by `name1` to the string literal "Joe":

```
if (name1.compareTo("Joe") == 0)
    System.out.println("The names are the same.");
```

The program in Code Listing 3-10 more fully demonstrates the `compareTo` method.

Code Listing 3-10 (StringCompareTo.java)

```
1  /**
2   * This program compares two String objects using
3   * the compareTo method.
4   */
5
6  public class StringCompareTo
7  {
8      public static void main(String[] args)
9      {
10         String name1 = "Mary",
11             name2 = "Mark";
12
13         // Compare "Mary" and "Mark"
```

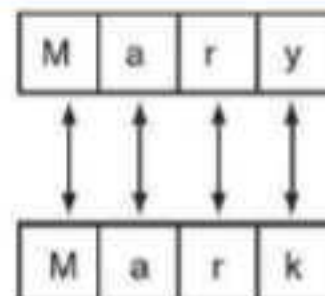
```
14
15     if (name1.compareTo(name2) < 0)
16     {
17         System.out.println(name1 + " is less than " + name2);
18     }
19     else if (name1.compareTo(name2) == 0)
20     {
21         System.out.println(name1 + " is equal to " + name2);
22     }
23     else if (name1.compareTo(name2) > 0)
24     {
25         System.out.println(name1 + " is greater than " + name2);
26     }
27 }
28 }
```

Program Output

Mary is greater than Mark

Let's take a closer look at this program. When you use the `compareTo` method to compare two strings, the strings are compared character by character. This is often called a *lexicographical comparison*. The program uses the `compareTo` method to compare the strings "Mary" and "Mark", beginning with the first, or leftmost, characters. This is illustrated in Figure 3-23.

Figure 3-23 String comparison of "Mary" and "Mark"



Here is how the comparison takes place:

1. The "M" in "Mary" is compared with the "M" in "Mark." Because these are the same, the next characters are compared.
2. The "a" in "Mary" is compared with the "a" in "Mark." Because these are the same, the next characters are compared.
3. The "r" in "Mary" is compared with the "r" in "Mark." Because these are the same, the next characters are compared.
4. The "y" in "Mary" is compared with the "k" in "Mark." Because these are not the same, the two strings are not equal. The character "y" is greater than "k", so it is determined that "Mary" is greater than "Mark."

If one of the strings in a comparison is shorter than the other, Java can only compare the corresponding characters. If the corresponding characters are identical, then the shorter

string is considered less than the longer string. For example, suppose the strings “High” and “Hi” were being compared. The string “Hi” would be considered less than “High” because it is shorter.

Ignoring Case in String Comparisons

The `equals` and `compareTo` methods perform case-sensitive comparisons, which means that uppercase letters are not considered the same as their lowercase counterparts. In other words, “A” is not the same as “a”. This can obviously lead to problems when you want to perform case-insensitive comparisons.

The `String` class provides the `equalsIgnoreCase` and `compareToIgnoreCase` methods. These methods work like the `equals` and `compareTo` methods, except the case of the characters in the strings is ignored. For example, the program in Code Listing 3-11 asks the user to enter the “secret word,” which is similar to a password. The secret word is “PROSPERO”, and the program performs a case-insensitive string comparison to determine whether the user has entered it.

Code Listing 3-11 (SecretWord.java)

```

1  import java.util.Scanner; // Needed for the Scanner class
2
3  /**
4   * This program demonstrates a case insensitive string comparison.
5   */
6
7  public class SecretWord
8  {
9      public static void main(String[] args)
10     {
11         String input;    // To hold the user's input
12
13         // Create a Scanner object for keyboard input.
14         Scanner keyboard = new Scanner(System.in);
15
16         // Prompt the user to enter the secret word.
17         System.out.print("Enter the secret word: ");
18         input = keyboard.nextLine();
19
20         // Determine whether the user entered the secret word.
21         if (input.equalsIgnoreCase("PROSPERO"))
22         {
23             System.out.println("Congratulations! You know the " +
24                               "secret word!");
25         }
26         else
27         {

```

```

28         System.out.println("Sorry, that is NOT the " +
29                             "secret word!");
30     }
31 }
32 }

```

Program Output with Example Input Shown in Bold

Enter the secret word: **Ferdinand** [Enter]
 Sorry, that is NOT the secret word!

Program Output with Example Input Shown in Bold

Enter the secret word: **Prospero** [Enter]
 Congratulations! You know the secret word!

The `compareToIgnoreCase` method works exactly like the `compareTo` method, except the case of the characters in the strings being compared is ignored.



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

- 3.20 Assume the variable `name` references a `String` object. Write an `if` statement that displays "Do I know you?" if the `String` object contains "Timothy".
- 3.21 Assume the variables `name1` and `name2` reference two different `String` objects, containing different strings. Write code that displays the strings referenced by these variables in alphabetical order.
- 3.22 Modify the statement you wrote in response to Checkpoint 3.20 so it performs a case-insensitive comparison.

3.7

More about Variable Declaration and Scope

CONCEPT: The scope of a variable is limited to the block in which it is declared.

Recall from Chapter 2 that a local variable is a variable that is declared inside a method. Java allows you to create local variables just about anywhere in a method. For example, look at the program in Code Listing 3-12. The `main` method declares two `String` reference variables: `firstName` and `lastName`. Notice that the declarations of these variables appear near the code that first uses the variables.

Code Listing 3-12 (VariableScope.java)

```

1 import javax.swing.JOptionPane; // Needed for JOptionPane
2
3 /**
4  * This program demonstrates how variables may be declared
5  * in various locations throughout a program.

```



```

6  */
7
8  public class VariableScope
9  {
10     public static void main(String[] args)
11     {
12         // Get the user's first name.
13         String firstName;
14         firstName = JOptionPane.showInputDialog("Enter your " +
15                                             "first name.");
16
17         // Get the user's last name.
18         String lastName;
19         lastName = JOptionPane.showInputDialog("Enter your " +
20                                             "last name.");
21
22         JOptionPane.showMessageDialog(null, "Hello, " + firstName +
23                                     " " + lastName);
24         System.exit(0);
25     }
26 }

```

Although it is a common practice to declare all of a method's local variables at the beginning of the method, it is possible to declare them at later points. Sometimes programmers declare certain variables near the part of the program where they are used in order to make their purpose more evident.

Recall from Chapter 2 that a variable's scope is the part of the program where the variable's name may be used. A local variable's scope always starts at the variable's declaration, and ends at the closing brace of the block of code in which it is declared. In Code Listing 3-12, the `firstName` variable is visible only to the code in lines 13 through 24. The `lastName` variable is visible only to the code in lines 18 through 24.



NOTE: When a program is running and it enters the section of code that constitutes a variable's scope, it is said that the variable "comes into scope." This simply means the variable is now visible and the program may reference it. Likewise, when a variable "leaves scope" it may not be used.

3.8

The Conditional Operator (Optional)

CONCEPT: You can use the conditional operator to create short expressions that work like `if-else` statements.

The *conditional operator* is powerful and unique. Because it takes three operands, it is considered a ternary operator. The conditional operator provides a shorthand method of

expressing a simple if-else statement. The operator consists of the question mark (?) and the colon (:). You use the operator to write a conditional expression, in the following format:

```
BooleanExpression ? Value1 : Value2;
```

The *BooleanExpression* is like the boolean expression in the parentheses of an if statement. If the *BooleanExpression* is true, then the value of the conditional expression is *Value1*. Otherwise, the value of the conditional expression is *Value2*. Here is an example of a statement using the conditional operator:

```
y = x < 0 ? 10 : 20;
```

This preceding statement performs the same operation as the following if-else statement:

```
if (x < 0)
    y = 10;
else
    y = 20;
```

The conditional operator gives you the ability to pack decision-making power into a concise line of code. With a little imagination it can be applied to many other programming problems. For instance, consider the following statement:

```
System.out.println("Your grade is: " +
    (score < 60 ? "Fail." : "Pass."));
```

Converted to an if-else statement, it would be written as follows:

```
if (score < 60)
    System.out.println("Your grade is: Fail.");
else
    System.out.println("Your grade is: Pass.");
```



NOTE: The parentheses are placed around the conditional expression because the + operator has higher precedence than the ?: operator. Without the parentheses, the + operator would concatenate the value in score with the string "Your grade is: ".

For a complete example using the conditional operator, see the program named *ConsultantCharges.java* in this chapter's source code folder, available for download from the book's companion Web site (www.pearsonhighered.com/gaddis).



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

3.23 Rewrite the following if-else statements as statements that use the conditional operator.

```
a) if (x > y)
    z = 1;
    else
    z = 20;
```

```

b) if (temp > 45)
    population = base * 10;
else
    population = base * 2;
c) if (hours > 40)
    wages *= 1.5;
else
    wages *= 1;
d) if (result >= 0)
    System.out.println("The result is positive.");
else
    System.out.println("The result is negative.");

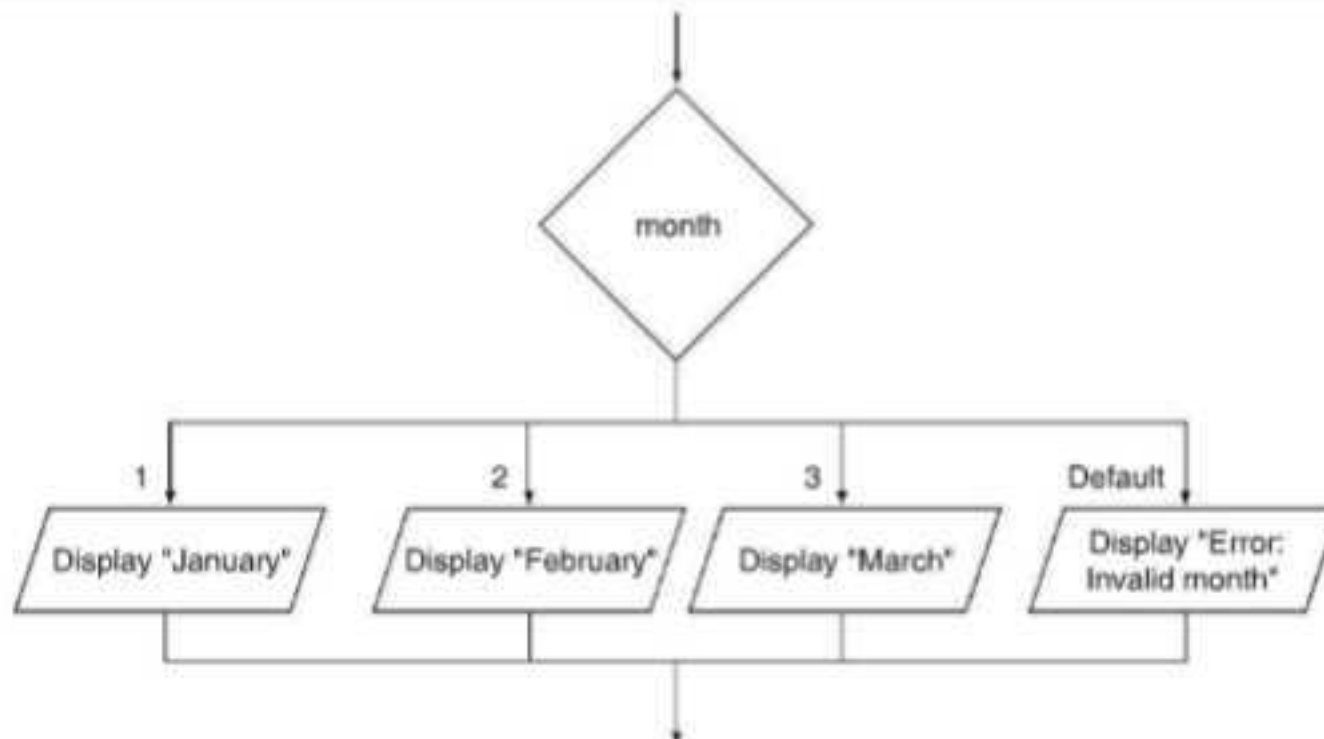
```

3.9 The switch Statement

CONCEPT: The `switch` statement lets the value of a variable or expression determine where the program will branch to.

The *switch statement* is a *multiple alternative decision structure*. It allows you to test the value of a variable or an expression and then use that value to determine which statement or set of statements to execute. Figure 3-24 shows an example of how a multiple alternative decision structure looks in a flowchart.

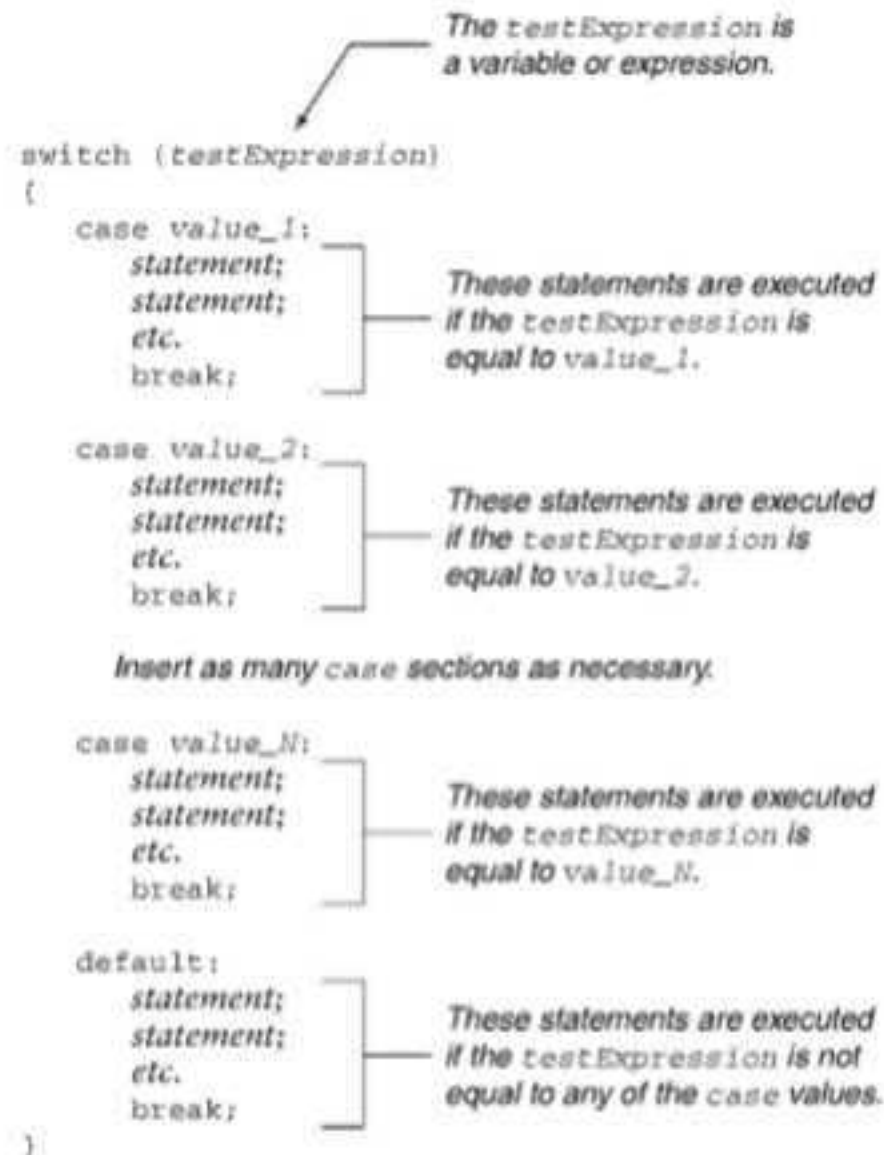
Figure 3-24 A multiple alternative decision structure



In the flowchart, the diamond symbol shows `month`, which is the name of a variable. If the `month` variable contains the value 1, the program displays *January*. If the `month` variable contains the value 2, the program displays *February*. If the `month` variable contains the

value 3, the program displays *March*. If the `month` variable contains none of these values, the action that is labeled *Default* is executed. In this case, the program displays *Error: Invalid month*.

Here is the general format of a switch statement in Java:



The first line of the statement starts with the word `switch`, followed by a `testExpression`, which is enclosed in parentheses. The `testExpression` is a variable or an expression that gives a `char`, `byte`, `short`, `int`, or `string` value. (If you are using a version of Java prior to Java 7, the `testExpression` cannot be a `string`.)

Beginning at the next line is a block of code enclosed in curly braces. Inside this block of code is one or more `case` sections. A `case` section begins with the word `case`, followed by a value, followed by a colon. Each `case` section contains one or more statements, followed by a `break` statement. After all of the `case` sections, an optional `default` section appears.

When the `switch` statement executes, it compares the value of the `testExpression` with the values that follow each of the `case` statements (from top to bottom). When it finds a `case` value that matches the `testExpression`'s value, the program branches to the `case` statement. The statements that follow the `case` statement are executed until a `break` statement is encountered. At that point, the program jumps out of the `switch` statement. If the `testExpression` does not match any of the `case` values, the program branches to the `default` statement and executes the statements that immediately follow it.



NOTE: Each of the case values must be unique.

For example, the following code performs the same operation as the flowchart in Figure 3-24. Assume `month` is an `int` variable.

```
switch (month)
{
    case 1:
        System.out.println("January");
        break;

    case 2:
        System.out.println("February");
        break;

    case 3:
        System.out.println("March");
        break;

    default:
        System.out.println("Error: Invalid month");
        break;
}
```

In this example the *testExpression* is the `month` variable. The `month` variable will be evaluated and one of the following actions will take place:

- If the value in the `month` variable is 1, the program will branch to the `case 1:` section and execute the `System.out.println("January")` statement that immediately follows it. The `break` statement then causes the program to exit the `switch` statement.
- If the value in the `month` variable is 2, the program will branch to the `case 2:` section and execute the `System.out.println("February")` statement that immediately follows it. The `break` statement then causes the program to exit the `switch` statement.
- If the value in the `month` variable is 3, the program will branch to the `case 3:` section and execute the `System.out.println("March")` statement that immediately follows it. The `break` statement then causes the program to exit the `switch` statement.
- If the value in the `month` variable is not 1, 2, or 3, the program will branch to the `default:` section and execute the `System.out.println("Error: Invalid month")` statement that immediately follows it.

The `switch` statement can be used as an alternative to an `if-else-if` statement that compares the same variable or expression to several different values. For example, the previously shown `switch` statement works like this `if-else-if` statement:

```
if (month == 1)
{
    System.out.println("January");
}
```

```
else if (month == 2)
{
    System.out.println("February");
}
else if (month == 3)
{
    System.out.println("March");
}
else
{
    System.out.println("Error: Invalid month");
}
```



NOTE: The default section is optional. If you leave it out, however, the program will have nowhere to branch to if the *testExpression* doesn't match any of the *case* values.

The program in Code Listing 3-13 shows how a simple switch statement works.

Code Listing 3-13 (SwitchDemo.java)

```
1 import java.util.Scanner; // Needed for Scanner class
2
3 /**
4  * This program demonstrates the switch statement.
5  */
6
7 public class SwitchDemo
8 {
9     public static void main(String[] args)
10 {
11     int number; // A number entered by the user
12
13     // Create a Scanner object for keyboard input.
14     Scanner keyboard = new Scanner(System.in);
15
16     // Get one of the numbers 1, 2, or 3 from the user.
17     System.out.print("Enter 1, 2, or 3: ");
18     number = keyboard.nextInt();
19
20     // Determine the number entered.
21     switch (number)
22     {
23         case 1:
24             System.out.println("You entered 1.");
```

```

25         break;
26     case 2:
27         System.out.println("You entered 2.");
28         break;
29     case 3:
30         System.out.println("You entered 3.");
31         break;
32     default:
33         System.out.println("That's not 1, 2, or 3!");
34     }
35 }
36 }

```

Program Output with Example Input Shown in Bold

Enter 1, 2, or 3: **2** [Enter]
 You entered 2.

Program Output with Example Input Shown in Bold

Enter 1, 2, or 3: **5** [Enter]
 That's not 1, 2, or 3!

Notice the `break` statements that are in the `case 1`, `case 2`, and `case 3` sections.

```

switch (number)
{
    case 1:
        System.out.println("You entered 1.");
        break;
    case 2:
        System.out.println("You entered 2.");
        break;
    case 3:
        System.out.println("You entered 3.");
        break;
    default:
        System.out.println("That's not 1, 2, or 3!");
}

```

The `case` statements show the program where to start executing in the block and the `break` statements show the program where to stop. Without the `break` statements, the program would execute all of the lines from the matching `case` statement to the end of the block.



NOTE: The default section (or the last `case` section if there is no default) does not need a `break` statement. Some programmers prefer to put one there anyway for consistency.

The program in Code Listing 3-14 is a modification of Code Listing 3-13, without the break statements.

Code Listing 3-14 (NoBreaks.java)

```
1 import java.util.Scanner;    // Needed for Scanner class
2
3 /**
4  * This program demonstrates the switch statement.
5  */
6
7 public class NoBreaks
8 {
9     public static void main(String[] args)
10    {
11        int number;           // A number entered by the user
12
13        // Create a Scanner object for keyboard input.
14        Scanner keyboard = new Scanner(System.in);
15
16        // Get one of the numbers 1, 2, or 3 from the user.
17        System.out.print("Enter 1, 2, or 3: ");
18        number = keyboard.nextInt();
19
20        // Determine the number entered.
21        switch (number)
22        {
23            case 1:
24                System.out.println("You entered 1.");
25            case 2:
26                System.out.println("You entered 2.");
27            case 3:
28                System.out.println("You entered 3.");
29            default:
30                System.out.println("That's not 1, 2, or 3!");
31        }
32    }
33 }
```

Program Output with Example Input Shown in Bold

```
Enter 1, 2, or 3: 1 [Enter]
You entered 1.
You entered 2.
You entered 3.
That's not 1, 2, or 3!
```

Program Output with Example Input Shown in Bold

Enter 1, 2, or 3: **3** [Enter]
 You entered 3.
 That's not 1, 2, or 3!

Without the `break` statement, the program “falls through” all of the statements below the one with the matching case expression. Sometimes this is what you want. For instance, the program in Code Listing 3-15 asks the user to select a grade of pet food. The available choices are A, B, and C. The `switch` statement will recognize either uppercase or lowercase letters.

Code Listing 3-15 (PetFood.java)

```

1  import java.util.Scanner; // Needed for the Scanner class
2
3  /**
4   * This program demonstrates a switch statement.
5   */
6
7  public class PetFood
8  {
9      public static void main(String[] args)
10     {
11         String input;      // To hold the user's input
12         char foodGrade;    // Grade of pet food
13
14         // Create a Scanner object for keyboard input.
15         Scanner keyboard = new Scanner(System.in);
16
17         // Prompt the user for a grade of pet food.
18         System.out.println("Our pet food is available in " +
19                             "three grades:");
20         System.out.print("A, B, and C. Which do you want " +
21                           "pricing for? ");
22         input = keyboard.nextLine();
23         foodGrade = input.charAt(0);
24
25         // Display pricing for the selected grade.
26         switch(foodGrade)
27         {
28             case 'a':
29             case 'A':
30                 System.out.println("30 cents per lb.");
31                 break;
32             case 'b':
33             case 'B':
34                 System.out.println("20 cents per lb.");
35                 break;

```

```

36         case 'c':
37         case 'C':
38             System.out.println("15 cents per lb.");
39             break;
40         default:
41             System.out.println("Invalid choice.");
42     }
43 }
44 }

```

Program Output with Example Input Shown in Bold

Our pet food is available in three grades:
A, B, and C. Which do you want pricing for? **b** [Enter]
20 cents per lb.

Program Output with Example Input Shown in Bold

Our pet food is available in three grades:
A, B, and C. Which do you want pricing for? **B** [Enter]
20 cents per lb.

When the user enters 'a' the corresponding case has no statements associated with it, so the program falls through to the next case, which corresponds with 'A'.

```

        case 'a':
        case 'A':
            System.out.println("30 cents per lb.");
            break;

```

The same technique is used for 'b' and 'c'.

If you are using a version of Java prior to Java 7, a *switch* statement's *testExpression* can be a char, byte, short, or int value. Beginning with Java 7, however, the *testExpression* can also be a string. The program in Code Listing 3-16 demonstrates.

Code Listing 3-16 (Seasons.java)

```

1  import java.util.Scanner;
2
3  /**
4   * This program translates the English names of
5   * the seasons into Spanish.
6   */
7
8  public class Seasons
9  {
10     public static void main(String[] args)
11     {
12         String input;
13

```



```

14      // Create a Scanner object for keyboard input.
15      Scanner keyboard = new Scanner(System.in);
16
17      // Get a day from the user.
18      System.out.print("Enter the name of a season: ");
19      input = keyboard.nextLine();
20
21      // Translate the season to Spanish.
22      switch (input)
23      {
24          case "spring":
25              System.out.println("la primavera");
26              break;
27          case "summer":
28              System.out.println("el verano");
29              break;
30          case "autumn":
31          case "fall":
32              System.out.println("el otono");
33              break;
34          case "winter":
35              System.out.println("el invierno");
36              break;
37          default:
38              System.out.println("Please enter one of these words:\n"
39                  + "spring, summer, autumn, fall, or winter.");
40      }
41  }
42  }

```

Program Output with Example Input Shown in Bold

Enter the name of a season: **summer** [Enter]
 el verano

Program Output with Example Input Shown in Bold

Enter the name of a season: **fall** [Enter]
 el otono

**Checkpoint**

MyProgrammingLab™ www.myprogramminglab.com

- 3.24 Complete the following program skeleton by writing a `switch` statement that displays “one” if the user has entered 1, “two” if the user has entered 2, and “three” if the user has entered 3. If a number other than 1, 2, or 3 is entered, the program should display an error message.

```

import java.util.Scanner;
public class CheckPoint
{

```

```

public static void main(String[] args)
{
    int userNum;
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Enter one of the numbers " +
                    "1, 2, or 3: ");
    userNum = keyboard.nextInt();
    //
    // Write the switch statement here.
    //
}
}

```

- 3.25 Rewrite the following if-else-if statement as a switch statement.

```

if (selection == 'A')
    System.out.println("You selected A.");
else if (selection == 'B')
    System.out.println("You selected B.");
else if (selection == 'C')
    System.out.println("You selected C.");
else if (selection == 'D')
    System.out.println("You selected D.");
else
    System.out.println("Not good with letters, eh?");

```

- 3.26 Explain why you cannot convert the following if-else-if statement into a switch statement.

```

if (temp == 100)
    x = 0;
else if (population > 1000)
    x = 1;
else if (rate < .1)
    x = -1;

```

- 3.27 What is wrong with the following switch statement?

```

// This code has errors!!!
switch (temp)
{
    case temp < 0 :
        System.out.println("Temp is negative.");
        break;
    case temp = 0:
        System.out.println("Temp is zero.");
        break;
    case temp > 0 :
        System.out.println("Temp is positive.");
        break;
}

```

3.28 What will the following code display?

```
int funny = 7, serious = 15;
funny = serious * 2;
switch (funny)
{ case 0 :
    System.out.println("That is funny.");
    break;
  case 30:
    System.out.println("That is serious.");
    break;
  case 32:
    System.out.println("That is seriously funny.");
    break;
  default:
    System.out.println(funny);
}
```

3.10 The `System.out.printf` Method

CONCEPT: The `System.out.printf` method allows you to format output in a variety of ways.

When you display numbers with the `System.out.println` or `System.out.print` method, you have little control over the way the numbers appear. For example, a value of the `double` data type can be displayed with as many as 15 decimal places, as demonstrated by the following code:

```
double number = 10.0 / 6.0;
System.out.println(number);
```

This code will display:

```
1.6666666666666667
```

Quite often, you want to format numbers so they are displayed in a particular way. For example, you might want to round a floating-point number to a specific number of decimal places, or insert comma separators to make a number easier to read. Fortunately, Java gives us a way to do just that, and more, with the `System.out.printf` method. The method's general format is as follows:

```
System.out.printf(FormatString, ArgumentList)
```

In the general format, *FormatString* is a string that contains text, special formatting specifiers, or both. *ArgumentList* is a list of zero or more additional arguments, which will be formatted according to the format specifiers listed in the format string.

The simplest way you can use the `System.out.printf` method is with only a format string, and no additional arguments. Here is an example:

```
System.out.printf("I love Java programming.\n");
```


The format string in this example is "I love Java programming.\n". This method call does not perform any special formatting, however. It simply prints the string "I love Java programming.\n". Using the method in this fashion is exactly like using the System.out.print method.

In most cases you will call the System.out.printf method in the following manner:

- The format string will contain one or more format specifiers. A *format specifier* is a placeholder for a value that will be inserted into the string when it is displayed.
- After the format string, one or more additional arguments will appear. Each of the additional arguments will correspond to a format specifier that appears inside the format string.

The following code shows an example:

```
double sales = 12345.67;  
System.out.printf("Our sales are %f for the day.\n", sales);
```

Notice the following characteristics of the System.out.printf method call:

- Inside the format string, the %f is a format specifier. The letter f indicates that a floating-point value will be inserted into the string when it is displayed.
- Following the format string, the sales variable is passed as an argument. This argument corresponds to the %f format specifier that appears inside the format string.

When the System.out.printf method executes, the %f will not be displayed on the screen. In its place, the value of the sales argument will be displayed. Here is the output of the code:

```
Our sales are 12345.670000 for the day.
```

The diagram in Figure 3-25 shows how the sales variable corresponds to the %f format specifier.

Figure 3-25 The value of the sales variable is displayed in the place of the %f format specifier



```
System.out.printf("Our sales is %f for the day.\n", sales);
```

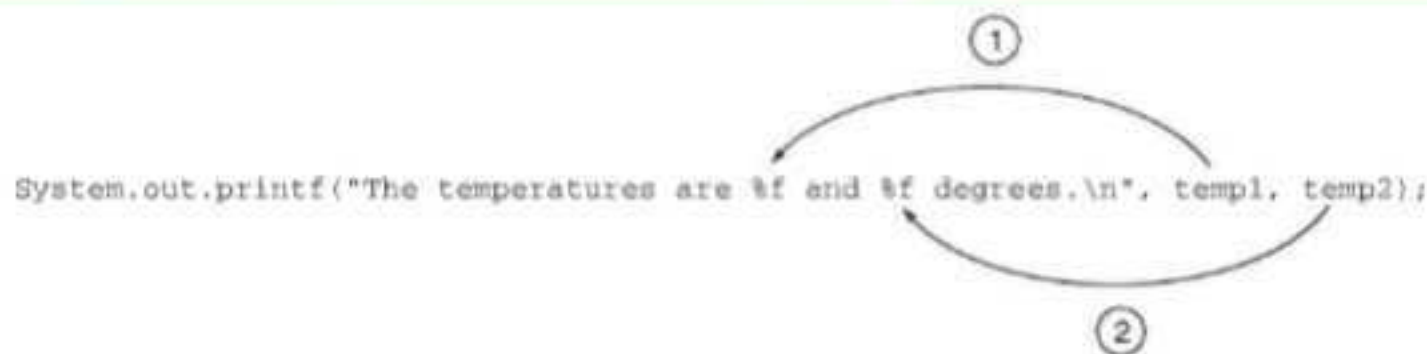
Here is another example:

```
double temp1 = 72.5, temp2 = 83.7;  
System.out.printf("The temperatures are %f and %f degrees.\n", temp1, temp2);
```

First, notice that this example uses two %f format specifiers in the format string. Also notice that two additional arguments appear after the format string. The value of the first argument, temp1, will be printed in place of the first %f, and the value of the second argument, temp2, will be printed in place of the second %f. The code will produce the following output:

```
The temperatures are 72.500000 and 83.700000 degrees.
```

There is a one-to-one correspondence between the format specifiers and the arguments that appear after the format string. The diagram in Figure 3-26 shows how the first format specifier corresponds to the first argument after the format string (the temp1 variable), and

Figure 3-26 The format specifiers and their corresponding arguments

the second format specifier corresponds to the second argument after the format string (the `temp2` variable).

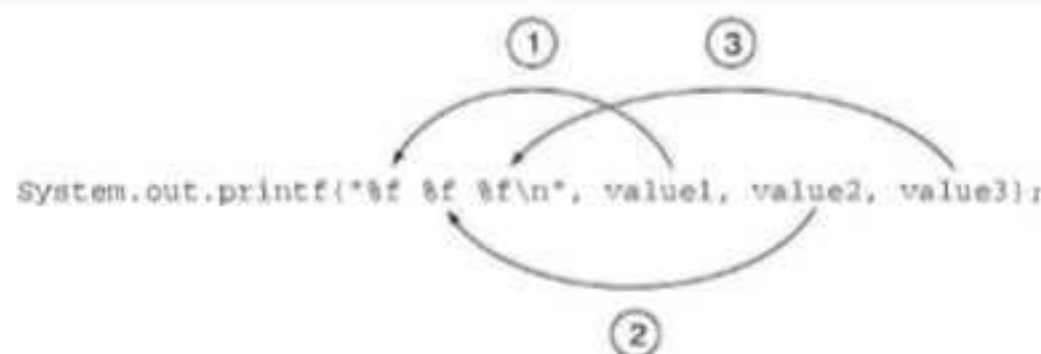
The following code shows another example:

```
double value1 = 3.0;
double value2 = 6.0;
double value3 = 9.0;
System.out.printf("%f %f %f\n", value1, value2, value3);
```

In the `System.out.printf` method call, there are three format specifiers and three additional arguments after the format string. This code will produce the following output:

```
3.000000 6.000000 9.000000
```

The diagram in Figure 3-27 shows how the format specifiers correspond to the arguments that appear after the format string.

Figure 3-27 The format specifiers and their corresponding arguments

The previous examples demonstrated how to format floating-point numbers with the `%f` format specifier. The letter `f` in the format specifier is a *conversion character* that indicates the data type of the argument that is being formatted. You use the `f` conversion character with any argument that is a `float` or a `double`.

If you want to format an integer value, you must use the `%d` format specifier. The `d` conversion character stands for decimal integer, and it can be used with arguments of the `int`, `short`, and `long` data types. Here is an example that displays an `int`:

```
int hours = 40;
System.out.printf("I worked %d hours this week.\n", hours);
```


In this example, the `%d` format specifier corresponds with the `hours` argument. This code will display the following:

```
I worked 40 hours this week.
```

Here is an example that displays two `int` values:

```
int dogs = 2;
int cats = 4;
System.out.printf("We have %d dogs and %d cats.\n", dogs, cats);
```

This code will display the following:

```
We have 2 dogs and 4 cats.
```

Keep in mind that `%f` must be used with floating-point values, and `%d` must be used with integer values. Otherwise, an error will occur at runtime.

Format Specifier Syntax

In the previous examples you saw how format specifiers correspond to the arguments that appear after the format string. Now you can learn how to use format specifiers to actually format the values that they correspond to. When displaying numbers, the general syntax for writing a format specifier is:

```
%[flags][width][.precision]conversion
```

The items that appear inside brackets are optional. Here is a summary of each item:

- `%`—All format specifiers begin with a `%` character.
- *flags*—After the `%` character, one or more optional flags may appear. Flags cause the value to be formatted in a variety of ways.
- *width*—After any flags, you can optionally specify the minimum field width for the value.
- *.precision*—If the value is a floating-point number, after the minimum field width, you can optionally specify the precision. This is the number of decimal places that the number should be rounded to.
- *conversion*—All format specifiers must end with a conversion character, such as `f` for floating-point, or `d` for decimal integer.

Let's take a closer look at each of the optional items, beginning with precision.

Precision

You probably noticed in the previous examples that the `%f` format specifier causes floating-point values to be displayed with six decimal places. You can change the number of decimal points that are displayed, as shown in the following example:

```
double temp = 78.42819;
System.out.printf("The temperature is %.2f degrees.\n", temp);
```

Notice that this example doesn't use the regular `%f` format specifier, but uses `%.2f` instead. The `.2` that appears between the `%` and the `f` specifies the *precision* of the displayed value. It

will cause the value of the `temp` variable to be rounded to two decimal places. This code will produce the following output:

```
The temperature is 78.43 degrees.
```

The following example displays the same value, rounded to one decimal place:

```
double temp = 78.42819;
System.out.printf("The temperature is %.1f degrees.\n", temp);
```

This code will produce the following output:

```
The temperature is 78.4 degrees.
```

The following code shows another example:

```
double value1 = 123.45678;
double value2 = 123.45678;
double value3 = 123.45678;
System.out.printf("%.1f %.2f %.3f\n", value1, value2, value3);
```

In this example, `value1` is rounded to one decimal place, `value2` is rounded to two decimal places, and `value3` is rounded to three decimal places. This code will produce the following output:

```
123.5 123.46 123.457
```

Keep in mind that you can specify precision only with floating-point values. If you specify a precision with the `%d` format specifier, an error will occur at runtime.

Specifying a Minimum Field Width

A format specifier can also include a *minimum field width*, which is the minimum number of spaces that should be used to display the value. The following example prints a floating-point number in a field that is 20 spaces wide:

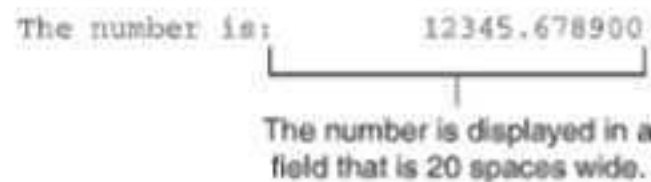
```
double number = 12345.6789;
System.out.printf("The number is:%20f\n", number);
```

Notice that the number 20 appears in the format specifier, between the `%` and the `f`. This code will produce the following output:

```
The number is:      12345.678900
```

In this example, the 20 that appears inside the `%f` format specifier indicates that the number should be displayed in a field that is a minimum of 20 spaces wide. This is illustrated in Figure 3-28.

Figure 3-28 The number is displayed in a field that is 20 spaces wide



```
The number is:      12345.678900
```

The number is displayed in a field that is 20 spaces wide.

In this case, the number that is displayed is shorter than the field in which it is displayed. The number 12345.678900 uses only 12 spaces on the screen, but it is displayed in a field that is 20 spaces wide. When this is the case, the number will be right-justified in the field. If a value is too large to fit in the specified field width, the field is automatically enlarged to accommodate it. The following example prints a floating-point number in a field that is only one space wide:

```
double number = 12345.6789;
System.out.printf("The number is:%1f\n", number);
```

The value of the number variable requires more than one space, however, so the field width is expanded to accommodate the entire number. This code will produce the following output:

```
The number is:12345.678900
```

You can specify a minimum field width for integers, as well as for floating-point values. The following example displays an integer 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
```

Combining Minimum Field Width and Precision in the Same Format Specifier

When specifying the minimum field width and the precision of a floating-point number in the same format specifier, remember that the field width must appear first, followed by the precision. For example, the following code displays a number in a field of 12 spaces, rounded to two decimal places:

```
double number = 12345.6789;
System.out.printf("The number is:%12.2f\n", number);
```

This code will produce the following output:

```
The number is:   12345.68
```

Field widths can help when you need to print numbers aligned in columns. For example, look at Code Listing 3-17. Each of the variables is displayed in a field that is eight spaces wide, and rounded to two decimal places. The numbers appear aligned in a column.

Code Listing 3-17 (Columns.java)

```
1  /**
2   * This program displays a variety of
3   * floating-point numbers in a column
4   * with their decimal points aligned.
5   */
6
7  public class Columns
```