

# CHAPTER 2

## FUNDAMENTAL DATA TYPES

### CHAPTER GOALS

- To declare and initialize variables and constants
- To understand the properties and limitations of integers and floating-point numbers
- To appreciate the importance of comments and good code layout
- To write arithmetic expressions and assignment statements
- To create programs that read and process inputs, and display the results
- To learn how to use the Java String type

### CHAPTER CONTENTS

#### 2.1 VARIABLES 30

- Syntax 2.1:* Variable Declaration 31
- Syntax 2.2:* Assignment 34
- Syntax 2.3:* Constant Declaration 35
- Common Error 2.1:* Using Undeclared or Uninitialized Variables 37
- Programming Tip 2.1:* Choose Descriptive Variable Names 38
- Common Error 2.2:* Overflow 38
- Common Error 2.3:* Roundoff Errors 38
- Programming Tip 2.2:* Do Not Use Magic Numbers 39
- Special Topic 2.1:* Numeric Types in Java 39
- Special Topic 2.2:* Big Numbers 40

#### 2.2 ARITHMETIC 41

- Syntax 2.4:* Cast 44
- Common Error 2.4:* Unintended Integer Division 46
- Common Error 2.5:* Unbalanced Parentheses 46
- Programming Tip 2.3:* Spaces in Expressions 47
- Special Topic 2.3:* Combining Assignment and Arithmetic 47
- Video Example 2.1:* Using Integer Division +

- Random Fact 2.1:* The Pentium Floating-Point Bug 48

#### 2.3 INPUT AND OUTPUT 48

- Syntax 2.5:* Input Statement 49
- Programming Tip 2.4:* Use the API Documentation 53
- How To 2.1:* Carrying out Computations 54
- Worked Example 2.1:* Computing the Cost of Stamps +

#### 2.4 PROBLEM SOLVING: FIRST DO IT BY HAND 57

- Worked Example 2.2:* Computing Travel Time +

#### 2.5 STRINGS 59

- Special Topic 2.4:* Instance Methods and Static Methods 64
- Special Topic 2.5:* Using Dialog Boxes for Input and Output 65
- Video Example 2.2:* Computing Distances on Earth +
- Random Fact 2.2:* International Alphabets and Unicode 66





Numbers and character strings (such as the ones on this display board) are important data types in any Java program. In this chapter, you will learn how to work with numbers and text, and how to write simple programs that perform useful tasks with them.

## 2.1 Variables

When your program carries out computations, you will want to store values so that you can use them later. In a Java program, you use **variables** to store values. In this section, you will learn how to declare and use variables.

To illustrate the use of variables, we will develop a program that solves the following problem. Soft drinks are sold in cans and bottles. A store offers a six-pack of 12-ounce cans for the same price as a two-liter bottle. Which should you buy? (Twelve fluid ounces equal approximately 0.355 liters.)

In our program, we will declare variables for the number of cans per pack and for the volume of each can. Then we will compute the volume of a six-pack in liters and print out the answer.



*What contains more soda? A six-pack of 12-ounce cans or a two-liter bottle?*

### 2.1.1 Variable Declarations

The following statement declares a variable named `cansPerPack`:

```
int cansPerPack = 6;
```

A variable is a storage location with a name.

A **variable** is a storage location in a computer program. Each variable has a name and holds a value.

A variable is similar to a parking space in a parking garage. The parking space has an identifier (such as “J 053”), and it can hold a vehicle. A variable has a name (such as `cansPerPack`), and it can hold a value (such as 6).



*Like a variable in a computer program, a parking space has an identifier and a contents.*

## Syntax 2.1 Variable Declaration

**Syntax** `typeName variableName = value;`  
or  
`typeName variableName;`

Types introduced in this chapter are the number types `int` and `double` (page 32) and the `String` type (page 59).

`int` cansPerPack = 6;

See page 33 for rules and examples of valid names.

A variable declaration ends with a semicolon.

Supplying an initial value is optional, but it is usually a good idea. See page 37.

Use a descriptive variable name. See page 38.

When declaring a variable, you usually specify an initial value.

When declaring a variable, you also specify the type of its values.

When declaring a variable, you usually want to **initialize** it. That is, you specify the value that should be stored in the variable. Consider again this variable declaration:

```
int cansPerPack = 6;
```

The variable `cansPerPack` is initialized with the value 6.

Like a parking space that is restricted to a certain type of vehicle (such as a compact car, motorcycle, or electric vehicle), a variable in Java stores data of a specific **type**. Java supports quite a few data types: numbers, text strings, files, dates, and many others. You must specify the type whenever you declare a variable (see Syntax 2.1).

The `cansPerPack` variable is an **integer**, a whole number without a fractional part. In Java, this type is called `int`. (See the next section for more information about number types in Java.)

Note that the type comes before the variable name:

```
int cansPerPack = 6;
```

After you have declared and initialized a variable, you can use it. For example,



```
int cansPerPack = 6;
System.out.println(cansPerPack);
int cansPerCrate = 4 * cansPerPack;
```

Table 1 shows several examples of variable declarations.



*Each parking space is suitable for a particular type of vehicle, just as each variable holds a value of a particular type.*

Table 1 Variable Declarations in Java

Variable Name	Comment
<code>int cans = 6;</code>	Declares an integer variable and initializes it with 6.
<code>int total = cans + bottles;</code>	The initial value need not be a fixed value. (Of course, <code>cans</code> and <code>bottles</code> must have been previously declared.)
 <code>bottles = 1;</code>	<b>Error:</b> The type is missing. This statement is not a declaration but an assignment of a new value to an existing variable—see Section 2.1.4.
 <code>int volume = "2";</code>	<b>Error:</b> You cannot initialize a number with a string.
<code>int cansPerPack;</code>	Declares an integer variable without initializing it. This can be a cause for errors—see Common Error 2.1 on page 37.
<code>int dollars, cents;</code>	Declares two integer variables in a single statement. In this book, we will declare each variable in a separate statement.

### 2.1.2 Number Types



Use the `int` type for numbers that cannot have a fractional part.

In Java, there are several different types of numbers. You use the `int` type to denote a whole number without a fractional part. For example, there must be an integer number of cans in any pack of cans—you cannot have a fraction of a can.

When a fractional part is required (such as in the number 0.335), we use **floating-point numbers**. The most commonly used type for floating-point numbers in Java is called `double`. (If you want to know the reason, read Special Topic 2.1 on page 39.) Here is the declaration of a floating-point variable:

```
double canVolume = 0.335;
```

Table 2 Number Literals in Java

Number	Type	Comment
6	<code>int</code>	An integer has no fractional part.
-6	<code>int</code>	Integers can be negative.
0	<code>int</code>	Zero is an integer.
0.5	<code>double</code>	A number with a fractional part has type <code>double</code> .
1.0	<code>double</code>	An integer with a fractional part .0 has type <code>double</code> .
1E6	<code>double</code>	A number in exponential notation: $1 \times 10^6$ or 1000000. Numbers in exponential notation always have type <code>double</code> .
2.96E-2	<code>double</code>	Negative exponent: $2.96 \times 10^{-2} = 2.96 / 100 = 0.0296$
 100,000		<b>Error:</b> Do not use a comma as a decimal separator.
 3 1/2		<b>Error:</b> Do not use fractions; use decimal notation: 3.5

Use the `double` type for floating-point numbers.

When a value such as 6 or 0.335 occurs in a Java program, it is called a **number literal**. If a number literal has a decimal point, it is a floating-point number; otherwise, it is an integer. Table 2 shows how to write integer and floating-point literals in Java.

### 2.1.3 Variable Names

When you declare a variable, you should pick a name that explains its purpose. For example, it is better to use a descriptive name, such as `canVolume`, than a terse name, such as `cv`.

In Java, there are a few simple rules for variable names:

1. Variable names must start with a letter or the underscore (`_`) character, and the remaining characters must be letters, numbers, or underscores. (Technically, the `$` symbol is allowed as well, but you should not use it—it is intended for names that are automatically generated by tools.)
2. You cannot use other symbols such as `?` or `%`. Spaces are not permitted inside names either. You can use uppercase letters to denote word boundaries, as in `cansPerPack`. This naming convention is called *camel case* because the uppercase letters in the middle of the name look like the humps of a camel.)
3. Variable names are **case sensitive**, that is, `canVolume` and `canvolume` are different names.
4. You cannot use **reserved words** such as `double` or `class` as names; these words are reserved exclusively for their special Java meanings. (See Appendix C for a listing of all reserved words in Java.)








By convention, variable names should start with a lowercase letter.

It is a convention among Java programmers that variable names should start with a lowercase letter (such as `canVolume`) and class names should start with an uppercase letter (such as `HelloPrinter`). That way, it is easy to tell them apart.

Table 3 shows examples of legal and illegal variable names in Java.

**Table 3** Variable Names in Java

Variable Name	Comment
<code>canVolume1</code>	Variable names consist of letters, numbers, and the underscore character.
<code>x</code>	In mathematics, you use short variable names such as $x$ or $y$ . This is legal in Java, but not very common, because it can make programs harder to understand (see Programming Tip 2.1 on page 38).
 <code>CanVolume</code>	<b>Caution:</b> Variable names are case sensitive. This variable name is different from <code>canVolume</code> , and it violates the convention that variable names should start with a lowercase letter.
 <code>6pack</code>	<b>Error:</b> Variable names cannot start with a number.
 <code>can volume</code>	<b>Error:</b> Variable names cannot contain spaces.
 <code>double</code>	<b>Error:</b> You cannot use a reserved word as a variable name.
 <code>ltr/fl.oz</code>	<b>Error:</b> You cannot use symbols such as <code>/</code> or <code>.</code>



## 2.1.4 The Assignment Statement

An assignment statement stores a new value in a variable, replacing the previously stored value.



**ANIMATION**  
Variable Initialization  
and Assignment

The assignment operator = does *not* denote mathematical equality.

You use the **assignment statement** to place a new value into a variable. Here is an example:

```
cansPerPack = 8;
```

The left-hand side of an assignment statement consists of a variable. The right-hand side is an expression that has a value. That value is stored in the variable, overwriting its previous contents.

There is an important difference between a variable declaration and an assignment statement:

```
int cansPerPack = 6; Variable declaration
```

```
...  
cansPerPack = 8; Assignment statement
```

The first statement is the declaration of `cansPerPack`. It is an instruction to create a new variable of type `int`, to give it the name `cansPerPack`, and to initialize it with 6. The second statement is an *assignment statement*: an instruction to replace the contents of the *existing* variable `cansPerPack` with another value.

The = sign doesn't mean that the left-hand side is *equal* to the right-hand side. The expression on the right is evaluated, and its value is placed into the variable on the left.

Do not confuse this *assignment operation* with the = used in algebra to denote *equality*. The assignment operator is an instruction to do something—namely, place a value into a variable. The mathematical equality states that two values are equal.

For example, in Java, it is perfectly legal to write

```
totalVolume = totalVolume + 2;
```

It means to look up the value stored in the variable `totalVolume`, add 2 to it, and place the result back into `totalVolume`. (See Figure 1.) The net effect of executing this statement is to increment `totalVolume` by 2. For example, if `totalVolume` was 2.13 before execution of the statement, it is set to 4.13 afterwards. Of course, in mathematics it would make no sense to write that  $x = x + 2$ . No value can equal itself plus 2.

## Syntax 2.2 Assignment

**Syntax** *variableName = value;*

This is an initialization of a new variable, NOT an assignment.

The name of a previously defined variable

```
double total = 0;
```

```
:
```

```
:
```

```
total = bottles * BOTTLE_VOLUME;
```

```
:
```

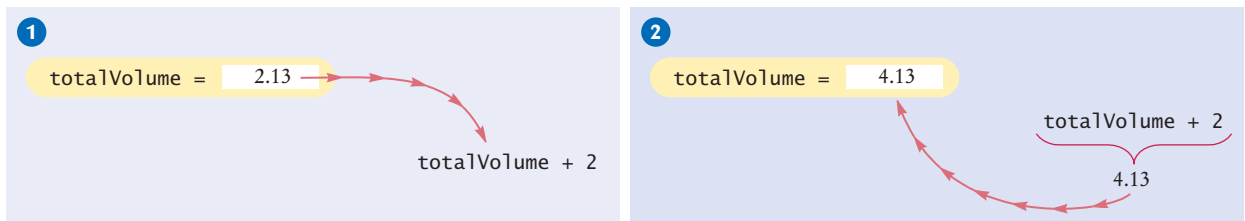
```
:
```

```
total = total + cans * CAN_VOLUME;
```

This is an assignment.

The expression that replaces the previous value

The same name can occur on both sides.  
See Figure 1.



**Figure 1** Executing the Assignment `totalVolume = totalVolume + 2`

### 2.1.5 Constants

You cannot change the value of a variable that is defined as `final`.

When a variable is defined with the reserved word `final`, its value can never change. Constants are commonly written using capital letters to distinguish them visually from regular variables:

```
final double BOTTLE_VOLUME = 2;
```

It is good programming style to use named constants in your program to explain the meanings of numeric values. For example, compare the statements

```
double totalVolume = bottles * 2;
```

and

```
double totalVolume = bottles * BOTTLE_VOLUME;
```

A programmer reading the first statement may not understand the significance of the number 2. The second statement, with a named constant, makes the computation much clearer.

### Syntax 2.3 Constant Declaration

**Syntax** `final typeName variableName = expression;`

The `final` reserved word indicates that this value cannot be modified.

```
final double CAN_VOLUME = 0.355; // Liters in a 12-ounce can
```

Use uppercase letters for constants.

This comment explains how the value for the constant was determined.

### 2.1.6 Comments

As your programs get more complex, you should add **comments**, explanations for human readers of your code. For example, here is a comment that explains the value used in a variable initialization:

```
final double CAN_VOLUME = 0.355; // Liters in a 12-ounce can
```

This comment explains the significance of the value 0.355 to a human reader. The compiler does not process comments at all. It ignores everything from a `//` delimiter to the end of the line.

Use comments to add explanations for humans who read your code. The compiler ignores comments.

It is a good practice to provide comments. This helps programmers who read your code understand your intent. In addition, you will find comments helpful when you review your own programs.

You use the `//` delimiter for short comments. If you have a longer comment, enclose it between `/*` and `*/` delimiters. The compiler ignores these delimiters and everything in between. For example,

```
/*
    There are approximately 0.335 liters in a 12-ounce can because one ounce
    equals 0.02957353 liter; see The International Systems of Units (SI) - Conversion
    Factors for General Use (NIST Special Publication 1038).
*/
```

Finally, start a comment that explains the purpose of a program with the `/**` delimiter instead of `/*`. Tools that analyze source files rely on that convention. For example,

```
/**
    This program computes the volume (in liters) of a six-pack of soda cans.
*/
```

The following program shows the use of variables, constants, and the assignment statement. The program displays the volume of a six-pack of cans and the total volume of the six-pack and a two-liter bottle. We use constants for the can and bottle volumes. The `totalVolume` variable is initialized with the volume of the cans. Using an assignment statement, we add the bottle volume. As you can see from the program output, the six-pack of cans contains over two liters of soda.

### section\_1/Volume1.java

```
1  /**
2   This program computes the volume (in liters) of a six-pack of soda
3   cans and the total volume of a six-pack and a two-liter bottle.
4   */
5   public class Volume1
6   {
7       public static void main(String[] args)
8       {
9           int cansPerPack = 6;
10          final double CAN_VOLUME = 0.355; // Liters in a 12-ounce can
11          double totalVolume = cansPerPack * CAN_VOLUME;
12
13          System.out.print("A six-pack of 12-ounce cans contains ");
14          System.out.print(totalVolume);
15          System.out.println(" liters.");
16
17          final double BOTTLE_VOLUME = 2; // Two-liter bottle
18
19          totalVolume = totalVolume + BOTTLE_VOLUME;
20
21          System.out.print("A six-pack and a two-liter bottle contain ");
22          System.out.print(totalVolume);
23          System.out.println(" liters.");
24      }
25  }
```

### Program Run

```
A six-pack of 12-ounce cans contains 2.13 liters.
A six-pack and a two-liter bottle contain 4.13 liters.
```



*Just as a television commentator explains the news, you use comments in your program to explain its behavior.*



1. Declare a variable suitable for holding the number of bottles in a case.
2. What is wrong with the following variable declaration?  

```
int ounces per liter = 28.35
```
3. Declare and initialize two variables, `unitPrice` and `quantity`, to contain the unit price of a single bottle and the number of bottles purchased. Use reasonable initial values.
4. Use the variables declared in Self Check 3 to display the total purchase price.
5. Some drinks are sold in four-packs instead of six-packs. How would you change the `Volume1.java` program to compute the total volume?
6. What is wrong with this comment?  

```
double canVolume = 0.355; /* Liters in a 12-ounce can //
```
7. Suppose the type of the `cansPerPack` variable in `Volume1.java` was changed from `int` to `double`. What would be the effect on the program?
8. Why can't the variable `totalVolume` in the `Volume1.java` program be declared as `final`?
9. How would you explain assignment using the parking space analogy?

**Practice It** Now you can try these exercises at the end of the chapter: R2.1, R2.2, P2.1.

### Common Error 2.1



### Using Undeclared or Uninitialized Variables

You must declare a variable before you use it for the first time. For example, the following sequence of statements would not be legal:

```
double canVolume = 12 * literPerOunce; // ERROR: literPerOunce is not yet declared
double literPerOunce = 0.0296;
```

In your program, the statements are compiled in order. When the compiler reaches the first statement, it does not know that `literPerOunce` will be declared in the next line, and it reports an error. The remedy is to reorder the declarations so that each variable is declared before it is used.

A related error is to leave a variable uninitialized:

```
int bottles;
int bottleVolume = bottles * 2; // ERROR: bottles is not yet initialized
```

The Java compiler will complain that you are using a variable that has not yet been given a value. The remedy is to assign a value to the variable before it is used.

### Programming Tip 2.1



### Choose Descriptive Variable Names

We could have saved ourselves a lot of typing by using shorter variable names, as in

```
double cv = 0.355;
```

Compare this declaration with the one that we actually used, though. Which one is easier to read? There is no comparison. Just reading `canVolume` is a lot less trouble than reading `cv` and then *figuring out* it must mean “can volume”.

In practical programming, this is particularly important when programs are written by more than one person. It may be obvious to *you* that `cv` stands for can volume and not current velocity, but will it be obvious to the person who needs to update your code years later? For that matter, will you remember yourself what `cv` means when you look at the code three months from now?

### Common Error 2.2



### Overflow

Because numbers are represented in the computer with a limited number of digits, they cannot represent arbitrary numbers.

The `int` type has a *limited range*: It can represent numbers up to a little more than two billion. For many applications, this is not a problem, but you cannot use an `int` to represent the world population.

If a computation yields a value that is outside the `int` range, the result *overflows*. No error is displayed. Instead, the result is truncated, yielding a useless value. For example,

```
int fiftyMillion = 50000000;
System.out.println(100 * fiftyMillion); // Expected: 5000000000
```

displays 705032704.

In situations such as this, you can switch to `double` values. However, read Common Error 2.3 for more information about a related issue: roundoff errors.

### Common Error 2.3



### Roundoff Errors

Roundoff errors are a fact of life when calculating with floating-point numbers. You probably have encountered that phenomenon yourself with manual calculations. If you calculate  $1/3$  to two decimal places, you get 0.33. Multiplying again by 3, you obtain 0.99, not 1.00.

In the processor hardware, numbers are represented in the binary number system, using only digits 0 and 1. As with decimal numbers, you can get roundoff errors when binary digits are lost. They just may crop up at different places than you might expect.

Here is an example:

```
double price = 4.35;
double quantity = 100;
double total = price * quantity; // Should be 100 * 4.35 = 435
System.out.println(total); // Prints 434.9999999999999
```

In the binary system, there is no exact representation for 4.35, just as there is no exact representation for  $1/3$  in the decimal system. The representation used by the computer is just a little less than 4.35, so 100 times that value is just a little less than 435.

You can deal with roundoff errors by rounding to the nearest integer (see Section 2.2.5) or by displaying a fixed number of digits after the decimal separator (see Section 2.3.2).

### Programming Tip 2.2



### Do Not Use Magic Numbers

A **magic number** is a numeric constant that appears in your code without explanation. For example,

```
totalVolume = bottles * 2;
```

Why 2? Are bottles twice as voluminous as cans? No, the reason is that every bottle contains 2 liters. Use a named constant to make the code self-documenting:

```
final double BOTTLE_VOLUME = 2;
totalVolume = bottles * BOTTLE_VOLUME;
```

There is another reason for using named constants. Suppose circumstances change, and the bottle volume is now 1.5 liters. If you used a named constant, you make a single change, and you are done. Otherwise, you have to look at every value of 2 in your program and ponder whether it meant a bottle volume, or something else. In a program that is more than a few pages long, that is incredibly tedious and error-prone.

Even the most reasonable cosmic constant is going to change one day. You think there are seven days per week? Your customers on Mars are going to be pretty unhappy about your silly prejudice. Make a constant

```
final int DAYS_PER_WEEK = 7;
```



*We prefer programs that are easy to understand over those that appear to work by magic.*

### Special Topic 2.1



### Numeric Types in Java

In addition to the `int` and `double` types, Java has several other numeric types.

Java has two floating-point types. The `float` type uses half the storage of the `double` type that we use in this book, but it can only store about 7 decimal digits. (In the computer, numbers are represented in the binary number system, using digits 0 and 1.) Many years ago, when computers had far less memory than they have today, `float` was the standard type for floating-point computations, and programmers would indulge in the luxury of “double precision” only when they needed the additional digits. Today, the `float` type is rarely used.

By the way, these numbers are called “floating-point” because of their internal representation in the computer. Consider numbers 29600, 2.96, and 0.0296. They can be represented in a very similar way: namely, as a sequence of the significant digits—296—and an indication of the position of the decimal point. When the values are multiplied or divided by 10, only the

position of the decimal point changes; it “floats”. Computers use base 2, not base 10, but the principle is the same.

In addition to the `int` type, Java has integer types `byte`, `short`, and `long`. Their ranges are shown in Table 4. (Their strange-looking limits are related to powers of 2, another consequence of the fact that computers use binary numbers.)

Table 4 Java Number Types		
Type	Description	Size
<code>int</code>	The integer type, with range $-2,147,483,648$ ( <code>Integer.MIN_VALUE</code> ) ... $2,147,483,647$ ( <code>Integer.MAX_VALUE</code> , about 2.14 billion)	4 bytes
<code>byte</code>	The type describing a single byte consisting of 8 bits, with range $-128 \dots 127$	1 byte
<code>short</code>	The short integer type, with range $-32,768 \dots 32,767$	2 bytes
<code>long</code>	The long integer type, with about 19 decimal digits	8 bytes
<code>double</code>	The double-precision floating-point type, with about 15 decimal digits and a range of about $\pm 10^{308}$	8 bytes
<code>float</code>	The single-precision floating-point type, with about 7 decimal digits and a range of about $\pm 10^{38}$	4 bytes
<code>char</code>	The character type, representing code units in the Unicode encoding scheme (see Random Fact 2.2)	2 bytes

Special Topic 2.2



Big Numbers

If you want to compute with really large numbers, you can use big number objects. Big number objects are objects of the `BigInteger` and `BigDecimal` classes in the `java.math` package. Unlike the number types such as `int` or `double`, big number objects have essentially no limits on their size and precision. However, computations with big number objects are much slower than those that involve number types. Perhaps more importantly, you can’t use the familiar arithmetic operators such as `(+ - *)` with them. Instead, you have to use methods called `add`, `subtract`, and `multiply`. Here is an example of how to create a `BigInteger` object and how to call the `multiply` method:

```
BigInteger oneHundred = new BigInteger("100");
BigInteger fiftyMillion = new BigInteger("50000000");
System.out.println(oneHundred.multiply(fiftyMillion)); // Prints 5000000000
```

The `BigDecimal` type carries out floating-point computations without roundoff errors. For example,

```
BigDecimal price = new BigDecimal("4.35");
BigDecimal quantity = new BigDecimal("100");
BigDecimal total = price.multiply(quantity);
System.out.println(total); // Prints 435.00
```

## 2.2 Arithmetic

In the following sections, you will learn how to carry out arithmetic calculations in Java.

### 2.2.1 Arithmetic Operators



Java supports the same four basic arithmetic operations as a calculator—addition, subtraction, multiplication, and division—but it uses different symbols for multiplication and division.

You must write  $a * b$  to denote multiplication. Unlike in mathematics, you cannot write  $a \cdot b$ ,  $a \cdot b$ , or  $a \times b$ . Similarly, division is always indicated with a  $/$ , never  $\div$  or a fraction bar.

For example,  $\frac{a+b}{2}$  becomes  $(a + b) / 2$ .

The combination of variables, literals, operators, and/or method calls is called an **expression**. For example,  $(a + b) / 2$  is an expression.

Parentheses are used just as in algebra: to indicate in which order the parts of the expression should be computed. For example, in the expression  $(a + b) / 2$ , the sum  $a + b$  is computed first, and then the sum is divided by 2. In contrast, in the expression

$$a + b / 2$$

only  $b$  is divided by 2, and then the sum of  $a$  and  $b / 2$  is formed. As in regular algebraic notation, multiplication and division have a *higher precedence* than addition and subtraction. For example, in the expression  $a + b / 2$ , the  $/$  is carried out first, even though the  $+$  operation occurs further to the left.

If you mix integer and floating-point values in an arithmetic expression, the result is a floating-point value. For example,  $7 + 4.0$  is the floating-point value 11.0.

Mixing integers and floating-point values in an arithmetic expression yields a floating-point value.

### 2.2.2 Increment and Decrement

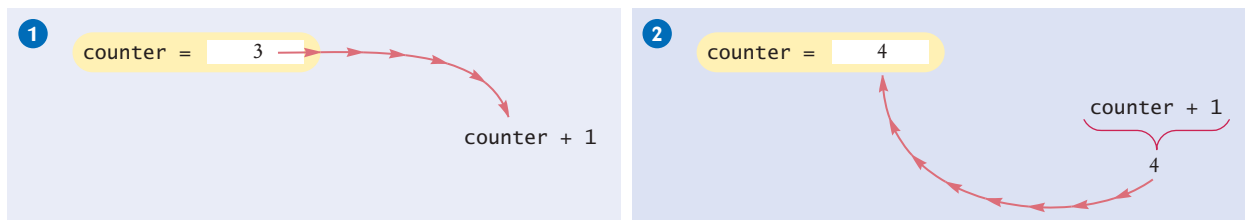
The `++` operator adds 1 to a variable; the `--` operator subtracts 1.

Changing a variable by adding or subtracting 1 is so common that there is a special shorthand for it. The `++` operator increments a variable—see Figure 2:

```
counter++; // Adds 1 to the variable counter
```

Similarly, the `--` operator decrements a variable:

```
counter--; // Subtracts 1 from counter
```



**Figure 2** Incrementing a Variable

### 2.2.3 Integer Division and Remainder

If both arguments of `/` are integers, the remainder is discarded.

Division works as you would expect, as long as at least one of the numbers involved is a floating-point number. That is,

`7.0 / 4.0`  
`7 / 4.0`  
`7.0 / 4`

all yield 1.75. However, if *both* numbers are integers, then the result of the division is always an integer, with the remainder discarded. That is,

`7 / 4`

evaluates to 1 because 7 divided by 4 is 1 with a remainder of 3 (which is discarded). This can be a source of subtle programming errors—see Common Error 2.4.

If you are interested in the remainder only, use the `%` operator:

`7 % 4`

is 3, the remainder of the integer division of 7 by 4. The `%` symbol has no analog in algebra. It was chosen because it looks similar to `/`, and the remainder operation is related to division. The operator is called **modulus**. (Some people call it *modulo* or *mod*.) It has no relationship with the percent operation that you find on some calculators.

Here is a typical use for the integer `/` and `%` operations. Suppose you have an amount of pennies in a piggybank:

`int pennies = 1729;`

You want to determine the value in dollars and cents. You obtain the dollars through an integer division by 100:

`int dollars = pennies / 100; // Sets dollars to 17`

The integer division discards the remainder. To obtain the remainder, use the `%` operator:

`int cents = pennies % 100; // Sets cents to 29`

See Table 5 for additional examples.



*Integer division and the `%` operator yield the dollar and cent values of a piggybank full of pennies.*

The `%` operator computes the remainder of an integer division.

Table 5 Integer Division and Remainder

Expression (where $n = 1729$ )	Value	Comment
<code>n % 10</code>	9	<code>n % 10</code> is always the last digit of $n$ .
<code>n / 10</code>	172	This is always $n$ without the last digit.
<code>n % 100</code>	29	The last two digits of $n$ .
<code>n / 10.0</code>	172.9	Because 10.0 is a floating-point number, the fractional part is not discarded.
<code>-n % 10</code>	-9	Because the first argument is negative, the remainder is also negative.
<code>n % 2</code>	1	<code>n % 2</code> is 0 if $n$ is even, 1 or -1 if $n$ is odd.



2.2.4 Powers and Roots

The Java library declares many mathematical functions, such as `Math.sqrt` (square root) and `Math.pow` (raising to a power).

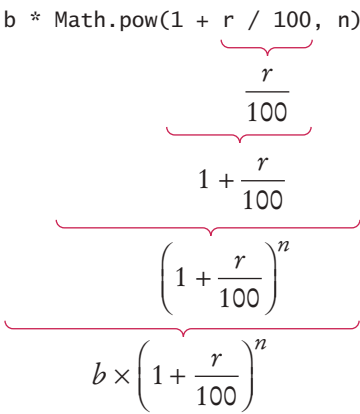
In Java, there are no symbols for powers and roots. To compute them, you must call methods. To take the square root of a number, you use the `Math.sqrt` method. For example,  $\sqrt{x}$  is written as `Math.sqrt(x)`. To compute  $x^n$ , you write `Math.pow(x, n)`.  
In algebra, you use fractions, exponents, and roots to arrange expressions in a compact two-dimensional form. In Java, you have to write all expressions in a linear arrangement. For example, the mathematical expression

$$b \times \left(1 + \frac{r}{100}\right)^n$$

becomes

```
b * Math.pow(1 + r / 100, n)
```

Figure 3 shows how to analyze such an expression. Table 6 shows additional mathematical methods.



**Figure 3**  
Analyzing an Expression

**Table 6** Mathematical Methods

Method	Returns
<code>Math.sqrt(x)</code>	Square root of $x$ ( $\geq 0$ )
<code>Math.pow(x, y)</code>	$x^y$ ( $x > 0$ , or $x = 0$ and $y > 0$ , or $x < 0$ and $y$ is an integer)
<code>Math.sin(x)</code>	Sine of $x$ ( $x$ in radians)
<code>Math.cos(x)</code>	Cosine of $x$
<code>Math.tan(x)</code>	Tangent of $x$
<code>Math.toRadians(x)</code>	Convert $x$ degrees to radians (i.e., returns $x \cdot \pi/180$ )
<code>Math.toDegrees(x)</code>	Convert $x$ radians to degrees (i.e., returns $x \cdot 180/\pi$ )
<code>Math.exp(x)</code>	$e^x$
<code>Math.log(x)</code>	Natural log ( $\ln(x)$ , $x > 0$ )

Table 6 Mathematical Methods	
Method	Returns
Math.log10(x)	Decimal log ( $\log_{10}(x)$ , $x > 0$ )
Math.round(x)	Closest integer to $x$ (as a long)
Math.abs(x)	Absolute value $ x $
Math.max(x, y)	The larger of $x$ and $y$
Math.min(x, y)	The smaller of $x$ and $y$

### 2.2.5 Converting Floating-Point Numbers to Integers

Occasionally, you have a value of type `double` that you need to convert to the type `int`. It is an error to assign a floating-point value to an integer:

```
double balance = total + tax;
int dollars = balance; // Error: Cannot assign double to int
```

The compiler disallows this assignment because it is potentially dangerous:

- The fractional part is lost.
- The magnitude may be too large. (The largest integer is about 2 billion, but a floating-point number can be much larger.)

You must use the **cast** operator (`int`) to convert a floating-point value to an integer. Write the cast operator before the expression that you want to convert:

```
double balance = total + tax;
int dollars = (int) balance;
```

The cast (`int`) converts the floating-point value `balance` to an integer by discarding the fractional part. For example, if `balance` is 13.75, then `dollars` is set to 13.

When applying the cast operator to an arithmetic expression, you need to place the expression inside parentheses:

```
int dollars = (int) (total + tax);
```

You use a cast (*typeName*) to convert a value to a different type.

### Syntax 2.4 Cast

Syntax

*(typeName) expression*

This is the type of the expression after casting.

These parentheses are a part of the cast operator.

`(int) (balance * 100)`

Use parentheses here if the cast is applied to an expression with arithmetic operators.

ONLINE EXAMPLE

⊕ A program demonstrating casts, rounding, and the % operator.

Discarding the fractional part is not always appropriate. If you want to round a floating-point number to the nearest whole number, use the `Math.round` method. This method returns a `long` integer, because large floating-point numbers cannot be stored in an `int`.

```
long rounded = Math.round(balance);
```

If `balance` is 13.75, then `rounded` is set to 14.

If you know that the result can be stored in an `int` and does not require a `long`, you can use a cast:

```
int rounded = (int) Math.round(balance);
```

Table 7 Arithmetic Expressions

Mathematical Expression	Java Expression	Comments
$\frac{x + y}{2}$	<code>(x + y) / 2</code>	The parentheses are required; <code>x + y / 2</code> computes <code>x + <math>\frac{y}{2}</math></code> .
$\frac{xy}{2}$	<code>x * y / 2</code>	Parentheses are not required; operators with the same precedence are evaluated left to right.
$\left(1 + \frac{r}{100}\right)^n$	<code>Math.pow(1 + r / 100, n)</code>	Use <code>Math.pow(x, n)</code> to compute <code>x<sup>n</sup></code> .
$\sqrt{a^2 + b^2}$	<code>Math.sqrt(a * a + b * b)</code>	<code>a * a</code> is simpler than <code>Math.pow(a, 2)</code> .
$\frac{i + j + k}{3}$	<code>(i + j + k) / 3.0</code>	If <code>i</code> , <code>j</code> , and <code>k</code> are integers, using a denominator of 3.0 forces floating-point division.
$\pi$	<code>Math.PI</code>	<code>Math.PI</code> is a constant declared in the <code>Math</code> class.



- 10. A bank account earns interest once per year. In Java, how do you compute the interest earned in the first year? Assume variables `percent` and `balance` of type `double` have already been declared.
- 11. In Java, how do you compute the side length of a square whose area is stored in the variable `area`?
- 12. The volume of a sphere is given by

$$V = \frac{4}{3}\pi r^3$$

If the radius is given by a variable `radius` of type `double`, write a Java expression for the volume.

- 13. What is the value of `1729 / 10` and `1729 % 10`?
- 14. If `n` is a positive number, what is `(n / 10) % 10`?

**Practice It** Now you can try these exercises at the end of the chapter: R2.3, R2.5, P2.4, P2.25.

## Common Error 2.4

**Unintended Integer Division**

It is unfortunate that Java uses the same symbol, namely /, for both integer and floating-point division. These are really quite different operations. It is a common error to use integer division by accident. Consider this segment that computes the average of three integers.

```
int score1 = 10;
int score2 = 4;
int score3 = 9;

double average = (score1 + score2 + score3) / 3; // Error
System.out.println("Average score: " + average); // Prints 7.0, not 7.666666666666667
```

What could be wrong with that? Of course, the average of score1, score2, and score3 is

$$\frac{\text{score1} + \text{score2} + \text{score3}}{3}$$

Here, however, the / does not mean division in the mathematical sense. It denotes integer division because both score1 + score2 + score3 and 3 are integers. Because the scores add up to 23, the average is computed to be 7, the result of the integer division of 23 by 3. That integer 7 is then moved into the floating-point variable average. The remedy is to make the numerator or denominator into a floating-point number:

```
double total = score1 + score2 + score3;
double average = total / 3;

or

double average = (score1 + score2 + score3) / 3.0;
```

## Common Error 2.5

**Unbalanced Parentheses**

Consider the expression

$$((a + b) * t / 2 * (1 - t))$$

What is wrong with it? Count the parentheses. There are three ( and two ). The parentheses are *unbalanced*. This kind of typing error is very common with complicated expressions. Now consider this expression.

$$(a + b) * t) / (2 * (1 - t))$$

This expression has three ( and three ), but it still is not correct. In the middle of the expression,

$$(a + b) * t) / (2 * (1 - t))$$

↑

there is only one ( but two ), which is an error. In the middle of an expression, the count of ( must be greater than or equal to the count of ), and at the end of the expression the two counts must be the same.

Here is a simple trick to make the counting easier without using pencil and paper. It is difficult for the brain to keep two counts simultaneously. Keep only one count when scanning the expression. Start with 1 at the first opening parenthesis, add 1 whenever you see an opening parenthesis, and subtract one whenever you see a closing parenthesis. Say the numbers aloud as you scan the



expression. If the count ever drops below zero, or is not zero at the end, the parentheses are unbalanced. For example, when scanning the previous expression, you would mutter

$$\begin{array}{cccc} (a + b) * t) / (2 * (1 - t) \\ 1 \quad 0 \quad -1 \end{array}$$

and you would find the error.

### Programming Tip 2.3



### Spaces in Expressions

It is easier to read

```
x1 = (-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a);
```

than

```
x1=(-b+Math.sqrt(b*b-4*a*c))/(2*a);
```

Simply put spaces around all operators + - \* / % =. However, don't put a space after a *unary* minus: a - used to negate a single quantity, such as -b. That way, it can be easily distinguished from a *binary* minus, as in a - b.

It is customary not to put a space after a method name. That is, write `Math.sqrt(x)` and not `Math.sqrt (x)`.

### Special Topic 2.3



### Combining Assignment and Arithmetic

In Java, you can combine arithmetic and assignment. For example, the instruction

```
total += cans;
```

is a shortcut for

```
total = total + cans;
```

Similarly,

```
total *= 2;
```

is another way of writing

```
total = total * 2;
```

Many programmers find this a convenient shortcut. If you like it, go ahead and use it in your own code. For simplicity, we won't use it in this book, though.

### VIDEO EXAMPLE 2.1



### Using Integer Division

A punch recipe calls for a given amount of orange soda. In this Video Example, you will see how to compute the required number of 12-ounce cans, using integer division.



⊕ Available online in WileyPLUS and at [www.wiley.com/college/horstmann](http://www.wiley.com/college/horstmann).



### Random Fact 2.1 The Pentium Floating-Point Bug

In 1994, Intel Corporation released what was then its most powerful processor, the Pentium. Unlike previous generations of its processors, it had a very fast floating-point unit. Intel's goal was to compete aggressively with the makers of higher-end processors for engineering workstations. The Pentium was a huge success immediately.

In the summer of 1994, Dr. Thomas Nicely of Lynchburg College in Virginia ran an extensive set of computations to analyze the sums of reciprocals of certain sequences of prime numbers. The results were not always what his theory predicted, even after he took into account the inevitable roundoff errors. Then Dr. Nicely noted that the same program did produce the correct results when running on the slower 486 processor that preceded the Pentium in Intel's lineup. This should not have happened. The optimal round-off behavior of floating-point calculations has been standardized by the Institute for Electrical and Electronic Engineers (IEEE) and Intel claimed to adhere to the IEEE standard in both the 486 and the Pentium processors. Upon further checking, Dr. Nicely discovered that indeed there was a very small set of numbers for which the product of two numbers was computed differently on the two processors. For example,

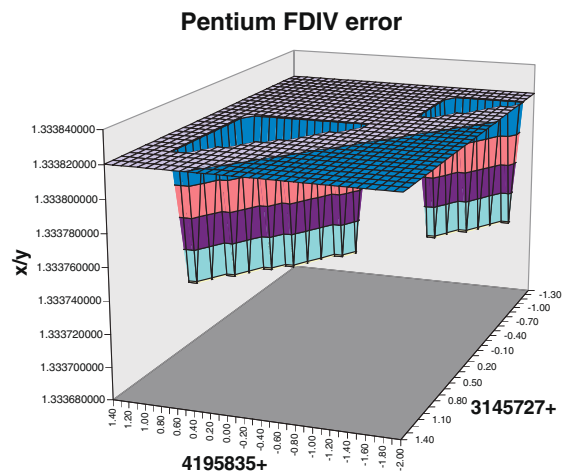
$$4,195,835 - ((4,195,835/3,145,727) \times 3,145,727)$$

is mathematically equal to 0, and it did compute as 0 on a 486 processor. On his Pentium processor the result was 256.

As it turned out, Intel had independently discovered the bug in its testing and had started to produce chips that fixed it. The bug was caused by an error in a table that was used to speed up the floating-point multiplication algorithm of the processor. Intel determined that the problem was exceedingly rare. They claimed that under normal use, a typical consumer would only notice the problem once every 27,000 years. Unfortunately for Intel, Dr. Nicely had not been a normal user.

Now Intel had a real problem on its hands. It figured that the cost of replacing all Pentium processors that it had sold so far would cost a great deal of money. Intel already had more orders for the chip than it could produce, and it would be particularly galling to have to give out the scarce chips as free replacements instead of selling them. Intel's management decided to punt on the issue and initially offered to replace the processors only for those customers who could prove that their work required absolute precision in mathematical calculations. Naturally, that did not go over well with the hundreds of thousands of customers who had paid retail prices of \$700 and more for a Pentium chip and did not want to live with the nagging feeling that perhaps, one day, their income tax program would produce a faulty return.

Ultimately, Intel caved in to public demand and replaced all defective chips, at a cost of about 475 million dollars.



*This graph shows a set of numbers for which the original Pentium processor obtained the wrong quotient.*

## 2.3 Input and Output

In the following sections, you will see how to read user input and how to control the appearance of the output that your programs produce.

### 2.3.1 Reading Input

You can make your programs more flexible if you ask the program user for inputs rather than using fixed values. Consider, for example, a program that processes prices





A supermarket scanner reads bar codes. The Java Scanner reads numbers and text.

and quantities of soda containers. Prices and quantities are likely to fluctuate. The program user should provide them as inputs.

When a program asks for user input, it should first print a message that tells the user which input is expected. Such a message is called a **prompt**.

```
System.out.print("Please enter the number of bottles: "); // Display prompt
```

Use the print method, not println, to display the prompt. You want the input to appear after the colon, not on the following line. Also remember to leave a space after the colon.

Because output is sent to System.out, you might think that you use System.in for input. Unfortunately, it isn't quite that simple. When Java was first designed, not much attention was given to reading keyboard input. It was assumed that all programmers would produce graphical user interfaces with text fields and menus. System.in was given a minimal set of features and must be combined with other classes to be useful.

To read keyboard input, you use a class called Scanner. You obtain a Scanner *object* by using the following statement:

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

You will learn more about objects and classes in Chapter 8. For now, simply include this statement whenever you want to read keyboard input.

When using the Scanner class, you need to carry out another step: import the class from its **package**. A package is a collection of classes with a related purpose. All classes in the Java library are contained in packages. The System class belongs to the package java.lang. The Scanner class belongs to the package java.util.

Only the classes in the java.lang package are automatically available in your programs. To use the Scanner class from the java.util package, place the following declaration at the top of your program file:

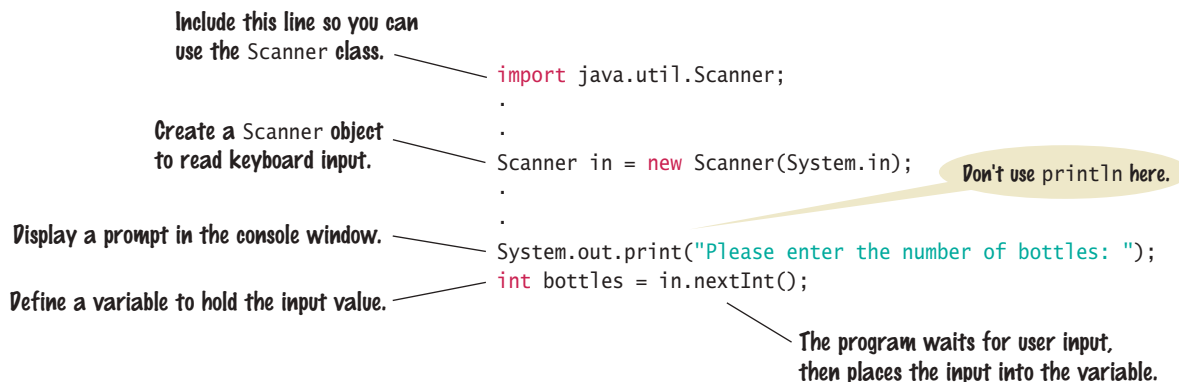
```
import java.util.Scanner;
```

Once you have a scanner, you use its nextInt method to read an integer value:

```
System.out.print("Please enter the number of bottles: ");
int bottles = in.nextInt();
```

Java classes are grouped into packages. Use the import statement to use classes from packages.

## Syntax 2.5 Input Statement



Use the Scanner class to read keyboard input in a console window.

When the `nextInt` method is called, the program waits until the user types a number and presses the Enter key. After the user supplies the input, the number is placed into the `bottles` variable, and the program continues.

To read a floating-point number, use the `nextDouble` method instead:

```
System.out.print("Enter price: ");
double price = in.nextDouble();
```

2.3.2 Formatted Output

When you print the result of a computation, you often want to control its appearance. For example, when you print an amount in dollars and cents, you usually want it to be rounded to two significant digits. That is, you want the output to look like

```
Price per liter: 1.22
```

instead of

```
Price per liter: 1.215962441314554
```

The following command displays the price with two digits after the decimal point:

```
System.out.printf("%.2f", price);
```

You can also specify a *field width*:

```
System.out.printf("%10.2f", price);
```

The price is printed using ten characters: six spaces followed by the four characters 1.22.



The construct `%10.2f` is called a *format specifier*: it describes how a value should be formatted. The letter `f` at the end of the format specifier indicates that we are displaying a floating-point number. Use `d` for an integer and `s` for a string; see Table 8 for examples.

Table 8 Format Specifier Examples

Format String	Sample Output	Comments
<code>"%d"</code>	24	Use <code>d</code> with an integer.
<code>"%5d"</code>	24	Spaces are added so that the field width is 5.
<code>"Quantity:%5d"</code>	Quantity: 24	Characters inside a format string but outside a format specifier appear in the output.
<code>"%f"</code>	1.21997	Use <code>f</code> with a floating-point number.
<code>"%.2f"</code>	1.22	Prints two digits after the decimal point.
<code>"%7.2f"</code>	1.22	Spaces are added so that the field width is 7.
<code>"%s"</code>	Hello	Use <code>s</code> with a string.
<code>"%d %.2f"</code>	24 1.22	You can format multiple values at once.

Use the `printf` method to specify how values should be formatted.

You use the `printf` method to line up your output in neat columns.

	Commencement			Term	Expiration	
	Month	Day	Year		Month	Day
Orneunard	June	4	1920	5-yr	June	4
Orneunard	April	24	1920	5-yr	April	24
Isager	Mar	14	1925	5-yr	Mar	14
Isager	Feb.	9	1925	5-yr	Mar	14
Isager	Oct	20	1925	5-yr	Oct	20
Isager	Mar	15	1925	5-yr	Mar	15
Quin	Nov.	14	1924	5-yr	Nov	14
Sr.	Sept	5	1925	5-yr	Sept	5
Donge	May	22	1926	5-yr	May	22
I	Mar	Ch.	1925		Oct	25
Isager	Mar	14	1928	5-yr	Mar	14
I	Feb	24	1928	5-yr	Feb	24

A format string contains format specifiers and literal characters. Any characters that are not format specifiers are printed verbatim. For example, the command

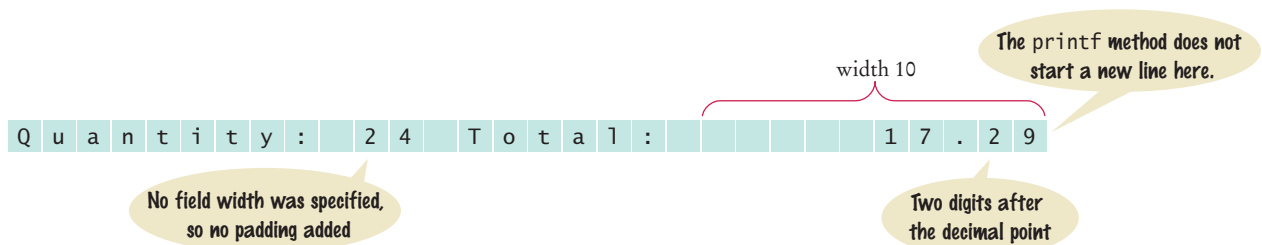
```
System.out.printf("Price per liter:%10.2f", price);
```

prints

```
Price per liter:      1.22
```

You can print multiple values with a single call to the `printf` method. Here is a typical example:

```
System.out.printf("Quantity: %d Total: %10.2f", quantity, total);
```



The `printf` method, like the `print` method, does not start a new line after the output. If you want the next output to be on a separate line, you can call `System.out.println()`. Alternatively, Section 2.5.4 shows you how to add a newline character to the format string.

Our next example program will prompt for the price of a six-pack and the volume of each can, then print out the price per ounce. The program puts to work what you just learned about reading input and formatting output.

### section\_3/Volume2.java

```
1 import java.util.Scanner;
2
3 /**
4  * This program prints the price per ounce for a six-pack of cans.
5  */
6 public class Volume2
7 {
8     public static void main(String[] args)
9     {
```

```

10 // Read price per pack
11
12 Scanner in = new Scanner(System.in);
13
14 System.out.print("Please enter the price for a six-pack: ");
15 double packPrice = in.nextDouble();
16
17 // Read can volume
18
19 System.out.print("Please enter the volume for each can (in ounces): ");
20 double canVolume = in.nextDouble();
21
22 // Compute pack volume
23
24 final double CANS_PER_PACK = 6;
25 double packVolume = canVolume * CANS_PER_PACK;
26
27 // Compute and print price per ounce
28
29 double pricePerOunce = packPrice / packVolume;
30
31 System.out.printf("Price per ounce: %8.2f", pricePerOunce);
32 System.out.println();
33 }
34 }

```

### Program Run

```

Please enter the price for a six-pack: 2.95
Please enter the volume for each can (in ounces): 12
Price per ounce:      0.04

```

### SELF CHECK



15. Write statements to prompt for and read the user's age using a Scanner variable named in.
16. What is wrong with the following statement sequence?  

```
System.out.print("Please enter the unit price: ");
double unitPrice = in.nextDouble();
int quantity = in.nextInt();
```
17. What is problematic about the following statement sequence?  

```
System.out.print("Please enter the unit price: ");
double unitPrice = in.nextInt();
```
18. What is problematic about the following statement sequence?  

```
System.out.print("Please enter the number of cans");
int cans = in.nextInt();
```
19. What is the output of the following statement sequence?  

```
int volume = 10;
System.out.printf("The volume is %5d", volume);
```
20. Using the printf method, print the values of the integer variables bottles and cans so that the output looks like this:  

```
Bottles:      8
Cans:         24
```

The numbers to the right should line up. (You may assume that the numbers have at most 8 digits.)

**Practice It** Now you can try these exercises at the end of the chapter: R2.10, P2.6, P2.7.

### Programming Tip 2.4



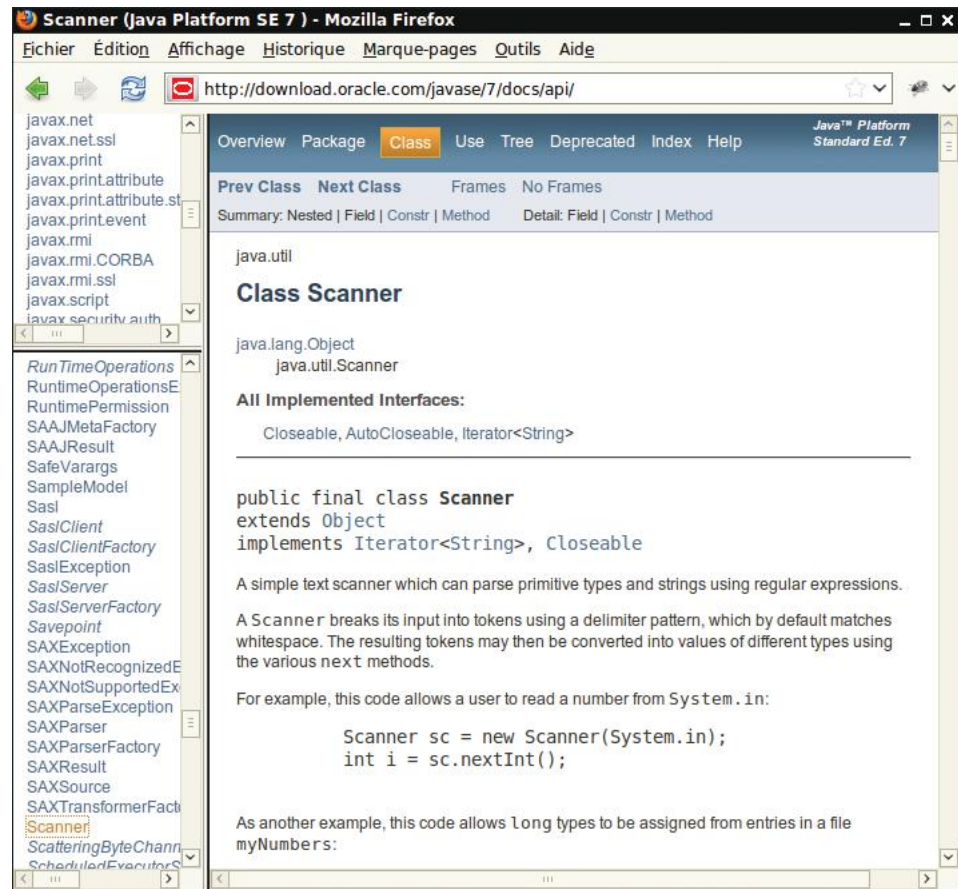
### Use the API Documentation

The classes and methods of the Java library are listed in the **API documentation**. The API is the “**application programming interface**”. A programmer who uses the Java classes to put together a computer program (or *application*) is an *application programmer*. That’s you. In contrast, the programmers who designed and implemented the library classes (such as `Scanner`) are *system programmers*.

The API (Application Programming Interface) documentation lists the classes and methods of the Java library.

You can find the API documentation at <http://download.oracle.com/javase/7/docs/api/>. The API documentation describes all classes in the Java library — there are thousands of them. Fortunately, only a few are of interest to the beginning programmer. To learn more about a class, click on its name in the left hand column. You can then find out the package to which the class belongs, and which methods it supports (see Figure 4). Click on the link of a method to get a detailed description.

Appendix D contains an abbreviated version of the API documentation.



**Figure 4** The API Documentation of the Standard Java Library

## HOW TO 2.1

## Carrying out Computations



Many programming problems require arithmetic computations. This How To shows you how to turn a problem statement into pseudocode and, ultimately, a Java program.

For example, suppose you are asked to write a program that simulates a vending machine. A customer selects an item for purchase and inserts a bill into the vending machine. The vending machine dispenses the purchased item and gives change. We will assume that all item prices are multiples of 25 cents, and the machine gives all change in dollar coins and quarters.

Your task is to compute how many coins of each type to return.

### Step 1 Understand the problem: What are the inputs? What are the desired outputs?

In this problem, there are two inputs:

- The denomination of the bill that the customer inserts
- The price of the purchased item

There are two desired outputs:

- The number of dollar coins that the machine returns
- The number of quarters that the machine returns

### Step 2 Work out examples by hand.

This is a very important step. If you can't compute a couple of solutions by hand, it's unlikely that you'll be able to write a program that automates the computation.

Let's assume that a customer purchased an item that cost \$2.25 and inserted a \$5 bill. The customer is due \$2.75, or two dollar coins and three quarters, in change.

That is easy for you to see, but how can a Java program come to the same conclusion? The key is to work in pennies, not dollars. The change due the customer is 275 pennies. Dividing by 100 yields 2, the number of dollars. Dividing the remainder (75) by 25 yields 3, the number of quarters.

### Step 3 Write pseudocode for computing the answers.

In the previous step, you worked out a specific instance of the problem. You now need to come up with a method that works in general.

Given an arbitrary item price and payment, how can you compute the coins due? First, compute the change due in pennies:

**change due = 100 x bill value - item price in pennies**

To get the dollars, divide by 100 and discard the remainder:

**dollar coins = change due / 100 (without remainder)**

The remaining change due can be computed in two ways. If you are familiar with the modulus operator, you can simply compute

**change due = change due % 100**

Alternatively, subtract the penny value of the dollar coins from the change due:

**change due = change due - 100 x dollar coins**

To get the quarters due, divide by 25:

**quarters = change due / 25**



**Step 4** Declare the variables and constants that you need, and specify their types.

Here, we have five variables:

- billValue
- itemPrice
- changeDue
- dollarCoins
- quarters

Should we introduce constants to explain 100 and 25 as `PENNIES_PER_DOLLAR` and `PENNIES_PER_QUARTER`? Doing so will make it easier to convert the program to international markets, so we will take this step.

It is very important that `changeDue` and `PENNIES_PER_DOLLAR` are of type `int` because the computation of `dollarCoins` uses integer division. Similarly, the other variables are integers.

**Step 5** Turn the pseudocode into Java statements.

If you did a thorough job with the pseudocode, this step should be easy. Of course, you have to know how to express mathematical operations (such as powers or integer division) in Java.

```
changeDue = PENNIES_PER_DOLLAR * billValue - itemPrice;
dollarCoins = changeDue / PENNIES_PER_DOLLAR;
changeDue = changeDue % PENNIES_PER_DOLLAR;
quarters = changeDue / PENNIES_PER_QUARTER;
```

**Step 6** Provide input and output.

Before starting the computation, we prompt the user for the bill value and item price:

```
System.out.print("Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): ");
billValue = in.nextInt();
System.out.print("Enter item price in pennies: ");
itemPrice = in.nextInt();
```

When the computation is finished, we display the result. For extra credit, we use the `printf` method to make sure that the output lines up neatly.

```
System.out.printf("Dollar coins: %6d", dollarCoins);
System.out.printf("Quarters:      %6d", quarters);
```

**Step 7** Provide a class with a main method.

Your computation needs to be placed into a class. Find an appropriate name for the class that describes the purpose of the computation. In our example, we will choose the name `VendingMachine`.

Inside the class, supply a main method.



*A vending machine takes bills and gives change in coins.*

In the main method, you need to declare constants and variables (Step 4), carry out computations (Step 5), and provide input and output (Step 6). Clearly, you will want to first get the input, then do the computations, and finally show the output. Declare the constants at the beginning of the method, and declare each variable just before it is needed.

Here is the complete program, `how_to_1/VendingMachine.java`:

```
import java.util.Scanner;

/**
 * This program simulates a vending machine that gives change.
 */
public class VendingMachine
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);

        final int PENNIES_PER_DOLLAR = 100;
        final int PENNIES_PER_QUARTER = 25;

        System.out.print("Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): ");
        int billValue = in.nextInt();
        System.out.print("Enter item price in pennies: ");
        int itemPrice = in.nextInt();

        // Compute change due

        int changeDue = PENNIES_PER_DOLLAR * billValue - itemPrice;
        int dollarCoins = changeDue / PENNIES_PER_DOLLAR;
        changeDue = changeDue % PENNIES_PER_DOLLAR;
        int quarters = changeDue / PENNIES_PER_QUARTER;

        // Print change due

        System.out.printf("Dollar coins: %d", dollarCoins);
        System.out.println();
        System.out.printf("Quarters:      %d", quarters);
        System.out.println();
    }
}
```

### Program Run

```
Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): 5
Enter item price in pennies: 25
Dollar coins:      2
Quarters:         3
```

### WORKED EXAMPLE 2.1

### Computing the Cost of Stamps



This Worked Example uses arithmetic functions to simulate a stamp vending machine.

Available online in WileyPLUS and at [www.wiley.com/college/horstmann](http://www.wiley.com/college/horstmann).

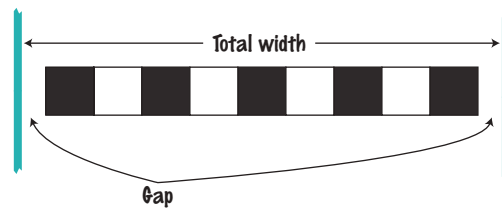
## 2.4 Problem Solving: First Do It By Hand

A very important step for developing an algorithm is to first carry out the computations *by hand*. If you can't compute a solution yourself, it's unlikely that you'll be able to write a program that automates the computation.

To illustrate the use of hand calculations, consider the following problem.

A row of black and white tiles needs to be placed along a wall. For aesthetic reasons, the architect has specified that the first and last tile shall be black.

Your task is to compute the number of tiles needed and the gap at each end, given the space available and the width of each tile.



Pick concrete values for a typical situation to use in a hand calculation.

To make the problem more concrete, let's assume the following dimensions:

- Total width: 100 inches
- Tile width: 5 inches

The obvious solution would be to fill the space with 20 tiles, but that would not work—the last tile would be white.

Instead, look at the problem this way: The first tile must always be black, and then we add some number of white/black pairs:



The first tile takes up 5 inches, leaving 95 inches to be covered by pairs. Each pair is 10 inches wide. Therefore the number of pairs is  $95 / 10 = 9.5$ . However, we need to discard the fractional part since we can't have fractions of tile pairs.

Therefore, we will use 9 tile pairs or 18 tiles, plus the initial black tile. Altogether, we require 19 tiles.

The tiles span  $19 \times 5 = 95$  inches, leaving a total gap of  $100 - 19 \times 5 = 5$  inches.

The gap should be evenly distributed at both ends. At each end, the gap is  $(100 - 19 \times 5) / 2 = 2.5$  inches.

This computation gives us enough information to devise an algorithm with arbitrary values for the total width and tile width.

**number of pairs** = integer part of  $(\text{total width} - \text{tile width}) / (2 \times \text{tile width})$

**number of tiles** =  $1 + 2 \times \text{number of pairs}$

**gap at each end** =  $(\text{total width} - \text{number of tiles} \times \text{tile width}) / 2$

As you can see, doing a hand calculation gives enough insight into the problem that it becomes easy to develop an algorithm.

### ONLINE EXAMPLE

- ⊕ A program that implements this algorithm.



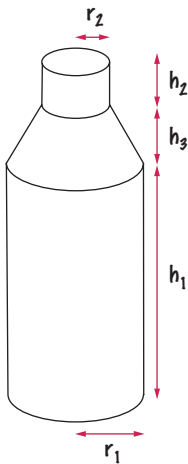
21. Translate the pseudocode for computing the number of tiles and the gap width into Java.
22. Suppose the architect specifies a pattern with black, gray, and white tiles, like this:



Again, the first and last tile should be black. How do you need to modify the algorithm?

23. A robot needs to tile a floor with alternating black and white tiles. Develop an algorithm that yields the color (0 for black, 1 for white), given the row and column number. Start with specific values for the row and column, and then generalize.

	1	2	3	4
1	Black	White	Black	White
2	White	Black	White	Black
3	Black	White	Black	White
4	White	Black	White	Black



24. For a particular car, repair and maintenance costs in year 1 are estimated at \$100; in year 10, at \$1,500. Assuming that the repair cost increases by the same amount every year, develop pseudocode to compute the repair cost in year 3 and then generalize to year  $n$ .
25. The shape of a bottle is approximated by two cylinders of radius  $r_1$  and  $r_2$  and heights  $h_1$  and  $h_2$ , joined by a cone section of height  $h_3$ . Using the formulas for the volume of a cylinder,  $V = \pi r^2 h$ , and a cone section,

$$V = \pi \frac{(r_1^2 + r_1 r_2 + r_2^2) h_3}{3},$$

develop pseudocode to compute the volume of the bottle. Using an actual bottle with known volume as a sample, make a hand calculation of your pseudocode.

**Practice It** Now you can try these exercises at the end of the chapter: R2.15, R2.17, R2.18.

## WORKED EXAMPLE 2.2

### Computing Travel Time



In this Worked Example, we develop a hand calculation to compute the time that a robot requires to retrieve an item from rocky terrain.



⊕ Available online in WileyPLUS and at [www.wiley.com/college/horstmann](http://www.wiley.com/college/horstmann).

## 2.5 Strings

Strings are sequences of characters.

Many programs process text, not numbers. Text consists of **characters**: letters, numbers, punctuation, spaces, and so on. A **string** is a sequence of characters. For example, the string "Harry" is a sequence of five characters.



### 2.5.1 The String Type

You can declare variables that hold strings.

```
String name = "Harry";
```

We distinguish between string variables (such as the variable name declared above) and string **literals** (character sequences enclosed in quotes, such as "Harry"). A string variable is simply a variable that can hold a string, just as an integer variable can hold an integer. A string literal denotes a particular string, just as a number literal (such as 2) denotes a particular number.

The number of characters in a string is called the *length* of the string. For example, the length of "Harry" is 5. You can compute the length of a string with the `length` method.

```
int n = name.length();
```

A string of length 0 is called the *empty string*. It contains no characters and is written as "".

The `length` method yields the number of characters in a string.

### 2.5.2 Concatenation

Given two strings, such as "Harry" and "Morgan", you can **concatenate** them to one long string. The result consists of all characters in the first string, followed by all characters in the second string. In Java, you use the `+` operator to concatenate two strings.

For example,

```
String fName = "Harry";
String lName = "Morgan";
String name = fName + lName;
```

results in the string

```
"HarryMorgan"
```

What if you'd like the first and last name separated by a space? No problem:

```
String name = fName + " " + lName;
```

This statement concatenates three strings: `fName`, the string literal " ", and `lName`. The result is

```
"Harry Morgan"
```

When the expression to the left or the right of a `+` operator is a string, the other one is automatically forced to become a string as well, and both strings are concatenated.

Use the `+` operator to *concatenate* strings; that is, to put them together to yield a longer string.

For example, consider this code:

```
String jobTitle = "Agent";
int employeeId = 7;
String bond = jobTitle + employeeId;
```

Whenever one of the arguments of the + operator is a string, the other argument is converted to a string.

Because `jobTitle` is a string, `employeeId` is converted from the integer 7 to the string "7". Then the two strings "Agent" and "7" are concatenated to form the string "Agent7".

This concatenation is very useful for reducing the number of `System.out.print` instructions. For example, you can combine

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

to the single call

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

The concatenation `"The total is " + total` computes a single string that consists of the string "The total is ", followed by the string equivalent of the number `total`.

### 2.5.3 String Input

Use the `next` method of the `Scanner` class to read a string containing a single word.

You can read a string from the console:

```
System.out.print("Please enter your name: ");
String name = in.next();
```

When a string is read with the `next` method, only one word is read. For example, suppose the user types

```
Harry Morgan
```

as the response to the prompt. This input consists of two words. The call `in.next()` yields the string "Harry". You can use another call to `in.next()` to read the second word.

### 2.5.4 Escape Sequences

To include a quotation mark in a literal string, precede it with a backslash (`\`), like this:

```
"He said \"Hello\""
```

The backslash is not included in the string. It indicates that the quotation mark that follows should be a part of the string and not mark the end of the string. The sequence `\` is called an **escape sequence**.

To include a backslash in a string, use the escape sequence `\\`, like this:

```
"C:\\Temp\\Secret.txt"
```

Another common escape sequence is `\n`, which denotes a **newline** character. Printing a newline character causes the start of a new line on the display. For example, the statement

```
System.out.print("*\n**\n***\n");
```

prints the characters

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

on three separate lines.



You often want to add a newline character to the end of the format string when you use `System.out.printf`:

```
System.out.printf("Price: %10.2f\n", price);
```

## 2.5.5 Strings and Characters

Strings are sequences of Unicode characters (see Random Fact 2.2). In Java, a **character** is a value of the type `char`. Characters have numeric values. You can find the values of the characters that are used in Western European languages in Appendix A. For example, if you look up the value for the character `'H'`, you can see that is actually encoded as the number 72.



*A string is a sequence of characters.*

Character literals are delimited by single quotes, and you should not confuse them with strings.

- `'H'` is a character, a value of type `char`.
- `"H"` is a string containing a single character, a value of type `String`.

String positions are counted starting with 0.

The `charAt` method returns a `char` value from a string. The first string position is labeled 0, the second one 1, and so on.

H	a	r	r	y
0	1	2	3	4

The position number of the last character (4 for the string `"Harry"`) is always one less than the length of the string.

For example, the statement

```
String name = "Harry";
char start = name.charAt(0);
char last = name.charAt(4);
```

sets `start` to the value `'H'` and `last` to the value `'y'`.

## 2.5.6 Substrings

Use the `substring` method to extract a part of a string.

Once you have a string, you can extract substrings by using the `substring` method. The method call

```
str.substring(start, pastEnd)
```

returns a string that is made up of the characters in the string `str`, starting at position `start`, and containing all characters up to, but not including, the position `pastEnd`. Here is an example:

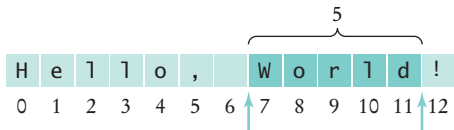
```
String greeting = "Hello, World!";
String sub = greeting.substring(0, 5); // sub is "Hello"
```

The `substring` operation makes a string that consists of the first five characters taken from the string `greeting`.

H	e	l	l	o	,		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11	12

Let's figure out how to extract the substring "World". Count characters starting at 0, not 1. You find that **W** has position number 7. The first character that you don't want, **!**, is the character at position 12. Therefore, the appropriate substring command is

```
String sub2 = greeting.substring(7, 12);
```



It is curious that you must specify the position of the first character that you do want and then the first character that you don't want. There is one advantage to this setup. You can easily compute the length of the substring: It is `pastEnd - start`. For example, the string "World" has length `12 - 7 = 5`.

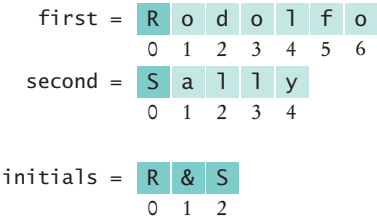
If you omit the end position when calling the `substring` method, then all characters from the starting position to the end of the string are copied. For example,

```
String tail = greeting.substring(7); // Copies all characters from position 7 on
```

sets `tail` to the string "World!".

Following is a simple program that puts these concepts to work. The program asks for your name and that of your significant other. It then prints out your initials.

The operation `first.substring(0, 1)` makes a string consisting of one character, taken from the start of `first`. The program does the same for the `second`. Then it concatenates the resulting one-character strings with the string literal `"&"` to get a string of length 3, the `initials` string. (See Figure 5.)



**Figure 5** Building the `initials` String

*Initials are formed from the first letter of each name.*

**section\_5/Initials.java**

```
1 import java.util.Scanner;
2
3 /**
4  * This program prints a pair of initials.
5  */
6 public class Initials
7 {
8     public static void main(String[] args)
9     {
10         Scanner in = new Scanner(System.in);
11     }
```

Copyright © 2012, Wiley. All rights reserved.

```

12 // Get the names of the couple
13
14 System.out.print("Enter your first name: ");
15 String first = in.next();
16 System.out.print("Enter your significant other's first name: ");
17 String second = in.next();
18
19 // Compute and display the inscription
20
21 String initials = first.substring(0, 1)
22     + "&" + second.substring(0, 1);
23 System.out.println(initials);
24 }
25 }

```

### Program Run

```

Enter your first name: Rodolfo
Enter your significant other's first name: Sally
R&S

```

**Table 9** String Operations

Statement	Result	Comment
string str = "Ja"; str = str + "va";	str is set to "Java"	When applied to strings, + denotes concatenation.
System.out.println("Please" + " enter your name: ");	Prints Please enter your name:	Use concatenation to break up strings that don't fit into one line.
team = 49 + "ers"	team is set to "49ers"	Because "ers" is a string, 49 is converted to a string.
String first = in.next(); String last = in.next(); (User input: Harry Morgan)	first contains "Harry" last contains "Morgan"	The next method places the next word into the string variable.
String greeting = "H & S"; int n = greeting.length();	n is set to 5	Each space counts as one character.
String str = "Sally"; char ch = str.charAt(1);	ch is set to 'a'	This is a char value, not a String. Note that the initial position is 0.
String str = "Sally"; String str2 = str.substring(1, 4);	str2 is set to "all"	Extracts the substring starting at position 1 and ending before position 4.
String str = "Sally"; String str2 = str.substring(1);	str2 is set to "ally"	If you omit the end position, all characters from the position until the end of the string are included.
String str = "Sally"; String str2 = str.substring(1, 2);	str2 is set to "a"	Extracts a String of length 1; contrast with str.charAt(1).
String last = str.substring( str.length() - 1);	last is set to the string containing the last character in str	The last character has position str.length() - 1.



26. What is the length of the string "Java Program"?
27. Consider this string variable.  

```
String str = "Java Program";
```

 Give a call to the substring method that returns the substring "gram".
28. Use string concatenation to turn the string variable str from Self Check 27 into "Java Programming".
29. What does the following statement sequence print?  

```
String str = "Harry";
int n = str.length();
String mystery = str.substring(0, 1) + str.substring(n - 1, n);
System.out.println(mystery);
```
30. Give an input statement to read a name of the form "John Q. Public".

**Practice It** Now you can try these exercises at the end of the chapter: R2.7, R2.11, P2.15, P2.23.

### Special Topic 2.4



### Instance Methods and Static Methods

In this chapter, you have learned how to read, process, and print numbers and strings. Many of these tasks involve various method calls. You may have noticed syntactical differences in these method calls. For example, to compute the square root of a number num, you call `Math.sqrt(num)`, but to compute the length of a string str, you call `str.length()`. This section explains the reasons behind these differences.

The Java language distinguishes between values of **primitive types** and **objects**. Numbers and characters, as well as the values `false` and `true` that you will see in Chapter 3, are primitive. All other values are objects. Examples of objects are

- a string such as "Hello".
- a Scanner object obtained by calling `in = new Scanner(System.in)`.
- `System.in` and `System.out`.

In Java, each object belongs to a **class**. For example,

- All strings are objects of the String class.
- A scanner object belongs to the Scanner class.
- `System.out` is an object of the `PrintStream` class. (It is useful to know this so that you can look up the valid methods in the API documentation; see Programming Tip 2.4 on page 53.)

A class declares the methods that you can use with its objects. Here are examples of methods that are invoked on objects:

```
"Hello".substring(0, 1)
in.nextDouble()
System.out.println("Hello")
```

A method is invoked with the **dot notation**: the object is followed by the name of the method, and the method is followed by parameters enclosed in parentheses.

The method is invoked on this object.      This is the name of the method.      These parameters are inputs to the method.

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

You cannot invoke methods on numbers. For example, the call `2.sqrt()` would be an error.

In Java, classes can declare methods that are *not* invoked on objects. Such methods are called **static methods**. (The term “static” is a historical holdover from the C and C++ programming languages. It has nothing to do with the usual meaning of the word.) For example, the `Math` class declares a static method `sqrt`. You call it by giving the name of the class and method, then the name of the numeric input: `Math.sqrt(2)`.

The name of the class
The name of the static method  
Math.sqrt(2)

In contrast, a method that is invoked on an object is called an **instance method**. As a rule of thumb, you use static methods when you manipulate numbers. You use instance methods when you process strings or perform input/output. You will learn more about the distinction between static and instance methods in Chapter 8.

### Special Topic 2.5



### Using Dialog Boxes for Input and Output

Most program users find the console window rather old-fashioned. The easiest alternative is to create a separate pop-up window for each input.



*An Input Dialog Box*

Call the static `showInputDialog` method of the `JOptionPane` class, and supply the string that prompts the input from the user. For example,

```
String input = JOptionPane.showInputDialog("Enter price:");
```

That method returns a `String` object. Of course, often you need the input as a number. Use the `Integer.parseInt` and `Double.parseDouble` methods to convert the string to a number:

```
double price = Double.parseDouble(input);
```

You can also display output in a dialog box:

```
JOptionPane.showMessageDialog(null, "Price: " + price);
```

#### ONLINE EXAMPLE

- + A complete program that uses option panes for input and output.

### VIDEO EXAMPLE 2.2



### Computing Distances on Earth

In this Video Example, you will see how to write a program that computes the distance between any two points on Earth.



+ Available online in WileyPLUS and at [www.wiley.com/college/horstmann](http://www.wiley.com/college/horstmann).



## Random Fact 2.2 International Alphabets and Unicode

The English alphabet is pretty simple: upper- and lowercase *a* to *z*. Other European languages have accent marks and special characters. For example, German has three so-called *umlaut* characters, ä, ö, ü, and a *double-s* character ß. These are not optional frills; you couldn't write a page of German text without using these characters a few times. German keyboards have keys for these characters.



The German Keyboard Layout

Many countries don't use the Roman script at all. Russian, Greek, Hebrew,

Arabic, and Thai letters, to name just a few, have completely different shapes. To complicate matters, Hebrew and Arabic are typed from right to left. Each of these alphabets has about as many characters as the English alphabet.



Hebrew, Arabic, and English

The Chinese languages as well as Japanese and Korean use Chinese characters. Each character represents an idea or thing. Words are made up of one or more of these ideographic characters. Over 70,000 ideographs are known.

Starting in 1988, a consortium of hardware and software manufacturers developed a uniform encoding scheme

called **Unicode** that is capable of encoding text in essentially all written languages of the world. An early version of Unicode used 16 bits for each character. The Java `char` type corresponds to that encoding.

Today Unicode has grown to a 21-bit code, with definitions for over 100,000 characters. There are even plans to add codes for extinct languages, such as Egyptian hieroglyphics. Unfortunately, that means that a Java `char` value does not always correspond to a Unicode character. Some characters in languages such as Chinese or ancient Egyptian occupy two `char` values.



The Chinese Script

## CHAPTER SUMMARY

### Declare variables with appropriate names and types.

- A variable is a storage location with a name.
- When declaring a variable, you usually specify an initial value.
- When declaring a variable, you also specify the type of its values.
- Use the `int` type for numbers that cannot have a fractional part.
- Use the `double` type for floating-point numbers.
- By convention, variable names should start with a lowercase letter.
- An assignment statement stores a new value in a variable, replacing the previously stored value.
- The assignment operator `=` does *not* denote mathematical equality.





- You cannot change the value of a variable that is defined as `final`.
- Use comments to add explanations for humans who read your code. The compiler ignores comments.



### Write arithmetic expressions in Java.

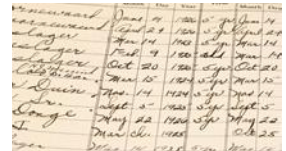


- Mixing integers and floating-point values in an arithmetic expression yields a floating-point value.
- The `++` operator adds 1 to a variable; the `--` operator subtracts 1.
- If both arguments of `/` are integers, the remainder is discarded.
- The `%` operator computes the remainder of an integer division.
- The Java library declares many mathematical functions, such as `Math.sqrt` (square root) and `Math.pow` (raising to a power).
- You use a cast (*typeName*) to convert a value to a different type.

### Write programs that read user input and print formatted output.



- Java classes are grouped into packages. Use the `import` statement to use classes from packages.
- Use the `Scanner` class to read keyboard input in a console window.
- Use the `printf` method to specify how values should be formatted.
- The API (Application Programming Interface) documentation lists the classes and methods of the Java library.



### Carry out hand calculations when developing an algorithm.

- Pick concrete values for a typical situation to use in a hand calculation.

### Write programs that process strings.



- Strings are sequences of characters.
- The `length` method yields the number of characters in a string.
- Use the `+` operator to *concatenate* strings; that is, to put them together to yield a longer string.
- Whenever one of the arguments of the `+` operator is a string, the other argument is converted to a string.
- Use the `next` method of the `Scanner` class to read a string containing a single word.
- String positions are counted starting with 0.
- Use the `substring` method to extract a part of a string.





## STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

<code>java.io.PrintStream</code>	<code>max</code>	<code>java.math.BigDecimal</code>
<code>printf</code>	<code>min</code>	<code>add</code>
<code>java.lang.Double</code>	<code>pow</code>	<code>multiply</code>
<code>parseDouble</code>	<code>round</code>	<code>subtract</code>
<code>java.lang.Integer</code>	<code>sin</code>	<code>java.math.BigInteger</code>
<code>MAX_VALUE</code>	<code>sqrt</code>	<code>add</code>
<code>MIN_VALUE</code>	<code>tan</code>	<code>multiply</code>
<code>parseInt</code>	<code>toDegrees</code>	<code>subtract</code>
<code>java.lang.Math</code>	<code>toRadians</code>	<code>java.util.Scanner</code>
<code>PI</code>	<code>java.lang.String</code>	<code>next</code>
<code>abs</code>	<code>charAt</code>	<code>nextDouble</code>
<code>cos</code>	<code>length</code>	<code>nextInt</code>
<code>exp</code>	<code>substring</code>	<code>javax.swing.JOptionPane</code>
<code>log</code>	<code>java.lang.System</code>	<code>showInputDialog</code>
<code>log10</code>	<code>in</code>	<code>showMessageDialog</code>

## REVIEW EXERCISES

- **R2.1** What is the value of `mystery` after this sequence of statements?

```
int mystery = 1;
mystery = 1 - 2 * mystery;
mystery = mystery + 1;
```

- **R2.2** What is wrong with the following sequence of statements?

```
int mystery = 1;
mystery = mystery + 1;
int mystery = 1 - 2 * mystery;
```

- **R2.3** Write the following mathematical expressions in Java.

$$s = s_0 + v_0 t + \frac{1}{2} g t^2$$

$$G = 4\pi^2 \frac{a^3}{p^2(m_1 + m_2)}$$

$$FV = PV \cdot \left(1 + \frac{INT}{100}\right)^{YRS}$$

$$c = \sqrt{a^2 + b^2 - 2ab \cos \gamma}$$

- **R2.4** Write the following Java expressions in mathematical notation.

- `dm = m * (Math.sqrt(1 + v / c) / Math.sqrt(1 - v / c) - 1);`
- `volume = Math.PI * r * r * h;`
- `volume = 4 * Math.PI * Math.pow(r, 3) / 3;`
- `z = Math.sqrt(x * x + y * y);`

- **R2.5** What are the values of the following expressions? In each line, assume that

```
double x = 2.5;
double y = -1.5;
```

```
int m = 18;
int n = 4;

a. x + n * y - (x + n) * y
b. m / n + m % n
c. 5 * x - n / 5
d. 1 - (1 - (1 - (1 - (1 - n))))
e. Math.sqrt(Math.sqrt(n))
```

- **R2.6** What are the values of the following expressions, assuming that *n* is 17 and *m* is 18?

```
a. n / 10 + n % 10
b. n % 2 + m % 2
c. (m + n) / 2
d. (m + n) / 2.0
e. (int) (0.5 * (m + n))
f. (int) Math.round(0.5 * (m + n))
```

- ■ **R2.7** What are the values of the following expressions? In each line, assume that

```
String s = "Hello";
String t = "World";

a. s.length() + t.length()
b. s.substring(1, 2)
c. s.substring(s.length() / 2, s.length())
d. s + t
e. t + s
```

- **R2.8** Find at least five *compile-time* errors in the following program.

```
public class HasErrors
{
    public static void main();
    {
        System.out.print(Please enter two numbers:)
        x = in.readDouble;
        y = in.readDouble;
        System.out.println("The sum is " + x + y);
    }
}
```

- ■ **R2.9** Find three *run-time* errors in the following program.

```
public class HasErrors
{
    public static void main(String[] args)
    {
        int x = 0;
        int y = 0;
        Scanner in = new Scanner("System.in");
        System.out.print("Please enter an integer:");
        x = in.readInt();
        System.out.print("Please enter another integer: ");
        x = in.readInt();
        System.out.println("The sum is " + x + y);
    }
}
```

- **R2.10** Consider the following code segment.

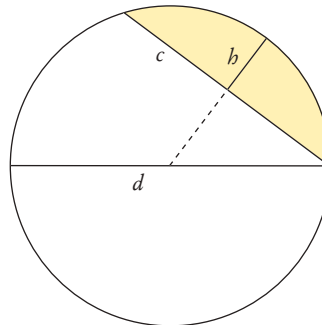
```
double purchase = 19.93;
double payment = 20.00;
double change = payment - purchase;
System.out.println(change);
```

The code segment prints the change as 0.070000000000000028. Explain why. Give a recommendation to improve the code so that users will not be confused.

- **R2.11** Explain the differences between 2, 2.0, '2', "2", and "2.0".
- **R2.12** Explain what each of the following program segments computes.

```
a. x = 2;
   y = x + x;
b. s = "2";
   t = s + s;
```

- **R2.13** Write pseudocode for a program that reads a word and then prints the first character, the last character, and the characters in the middle. For example, if the input is Harry, the program prints H y arr.
- **R2.14** Write pseudocode for a program that reads a name (such as Harold James Morgan) and then prints a monogram consisting of the initial letters of the first, middle, and last name (such as HJM).
- **R2.15** Write pseudocode for a program that computes the first and last digit of a number. For example, if the input is 23456, the program should print 2 and 6. *Hint:* %, Math.log10.
- **R2.16** Modify the pseudocode for the program in How To 2.1 so that the program gives change in quarters, dimes, and nickels. You can assume that the price is a multiple of 5 cents. To develop your pseudocode, first work with a couple of specific values.
- **R2.17** A cocktail shaker is composed of three cone sections. Using realistic values for the radii and heights, compute the total volume, using the formula given in Self Check 25 for a cone section. Then develop an algorithm that works for arbitrary dimensions.
- **R2.18** You are cutting off a piece of pie like this, where  $c$  is the length of the straight part (called the chord length) and  $h$  is the height of the piece.



There is an approximate formula for the area:  $A \approx \frac{2}{3}ch + \frac{h^3}{2c}$

However,  $h$  is not so easy to measure, whereas the diameter  $d$  of a pie is usually well-known. Calculate the area where the diameter of the pie is 12 inches and the chord length of the segment is 10 inches. Generalize to an algorithm that yields the area for any diameter and chord length.

- ■ **R2.19** The following pseudocode describes how to obtain the name of a day, given the day number (0 = Sunday, 1 = Monday, and so on.)

*Declare a string called names containing "SunMonTueWedThuFriSat".*

*Compute the starting position as 3 x the day number.*

*Extract the substring of names at the starting position with length 3.*

Check this pseudocode, using the day number 4. Draw a diagram of the string that is being computed, similar to Figure 5.

- ■ ■ **R2.20** The following pseudocode describes how to swap two letters in a word.

*We are given a string str and two positions i and j. (i comes before j)*

*Set first to the substring from the start of the string to the last position before i.*

*Set middle to the substring from positions i + 1 to j - 1.*

*Set last to the substring from position j + 1 to the end of the string.*

*Concatenate the following five strings: first, the string containing just the character at position j, middle, the string containing just the character at position i, and last.*

Check this pseudocode, using the string "Gateway" and positions 2 and 4. Draw a diagram of the string that is being computed, similar to Figure 5.

- ■ **R2.21** How do you get the first character of a string? The last character? How do you remove the first character? The last character?

- ■ ■ **R2.22** Write a program that prints the values

$3 * 1000 * 1000 * 1000$

$3.0 * 1000 * 1000 * 1000$

Explain the results.

- **R2.23** This chapter contains a number of recommendations regarding variables and constants that make programs easier to read and maintain. Briefly summarize these recommendations.

## PROGRAMMING EXERCISES

- **P2.1** Write a program that displays the dimensions of a letter-size ( $8.5 \times 11$  inches) sheet of paper in millimeters. There are 25.4 millimeters per inch. Use constants and comments in your program.
- **P2.2** Write a program that computes and displays the perimeter of a letter-size ( $8.5 \times 11$  inches) sheet of paper and the length of its diagonal.
- **P2.3** Write a program that reads a number and displays the square, cube, and fourth power. Use the Math.pow method only for the fourth power.
- ■ **P2.4** Write a program that prompts the user for two integers and then prints
  - The sum
  - The difference

- The product
- The average
- The distance (absolute value of the difference)
- The maximum (the larger of the two)
- The minimum (the smaller of the two)

*Hint:* The `max` and `min` functions are declared in the `Math` class.

- ■ **P2.5** Enhance the output of Exercise P2.4 so that the numbers are properly aligned:

```
Sum:           45
Difference:    -5
Product:       500
Average:      22.50
Distance:      5
Maximum:      25
Minimum:      20
```

- ■ **P2.6** Write a program that prompts the user for a measurement in meters and then converts it to miles, feet, and inches.

- **P2.7** Write a program that prompts the user for a radius and then prints

- The area and circumference of a circle with that radius
- The volume and surface area of a sphere with that radius

- ■ **P2.8** Write a program that asks the user for the lengths of the sides of a rectangle. Then print

- The area and perimeter of the rectangle
- The length of the diagonal (use the Pythagorean theorem)

- **P2.9** Improve the program discussed in How To 2.1 to allow input of quarters in addition to bills.

- ■ ■ **P2.10** Write a program that helps a person decide whether to buy a hybrid car. Your program's inputs should be:

- The cost of a new car
- The estimated miles driven per year
- The estimated gas price
- The efficiency in miles per gallon
- The estimated resale value after 5 years

Compute the total cost of owning the car for five years. (For simplicity, we will not take the cost of financing into account.) Obtain realistic prices for a new and used hybrid and a comparable car from the Web. Run your program twice, using today's gas price and 15,000 miles per year. Include pseudocode and the program runs with your assignment.



- ■ **P2.11** Write a program that asks the user to input
- The number of gallons of gas in the tank
- The fuel efficiency in miles per gallon
- The price of gas per gallon

Then print the cost per 100 miles and how far the car can go with the gas in the tank.

- **P2.12** *File names and extensions.* Write a program that prompts the user for the drive letter (C), the path (\Windows\System), the file name (Readme), and the extension (txt). Then print the complete file name C:\Windows\System\Readme.txt. (If you use UNIX or a Macintosh, skip the drive name and use / instead of \ to separate directories.)

- **P2.13** Write a program that reads a number between 1,000 and 999,999 from the user, where the user enters a comma in the input. Then print the number without a comma. Here is a sample dialog; the user input is in color:

Please enter an integer between 1,000 and 999,999: 23,456  
23456

*Hint:* Read the input as a string. Measure the length of the string. Suppose it contains  $n$  characters. Then extract substrings consisting of the first  $n - 4$  characters and the last three characters.

- **P2.14** Write a program that reads a number between 1,000 and 999,999 from the user and prints it with a comma separating the thousands. Here is a sample dialog; the user input is in color:

Please enter an integer between 1000 and 999999: 23456  
23,456

- **P2.15** *Printing a grid.* Write a program that prints the following grid to play tic-tac-toe.

```
+---+---+
|   |   |
+---+---+
|   |   |
+---+---+
|   |   |
+---+---+
```

Of course, you could simply write seven statements of the form

```
System.out.println("+---+---+");
```

You should do it the smart way, though. Declare string variables to hold two kinds of patterns: a comb-shaped pattern and the bottom line. Print the comb three times and the bottom line once.

- **P2.16** Write a program that reads in an integer and breaks it into a sequence of individual digits. For example, the input 16384 is displayed as

1 6 3 8 4

You may assume that the input has no more than five digits and is not negative.

- **P2.17** Write a program that reads two times in military format (0900, 1730) and prints the number of hours and minutes between the two times. Here is a sample run. User input is in color.

Please enter the first time: 0900  
Please enter the second time: 1730  
8 hours 30 minutes

Extra credit if you can deal with the case where the first time is later than the second:

Please enter the first time: 1730  
Please enter the second time: 0900  
15 hours 30 minutes

- **P2.18** *Writing large letters.* A large letter H can be produced like this:

```

*           *
*           *
*****
*           *
*           *

```

It can be declared as a string literal like this:

```
final string LETTER_H = "* *\\n* *\\n*****\\n* *\\n* *\\n";
```

(The `\n` escape sequence denotes a “newline” character that causes subsequent characters to be printed on a new line.) Do the same for the letters E, L, and O. Then write the message

HELLO

in large letters.

- **P2.19** Write a program that transforms numbers 1, 2, 3, ..., 12 into the corresponding month names January, February, March, ..., December. *Hint:* Make a very long string "January February March ...", in which you add spaces such that each month name has *the same length*. Then use substring to extract the month you want.



- ■ **P2.20** Write a program that prints a Christmas tree:

Remember to use escape sequences.

- **P2.21** Easter Sunday is the first Sunday after the first full moon of spring. To compute the date, you can use this algorithm, invented by the mathematician Carl Friedrich Gauss in 1800:

1. Let  $y$  be the year (such as 1800 or 2001).
2. Divide  $y$  by 19 and call the remainder  $a$ . Ignore the quotient.
3. Divide  $y$  by 100 to get a quotient  $b$  and a remainder  $c$ .
4. Divide  $b$  by 4 to get a quotient  $d$  and a remainder  $e$ .
5. Divide  $8 * b + 13$  by 25 to get a quotient  $g$ . Ignore the remainder.
6. Divide  $19 * a + b - d - g + 15$  by 30 to get a remainder  $h$ . Ignore the quotient.
7. Divide  $c$  by 4 to get a quotient  $j$  and a remainder  $k$ .
8. Divide  $a + 11 * h$  by 319 to get a quotient  $m$ . Ignore the remainder.
9. Divide  $2 * e + 2 * j - k - h + m + 32$  by 7 to get a remainder  $r$ . Ignore the quotient.



**10.** Divide  $h - m + r + 90$  by 25 to get a quotient  $n$ . Ignore the remainder.

**11.** Divide  $h - m + r + n + 19$  by 32 to get a remainder  $p$ . Ignore the quotient.

Then Easter falls on day  $p$  of month  $n$ . For example, if  $y$  is 2001:

$a = 6$	$h = 18$	$n = 4$
$b = 20, c = 1$	$j = 0, k = 1$	$p = 15$
$d = 5, e = 0$	$m = 0$	
$g = 6$	$r = 6$	

Therefore, in 2001, Easter Sunday fell on April 15. Write a program that prompts the user for a year and prints out the month and day of Easter Sunday.

- **Business P2.22** The following pseudocode describes how a bookstore computes the price of an order from the total price and the number of the books that were ordered.

**Read the total book price and the number of books.**

**Compute the tax (7.5 percent of the total book price).**

**Compute the shipping charge (\$2 per book).**

**The price of the order is the sum of the total book price, the tax, and the shipping charge.**

**Print the price of the order.**

Translate this pseudocode into a Java program.

- **Business P2.23** The following pseudocode describes how to turn a string containing a ten-digit phone number (such as "4155551212") into a more readable string with parentheses and dashes, like this: "(415) 555-1212".

**Take the substring consisting of the first three characters and surround it with "(" and ")". This is the area code.**

**Concatenate the area code, the substring consisting of the next three characters, a hyphen, and the substring consisting of the last four characters. This is the formatted number.**

Translate this pseudocode into a Java program that reads a telephone number into a string variable, computes the formatted number, and prints it.

- **Business P2.24** The following pseudocode describes how to extract the dollars and cents from a price given as a floating-point value. For example, a price 2.95 yields values 2 and 95 for the dollars and cents.

**Assign the price to an integer variable dollars.**

**Multiply the difference price - dollars by 100 and add 0.5.**

**Assign the result to an integer variable cents.**

Translate this pseudocode into a Java program. Read a price and print the dollars and cents. Test your program with inputs 2.95 and 4.35.

- **Business P2.25** *Giving change.* Implement a program that directs a cashier how to give change. The program has two inputs: the amount due and the amount received from the customer. Display the dollars, quarters, dimes, nickels, and pennies that the customer should receive in return. In order to avoid roundoff errors, the program user should supply both amounts in pennies, for example 274 instead of 2.74.



- **Business P2.26** An online bank wants you to create a program that shows prospective customers how their deposits will grow. Your program should read the initial balance and the

annual interest rate. Interest is compounded monthly. Print out the balances after the first three months. Here is a sample run:

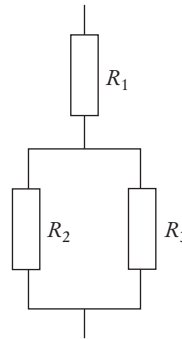
```
Initial balance: 1000
Annual interest rate in percent: 6.0
After first month: 1005.00
After second month: 1010.03
After third month: 1015.08
```

- **Business P2.27** A video club wants to reward its best members with a discount based on the member's number of movie rentals and the number of new members referred by the member. The discount is in percent and is equal to the sum of the rentals and the referrals, but it cannot exceed 75 percent. (*Hint:* Math.min.) Write a program Discount-Calculator to calculate the value of the discount.

Here is a sample run:

```
Enter the number of movie rentals: 56
Enter the number of members referred to the video club: 3
The discount is equal to: 59.00 percent.
```

- **Science P2.28** Consider the following circuit.



Write a program that reads the resistances of the three resistors and computes the total resistance, using Ohm's law.

- ■ **Science P2.29** The dew point temperature  $T_d$  can be calculated (approximately) from the relative humidity  $RH$  and the actual temperature  $T$  by

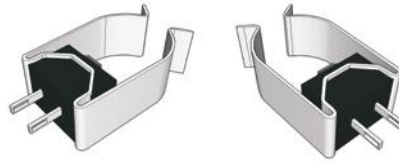
$$T_d = \frac{b \cdot f(T, RH)}{a - f(T, RH)}$$

$$f(T, RH) = \frac{a \cdot T}{b + T} + \ln(RH)$$

where  $a = 17.27$  and  $b = 237.7^\circ \text{C}$ .

Write a program that reads the relative humidity (between 0 and 1) and the temperature (in degrees C) and prints the dew point value. Use the Java function `log` to compute the natural logarithm.

- ■ ■ **Science P2.30** The pipe clip temperature sensors shown here are robust sensors that can be clipped directly onto copper pipes to measure the temperature of the liquids in the pipes.



Each sensor contains a device called a *thermistor*. Thermistors are semiconductor devices that exhibit a temperature-dependent resistance described by:

$$R = R_0 e^{\beta \left( \frac{1}{T} - \frac{1}{T_0} \right)}$$

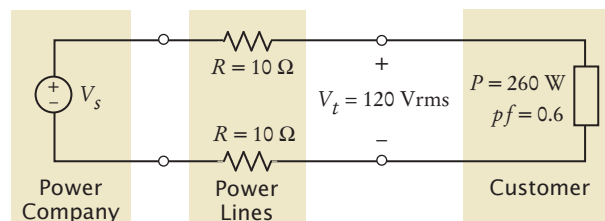
where  $R$  is the resistance (in  $\Omega$ ) at the temperature  $T$  (in  $^{\circ}\text{K}$ ), and  $R_0$  is the resistance (in  $\Omega$ ) at the temperature  $T_0$  (in  $^{\circ}\text{K}$ ).  $\beta$  is a constant that depends on the material used to make the thermistor. Thermistors are specified by providing values for  $R_0$ ,  $T_0$ , and  $\beta$ .

The thermistors used to make the pipe clip temperature sensors have  $R_0 = 1075 \Omega$  at  $T_0 = 85^{\circ}\text{C}$ , and  $\beta = 3969^{\circ}\text{K}$ . (Notice that  $\beta$  has units of  $^{\circ}\text{K}$ . Recall that the temperature in  $^{\circ}\text{K}$  is obtained by adding 273 to the temperature in  $^{\circ}\text{C}$ .) The liquid temperature, in  $^{\circ}\text{C}$ , is determined from the resistance  $R$ , in  $\Omega$ , using

$$T = \frac{\beta T_0}{T_0 \ln \left( \frac{R}{R_0} \right) + \beta} - 273$$

Write a Java program that prompts the user for the thermistor resistance  $R$  and prints a message giving the liquid temperature in  $^{\circ}\text{C}$ .

■■■ **Science P2.31** The circuit shown below illustrates some important aspects of the connection between a power company and one of its customers. The customer is represented by three parameters,  $V_t$ ,  $P$ , and  $pf$ .  $V_t$  is the voltage accessed by plugging into a wall outlet. Customers depend on having a dependable value of  $V_t$  in order for their appliances to work properly. Accordingly, the power company regulates the value of  $V_t$  carefully.  $P$  describes the amount of power used by the customer and is the primary factor in determining the customer's electric bill. The power factor,  $pf$ , is less familiar. (The power factor is calculated as the cosine of an angle so that its value will always be between zero and one.) In this problem you will be asked to write a Java program to investigate the significance of the power factor.

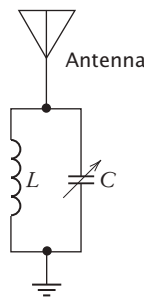


In the figure, the power lines are represented, somewhat simplistically, as resistances in Ohms. The power company is represented as an AC voltage source. The source voltage,  $V_s$ , required to provide the customer with power  $P$  at voltage  $V_t$  can be determined using the formula

$$V_s = \sqrt{\left(V_t + \frac{2RP}{V_t}\right)^2 + \left(\frac{2RP}{pfV_t}\right)^2 (1 - pf^2)}$$

( $V_s$  has units of Vrms.) This formula indicates that the value of  $V_s$  depends on the value of  $pf$ . Write a Java program that prompts the user for a power factor value and then prints a message giving the corresponding value of  $V_s$ , using the values for  $P$ ,  $R$ , and  $V_t$  shown in the figure above.

- ■ ■ **Science P2.32** Consider the following tuning circuit connected to an antenna, where  $C$  is a variable capacitor whose capacitance ranges from  $C_{\min}$  to  $C_{\max}$ .



The tuning circuit selects the frequency  $f = \frac{2\pi}{\sqrt{LC}}$ . To design this circuit for a given frequency, take  $C = \sqrt{C_{\min} C_{\max}}$  and calculate the required inductance  $L$  from  $f$  and  $C$ . Now the circuit can be tuned to any frequency in the range  $f_{\min} = \frac{2\pi}{\sqrt{LC_{\max}}}$  to  $f_{\max} = \frac{2\pi}{\sqrt{LC_{\min}}}$ .

Write a Java program to design a tuning circuit for a given frequency, using a variable capacitor with given values for  $C_{\min}$  and  $C_{\max}$ . (A typical input is  $f = 16.7$  MHz,  $C_{\min} = 14$  pF, and  $C_{\max} = 365$  pF.) The program should read in  $f$  (in Hz),  $C_{\min}$  and  $C_{\max}$  (in F), and print the required inductance value and the range of frequencies to which the circuit can be tuned by varying the capacitance.

- **Science P2.33** According to the Coulomb force law, the electric force between two charged particles of charge  $Q_1$  and  $Q_2$  Coulombs, that are a distance  $r$  meters apart, is

$$F = \frac{Q_1 Q_2}{4\pi\epsilon r^2} \text{ Newtons, where } \epsilon = 8.854 \times 10^{-12} \text{ Farads/meter.}$$

Write a program that calculates the force on a pair of charged particles, based on the user input of  $Q_1$  Coulombs,  $Q_2$  Coulombs, and  $r$  meters, and then computes and displays the electric force.

## ANSWERS TO SELF-CHECK QUESTIONS

1. One possible answer is  

```
int bottlesPerCase = 8;
```

You may choose a different variable name or a different initialization value, but your variable should have type `int`.
2. There are three errors:
  - You cannot have spaces in variable names.
  - The variable type should be `double` because it holds a fractional value.
  - There is a semicolon missing at the end of the statement.
3. 

```
double unitPrice = 1.95;
int quantity = 2;
```
4. 

```
System.out.print("Total price: ");
System.out.println(unitPrice * quantity);
```
5. Change the declaration of `cansPerPack` to  

```
int cansPerPack = 4;
```
6. You need to use a `*/` delimiter to close a comment that begins with a `/*`:  

```
double canVolume = 0.355;
/* Liters in a 12-ounce can */
```
7. The program would compile, and it would display the same result. However, a person reading the program might find it confusing that fractional cans are being considered.
8. Its value is modified by the assignment statement.
9. Assignment would occur when one car is replaced by another in the parking space.
10. 

```
double interest = balance * percent / 100;
```
11. 

```
double sideLength = Math.sqrt(area);
```
12. 

```
4 * PI * Math.pow(radius, 3) / 3
```

or 

```
(4.0 / 3) * PI * Math.pow(radius, 3),
```

  
but not 

```
(4 / 3) * PI * Math.pow(radius, 3)
```
13. 172 and 9
14. It is the second-to-last digit of `n`. For example, if `n` is 1729, then `n / 10` is 172, and `(n / 10) % 10` is 2.
15. 

```
System.out.print("How old are you? ");
int age = in.nextInt();
```
16. There is no prompt that alerts the program user to enter the quantity.
17. The second statement calls `nextInt`, not `nextDouble`. If the user were to enter a price such as 1.95, the program would be terminated with an “input mismatch exception”.
18. There is no colon and space at the end of the prompt. A dialog would look like this:  
Please enter the number of cans6
19. The total volume is 10  
There are four spaces between `is` and 10. One space originates from the format string (the space between `s` and `%`), and three spaces are added before 10 to achieve a field width of 5.
20. Here is a simple solution:  

```
System.out.printf("Bottles: %8d\n", bottles);
System.out.printf("Cans:   %8d\n", cans);
```

Note the spaces after `Cans:`. Alternatively, you can use format specifiers for the strings. You can even combine all output into a single statement:  

```
System.out.printf("%-9s%8d\n%-9s%8d\n",
"Bottles: ", bottles, "Cans:", cans);
```
21. 

```
int pairs = (totalWidth - tileWidth)
/ (2 * tileWidth);
int tiles = 1 + 2 * pairs;
double gap = (totalWidth -
tiles * tileWidth) / 2.0;
```

Be sure that `pairs` is declared as an `int`.
22. Now there are groups of four tiles (gray/white/gray/black) following the initial black tile. Therefore, the algorithm is now  

$$\text{number of groups} = \text{integer part of } (\text{total width} - \text{tile width}) / (4 \times \text{tile width})$$

$$\text{number of tiles} = 1 + 4 \times \text{number of groups}$$

The formula for the gap is not changed.
23. Clearly, the answer depends only on whether the row and column numbers are even or odd, so let's first take the remainder after dividing by 2. Then we can enumerate all expected answers:

Row % 2	Column % 2	Color
0	0	0
0	1	1
1	0	1
1	1	0

In the first three entries of the table, the color is simply the sum of the remainders. In the fourth entry, the sum would be 2, but we want a zero. We can achieve that by taking another remainder operation:

**color = ((row % 2) + (column % 2)) % 2**

- 24.** In nine years, the repair costs increased by \$1,400. Therefore, the increase per year is  $\$1,400 / 9 \approx \$156$ . The repair cost in year 3 would be  $\$100 + 2 \times \$156 = \$412$ . The repair cost in year *n* is  $\$100 + n \times \$156$ . To avoid accumulation of roundoff errors, it is actually a good idea to use the original expression that yielded \$156, that is,

**Repair cost in year *n* =  $100 + n \times 1400 / 9$**

- 25.** The pseudocode follows easily from the equations:

**bottom volume =  $\pi \times r_1^2 \times h_1$**

**top volume =  $\pi \times r_2^2 \times h_2$**

**middle volume =  $\pi \times (r_1^2 + r_1 \times r_2 + r_2^2) \times h_3 / 3$**

**total volume = bottom volume + top volume + middle volume**

Measuring a typical wine bottle yields  $r_1 = 3.6$ ,  $r_2 = 1.2$ ,  $h_1 = 15$ ,  $h_2 = 7$ ,  $h_3 = 6$  (all in centimeters). Therefore,

bottom volume = 610.73

top volume = 31.67

middle volume = 135.72

total volume = 778.12

The actual volume is 750 ml, which is close enough to our computation to give confidence that it is correct.

- 26.** The length is 12. The space counts as a character.

- 27.** `str.substring(8, 12)` or `str.substring(8)`

- 28.** `str = str + "ming";`

- 29.** `Hy`

- 30.** `String first = in.next();`  
`String middle = in.next();`  
`String last = in.next();`