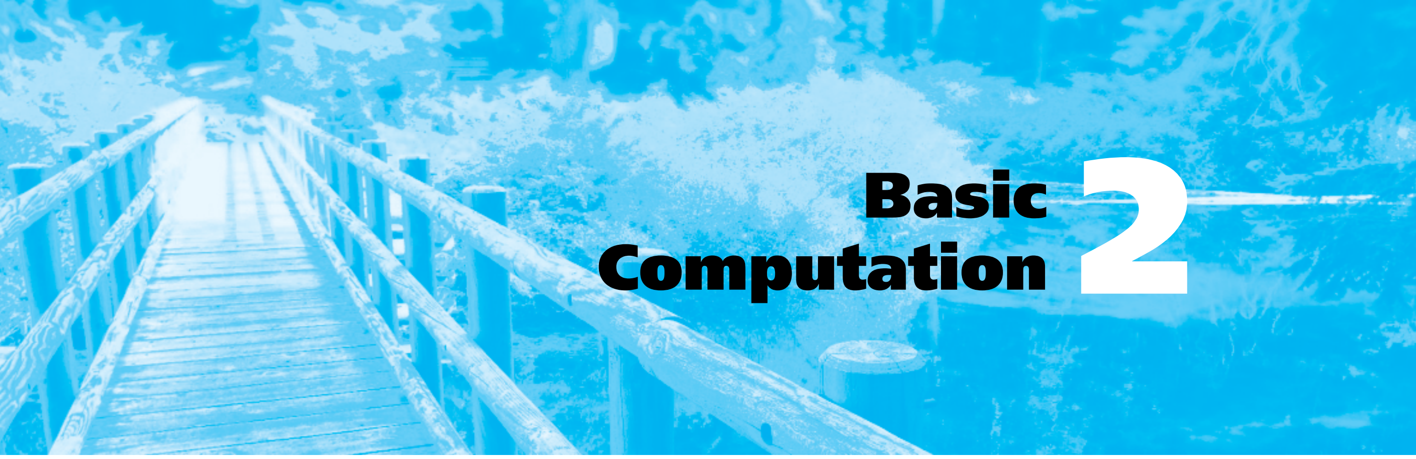


Basic Computation 2

Beauty without expression tires.

—RALPH WALDO EMERSON, *The Conduct of Life*, (1876)

In this chapter, we explain enough about the Java language to allow you to write simple Java programs. You do not need any programming experience to understand this chapter. If you are already familiar with some other programming language, such as Visual Basic, C, C++, or C#, much that is in Section 2.1 will already be familiar to you. However, even if you know the concepts, you should learn the Java way of expressing them.

OBJECTIVES

After studying this chapter, you should be able to

- Describe the Java data types that are used for simple data like numbers and characters
- Write Java statements to declare variables and define named constants
- Write assignment statements and expressions containing variables and constants
- Define strings of characters and perform simple string processing
- Write Java statements that accomplish keyboard input and screen output
- Adhere to stylistic guidelines and conventions
- Write meaningful comments within a program
- Use the class `JOptionPane` to perform window-based input and output

PREREQUISITES

If you have not read Chapter 1, you should read at least the section of Chapter 1 entitled "A First Java Application Program" to familiarize yourself with the notions of class, object, and method. Also, material from the graphics supplement in Chapter 1 is used in the section "Style Rules Applied to a Graphics Applet" in the graphics supplement of this chapter.

2.1 VARIABLES AND EXPRESSIONS

In this section, we explain how simple variables and arithmetic expressions are used in Java programs. Some of the terms used here were introduced in Chapter 1. We will, however, review them again.

Variables

Variables in a program are used to store data such as numbers and letters. They can be thought of as containers of a sort. The number, letter, or other data item in a variable is called its **value.** This value can be changed, so that at one time the variable contains, say, 6, and at another time, after the program has run for a while, the variable contains a different value, such as 4.

A variable is a program component used to store or represent data

For example, the program in Listing 2.1 uses the variables `numberOfBaskets`, `eggsPerBasket`, and `totalEggs`. When this program is run, the statement

```
eggsPerBasket = 6;
```

sets the value of `eggsPerBasket` to 6.

Variables represent memory locations

In Java, variables are implemented as memory locations, which we described in Chapter 1. Each variable is assigned one memory location. When the variable is given a value, the value is encoded as a string of 0s and 1s and is placed in the variable's memory location.

LISTING 2.1 A Simple Java Program

```
public class EggBasket
{
    public static void main(String[] args)
    {
        int numberOfBaskets, eggsPerBasket, totalEggs;

        numberOfBaskets = 10;
        eggsPerBasket = 6;

        totalEggs = numberOfBaskets * eggsPerBasket;

        System.out.println("If you have");
        System.out.println(eggsPerBasket + " eggs per basket and");
        System.out.println(numberOfBaskets + " baskets, then");
        System.out.println("the total number of eggs is " + totalEggs);
    }
}
```

Variable declarations

Assignment statement

Sample Screen Output

```
If you have
6 eggs per basket and
10 baskets, then
the total number of eggs is 60
```

You should choose variable names that are helpful. The names should suggest the variables' use or indicate the kind of data they will hold. For example, if you use a variable to count something, you might name it `count`. If the variable is used to hold the speed of an automobile, you might call the variable `speed`. You should almost never use single-letter variable names like `x` and `y`. Somebody reading the statement

Choose meaningful variable names

```
x = y + z;
```

would have no idea of what the program is really adding. The names of variables must also follow certain spelling rules, which we will detail later in the section "Java Identifiers."

Before you can use a variable in your program, you must state some basic information about each one. The compiler—and so ultimately the computer—needs to know the name of the variable, how much computer memory to reserve for the variable, and how the data item in the variable is to be coded as strings of 0s and 1s. You give this information in a **variable declaration.** Every variable in a Java program must be declared before it is used for the first time.

A variable declaration tells the computer what type of data the variable will hold. That is, you declare the variable's data type. Since different types of data are stored in the computer's memory in different ways, the computer must know the type of a variable so it knows how to store and retrieve the value of the variable from the computer's memory. For example, the following line from Listing 2.1 declares `numberOfBaskets`, `eggsPerBasket`, and `totalEggs` to be variables of data type `int`:

```
int numberOfBaskets, eggsPerBasket, totalEggs;
```

A variable declaration consists of a type name, followed by a list of variable names separated by commas. The declaration ends with a semicolon. All the variables named in the list are declared to have the same data type, as given at the start of the declaration.

If the data type is `int`, the variable can hold whole numbers, such as 42, −99, 0, and 2001. A whole number is called an **integer.** The word `int` is an abbreviation of *integer.* If the type is `double`, the variable can hold numbers having a decimal point and a fractional part after the decimal point. If the type is `char`, the variables can hold any one character that appears on the computer keyboard.

Every variable in a Java program must be declared before the variable can be used. Normally, a variable is declared either just before it is used or at the start of a section of your program that is enclosed in braces { }. In the simple programs we have seen so far, this means that variables are declared either just before they are used or right after the lines

Declare variables before using them

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

RECAP Variable Declarations

In a Java program, you must declare a variable before it can be used. A variable declaration has the following form:

SYNTAX

```
Type Variable_1, Variable_2, . . .;
```

EXAMPLES

```
int styleNumber, numberOfChecks, numberOfDeposits;
double amount, interestRate;
char answer;
```

Data Types

As you have learned, a data type specifies a set of values and their operations. In fact, the values have a particular data type because they are stored in memory in the same format and have the same operations defined for them.

A data type specifies a set of values and operations

REMEMBER Syntactic Variables

When you see something in this book like *Type*, *Variable_1,* or *Variable_2* used to describe Java syntax, these words do not literally appear in your Java code. They are **syntactic variables,** which are a kind of blank that you fill in with something from the category that they describe. For example, *Type* can be replaced by `int`, `double`, `char`, or any other type name. *Variable_1* and *Variable_2* can each be replaced by any variable name.

Class types and primitive types

Java has two main kinds of data types: class types and primitive types. As the name implies, a **class type** is a data type for objects of a class. Since a class is like a blueprint for objects, the class specifies how the values of its type are stored and defines the possible operations on them. As we implied in the previous chapter, a class type has the same name as the class. For example, quoted strings such as `"Java is fun"` are values of the class type `String`, which is discussed later in this chapter.

Variables of a **primitive type** are simpler than objects (values of a class type), which have both data and methods. A value of a primitive type is an indecomposable value, such as a single number or a single letter. The types `int`, `double`, and `char` are examples of primitive types.

FIGURE 2.1 **Primitive Types**

Type Name	Kind of Value	Memory Used	Range of Values
`byte`	Integer	1 byte	–128 to 127
`short`	Integer	2 bytes	–32,768 to 32,767
`int`	Integer	4 bytes	–2,147,483,648 to 2,147,483,647
`long`	Integer	8 bytes	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	Floating-point	4 bytes	$\pm 3.40282347 \times 10^{+38}$ to $\pm 1.40239846 \times 10^{-45}$
`double`	Floating-point	8 bytes	$\pm 1.79769313486231570 \times 10^{+308}$ to $\pm 4.94065645841246544 \times 10^{-324}$
`char`	Single character (Unicode)	2 bytes	All Unicode values from 0 to 65,535
`boolean`		1 or more bytes	True or false

Figure 2.1 lists all of Java's primitive types. Four types are for integers, namely, `byte`, `short`, `int`, and `long`. The only differences among the various integer types are the range of integers they represent and the amount of computer memory they use. If you cannot decide which integer type to use, use the type `int`.

A number having a fractional part—such as the numbers 9.99, 3.14159, −5.63, and 5.0—is called a **floating-point number.** Notice that 5.0 is a floating-point number, not an integer. If a number has a fractional part, even if the fractional part is zero, it is a floating-point number. As shown in Figure 2.1, Java has two data types for floating-point numbers, `float` and `double`. For example, the following code declares two variables, one of type `float` and one of type `double`:

A floating-point number has a fractional part

```
float cost;
double capacity;
```

As with integer types, the differences between `float` and `double` involve the range of their values and their storage requirements. If you cannot decide between the types `float` and `double`, use `double`.

The primitive type `char` is used for single characters, such as letters, digits, or punctuation. For example, the following declares the variable `symbol` to be of type `char`, stores the character for uppercase *A* in `symbol`, and then displays *A* on the screen:

```
char symbol;
symbol = 'A';
System.out.println(symbol);
```

In a Java program, we enclose a single character in single quotes, as in `'A'`. Note that there is only one symbol for a single quote. The same quote symbol is used on both sides of the character. Finally, remember that uppercase letters and lowercase letters are different characters. For example, `'a'` and `'A'` are two different characters.

Single quotes enclose a character

The last primitive type we have to discuss is the type `boolean`. This data type has two values, true and false. We could, for example, use a variable of type `boolean` to store the answer to a true/false question such as "Is `eggCount` less than 12?" We will have more to say about the data type `boolean` in the next chapter. Although a boolean requires only 1 bit of storage, in practice 1 or more bytes is used, depending upon the context. This is because the CPU is designed to read 1 or more bytes at a time, so it is actually more time consuming to access an individual bit.

All primitive type names in Java begin with a lowercase letter. In the next section, you will learn about a convention in which class type names—that is, the names of classes—begin with an uppercase letter.

Although you declare variables for class types and primitive types in the same way, these two kinds of variables store their values using different mechanisms. Chapter 5 will explain class type variables in more detail. In this chapter and the next two, we will concentrate on primitive types. We will occasionally use variables of a class type before Chapter 5, but only in contexts where they behave pretty much the same as variables of a primitive type.

Java Identifiers

The technical term for a name in a programming language, such as the name of a variable, is an **identifier.** In Java, an identifier (a name) can contain only letters, digits 0 through 9, and the underscore character (_). The first character in an identifier cannot be a digit.[1] In particular, no name can contain a space or any other character such as a dot (period) or an asterisk (*). There is no limit to the length of an identifier. Well, in practice, there is a limit, but Java has no official limit and will accept even absurdly long names. Java is **case sensitive.** That is, uppercase and lowercase letters are considered to be different characters. For example, Java considers `mystuff`, `myStuff`, and `MyStuff` to be three different identifiers, and you could have three different variables with these three names. Of course, writing variable names that differ only in their capitalization is a poor programming practice, but the Java compiler would happily accept them. Within these constraints, you can use any name you want for a variable, a class, or any other item you define in a Java program. But there are some style guidelines for choosing names.

Java is case sensitive

[1] Java does allow the dollar sign ($) to appear in an identifier, treating it as a letter. But such identifiers have a special meaning. It is intended to identify code generated by a machine, so you should not use the $ symbol in your identifiers.

Our somewhat peculiar use of uppercase and lowercase letters, such as `numberOfBaskets`, deserves some explanation. It would be perfectly legal to use `NumberOfBaskets` or `number_of_baskets` instead of `numberOfBaskets`, but these other names would violate some well-established conventions about how you should use uppercase and lowercase letters. Under these conventions, we write the names of variables using only letters and digits. We "punctuate" multiword names by using uppercase letters—since we cannot use spaces. The following are all legal names that follow these conventions:

Legal identifiers

```
inputStream YourClass CarWash hotCar theTimeOfDay
```

Notice that some of these legal names start with an uppercase letter and others, such as `hotCar`, start with a lowercase letter. We will always follow the convention that the names of classes start with an uppercase letter, and the names of variables and methods start with a lowercase letter.

The following identifiers are all illegal in Java, and the compiler will complain if you use any of them:

Illegal identifiers

```
prenhall.com go-team Five* 7eleven
```

The first three contain illegal characters, either a dot, a hyphen, or an asterisk. The last name is illegal because it starts with a digit.

Some words in a Java program, such as the primitive types and the word `if`, are called **keywords** or **reserved words.** They have a special predefined meaning in the Java language and cannot be used as the names of variables, classes, or methods, or for anything other than their intended meaning. All Java keywords are entirely in lowercase. A full list of these keywords appears in Appendix 11, which is online, and you will learn them as we go along. The program listings in this book show keywords, such as `public`, `class`, `static`, and `void`, in a special color. The text editors within an IDE often identify keywords in a similar manner.

Java keywords have special meanings

Some other words, such as `main` and `println`, have a predefined meaning but are not keywords. That means you can change their meaning, but it is a bad idea to do so, because it could easily confuse you or somebody else reading your program.

RECAP Identifiers (Names)

The name of something in a Java program, such as a variable, class, or method, is called an identifier. It must not start with a digit and may contain only letters, digits 0 through 9, and the underscore character (_). Uppercase and lowercase letters are considered to be different characters. (The symbol $ is also allowed, but it is reserved for special purposes, and so you should not use $ in a Java name.)

Although it is not required by the Java language, the common practice, and the one followed in this book, is to start the names of classes with uppercase letters and to start the names of variables and methods with lowercase letters. These names are usually spelled using only letters and digits.

GOTCHA Java Is Case Sensitive

Do not forget that Java is case sensitive. If you use an identifier, like `myNumber`, and then in another part of your program you use the spelling `MyNumber`, Java will not recognize them as being the same identifier. To be seen as the same identifier, they must use exactly the same capitalization. ■

FAQ Why should I follow naming conventions? And who sets the rules?

By following naming conventions, you can make your programs easier to read and to understand. Typically, your supervisor or instructor determines the conventions that you should follow when writing Java programs. However, the naming conventions that we just gave are almost universal among Java programmers. We will mention stylistic conventions for other aspects of a Java program as we go forward. Sun Microsystems provides its own conventions on its Web site. While the company suggests that all Java programmers follow these conventions, not everyone does.

Assignment Statements

The most straightforward way to give a variable a value or to change its value is to use an **assignment statement.** For example, if `answer` is a variable of type `int` and you want to give it the value 42, you could use the following assignment statement:

```
    answer = 42;
```

The equal sign, `=`, is called the **assignment operator** when it is used in an assignment statement. It does not mean what the equal sign means in other contexts. The assignment statement is an order telling the computer to change the value stored in the variable on the left side of the assignment operator to the value of the **expression** on the right side. Thus, an assignment statement always consists of a single variable followed by the assignment operator (the equal sign) followed by an expression. The assignment statement ends with a semicolon. So assignment statements take the form

An assignment statement gives a value to a variable

```
    Variable = Expression;
```

The expression can be another variable, a number, or a more complicated expression made up by using **arithmetic operators**, such as + and -, to combine variables and numbers. For example, the following are all examples of assignment statements:

```
amount = 3.99;
firstInitial = 'B';
score = numberOfCards + handicap;
eggsPerBasket = eggsPerBasket - 2;
```

All the names, such as amount, score, and numberOfCards, are variables. We are assuming that the variable amount is of type double, firstInitial is of type char, and the rest of the variables are of type int.

When an assignment statement is executed, the computer first evaluates the expression on the right side of the assignment operator (=) to get the value of the expression. It then uses that value to set the value of the variable on the left side of the assignment operator. You can think of the assignment operator as saying, "Make the value of the variable equal to what follows."

For example, if the variable numberOfCards has the value 7 and handicap has the value 2, the following assigns 9 as the value of the variable score:

```
score = numberOfCards + handicap;
```

In the program in Listing 2.1, the statement

* means multiply

```
totalEggs = numberOfBaskets * eggsPerBasket;
```

is another example of an assignment statement. It tells the computer to set the value of totalEggs equal to the number in the variable numberOfBaskets multiplied by the number in the variable eggsPerBasket. The asterisk character (*) is the symbol used for multiplication in Java.

Note that a variable can meaningfully occur on both sides of the assignment operator and can do so in ways that might at first seem a little strange. For example, consider

The same variable can occur on both sides of the =

```
count = count + 10;
```

This does not mean that the value of count is equal to the value of count plus 10, which, of course, is impossible. Rather, the statement tells the computer to add 10 to the *old* value of count and then make that the *new* value of count. In effect, the statement will increase the value of count by 10. Remember that when an assignment statement is executed, the computer first evaluates the expression on the right side of the assignment operator and then makes that result the new value of the variable on the left side of the assignment operator. As another example, the following assignment statement will decrease the value of eggsPerBasket by 2:

```
eggsPerBasket = eggsPerBasket - 2;
```

RECAP Assignment Statements Involving Primitive Types

An assignment statement that has a variable of a primitive type on the left side of the equal sign causes the following action: First, the expression on the right side of the equal sign is evaluated, and then the variable on the left side of the equal sign is set to this value.

SYNTAX

```
Variable = Expression;
```

EXAMPLE

```
score = goals – errors;
interest = rate * balance;
number = number + 5;
```

PROGRAMMING TIP Initialize Variables

A variable that has been declared, but that has not yet been given a value by an assignment statement (or in some other way), is said to be **uninitialized.** If the variable is a variable of a class type, it literally has no value. If the variable has a primitive type, it likely has some default value. However, your program will be clearer if you explicitly give the variable a value, even if you are simply reassigning the default value. (The exact details on default values have been known to change and should not be counted on.)

One easy way to ensure that you do not have an uninitialized variable is to initialize it within the declaration. Simply combine the declaration and an assignment statement, as in the following examples:

You can initialize a variable when you declare it

```
int count = 0;
double taxRate = 0.075;
char grade = 'A';
int balance = 1000, newBalance;
```

Note that you can initialize some variables and not initialize others in a declaration.

Sometimes the compiler may complain that you have failed to initialize a variable. In most cases, that will indeed be true. Occasionally, though, the compiler is mistaken in giving this advice. However, the compiler will not compile your program until you convince it that the variable in question is initialized. To make the compiler happy, initialize the variable when you declare it, even if the variable will be given another value before it is used for anything. In such cases, you cannot argue with the compiler. ■

RECAP Combining a Variable Declaration and an Assignment

You can combine the declaration of a variable with an assignment statement that gives the variable a value.

SYNTAX

```
Type Variable_1 = Expression_1, Variable_2 = Expression_2,
. . .;
```

EXAMPLES

```
int numberSeen = 0, increment = 5;
double height = 12.34, prize = 7.3 + increment;
char answer = 'y';
```

Simple Input

In Listing 2.1, we set the values of the variables `eggsPerBasket` and `numberOfBaskets` to specific numbers. It would make more sense to obtain the values needed for the computation from the user, so that the program could be run again with different numbers. Listing 2.2 shows a revision of the program in Listing 2.1 that asks the user to enter numbers as input at the keyboard.

Use the standard class Scanner to accept keyboard input

We use the class `Scanner`, which Java supplies, to accept keyboard input. Our program must import the definition of the `Scanner` class from the package `java.util`. Thus, we begin the program with the following statement:

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

The following line sets things up so that data can be entered from the keyboard:

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

This line must appear before the first statement that takes input from the keyboard. That statement in our example is

```
eggsPerBasket = keyboard.nextInt();
```

This assignment statement gives a value to the variable `eggsPerBasket`. The expression on the right side of the equal sign, namely

```
keyboard.nextInt()
```

reads one `int` value from the keyboard. The assignment statement makes this `int` value the value of the variable `eggsPerBasket`, replacing any value that the variable might have had. When entering numbers at the keyboard, the user must either separate multiple numbers with one or more spaces or place each number on its own line. Section 2.3 will explain such keyboard input in detail.

LISTING 2.2 A Program with Keyboard Input

```
import java.util.Scanner;
public class EggBasket2
{
   public static void main(String[] args)
   {
      int numberOfBaskets, eggsPerBasket, totalEggs;
      Scanner keyboard = new Scanner(System.in);
      System.out.println("Enter the number of eggs in each basket:");
      eggsPerBasket = keyboard.nextInt();
      System.out.println("Enter the number of baskets:");
      numberOfBaskets = keyboard.nextInt();

      totalEggs = numberOfBaskets * eggsPerBasket;

      System.out.println("If you have");
      System.out.println(eggsPerBasket + " eggs per basket and");
      System.out.println(numberOfBaskets + " baskets, then");
      System.out.println("the total number of eggs is " + totalEggs);

      System.out.println("Now we take two eggs out of each basket.");

      eggsPerBasket = eggsPerBasket - 2;
      totalEggs = numberOfBaskets * eggsPerBasket;

      System.out.println("You now have");
      System.out.println(eggsPerBasket + " eggs per basket and");
      System.out.println(numberOfBaskets + " baskets.");
      System.out.println("The new total number of eggs is " + totalEggs);
   }
}
```

Gets the `Scanner` *class from the package (library)* `java.util`

Sets up things so the program can accept keyboard input

Reads one whole number from the keyboard

Sample Screen Output

```
Enter the number of eggs in each basket:
6
Enter the number of baskets:
10
If you have
6 eggs per basket and
10 baskets, then
the total number of eggs is 60
Now we take two eggs out of each basket.
You now have
4 eggs per basket and
10 baskets.
The new total number of eggs is 40
```

Simple Screen Output

Now we will give you a brief overview of screen output—just enough to allow you to write and understand programs like the one in Listing 2.2. `System` is a class that is part of the Java language, and `out` is a special object within that class. The object `out` has `println` as one of its methods. It may seem strange to write `System.out.println` to call a method, but that need not concern you at this point. Chapter 6 will provide some details about this notation.

So

```
System.out.println(eggsPerBasket + "eggs per basket.");
```

displays the value of the variable `eggsPerBasket` followed by the phrase *eggs per basket.* Notice that the + symbol does not indicate arithmetic here. It denotes another kind of "and." You can read the preceding Java statement as an instruction to display the value of the variable `eggsPerBasket` and then to display the string `"eggs per basket."`

Section 2.3 will continue the discussion of screen output.

Constants

A variable can have its value changed. That is why it is called a variable: Its value *varies*. A number like 2 cannot change. It is always 2. It is never 3. In Java, terms like `2` or `3.7` are called **constants,** or **literals,** because their values do not change.

A constant does not change in value

Constants need not be numbers. For example, `'A'`, `'B'`, and `'$'` are three constants of type `char`. Their values cannot change, but they can be used in an assignment statement to change the value of a variable of type `char`. For example, the statement

```
firstInitial = 'B';
```

changes the value of the char variable `firstInitial` to `'B'`.

There is essentially only one way to write a constant of type `char`, namely, by placing the character between single quotes. On the other hand, some of the rules for writing numeric constants are more involved. Constants of integer types are written the way you would expect them to be written, such as `2`, `3`, `0`, `-3`, or `752`. An integer constant can be prefaced with a plus sign or a minus sign, as in `+12` and `−72`. Numeric constants cannot contain commas. The number 1,000 is *not* correct in Java. Integer constants cannot contain a decimal point. A number with a decimal point is a floating-point number.

Floating-point constant numbers may be written in either of two forms. The simple form is like the everyday way of writing numbers with digits after the decimal point. For example, `2.5` is a floating-point constant. The other, slightly more complicated form is similar to a notation commonly used in mathematics and the physical sciences, **scientific notation.** For instance, consider the number 865000000.0. This number can be expressed more clearly in the following scientific notation:

Java's e notation is like scientific notation

```
8.65 × 10^8
```

Java has a similar notation, frequently called either **e notation** or **floating-point notation.** Because keyboards have no way of writing exponents, the 10 is omitted and both the multiplication sign and the 10 are replaced by the letter `e`. So in Java, 8.65×10^8 is written as `8.65e8`. The `e` stands for *exponent*, since it is followed by a number that is thought of as an exponent of 10. This form and the less convenient form `865000000.0` are equivalent in a Java program. Similarly, the number 4.83×10^{-4}, which is equal to 0.000483, could be written in Java as either `0.000483` or `4.83e−4`. Note that you also could write this number as `0.483e−3` or `48.3e−5`. Java does not restrict the position of the decimal point.

VideoNote
Another sample program

The number before the `e` may contain a decimal point, although it doesn't have to. The number after the `e` cannot contain a decimal point. Because multiplying by 10 is the same as moving the decimal point in a number, you can think of a positive number after the `e` as telling you to move the decimal point that many digits to the right. If the number after the `e` is negative, you move the decimal point that many digits to the left. For example, `2.48e4` is the same number as `24800.0`, and `2.48e-2` is the same number as `0.0248`.

FAQ What is "floating" in a floating-point number?

Floating-point numbers got their name because, with the e notation we just described, the decimal point can be made to "float" to a new location by adjusting the exponent. You can make the decimal point in `0.000483` float to after the 4 by expressing this number as the equivalent expression `4.83e−4`. Computer language implementers use this trick to store each floating-point number as a number with exactly one digit before the decimal point (and some suitable exponent). Because the implementation always floats the decimal point in these numbers, they are called floating-point numbers. Actually, the numbers are stored in another base, such as 2 or 16, rather than as the decimal (base 10) numbers we used in our example, but the principle is the same.

FAQ Is there an actual difference between the constants `5` and `5.0`?

The numbers 5 and 5.0 are conceptually the same number. But Java considers them to be different. Thus, `5` is an integer constant of type `int`, but `5.0` is a floating-point constant of type `double`. The number 5.0 contains a fractional part, even though the fraction is 0. Although you might see the numbers 5 and 5.0 as having the same value, Java stores them differently. Both integers and floating-point numbers contain a finite number of digits when stored in a computer, but only integers are considered exact quantities. Because floating-point numbers have a fractional portion, they are seen as approximations.

GOTCHA Imprecision in Floating-Point Numbers

Floating-point numbers are stored with a limited amount of precision and so are, for all practical purposes, only approximate quantities. For example, the fraction one third is equal to

```
0.3333333 . . .
```

where the three dots indicate that the 3s go on forever. The computer stores numbers in a format somewhat like the decimal representation on the previously displayed line, but it has room for only a limited number of digits. If it can store only ten digits after the decimal, then one third is stored as

```
0.3333333333 (and no more 3s)
```

This number is slightly smaller than one third and so is only approximately equal to one third. In reality, the computer stores numbers in binary notation, rather than in base 10, but the principles are the same and the same sorts of things happen.

Not all floating-point numbers lose accuracy when they are stored in the computer. Integral values like 29.0 can be stored exactly in floating-point notation, and so can some fractions like one half. Even so, we usually will not know whether a floating-point number is exact or an approximation. When in doubt, assume that floating-point numbers are stored as approximate quantities. ■

Named Constants

Java provides a mechanism that allows you to define a variable, initialize it, and moreover fix the variable's value so that it cannot be changed. The syntax is

Name important constants

```
public static final Type Variable = Constant;
```

For example, we can give the name `PI` to the constant `3.14159` as follows:

```
public static final double PI = 3.14159;
```

You can simply take this as a long, peculiarly worded way of giving a name (like `PI`) to a constant (like `3.14159`), but we can explain most of what is on this line. The part

```
double PI = 3.14159;
```

simply declares `PI` as a variable and initializes it to 3.14159. The words that precede this modify the variable `PI` in various ways. The word `public` says that there are no restrictions on where you can use the name `PI`. The word `static` will have to wait until Chapter 6 for an explanation; for now, just be sure to include it. The word `final` means that the value 3.14159 is the final value assigned to `PI` or, to phrase it another way, that the program is not allowed to change the value of `PI`.

The convention for naming constants is to use all uppercase letters, with an underscore symbol (_) between words. For example, in a calendar program, you might define the following constant:

```
public static final int DAYS_PER_WEEK = 7;
```

Although this convention is not required by the definition of the Java language, most programmers adhere to it. Your programs will be easier to read if you can readily identify variables, constants, and so forth.

RECAP Named Constants

To define a name for a constant, write the keywords `public static final` in front of a variable declaration that includes the constant as the initializing value. Place this declaration within the class definition but outside of any method definitions, including the `main` method.

SYNTAX

```
public static finalType Variable = Constant;
```

EXAMPLES

```
public static final int MAX_STRIKES = 3;
public static final double MORTGAGE_INTEREST_RATE = 6.99;
public static final String MOTTO =
                            "The customer is right!";
public static final char SCALE = 'K';
```

Although it is not required, most programmers spell named constants using all uppercase letters, with an underscore to separate words.

Assignment Compatibilities

As the saying goes, "You can't put a square peg in a round hole," and you can't put a `double` value like 3.5 in a variable of type `int`. You cannot even put the `double` value 3.0 in a variable of type `int`. You cannot store a value of one type in a variable of another type unless the value is somehow converted to match the type of the variable. However, when dealing with numbers, this conversion will sometimes—but not always—be performed automatically for you. The conversion will always be done when you assign a value of an integer type to a variable of a floating-point type, such as

```
double doubleVariable = 7;
```

Slightly more subtle assignments, such as the following, also perform the conversion automatically:

```
int intVariable = 7;
double doubleVariable = intVariable;
```

More generally, you can assign a value of any type in the following list to a variable of any type that appears further down in the list:

```
byte → short → int → long → float → double
```

For example, you can assign a value of type `long` to a variable of type `float` or to a variable of type `double` (or, of course, to a variable of type `long`), but you cannot assign a value of type `long` to a variable of type `byte`, `short`, or `int`. Note that this is not an arbitrary ordering of the types. As you move down the list from left to right, the types become more precise, either because they allow larger values or because they allow decimal points in the numbers. Thus, you can store a value into a variable whose type allows more precision than the type of the value allows.

A value can be assigned to a variable whose type allows more precision

In addition, you can assign a value of type `char` to a variable of type `int` or to any of the numeric types that follow `int` in our list of types. This particular assignment compatibility will be important when we discuss keyboard input. However, we do not advise assigning a value of type `char` to a variable of type `int` except in certain special cases.[2]

If you want to assign a value of type `double` to a variable of type `int`, you must change the type of the value using a type cast, as we explain in the next section.

RECAP Assignment Compatibilities

You can assign a value of any type on the following list to a variable of any type that appears further down on the list:

```
byte → short → int → long → float → double
```

In particular, note that you can assign a value of any integer type to a variable of any floating-point type.

It is also legal to assign a value of type `char` to a variable of type `int` or to any of the numeric types that follow `int` in our list of types.

[2] Readers who have used certain other languages, such as C or C++, may be surprised to learn that you cannot assign a value of type `char` to a variable of type `byte`. This is because Java reserves two bytes of memory for each value of type `char` but naturally reserves only one `byte` of memory for values of type `byte`.

Type Casting

The title of this section has nothing to do with the Hollywood notion of typecasting. In fact, it is almost the opposite. In Java—and in most programming languages—a **type cast** changes the data type of a value from its normal type to some other type. For example, changing the type of the value 2.0 from `double` to `int` involves a type cast. The previous section described when you can assign a value of one type to a variable of another type and have the type conversion occur automatically. In all other cases, if you want to assign a value of one type to a variable of another type, you must perform a type cast. Let's see how this is done in Java.

Suppose you have the following:

```
double distance = 9.0;
int points = distance;  <-------- This assignment is illegal.
```

As the note indicates, the last statement is illegal in Java. You cannot assign a value of type `double` to a variable of type `int`, even if the value of type `double` happens to have all zeros after the decimal point and so is conceptually a whole number.

In order to assign a value of type `double` to a value of type `int`, you must place `(int)` in front of the value or the variable holding the value. For example, you can replace the preceding illegal assignment with the following and get a legal assignment:

A type cast changes the data type of a value

```
int points = (int)distance;  <-------- This assignment is legal.
```

The expression `(int)distance` is called a type cast. Neither `distance` nor the value stored in `distance` is changed in any way. But the value stored in `points` is the "`int` version" of the value stored in `distance`. If the value of `distance` is 25.36, the value of `(int)distance` is 25. So `points` contains 25, but the value of `distance` is still 25.36. If the value of `distance` is 9.0, the value assigned to `points` is 9, and the value of `distance` remains unchanged.

An expression like `(int) 25.36` or `(int)distance` is an expression that *produces* an `int` value. A type cast does not change the value of the source variable. The situation is analogous to computing the number of (whole) dollars you have in an amount of money. If you have $25.36, the number of dollars you have is 25. The $25.36 has not changed; it has merely been used to produce the whole number 25.

For example, consider the following code:

```
double dinnerBill = 25.36;
int dinnerBillPlusTip = (int)dinnerBill + 5;
System.out.println("The value of dinnerBillPlusTip is " +
   dinnerBillPlusTip);
```

Truncation discards the fractional part

The expression `(int)dinnerBill` produces the value 25, so the output of this code would be

```
The value of dinnerBillPlusTip is 30
```

But the variable `dinnerBill` still contains the value 25.36.

Be sure to note that when you type cast from a `double` to an `int`—or from any floating-point type to any integer type—the amount is not rounded. The part after the decimal point is simply discarded. This is known as **truncating.** For example, the following statements

```
double dinnerBill = 26.99;
int numberOfDollars = (int)dinnerBill;
```

set `numberOfDollars` to 26, not 27. The result is *not rounded*.

As we mentioned previously, when you assign an integer value to a variable of a floating-point type—`double`, for example—the integer is automatically type cast to the type of the variable. For example, the assignment statement

```
double point = 7;
```

is equivalent to

```
double point = (double)7;
```

The type cast `(double)` is implicit in the first version of the assignment. The second version, however, is legal.

RECAP Type Casting

In many situations, you cannot store a value of one type in a variable of another type unless you use a type cast that converts the value to an equivalent value of the target type.

SYNTAX

```
(Type_Name)Expression
```

EXAMPLES

```
double guess = 7.8;
int answer = (int)guess;
```

The value stored in `answer` will be 7. Note that the value is truncated, not rounded. Note also that the variable *guess* is not changed in any way; it still contains 7.8. The last assignment statement affects only the value stored in *answer*.

PROGRAMMING TIP Type Casting a Character to an Integer

Java sometimes treats values of type `char` as integers, but the assignment of integers to characters has no connection to the meaning of the characters. For example, the following type cast will produce the `int` value corresponding to the character `'7'`:

```
char symbol = '7';
System.out.println((int)symbol);
```

You might expect the preceding to display 7, but it does not. It displays the number 55. Java, like all other programming languages, uses an arbitrary numbering of characters to encode them. Thus, each character corresponds to an integer. In this correspondence, the digits 0 through 9 are characters just like the letters or the plus sign. No effort was made to have the digits correspond to their intuitive values. Basically, they just wrote down all the characters and then numbered them in the order they were written down. The character `'7'` just happened to get 55. This numbering system is called the Unicode system, which we discuss later in the chapter. If you have heard of the ASCII numbering system, the Unicode system is the same as the ASCII system for the characters in the English language. ■

SELF-TEST QUESTIONS

1. Which of the following may be used as variable names in Java?

   `rate1, 1stPlayer, myprogram.java, long, TimeLimit, numberOfWindows`

2. Can a Java program have two different variables with the names `aVariable` and `avariable`?

3. Give the declaration for a variable called `count` of type `int`. The variable should be initialized to zero in the declaration.

4. Give the declaration for two variables of type `double`. The variables are to be named `rate` and `time`. Both variables should be initialized to zero in the declaration.

5. Write the declaration for two variables called `miles` and `flowRate`. Declare the variable `miles` to be of type `int` and initialize it to zero in the declaration. Declare the variable `flowRate` to be of type `double` and initialize it to 50.56 in the declaration.

6. What is the normal spelling convention for named constants?

7. Give a definition for a named constant for the number of hours in a day.

8. Write a Java assignment statement that will set the value of the variable `interest` to the value of the variable `balance` multiplied by 0.05.

9. Write a Java assignment statement that will set the value of the variable `interest` to the value of the variable `balance` multiplied by the value of the variable `rate`. The variables are of type `double`.

10. Write a Java assignment statement that will increase the value of the variable `count` by 3. The variable is of type `int`.

11. What is the output produced by the following lines of program code?

```
char a, b;
a = 'b';
System.out.println(a);
b = 'c';
System.out.println(b);
a = b;
System.out.println(a);
```

12. In the Programming Tip entitled "Type Casting a Character to an Integer," you saw that the following does not display the integer 7:

```
char symbol = '7';
System.out.println((int)symbol);
```

Thus, `(int)symbol` does not produce the number corresponding to the digit in `symbol`. Can you write an expression that will work to produce the integer that intuitively corresponds to the digit in `symbol`, assuming that `symbol` contains one of the ten digits 0 through 9? (*Hint*: The digits do correspond to consecutive integers, so if `(int)'7'` is 55, then `(int)'8'` is 56.)

Arithmetic Operators

In Java, you can perform arithmetic involving addition, subtraction, multiplication, and division by using the arithmetic operators +, −, *, and /, respectively. You indicate arithmetic in basically the same way that you do in ordinary arithmetic or algebra. You can combine variables or numbers—known collectively as **operands**—with these operators and parentheses to form an **arithmetic expression.** Java has a fifth arithmetic operator, %, that we will define shortly.

An arithmetic expression combines operands, operators, and parentheses

The meaning of an arithmetic expression is basically what you expect it to be, but there are some subtleties about the type of the result and, occasionally, even about the value of the result. All five of the arithmetic operators can be used with operands of any of the integer types, any of the floating-point types, and even with operands of differing types. The type of the value produced depends on the types of the operands being combined.

Let's start our discussion with simple expressions that combine only two operands, that is, two variables, two numbers, or a variable and a number. If both operands are of the same type, the result is of that type. If one of the operands is of a floating-point type and the other is of an integer type, the result is of the floating-point type.

Operands in an arithmetic expression can have mixed data types

For example, consider the expression

```
amount + adjustment
```

If the variables `amount` and `adjustment` are both of type `int`, the result—that is, the value returned by the operation—is of type `int`. If either `amount` or `adjustment`, or both, are of type `double`, the result is of type `double`. If you replace the addition operator, +, with any of the operators −, *, /, or %, the type of the result is determined in the same way.

Larger expressions using more than two operands can always be viewed as a series of steps, each of which involves only two operands. For example, to evaluate the expression

```
balance + (balance * rate)
```

you (or the computer) evaluate `balance * rate` and obtain a number, and then you form the sum of that number and `balance`. This means that the same rule we used to determine the type of an expression containing two operands can also be used for more complicated expressions: If all of the items being combined are of the same type, the result is of that type. If some of the items being combined are of integer types and some are of floating-point types, the result is of a floating-point type.

Knowing whether the value produced has an integer type or a floating-point type is typically all that you need to know. However, if you need to know the exact type of the value produced by an arithmetic expression, you can find out as follows: The type of the result produced is one of the types used in the expression. Of all the types used in the expression, it is the one that appears rightmost in the following list:

`byte` → `short` → `int` → `long` → `float` → `double`

Note that this is the same sequence as the one used to determine automatic type conversions.

The division operator (`/`) deserves special attention, because the type of the result can affect the value produced in a dramatic way. When you combine two operands with the division operator and at least one of the operands is of type `double`—or of some other floating-point type—the result is what you would normally expect of a division. For example, `9.0/2` has one operand of type `double`, namely, `9.0`. Hence, the result is the type `double` number 4.5. However, when both operands are of an integer type, the result can be surprising. For example, `9/2` has two operands of type `int`, so it yields the type `int` result 4, not 4.5. The fraction after the decimal point is simply lost. Be sure to notice that when you divide two integers, the result *is not rounded;* the part

Integer division truncates the result

after the decimal point is discarded—that is, truncated—no matter how large it is. So `11/3` is 3, not 3.6666. . . . Even if the fractional portion after the decimal point is zero, that decimal point and zero are still lost. Surprisingly, this seemingly trivial difference can be of some significance. For example, `8.0/2` evaluates to the type `double` value 4.0, which is only an approximate quantity. However, `8/2` evaluates to the `int` value 4, which is an exact quantity. The approximate nature of 4.0 can affect the accuracy of any further calculation that is performed using this result.

Java's fifth arithmetic operator is the **remainder operator,** or **modulus operator,** denoted by %. When you divide one number by another, you get a result (which some call a quotient) and a remainder—that is, the amount left over. The % operator gives you the remainder. Typically, the % operator is used with operands of integer types to recover something equivalent to the fraction after the decimal point. For example, 14 divided by 4 yields 3 with a remainder of 2, that is, with 2 left over. The % operation gives the remainder after doing the division. So `14/4` evaluates to 3, and `14 % 4` evaluates to 2, because 14 divided by 4 is 3 with 2 left over.

The % operator gets the remainder after division

The % operator has more applications than you might at first suspect. It allows your program to count by 2s, 3s, or any other number. For example, if you want to perform a certain action only on even integers, you need to know whether a given integer is even or odd. An integer `n` is even if `n % 2` is equal to 0, and the integer is odd if `n % 2` is equal to 1. Similarly, if you want your program to do something with every third integer, you can have it step through all the integers, using an `int` variable `n` to store each one. Your program would then perform the action only when `n % 3` is equal to 0.

FAQ How does the % operator behave with floating-point numbers?

The remainder operator is usually used with integer operands, but Java does allow you to use it with floating-point operands. If *n* and *d* are floating-point numbers, *n* % *d* equals *n* – (*d* **q*), where *q* is the integer portion of *n* /*d*. Note that the sign of *q* is the same as the sign of *n* / *d*. For example, `6.5 % 2.0` is `0.5`, `–6.5 % 2.0` is `–0.5`, and `6.5 % -2.0` is `0.5`.

Finally, notice that sometimes we use + and - as the sign of a number, while at other times + and − indicate addition and subtraction. In fact, Java always treats + and − as operators. A **unary operator** is an operator that has only one operand (one thing that it applies to), such as the operator − in the assignment statement

```
bankBalance = -cost;
```

A **binary operator** has two operands, such as the operators + and * in

```
total = cost + (tax * discount);
```

Note that the same operator symbol can sometimes be used as both a unary operator and a binary operator. For example, the − and + symbols can serve as either binary or unary operators.

FAQ Do spaces matter in an arithmetic expression?

The spaces in *any* Java statement usually do not matter. The only exception is when spaces appear within a pair of double quotes or single quotes. However, adding spaces in other situations can make a Java statement easier to read. For example, you should add a space both before and after any binary operator, as we have done in our examples.

Parentheses and Precedence Rules

Parentheses can be used to group items in an arithmetic expression in the same way that you use parentheses in algebra and arithmetic. With the aid of parentheses, you can tell the computer which operations to perform first, second, and so forth. For example, consider the following two expressions that differ only in the positioning of their parentheses:

```
(cost + tax) * discount
cost + (tax * discount)
```

To evaluate the first expression, the computer first adds `cost` and `tax` and then multiplies the result by `discount`. To evaluate the second expression, it multiplies `tax` and `discount` and then adds the result to `cost`. If you evaluate these expressions, using some numbers for the values of the variables, you will see that they produce different results.

If you omit the parentheses, the computer will still evaluate the expression. For example, the following assignment statement

```
total = cost + tax * discount;
```

is equivalent to

```
total = cost + (tax * discount);
```

When parentheses are omitted, the computer performs multiplication before addition. More generally, when the order of operations is not determined by parentheses, the computer will perform the operations in an order specified by

FIGURE 2.2 Precedence Rules

Highest Precedence

First: the unary operators `+`, `-`, `!`, `++`, and `--`

Second: the binary arithmetic operators `*`, `/`, and `%`

Third: the binary arithmetic operators `+` and `–`

Lowest Precedence

Precedence rules and parentheses determine the order of operations

the **precedence rules** shown in Figure 2.2.[3] Operators that are higher on the list are said to have **higher precedence.** When the computer is deciding which of two operations to perform first and the order is not dictated by parentheses, it begins with the operation having higher precedence and then performs the one having lower precedence. Some operators have equal precedence, in which case the order of operations is determined by where the operators appear in the expression. Binary operators of equal precedence are performed in left-to-right order. Unary operators of equal precedence are performed in right-to-left order.

These precedence rules are similar to the rules used in algebra. Except for some very standard cases, it is best to include the parentheses, even if the intended order of operations is the one indicated by the precedence rules, because the parentheses can make the expression clearer to a person reading the program code. Too many unnecessary parentheses can have the opposite effect, however. One standard case in which it is normal to omit parentheses is a multiplication within an addition. Thus,

```
balance = balance + (interestRate * balance);
```

would usually be written

```
balance = balance + interestRate * balance;
```

VideoNote
Writing arithmetic expressions and statements

Both forms are acceptable, and the two forms have the same meaning.

Figure 2.3 shows some examples of how to write arithmetic expressions in Java and indicates in color some of the parentheses that you can normally omit.

Specialized Assignment Operators

You can precede the simple assignment operator (=) with an arithmetic operator, such as +, to produce a kind of special-purpose assignment operator.

[3] Figure 2.2 shows all the operators we will use in this chapter. More precedence rules will be given in Chapter 3.

FIGURE 2.3 Some Arithmetic Expressions in Java

Ordinary Math	Java (Preferred Form)	Java (Parenthesized)
$rate^2 + delta$	`rate * rate + delta`	`(rate * rate) + delta`
$2(salary + bonus)$	`2 * (salary + bonus)`	`2 * (salary + bonus)`
$\frac{1}{time + 3mass}$	`1 / (time + 3 * mass)`	`1 / (time + (3 * mass))`
$\frac{a - 7}{t + 9v}$	`(a - 7) / (t + 9 * v)`	`(a - 7) / (t + (9 * v))`

For example, the following will increase the value of the variable amount by 5:

```
amount += 5;
```

This statement is really just shorthand for

```
amount = amount + 5;
```

This is hardly a big deal, but it can sometimes be handy.

You can combine an arithmetic operator with = as a shorthand notation

You can do the same thing with any of the other arithmetic operators, −, *, /, and %. For example, you could replace the line

```
amount = amount * 25;
```

with

```
amount *= 25;
```

Although you might not care to use these special assignment operators right now, many Java programmers do use them, so you are likely to encounter them in other programmers' code.

SELF-TEST QUESTIONS

13. What is the output produced by the following lines of program code?

```
int quotient = 7 / 3;
int remainder = 7 % 3;
System.out.println("quotient = " + quotient);
System.out.println("remainder = " + remainder);
```

14. What is the output produced by the following lines of program code?

```
double result = (1 / 2) * 2;
System.out.println("(1 / 2) * 2 equals " + result);
```

15. What is the output produced by the following code?

```
int result = 3 * 7 % 3 - 4 - 6;
System.out.println("result is " + result);
```

16. What is the output produced by the following code?

```
int result = 11;
result /= 2;
System.out.println("resu lt is " + result);
```

CASE STUDY Vending Machine Change

Vending machines often have small computers to control their operation. In this case study, we will write a program that handles one of the tasks that such a computer would need to perform. The input and output will be performed via the keyboard and screen. To integrate this program into a vending machine computer, you would have to embed the code into a larger program that takes its data from some place other than the keyboard and sends its results to some place other than the screen, but that's another story. In this case study, the user enters an amount of change from 1 to 99 cents. The program responds by telling the user one combination of coins that equals that amount of change.

Specify the task

For example, if the user enters 55 for 55 cents, the program tells the user that 55 cents can be given as two quarters and one nickel—that is, two 25-cent coins and one 5-cent coin. Suppose we decide that the interaction between the user and the program should be like this:

Write down a sample dialogue with the user

```
Enter a whole number from 1 to 99.
I will find a combination of coins
that equals that amount of change.
87
87 cents in coins:
3 quarters
1 dime
0 nickels and
2 pennies
```

Actually writing a sample dialogue, as we did here, before coding the program will help us to solve our problem.

The program will need variables to store the amount of change and the number of each type of coin. So it will need at least the following variables:

```
int amount, quarters, dimes, nickels, pennies;
```

That takes care of some routine matters; now we are ready to tackle the heart of the problem.

Pseudocode (first try)

We need an algorithm to compute the number of each kind of coin. Suppose we come up with the following pseudocode:

Algorithm to compute the number of coins in amount cents:

```
1. Read the amount into the variable amount.
2. Set the variable quarters equal to the maximum number of quarters in amount.
3. Reset amount to the change left after giving out that many quarters.
4. Set the variable dimes equal to the maximum number of dimes in amount.
5. Reset amount to the change left after giving out that many dimes.
6. Set the variable nickels equal to the maximum number of nickels in amount.
7. Reset amount to the change left after giving out that many nickels.
8. pennies = amount;
9. Display the original amount and the numbers of each coin.
```

These steps look reasonable, but before we rush into code, let's try an example. If we have 87 cents, we set `amount` to 87. How many quarters are in 87? Three, so `quarters` becomes 3, and we have 87 – 3 * 25, or 12, cents left in `amount`. We extract one dime and are left with 2 cents in `amount`. Thus, `dimes` is 1, `nickels` is 0, and `pennies` is 2.

Let's display our results. How much did we have originally? We look in `amount` and find 2. What happened to our 87? The algorithm changes the value of `amount`, but we need the original amount at the end so that we can display it. To fix the algorithm, we could either display the original value in `amount` before we change it or copy the original value into one more variable—called `originalAmount`. If we choose the latter approach, we can modify our pseudocode as follows:

Pseudocode (revised)

Algorithm to compute the number of coins in `amount` ***cents:***

```
1. Read the amount into the variable amount.
2. originalAmount = amount;
3. Set the variable quarters equal to the maximum number of quarters in amount.
4. Reset amount to the change left after giving out that many quarters.
5. Set the variable dimes equal to the maximum number of dimes in amount.
6. Reset amount to the change left after giving out that many dimes.
7. Set the variable nickels equal to the maximum number of nickels in amount.
8. Reset amount to the change left after giving out that many nickels.
9. pennies = amount;
10. Display originalAmount and the numbers of each coin.
```

We now need to produce Java code that does the same thing as our pseudocode. Much of it is routine. The first line of pseudocode simply calls for

Convert the pseudocode to Java code

prompting the user and then reading input from the keyboard. The following Java code corresponds to this first line of pseudocode:

```
System.out.println("Enter a whole number from 1 to 99.");
System.out.println("I will find a combination of coins");
System.out.println("that equals that amount of change.");

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

The next line of pseudocode, which sets the value of `originalAmount`, is already Java code, so you need not do any translating.

Thus far, the main part of our program reads as follows:

```
public static void main(String[] args)
{
    int amount, originalAmount,
       quarters, dimes, nickels, pennies;
    System.out.println("Enter a whole number from 1 to 99.");
    System.out.println("I will find a combination of coins");
    System.out.println("that equals that amount of change.");

    Scanner keyboard = new Scanner(System.in);
    amount = keyboard.nextInt();
    originalAmount = amount;
```

Next, we need to translate the following to Java code:

3. Set the variable `quarters` equal to the maximum number of quarters in `amount`.
4. Reset `amount` to the change left after giving out that many quarters.

Let's think. In our earlier example, we had 87 cents. To get the number of quarters in 87 cents, we see how many times 25 goes into 87. That is, we divide 87 by 25 to get 3 with 12 left over. So `quarters` is 3. Since the remainder, 12, is less than 25, we don't have four quarters in 87 cents, only three. Ah! We realize that we can use the operators / and % for this kind of division. For example,

Use / and % with integers to solve our problem

`87 / 25` is 3 (the maximum number of 25s in 87)
`87 % 25` is 12 (the remainder)

Using `amount` instead of 87 gives us the following two statements:

```
quarters = amount / 25;
amount = amount % 25;
```

We can treat dimes and nickels in a similar way, so we get the following code:

```
dimes = amount / 10;
amount = amount % 10;
nickels = amount / 5;
amount = amount % 5;
```

The rest of the program coding is straightforward. Our final program is shown in Listing 2.3.

LISTING 2.3 A Change-Making Program

```
import java.util.Scanner;

public class ChangeMaker
{
   public static void main(String[] args)
   {
      int amount, originalAmount,
         quarters, dimes, nickels, pennies;

      System.out.println("Enter a whole number from 1 to 99.");
      System.out.println("I will find a combination of coins");
      System.out.println("that equals that amount of change.");

      Scanner keyboard = new Scanner(System.in);
      amount = keyboard.nextInt();

      originalAmount = amount;
      quarters = amount / 25;
      amount = amount % 25;
      dimes = amount / 10;
      amount = amount % 10;
      nickels = amount / 5;
      amount = amount % 5;
      pennies = amount;

      System.out.println(originalAmount +
                         " cents in coins can be given as:");
      System.out.println(quarters + " quarters");
      System.out.println(dimes + " dimes");
      System.out.println(nickels + " nickels and");
      System.out.println(pennies + " pennies");
   }
}
```

25 goes into 87 three times with 12 left over.
`87 / 25` *is 3.*
`87 % 25` *is 12.*
87 cents is three quarters with 12 cents left over.

Sample Screen Output

```
Enter a whole number from 1 to 99.
I will find a combination of coins
that equals that amount of change.
87
87 cents in coins can be given as:
3 quarters
1 dimes
0 nickels and
2 pennies
```

Test your program using data that produces as many different kinds of results as you can think of

After writing a program, you need to test it on a number of different kinds of data. For our program, we should try data values that give zero for all possible coins, as well as other data, such as 25 and 26 cents, that produce results ranging from all quarters to quarters and another coin. For example, we could test our program on each of the following inputs: 0, 4, 5, 6, 10, 11, 25, 26, 35, 55, 65, 75, 76, and a number of other cases.

Although all our tests will be successful, the output does not exactly use correct grammar. For example, an input of 26 cents produces the output

```
26 cents in coins:
1 quarters
0 dimes
0 nickels and
1 pennies
```

The numbers are correct, but the labels would be better if they said `1 quarter` instead of `1 quarters` and `1 penny` instead of `1 pennies`. The techniques you need to produce this nicer-looking output will be presented in the next chapter. For now, let's end this project here. The output is correct and understandable.

PROGRAMMING TIP The Basic Structure of a Program

Many application programs, including the one we just wrote in the previous case study, have a similar basic structure. Their fundamental steps are like the advice once given to public speakers by Dale Carnegie (1888–1955): "Tell the audience what you're going to say, say it; then tell them what you've said." Programs often take the following steps:

1. **P**repare: Declare variables and explain the program to the user.
2. **I**nput: Prompt for and get input from the user.
3. **P**rocess: Perform the task at hand.
4. **O**utput: Display the results.

This structure, which we can abbreviate as PIPO, is particularly true of our initial programs. Keeping these steps in mind can help you to organize your thoughts as you design and code your programs.

SELF-TEST QUESTION

17. Consider the following statement from the program in Listing 2.3:

```
System.out.println(originalAmount +
                   " cents in coins can be given as:");
```

Suppose that you replaced the preceding line with the following:

```
System.out.println(amount +
                       "cents in coins can be given as:");
```

How would this change the sample output given in Listing 2.3?

Increment and Decrement Operators

Java has two special operators that are used to increase or decrease the value of a variable by 1. Since these operators are so specialized, both you and Java could easily get along without them. But they are sometimes handy, and they are of cultural significance because programmers use them. So to be "in the club," you should know how they work, even if you do not want to use them in your own programs.

The **increment operator** is written as two plus signs (++). For example, the following will increase the value of the variable count by 1:

Increment operator ++

```
count++;
```

This is a Java statement. If the variable count has the value 5 before this statement is executed, it will have the value 6 after the statement is executed. The statement is equivalent to

```
count = count + 1;
```

Decrement operator – –

The **decrement operator** is similar, except that it subtracts 1 rather than adds 1 to the value of the variable. The decrement operator is written as two minus signs (--). For example, the following will decrease the value of the variable count by 1:

```
count--;
```

If the variable count has the value 5 before this statement is executed, it will have the value 4 after the statement is executed. The statement is equivalent to

```
count = count - 1;
```

You can use the increment operator and the decrement operator with variables of any numeric type, but they are used most often with variables of integer types, such as the type int.

As you can see, the increment and decrement operators are really very specialized. Why does Java have such specialized operators? It inherited them from C++, and C++ inherited them from C. In fact, this increment operator is where the ++ came from in the name of the C++ programming language. Why was it added to the C and C++ languages? Because adding or subtracting 1 is a very common thing to do when programming.

More About the Increment and Decrement Operators

Although we do not recommend doing so, the increment and decrement operators can be used in expressions. When used in an expression, these operators both change the value of the variable they are applied to and **return,** or produce, a value.

Be careful if you use the operators ++ and -- in expressions

In expressions, you can place the ++ or -- either before or after the variable, but the meaning is different, depending on where the operator is placed. For example, consider the code

```
int n = 3;
int m = 4;
int result = n * (++m);
```

After this code is executed, the value of n is unchanged at 3, the value of m is 5, and the value of result is 15. Thus, ++m both changes the value of m and returns that changed value to be used in the arithmetic expression.

In the previous example, we placed the increment operator before the variable. If we place it after the variable m, something slightly different happens. Consider the code

```
int n = 3;
int m = 4;
int result = n * (m++);
```

In this case, after the code is executed, the value of n is 3 and the value of m is 5, just as in the previous case, but the value of result is 12, not 15. What is the story?

The two expressions n*(++m) and n*(m++) both increase the value of m by 1, but the first expression increases the value of m *before* it does the multiplication, whereas the second expression increases the value of m *after* it does the multiplication. Both ++m and m++ have the same effect on the final value of m, but when you use them as part of an arithmetic expression, they give a different value to the expression.

The -- operator works in the same way when it is used in an arithmetic expression. Both --m and m-- have the same effect on the final value of m, but when you use them as part of an arithmetic expression, they give a different value to the expression. For --m, the value of m is decreased *before* its value is used in the expression, but for m--, the value of m is decreased *after* its value is used in the expression.

When an increment or decrement operator is placed before a variable, we get what is called the **prefix form.** When it is placed after a variable, we get a **postfix form.** Note that the increment and decrement operators can be applied only to variables. They cannot be applied to constants or to more complicated arithmetic expressions.

SELF-TEST QUESTION

18. What output is produced by the following lines of program code?

```
int n = 2;
n++;
System.out.println("n is " + n);
n—;
System.out.println("n is " + n);
```

2.2 THE CLASS String

Words, words, mere words, no matter from the heart.

—WILLIAM SHAKESPEARE, *Troilus and Cressida*

Strings of characters, such as `"Enter the amount:"`, are treated slightly differently than values of the primitive types. Java has no primitive type for strings. However, Java supplies a class called `String` that can be used to create and process strings of characters. In this section, we introduce you to the class `String`.

Classes are central to Java, and you will soon define and use your own classes. However, this discussion of the class `String` gives us an opportunity to review some of the notation and terminology used for classes that Chapter 1 introduced.

The class `String` comes with Java

String Constants and Variables

You have already been using constants of type `String`. The quoted string

```
"Enter a whole number from 1 to 99."
```

which appears in the following statement from the program in Listing 2.3, is a string constant:

```
System.out.println("Enter a whole number from 1 to 99.");
```

A value of type `String` is one of these quoted strings. That is, a value of type `String` is a sequence of characters treated as a single item. A variable of type `String` can name one of these string values.

The following declares `greeting` to be the name for a `String` variable:

```
String greeting;
```

The next statement sets the value of `greeting` to the `String` value `"Hello!"`:

```
greeting = "Hello!";
```

These two statements are often combined into one, as follows:

```
String greeting = "Hello!";
```

Once a `String` variable, such as `greeting`, has been given a value, you can display it on the screen as follows:

```
System.out.println(greeting);
```

This statement displays

```
Hello!
```

assuming that the value of `greeting` has been set as we just described.

A string can have any number of characters. For example, `"Hello"` has five characters. A string can even have zero characters. Such a string is called the **empty string** and is written as a pair of adjacent double quotes, like so:`""`. You will encounter the empty string more often than you might think. Note that the string `" "` is not empty: It consists of one blank character.

Concatenation of Strings

You can connect—or join or paste—two strings together to obtain a larger string. This operation is called **concatenation** and is performed by using the + operator. When this operator is used with strings, it is sometimes called the **concatenation operator.** For example, consider the following code:

You can use + to join, or concatenate, strings together

```
String greeting, sentence;
greeting = "Hello";

sentence = greeting + "my friend";
System.out.println(sentence);
```

This code sets the variable `sentence` to the string `"Hellomy friend"` and will write the following on the screen:

```
Hellomy friend
```

Notice that no spaces are added when you concatenate two strings by means of the + operator. If you wanted `sentence` set to `"Hello my friend"`, you could change the assignment statement to

```
sentence = greeting + " my friend";
```

Notice the space before the word *my*.

You can concatenate any number of `String` objects by using the + operator. You can even use + to connect a `String` object to any other type of object. The result is always a `String` object. Java will figure out some way to express any object as a string when you connect it to a string using the + operator. For simple things like numbers, Java does the obvious thing. For example,

```
String solution = "The answer is " + 42;
```

will set the `String` variable `solution` to the string `"The answer is 42"`. This is so natural that it may seem as though nothing special is happening, but a conversion from one type to another does occur. The constant 42 is a number—not even an object—whereas `"42"` is a `String` object consisting of the character `'4'` followed by the character `'2'`. Java converts the number constant 42 to the string constant `"42"` and then concatenates the two strings `"The answer is"` and `"42"` to obtain the longer string `"The answer is 42"`.

RECAP Using the + Symbol with Strings

You can concatenate two strings by connecting them with the + operator.

EXAMPLE

```
String name = "Chiana";
String greeting = "Hi " + name;
System.out.println(greeting);
```

This sets `greeting` to the string `"Hi Chiana"` and then displays the following on the screen:

```
Hi Chiana
```

Note that we added a space at the end of `"Hi"` to separate the words in the output.

String Methods

A `String` variable is not a simple variable, as a variable of type `int` is. A `String` variable is a variable of a class type that names a `String` object. Recall that an object has methods as well as data. For example, objects of the class `String` store data consisting of strings of characters, such as `"Hello"`. The methods provided by the class `String` can be used to manipulate this data.

Most of the `String` methods return some value. For example, the method `length` returns the number of characters in a `String` object. So `"Hello".length( )` returns the integer 5. That is, the value of `"Hello".length( )` is 5, which we can store in an `int` variable as follows:

Call the method `length` to get the length of a string

```
int n = "Hello".length( );
```

As you have learned, you call, or invoke, a method into action by writing the name of the object, followed by a dot, followed by the method name, and ending with parentheses. Although the object can be a constant, such as `"Hello"`, it is more common to use a variable that names the object, as illustrated by the following:

```
String greeting = "Hello";
int n = greeting.length( );
```

For some methods—such as `length`—no arguments are needed, so the parentheses are empty. For other methods, as you will see soon, some information must be provided inside the parentheses.

All objects of a class have the same methods, but each object can have different data. For example, the two `String` objects `"Hello"` and `"Good-bye"` have different data—that is, different strings of characters. However, they have the same methods. Thus, since we know that the `String` object `"Hello"` has the method `length`, we know that the `String` object `"Good-bye"` must also have the method `length`. All string objects have this method.

Spaces, special symbols, and repeated characters are all counted when computing the length of a string. For example, suppose we declare `String` variables as follows:

```
String command = "Sit Fido!";
String answer = "bow-wow";
```

Then `command.length( )` returns 9 and `answer.length( )` returns 7.

You can use a call to the method `length` anywhere that you can use a value of type `int`. For example, all of the following are legal Java statements:

Positions, or indices, in a string begin with 0

```
int count = command.length( );
System.out.println("Length is " + command.length( ));
count = command.length( ) + 3;
```

Many of the methods for the class `String` depend on counting **positions** in the string. Positions in a string begin with 0, not with 1. In the string `"Hi Mom"`, `'H'` is in position 0, `'i'` is in position 1, the blank character is in position 2, and so forth. A position is usually referred to as an **index** in computer parlance. So it would be more normal to say that `'H'` is at index 0, `'i'` is at index 1, and so on. Figure 2.4 illustrates how index positions are numbered in a string. The 12 characters in the string `"Java is fun."` have indices 0 through 11. The index of each character is shown above it in the figure.

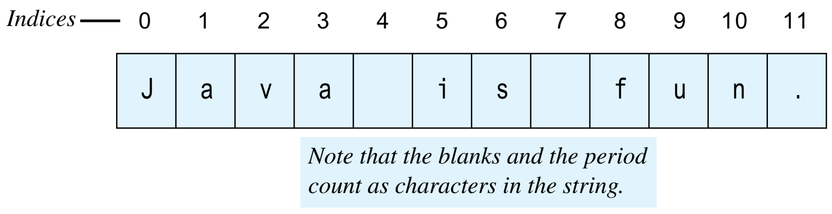
A **substring** is simply a portion of a string. For example, the string defined by

A substring is a portion of a string

```
String phrase = "Java is fun.";
```

has a substring `"fun"` that begins at index 8. The method `indexOf` returns the index of a substring given as its argument. The invocation `phrase.indexOf("fun")` will return 8 because the `'f'` in `"fun"` is at index 8. If the

FIGURE 2.4 String Indices


Indices
0
1
2
3
4
5
6
7
8
9
10
11
J
a
v
a
i
s
f
u
n
.
Note that the blanks and the period count as characters in the string.

substring occurs more than once in a string, `indexOf` returns the index of the first occurrence of its substring argument.

Figure 2.5 describes some of the methods for the class `String`. In the next chapter, you will learn how to make use of methods such as `equals` and `compareTo` that compare strings. Other methods listed in this figure may become more useful to you in subsequent chapters. You can learn about additional `String` methods by consulting the documentation for the Java Class Library on the Oracle Web site.

FAQ When calling a method, we write the name of an object and a dot before the method's name. What term do we use to refer to this object?

An object has methods. When you invoke one of these methods, the object receives the call and performs the actions of the method. Thus, the object is known as a **receiving object,** or **receiver.** Documentation, such as that in Figure 2.5, often describes the receiving object simply as **this object.**

FAQ What is whitespace?

Any characters that are not visible when displayed are collectively known as whitespace. Such characters include blanks, tabs, and new-line characters.

String Processing

Technically, objects of type `String` cannot be changed. Notice that none of the methods in Figure 2.5 changes the value of a `String` object. The class `String` has more methods than those shown in Figure 2.5, but none of them lets you write statements that do things like "Change the fifth character in this

FIGURE 2.5 Some Methods in the Class `String`

Method	Return Type	Example for `String s = "Java";`	Description
`charAt` (*index*)	char	`c = s.charAt(2);` `// c='v'`	Returns the character at *index* in the string. Index numbers begin at 0.
`compareTo` (*a_string*)	int	`i = s.compareTo("C++");` `// i is positive`	Compares this string with *a_string* to see which comes first in lexicographic (alphabetic, with upper- before lowercase) ordering. Returns a negative integer if this string is first, zero if the two strings are equal, and a positive integer if *a_string* is first.
`concat` (*a_string*)	`String`	`s2 = s.concat("rocks");` `// s2 = "Javarocks"`	Returns a new string with this string concatenated with *a_string*. You can use the + operator instead.
`equals` (*a_string*)	boolean	`b = s.equals("Java");` `// b = true`	Returns true if this string and *a_string* are equal. Otherwise returns false.
`equals IgnoreCase` (*a_string*)	boolean	`b = s.equals("java");` `// b = true`	Returns true if this string and *a_string* are equal, considering upper- and lowercase versions of a letter to be the same. Otherwise returns false.
`indexOf` (*a_string*)	int	`i = s.indexOf("va");` `// i = 2`	Returns the index of the first occurrence of the substring *a_string* within this string or –1 if *a_string* is not found. Index numbers begin at 0.
`lastIndexOf` (*a_string*)	int	`i = s.lastIndexOf("a");` `// i = 3`	Returns the index of the last occurrence of the substring *a_string* within this string or –1 if *a_string* is not found. Index numbers begin at 0.
`length()`	int	`i = s.length();` `// i = 4`	Returns the length of this string.
`toLower Case()`	`String`	`s2 = s.toLowerCase();` `// s2 = "java"`	Returns a new string having the same characters as this string, but with any uppercase letters converted to lowercase. This string is unchanged.
`toUpper Case()`	`String`	`s2 = s.toUpperCase();` `// s2 = "JAVA"`	Returns a new string having the same characters as this string, but with any lowercase letters converted to uppercase. This string is unchanged.
`replace` (*oldchar, newchar*)	`String`	`s2 = s.replace('a','o');` `// s2 = "Jovo";`	Returns a new string having the same characters as this string, but with each occurrence of *oldchar* replaced by *newchar*.
`substring` (*start*)	`String`	`s2 = s.substring(2);` `// s2 = "va";`	Returns a new string having the same characters as the substring that begins at index *start* through to the end of the string. Index numbers begin at 0.
`substring` (*start,end*)	`String`	`s2 = s.substring(1,3);` `// s2 = "av";`	Returns a new string having the same characters as the substring that begins at index *start* through to but not including the character at index *end*. Index numbers begin at 0.
`trim( )`	`String`	`s = "   Java     ";` `s2 = s.trim();` `// s2 = "Java"`	Returns a new string having the same characters as this string, but with leading and trailing whitespace removed.

string to 'z'". This was done intentionally to make the implementation of the String class more efficient—that is, to make the methods execute faster and use less computer memory. Java has another string class, StringBuilder, that has methods for altering its objects. But we will not discuss this class here because we do not need it.

Although you cannot change the value of a String object, such as "Hello", you can still write programs that change the value of a String *variable*, which is probably all you want to do anyway. To make the change, you simply use an assignment statement, as in the following example:

```
String name = "D'Aargo";
name = "Ka " + name;
```

The assignment statement in the second line changes the value of the name variable from "D' Aargo" to "Ka D'Aargo". Listing 2.4 shows a sample program that performs some simple string processing and changes the value of a String variable. The backslash that appears within the argument to the println method is explained in the next section.

VideoNote
Processing strings

LISTING 2.4 Using the String Class

The meaning of \" is discussed in the section entitled "Escape Characters."

```
public class StringDemo
{
    public static void main(String[] args)
    {
        String sentence = "Text processing is hard!";
        int position = sentence.indexOf("hard");
        System.out.println(sentence);
        System.out.println("012345678901234567890123");
        System.out.println("The word \"hard\" starts at index "
                           + position);
        sentence = sentence.substring(0, position) + "easy!";
        sentence = sentence.toUpperCase();
        System.out.println("The changed string is:");
        System.out.println(sentence);
    }
}
```

Screen Output

```
Text processing is hard!
012345678901234567890123
The word "hard" starts at index 19
The changed string is:
TEXT PROCESSING IS EASY!
```

GOTCHA String Index Out of Bounds

The first character in a string is at index 0, not 1. So if a string contains n characters, the last character is at index $n - 1$. Whenever you call a string method—such as `charAt`—that takes an index as an argument, the value of that index must be valid. That is, the index value must be greater than or equal to zero and less than the length of the string. An index value outside of this range is said to be **out of bounds** or simply **invalid.** Such an index will cause a run-time error. ■

Escape Characters

Suppose you want to display a string that contains quotation marks. For example, suppose you want to display the following on the screen:

Quotes within quotes

```
The word "Java" names a language, not just a drink!
```

The following statement will not work:

```
System.out.println("The word "Java" names a language, " +
                   "not just a drink!");
```

This will produce a compiler error message. The problem is that the compiler sees

```
"The word "
```

as a perfectly valid quoted string. Then the compiler sees `Java"`, which is not anything valid in the Java language (although the compiler might guess that it is a quoted string with one missing quote or guess that you forgot a + sign). The compiler has no way to know that you mean to include the quotation marks as part of the quoted string, unless you tell it that you mean to do so. You tell the compiler that you mean to include the quote in the string by placing a backslash (`\`) before the troublesome character, like so:

```
System.out.println("The word \"Java\" names a language, " +
                   "not just a drink!");
```

Figure 2.6 lists some other special characters that are indicated with a backslash. These are often called **escape sequences** or **escape characters,** because they escape from the usual meaning of a character, such as the usual meaning of the double quote.

It is important to note that each escape sequence represents one character, even though it is written as two symbols. So the string `"\"Hi\""` contains four characters—a quote, H, i, and another quote—not six characters. This point is significant when you are dealing with index positions.

An escape character requires two symbols to represent it

Including a backslash in a quoted string is a little tricky. For example, the string `"abc\def"` is likely to produce the error message "Invalid escape character." To include a backslash in a string, you need to use two backslashes. The string `"abc\\def"`, if displayed on the screen, would produce

```
abc\def
```

FIGURE 2.6 **Escape Characters**

`\"`	Double quote.
`\'`	Single quote.
`\\`	Backslash.
`\n`	New line. Go to the beginning of the next line.
`\r`	Carriage return. Go to the beginning of the current line.
`\t`	Tab. Add whitespace up to the next tab stop.

The escape sequence `\n` indicates that the string starts a new line at the `\n`. For example, the statement

```
System.out.println("The motto is\nGo for it!");
```

will write the following two lines to the screen:

```
The motto is
Go for it!
```

Including a single quote inside a quoted string, such as `"How's this"`, is perfectly valid. But if you want to define a single quote as a constant, you need to use the escape character `\'`, as in

```
char singleQuote = '\";
```

The Unicode Character Set

A **character set** is a list of characters, each associated with a standard number. The **ASCII** character set includes all the characters normally used on an English-language keyboard. (ASCII stands for American Standard Code for Information Interchange.) Each character in ASCII is represented as a 1-byte binary number. This encoding provides for up to 256 characters. Many programming languages other than Java use the ASCII character set.

The **Unicode** character set includes the entire ASCII character set, plus many of the characters used in languages other than English. A Unicode character occupies 2 bytes. This encoding provides more than 65,000 different characters.

Unicode includes ASCII as a subset

To appeal to international users, the developers of Java adopted the Unicode character set. As it turns out, this is not likely to be a big issue if you are using an English-language keyboard. Normally, you can just program as though Java were using the ASCII character set, because the ASCII character set is a subset of the Unicode character set. The advantage of the Unicode character set is that it makes it easy to handle languages other than English. The disadvantage is that it sometimes requires more computer memory to store each character than it would if Java used only the ASCII character set. You can see the subset of the Unicode character set that is synonymous with the ASCII character set in the appendix of this book.

SELF-TEST QUESTIONS

19. What output is produced by the following statements?

```
String greeting = "How do you do";
System.out.println(greeting + "Seven of Nine.");
```

20. What output is produced by the following statements?

```
String test = "abcdefg";
System.out.println(test.length());
System.out.println(test.charAt(1));
```

21. What output is produced by the following statements?

```
String test = "abcdefg";
System.out.println(test.substring(3));
```

22. What output is produced by the following statement?

```
System.out.println("abc\ndef");
```

23. What output is produced by the following statement?

```
System.out.println("abc\\ndef");
```

24. What output is produced by the following statements?

```
String test = "Hello John";
test = test.toUpperCase();
System.out.println(test);
```

25. What is the value of the expression `s1.equals(s2)` after the following statements execute?

```
String s1 = "Hello John";
String s2 = "hello john";
```

26. What is the value of the expression `s1.equals(s2)` after the following statements execute?

```
String s1 = "Hello John";
String s2 = "hello john";
s1 = s1.toUpperCase();
s2 = s2.toUpperCase();
```

2.3 KEYBOARD AND SCREEN I/O

Garbage in, garbage out.

—PROGRAMMER'S SAYING

Input and output of program data are usually referred to as **I/O.** A Java program can perform I/O in many different ways. This section presents some very simple ways to handle text as either input typed at the keyboard or output sent to the screen. In later chapters, we will discuss more elaborate ways to do I/O.

Screen Output

We have been using simple output statements since the beginning of this book. This section will summarize and explain what we have already been doing. In Listing 2.3, we used statements such as the following to send output to the display screen:

```
System.out.println("Enter a whole number from 1 to 99.");
            . . .
System.out.println(quarters + "quarters");
```

Earlier in this chapter, we noted that `System` is a standard class, `out` is a special object within that class, and `out` has `println` as one of its methods. Of course, you need not be aware of these details in order to use these output statements. You can simply consider `System.out.println` to be one peculiarly spelled statement. However, you may as well get used to this dot notation and the notion of methods and objects.

To use output statements of this form, simply follow the expression `System.out.println` with what you want to display, enclosed in parentheses, and then follow that with a semicolon. You can output strings of text in double quotes, like `"Enter a whole number from 1 to 99"` or `"quarters"`; variables, like `quarters`; numbers, like `5` or `7.3`; and almost any other object or value. If you want to display more than one thing, simply place a + between the things you want to display. For example,

Use + to join the things you want to display

```
System.out.println("Lucky number = " + 13 +
                   "Secret number = " + number);
```

If the value of `number` is 7, the output will be

```
Lucky number = 13Secret number = 7
```

Notice that no spaces are added. If you want a space between the `13` and the word `Secret` in the preceding output—and you probably do—you should add a space at the beginning of the string

```
"Secret number = "
```

so that it becomes

```
"Secret number = "
```

Notice that you use double quotes, not single quotes, and that the left and right quotes are the same symbol. Finally, notice that you can place the statement on several lines if it is too long. However, you cannot break a line in the middle of a variable name or a quoted string. For readability, you should break the line before or after a + operator, and you should indent the continuation line.

You can also use the `println` method to display the value of a `String` variable, as illustrated by the following:

```
String greeting = "Hello Programmers!";
System.out.println(greeting);
```

This will cause the following to be written on the screen:

```
Hello Programmers!
```

Every invocation of `println` ends a line of output. For example, consider the following statements:

```
System.out.println("One, two, buckle my shoe.");
System.out.println("Three, four, shut the door.");
```

These two statements will cause the following output to appear on the screen:

```
One, two, buckle my shoe.
Three, four, shut the door.
```

`print` versus `println`

If you want two or more output statements to place all of their output on a single line, use `print` instead of `println`. For example,

```
System.out.print("One, two,");
System.out.print(" buckle my shoe.");
System.out.print(" Three, four,");
System.out.println("shut the door.");
```

will produce the following output:

```
One, two, buckle my shoe. Three, four, shut the door.
```

Notice that a new line is not started until you use a `println` instead of a `print`. Notice also that the new line starts *after* the items specified in the `println` have been displayed. This is the only difference between `print` and `println`.

That is all you need to know in order to write programs with this sort of output, but we can still explain a bit more about what is happening. Consider the following statement:

```
System.out.println("The answer is " + 42);
```

The expression inside the parentheses should look familiar:

```
"The answer is " + 42
```

In our discussion of the class `String` in Section 2.2, we said that you could use the + operator to concatenate a string, such as `"The answer is"`, and another item, such as the number constant 42. The + operator within a `System.out.println` statement is the same + operator that performs string concatenation. In the preceding example, Java converts the number constant 42 to the string `"42"` and then uses the + operator to obtain the string `"The answer is 42"`. The `System.out.println` statement then displays this string. The `println` method always outputs strings. Technically speaking, it never outputs numbers, even though it looks as though it does.

RECAP `println`

You can display lines of text using the method `System.out.println`. The items of output can be quoted strings, variables, constants such as numbers, or almost any object you can define in Java.

SYNTAX

```
System.out.println (Output_1 + Output_2 + … + Output_Last);
```

EXAMPLES

```
System.out.println("Hello out there!");
System.out.println("Area = " + theArea + " square inches");
```

RECAP `println` **Versus** `print`

`System.out.println` and `System.out.print` are almost the same method. The `println` method advances to a new line *after* it displays its output, whereas the `print` method does not. For example,

```
System.out.print("one");
System.out.print("two");
System.out.println("three");
System.out.print("four");
```

produces the following output:

```
one two three
four
```

The output would look the same whether the last statement involved `print` or `println`. However, since our last statement uses `print`, subsequent output will appear on the same line as `four`.

Keyboard Input

As we mentioned earlier in this chapter, you can use the class Scanner for handling keyboard input. This standard class is in the package java.util. To make Scanner available to your program, you write the following line near the beginning of the file containing your program:

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

You use an object of the class Scanner to perform keyboard input. You create such an object by writing a statement in the following form:

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

where *Scanner_Object_Name* is any Java variable. For example, in Listing 2.3, we used the identifier keyboard for the *Scanner_Object_Name*, as follows:

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

We often use the identifier keyboard for our Scanner object because it suggests keyboard input. However, you may use other names instead. For example, you could use the variable scannerObject everywhere we use keyboard.

After you define a Scanner object, you can use methods of the class Scanner to read data typed at the keyboard. For example, the method invocation

nextInt reads an int value

```
keyboard.nextInt()
```

reads one int value typed at the keyboard and returns that int value. You can assign that value to a variable of type int, as follows:

```
int n1 = keyboard.nextInt();
```

nextDouble reads a double value

What if you want to read a number of some type other than int? The method nextDouble works in exactly the same way as nextInt, except that it reads a value of type double. Scanner has similar methods for reading values of other numeric types.

The method next reads a word, as illustrated by the following statements:

next reads a word

```
String s1 = keyboard.next();
String s2 = keyboard.next();
```

If the input line is

```
plastic spoons
```

the string "plastic" is assigned to s1 and the string "spoons" is assigned to s2. Note that any two input values entered at the keyboard must be separated by whitespace characters such as one or more blanks or one or more line breaks or some combination of blanks and line breaks. In this context, the separators are called **delimiters.** For the method next, a word is any string of nonwhitespace characters delimited by whitespace characters.

A delimiter is a separator in input data; by default, it is whitespace

If you want to read an entire line, you would use the method nextLine. For example,

```
String sentence = keyboard.nextLine();
```

reads in one line of input and places the resulting string into the variable `sentence`. The end of an input line is indicated by the escape character `'\n'`. When you press the Enter (Return) key at the keyboard, you enter the `'\n'` character. On the screen, however, you simply see one line end and another begin. When `nextLine` reads a line of text, it reads this `'\n'` character, but the `'\n'` does not become part of the string value returned. So in the previous example, the string named by the variable `sentence` does not end with the `'\n'` character.

`nextLine` reads an entire line

Listing 2.5 shows a program that demonstrates the `Scanner` methods that we just introduced.

LISTING 2.5 A Demonstration of Keyboard Input *(part 1 of 2)*

```
import java.util.Scanner;

public class ScannerDemo
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);

        System.out.println("Enter two whole numbers");
        System.out.println("separated by one or more spaces:");

        int n1, n2;
        n1 = keyboard.nextInt();
        n2 = keyboard.nextInt();
        System.out.println("You entered " + n1 + " and " + n2);

        System.out.println("Next enter two numbers.");
        System.out.println("A decimal point is OK.");

        double d1, d2;
        d1 = keyboard.nextDouble();
        d2 = keyboard.nextDouble();
        System.out.println("You entered " + d1 + " and " + d2);

        System.out.println("Next enter two words:");

        String s1, s2;
        s1 = keyboard.next();
        s2 = keyboard.next();
        System.out.println("You entered \"" +
                           s1 + "\" and \"" + s2 + "\"");

        s1 = keyboard.nextLine(); //To get rid of '\n'

        System.out.println("Next enter a line of text:");
        s1 = keyboard.nextLine();
        System.out.println("You entered: \"" + s1 + "\"");
    }
}
```

Gets the `Scanner` *class from the package (library)* `java.util`

Sets things up so the program can accept keyboard input

Reads one `int` *value from the keyboard*

Reads one `double` *value from the keyboard*

Reads one word from the keyboard

This line is explained in the next Gotcha section.

Reads an entire line

(continued)

LISTING 2.5 A Demonstration of Keyboard Input *(part 2 of 2)*

Sample Screen Output

```
Enter two whole numbers
separated by one or more spaces:
 42      43
You entered 42 and 43
Next enter two numbers.
A decimal point is OK.
 9.99  21
You entered 9.99 and 21.0
Next enter two words:
plastic spoons
You entered "plastic" and "spoons"
Next enter a line of text:
May the hair on your toes grow long and curly.
You entered "May the hair on your toes grow long and curly."
```

RECAP Keyboard Input Using the Class `Scanner`

You can use an object of the class `Scanner` to read input from the keyboard. To set things up, you place the following statement at the beginning of your program file:

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

You also need a statement in the following form before the first statement involving keyboard input:

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

where *Scanner_Object_Name* is any Java identifier that is not a keyword. For example,

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

The methods `nextInt`, `nextDouble`, and `next` read and return, respectively, a value of type `int`, a value of type `double`, and a word as the value of a `String` object. The method `nextLine` reads and returns the remainder of the current input line as a string. The terminating `'\n'` is read but not included in the string value returned.

(continued)

SYNTAX

```
Int_Variable= Scanner_Object_Name.nextInt();
Double_Variable= Scanner_Object_Name.nextDouble()
String_Variable= Scanner_Object_Name.next();
String_Variable= Scanner_Object_Name.nextLine();
```

EXAMPLES

```
int count = scannerObject.nextInt();
double distance = scannerObject.nextDouble();
String word = scannerObject.next();
String wholeLine = scannerObject.nextLine();
```

Figure 2.7 lists some other methods in the class Scanner.

REMEMBER Prompt for Input

Your program should always display a **prompt** when it needs the user to enter some data as input, as in the following example:

```
System.out.println("Enter a whole number:");
```

GOTCHA Problems with the Methods next and nextLine

VideoNote
Pitfalls involving nextLine()

The methods next and nextLine of the class Scanner read text starting wherever the last keyboard reading left off. For example, suppose you create an object of the class Scanner as follows:

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

and suppose you continue with the following code:

```
int n = keyboard.nextInt();
String s1 = keyboard.nextLine();
String s2 = keyboard.nextLine();
```

Finally, assume that the corresponding input is typed at the keyboard, as follows:

```
42 is the answer
and don't you
forget it.
```

This will set the value of the variable n to 42, the variable s1 to "is the answer", and the variable s2 to "and don't you".

FIGURE 2.7 Some Methods in the Class `Scanner`

Method for `Scanner kbd;`	Return Type	Description
`next()`	`String`	Returns the string value consisting of the next keyboard characters up to, but not including, the first delimiter character. The default delimiters are whitespace characters.
`nextLine()`	`String`	Reads the rest of the current keyboard input line and returns the characters read as a value of type `String`. Note that the line terminator `'\n'` is read and discarded; it is not included in the string returned.
`nextInt()`	int	Returns the next keyboard input as a value of type `int`.
`nextDouble()`	double	Returns the next keyboard input as a value of type `double`.
`nextFloat()`	float	Returns the next keyboard input as a value of type `float`.
`nextLong()`	long	Returns the next keyboard input as a value of type `long`.
`nextByte()`	byte	Returns the next keyboard input as a value of type `byte`.
`nextShort()`	short	Returns the next keyboard input as a value of type `short`.
`nextBoolean()`	boolean	Returns the next keyboard input as a value of type `boolean`. The values of `true` and `false` are entered as the words true and false. Any combination of uppercase and lowercase letters is allowed in spelling true and false.
`useDelimiter` (*Delimiter_Word*)	`Scanner`	Makes the string *Delimiter_Word* the only delimiter used to separate input. Only the exact word will be a delimiter. In particular, blanks, line breaks, and other whitespace will no longer be delimiters unless they are a part of *Delimiter_Word*. This is a simple case of the use of the `useDelimiter` method. There are many ways to set the delimiters to various combinations of characters and words, but we will not go into them in this book.

So far it may not seem as though there is any potential for problems, but suppose the input were instead

```
42
and don't you
forget it.
```

Under these circumstances, you might expect that `n` is set to 42, `s1` to `"and don't you"`, and `s2` to `"forget it"`. But that is not what happens.

Actually, the value of the variable `n` is set to 42, the variable `s1` is set to the empty string, and the variable `s2` is set to `"and don't you"`. The method `nextInt` reads the 42 but does not read the end-of-line character `'\n'`. So the first `nextLine` invocation reads the rest of the line that contains the 42. There is nothing more on that line, except for `'\n'`. Thus, `nextLine` reads and discards the end-of-line character `'\n'` and then returns the empty string, which is assigned to `s1`. The next invocation of `nextLine` begins on the next line and reads `"and don't you"`.

When combining methods that read numbers from the keyboard with methods that read strings, you sometimes have to include an extra invocation of `nextLine` to get rid of the end-of-line character `'\n'`. This problem is illustrated near the end of the program in Listing 2.5.

REMEMBER The Empty String

Recall that an empty string has zero characters and is written as `""`. If the `nextLine` method executes, and the user simply presses the Enter (Return) key, the `nextLine` method returns the empty string.

Other Input Delimiters (Optional)

When using the `Scanner` class for keyboard input, you can change the delimiters that separate keyboard input to almost any combination of characters and strings, but the details are a bit involved. In this book we will describe only one simple kind of delimiter change. We will tell you how to change the delimiters from whitespace to one specific delimiter string.

For example, suppose you create a `Scanner` object as follows:

```
Scanner keyboard2 = new Scanner(System.in);
```

You can change the delimiter for the object `keyboard2` to `"##"` as follows:

```
keyboard2.useDelimiter("##");
```

After this invocation of the `useDelimiter` method, `"##"` will be the *only* input delimiter for the input object `keyboard2`. Note that whitespace will no longer be a delimiter for keyboard input involving `keyboard2`. So given the keyboard input

```
funny wo##rd ##
```

the following code would read the two strings `"funny wo"` and `"rd "`:

```
System.out.println("Enter two words as a line of text:");
String s1 = keyboard2.next();
String s2 = keyboard2.next();
```

Note that no whitespace characters, not even line breaks, serve as an input delimiter once this change is made to `keyboard2`. Also note that you can have two different objects of the class `Scanner` with different delimiters in the same program. These points are illustrated by the program in Listing 2.6.

LISTING 2.6 Changing Delimiters *(Optional)*

```
import java.util.Scanner;

public class DelimitersDemo
{
    public static void main(String[] args)
    {
        Scanner keyboard1 = new Scanner(System.in);
        Scanner keyboard2 = new Scanner(System.in);
        keyboard2.useDelimiter("##");
        //The delimiters for keyboard1 are the whitespace
        //characters.
        //The only delimiter for keyboard2 is ##.

        String s1, s2;

        System.out.println("Enter a line of text with two words:");
        s1 = keyboard1.next();
        s2 = keyboard1.next();
        System.out.println("The two words are \"" + s1 +
                           "\" and \"" + s2 + "\"");

        System.out.println("Enter a line of text with two words");
        System.out.println("delimited by ##:");
        s1 = keyboard2.next();
        s2 = keyboard2.next();
        System.out.println("The two words are \"" + s1 +
                           "\" and \"" + s2 + "\"");
    }
}
```

`keyboard1` *and* `keyboard2` *have different delimiters.*

Sample Screen Output

```
Enter a line of text with two words:
funny wo##rd##
The two words are "funny" and "wo##rd##"
Enter a line of text with two words
delimited by ##:
funny wo##rd##
The two words are "funny wo" and "rd"
```

Formatted Output with printf (Optional)

VideoNote
Using printf

Starting with version 5.0, Java includes a method named printf that can be used to give output in a specific format. It is used in the same manner as the printf function in the C programming language. The method works like the print method except it allows you to add formatting instructions that specify things such as the number of digits to include after a decimal point. For example, consider the following:

```
double price = 19.5;
System.out.println("Price using println:" + price);
System.out.printf("Price using printf formatting:%6.2f",
                  price);
```

This code outputs the following lines:

```
Price using println:19.5
Price using printf formatting: 19.50
```

Using println the price is output as "19.5" immediately after the colon because we did not add an additional space. Using printf the string after the colon is " 19.50" with a blank preceding the 19.50. In this simple example, the first argument to printf is a string known as the **format specifier** and the second argument is the number or other value to be output in that format.

The format specifier %6.2f says to output a floating-point number in a **field** (number of spaces) of width six (room for six characters) and to show exactly two digits after the decimal point. So, 19.5 is expressed as "19.50" in a field of width six. Because "19.50" only has five characters, a blank character is added to obtain the six-character string " 19.50". Any extra blank space is added to the front of the value output. If the output requires more characters than specified in the field (e.g., if the field in this case was set to 1 via %1.2f), then the field is automatically expanded to the exact size of the output (five in our example). The f in %6.2f means the output is a floating-point number, that is, a number with a decimal point.

Figure 2.8 summarizes some of the common format specifiers.

We can combine multiple format specifiers into a single string. For example, given

```
double price = 19.5; int quantity = 2;
String item = "Widgets";
System.out.printf("%10s sold:%4d at $%5.2f. Total = $%1.2f",
                  item, quantity, price, quantity * price);
```

the output is: " Widgets sold:2 at $19.50. Total = $39.00". There are three blank spaces in front of "Widgets" to make a field of ten characters. Similarly, there are three blank spaces in front of the 2 to make a field of four characters. The 19.50 fits in exactly five characters and the last field for the total is expanded to five from one so it will fit the 39.00.

FIGURE 2.8 Selected Format Specifiers for `System.out.printf`

Format Specifier	Type of Output	Examples
`%c`	`Character`	`A single character: %c`
		`A single character in a field of two spaces: %2c`
`%d`	`Decimal integer number`	`An integer: %d`
		`An integer in a field of 5 spaces: %5d`
`%f`	`Floating-point number`	`A floating-point number: %f`
		`A floating-point number with 2 digits after the decimal: %1.2f`
		`A floating-point number with 2 digits after the decimal in a field of 6 spaces: %6.2f`
`%e`	`Exponential floating-point number`	`A floating-point number in exponential format: %e`
`%s`	`String`	`A string formatted to a field of 10 spaces: %10s`

SELF-TEST QUESTIONS

27. Write Java statements that will cause the following to be written to the screen:

```
Once upon a time,
there were three little programmers.
```

28. What is the difference between the methods `System.out.println` and `System.out.print`?

29. Write a complete Java program that reads a line of keyboard input containing two values of type int—separated by one or more spaces—and then displays the two numbers.

30. Write a complete Java program that reads one line of text containing exactly three words—separated by any kind or amount of whitespace—and then displays the line with spacing corrected as follows: The output has no space before the first word and exactly two spaces between each pair of adjacent words.

31. What output is produced by the following statements?

```
String s = "Hello" + "" + "Joe";
System.out.println(s);
```

2.4 DOCUMENTATION AND STYLE

"Don't stand there chattering to yourself like that," Humpty Dumpty said, looking at her for the first time, "but tell me your name and your business."

"My name is Alice, but—"

"It's a stupid name enough!" Humpty Dumpty interrupted impatiently. "What does it mean?"

"Must a name mean something?" Alice asked doubtfully.

"Of course it must," Humpty Dumpty said with a short laugh: "my name means the shape i am—and a good handsome shape it is too. With a name like yours, you might be any shape, almost."

—LEWIS CARROLL, *Through the Looking-Glass*

A program that gives the correct output is not necessarily a good program. Obviously, you want your program to give the correct output, but that is not the whole story. Most programs are used many times and are changed at some point either to fix bugs or to accommodate new demands by the user. If the program is not easy to read and understand, it will not be easy to change, and it might even be impossible to change with any realistic effort. Even if the program will be used only once, you should pay some attention to readability. After all, you will have to read the program to debug it.

In this section, we discuss four aspects of a program that help to make it more readable: meaningful names, comments, indentation, and named constants.

Meaningful Variable Names

As we mentioned earlier in this chapter, the names x and y are almost never good variable names. The name you give to a variable should be suggestive of its intended use. If the variable holds a count of something, you might name it `count`. If the variable holds a tax rate, you might name it `taxRate`.

In addition to giving variables meaningful names and giving them names that the compiler will accept, you should choose names that follow the normal practice of programmers. That way, your code will be easier for others to read and to combine with their code, should you work on a project with more than one programmer. Typically, variable names are made up entirely of letters and digits. You start each name with a lowercase letter, as we have been doing so far. The practice of starting with a lowercase letter may look strange at first, but it is a convention that is commonly used, and you will quickly get used to it. We use names that start with an uppercase letter for something else, namely, for class names like `String`.

Name your variables to suggest their use

If the name consists of more than one word, "punctuate" it by using capital letters at the word boundaries, as in `taxRate`, `numberOfTries`, and `timeLeft`.

Comments

Write programs that are self-documenting

The documentation for a program tells what the program does and how it does it. The best programs are **self-documenting.** This means that, thanks to a very clean style and very well-chosen identifiers, what the program does and how it does it will be obvious to any programmer who reads the program. You should strive for such self-documenting programs, but your programs may also need a bit of additional explanation to make them completely clear. This explanation can be given in the form of comments.

Use comments to explain details

Comments are notes that you write into your program to help a person understand the program but that are ignored by the compiler. You can insert comments into a Java program in three ways. The first way is to use the two symbols `//` at the beginning of a comment. Everything after these symbols up to the end of the line is treated as a comment and is ignored by the compiler. This technique is handy for short comments, such as

`//` comment

```
String sentence; //Spanish version
```

If you want a comment of this form to span several lines, each line must contain the symbols `//` at the beginning of the comment.

The second kind of comment can more easily span multiple lines. Anything written between the matching symbol pairs `/*` and `*/` is a comment and is ignored by the compiler.

`/*  */` comment

For example,

```
/*
 This program should only
 be used on alternate Thursdays,
 except during leap years, when it should
 only be used on alternate Tuesdays.
*/
```

This is not a very likely comment, but it does illustrate the use of `/*` and `*/`.

Many text editors automatically highlight comments by showing them in a special color. In this book, we will also write comments in a different color, as illustrated by the following comment

`/**  */` comment

```
/**
 This program should only
 be used on alternate Thursdays,
 except during leap years, when it should
 only be used on alternate Tuesdays.
*/
```

Notice that this comment uses two asterisks rather than one in the opening /**. This is not required to make it a comment, but it is needed when we use a program named javadoc that automatically extracts documentation from Java software. We will discuss how to use javadoc later in this book, but we will start using the double asterisks now.

It is difficult to explain just when you should and should not insert a comment. Too many comments can be as bad as too few comments. With too many comments, the really important information can be lost in a sea of comments that just state the obvious. As we show you more Java features, we will mention likely places for comments. For now, you should normally need them in only two situations.

First, every program file should have an explanatory comment at its beginning. This comment should give all the important information about the file: what the program does, the name of the author, how to contact the author, and the date the file was last changed, as well as other information particular to your situation, such as course assignment number. This comment should be similar to the one shown at the top of Listing 2.7.

Second, you should write comments that explain any nonobvious details. For example, look at the program in Listing 2.7. Note the two variables named radius and area. Obviously, these two variables will hold the values for the radius and area of a circle, respectively. You should *not* include comments like the following:

```
double radius; //the radius of a circle  <------  A poor comment
```

However, something is not obvious. What units are used for the radius? Inches? Feet? Meters? Centimeters? You should add a comment that explains the units used, as follows:

```
double radius; //in inches
double area; //in square inches
```

These two comments are also shown in Listing 2.7.

REMEMBER Write Self-Documenting Code

Self-documenting code uses well-chosen names and has a clear style. The program's purpose and workings should be obvious to any programmer who reads the program, even if the program has no comments. To the extent that it is possible, you should strive to make your programs self-documenting.

LISTING 2.7 Comments and Indentation

```
import java.util.Scanner;

/**
 Program to compute area of a circle.
 Author: Jane Q. Programmer.
 E-mail Address: janeq@somemachine.etc.etc.
 Programming Assignment 2.
 Last Changed: October 7, 2017.
*/

public class CircleCalculation
{
    public static void main(String[] args)
    {
        double radius; //in inches
        double area;   //in square inches
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter the radius of a circle in inches:");
        radius = keyboard.nextDouble();
        area = 3.14159 * radius * radius;
        System.out.println("A circle of radius " + radius + " inches");
        System.out.println("has an area of " + area + " square inches.");
    }
}
```

This `import` can go after the big comment if you prefer.

The vertical lines indicate the indenting pattern.

Later in this chapter, we will give an improved version of this program.

Sample Screen Output

```
Enter the radius of a circle in inches:
2.5
A circle of radius 2.5 inches
has an area of 19.6349375 square inches.
```

RECAP Java Comments

There are three ways to add comments in Java:

- Everything after the two symbols // to the end of the line is a comment and is ignored by the compiler.
- Anything written between the symbol pairs /* and */ is a comment and is ignored by the compiler.
- Anything written between the symbol pairs /** and */ is a comment that is processed by the documentation program `javadoc` but is ignored by the compiler.

Indentation

A program has a lot of structure. There are smaller parts within larger parts. For example, one part starts with

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

The body of this main method begins with an open brace { and ends with a closing brace }. Within these braces are Java statements, which we indent by a consistent number of spaces.

The program in Listing 2.7 has three levels of indentation—as indicated by the vertical lines—that clearly show its **nested structure.** The outermost structure, which defines the class `CircleCalculation`, is not indented at all. The next level of nested structure—the `main` method—is indented. The body of that method is indented yet again.

Use consistent indentation to show a program's structure

We prefer to indent by four spaces for each level of indenting. Indenting more than that leaves too little room on the line for the statement itself, whereas a smaller indent might not show up well. Indenting two or three spaces is not unreasonable, but we find four spaces to be the clearest. If you are in a course, follow the rules given by your instructor. On a programming project, you likely will have a style sheet that dictates the number of spaces you should indent. In any event, you should indent consistently within any one program.

If a statement does not fit on one line, you can write it on two or more lines. However, when you write a single statement on more than one line, indent the second and all subsequent continuation lines more than the first line.

Although the levels of nesting shown in Listing 2.7 are delimited by braces, {}, that is not always the case. Regardless of whether there are any braces, you should still indent each level of nesting.

Indent each level of nesting

Using Named Constants

Look again at the program in Listing 2.7. You probably recognize the number 3.14159 as the approximate value of pi, the number that is used in many calculations involving a circle and that is often written as `PI`. However, you might not be sure that 3.14159 is pi and not some other number. Somebody other than you might have no idea as to where the number 3.14159 came from. To avoid such confusion, you should always give a name to constants such as 3.14159 and use the name instead of writing out the number.

For example, you might give the number 3.14159 the name `PI` and define it as follows:

```
public static final double PI = 3.14159;
```

Then the assignment statement

```
area = 3.14159 * radius * radius;
```

could be written more clearly as

```
area = PI * radius * radius;
```

In Listing 2.8, we have rewritten the program from Listing 2.7 so that it uses the name `PI` as a defined name for the constant 3.14159. Note that the definition of `PI` is placed outside of the `main` method. Although named constants need not be defined near the beginning of a file, it is a good practice to place them there. That way, they will be handy if you need to correct their

LISTING 2.8 Naming a Constant

```
import java.util.Scanner;

/**
 Program to compute area of a circle.
 Author: Jane Q. Programmer.
 E-mail Address: janeq@somemachine.etc.etc.
 Programming Assignment 2.
 Last Changed: October 7, 2017.
*/

public class CircleCalculation2
{
    public static final double PI = 3.14159;

    public static void main(String[] args)
    {
        double radius; //in inches
        double area; //in square inches
        Scanner keyboard = new Scanner(System.in);

        System.out.println("Enter the radius of a circle in inches:");
        radius = keyboard.nextDouble();
        area = PI * radius * radius;
        System.out.println("A circle of radius " + radius + " inches");
        System.out.println("has an area of " + area + " square inches.");
    }
}
```

Although it would not be as clear, it is legal to place the definition of `PI` *here instead.*

Sample Screen Output

```
Enter the radius of a circle in inches:
2.5
A circle of radius 2.5 inches
has an area of 19.6349375 square inches.
```

values. For example, suppose you have a banking program that contains the named constant

```
public static final double MORTGAGE_INTEREST_RATE = 6.99;
```

and suppose the interest rate changes to 8.5 percent. You can simply change the value of the named constant to

```
public static final double MORTGAGE_INTEREST_RATE = 8.5;
```

You would then need to recompile your program, but you need not change anything else in it.

Using a named constant, like MORTGAGE_INTEREST_RATE, can save you a lot of work. To change the mortgage interest rate from 6.99 percent to 8.5 percent, you change only one number. If the program did not use a named constant, you would have to change every occurrence of 6.99 to 8.5. Although a text editor makes this task easy, this change might not be right. If some occurrences of 6.99 represent the mortgage interest rate, while other occurrences of 6.99 represent something else, you would have to decide just what each 6.99 means. That would surely produce confusion and probably introduce errors.

Name your constants to make your program clearer and easier to maintain

SELF-TEST QUESTIONS

32. What are the kinds of comments in Java?

33. What is the output produced by the following Java code:

```
/**
  Code for Question 33  .
*/
System.out.println("One");
//System.out.println("Two");
System.out.println("And hit it!");
```

34. Although it is kind of silly, state legislatures have been known to pass laws that "change" the value of pi. Suppose you live in a state where, by law, the value of pi is exactly 3.14. How must you change the program in Listing 2.8 to make it comply with the law?

2.5 GRAPHICS SUPPLEMENT

It ain't over till it's over.

—YOGI BERRA

This section begins with a JavaFX application from Chapter 1 redone following the style rules discussed in this chapter. However, the rest of the section is devoted to the JOptionPane class. This class provides you with a way to use windowing for I/O in your Java programs.

Chapter 3 also has a small amount of material on `JOptionPane`. All the material on `JOptionPane` in this chapter and in Chapter 3 is independent of the other material in this book. You may cover the material on `JOptionPane` whether you cover the other material in the other graphics supplements or not. You can also omit the material on `JOptionPane` and still cover the other material in the graphics supplements.

Style Rules Applied to a JavaFX Application

Listing 2.9 revises the program that appears in Listing 1.2 of Chapter 1, adding named constants for all the integer arguments as well as explanatory comments. At first glance it may seem that all these named constants simply complicate the code. However, they make writing and changing the code much easier.

Writing such constants helps you to plan and organize your drawing. The named constants enable you to clearly and explicitly specify constraints. For example, the statement

```
public static final int Y_LEFT_EYE = Y_RIGHT_EYE;
```

ensures that the two eyes appear at the same level.

A graphics program like this one often needs to be tuned by adjusting the various integer values. Finding the right value to change when you need to adjust, say, the mouth width is much easier when you use named constants.

LISTING 2.9 Revision of Listing 1.2 Using Comments and Named Constants *(part 1 of 2)*

```
import javafx.application.Application;
import javafx.scene.canvas.Canvas;
import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.stage.Stage;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.shape.ArcType;
/**
 JavaFX Application that displays a happy face.
 Author: Jane Q. Programmer
 Revision of Listing 1.2.
*/
public class HappyFace extends Application
{
   public static final int WINDOW_WIDTH = 400;
   public static final int WINDOW_HEIGHT = 300;

   public static final int FACE_DIAMETER = 200;
   public static final int X_FACE = 100;
   public static final int Y_FACE = 50;
```

These can go after the big comment if you prefer.

LISTING 2.9 Revision of Listing 1.2 Using Comments and Named Constants *(part 2 of 2)*

```
   public static final int EYE_WIDTH = 10;
   public static final int EYE_HEIGHT = 20;
   public static final int X_RIGHT_EYE = 155;
   public static final int Y_RIGHT_EYE = 100;
   public static final int X_LEFT_EYE = 230;
   public static final int Y_LEFT_EYE = Y_RIGHT_EYE;

   public static final int MOUTH_WIDTH = 100;
   public static final int MOUTH_HEIGHT = 50;
   public static final int X_MOUTH = 150;
   public static final int Y_MOUTH = 160;
   public static final int MOUTH_START_ANGLE = 180;
   public static final int MOUTH_DEGREES_SHOWN = 180;

   public static void main(String[] args)
   {
      launch(args);
   }
   @Override
   public void start(Stage primaryStage) throws Exception
   {
        Group root = new Group();
        Scene scene = new Scene(root);
        Canvas canvas = new Canvas(WINDOW_WIDTH, WINDOW_HEIGHT);
        GraphicsContext gc = canvas.getGraphicsContext2D();
        // Draw face outline
        gc.strokeOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
        // Draw eyes
        gc.fillOval(X_RIGHT_EYE, Y_RIGHT_EYE, EYE_WIDTH, EYE_HEIGHT);
        gc.fillOval(X_LEFT_EYE, Y_LEFT_EYE, EYE_WIDTH, EYE_HEIGHT);
        // Draw mouth
        gc.strokeArc(X_MOUTH, Y_MOUTH, MOUTH_WIDTH, MOUTH_HEIGHT,
                     MOUTH_START_ANGLE, MOUTH_DEGREES_SHOWN, ArcType.OPEN);
        root.getChildren().add(canvas);
        primaryStage.setTitle("HappyFace in JavaFX");
        primaryStage.setScene(scene);
        primaryStage.show();
   }
}
```

The JavaFX application drawing is the same as the one shown in Listing 1.2.

■ PROGRAMMING TIP Use Named Constants in a Graphics Application

When designing a drawing, identify the components and their dimensions. Give names to these dimensions and define them as named constants. As much as possible and reasonable, make these constants interdependent. That is, since it is likely that certain dimensions are related to others, define one named constant in terms of another one. Fine-tuning the values of these constants is easier if you can describe the relationships among the various dimensions symbolically by using named constants. ■

Introducing the Class JOptionPane

Any application program can use windows for I/O

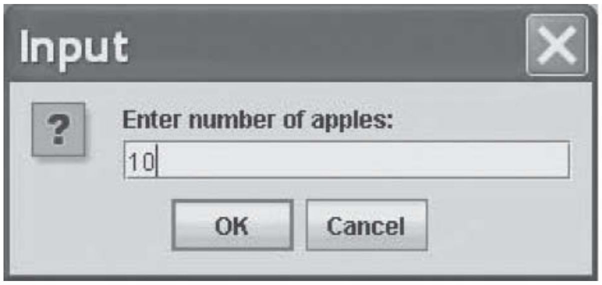
Listing 2.10 contains a very simple Java application program that has a windowing interface. The program produces three windows, one at a time. The first window to appear is labeled Dialog 1 in Listing 2.10. The user enters a number in the text field of this window and then clicks the OK button with the mouse. The first window then goes away and the second window appears. The user handles the second window in a similar way. When the user clicks the OK button in the second window, the second window goes away and the third window appears. Let's look at the details.

LISTING 2.10 Program Using JOptionPane for I/O *(part 1 of 2)*

```
import javax.swing.JOptionPane;

public class JOptionPaneDemo
{
    public static void main(String[] args)
    {
        String appleString =
          JOptionPane.showInputDialog("Enter number of apples:");
          int appleCount = Integer.parseInt(appleString);

        String orangeString =
           JOptionPane.showInputDialog("Enter number of oranges:");
        int orangeCount = Integer.parseInt(orangeString);

        int totalFruitCount = appleCount + orangeCount;

        JOptionPane.showMessageDialog(null,
               "The total number of fruits = " + totalFruitCount);
        System.exit(0);
    }
}
```

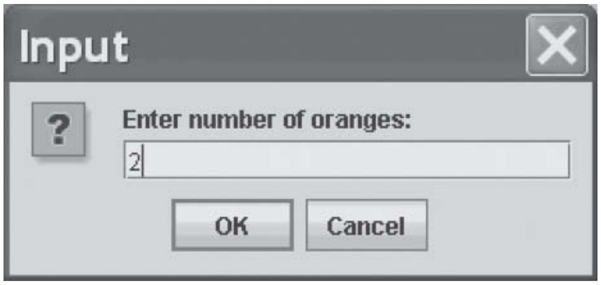
LISTING 2.10 Program Using `JOptionPane` for I/O *(part 2 of 2)*

Dialog 1

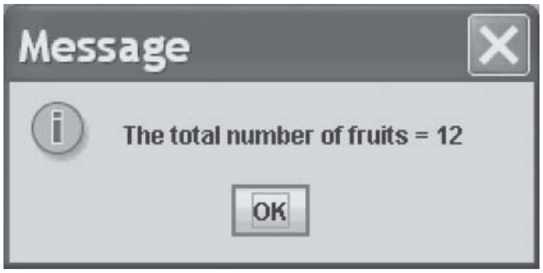


When the user clicks OK, the window goes away and the next window (if any) is displayed.

Dialog 2



Dialog 3



```
The total number of fruits = 12
```

REMEMBER Running a `JOptionPane` Program

You run a program that uses `JOptionPane`, such as the one in Listing 2.10, in the same way you run any application program. You do not run it as if it were an applet.

This program uses the class `JOptionPane` to construct the windows that interact with the user. `JOptionPane` is a standard, predefined class that comes with every installation of Java. To make it available to your program, you write

Import `JOptionPane` from the Swing package

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

This statement tells the computer where to find the definition of the `JOptionPane` class. You may recall that we mentioned a library called Swing, which is a library of classes that we can use for windowing interfaces. These libraries are called packages, and in a Java program the Swing package is denoted `javax.swing`, with a lowercase *s*. The class `JOptionPane` is in this package. The previous `import` statement indicates this fact. You put this statement at the start of any program file that uses the class `JOptionPane`.

In Chapter 1 we indicated that JavaFX is replacing Swing. However, at this time there is not an equivalent version of `JOptionPane` in JavaFX with the same simplicity as the version discussed here. For this reason we continue to use `JOptionPane` until a better alternative is made available in JavaFX.

The first program instruction for the computer is

```
String appleString =
  JOptionPane.showInputDialog("Enter number of apples:");
```

It declares `appleString` to be a variable of type `String` and then starts the windowing action. The two lines are a single statement, or instruction, and would normally be written on one line, except that doing so would make the line inconveniently long.

`JOptionPane` is a class used for producing special windows—called **dialog windows, dialog boxes,** or simply **dialogs**—that either obtain input or display output from your program. The method `showInputDialog` produces a dialog for obtaining input. The string argument, in this case `"Enter number of apples:"`, is written in the window to tell the user what to enter. You the programmer choose this string, depending on what sort of input you want. This invocation of the method `showInputDialog` will produce the first dialog shown in Listing 2.10. The user clicks the mouse in the text field and then types in some input. The user can use the backspace key to back up and change the input if necessary. Once the user is happy with the input, the user clicks the OK button and the window goes away. As an alternative, the user can press the Enter (Return) key instead of clicking the OK button. That takes care of the responsibility of the user, but how does this input get to your program? Read on.

The method invocation

Defining a dialog for input

```
JOptionPane.showInputDialog("Enter number of apples:");
```

returns—that is, produces—the input that the user typed into the text field. This invocation is within an assignment statement that stores this input. Specifically, the string input is stored in the variable `appleString`. When you use `JOptionPane` to read user input, only string values are read. If you want numbers, your program must convert the input string to a number.

The next statement begins by declaring the variable appleCount as an int. The int says that the data stored in the variable appleCount must be an *integer*. The programmer who wrote this program wants the user to enter an integer into that first input window and wants the program to store this integer in the variable appleCount. Because JOptionPane reads only strings, this means converting the string to a value of type int. To see why this conversion is necessary, let's say that the user types 10 into the dialog box, indicating that there are 10 apples. What he or she has actually typed is the character '1' followed by the character '0' to produce the string "10". When using these input windows, you must be aware of the fact that all program input from the user—and all output to the user for that matter—consists of strings of characters. If you want your program to think of the input from an input window as a number, your program must convert the string, such as "10", into the corresponding number, in this case 10. To computers, "10" and 10 are very different things. (In real life they are also different, but we usually ignore that difference.) "10" is a string consisting of two characters, while 10 is a number that can, for example, be added to or subtracted from another number.

ASIDE All Input and Output Are Strings

All program input from the user and all output to the user consist of strings of characters. When using the class JOptionPane to read numeric input, you must convert the string that is read to the desired numeric value, as you will see next. Even the class Scanner reads user input as a string. However, when you call a method like nextInt or nextDouble, for example, the conversion from string input to numeric input is done for you.

In Listing 2.10, the string typed in by the user is stored in the variable appleString. Since we expect the user to type the digits of an integer, our program needs to convert this string to an int. We store the resulting int value in the variable appleCount, as follows:

```
int appleCount = Integer.parseInt(appleString);
```

parseInt is a method of the class Integer

Integer is a class provided by Java, and parseInt is a method of the class Integer. The method invocation Integer.parseInt(appleString) converts the string stored in the variable appleString into the corresponding integer number. For example, if the string stored in appleString is "10", this method invocation will return the integer 10.

FAQ Why do you invoke some methods using a class name instead of an object name?

Normally, a method invocation uses an object name. For example, if greeting is a variable of type String, we write greeting.length() to invoke the method length. However, when we call the methods of the class JOptionPane, we use the class name JOptionPane in place of an object name. The same is true of the method parseInt in the

(continued)

class `Integer`. What is the story? Some special methods do not require an object to be invoked and, instead, are called using the class name. These methods are called static methods and are discussed in Chapter 6. Although static methods are only a very small fraction of all methods, they are used for certain fundamental tasks, such as I/O, and so we have encountered them early. You can tell whether a standard method in the Java Class Library is static by looking at the documentation for its class on the Oracle Web site.

GOTCHA Inappropriate Input

A program is said to crash when it ends abnormally, usually because something went wrong. When a program uses the method `JOptionPane.showInputDialog` to get input—as in Listing 2.10—the user must enter the input in the correct format, or else the program is likely to crash. If your program expects an integer to be entered, and the user enters 2,000 instead of 2000, your program will crash, because integers in Java cannot contain a comma. Later you will learn how to write more robust windowing programs that do not require that the user be so knowledgeable and careful. Until then, you will have to simply tell the user to be very careful. ■

A crash is an abnormal end to a program's execution

The next few statements in Listing 2.10 are only a slight variation on what we have just discussed. A dialog box—the second one in Listing 2.10—is produced and gets an input string from the user. The string is converted to the corresponding integer and stored in the variable `orangeCount`.

The next line of the program contains nothing new to you:

```
int totalFruitCount = appleCount + orangeCount;
```

It declares `totalFruitCount` as a variable of type `int` and sets its value to the sum of the `int` values in the two variables `appleCount` and `orangeCount`.

The program now should display the number stored in the variable `totalFruitCount`. This output is accomplished by the following statement:

Defining a dialog for output

```
JOptionPane.showMessageDialog(null,
        "The total number of fruits = " + totalFruitCount);
```

This statement calls the method `showMessageDialog`, which is another method in the class `JOptionPane`. This method displays a dialog window that shows some output. The method has two arguments, which are separated by a comma. For now the first argument will always be written as `null`. You will have to wait for an explanation of what this `null` is. Until then, you will not go too far wrong in thinking of null as a place holder that is being used because we do not need any "real" first argument. The second argument is easy to explain; it is the string that is written in the output dialog. So the previous method invocation produces the third dialog shown in Listing 2.10. This dialog stays on the screen until the user clicks the OK button with the mouse or presses the Enter (Return) key, at which time the window disappears.

Note that you can give the method showMessageDialog its string argument using the plus symbol in the same way you do when you give a string as an argument to System.out.println. That is, you can append the integer value stored in totalFruitCount to a string literal. Moreover, Java will automatically convert the integer value stored in totalFruitCount to the corresponding string.

GOTCHA Displaying Only a Number

In Listing 2.10 the final program output is sent to an output window by the following statement:

```
JOptionPane.showMessageDialog(null,
    "The total number of fruits = " + totalFruitCount);
```

It is good style to always label any output. So displaying the string

```
"The total number of fruits = "
```

is very important to the style and understandability of the program. Moreover, the invocation of the method showMessageDialog will not even compile unless you include a string in the output. For example, the following will not compile

```
JOptionPane.showMessageDialog(null, totalFruitCount);
   //Illegal
```

The method showMessageDialog will not accept an int value—or a value of any other primitive type—as its second argument. But if you connect the variable or number to a string by using the plus symbol, the number is converted to a string, and the argument will then be accepted. ■

The last program statement in Listing 2.10,

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

System.exit(0) must end a program that uses dialogs

simply says that the program should end. System is a predefined Java class that is automatically provided by Java, and exit is a method in the class System. The method exit ends the program as soon as it is invoked. In the programs that we will write, the integer argument 0 can be any integer, but by tradition we use 0 because it is used to indicate the normal ending of a program. The next chapter formally presents this method.

GOTCHA Forgetting `System.exit(0);`

If you omit the last line

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

from the program in Listing 2.10, everything will work as we described. The user will enter input using the input windows, and the output window will show the output. However, when the user clicks the OK button in the output window, the output window will go away, but the program will not end. The "invisible" program will still be there, using up computer resources and possibly keeping you from doing other things. With windowing programs, *it ain't over till it's over*. `System.exit(0)` is what really ends the program, not running out of statements to execute. So do not forget to invoke `System.exit(0)` in all of your windowing programs.

What do you do if you forget to call `System.exit(0)` and the program does not end by itself? You can end a program that does not end by itself, but the way to do so depends on your particular operating system. On many systems (but not all), you can stop a program by typing control-C, which you type by holding down the control (Ctrl) key while pressing the C key.

When you write an application program that has a windowing interface you always need to end the program with the statement

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

If the program does not use a windowing interface, such as the programs in Sections 2.1 through 2.4, you do not need to invoke `System.exit(0)`. ■

FAQ Why do some programs need `System.exit` and some do not?

An application program that uses a windowing interface, such as the one in Listing 2.10, must end with the following method invocation:

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

A program that uses simple text input and output—like the ones in previous sections of this chapter—does not require this call. What is the reason for this difference?

If a program uses simple text input and output, Java can easily tell when it should end; the program should end when all the statements have executed.

(continued)

The situation is not so simple for application programs that have windowing interfaces. Java cannot easily tell when a windowing program should end. Many windowing programs end only when the user clicks a certain button or takes certain other actions. Those details are determined by the programmer, not by the Java language. The simple program in this section does happen to end when all its statements have executed. But for more complicated windowing programs, the end is not so easy to find, so you must tell Java when to end a program's execution by invoking `System.exit`.

RECAP `JOptionPane` for Windowing Input/Output

You can use the methods `showInputDialog` and `showMessageDialog` to produce input and output windows—called dialogs—for your Java programs. When using these methods, you include the following at the start of the file that contains your program:

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

The syntax for input and output statements using these methods is given below:

SYNTAX FOR INPUT

```
String_Variable = JOptionPane.showInputDialog(String_
                  Expression);
```

The *String_Expression* is displayed in a dialog window that has both a text field in which the user can enter input and a button labeled OK. When the user types in a string and clicks the OK button in the window, the method returns the string. That string is stored in the *String_Variable*. As an alternative, pressing the Enter (Return) key is equivalent to clicking the OK button. Note that when input is done in this way, it is read as a string. If you want the user to enter, for example, integers, your program must convert the input string to the equivalent number.

EXAMPLE

```
String orangeString =
    JOptionPane.showInputDialog(
     "Enter number of oranges:");
```

(continued)

SYNTAX FOR OUTPUT

```
JOptionPane.showMessageDialog(null,  String_Expression);
```

The *String_Expression* is displayed in a dialog window that has a button labeled OK. When the user clicks the OK button with the mouse or presses the Enter (Return) key, the window disappears.

EXAMPLE

```
JOptionPane.showMessageDialog(null,
    "The total number of fruits = " + totalFruitCount);
```

SELF-TEST QUESTIONS

35. In the following two lines, one identifier names a class, one identifier names a method, and something is an argument. What is the class name? What is the method name? What is the argument?

```
appleString =
    JOptionPane.showInputDialog("Enter number of apples:");
```

36. Give a Java statement that will display a dialog window on the screen with the message

```
I Love You.
```

37. Give a Java statement that, when executed, will end the program.

38. What would happen if you omitted `System.exit(0)` from the program in Listing 2.10? Would the program compile? Would it run without problems?

39. Write a complete Java program that produces a dialog window containing the message `HelloWorld!`. Your program does nothing else.

40. Write a complete Java program that behaves as follows. The program displays an input dialog window asking the user to enter a whole number. When the user enters a whole number and clicks the OK button, the window goes away and an output dialog window appears. This window simply tells the user what number was entered. When the user clicks the

OK button in the window, the program ends. (Hey, this is only Chapter 2. The programs will get complicated soon enough.)

Reading Input as Other Numeric Types

Since the method `JOptionPane.showInputDialog` reads a string from the user, we had to convert it to an `int` in our previous program by using the method `parseInt`from the class `Integer`. You can convert a number represented as a string to any of the other numeric types by using other methods. For example, the following code asks the user to enter a value of type `double` and stores it in the variable `decimalNumber` of type double:

```
String numberString = JOptionPane.showInputDialog(
                      "Enter a number with a decimal point:");
double decimalNumber = Double.parseDouble(numberString);
```

Figure 2.9 lists the correct conversion method for each numeric primitive type. To convert a value of type `String` to a value of the type given in the first column of the figure, you use the method given in the second column. Each of the methods in this second column returns a value of the type given in the first column. The *String_To_Convert* must be a correct string representation of a value of the type given in the first column. For example, to convert to an `int`, *String_To_Convert* must be a whole number written in the usual way without any decimal point and in the range of the type `int`. Chapter 6 will discuss these classes and methods in more detail.

FIGURE 2.9 Methods for Converting Strings to Numbers

Result Type	Method for Converting
`byte`	`Byte.parseByte(`*String_To_Convert*`)`
`short`	`Short.parseShort(`*String_To_Convert*`)`
`int`	`Integer.parseInt(`*String_To_Convert*`)`
`long`	`Long.parseLong(`*String_To_Convert*`)`
`float`	`Float.parseFloat(`*String_To_Convert*`)`
`double`	`Double.parseDouble(`*String_To_Convert*`)`

PROGRAMMING TIP Multiline Output in a Dialog Window

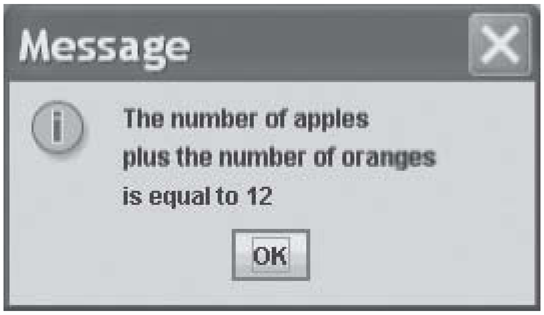
If you want to display multiple lines by using JOptionPane's method showMessageDialog, you can insert the new-line character '\n' into the string used as the second argument. If the string becomes too long, which almost always happens with multiline output, you can write each line as a separate string ending with '\n' and connect them with plus symbols. If the lines are long or numerous, the window will expand as needed to hold all the output.

For example, consider

```
JOptionPane.showMessageDialog(null,
                        "The number of apples\n"
                        + "plus the number of oranges\n"
                        + "is equal to " + totalFruit);
```

This invocation will produce the dialog shown in Figure 2.10, provided totalFruit is a variable of type int whose value is 12.

FIGURE 2.10 A Dialog Window Containing Multiline Output



```
The number of apples
plus the number of oranges
is equal to 12
```

PROGRAMMING EXAMPLE Change-Making Program with Windowing I/O

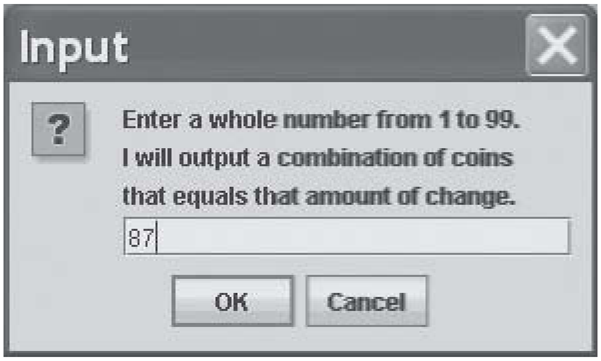
The program in Listing 2.11 is the same as the one in Listing 2.3, but it has a windowing interface. Notice that both the input dialog and the output dialog display multiple lines of text. If any of the details about calculating the numbers of coins is unclear, look back at the explanation of Listing 2.3.

LISTING 2.11 A Change-Making Program with Windows for I/O *(part 1 of 2)*

```
import javax.swing.JOptionPane;
public class ChangeMakerWindow

{
    public static void main(String[] args)
    {
        String amountString = JOptionPane.showInputDialog(
                    "Enter a whole number from 1 to 99.\n" +
                    "I will output a combination of coins\n" +
                    "that equals that amount of change.");
        int amount, originalAmount,
        quarters, dimes, nickels, pennies;
        amount = Integer.parseInt(amountString);
        originalAmount = amount;

        quarters = amount / 25;
        amount = amount % 25;
        dimes = amount / 10;
        amount = amount % 10;
        nickels = amount / 5;
        amount = amount % 5;
        pennies = amount;

        JOptionPane.showMessageDialog(null, originalAmount +
                       " cents in coins can be given as:\n" +
                       quarters + " quarters\n" +
                       dimes    + " dimes\n" +
                       nickels  + " nickels and\n" +
                       pennies  + " pennies");
        System.exit(0);
    }
}
```

Do not forget that you need `System.exit` *in a program with input or output windows.*

Input Dialog



Input

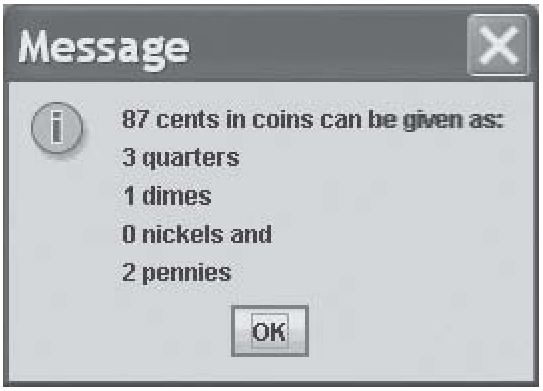
Enter a whole number from 1 to 99.
I will output a combination of coins
that equals that amount of change.

87

(continued)

LISTING 2.11 A Change-Making Program with Windows for I/O *(part 2 of 2)*

Output Dialog



```
87 cents in coins can be given as:
3 quarters
1 dimes
0 nickels and
2 pennies
```

Note that we did not forget the `import` statement

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

nor did we forget that an application program using `JOptionPane` must invoke the method `System.exit` to terminate execution. If you forget the `import` statement, the compiler will complain. However, if you omit the invocation of the method `System.exit`, the compiler will not complain, but your program will not end, even after all the statements are executed and all the windows have disappeared.

CHAPTER SUMMARY

- A variable can hold values, such as numbers. The type of the variable must match the type of the value stored in the variable.
- Variables and all other items in a program should be given names that indicate how they are used. These names are called identifiers in Java.
- All variables should be given an initial value before you use them in the program. You can do this using an assignment statement, optionally combined with the variable declaration.
- Parentheses in arithmetic expressions indicate the order in which the operations are performed.
- When you give a value to a variable by using an assignment statement, the data type of the variable must be compatible with the type of the value. Otherwise, a type cast is necessary.

- The methods in the class `Scanner` can be used to read keyboard input.
- Your program should display a message to prompt the user to enter data at the keyboard.
- The method `println` advances to a new line after it displays its output, whereas the method `print` does not.
- The method `printf` may be used for formatted output.
- You can have variables and constants of type `String`. `String` is a class type that behaves very much like a primitive type.
- You can use the plus symbol to indicate the concatenation of two strings.
- The class `String` has methods that you can use for string processing.
- You should define names for number constants in a program and use these names rather than writing out the numbers within your program.
- Programs should be self-documenting to the extent possible. However, you should also insert comments to explain any unclear points. Use // to begin a one-line comment, or use either of the pairs /** and */ or /* and */ to enclose multiline comments.
- You can use the class `JOptionPane` to create a windowing interface for input and output.

Exercises

1. Write a program that demonstrates the approximate nature of `double` values by performing the following tasks:

   - Use `Scanner` to read a double value *x*.
   - Compute 5.0 * *x* and store the result in *y*.
   - Display *x*, *y*, and the sum of *x* and *y*.
   - Divide 5.0 from *x* and store the result in *z*. Display the value of *z*.

   Try your program with values of *x* that range from `2e-11` to `2e11`.

2. Write a program that demonstrates type casting of `double` values by performing the following tasks:

   - Use `Scanner` to read a floating-point value *x*.
   - Type cast *x* to an `int` value and store the result in *y*.
   - Display *x* and *y* clearly labeled.
   - Type cast *x* to a `byte` value and store the result in *z*.
   - Display *x* and *z* clearly labeled.

   Try your program with positive and negative values of *x* that range in magnitude from `2e-11` to `2e11`. What can you conclude?

3. Write a program that demonstrates the operator ++ by performing the following tasks:

   - Use `Scanner` to read an integer value *x*.
   - Compute *x*++ and store the result in *y*.
   - Display *x* and *y* clearly labeled.
   - Compute ++*x* and store the result in *z*.
   - Display *x* and *z* clearly labeled.

   Try your program with positive and negative values of *x*. What can you conclude about pre- and post-increment?

4. If `u = 2, v = 3`, `w = 5`, `x = 7`, and `y = 11`, what is the value of each of the following expressions, assuming `int` variables?

   - u + v * w + x
   - u + y % v * w + x
   - u++ / v + u++ * w

5. What changes to the `ChangeMaker` program in Listing 2.3 are necessary if it also accepts coins for one dollar and half a dollar?

6. If the `int` variable `x` contains 10, what will the following Java statements display?

   ```
   System.out.println("Test 1" + x * 3 * 2.0);
   System.out.println("Test 2" + x * 3 + 2.0);
   ```

   Given these results, explain why the following Java statement will not compile:

   ```
   System.out.println("Test 3" + x * 3 - 2.0);
   ```

7. Write some Java statements that use the `String` methods `lastIndexOf` and `substring` to find the last word in a string. We define *word* to be a string of characters that does not include whitespace. For example, the last word of the string

   ```
   "Hello, my good friend!"
   ```

   is the string `"friend!"`.

8. Repeat the previous exercise, but find the second last word in the string.

9. What does the following Java statement display?

   ```
   System.out.println("\"\\Release\"\\\tYour\n\\\'Stress\'\\\"\
   robot\"");
   ```

   What is the significance of `\R` and `\r` here?

10. Write a single Java statement that will display the words *Alice, In,* and *WonderLand,* adding tab between the words and the third word within double quotes.

11. What does the Java code

    ```
    Scanner keyboard = new Scanner(System.in);
    System.out.println("Enter a value");
    double val = keyboard.nextDouble();
    System.out.println("Value entered is " + val);
    ```

    display when the keyboard input is `4pi`?

12. What does the Java code

    ```
    Scanner keyboard = new Scanner(System.in);
    keyboard.useDelimiter("m");
    System.out.println("Enter a string.");
    String string1 = keyboard.next();
    String string2 = keyboard.next();
    System.out.println("String1 is " + string1);
    System.out.println("String2 is " + string2);
    ```

    display when the keyboard input is

    ```
    Dream
    Think
    Imagine
    ```

13. Repeat the previous exercise, but remove the statement `keyboard.useDelimiter("m")`. Enter `'Dream'` as the first string and `'Think Imagine'` as the second string.

14. Many sports have constants embedded in their rules. For example, baseball has 9 innings, 3 outs per inning, 3 strikes in an out, and 4 balls per walk. We might encode the constants for a program involving baseball as follows:

    ```
    public static final int INNINGS = 9;
    public static final int OUTS_PER_INNING = 3;
    public static final int STRIKES_PER_OUT = 3;
    public static final int BALLS_PER_WALK = 4;
    ```

    For each of the following popular sports, give Java named constants that could be used in a program involving that sport:

    - Basketball
    - American football
    - Soccer
    - Cricket
    - Bowling

15. Repeat Exercise 18 in Chapter 1, but define and use named constants. Graphics

16. Define named constants that you could use in Programming Project 6 in Chapter 1.

PRACTICE PROGRAMS

Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.

1. Write a program that reads Principal, Rate and Time from the user. The program then calculates and displays the Simple Interest and the Amount.

2. Write a program that uses `Scanner to` read two strings from the keyboard. Display each string, along with its length, on two separate lines. Then create a new string by joining the two strings, separated by a blank. Display the new string and its length on a third line.

3. Write a program that reads the amount of a monthly mortgage payment and the amount still owed—the outstanding balance—and then displays the amount of the payment that goes to interest and the amount that goes to principal (i.e., the amount that goes to reducing the debt). Assume that the annual interest rate is 7.49 percent. Use a named constant for the interest rate. Note that payments are made monthly, so the interest is only one twelfth of the annual interest of 7.49 percent.

4. Write a program that reads a four-digit integer, such as 2014, and then displays it, one digit per line in reverse order, like so:

   ```
   4
   1
   0
   2
   ```

   Your prompt should tell the user to enter a four-digit integer. You can then assume that the user follows directions. (*Hint*: Use the division and remainder operators.)

5. Repeat the previous project, but read the input in string and display the alternate characters from last.

6. The following program has some compile time errors. Find and fix the errors.

   ```
   import java.util.Scanner;
   public class CompErrors
   {
    public static void main(String[] args)
    {
      Scanner keyboard = new Scanner(System.in);
      string mbBrand;
      double price;
      System.out.println("Enter the Mobile Brand name");
      MbBrand = keyboard.nextLine();
      System.out.println("Enter the price ");
      price = keyboard.nextInt();
      MbBrand.toLowercase()
   ```

```
      System.out.printn("The Mobile Brand is " +
      MbBrand + " and its price is " + Price);
    }
  }
```

PROGRAMMING PROJECTS

Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways. Visit www.myprogramminglab.com to complete many of these Programming Projects online and get instant feedback.

VideoNote
Solving a conversion problem

1. Write a program that converts degrees from Fahrenheit to Celsius, using the formula

   DegreesC = 5(*DegreesF* −32)/9

   Prompt the user to enter a temperature in degrees Fahrenheit as a whole number without a fractional part. Then have the program display the equivalent Celsius temperature, including the fractional part to at least one decimal point. A possible dialogue with the user might be

   ```
   Enter a temperature in degrees Fahrenheit: 72
   72 degrees Fahrenheit is 22.2 degrees Celsius.
   ```

2. Write a program that reads a line of text and then displays the line, but with the first occurrence of *hate* changed to *love*. For example, a possible sample dialogue might be

   ```
   Enter a line of text.
   I hate you.
   I have rephrased that line to read:
   I love you.
   ```

   You can assume that the word *hate* occurs in the input. If the word *hate* occurs more than once in the line, your program will replace only its first occurrence.

3. Write a program that will read a line of text as input and then display the line with the first word moved to the end of the line. For example, a possible sample interaction with the user might be

   ```
   Enter a line of text. No punctuation please.
   Java is the language
   I have rephrased that line to read:
   Is the language Java
   ```

   Assume that there is no space before the first word and that the end of the first word is indicated by a blank, not by a comma or other punctuation. Note that the new first word must begin with a capital letter.

4. Write a program that asks the user to enter a favorite color, a favorite food, a favorite animal, and the first name of a friend or relative. The program should then print the following two lines, with the user's input replacing the items in italics:

```
I had a dream that Name ate a Color Animal
and said it tasted like Food!
```

For example, if the user entered `blue` for the color, `hamburger` for the food, `dog` for the animal, and `Jake` for the person's name, the output would be

```
I had a dream that Jake ate a blue dog
and said it tasted like hamburger!
```

Don't forget to put the exclamation mark at the end.

5. Write a program that determines the change to be dispensed from a vending machine. An item in the machine can cost between 25 cents and a dollar, in 5-cent increments (25, 30, 35, ... , 90, 95, or 100), and the machine accepts only a single dollar bill to pay for the item. For example, a possible dialogue with the user might be

```
Enter price of item
(from 25 cents to a dollar, in 5-cent increments): 45

You bought an item for 45 cents and gave me a dollar,
so your change is
2 quarters,
0 dimes, and
1 nickel.
```

6. Write a program that reads a 4-bit binary number from the keyboard as a string and then converts it into decimal. For example, if the input is 1100, the output should be 12. (*Hint:* Break the string into substrings and then convert each substring to a value for a single bit. If the bits are b_0, b_1, b_2, and b_3, the decimal equivalent is $8b_0 + 4b_1 + 2b_2 + b_3$.)

7. Many private water wells produce only 1 or 2 gallons of water per minute. One way to avoid running out of water with these low-yield wells is to use a holding tank. A family of 4 will use about 250 gallons of water per day. However, there is a "natural" water holding tank in the casing (i.e., the hole) of the well itself. The deeper the well, the more water that will be stored that can be pumped out for household use. But how much water will be available?

   Write a program that allows the user to input the radius of the well casing in inches (a typical well will have a 3-inch radius) and the depth of the well in feet (assume water will fill this entire depth, although in practice that will not be true since the static water level will generally be 50 feet or more below the ground surface). The program should output the number of gallons stored in the well casing. For your reference:

   The volume of a cylinder is $\pi r^2 h$, where r is the radius and h is the height.
   1 cubic foot = 7.48 gallons of water.

   For example, a 300-foot well full of water with a radius of 3 inches for the casing holds about 441 gallons of water—plenty for a family of 4 and no need to install a separate holding tank.

8. The Harris-Benedict equation estimates the number of calories your body needs to maintain your weight if you do no exercise. This is called your basal metabolic rate, or BMR.

   VideoNote
   Solution to Project 8

   The calories needed for a woman to maintain her weight is:

   BMR = 655 + (4.3 × weight in pounds) + (4.7 × height in inches) – (4.7× age in years)

   The calories needed for a man to maintain his weight is:

   BMR = 66 + (6.3 × weight in pounds) + (12.9 × height in inches) – (6.8 × age in years)

   A typical chocolate bar will contain around 230 calories. Write a program that allows the user to input his or her weight in pounds, height in inches, and age in years. The program should then output the number of chocolate bars that should be consumed to maintain one's weight for both a woman and a man of the input weight, height, and age.

9. Repeat any of the previous programming projects using `JOptionPane`, which is described in the graphics supplement.

   Graphics

10. Write a program that reads a string for a date in the format *month* / *day* / *year* and displays it in the format *day* . *month* . *year*, which is a typical format used in Europe. For example, if the input is 06 /17/11, the output should be 17.06.11.Your program should use JOptionPane for input and output.

    Graphics

11. It is important to consider the effect of thermal expansion when building a structure that must withstand changes in temperature. For example, a metal beam will expand in hot temperatures. The additional stress could cause the structure to fail. Similarly, a material will contract in cold temperatures. The linear change in length of a material if it is allowed to freely expand is described by the following equation:

    $L_\Delta = \alpha L_0 T_\Delta$

    Here, L_0 is the initial length of the material in meters, L_Δ is the displacement in meters, T_Δ is the change in temperature in Celsius, and α is a coefficient for linear expansion.

    Write a program that inputs α, L_Δ, T_Δ, and the name of the material, then calculates and outputs the material's name and the amount of linear displacement. Here are some values for α for different materials.

Aluminum	2.31×10^{-5}
Copper	1.70×10^{-5}
Glass	8.50×10^{-6}
Steel	1.20×10^{-5}

Answers to Self-Test Questions

1. The following are all legal variable names:

   ```
   rate1, TimeLimit, numberOfWindows
   ```

   `TimeLimit` is a poor choice, however, since it violates the normal convention that variables should start with a lowercase letter. A better choice would be `timeLimit`. `1stPlayer` is illegal because it starts with a digit. `myprogram.java` is illegal because it contains an illegal character, the dot. Finally, `long` is illegal as a variable because it is a keyword.

2. Yes, a Java program can have two different variables with the names `aVariable` and `avariable`, since they use different capitalization and so are different identifiers in Java. However, it is not a good idea to use identifiers that differ only in the way they are capitalized.

3. `int count = 0;`

4. `double rate = 0.0, time = 0.0;`

   You could write this declaration as two statements, as follows:

   ```
   double rate = 0.0;
   double time = 0.0;
   ```

   It is also correct to replace 0.0 with 0, because Java will automatically convert the `int` value 0 to the `double` value 0.0:

5.
   ```
   int miles = 0;
   double flowRate = 50.56;
   ```

6. The normal practice of programmers is to spell named constants with all uppercase letters, using the underscore symbol to separate words.

7. `public static final int HOURS_PER_DAY = 24;`

8. `interest = 0.05 * balance;`

   The following is also correct:

   ```
   interest = balance * 0.05;
   ```

9. `interest = balance * rate;`

10. `count = count + 3;`

11.
    ```
    b
    c
    c
    ```

    The last output is `c`, because the last assignment (`a = b`) has no quotes. This last assignment sets the variable a equal to the value of the variable b, which is `'c'`.

12. `(int)symbol – (int)'0'`

    To see that this works, note that it works for `'0'`, and then see that it works for `'1'`, and then `'2'`, and so forth. You can use an actual number in place of `(int)'0'`, but `(int)'0'`is a bit easier to understand.

13.
    ```
    quotient = 2
    remainder = 1
    ```

14. `(1 / 2) * 2 is equal to 0.0`

    Because 1/2 is integer division, the part after the decimal point is discarded, producing 0 instead of 0.5.

15. `result is -10`

    The expression `–3 * 7 % 3– 4– 6` is equivalent to `(((–3 * 7) % 3) -4) - 6`

16. `result is 5`

17. The output would change to the following:

    ```
    Enter a whole number from 1 to 99
    I will find a combination of coins
    that equals that amount of change.
    87
    2 cents in coins can be given as:
    3 quarters
    1 dimes
    0 nickels and
    2 pennies
    ```

18.
    ```
    n is 3
    n is 2
    ```

19. How do you `doSeven of Nine`.
    Note that there is no space in `doSeven`.

20.
    ```
    7
    b
    ```

21. `defg`

22.
    ```
    abc
    def
    ```

23. `abc\ndef`

24. `HELLO JOHN`

25. False, because the strings are not equal.

26. True, because the strings are equal.

27.
```
System.out.println("Once upon a time,");
System.out.println("there were three little programmers.");
```

Since we did not specify where the next output goes, the second of these statements could use `print` instead of `println`.

28. The method `System.out.println` displays its output and then advances to the next line. Subsequent output would begin on this next line. In contrast, `System.out.println` displays its output but does not advance to the next line. Thus, subsequent output would begin on the same line.

29.
```
import java.util.Scanner;
public class Question29
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter two whole numbers:");
        int n1 = keyboard.nextInt( );
        int n2 = keyboard.nextInt( );
        System.out.println("You entered: " + n1 + " " + n2);
    }
}
```

30.
```
import java.util.Scanner;
public class Question30
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        String word1, word2, word3;
        System.out.println("Type three words on one line:");
        word1 = keyboard.next( );
        word2 = keyboard.next( );
        word3 = keyboard.next( );
        System.out.println("You typed the words");
        System.out.println(word1 + " " + word2 +" " + word3);
    }
}
```

31. Since the empty string is the second of the three strings, the output is

```
HelloJoe
```

32. There are `//` comments, `/* */` comments, and `/** */` comments. Everything following a `//` on the same line is a comment. Everything between a `/*` and a matching `*/` is a comment. Everything between a `/**` and a matching `*/` is a comment that is recognized by the program `javadoc`.

33. One

```
And hit it
```

34. Change the line

```
public static final double PI = 3.14159;
```

to

```
public static final double PI = 3.14;
```

Since values of type `double` are stored with only a limited amount of accuracy, you could argue that this is not "exactly" 3.14, but any legislator who is stupid enough to legislate the value of pi is unlikely to be aware of this subtlety.

35. `JOptionPane` is a class, `showInputDialog` is a method, and `"Enter number of apples:"` is an argument.

36. `JOptionPane.showMessageDialog(null, "I Love You.");`

37. `System.exit(0);`

38. It would compile. It would run. It would even appear to run with no problems. However, even after all the windows have disappeared, the program would still be running. It would continue to run and consume resources, and nothing in the Java code would ever end the program. You would have to use the operating system to end the program.

39.
```
import javax.swing.JOptionPane;
public class Question39
{
    public static void main(String[] args)
    {
        JOptionPane.showMessageDialog(null, "Hello World!");
        System.exit(0);
    }
}
```

40. You might want to convert `numberString` to an `int`, but that will not affect anything the user sees when the program is run.

```
import javax.swing.JOptionPane;
public class Question40
{
    public static void main(String[] args)
    {
        String numberString = JOptionPane.showInputDialog(
                              "Enter a whole number:");
        JOptionPane.showMessageDialog(null, "The number is "
                                      + numberString);
        System.exit(0);
    }
}
```