CHAPTER 2

Java Fundamentals

TOPICS

2.1 The Parts of a Java Program

CONCEPT: A Java program has parts that serve specific purposes.

Java programs are made up of different parts. Your first step in learning Java is to learn what the parts are. We will begin by looking at a simple example, shown in Code Listing 2-1.

Code Listing 2-1 (`Simple.java`)

```
1  // This is a simple Java program.
2
3  public class Simple
4  {
5     public static void main(String[] args)
6     {
7        System.out.println("Programming is great fun!");
8     }
9  }
```

TIP: Remember, the line numbers shown in the program listings are not part of the program. The numbers are shown so we can refer to specific lines in the programs.

As mentioned in Chapter 1, the names of Java source code files end with *.java*. The program shown in Code Listing 2-1 is named *Simple.java*. Using the Java compiler, this program may be compiled with the following command:

```
javac Simple.java
```

The compiler will create another file named *Simple.class*, which contains the translated Java byte code. This file can be executed with the following command:

```
java Simple
```

TIP: Remember, you do not type the *.class* extension when using the `java` command.

The output of the program is as follows. This is what appears on the screen when the program runs.

Program Output

```
Programming is great fun!
```

Let's examine the program line by line. Here's the statement in line 1:

```
// This is a simple Java program.
```

Other than the two slash marks that begin this line, it looks pretty much like an ordinary sentence. The `//` marks the beginning of a comment. The compiler ignores everything from the double-slash to the end of the line. That means you can type anything you want on that line and the compiler never complains. Although comments are not required, they are very important to programmers. Most programs are much more complicated than this example, and comments help explain what's going on.

Line 2 is blank. Programmers often insert blank lines in programs to make them easier to read. Line 3 reads:

```
public class Simple
```

This line is known as a *class header*, and it marks the beginning of a *class definition*. One of the uses of a class is to serve as a container for an application. As you progress through this book, you will learn more and more about classes. For now, just remember that a Java program must have at least one class definition. This line of code consists of three words: `public`, `class`, and `Simple`. Let's take a closer look at each word.

- `public` is a Java key word, and it must be written in all lowercase letters. It is known as an *access specifier*, and it controls where the class may be accessed from. The `public` specifier means access to the class is unrestricted. (In other words, the class is "open to the public.")
- `class`, which must also be written in lowercase letters, is a Java key word that indicates the beginning of a class definition.

- `Simple` is the class name. This name was made up by the programmer. The class could have been called `Pizza`, or `Dog`, or anything else the programmer wanted. Programmer-defined names may be written in lowercase letters, uppercase letters, or a mixture of both.

In a nutshell, this line of code tells the compiler that a publicly accessible class named `Simple` is being defined. Here are two more points to know about classes:

- You may create more than one class in a file, but you may have only one `public class` per Java file.
- When a Java file has a `public` class, the name of the public class must be the same as the name of the file (without the *.java* extension). For instance, the program in Code Listing 2-1 has a `public class` named `Simple`, so it is stored in a file named *Simple.java*.

NOTE: Java is a case-sensitive language. That means it regards uppercase letters as being entirely different characters than their lowercase counterparts. The word `Public` is not the same as `public`, and `Class` is not the same as `class`. Some words in a Java program must be entirely in lowercase, while other words may use a combination of lower and uppercase characters. Later in this chapter, you will see a list of all the Java key words, which must appear in lowercase.

Line 4 contains only a single character:

```
{
```

This is called a left brace, or an opening brace, and is associated with the beginning of the class definition. All of the programming statements that are part of the class are enclosed in a set of braces. If you glance at the last line in the program, line 9, you'll see the closing brace. Everything between the two braces is the *body* of the class named `Simple`. Here is the program code again, this time the body of the class definition is shaded.

```
// This is a simple Java program.
public class Simple
{
   public static void main(String[] args)
   {
      System.out.println("Programming is great fun!");
   }
}
```

WARNING! Make sure you have a closing brace for every opening brace in your program!

Line 5 reads:

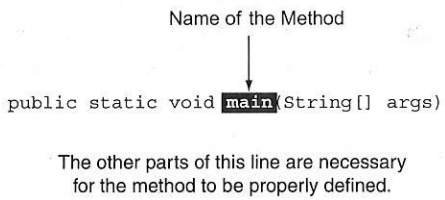
```
public static void main(String[] args)
```

This line is known as a *method header*. It marks the beginning of a *method*. A method can be thought of as a group of one or more programming statements that collectively has a name. When creating a method, you must tell the compiler several things about it. That is

why this line contains so many words. At this point, the only thing you should be concerned about is that the name of the method is `main`, and the rest of the words are required for the method to be properly defined. This is shown in Figure 2-1.

Recall from Chapter 1 that a stand-alone Java program that runs on your computer is known as an application. Every Java application must have a method named `main`. The `main` method is the starting point of an application.

Figure 2-1 The `main` method header



NOTE: For the time being, all the programs you will write will consist of a class with a `main` method whose header looks exactly like the one shown in Code Listing 2-1. As you progress through this book you will learn what `public static void` and `(String[] args)` mean. For now, just assume that you are learning a "recipe" for assembling a Java program.

Line 6 has another opening brace:

```
{
```

This opening brace belongs to the `main` method. Remember that braces enclose statements, and every opening brace must have an accompanying closing brace. If you look at line 8 you will see the closing brace that corresponds with this opening brace. Everything between these braces is the *body* of the `main` method.

Line 7 appears as follows:

```
System.out.println("Programming is great fun!");
```

To put it simply, this line displays a message on the screen. The message, "Programming is great fun!" is printed without the quotation marks. In programming terms, the group of characters inside the quotation marks is called a *string literal.*

NOTE: This is the only line in the program that causes anything to be printed on the screen. The other lines, like `public class Simple` and `public static void main(String[] args)`, are necessary for the framework of your program, but they do not cause any screen output. Remember, a program is a set of instructions for the computer. If something is to be displayed on the screen, you must use a programming statement for that purpose.

At the end of the line is a *semicolon.* Just as a period marks the end of a sentence, a semicolon marks the end of a statement in Java. Not every line of code ends with a semicolon, however. Here is a summary of where you do not place a semicolon:

- Comments do not have to end with a semicolon because they are ignored by the compiler.
- Class headers and method headers do not end with a semicolon because they are terminated with a body of code inside braces.
- The brace characters, `{` and `}`, are not statements, so you do not place a semicolon after them.

It might seem that the rules for where to put a semicolon are not clear at all. For now, just concentrate on learning the parts of a program. You'll soon get a feel for where you should and should not use semicolons.

As has already been pointed out, lines 8 and 9 contain the closing braces for the `main` method and the class definition:

```
    }
}
```

Before continuing, let's review the points we just covered, including some of the more elusive rules.

- Java is a case-sensitive language. It does not regard uppercase letters as being the same character as their lowercase equivalents.
- All Java programs must be stored in a file with a name that ends with *.java*.
- Comments are ignored by the compiler.
- A *.java* file may contain many classes, but may have only one `public` class. If a *.java* file has a public class, the class must have the same name as the file. For instance, if the file *Pizza.java* contains a `public class`, the class's name would be `Pizza`.
- Every Java application program must have a method named `main`.
- For every left brace, or opening brace, there must be a corresponding right brace, or closing brace.
- Statements are terminated with semicolons. This does not include comments, class headers, method headers, or braces.

In the sample program, you encountered several special characters. Table 2-1 summarizes how they were used.

Table 2-1 Special characters

Characters	Name	Meaning
`//`	Double slash	Marks the beginning of a comment
`( )`	Opening and closing parentheses	Used in a method header
`{ }`	Opening and closing braces	Encloses a group of statements, such as the contents of a class or a method
`" "`	Quotation marks	Encloses a string of characters, such as a message that is to be printed on the screen
`;`	Semicolon	Marks the end of a complete programming statement

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

2.1 The following program will not compile because the lines have been mixed up.

```
public static void main(String[] args)
}
// A crazy mixed up program
public class Columbus
{
System.out.println("In 1492 Columbus sailed the ocean blue.");
{
}
```

When the lines are properly arranged, the program should display the following on the screen:

```
In 1492 Columbus sailed the ocean blue.
```

Rearrange the lines in the correct order. Test the program by entering it on the computer, compiling it, and running it.

2.2 When the program in Question 2.1 is saved to a file, what should the file be named?

2.3 Complete the following program skeleton so it displays the message "Hello World" on the screen.

```
public class Hello
{
    public static void main(String[] args)
    {
        // Insert code here to complete the program
    }
}
```

2.4 On paper, write a program that will display your name on the screen. Place a comment with today's date at the top of the program. Test your program by entering, compiling, and running it.

2.5 All Java source code filenames must end with _____.

a) a semicolon
b) *.class*
c) *.java*
d) none of the above

2.6 Every Java application program must have _____.

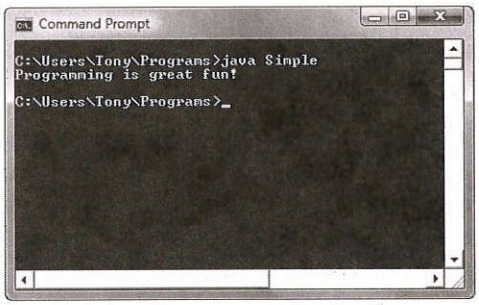
a) a method named `main`
b) more than one class definition
c) one or more comments

2.2 The `print` and `println` Methods, and the Java API

CONCEPT: The **`print`** and **`println`** methods are used to display text output. They are part of the Java API, which is a collection of prewritten classes and methods for performing specific operations.

In this section, you will learn how to write programs that produce output on the screen. The simplest type of output that a program can display on the screen is console output. *Console output* is merely plain text. When you display console output in a system that uses a graphical user interface, such as Windows or Mac OS, the output usually appears in a window similar to the one shown in Figure 2-2.

Figure 2-2 A console window (Microsoft Corporation)




VideoNote
Displaying Console Output

The word *console* is an old computer term. It comes from the days when the operator of a large computer system interacted with the system by typing on a terminal that consisted of a simple screen and keyboard. This terminal was known as the *console*. The console screen, which displayed only text, was known as the standard output device. Today, the term *standard output device* typically refers to the device that displays console output.

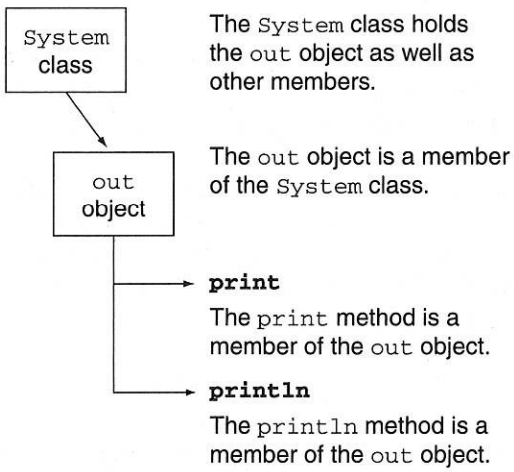
Performing output in Java, as well as many other tasks, is accomplished by using the Java API. The term *API* stands for *Application Programmer Interface*. The API is a standard library of prewritten classes for performing specific operations. These classes and their methods are available to all Java programs. The `print` and `println` methods are part of the API and provide ways for output to be displayed on the standard output device.

The program in Code Listing 2-1 (`Simple.java`) uses the following statement to print a message on the screen:

```
System.out.println("Programming is great fun!");
```

`System` is a class that is part of the Java API. The `System` class contains objects and methods that perform system-level operations. One of the objects contained in the `System` class is named `out`. The `out` object has methods, such as `print` and `println`, for performing output on the system console, or standard output device. The hierarchical relationship among `System`, `out`, `print`, and `println` is shown in Figure 2-3.

Figure 2-3 Relationship among the `System` class, the `out` object, and the `print` and `println` methods



Here is a brief summary of how it all works together:

- The `System` class is part of the Java API. It has member objects and methods for performing system-level operations, such as sending output to the console.
- The `out` object is a member of the `System` class. It provides methods for sending output to the screen.
- The `print` and `println` methods are members of the `out` object. They actually perform the work of writing characters on the screen.

This hierarchy explains why the statement that executes `println` is so long. The sequence `System.out.println` specifies that `println` is a member of `out`, which is a member of `System`.

NOTE: The period that separates the names of the objects is pronounced "dot." `System.out.println` is pronounced "system dot out dot print line."

The value that is to be displayed on the screen is placed inside the parentheses. This value is known as an *argument*. For example, the following statement executes the `println` method using the string `"King Arthur"` as its argument. This will print "King Arthur" on the screen. (The quotation marks are not displayed.)

```
System.out.println("King Arthur");
```

An important thing to know about the `println` method is that after it displays its message, it advances the cursor to the beginning of the next line. The next item printed on the screen will begin in this position. For example, look at the program in Code Listing 2-2.

Because each string is printed with separate `println` statements in Code Listing 2-2, they appear on separate lines in the Program Output.

Code Listing 2-2 (TwoLines.java)

```
1  // This is another simple Java program.
2
3  public class TwoLines
4  {
5     public static void main(String[] args)
6     {
7        System.out.println("Programming is great fun!");
8        System.out.println("I can't get enough of it!");
9     }
10 }
```

Program Output

```
Programming is great fun!
I can't get enough of it!
```

The print Method

The print method, which is also part of the System.out object, serves a purpose similar to that of println—to display output on the screen. The print method, however, does not advance the cursor to the next line after its message is displayed. Look at Code Listing 2-3.

Code Listing 2-3 (GreatFun.java)

```
1  // This is another simple Java program.
2
3  public class GreatFun
4  {
5     public static void main(String[] args)
6     {
7        System.out.print("Programming is ");
8        System.out.println("great fun!");
9     }
10 }
```

Program Output

```
Programming is great fun!
```

An important concept to understand about Code Listing 2-3 is that, although the output is broken up into two programming statements, this program will still display the message on one line. The data that you send to the print method is displayed in a continuous stream. Sometimes this can produce less-than-desirable results. The program in Code Listing 2-4 is an example.

Code Listing 2-4 (Unruly.java)

```
1  // An unruly printing program
2
3  public class Unruly
4  {
5     public static void main(String[] args)
6     {
7        System.out.print("These are our top sellers:");
8        System.out.print("Computer games");
9        System.out.print("Coffee");
10       System.out.println("Aspirin");
11    }
12 }
```

Program Output

```
These are our top sellers:Computer gamesCoffeeAspirin
```

The layout of the actual output looks nothing like the arrangement of the strings in the source code. First, even though the output is broken up into four lines in the source code (lines 7 through 10), it comes out on the screen as one line. Second, notice that some of the words that are displayed are not separated by spaces. The strings are displayed exactly as they are sent to the `print` method. If spaces are to be displayed, they must appear in the strings.

There are two ways to fix this program. The most obvious way is to use `println` methods instead of `print` methods. Another way is to use escape sequences to separate the output into different lines. An *escape sequence* starts with the backslash character (\), and is followed by one or more *control characters*. It allows you to control the way output is displayed by embedding commands within the string itself. The escape sequence that causes the output cursor to go to the next line is `\n`. Code Listing 2-5 illustrates its use.

Code Listing 2-5 (Adjusted.java)

```
1  // A well adjusted printing program
2
3  public class Adjusted
4  {
5     public static void main(String[] args)
6     {
7        System.out.print("These are our top sellers:\n");
8        System.out.print("Computer games\nCoffee\n");
9        System.out.println("Aspirin");
10    }
11 }
```

Program Output

```
These are our top sellers:
Computer games
Coffee
Aspirin
```

The \n characters are called the newline escape sequence. When the print or println method encounters \n in a string, it does not print the \n characters on the screen, but interprets them as a special command to advance the output cursor to the next line. There are several other escape sequences as well. For instance, \t is the tab escape sequence. When print or println encounters it in a string, it causes the output cursor to advance to the next tab position. Code Listing 2-6 shows it in use.

Code Listing 2-6 (Tabs.java)

```
1  // Another well-adjusted printing program
2
3  public class Tabs
4  {
5     public static void main(String[] args)
6     {
7        System.out.print("These are our top sellers:\n");
8        System.out.print("\tComputer games\n\tCoffee\n");
9        System.out.println("\tAspirin");
10    }
11 }
```

Program Output

```
These are our top sellers:
        Computer games
        Coffee
        Aspirin
```

NOTE: Although you have to type two characters to write an escape sequence, they are stored in memory as a single character.

Table 2-2 lists the common escape sequences and describes them.

Table 2-2 Common escape sequences

Escape Sequence	Name	Description
\n	Newline	Advances the cursor to the next line for subsequent printing
\t	Horizontal tab	Causes the cursor to skip over to the next tab stop
\b	Backspace	Causes the cursor to back up, or move left, one position
\r	Return	Causes the cursor to go to the beginning of the current line, not the next line
\\	Backslash	Causes a backslash to be printed
\'	Single quote	Causes a single quotation mark to be printed
\"	Double quote	Causes a double quotation mark to be printed

WARNING! Do not confuse the backslash (\) with the forward slash (/). An escape sequence will not work if you accidentally start it with a forward slash. Also, do not put a space between the backslash and the control character.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

2.7 The following program will not compile because the lines have been mixed up.

```
System.out.print("Success\n");
}
public class Success
{
System.out.print("Success\n");
public static void main(String[] args)
System.out.print("Success ");
}
// It's a mad, mad program.
System.out.println("\nSuccess");
{
```

When the lines are arranged properly, the program should display the following output on the screen:

Program Output

```
Success
Success Success

Success
```

Rearrange the lines in the correct order. Test the program by entering it on the computer, compiling it, and running it.

2.8 Study the following program and show what it will print on the screen.

```
// The Works of Wolfgang
public class Wolfgang
{
   public static void main(String[] args)
   {
      System.out.print("The works of Wolfgang\ninclude ");
      System.out.print("the following");
      System.out.print("\nThe Turkish March ");
      System.out.print("and Symphony No. 40 ");
      System.out.println("in G minor.");
   }
}
```

2.9 On paper, write a program that will display your name on the first line; your street address on the second line; your city, state, and ZIP code on the third line; and your telephone number on the fourth line. Place a comment with today's date at the top of the program. Test your program by entering, compiling, and running it.

2.3 Variables and Literals

CONCEPT: **A variable is a named storage location in the computer's memory. A literal is a value that is written into the code of a program.**

As you discovered in Chapter 1, variables allow you to store and work with data in the computer's memory. Part of the job of programming is to determine how many variables a program will need and what types of data they will hold. The program in Code Listing 2-7 is an example of a Java program with a variable.

Code Listing 2-7 (Variable.java)

```
1  // This program has a variable.
2
3  public class Variable
4  {
5     public static void main(String[] args)
6     {
7        int value;
8
9        value = 5;
10       System.out.print("The value is ");
11       System.out.println(value);
12    }
13 }
```

Program Output

```
The value is 5
```

Let's look more closely at this program. Here is line 7:

```
int value;
```

VideoNote
Declaring Variables

This is called a *variable declaration*. Variables must be declared before they can be used. A variable declaration tells the compiler the variable's name and the type of data it will hold. This line indicates the variable's name is value. The word `int` stands for integer, so value will only be used to hold integer numbers. Notice that variable declarations end with a semicolon. The next statement in this program appears in line 9:

```
value = 5;
```

This is called an *assignment statement*. The equal sign is an operator that stores the value on its right (in this case 5) into the variable named on its left. After this line executes, the value variable will contain the value 5.

NOTE: This line does not print anything on the computer screen. It runs silently behind the scenes.

Now look at lines 10 and 11:

```
System.out.print("The value is ");
System.out.println(value);
```

The statement in line 10 sends the string literal "The value is " to the `print` method. The statement in line 11 sends the name of the value variable to the `println` method. When you send a variable name to `print` or `println`, the variable's contents are displayed. Notice there are no quotation marks around `value`. Look at what happens in Code Listing 2-8.

Code Listing 2-8 (`Variable2.java`)

```
1  // This program has a variable.
2
3  public class Variable2
4  {
5     public static void main(String[] args)
6     {
7        int value;
8
9        value = 5;
10       System.out.print("The value is ");
11       System.out.println("value");
12    }
13 }
```

Program Output

```
The value is value
```

When double quotation marks are placed around the word `value` it becomes a string literal, not a variable name. When string literals are sent to `print` or `println`, they are displayed exactly as they appear inside the quotation marks.

Displaying Multiple Items with the + Operator

When the + operator is used with strings, it is known as the *string concatenation operator*. To concatenate means to append, so the string concatenation operator appends one string to another. For example, look at the following statement:

```
System.out.println("This is " + "one string.");
```

This statement will print:

```
This is one string.
```

The + operator produces a string that is the combination of the two strings used as its operands. You can also use the + operator to concatenate the contents of a variable to a string. The following code shows an example:

```
number = 5;
System.out.println("The value is " + number);
```

The second line uses the + operator to concatenate the contents of the `number` variable with the string "The value is ". Although `number` is not a string, the + operator converts its value to a string and then concatenates that value with the first string. The output that will be displayed is:

```
The value is 5
```

Sometimes the argument you use with `print` or `println` is too long to fit on one line in your program code. However, a string literal cannot begin on one line and end on another. For example, the following will cause an error:

```
// This is an error!
System.out.println("Enter a value that is greater than zero
                   and less than 10." );
```

You can remedy this problem by breaking the argument up into smaller string literals, and then using the string concatenation operator to spread them out over more than one line. Here is an example:

```
System.out.println("Enter a value that is " +
                   "greater than zero and less " +
                   "than 10." );
```

In this statement, the argument is broken up into three strings and joined using the + operator. The following example shows the same technique used when the contents of a variable are part of the concatenation:

```
sum = 249;
System.out.println("The sum of the three " +
                   "numbers is " + sum);
```

Be Careful with Quotation Marks

As shown in Code Listing 2-8, placing quotation marks around a variable name changes the program's results. In fact, placing double quotation marks around anything that is not intended to be a string literal will create an error of some type. For example, in Code Listings 2-7 and 2-8, the number 5 was assigned to the variable `value`. It would have been an error to perform the assignment this way:

```
value = "5";     // Error!
```

In this statement, 5 is no longer an integer, but a string literal. Because `value` was declared as an integer variable, you can only store integers in it. In other words, 5 and "5" are not the same thing.

The fact that numbers can be represented as strings frequently confuses students who are new to programming. Just remember that strings are intended for humans to read. They are to be printed on computer screens or paper. Numbers, however, are intended primarily for mathematical operations. You cannot perform math on strings, and before numbers can be displayed on the screen, first they must be converted to strings. (Fortunately, `print` and `println` handle the conversion automatically when you send numbers to them.) Don't fret if this still bothers you. Later in this chapter, we will shed more light on the differences among numbers, characters, and strings by discussing their internal storage.

More about Literals

A literal is a value that is written in the code of a program. Literals are commonly assigned to variables or displayed. Code Listing 2-9 contains both literals and a variable.

Code Listing 2-9 (Literals.java)

```
1  // This program has literals and a variable.
2
3  public class Literals
4  {
5     public static void main(String[] args)
6     {
7        int apples;
8
9        apples = 20;
10       System.out.println("Today we sold " + apples +
11                          " bushels of apples.");
12    }
13 }
```

Program Output

```
Today we sold 20 bushels of apples.
```

Of course, the variable in this program is `apples`. It is declared as an integer. Table 2-3 shows a list of the literals found in the program.

Table 2-3 Literals

Literal	Type of Literal
20	Integer literal
"Today we sold "	String literal
" bushels of apples."	String literal

Identifiers

An *identifier* is a programmer-defined name that represents some element of a program. Variable names and class names are examples of identifiers. You may choose your own variable names and class names in Java, as long as you do not use any of the Java key words. The *key words* make up the core of the language and each has a specific purpose. Table 1-3 in Chapter 1 and Appendix D (available on the book's companion Web site) show a complete list of Java key words.

You should always choose names for your variables that give an indication of what they are used for. You may be tempted to declare variables with names like this:

```
int x;
```

The rather nondescript name, x, gives no clue as to what the variable's purpose is. Here is a better example.

```
int itemsOrdered;
```

The name `itemsOrdered` gives anyone reading the program an idea of what the variable is used for. This method of coding helps produce *self-documenting programs*, which means you get an understanding of what the program is doing just by reading its code. Because real-world programs usually have thousands of lines of code, it is important that they be as self-documenting as possible.

You have probably noticed the mixture of uppercase and lowercase letters in the name `itemsOrdered`. Although all of Java's key words must be written in lowercase, you may use uppercase letters in variable names. The reason the `O` in `itemsOrdered` is capitalized is to improve readability. Normally "items ordered" is used as two words. Variable names cannot contain spaces, however, so the two words must be combined. When "items" and "ordered" are stuck together, you get a variable declaration like this:

```
int itemsordered;
```

Capitalization of the letter `O` makes `itemsOrdered` easier to read. Typically, variable names begin with a lowercase letter, and after that, the first letter of each individual word that makes up the variable name is capitalized.

The following are some specific rules that must be followed with all identifiers:

- The first character must be one of the letters a–z or A–Z, an underscore (_), or a dollar sign ($).
- After the first character, you may use the letters a–z or A–Z, the digits 0–9, underscores (_), or dollar signs ($).
- Uppercase and lowercase characters are distinct. This means `itemsOrdered` is not the same as `itemsordered`.
- Identifiers cannot include spaces.

NOTE: Although the $ is a legal identifier character, it is normally used for special purposes. So, don't use it in your variable names.

Table 2-4 shows a list of variable names and tells whether each is legal or illegal in Java.

Table 2-4 Some variable names

Variable Name	Legal or Illegal?
`dayOfWeek`	Legal
`3dGraph`	Illegal because identifiers cannot begin with a digit
`june1997`	Legal
`mixture#3`	Illegal because identifiers may use only alphabetic letters, digits, underscores, or dollar signs
`week day`	Illegal because identifiers cannot contain spaces

Class Names

As mentioned before, it is standard practice to begin variable names with a lowercase letter, and then capitalize the first letter of each subsequent word that makes up the name. It is also a standard practice to capitalize the first letter of a class name, as well as the first letter of each subsequent word it contains. This helps differentiate the names of variables from the names of classes. For example, `payRate` would be a variable name, and `Employee` would be a class name.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

2.10 Examine the following program.

```
// This program uses variables and literals.
public class BigLittle
{
   public static void main(String[] args)
   {
      int little;
      int big;
      little = 2;
      big = 2000;
      System.out.println("The little number is " + little);
      System.out.println("The big number is " + big);
   }
}
```

List the variables and literals found in the program.

2.11 What will the following program display on the screen?

```
public class CheckPoint
{
   public static void main(String[] args)
   {
      int number;
      number = 712;
      System.out.println("The value is " + "number");
   }
}
```

2.4 Primitive Data Types

Computer programs collect pieces of data from the real world and manipulate them in various ways. There are many different types of data. In the realm of numeric data, for example, there are whole and fractional numbers, negative and positive numbers, and numbers so large and others so small that they don't even have a name. Then there is textual information. Names and addresses, for instance, are stored as strings of characters. When you write a program you must determine what types of data it is likely to encounter.

Each variable has a *data type*, which is the type of data that the variable can hold. Selecting the proper data type is important because a variable's data type determines the amount of memory the variable uses, and the way the variable formats and stores data. It is important to select a data type that is appropriate for the type of data that your program will work with. If you are writing a program to calculate the number of miles to a distant star, you need variables that can hold very large numbers. If you are designing software to record microscopic dimensions, you need variables that store very small and precise numbers. If you are writing a program that must perform thousands of intensive calculations, you want variables that can be processed quickly. The data type of a variable determines all of these factors.

Table 2-5 shows all of the Java *primitive data types* for holding numeric data.

The words listed in the left column of Table 2-5 are the key words that you use in variable declarations. A variable declaration takes the following general format:

```
DataType VariableName;
```

Table 2-5 Primitive data types for numeric data

Data Type	Size	Range
`byte`	1 byte	Integers in the range of -128 to +127
`short`	2 bytes	Integers in the range of -32,768 to +32,767
`int`	4 bytes	Integers in the range of -2,147,483,648 to +2,147,483,647
`long`	8 bytes	Integers in the range of -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
`float`	4 bytes	Floating-point numbers in the range of $\pm 3.4 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$, with 7 digits of accuracy
`double`	8 bytes	Floating-point numbers in the range of $\pm 1.7 \times 10^{-308}$ to $\pm 1.7 \times 10^{308}$, with 15 digits of accuracy

`DataType` is the name of the data type and `VariableName` is the name of the variable. Here are some examples of variable declarations:

```
byte inches;
int speed;
short month;
float salesCommission;
double distance;
```

The size column in Table 2-5 shows the number of bytes that a variable of each of the data types uses. For example, an `int` variable uses 4 bytes, and a `double` variable uses 8 bytes.

The range column shows the ranges of numbers that may be stored in variables of each data type. For example, an `int` variable can hold numbers from -2,147,483,648 up to +2,147,483,647. One of the appealing characteristics of the Java language is that the sizes and ranges of all the primitive data types are the same on all computers.

NOTE: These data types are called "primitive" because you cannot use them to create objects. Recall from Chapter 1's discussion on object-oriented programming that an object has attributes and methods. With the primitive data types, you can only create variables, and a variable can only be used to hold a single value. Such variables do not have attributes or methods.

The Integer Data Types

The first four data types listed in Table 2-5, `byte`, `int`, `short`, and `long`, are all integer data types. An integer variable can hold whole numbers such as 7, 125, -14, and 6928. The program in Code Listing 2-10 shows several variables of different integer data types being used.

Code Listing 2-10 (IntegerVariables.java)

```
1  // This program has variables of several of the integer types.
2
3  public class IntegerVariables
4  {
5     public static void main(String[] args)
6     {
7        int checking; // Declare an int variable named checking.
8        byte miles; // Declare a byte variable named miles.
9        short minutes; // Declare a short variable named minutes.
10       long days; // Declare a long variable named days.
11
12       checking = -20;
13       miles = 105;
14       minutes = 120;
15       days = 189000;
16       System.out.println("We have made a journey of " + miles +
17                          " miles.");
18       System.out.println("It took us " + minutes + " minutes.");
19       System.out.println("Our account balance is $" + checking);
20       System.out.println("About " + days + " days ago Columbus " +
21                          "stood on this spot.");
22    }
23 }
```

Program Output

```
We have made a journey of 105 miles.
It took us 120 minutes.
Our account balance is $-20
About 189000 days ago Columbus stood on this spot.
```

In most programs you will need more than one variable of any given data type. If a program uses three integers, `length`, `width`, and `area`, they could be declared separately, as follows:

```
int length;
int width;
int area;
```

It is easier, however, to combine the three variable declarations:

```
int length, width, area;
```

You can declare several variables of the same type, simply by separating their names with commas.

Integer Literals

When you write an integer literal in your program code, Java assumes it to be of the `int` data type. For example, in Code Listing 2-10, the literals -20, 105, 120, and 189000 are all treated as `int` values. You can force an integer literal to be treated as a long, however, by suffixing it with the letter L. For example, the value 57L would be treated as a long. Although you can use either an uppercase or a lowercase L, it is advisable to use the uppercase L because the lowercase l looks too much like the number 1.

WARNING! You cannot embed commas in numeric literals. For example, the following statement will cause an error:

```
number = 1,257,649;        // ERROR!
```

This statement must be written as:

```
number = 1257649;          // Correct.
```

Floating-Point Data Types

Whole numbers are not adequate for many jobs. If you are writing a program that works with dollar amounts or precise measurements, you need a data type that allows fractional values. In programming terms, these are called *floating-point* numbers. Values such as 1.7 and -45.316 are floating-point numbers.

In Java there are two data types that can represent floating-point numbers. They are `float` and `double`. The `float` data type is considered a single precision data type. It can store a floating-point number with 7 digits of accuracy. The `double` data type is considered a double precision data type. It can store a floating-point number with 15 digits of accuracy. The `double` data type uses twice as much memory as the `float` data type, however. A `float` variable occupies 4 bytes of memory, whereas a `double` variable uses 8 bytes.

Code Listing 2-11 shows a program that uses three `double` variables.

Code Listing 2-11 (`Sale.java`)

```
1  // This program demonstrates the double data type.
2
3  public class Sale
4  {
```

```
5     public static void main(String[] args)
6     {
7        double price, tax, total;
8
9        price = 29.75;
10       tax = 1.76;
11       total = 31.51;
12       System.out.println("The price of the item " +
13                          "is " + price);
14       System.out.println("The tax is " + tax);
15       System.out.println("The total is " + total);
16    }
17 }
```

Program Output

```
The price of the item is 29.75
The tax is 1.76
The total is 31.51
```

Floating-Point Literals

When you write a floating-point literal in your program code, Java assumes it to be of the `double` data type. For example, in Code Listing 2-11, the literals 29.75, 1.76, and 31.51 are all treated as `double` values. Because of this, a problem can arise when assigning a floating-point 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. A `double` value is not compatible with a `float` variable because a `double` can be much larger or much smaller than the allowable range for a `float`. As a result, code such as the following will cause an error:

```
float number;
number = 23.5;          // Error!
```

You can force a `double` literal to be treated as a `float`, however, by suffixing it with the letter F or f. The preceding code can be rewritten in the following manner to prevent an error:

```
float number;
number = 23.5F;         // This will work.
```

WARNING! If you are working with literals that represent dollar amounts, remember that you cannot embed currency symbols (such as $) or commas in the literal. For example, the following statement will cause an error:

```
grossPay = $1,257.00;     // ERROR!
```

This statement must be written as:

```
grossPay = 1257.00;       // Correct.
```

Scientific and E Notation

Floating-point literals can be represented in scientific notation. Take the number 47,281.97. In scientific notation this number is 4.728197×10^4. (10^4 is equal to 10,000, and $4.728197 \times 10{,}000$ is 47,281.97.)

Java uses E notation to represent values in scientific notation. In E notation, the number 4.728197×10^4 would be 4.728197E4. Table 2-6 shows other numbers represented in scientific and E notation.

Table 2-6 Floating-point representations

Decimal Notation	Scientific Notation	E Notation
247.91	2.4791×10^2	2.4791E2
0.00072	7.2×10^{-4}	7.2E–4
2,900,000	2.9×10^6	2.9E6

NOTE: The E can be uppercase or lowercase.

Code Listing 2-12 demonstrates the use of floating-point literals expressed in E notation.

Code Listing 2-12 (SunFacts.java)

```
1  // This program uses E notation.
2
3  public class SunFacts
4  {
5     public static void main(String[] args)
6     {
7        double distance, mass;
8
9        distance = 1.495979E11;
10       mass = 1.989E30;
11       System.out.println("The sun is " + distance +
12                          " meters away.");
13       System.out.println("The sun's mass is " + mass +
14                          " kilograms.");
15    }
16 }
```

Program Output

```
The sun is 1.495979E11 meters away.
The sun's mass is 1.989E30 kilograms.
```

The boolean Data Type

The boolean data type allows you to create variables that may hold one of two possible values: true or false. Code Listing 2-13 demonstrates the declaration and assignment of a boolean variable.

Code Listing 2-13 (TrueFalse.java)

```
1  // A program for demonstrating boolean variables
2
3  public class TrueFalse
4  {
5     public static void main(String[] args)
6     {
7        boolean bool;
8
9        bool = true;
10       System.out.println(bool);
11       bool = false;
12       System.out.println(bool);
13    }
14 }
```

Program Output

```
true
false
```

Variables of the boolean data type are useful for evaluating conditions that are either true or false. You will not be using them until Chapter 3, however, so for now just remember the following things:

- boolean variables may hold only the value true or false.
- The contents of a boolean variable may not be copied to a variable of any type other than boolean.

The char Data Type

The char data type is used to store characters. A variable of the char data type can hold one character at a time. Character literals are enclosed in *single quotation marks*. The program in Code Listing 2-14 uses a char variable. The character literals 'A' and 'B' are assigned to the variable.

Code Listing 2-14 (Letters.java)

```
1  // This program demonstrates the char data type.
2
3  public class Letters
4  {
5     public static void main(String[] args)
```

```
6     {
7        char letter;
8
9        letter = 'A';
10       System.out.println(letter);
11       letter = 'B';
12       System.out.println(letter);
13    }
14 }
```

Program Output

```
A
B
```

It is important that you do not confuse character literals with string literals, which are enclosed in double quotation marks. String literals cannot be assigned to `char` variables.

Unicode

Characters are internally represented by numbers. Each printable character, as well as many non-printable characters, is assigned a unique number. Java uses Unicode, which is a set of numbers that are used as codes for representing characters. Each Unicode number requires two bytes of memory, so `char` variables occupy two bytes. When a character is stored in memory, it is actually the numeric code 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.

You may want to refer to Appendix B, available on the book's companion Web site (at www.pearsonglobaleditions.com/Gaddis), which shows a portion of the Unicode character set. Notice that the number 65 is the code for A, 66 is the code for B, and so on. Code Listing 2-15 demonstrates that when you work with characters, you are actually working with numbers.

Code Listing 2-15 (`Letters2.java`)

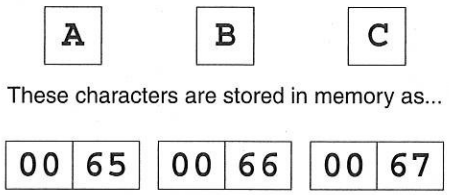
```
1  // This program demonstrates the close relationship between
2  // characters and integers.
3
4  public class Letters2
5  {
6     public static void main(String[] args)
7     {
8        char letter;
9
10       letter = 65;
11       System.out.println(letter);
12       letter = 66;
13       System.out.println(letter);
14    }
15 }
```

Program Output

```
A
B
```

Figure 2-4 illustrates that when you think of the characters A, B, and C being stored in memory, it is really the numbers 65, 66, and 67 that are stored.

Figure 2-4 Characters and how they are stored in memory



Variable Assignment and Initialization

As you have already seen in several examples, a value is put into a variable with an *assignment statement*. For example, the following statement assigns the value 12 to the variable `unitsSold`:

```
unitsSold = 12;
```

The = symbol is called the assignment operator. Operators perform operations on data. The data that operators work with are called operands. The assignment operator has two operands. In the statement above, the operands are `unitsSold` and `12`.

In an assignment statement, the name of the variable receiving the assignment must appear on the left side of the operator, and the value being assigned must appear on the right side. The following statement is incorrect:

```
12 = unitsSold;       // ERROR!
```

The operand on the left side of the = operator must be a variable name. The operand on the right side of the = symbol must be an expression that has a value. The assignment operator takes the value of the right operand and puts it in the variable identified by the left operand. Assuming that `length` and `width` are both `int` variables, the following code illustrates that the assignment operator's right operand may be a literal or a variable:

```
length = 20;
width = length;
```

It is important to note that the assignment operator only changes the contents of its left operand. The second statement assigns the value of the `length` variable to the `width` variable. After the statement has executed, `length` still has the same value, 20.

You may also assign values to variables as part of the declaration statement. This is known as *initialization*. Code Listing 2-16 shows how it is done.

The variable declaration statement in this program is in line 7:

```
int month = 2, days = 28;
```

Code Listing 2-16 (Initialize.java)

```
1  // This program shows variable initialization.
2
3  public class Initialize
4  {
5     public static void main(String[] args)
6     {
7        int month = 2, days = 28;
8
9        System.out.println("Month " + month + " has " +
10                          days + " days.");
11    }
12 }
```

Program Output

```
Month 2 has 28 days.
```

This statement declares the `month` variable and initializes it with the value 2, and declares the `days` variable and initializes it with the value 28. As you can see, this simplifies the program and reduces the number of statements that must be typed by the programmer. Here are examples of other declaration statements that perform initialization:

```
double payRate = 25.52;
float interestRate = 12.9F;
char stockCode = 'D';
int customerNum = 459;
```

Of course, there are always variations on a theme. Java allows you to declare several variables and initialize only some of them. Here is an example of such a declaration:

```
int flightNum = 89, travelTime, departure = 10, distance;
```

The variable `flightNum` is initialized to 89 and `departure` is initialized to 10. The `travelTime` and `distance` variables remain uninitialized.

WARNING! When a variable is declared inside a method, it must have a value stored in it before it can be used. If the compiler determines that the program might be using such a variable before a value has been stored in it, an error will occur. You can avoid this type of error by initializing the variable with a value.

Variables Hold Only One Value at a Time

Remember, a variable can hold only one value at a time. When you assign a new value to a variable, the new value takes the place of the variable's previous contents. For example, look at the following code.

```
int x = 5;
System.out.println(x);
x = 99;
System.out.println(x);
```

In this code, the variable x is initialized with the value 5 and its contents are displayed. Then the variable is assigned the value 99. This value overwrites the value 5 that was previously stored there. The code will produce the following output:

```
5
99
```

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

2.12 Which of the following are illegal variable names and why?

```
x
99bottles
july97
theSalesFigureForFiscalYear98
r&d
grade_report
```

2.13 Is the variable name `Sales` the same as `sales`? Why or why not?

2.14 Refer to the Java primitive data types listed in Table 2-5 for this question.

a) If a variable needs to hold whole numbers in the range 32 to 6,000, what primitive data type would be best?

b) If a variable needs to hold whole numbers in the range –40,000 to +40,000, what primitive data type would be best?

c) Which of the following literals use more memory? `22.1` or `22.1F`?

2.15 How would the number 6.31×10^{17} be represented in E notation?

2.16 A program declares a `float` variable named `number`, and the following statement causes an error. What can be done to fix the error?

```
number = 7.4;
```

2.17 What values can `boolean` variables hold?

2.18 Write statements that do the following:

a) Declare a `char` variable named `letter`.

b) Assign the letter A to the `letter` variable.

c) Display the contents of the `letter` variable.

2.19 What are the Unicode codes for the characters 'C', 'F', and 'W'? (You may need to refer to Appendix B on the book's companion Web site, at www.pearsonglobaleditions.com/Gaddis.)

2.20 Which is a character literal, `'B'` or `"B"`?

2.21 What is wrong with the following statement?

```
char letter = "Z";
```

2.5 Arithmetic Operators

CONCEPT: There are many operators for manipulating numeric values and performing arithmetic operations.

Java offers a multitude of operators for manipulating data. Generally, there are three types of operators: *unary*, *binary*, and *ternary*. These terms reflect the number of operands an operator requires.

VideoNote
Simple Math Expressions

Unary operators require only a single operand. For example, consider the following expression:

```
-5
```

Of course, we understand this represents the value negative five. We can also apply the operator to a variable, as follows:

```
-number
```

This expression gives the negative of the value stored in `number`. The minus sign, when used this way, is called the *negation operator*. Because it requires only one operand, it is a unary operator.

Binary operators work with two operands. The assignment operator is in this category. Ternary operators, as you may have guessed, require three operands. Java has only one ternary operator, which is discussed in Chapter 3.

Arithmetic operations are very common in programming. Table 2-7 shows the arithmetic operators in Java.

Table 2-7 Arithmetic operators

Operator	Meaning	Type	Example
+	Addition	Binary	`total = cost + tax;`
-	Subtraction	Binary	`cost = total - tax;`
*	Multiplication	Binary	`tax = cost * rate;`
/	Division	Binary	`salePrice = original / 2;`
%	Modulus	Binary	`remainder = value % 3;`

Each of these operators works as you probably expect. The addition operator returns the sum of its two operands. Here are some example statements that use the addition operator:

```
amount = 4 + 8;                 // Assigns 12 to amount
total = price + tax;            // Assigns price + tax to total
number = number + 1;            // Assigns number + 1 to number
```

The subtraction operator returns the value of its right operand subtracted from its left operand. Here are some examples:

```
temperature = 112 - 14;         // Assigns 98 to temperature
sale = price - discount;        // Assigns price - discount to sale
number = number - 1;            // Assigns number - 1 to number
```

The multiplication operator returns the product of its two operands. Here are some examples:

```
markUp = 12 * 0.25;             // Assigns 3 to markUp
commission = sales * percent;   // Assigns sales * percent to commission
population = population * 2;    // Assigns population * 2 to population
```

The division operator returns the quotient of its left operand divided by its right operand. Here are some examples:

```
points = 100 / 20;              // Assigns 5 to points
teams = players / maxEach;      // Assigns players / maxEach to teams
half = number / 2;              // Assigns number / 2 to half
```

The modulus operator returns the remainder of a division operation involving two integers. The following statement assigns 2 to `leftOver`:

```
leftOver = 17 % 3;
```

Situations arise where you need to get the remainder of a division. Computations that detect odd numbers or are required to determine how many items are left over after division use the modulus operator.

The program in Code Listing 2-17 demonstrates some of these operators used in a simple payroll calculation.

Code Listing 2-17 (Wages.java)

```
1  // This program calculates hourly wages plus overtime.
2
3  public class Wages
4  {
5     public static void main(String[] args)
6     {
7        double regularWages;       // The calculated regular wages.
8        double basePay = 25;       // The base pay rate.
9        double regularHours = 40;  // The hours worked less overtime.
10       double overtimeWages;      // Overtime wages
11       double overtimePay = 37.5; // Overtime pay rate
12       double overtimeHours = 10; // Overtime hours worked
13       double totalWages;         // Total wages
14
15       regularWages = basePay * regularHours;
16       overtimeWages = overtimePay * overtimeHours;
17       totalWages = regularWages + overtimeWages;
18       System.out.println("Wages for this week are $" +
19                          totalWages);
20    }
21 }
```

Program Output

```
Wages for this week are $1375.0
```

Code Listing 2-17 calculates the total wages an hourly paid worker earned in one week. As mentioned in the comments, there are variables for regular wages, base pay rate, regular hours worked, overtime wages, overtime pay rate, overtime hours worked, and total wages.

Line 15 in the program multiplies basePay times regularHours and stores the result, which is 1000, in regularWages:

```
regularWages = basePay * regularHours;
```

Line 16 multiplies overtimePay times overtimeHours and stores the result, which is 375, in overtimeWages:

```
overtimeWages = overtimePay * overtimeHours;
```

Line 17 adds the regular wages and the overtime wages and stores the result, 1375, in totalWages:

```
totalWages = regularWages + overtimeWages;
```

The println statement in lines 18 and 19 displays the message on the screen reporting the week's wages.

Integer Division

When both operands of the division operator are integers, the operator will perform *integer division*. This means the result of the division will be an integer as well. If there is a remainder, it will be discarded. For example, look at the following code:

```
double number;
number = 5 / 2;
```

This code divides 5 by 2 and assigns the result to the number variable. What value will be stored in number? You would probably assume that 2.5 would be stored in number because that is the result your calculator shows when you divide 5 by 2; however, that is not what happens when the previous Java code is executed. Because the numbers 5 and 2 are both integers, the fractional part of the result will be thrown away, or *truncated*. As a result, the value 2 will be assigned to the number variable.

In the previous code, it doesn't matter that number is declared as a double because the fractional part of the result is discarded before the assignment takes place. In order for a division operation to return a floating-point value, one of the operands must be of a floating-point data type. For example, the previous code could be written as follows:

```
double number;
number = 5.0 / 2;
```

In this code, 5.0 is treated as a floating-point number, so the division operation will return a floating-point number. The result of the division is 2.5.

Operator Precedence

It is possible to build mathematical expressions with several operators. The following statement assigns the sum of 17, x, 21, and y to the variable answer:

```
answer = 17 + x + 21 + y;
```

Some expressions are not that straightforward, however. Consider the following statement:

```
outcome = 12 + 6 / 3;
```

What value will be stored in `outcome`? The 6 is used as an operand for both the addition and division operators. The `outcome` variable could be assigned either 6 or 14, depending on when the division takes place. The answer is 14 because the division operator has higher *precedence* than the addition operator.

Mathematical expressions are evaluated from left to right. When two operators share an operand, the operator with the highest precedence works first. Multiplication and division have higher precedence than addition and subtraction, so the statement above works like this:

1. 6 is divided by 3, yielding a result of 2
2. 12 is added to 2, yielding a result of 14

It could be diagrammed as shown in Figure 2-5.

Figure 2-5 Precedence illustrated

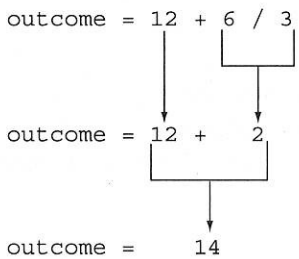


Table 2-8 shows the precedence of the arithmetic operators. The operators at the top of the table have higher precedence than the ones below them.

Table 2-8 Precedence of arithmetic operators (highest to lowest)

Highest Precedence →	- (unary negation)
	* / %
Lowest Precedence →	+ -

The multiplication, division, and modulus operators have the same precedence. The addition and subtraction operators have the same precedence. If two operators sharing an operand have the same precedence, they work according to their *associativity*. Associativity is either *left to right* or *right to left*. Table 2-9 shows the arithmetic operators and their associativity.

Table 2-9 Associativity of arithmetic operators

Operator	Associativity
- (unary negation)	Right to left
* / %	Left to right
+ -	Left to right

Table 2-10 shows some expressions and their values.

Table 2-10 Some expressions and their values

Expression	Value
`5 + 2 * 4`	13
`10 / 2 - 3`	2
`8 + 12 * 2 - 4`	28
`4 + 17 % 2 - 1`	4
`6 - 3 * 2 + 7 - 1`	6

Grouping with Parentheses

Parts of a mathematical expression may be grouped with parentheses to force some operations to be performed before others. In the statement below, the sum of `a`, `b`, `c`, and `d` is divided by `4.0`.

```
average = (a + b + c + d) / 4.0;
```

Without the parentheses, however, `d` would be divided by `4` and the result added to `a`, `b`, and `c`. Table 2-11 shows more expressions and their values.

Table 2-11 More expressions and their values

Expression	Value
`(5 + 2) * 4`	28
`10 / (5 - 3)`	5
`8 + 12 * (6 - 2)`	56
`(4 + 17) % 2 - 1`	0
`(6 - 3) * (2 + 7) / 3`	9

In the Spotlight:

Calculating Percentages and Discounts

Determining percentages is a common calculation in computer programming. Although the % symbol is used in general mathematics to indicate a percentage, most programming languages (including Java) do not use the % symbol for this purpose. In a program, you have to convert a percentage to a floating-point number, just as you would if you were using a calculator. For example, 50 percent would be written as 0.5 and 2 percent would be written as 0.02.

Let's look at an example. Suppose you earn $6,000 per month and you are allowed to contribute a portion of your gross monthly pay to a retirement plan. You want to determine the amount of your pay that will go into the plan if you contribute 5 percent, 8 percent, or 10 percent of your gross wages. To make this determination you write a program like the one shown in Code Listing 2-18.

Code Listing 2-18 (Contribution.java)

```
1 // This program calculates the amount of pay that
2 // will be contributed to a retirement plan if 5%,
3 // 8%, or 10% of monthly pay is withheld.
4
5 public class Contribution
6 {
7    public static void main(String[] args)
8    {
9       // Variables to hold the monthly pay and
10      // the amount of contribution.
11      double monthlyPay = 6000.0;
12      double contribution;
13
14      // Calculate and display a 5% contribution.
15      contribution = monthlyPay * 0.05;
16      System.out.println("5 percent is $" +
17                         contribution +
18                         " per month.");
19
20      // Calculate and display an 8% contribution.
21      contribution = monthlyPay * 0.08;
22      System.out.println("8 percent is $" +
23                         contribution +
24                         " per month.");
25
26      // Calculate and display a 10% contribution.
27      contribution = monthlyPay * 0.1;
28      System.out.println("10 percent is $" +
29                         contribution +
30                         " per month.");
31    }
32 }
```

Program Output

```
5 percent is $300.0 per month.
8 percent is $480.0 per month.
10 percent is $600.0 per month.
```

Lines 11 and 12 declare two variables: `monthlyPay` and `contribution`. The `monthlyPay` variable, which is initialized with the value `6000.0`, holds the amount of your monthly pay. The `contribution` variable will hold the amount of a contribution to the retirement plan.

The statements in lines 15 through 18 calculate and display 5 percent of the monthly pay. The calculation is done in line 15, where the `monthlyPay` variable is multiplied by 0.05. The result is assigned to the `contribution` variable, which is then displayed by the statement in lines 16 through 18.

Similar steps are taken in lines 21 through 24, which calculate and display 8 percent of the monthly pay, and lines 27 through 30, which calculate and display 10 percent of the monthly pay.

Calculating a Percentage Discount

Another common calculation is determining a percentage discount. For example, suppose a retail business sells an item that is regularly priced at $59, and is planning to have a sale where the item's price will be reduced by 20 percent. You have been asked to write a program to calculate the sale price of the item.

To determine the sale price you perform two calculations:

- First, you get the amount of the discount, which is 20 percent of the item's regular price.
- Second, you subtract the discount amount from the item's regular price. This gives you the sale price.

Code Listing 2-19 shows how this is done in Java.

Code Listing 2-19 (Discount.java)

```
1 // This program calculates the sale price of an
2 // item that is regularly priced at $59, with
3 // a 20 percent discount subtracted.
4
5 public class Discount
6 {
7     public static void main(String[] args)
8     {
9        // Variables to hold the regular price, the
10       // amount of a discount, and the sale price.
11       double regularPrice = 59.0;
12       double discount;
13       double salePrice;
14
15       // Calculate the amount of a 20% discount.
16       discount = regularPrice * 0.2;
17
18       // Calculate the sale price by subtracting
19       // the discount from the regular price.
20       salePrice = regularPrice - discount;
21
22       // Display the results.
23       System.out.println("Regular price: $" + regularPrice);
24       System.out.println("Discount amount $" + discount);
25       System.out.println("Sale price: $" + salePrice);
26    }
27 }
```

Program Output

```
Regular price: $59.0
Discount amount $11.8
Sale price: $47.2
```

Lines 11 through 13 declare three variables. The `regularPrice` variable holds the item's regular price, and is initialized with the value `59.0`. The `discount` variable will hold the amount of the discount once it is calculated. The `salePrice` variable will hold the item's sale price.

Line 16 calculates the amount of the 20 percent discount by multiplying `regularPrice` by 0.2. The result is stored in the `discount` variable. Line 20 calculates the sale price by subtracting `discount` from `regularPrice`. The result is stored in the `salePrice` variable. The statements in lines 23 through 25 display the item's regular price, the amount of the discount, and the sale price.

The `Math` Class

The Java API provides a class named `Math`, which contains numerous methods that are useful for performing complex mathematical operations. In this section we will briefly look at the `Math.pow` and `Math.sqrt` methods.

The `Math.pow` Method

In Java, raising a number to a power requires the `Math.pow` method. Here is an example of how the `Math.pow` method is used:

```
result = Math.pow(4.0, 2.0);
```

The method takes two `double` arguments. It raises the first argument to the power of the second argument, and returns the result as a `double`. In this example, 4.0 is raised to the power of 2.0. This statement is equivalent to the following algebraic statement:

$result = 4^2$

Here is another example of a statement using the `Math.pow` method. It assigns 3 times 6^3 to `x`:

```
x = 3 * Math.pow(6.0, 3.0);
```

And the following statement displays the value of 5 raised to the power of 4:

```
System.out.println(Math.pow(5.0, 4.0));
```

The `Math.sqrt` Method

The `Math.sqrt` method accepts a `double` value as its argument and returns the square root of the value. Here is an example of how the method is used:

```
result = Math.sqrt(9.0);
```

In this example the value 9.0 is passed as an argument to the `Math.sqrt` method. The method will return the square root of 9.0, which is assigned to the `result` variable. The following statement shows another example. In this statement the square root of 25.0 (which is 5.0) is displayed on the screen:

```
System.out.println(Math.sqrt(25.0));
```

For more information about the `Math` class, see Appendix G, available on the book's companion Web site at www.pearsonglobaleditions.com/Gaddis.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

2.22 Complete the following table by writing the value of each expression in the Value column.

Expression	Value
`6 + 3 * 5`	_____
`12 / 2 - 4`	_____
`9 + 14 * 2 - 6`	_____
`5 + 19 % 3 - 1`	_____
`(6 + 2) * 3`	_____
`14 / (11 - 4)`	_____
`9 + 12 * (8 - 3)`	_____

2.23 Is the division statement in the following code an example of integer division or floating-point division? What value will be stored in `portion`?

```
double portion;
portion = 70 / 3;
```

2.6 Combined Assignment Operators

CONCEPT: The combined assignment operators combine the assignment operator with the arithmetic operators.

Quite often, programs have assignment statements of the following form:

```
x = x + 1;
```

On the right side of the assignment operator, 1 is added to `x`. The result is then assigned to `x`, replacing the value that was previously there. Effectively, this statement adds 1 to x. Here is another example:

```
balance = balance + deposit;
```

Assuming that `balance` and `deposit` are variables, this statement assigns the value of `balance + deposit` to `balance`. The effect of this statement is that `deposit` is added to the value stored in `balance`. Here is another example:

```
balance = balance - withdrawal;
```

Assuming that `balance` and `withdrawal` are variables, this statement assigns the value of `balance - withdrawal` to `balance`. The effect of this statement is that withdrawal is subtracted from the value stored in `balance`.

If you have not seen these types of statements before, they might cause some initial confusion because the same variable name appears on both sides of the assignment operator. Table 2-12 shows other examples of statements written this way.

Table 2-12 Various assignment statements (assume x = 6 in each statement)

Statement	What It Does	Value of x after the Statement
`x = x + 4;`	Adds 4 to x	10
`x = x - 3;`	Subtracts 3 from x	3
`x = x * 10;`	Multiplies x by 10	60
`x = x / 2;`	Divides x by 2	3
`x = x % 4`	Assigns the remainder of x / 4 to x.	2

These types of operations are common in programming. For convenience, Java offers a special set of operators designed specifically for these jobs. Table 2-13 shows the *combined assignment operators*, also known as *compound operators*.

Table 2-13 Combined assignment operators

Operator	Example Usage	Equivalent To
`+=`	`x += 5;`	`x = x + 5;`
`-=`	`y -= 2;`	`y = y - 2;`
`*=`	`z *= 10;`	`z = z * 10;`
`/=`	`a /= b;`	`a = a / b;`
`%=`	`c %= 3;`	`c = c % 3;`

As you can see, the combined assignment operators do not require the programmer to type the variable name twice. The following statement:

```
balance = balance + deposit;
```

could be rewritten as

```
balance += deposit;
```

Similarly, the statement

```
balance = balance - withdrawal;
```

could be rewritten as

```
balance -= withdrawal;
```

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

2.24 Write statements using combined assignment operators to perform the following:
   a) Add 6 to x
   b) Subtract 4 from `amount`

c) Multiply y by 4
d) Divide `total` by 27
e) Store in x the remainder of x divided by 7

2.7 Conversion between Primitive Data Types

CONCEPT: **Before a value can be stored in a variable, the value's data type must be compatible with the variable's data type. Java performs some conversions between data types automatically, but does not automatically perform any conversion that can result in the loss of data. Java also follows a set of rules when evaluating arithmetic expressions containing mixed data types.**

Java is a *strongly typed* language. This means that before a value is assigned to a variable, Java checks the data types of the variable and the value being assigned to it to determine whether they are compatible. For example, look at the following statements:

```
int x;
double y = 2.5;
x = y;
```

The assignment statement is attempting to store a `double` value (2.5) in an `int` variable. When the Java compiler encounters this line of code, it will respond with an error message. (The JDK displays the message "possible loss of precision.")

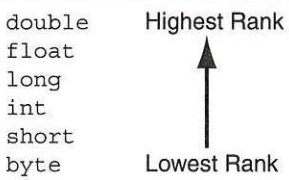
Not all assignment statements that mix data types are rejected by the compiler, however. For instance, look at the following program segment:

```
int x;
short y = 2;
x = y;
```

This assignment statement, which stores a `short` in an `int`, will work with no problems. So why does Java permit a `short` to be stored in an `int`, but does not permit a `double` to be stored in an `int`? The obvious reason is that a `double` can store fractional numbers and can hold values much larger than an `int` can hold. If Java were to permit a `double` to be assigned to an `int`, a loss of data would be likely.

Just like officers in the military, the primitive data types are ranked. One data type outranks another if it can hold a larger number. For example, a `float` outranks an `int`, and an `int` outranks a `short`. Figure 2-6 shows the numeric data types in order of their rank. The higher a data type appears in the list, the higher is its rank.

Figure 2-6 Primitive data type ranking



In assignment statements where values of lower-ranked data types are stored in variables of higher-ranked data types, Java automatically converts the lower-ranked value to the higher-ranked type. This is called a *widening conversion*. For example, the following code demonstrates a widening conversion, which takes place when an `int` value is stored in a `double` variable:

```
double x;
int y = 10;
x = y;          // Performs a widening conversion
```

A *narrowing conversion* is the conversion of a value to a lower-ranked type. For example, converting a `double` to an `int` would be a narrowing conversion. Because narrowing conversions can potentially cause a loss of data, Java does not automatically perform them.

Cast Operators

The *cast operator* lets you manually convert a value, even if it means that a narrowing conversion will take place. Cast operators are unary operators that appear as a data type name enclosed in a set of parentheses. The operator precedes the value being converted. Here is an example:

```
x = (int)number;
```

The cast operator in this statement is the word `int` inside the parentheses. It returns the value in `number`, converted to an `int`. This converted value is then stored in `x`. If number were a floating-point variable, such as a `float` or a `double`, the value that is returned would be *truncated*, which means the fractional part of the number is lost. The original value in the number variable is not changed, however.

Table 2-14 shows several statements using cast operators.

Table 2-14 Example uses of cast operators

Statement	Description
`littleNum = (short)bigNum;`	The cast operator returns the value in `bigNum`, converted to a `short`. The converted value is assigned to the variable `littleNum`.
`x = (long)3.7;`	The cast operator is applied to the expression 3.7. The operator returns the value 3, which is assigned to the variable `x`.
`number = (int)72.567;`	The cast operator is applied to the expression 72.567. The operator returns 72, which is used to initialize the variable `number`.
`value = (float)x;`	The cast operator returns the value in `x`, converted to a `float`. The converted value is assigned to the variable `value`.
`value = (byte)number;`	The cast operator returns the value in `number`, converted to a `byte`. The converted value is assigned to the variable `value`.

Note that when a cast operator is applied to a variable, it does not change the contents of the variable. It only returns the value stored in the variable, converted to the specified data type.

Recall from our earlier discussion that when both operands of a division are integers, the operation will result in integer division. This means that the result of the division will be

an integer, with any fractional part of the result thrown away. For example, look at the following code:

```
int pies = 10, people = 4;
double piesPerPerson;
piesPerPerson = pies / people;
```

Although 10 divided by 4 is 2.5, this code will store 2 in the `piesPerPerson` variable. Because both `pies` and `people` are `int` variables, the result will be an `int`, and the fractional part will be thrown away. We can modify the code with a cast operator, however, so it gives the correct result as a floating-point value:

```
piesPerPerson = (double)pies / people;
```

The variable `pies` is an `int` and holds the value 10. The expression `(double)pies` returns the value in `pies` converted to a `double`. This means that one of the division operator's operands is a `double`, so the result of the division will be a `double`. The statement could also have been written as follows:

```
piesPerPerson = pies / (double)people;
```

In this statement, the cast operator returns the value of the `people` variable converted to a `double`. In either statement, the result of the division is a `double`.

WARNING! The cast operator can be applied to an entire expression enclosed in parentheses. For example, look at the following statement:

```
piesPerPerson = (double)(pies / people);
```

This statement does not convert the value in `pies` or `people` to a `double`, but converts the result of the expression `pies / people`. If this statement were used, an integer division operation would still have been performed. Here's why: The result of the expression `pies / people` is 2 (because integer division takes place). The value 2 converted to a `double` is 2.0. To prevent the integer division from taking place, one of the operands must be converted to a `double`.

Mixed Integer Operations

One of the nuances of the Java language is the way it internally handles arithmetic operations on `int`, `byte`, and `short` variables. When values of the `byte` or `short` data types are used in arithmetic expressions, they are temporarily converted to `int` values. The result of an arithmetic operation using only a mixture of `byte`, `short`, or `int` values will always be an `int`.

For example, assume that `b` and `c` in the following expression are `short` variables:

```
b + c
```

Although both `b` and `c` are `short` variables, the result of the expression `b + c` is an `int`. This means that when the result of such an expression is stored in a variable, the variable must be an `int` or higher data type. For example, look at the following code:

```
short firstNumber = 10,
      secondNumber = 20,
      thirdNumber;
```

```
// The following statement causes an error!
thirdNumber = firstNumber + secondNumber;
```

When this code is compiled, the following statement causes an error:

```
thirdNumber = firstNumber + secondNumber;
```

The error results from the fact that `thirdNumber` is a `short`. Although `firstNumber` and `secondNumber` are also `short` variables, the expression `firstNumber + secondNumber` results in an `int` value. The program can be corrected if `thirdNumber` is declared as an `int`, or if a cast operator is used in the assignment statement, as shown here:

```
thirdNumber = (short)(firstNumber + secondNumber);
```

Other Mixed Mathematical Expressions

In situations where a mathematical expression has one or more values of the `double`, `float`, or `long` data types, Java strives to convert all of the operands in the expression to the same data type. Let's look at the specific rules that govern evaluation of these types of expressions.

1. If one of an operator's operands is a `double`, the value of the other operand will be converted to a `double`. The result of the expression will be a `double`. For example, in the following statement assume that `b` is a `double` and `c` is an `int`:

   ```
   a = b + c;
   ```

   The value in `c` will be converted to a `double` prior to the addition. The result of the addition will be a `double`, so the variable `a` must also be a `double`.
2. If one of an operator's operands is a `float`, the value of the other operand will be converted to a `float`. The result of the expression will be a `float`. For example, in the following statement assume that `x` is a `short` and `y` is a `float`:

   ```
   z = x * y;
   ```

   The value in `x` will be converted to a `float` prior to the multiplication. The result of the multiplication will be a `float`, so the variable `z` must also be either a `double` or a `float`.
3. If one of an operator's operands is a `long`, the value of the other operand will be converted to a `long`. The result of the expression will be a `long`. For example, in the following statement assume that `a` is a `long` and `b` is a `short`:

   ```
   c = a - b;
   ```

   The variable `b` will be converted to a `long` prior to the subtraction. The result of the subtraction will be a `long`, so the variable `c` must also be a `long`, `float`, or `double`.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

2.25 The following declaration appears in a program:

```
short totalPay, basePay = 500, bonus = 1000;
```

The following statement appears in the same program:

```
totalPay = basePay + bonus;
```

a) Will the statement compile properly or cause an error?
b) If the statement causes an error, why? How can you fix it?

2.26 The variable a is a float and the variable b is a double. Write a statement that will assign the value of b to a without causing an error when the program is compiled.

2.8 Creating Named Constants with final

CONCEPT: The final key word can be used in a variable declaration to make the variable a named constant. Named constants are initialized with a value, and that value cannot change during the execution of the program.

Assume that the following statement appears in a banking program that calculates data pertaining to loans:

```
amount = balance * 0.069;
```

In such a program, two potential problems arise. First, it is not clear to anyone other than the original programmer what 0.069 is. It appears to be an interest rate, but in some situations there are fees associated with loan payments. How can the purpose of this statement be determined without painstakingly checking the rest of the program?

The second problem occurs if this number is used in other calculations throughout the program and must be changed periodically. Assuming the number is an interest rate, what if the rate changes from 6.9 percent to 8.2 percent? The programmer would have to search through the source code for every occurrence of the number.

Both of these problems can be addressed by using named constants. A *named constant* is a variable whose value is read only and cannot be changed during the program's execution. You can create such a variable in Java by using the final key word in the variable declaration. The word final is written just before the data type. Here is an example:

```
final double INTEREST_RATE = 0.069;
```

This statement looks just like a regular variable declaration except that the word final appears before the data type, and the variable name is written in all uppercase letters. It is not required that the variable name appear in all uppercase letters, but many programmers prefer to write them this way so they are easily distinguishable from regular variable names.

An initialization value must be given when declaring a variable with the final modifier, or an error will result when the program is compiled. A compiler error will also result if there are any statements in the program that attempt to change the value of a final variable.

An advantage of using named constants is that they make programs more self-documenting. The following statement:

```
amount = balance * 0.069;
```

can be changed to read

```
amount = balance * INTEREST_RATE;
```

A new programmer can read the second statement and know what is happening. It is evident that balance is being multiplied by the interest rate. Another advantage to this approach is that widespread changes can easily be made to the program. Let's say the interest rate appears in a dozen different statements throughout the program. When the rate changes, the initialization

value in the definition of the named constant is the only value that needs to be modified. If the rate increases to 8.2 percent, the declaration can be changed to the following:

```
final double INTEREST_RATE = 0.082;
```

The program is then ready to be recompiled. Every statement that uses `INTEREST_RATE` will use the new value.

The `Math.PI` Named Constant

The `Math` class, which is part of the Java API, provides a predefined named constant, `Math.PI`. This constant is assigned the value 3.14159265358979323846, which is an approximation of the mathematical value pi. For example, look at the following statement:

```
area = Math.PI * radius * radius;
```

Assuming the `radius` variable holds the radius of a circle, this statement uses the `Math.PI` constant to calculate the area of the circle.

> For more information about the `Math` class, see Appendix F, available on the book's companion Web site at www.pearsonglobaleditions.com/Gaddis.

2.9 The `String` Class

CONCEPT: **The `String` class allows you to create objects for holding strings. It also has various methods that allow you to work with strings.**

You have already encountered strings and examined programs that display them on the screen, but let's take a moment to make sure you understand what a string is. A string is a sequence of characters. It can be used to represent any type of data that contains text, such as names, addresses, warning messages, and so forth. String literals are enclosed in double-quotation marks, such as the following:

```
"Hello World"
"Joe Mahoney"
```

Although programs commonly encounter strings and must perform a variety of tasks with them, Java does not have a primitive data type for storing them in memory. Instead, the Java API provides a class for handling strings. You use this class to create objects that are capable of storing strings and performing operations on them. Before discussing this class, let's briefly discuss how classes and objects are related.

Objects Are Created from Classes

Chapter 1 introduced you to objects as software entities that can contain attributes and methods. An object's attributes are data values that are stored in the object. An object's methods are procedures that perform operations on the object's attributes. Before an object can be created, however, it must be designed by a programmer. The programmer determines the attributes and methods that are necessary, and then creates a class that describes the object.

You have already seen classes used as containers for applications. A class can also be used to specify the attributes and methods that a particular type of object may have. Think of a class

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

TIP: Don't worry if these concepts seem a little fuzzy to you. As you progress through this book, the concepts of classes and objects will be reinforced again and again.

The String Class

The class that is provided by the Java API for handling strings is named `String`. The first step in using the `String` class is to declare a variable of the `String` class data type. Here is an example of a `String` variable declaration:

```
String name;
```

TIP: The `S` in `String` is written in an uppercase letter. By convention, the first character of a class name is always written in an uppercase letter.

This statement declares `name` as a `String` variable. Remember that `String` is a class, not a primitive data type. Let's briefly look at the difference between primitive type variables and class type variables.

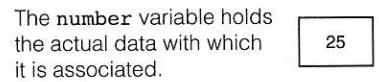
Primitive Type Variables and Class Type Variables

A variable of any type can be associated with an item of data. *Primitive type variables* hold the actual data items with which they are associated. For example, assume that `number` is an `int` variable. The following statement stores the value 25 in the variable:

```
number = 25;
```

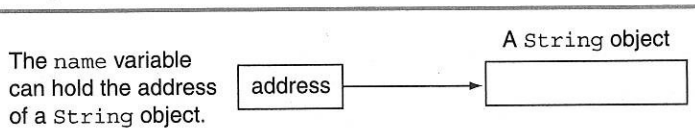
This is illustrated in Figure 2-7.

Figure 2-7 A primitive type variable holds the data with which it is associated



A *class type variable* does not hold the actual data item that it is associated with, but holds the memory address of the data item it is associated with. If `name` is a `String` class variable, then `name` can hold the memory address of a `String` object. This is illustrated in Figure 2-8.

Figure 2-8 A `String` class variable can hold the address of a `String` object



When a class type variable holds the address of an object, it is said that the variable references the object. For this reason, class type variables are commonly known as *reference variables*.

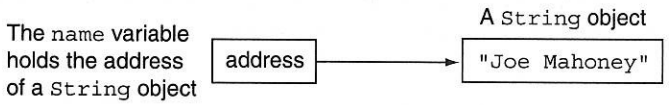
Creating a `String` Object

Any time you write a string literal in your program, Java will create a `String` object in memory to hold it. You can create a `String` object in memory and store its address in a `String` variable with a simple assignment statement. Here is an example:

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

Here, the string literal causes a `String` object to be created in memory with the value "Joe Mahoney" stored in it. Then the assignment operator stores the address of that object in the `name` variable. After this statement executes, it is said that the `name` variable references a `String` object. This is illustrated in Figure 2-9.

Figure 2-9 The `name` variable holds the address of a `String` object



You can also use the = operator to initialize a `String` variable, as shown here:

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

This statement declares `name` as a `String` variable, creates a `String` object with the value "Joe Mahoney" stored in it, and assigns the object's memory address to the `name` variable. Code Listing 2-20 shows `String` variables being declared, initialized, and then used in a `println` statement.

Code Listing 2-20 (`StringDemo.java`)

```
1  // A simple program demonstrating String objects.
2
3  public class StringDemo
4  {
5     public static void main(String[] args)
6     {
7        String greeting = "Good morning, ";
8        String name = "Herman";
9
10       System.out.println(greeting + name);
11    }
12 }
```

Program Output

```
Good morning, Herman
```

Because the String type is a class instead of a primitive data type, it provides numerous methods for working with strings. For example, the String class has a method named length that returns the length of the string stored in an object. Assuming the name variable references a String object, the following statement stores the length of its string in the variable stringSize (assume that stringSize is an int variable):

```
stringSize = name.length();
```

This statement calls the length method of the object that name refers to. To *call* a method means to execute it. The general form of a method call is as follows:

```
referenceVariable.method(arguments. . .)
```

referenceVariable is the name of a variable that references an object, *method* is the name of a method, and *arguments. . .* is zero or more arguments that are passed to the method. If no arguments are passed to the method, as is the case with the length method, a set of empty parentheses must follow the name of the method.

The String class's length method *returns* an int value. This means that the method sends an int value back to the statement that called it. This value can be stored in a variable, displayed on the screen, or used in calculations. Code Listing 2-21 demonstrates the length method.

Code Listing 2-21 (StringLength.java)

```
1  // This program demonstrates the String class's length method.
2
3  public class StringLength
4  {
5     public static void main(String[] args)
6     {
7        String name = "Herman";
8        int stringSize;
9
10       stringSize = name.length();
11       System.out.println(name + " has " + stringSize +
12                          " characters.");
13    }
14 }
```

Program Output

```
Herman has 6 characters.
```

NOTE: The String class's length method returns the number of characters in the string, including spaces.

You will study the String class methods in detail in Chapter 9, but let's look at a few more examples now. In addition to length, Table 2-15 describes the charAt, toLowerCase, and toUpperCase methods.

Table 2-15 A few `String` class methods

Method	Description and Example

`charAt(index)`	The argument *index* is an `int` value and specifies a character position in the string. The first character is at position 0, the second character is at position 1, and so forth. The method returns the character at the specified position. The return value is of the type `char`.
	Example:
	`char letter;`
	`String name = "Herman";`
	`letter = name.charAt(3);`
	After this code executes, the variable `letter` will hold the character 'm'.

`length()`	This method returns the number of characters in the string. The return value is of the type `int`.
	Example:
	`int stringSize;`
	`String name = "Herman";`
	`stringSize = name.length();`
	After this code executes, the `stringSize` variable will hold the value 6.

`toLowerCase()`	This method returns a new string that is the lowercase equivalent of the string contained in the calling object.
	Example:
	`String bigName = "HERMAN";`
	`String littleName = bigName.toLowerCase();`
	After this code executes, the object referenced by `littleName` will hold the string "herman".

`toUpperCase()`	This method returns a new string that is the uppercase equivalent of the string contained in the calling object.
	Example:
	`String littleName = "herman";`
	`String bigName = littleName.toUpperCase();`
	After this code executes, the object referenced by `bigName` will hold the string "HERMAN".

The program in Code Listing 2-22 demonstrates these methods.

Code Listing 2-22 (`StringMethods.java`)

```
1  // This program demonstrates a few of the String methods.
2
3  public class StringMethods
4  {
5     public static void main(String[] args)
6     {
7        String message = "Java is Great Fun!";
8        String upper = message.toUpperCase();
```

```
9         String lower = message.toLowerCase();
10        char letter = message.charAt(2);
11        int stringSize = message.length();
12
13        System.out.println(message);
14        System.out.println(upper);
15        System.out.println(lower);
16        System.out.println(letter);
17        System.out.println(stringSize);
18     }
19  }
```

Program Output

```
Java is Great Fun!
JAVA IS GREAT FUN!
java is great fun!
v
18
```

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

2.27 Write a statement that declares a `String` variable named `city`. The variable should be initialized so it references an object with the string "San Francisco".

2.28 Assume that `stringLength` is an `int` variable. Write a statement that stores the length of the string referenced by the `city` variable (declared in Checkpoint 2.27) in `stringLength`.

2.29 Assume that `oneChar` is a `char` variable. Write a statement that stores the first character in the string referenced by the `city` variable (declared in Checkpoint 2.27) in `oneChar`.

2.30 Assume that `upperCity` is a `String` reference variable. Write a statement that stores the uppercase equivalent of the string referenced by the `city` variable (declared in Checkpoint 2.27) in `upperCity`.

2.31 Assume that `lowerCity` is a `String` reference variable. Write a statement that stores the lowercase equivalent of the string referenced by the `city` variable (declared in Checkpoint 2.27) in `lowerCity`.

2.10 Scope

CONCEPT: A variable's scope is the part of the program that has access to the variable.

Every variable has a *scope*. The scope of a variable is the part of the program where the variable may be accessed by its name. A variable is visible only to statements inside the variable's scope. The rules that define a variable's scope are complex, and you are only

introduced to the concept here. In other chapters of the book we revisit this topic and expand on it.

So far, you have only seen variables declared inside the `main` method. Variables that are declared inside a method are called *local variables*. Later you will learn about variables that are declared outside a method, but for now, let's focus on the use of local variables.

A local variable's scope begins at the variable's declaration and ends at the end of the method in which the variable is declared. The variable cannot be accessed by statements that are outside this region. This means that a local variable cannot be accessed by code that is outside the method, or inside the method but before the variable's declaration. The program in Code Listing 2-23 shows an example.

Code Listing 2-23 (Scope.java)

```
1  // This program can't find its variable.
2
3  public class Scope
4  {
5     public static void main(String[] args)
6     {
7        System.out.println(value); // ERROR!
8        int value = 100;
9     }
10 }
```

The program does not compile because it attempts to send the contents of the variable `value` to `println` before the variable is declared. It is important to remember that the compiler reads your program from top to bottom. If it encounters a statement that uses a variable before the variable is declared, an error will result. To correct the program, the variable declaration must be written before any statement that uses it.

NOTE: If you compile this program, the compiler will display an error message such as "cannot resolve symbol." This means that the compiler has encountered a name for which it cannot determine a meaning.

Another rule that you must remember about local variables is that you cannot have two local variables with the same name in the same scope. For example, look at the following method.

```
public static void main(String[] args)
{
   // Declare a variable named number and
   // display its value.
   int number = 7;
   System.out.println(number);
```

```
        // Declare another variable named number and
        // display its value.
        int number = 100;                  // ERROR!!!
        System.out.println(number);        // ERROR!!!
    }
```

This method declares a variable named `number` and initializes it with the value 7. The variable's scope begins at the declaration statement and extends to the end of the method. Inside the variable's scope a statement appears that declares another variable named `number`. This statement will cause an error because you cannot have two local variables with the same name in the same scope.

2.11 Comments

CONCEPT: **Comments are notes of explanation that document lines or sections of a program. Comments are part of the program, but the compiler ignores them. They are intended for people who may be reading the source code.**

Comments are short notes that are placed in different parts of a program, explaining how those parts of the program work. Comments are not intended for the compiler. They are intended for programmers to read, to help them understand the code. The compiler skips all of the comments that appear in a program.

As a beginning programmer, you might resist the idea of writing a lot of comments in your programs. After all, it's a lot more fun to write code that actually does something! However, it's crucial that you take the extra time to write comments. They will almost certainly save you time in the future when you have to modify or debug the program. Even large and complex programs can be made easy to read and understand if they are properly commented.

In Java there are three types of comments: single-line comments, multiline comments, and documentation comments. Let's briefly discuss each type.

Single-Line Comments

You have already seen the first way to write comments in a Java program. You simply place two forward slashes (`//`) where you want the comment to begin. The compiler ignores everything from that point to the end of the line. Code Listing 2-24 shows that comments may be placed liberally throughout a program.

Code Listing 2-24 (`Comment1.java`)

```
1  // PROGRAM: Comment1.java
2  // Written by Herbert Dorfmann
3  // This program calculates company payroll
4
5  public class Comment1
```

```
{
   public static void main(String[] args)
   {
      double payRate;       // Holds the hourly pay rate
      double hours;         // Holds the hours worked
      int employeeNumber;   // Holds the employee number

      // The Remainder of This Program is Omitted.
   }
}
```

In addition to telling who wrote the program and describing the purpose of variables, comments can also be used to explain complex procedures in your code.

Multi-Line Comments

The second type of comment in Java is the multi-line comment. *Multi-line comments* start with `/*` (a forward slash followed by an asterisk) and end with `*/` (an asterisk followed by a forward slash). Everything between these markers is ignored. Code Listing 2-25 illustrates how multi-line comments may be used.

Code Listing 2-25 (Comment2.java)

```
/*
   PROGRAM: Comment2.java
   Written by Herbert Dorfmann
   This program calculates company payroll
*/

public class Comment2
{
   public static void main(String[] args)
   {
      double payRate;       // Holds the hourly pay rate
      double hours;         // Holds the hours worked
      int employeeNumber;   // Holds the employee number

      // The Remainder of This Program is Omitted.
   }
}
```

Unlike a comment started with `//`, a multi-line comment can span several lines. This makes it more convenient to write large blocks of comments because you do not have to

mark every line. Consequently, the multi-line comment is inconvenient for writing single-line comments because you must type both a beginning and an ending comment symbol.

Remember the following advice when using multi-line comments:

- Be careful not to reverse the beginning symbol with the ending symbol.
- Be sure not to forget the ending symbol.

Many programmers use asterisks or other characters to draw borders or boxes around their comments. This helps to visually separate the comments from surrounding code. These are called block comments. Table 2-16 shows four examples of block comments.

Table 2-16 Block comments

```
/*
 *   This program demonstrates the
 *   way to write comments.
 */
```

```
//**********************************
//    This program demonstrates the *
//    way to write comments.        *
//**********************************
```

```
////////////////////////////////////
//    This program demonstrates the
//    way to write comments.
////////////////////////////////////
```

```
//-----------------------------------
//    This program demonstrates the
//    way to write comments.
//-----------------------------------
```

Documentation Comments

The third type of comment is known as a documentation comment. *Documentation comments* can be read and processed by a program named `javadoc`, which comes with the JDK. The purpose of the `javadoc` program is to read Java source code files and generate attractively formatted HTML files that document the source code. If the source code files contain any documentation comments, the information in the comments becomes part of the HTML documentation. The HTML documentation files may be viewed in a Web browser.

Any comment that starts with `/**` and ends with `*/` is considered a documentation comment. Normally you write a documentation comment just before a class header, giving a brief description of the class. You also write a documentation comment just before each method header, giving a brief description of the method. For example, Code Listing 2-26 shows a program with documentation comments. This program has a documentation comment just before the class header, and just before the `main` method header.

Code Listing 2-26 (Comment3.java)

```
1  /**
2    This class creates a program that calculates company payroll.
3  */
4
5  public class Comment3
6  {
7     /**
8        The main method is the program's starting point.
9     */
10
11    public static void main(String[] args)
12    {
13       double payRate;        // Holds the hourly pay rate
14       double hours;          // Holds the hours worked
15       int employeeNumber;    // Holds the employee number
16
17       // The Remainder of This Program is Omitted.
18    }
19 }
```

You run the `javadoc` program from the operating system command prompt. Here is the general format of the `javadoc` command:

```
javadoc SourceFile.java
```

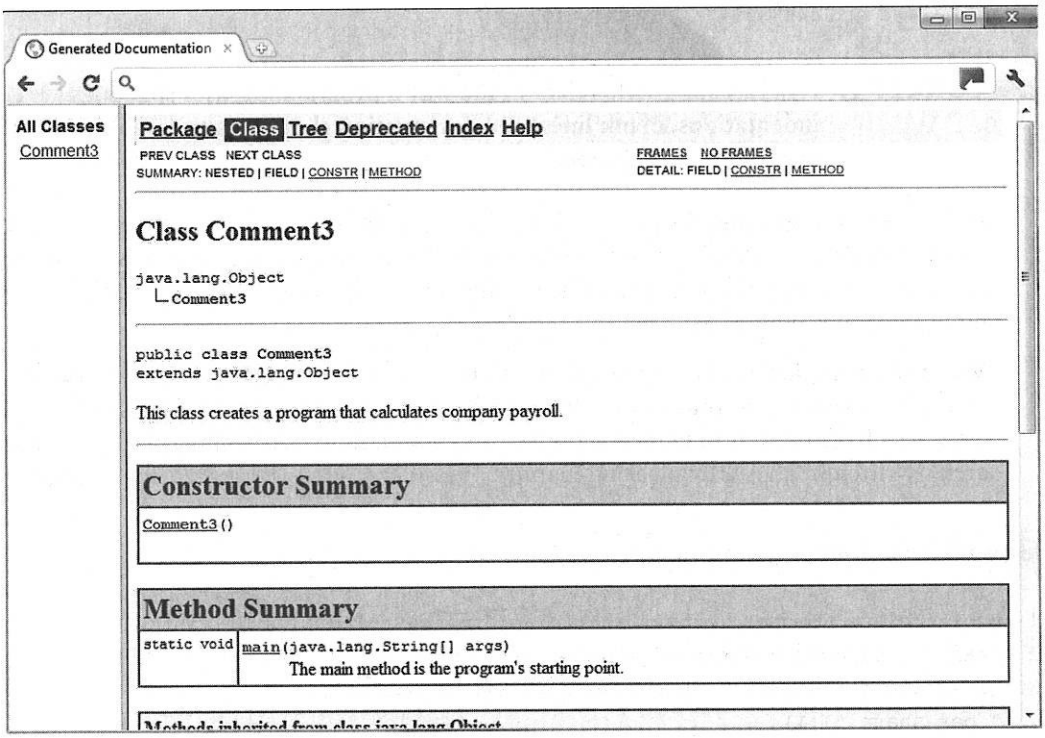
`SourceFile``.java` is the name of a Java source code file, including the `.java` extension. The file will be read by `javadoc` and documentation will be produced for it. For example, the following command will produce documentation for the *Comment3.java* source code file, which is shown in Code Listing 2-26:

```
javadoc Comment3.java
```

After this command executes, several documentation files will be created in the same directory as the source code file. One of these files will be named *index.html*. Figure 2-10 shows the *index.html* file being viewed in a Web browser. Notice that the text that was written in the documentation comments appears in the file.

TIP: When you write a documentation comment for a method, the HTML documentation file that is produced by `javadoc` will have two sections for the method: a summary section and a detail section. The first sentence in the method's documentation comment is used as the summary of the method. Note that `javadoc` considers the end of the sentence as a period followed by a whitespace character. For this reason, when a method description contains more than one sentence, you should always end the first sentence with a period followed by a whitespace character. The method's detail section will contain all of the description that appears in the documentation comment.

Figure 2-10 Documentation generated by `javadoc` (Google Inc.)



If you look at the JDK documentation, which are HTML files that you view in a Web browser, you will see that they are formatted in the same way as the files generated by `javadoc`. A benefit of using `javadoc` to document your source code is that your documentation will have the same professional look and feel as the standard Java documentation.

From this point forward in the book, we will use documentation comments in the example source code. As we progress through various topics, you will see additional uses of documentation comments and the `javadoc` program.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

2.32 How do you write a single line comment? How do you write a multi-line comment? How do you write a documentation comment?

2.33 How are documentation comments different from other types of comments?

2.12 Programming Style

CONCEPT: **Programming style refers to the way a programmer uses spaces, indentations, blank lines, and punctuation characters to visually arrange a program's source code.**

In Chapter 1, you learned that syntax rules govern the way a language may be used. The syntax rules of Java dictate how and where to place key words, semicolons, commas, braces, and other elements of the language. The compiler checks for syntax errors, and if there are none, generates byte code.

When the compiler reads a program it processes it as one long stream of characters. The compiler doesn't care that each statement is on a separate line, or that spaces separate operators from operands. Humans, on the other hand, find it difficult to read programs that aren't written in a visually pleasing manner. Consider Code Listing 2-27 for example.

Code Listing 2-27 (Compact.java)

```
1  public class Compact {public static void main(String [] args){int
2  shares=220; double averagePrice=14.67; System.out.println(
3  "There were "+shares+" shares sold at $"+averagePrice+
4  " per share.");}}
```

Program Output

```
There were 220 shares sold at $14.67 per share.
```

Although the program is syntactically correct (it doesn't violate any rules of Java), it is very difficult to read. The same program is shown in Code Listing 2-28, written in a more understandable style.

Code Listing 2-28 (Readable.java)

```
1  /**
2    This example is much more readable than Compact.java.
3  */
4
5  public class Readable
6  {
7     public static void main(String[] args)
8     {
9        int shares = 220;
10       double averagePrice = 14.67;
11
12       System.out.println("There were " + shares +
13                          " shares sold at $" +
14                          averagePrice + " per share.");
```

```
15     }
16  }
```

Program Output

```
There were 220 shares sold at $14.67 per share.
```

The term *programming style* usually refers to the way source code is visually arranged. It includes techniques for consistently putting spaces and indentations in a program so visual cues are created. These cues quickly tell a programmer important information about a program.

For example, notice in Code Listing 2-28 that inside the class's braces each line is indented, and inside the `main` method's braces each line is indented again. It is a common programming style to indent all the lines inside a set of braces, as shown in Figure 2-11.

Figure 2-11 Indentation

```
/**
    This example is much more readable than Compact.java.
*/

public class Readable
{
INDENT public static void main(String[] args)
INDENT {
INDENT INDENT int shares = 220;
INDENT INDENT double averagePrice = 14.67;

INDENT INDENT System.out.println("There were " + shares +
INDENT INDENT  Extra spaces inserted here " shares sold at $" +
INDENT INDENT  Extra spaces inserted here averagePrice + " per share.");
INDENT }
}
```

Another aspect of programming style is how to handle statements that are too long to fit on one line. Notice that the `println` statement is spread out over three lines. Extra spaces are inserted at the beginning of the statement's second and third lines, which indicate that they are continuations.

When declaring multiple variables of the same type with a single statement, it is a common practice to write each variable name on a separate line with a comment explaining the variable's purpose. Here is an example:

```
int fahrenheit,    // To hold the Fahrenheit temperature
    celsius,       // To hold the Celsius temperature
    kelvin;        // To hold the Kelvin temperature
```

You may have noticed in the example programs that a blank line is inserted between the variable declarations and the statements that follow them. This is intended to separate the declarations visually from the executable statements.

There are many other issues related to programming style. They will be presented throughout the book.

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

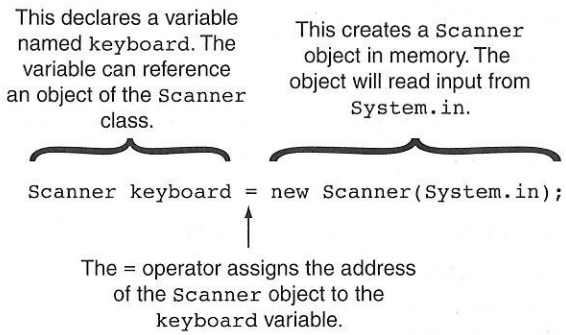
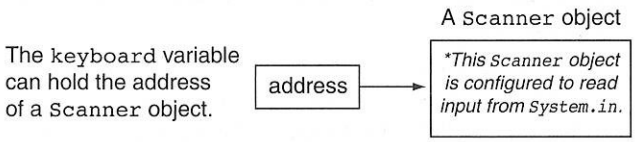


Figure 2-13 The keyboard variable references a Scanner object



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

`nextByte`	**Example Usage:**
	`byte x;`
	`Scanner keyboard = new Scanner(System.in);`
	`System.out.print("Enter a byte value: ");`
	`x = keyboard.nextByte();`
	Description: Returns input as a `byte`.

`nextDouble`	**Example Usage:**
	`double number;`
	`Scanner keyboard = new Scanner(System.in);`
	`System.out.print("Enter a double value: ");`
	`number = keyboard.nextDouble();`
	Description: Returns input as a `double`.

`nextFloat`	**Example Usage:**
	`float number;`
	`Scanner keyboard = new Scanner(System.in);`
	`System.out.print("Enter a float value: ");`
	`number = keyboard.nextFloat();`
	Description: Returns input as a `float`.

`nextInt`	**Example Usage:**
	`int number;`
	`Scanner keyboard = new Scanner(System.in);`
	`System.out.print("Enter an integer value: ");`
	`number = keyboard.nextInt();`
	Description: Returns input as an `int`.

`nextLine`	**Example Usage:**
	`String name;`
	`Scanner keyboard = new Scanner(System.in);`
	`System.out.print("Enter your name: ");`
	`name = keyboard.nextLine();`
	Description: Returns input as a `String`.

`nextLong`	**Example Usage:**
	`long number;`
	`Scanner keyboard = new Scanner(System.in);`
	`System.out.print("Enter a long value: ");`
	`number = keyboard.nextLong();`
	Description: Returns input as a `long`.

`nextShort`	**Example Usage:**
	`short number;`
	`Scanner keyboard = new Scanner(System.in);`
	`System.out.print("Enter a short value: ");`
	`number = keyboard.nextShort();`
	Description: Returns input as a `short`.

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 newline 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

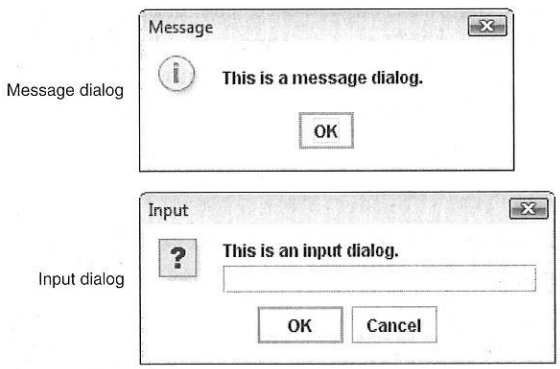
CONCEPT: The `JOptionPane` class allows you to quickly display a dialog box, which is a small graphical window displaying a message or requesting input.

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.

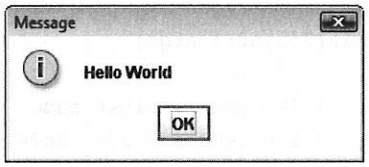
Displaying Message Dialogs

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



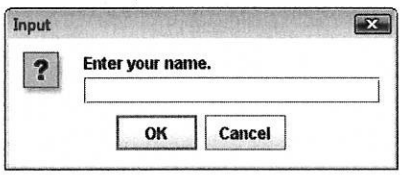
Displaying Input Dialogs

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)

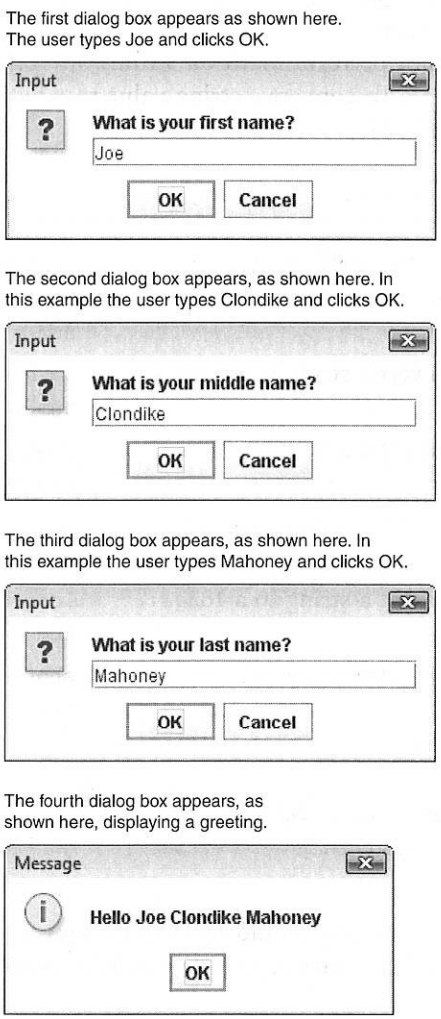
```
1  import javax.swing.JOptionPane;
2
3  /**
4     This program demonstrates using dialogs with
5     JOptionPane.
6  */
7
8  public class NamesDialog
9  {
10    public static void main(String[] args)
11    {
12       String firstName;  // The user's first name
13       String middleName; // The user's 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



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
`Byte.parseByte`	Convert a string to a `byte`.	`byte num;` `num = Byte.parseByte(str);`
`Double.parseDouble`	Convert a string to a `double`.	`double num;` `num = Double.parseDouble(str);`
`Float.parseFloat`	Convert a string to a `float`.	`float num;` `num = Float.parseFloat(str);`
`Integer.parseInt`	Convert a string to an `int`.	`int num;` `num = Integer.parseInt(str);`
`Long.parseLong`	Convert a string to a `long`.	`long num;` `num = Long.parseLong(str);`
`Short.parseShort`	Convert a string to a `short`.	`short num;` `num = Short.parseShort(str);`

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);
```

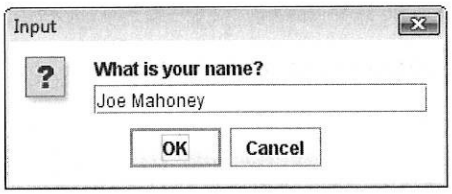
After this code executes, the `price` variable will hold the value entered by the user, converted to a `double`. Code Listing 2-33 shows a complete program. This is a modification of the `Payroll.java` program in Code Listing 2-29. When this program executes, the dialog boxes shown in Figure 2-18 will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

Code Listing 2-33 (PayrollDialog.java)

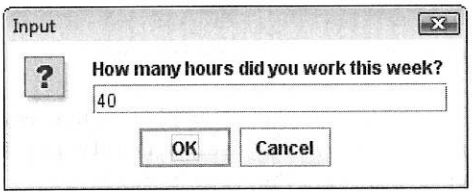
```
1  import javax.swing.JOptionPane;
2
3  /**
4     This program demonstrates using dialogs with
5     JOptionPane.
6  */
7
8  public class PayrollDialog
9  {
10    public static void main(String[] args)
11    {
12       String inputString;      // For reading input
13       String name;             // 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`

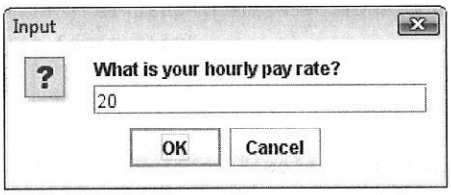
The first dialog box appears as shown here. The user enters his or her name and then clicks OK.



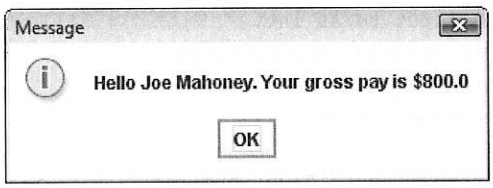
The second dialog box appears, as shown here. The user enters the number of hours worked and then clicks OK.



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?

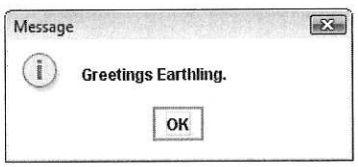
Message dialog

Input dialog

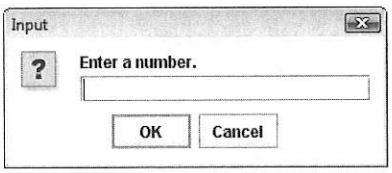
2.35 Write code that will display each of the dialog boxes shown in Figure 2-19.

Figure 2-19 Dialog boxes

a)



b)



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. _____ are predefined reserved words which serve a special purpose in programming language syntax.
   a. Key words
   b. Variables
   c. Data types
   d. Constants
3. Java is a _____ language, which means that it allows only values of compatible data types to be stored in variables.
   a. case-sensitive
   b. strongly typed
   c. object-oriented
   d. a and c
4. This is an example of a primitive data type.
   a. `double`
   b. `char`
   c. `long`
   d. all of the above

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. Characters in Java are represented by a uniquely identifiable code known as _____.
   a. bits
   b. bytes
   c. Unicode
   d. literals
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. The `String` variable is a/an _____ because it holds the memory address of the associated data item.
    a. primitive type variable
    b. class type variable
    c. instance variable
    d. named constant
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 num1 = 15, num2 = 5, result;
result = num1%num2;
System.out.println("The output is:\t "+ result);
```

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. Write a statement containing the three floating type variables length, width, and area, with length initialized to 1.5 and width initialized to 4.5.
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 be the following code output if the input value entered by the user is 7?

```
Scanner keyboard = new Scanner(System.in);
int number = keyboard.nextInt();
System.out.print("The number entered is : " + number);
```

9. What will be the following code output?

```
char ch = a';
String name = "Adam";
System.out.println("Have "+ch+" nice day, "+name);
```

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 90 in the rate integer variable.
Store 10 in the hour integer variable.
Multiply rate by time and store the result in the total wage variable.
Display the contents of the total wage 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. What do you understand by the term "scope of a variable"?

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. What is the significance of `boolean` data types in programming?
8. What is the purpose of using the key word `final` in the declaration statement of 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. Write an expression that reads `double` type values that are keyboard inputs.

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. Warehouse Inventories

Write a program that contains the following `String` variables: `productName`, `itemCode`, and `CustomerId`. Initialize these variables with the values "video-game", "A101", and "VD112" respectively. The program should display the contents of these variables on the screen.

3. Swimming Club Membership

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

- Name of the member
- Membership fees
- Registration date
- Expiry date

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

4. Vowel Triangle Pattern

Write a program that displays the following pattern:

```
A
AE
AEI
AEIO
AEIOU
```

5. 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.*

6. Volume Calculation

A cylindrical vessel has a radius of 7 cm and a height of 8 cm. Write a program that calculates the volume of this vessel using the following formula: volume of cylinder = 3.14 * $(\text{radius})^2$ * height

7. 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.*

8. Cookie Calories

A bag of cookies holds 40 cookies. The calorie information on the bag claims that there are 10 servings in the bag and that a serving equals 300 calories. Write a program that lets the user enter the number of cookies he or she actually ate and then reports the number of total calories consumed.

VideoNote
The Miles-per-Gallon Problem

9. Miles-per-Gallon

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

MPG = Miles driven / 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. Production Average

Write a program that asks the user to enter the values of the rate of egg production of European and African honey bees and calculates the average egg production per honey bee in a week.

11. Stationary Expenditure

A stationery store sells pencils at $1.20 each, erasers at $2 each, and compasses at $5 each. Write a program that asks the user for the number of each of these items purchased, calculates the total expenditure made by the user, and then displays the result 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 20 percent of the total after adding the tax. Display the meal charge, tax amount, tip amount, and total bill on the screen.

14. Male and Female Percentages

Write a program that asks the user for the number of males and the number of females registered in a class. The program should display the percentage of males and females in the class.

Hint: Suppose there are 8 males and 12 females in a class. There are 20 students in the class. The percentage of males can be calculated as 8 ÷ 20 = 0.4, or 40%. The percentage of females can be calculated as 12 ÷ 20 = 0.6, or 60%.

15. Stock Commission

Kathryn bought 600 shares of stock at a price of $21.77 per share. She must pay her stockbroker 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)

16. 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

17. Ingredient Adjuster

A cookie recipe calls for the following ingredients:

- 1.5 cups of sugar
- 1 cup of butter
- 2.75 cups of flour

The recipe produces 48 cookies with these amounts of the ingredients. Write a program that asks the user how many cookies he or she wants to make, and then displays the number of cups of each ingredient needed for the specified number of cookies.

18. 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!
```

19. 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.)