2.1 Introduction

Key Point

The focus of this chapter is on learning elementary programming techniques to solve problems.

In Chapter 1 you learned how to create, compile, and run very basic Java programs. Now you will learn how to solve problems by writing programs. Through these problems, you will learn elementary programming using primitive data types, variables, constants, operators, expressions, and input and output.

Suppose, for example, that you need to take out a student loan. Given the loan amount, loan term, and annual interest rate, can you write a program to compute the monthly payment and total payment? This chapter shows you how to write programs like this. Along the way, you learn the basic steps that go into analyzing a problem, designing a solution, and implementing the solution by creating a program.

2.2 Writing a Simple Program

Key Point

Writing a program involves designing a strategy for solving the problem and then using a programming language to implement that strategy.

problem

Let's first consider the simple problem of computing the area of a circle. How do we write a program for solving this problem?

algorithm

pseudocode

Writing a program involves designing algorithms and translating algorithms into programming instructions, or code. An *algorithm* describes how a problem is solved by listing the actions that need to be taken and the order of their execution. Algorithms can help the programmer plan a program before writing it in a programming language. Algorithms can be described in natural languages or in *pseudocode* (natural language mixed with some programming code). The algorithm for calculating the area of a circle can be described as follows:

1. Read in the circle's radius.
2. Compute the area using the following formula:

$$area = radius \times radius \times \pi$$

3. Display the result.

Tip

It's always good practice to outline your program (or its underlying problem) in the form of an algorithm before you begin coding.

When you *code*—that is, when you write a program—you translate an algorithm into a program. You already know that every Java program begins with a class definition in which the keyword `class` is followed by the class name. Assume that you have chosen `ComputeArea` as the class name. The outline of the program would look like this:

```
public class ComputeArea {
  // Details to be given later
}
```

As you know, every Java program must have a `main` method where program execution begins. The program is then expanded as follows:

```
public class ComputeArea {
  public static void main(String[] args) {
    // Step 1: Read in radius

    // Step 2: Compute area
```

```
    // Step 3: Display the area
  }
}
```

The program needs to read the radius entered by the user from the keyboard. This raises two important issues:

- Reading the radius.
- Storing the radius in the program.

Let's address the second issue first. In order to store the radius, the program needs to declare a symbol called a *variable.* A variable represents a value stored in the computer's memory.

variable

Rather than using **x** and **y** as variable names, choose descriptive names: in this case, **radius** for radius, and **area** for area. To let the compiler know what **radius** and **area** are, specify their data types. That is the kind of data stored in a variable, whether integer, real number, or something else. This is known as *declaring variables.* Java provides simple data types for representing integers, real numbers, characters, and Boolean types. These types are known as *primitive data types* or *fundamental types.*

descriptive names

data type

declare variables

primitive data types

Real numbers (i.e., numbers with a decimal point) are represented using a method known as *floating-point* in computers. So, the real numbers are also called *floating-point numbers*. In Java, you can use the keyword **double** to declare a floating-point variable. Declare **radius** and **area** as **double**. The program can be expanded as follows:

floating-point number

```
public class ComputeArea {
  public static void main(String[] args) {
    double radius;
    double area;

    // Step 1: Read in radius

    // Step 2: Compute area

    // Step 3: Display the area
  }
}
```

The program declares **radius** and **area** as variables. The reserved word **double** indicates that **radius** and **area** are floating-point values stored in the computer.

The first step is to prompt the user to designate the circle's **radius**. You will soon learn how to prompt the user for information. For now, to learn how variables work, you can assign a fixed value to **radius** in the program as you write the code; later, you'll modify the program to prompt the user for this value.

The second step is to compute **area** by assigning the result of the expression **radius * radius * 3.14159** to **area**.

In the final step, the program will display the value of **area** on the console by using the **System.out.println** method.

Listing 2.1 shows the complete program, and a sample run of the program is shown in Figure 2.1.

LISTING 2.1 ComputeArea.java

```
public class ComputeArea {
  public static void main(String[] args) {
    double radius; // Declare radius
    double area; // Declare area

    // Assign a radius
```

```
 7       radius = 20; // radius is now 20
 8
 9       // Compute area
10       area = radius * radius * 3.14159;
11
12       // Display results
13       System.out.println("The area for the circle of radius " +
14         radius + " is " + area);
15     }
16   }
```

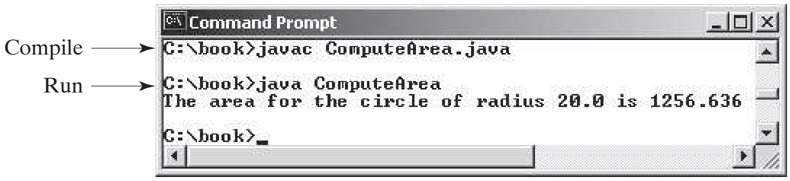


FIGURE 2.1 The program displays the area of a circle.

declare variable

assign value

tracing program

Variables such as **radius** and **area** correspond to memory locations. Every variable has a name, a type, a size, and a value. Line 3 declares that **radius** can store a **double** value. The value is not defined until you assign a value. Line 7 assigns **20** into variable **radius**. Similarly, line 4 declares variable **area**, and line 10 assigns a value into **area**. The following table shows the value in the memory for **area** and **radius** as the program is executed. Each row in the table shows the values of variables after the statement in the corresponding line in the program is executed. This method of reviewing how a program works is called *tracing a program*. Tracing programs are helpful for understanding how programs work, and they are useful tools for finding errors in programs.

line#	radius	area
3	no value	
4		no value
7	20	
10		1256.636

concatenate strings

concatenate strings with numbers

The plus sign (+) has two meanings: one for addition and the other for concatenating (combining) strings. The plus sign (+) in lines 13–14 is called a *string concatenation operator*. It combines two strings into one. If a string is combined with a number, the number is converted into a string and concatenated with the other string. Therefore, the plus signs (+) in lines 13–14 concatenate strings into a longer string, which is then displayed in the output. Strings and string concatenation will be discussed further in Chapter 4.

Caution

A string cannot cross lines in the source code. Thus, the following statement would result in a compile error:

```
System.out.println("Introduction to Java Programming,
  by Y. Daniel Liang");
```

break a long string

To fix the error, break the string into separate substrings, and use the concatenation operator (+) to combine them:

```
System.out.println("Introduction to Java Programming, " +
  "by Y. Daniel Liang");
```

Check Point

2.1 Identify and fix the errors in the following code:

```
public class Test {
  public void main(string[] args) {
    double i = 50.0;
    double k = i + 50.0;
    double j = k + 1;

    System.out.println("j is " + j + " and
      k is " + k);
  }
}
```

2.3 Reading Input from the Console

Reading input from the console enables the program to accept input from the user.

Key Point

In Listing 2.1, the radius is fixed in the source code. To use a different radius, you have to modify the source code and recompile it. Obviously, this is not convenient, so instead you can use the `Scanner` class for console input.

VideoNote
Obtain input

Java uses `System.out` to refer to the standard output device and `System.in` to the standard input device. By default, the output device is the display monitor and the input device is the keyboard. To perform console output, you simply use the `println` method to display a primitive value or a string to the console. Console input is not directly supported in Java, but you can use the `Scanner` class to create an object to read input from `System.in`, as follows:

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

The syntax `new Scanner(System.in)` creates an object of the `Scanner` type. The syntax `Scanner input` declares that `input` is a variable whose type is `Scanner`. The whole line `Scanner input = new Scanner(System.in)` creates a `Scanner` object and assigns its reference to the variable `input`. An object may invoke its methods. To invoke a method on an object is to ask the object to perform a task. You can invoke the `nextDouble()` method to read a `double` value as follows:

```
double radius = input.nextDouble();
```

This statement reads a number from the keyboard and assigns the number to `radius`.
Listing 2.2 rewrites Listing 2.1 to prompt the user to enter a radius.

LISTING 2.2 ComputeAreaWithConsoleInput.java

```
import java.util.Scanner; // Scanner is in the java.util package

public class ComputeAreaWithConsoleInput {
  public static void main(String[] args) {
    // Create a Scanner object
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter a radius
    System.out.print("Enter a number for radius: ");
    double radius = input.nextDouble();

    // Compute area
    double area = radius * radius * 3.14159;

    // Display results
```

import class

create a `Scanner`

read a `double`

```
16       System.out.println("The area for the circle of radius " +
17         radius + " is " + area);
18     }
19   }
```

```
Enter a number for radius:  2.5 ↵Enter
The area for the circle of radius 2.5 is 19.6349375
```

```
Enter a number for radius:  23 ↵Enter
The area for the circle of radius 23.0 is 1661.90111
```

prompt

Line 9 displays a string **"Enter a number for radius: "** to the console. This is known as a *prompt*, because it directs the user to enter an input. Your program should always tell the user what to enter when expecting input from the keyboard.

The **print** method in line 9

```
System.out.print("Enter a number for radius: ");
```

print vs. println

is identical to the **println** method except that **println** moves to the beginning of the next line after displaying the string, but **print** does not advance to the next line when completed.

Line 6 creates a **Scanner** object. The statement in line 10 reads input from the keyboard.

```
double radius = input.nextDouble();
```

After the user enters a number and presses the *Enter* key, the program reads the number and assigns it to **radius**.

More details on objects will be introduced in Chapter 9. For the time being, simply accept that this is how to obtain input from the console.

specific import

The **Scanner** class is in the **java.util** package. It is imported in line 1. There are two types of **import** statements: *specific import* and *wildcard import*. The *specific import* specifies a single class in the import statement. For example, the following statement imports **Scanner** from the package **java.util**.

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

wildcard import

The *wildcard import* imports all the classes in a package by using the asterisk as the wildcard. For example, the following statement imports all the classes from the package **java.util**.

```
import java.uitl.*;
```

no performance difference

The information for the classes in an imported package is not read in at compile time or runtime unless the class is used in the program. The import statement simply tells the compiler where to locate the classes. There is no performance difference between a specific import and a wildcard import declaration.

Listing 2.3 gives an example of reading multiple input from the keyboard. The program reads three numbers and displays their average.

Listing 2.3 ComputeAverage.java

import class

create a Scanner

```
import java.util.Scanner; // Scanner is in the java.util package

public class ComputeAverage {
  public static void main(String[] args) {
    // Create a Scanner object
    Scanner input = new Scanner(System.in);

```

```
8       // Prompt the user to enter three numbers
9       System.out.print("Enter three numbers: ");
10      double number1 = input.nextDouble();
11      double number2 = input.nextDouble();
12      double number3 = input.nextDouble();
13
14      // Compute average
15      double average = (number1 + number2 + number3) / 3;
16
17      // Display results
18      System.out.println("The average of " + number1 + " " + number2
19        + " " + number3 + " is " + average);
20    }
21  }
```

read a double

```
Enter three numbers: 1 2 3 ↵Enter
The average of 1.0 2.0 3.0 is 2.0
```

enter input in one line

```
Enter three numbers: 10.5 ↵Enter
11 ↵Enter
11.5 ↵Enter
The average of 10.5 11.0 11.5 is 11.0
```

enter input in multiple lines

The code for importing the **Scanner** class (line 1) and creating a **Scanner** object (line 6) are the same as in the preceding example as well as in all new programs you will write for reading input from the keyboard.

Line 9 prompts the user to enter three numbers. The numbers are read in lines 10–12. You may enter three numbers separated by spaces, then press the *Enter* key, or enter each number followed by a press of the *Enter* key, as shown in the sample runs of this program.

If you entered an input other than a numeric value, a runtime error would occur. In Chapter 12, you will learn how to handle the exception so that the program can continue to run.

runtime error

Note

Most of the programs in the early chapters of this book perform three steps—input, process, and output—called *IPO*. Input is receiving input from the user; process is producing results using the input; and output is displaying the results.

IPO

Check Point

2.2 How do you write a statement to let the user enter a double value from the keyboard? What happens if you entered **5a** when executing the following code?

```
double radius = input.nextDouble();
```

2.3 Are there any performance differences between the following two **import** statements?

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

2.4 Identifiers

Identifiers are the names that identify the elements such as classes, methods, and variables in a program.

Key Point

As you see in Listing 2.3, **ComputeAverage**, **main**, **input**, **number1**, **number2**, **number3**, and so on are the names of things that appear in the program. In programming terminology, such names are called *identifiers*. All identifiers must obey the following rules:

identifiers

identifier naming rules

- An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar signs (**$**).

- An identifier must start with a letter, an underscore (_), or a dollar sign ($). It cannot start with a digit.
- An identifier cannot be a reserved word. (See Appendix A for a list of reserved words.)
- An identifier cannot be `true`, `false`, or `null`.
- An identifier can be of any length.

For example, `$2`, `ComputeArea`, `area`, `radius`, and `print` are legal identifiers, whereas `2A` and `d+4` are not because they do not follow the rules. The Java compiler detects illegal identifiers and reports syntax errors.

case sensitive

Note

Since Java is case sensitive, `area`, `Area`, and `AREA` are all different identifiers.

descriptive names

Tip

Identifiers are for naming variables, methods, classes, and other items in a program. Descriptive identifiers make programs easy to read. Avoid using abbreviations for identifiers. Using complete words is more descriptive. For example, `numberOfStudents` is better than `numStuds`, `numOfStuds`, or `numOfStudents`. We use descriptive names for complete programs in the text. However, we will occasionally use variable names such as `i`, `j`, `k`, `x`, and `y` in the code snippets for brevity. These names also provide a generic tone to the code snippets.

the $ character

Tip

Do not name identifiers with the `$` character. By convention, the `$` character should be used only in mechanically generated source code.

Check Point

2.4 Which of the following identifiers are valid? Which are Java keywords?

`miles`, `Test`, `a++`, `--a`, `4#R`, `$4`, `#44`, `apps`

`class`, `public`, `int`, `x`, `y`, `radius`

2.5 Variables

Key Point

Variables are used to represent values that may be changed in the program.

why called variables?

As you see from the programs in the preceding sections, variables are used to store values to be used later in a program. They are called variables because their values can be changed. In the program in Listing 2.2, `radius` and `area` are variables of the `double` type. You can assign any numerical value to `radius` and `area`, and the values of `radius` and `area` can be reassigned. For example, in the following code, `radius` is initially `1.0` (line 2) and then changed to `2.0` (line 7), and area is set to `3.14159` (line 3) and then reset to `12.56636` (line 8).

```
1  // Compute the first area
2  radius = 1.0;                                    radius: [1.0     ]
3  area = radius * radius * 3.14159;                  area: [3.14159 ]
4  System.out.println("The area is " + area + " for radius " + radius);
5
6  // Compute the second area
7  radius = 2.0;                                    radius: [2.0     ]
8  area = radius * radius * 3.14159;                  area: [12.56636]
9  System.out.println("The area is " + area + " for radius " + radius);
```

Variables are for representing data of a certain type. To use a variable, you declare it by telling the compiler its name as well as what type of data it can store. The *variable declaration* tells the compiler to allocate appropriate memory space for the variable based on its data type. The syntax for declaring a variable is

```
datatype variableName;
```

Here are some examples of variable declarations:

declare variable

```
int count;           // Declare count to be an integer variable
double radius;       // Declare radius to be a double variable
double interestRate; // Declare interestRate to be a double variable
```

These examples use the data types **int** and **double**. Later you will be introduced to additional data types, such as **byte**, **short**, **long**, **float**, **char**, and **boolean**.

If variables are of the same type, they can be declared together, as follows:

```
datatype variable1, variable2, ..., variablen;
```

The variables are separated by commas. For example,

```
int i, j, k; // Declare i, j, and k as int variables
```

initialize variables

Variables often have initial values. You can declare a variable and initialize it in one step. Consider, for instance, the following code:

```
int count = 1;
```

This is equivalent to the next two statements:

```
int count;
count = 1;
```

You can also use a shorthand form to declare and initialize variables of the same type together. For example,

```
int i = 1, j = 2;
```

Tip

A variable must be declared before it can be assigned a value. A variable declared in a method must be assigned a value before it can be used.

Whenever possible, declare a variable and assign its initial value in one step. This will make the program easy to read and avoid programming errors.

Every variable has a scope. The *scope of a variable* is the part of the program where the variable can be referenced. The rules that define the scope of a variable will be introduced gradually later in the book. For now, all you need to know is that a variable must be declared and initialized before it can be used.

Check Point

2.5 Identify and fix the errors in the following code:

```
public class Test {
  public static void main(String[] args) {
    int i = k + 2;
    System.out.println(i);
  }
}
```

2.6 Assignment Statements and Assignment Expressions

Key Point

An assignment statement designates a value for a variable. An assignment statement can be used as an expression in Java.

assignment statement
assignment operator

After a variable is declared, you can assign a value to it by using an *assignment statement*. In Java, the equal sign (`=`) is used as the *assignment operator*. The syntax for assignment statements is as follows:

```
variable = expression;
```

expression

An *expression* represents a computation involving values, variables, and operators that, taking them together, evaluates to a value. For example, consider the following code:

```
int y = 1;                    // Assign 1 to variable y
double radius = 1.0;          // Assign 1.0 to variable radius
int x = 5 * (3 / 2);          // Assign the value of the expression to x
x = y + 1;                    // Assign the addition of y and 1 to x
double area = radius * radius * 3.14159; // Compute area
```

You can use a variable in an expression. A variable can also be used in both sides of the = operator. For example,

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

In this assignment statement, the result of `x + 1` is assigned to `x`. If `x` is `1` before the statement is executed, then it becomes `2` after the statement is executed.

To assign a value to a variable, you must place the variable name to the left of the assignment operator. Thus, the following statement is wrong:

```
1 = x;  // Wrong
```

Note

In mathematics, `x = 2 * x + 1` denotes an equation. However, in Java, `x = 2 * x + 1` is an assignment statement that evaluates the expression `2 * x + 1` and assigns the result to `x`.

assignment expression

In Java, an assignment statement is essentially an expression that evaluates to the value to be assigned to the variable on the left side of the assignment operator. For this reason, an assignment statement is also known as an *assignment expression*. For example, the following statement is correct:

```
System.out.println(x = 1);
```

which is equivalent to

```
x = 1;
System.out.println(x);
```

If a value is assigned to multiple variables, you can use this syntax:

```
i = j = k = 1;
```

which is equivalent to

```
k = 1;
j = k;
i = j;
```

Note

In an assignment statement, the data type of the variable on the left must be compatible with the data type of the value on the right. For example, `int x = 1.0` would be illegal, because the data type of `x` is `int`. You cannot assign a `double` value (`1.0`) to an `int` variable without using type casting. Type casting is introduced in Section 2.15.

Check Point

2.6 Identify and fix the errors in the following code:

```
public class Test {
  public static void main(String[] args) {
    int i = j = k = 2;
    System.out.println(i + " " + j + " " + k);
  }
}
```

2.7 Named Constants

A named constant is an identifier that represents a permanent value.

Key Point

constant

The value of a variable may change during the execution of a program, but a *named constant,* or simply *constant*, represents permanent data that never changes. In our `ComputeArea` program, π is a constant. If you use it frequently, you don't want to keep typing `3.14159`; instead, you can declare a constant for π. Here is the syntax for declaring a constant:

```
final datatype CONSTANTNAME = value;
```

final keyword

A constant must be declared and initialized in the same statement. The word `final` is a Java keyword for declaring a constant. For example, you can declare π as a constant and rewrite Listing 2.1 as in Listing 2.4.

LISTING 2.4 ComputeAreaWithConstant.java

```
import java.util.Scanner; // Scanner is in the java.util package

public class ComputeAreaWithConstant {
  public static void main(String[] args) {
    final double PI = 3.14159; // Declare a constant

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

    // Prompt the user to enter a radius
    System.out.print("Enter a number for radius: ");
    double radius = input.nextDouble();

    // Compute area
    double area = radius * radius * PI;

    // Display result
    System.out.println("The area for the circle of radius " +
      radius + " is " + area);
  }
}
```

benefits of constants

There are three benefits of using constants: (1) you don't have to repeatedly type the same value if it is used multiple times; (2) if you have to change the constant value (e.g., from `3.14` to `3.14159` for `PI`), you need to change it only in a single location in the source code; and (3) a descriptive name for a constant makes the program easy to read.

2.8 Naming Conventions

Key Point

Sticking with the Java naming conventions makes your programs easy to read and avoids errors.

Make sure that you choose descriptive names with straightforward meanings for the variables, constants, classes, and methods in your program. As mentioned earlier, names are case sensitive. Listed below are the conventions for naming variables, methods, and classes.

name variables and methods

- Use lowercase for variables and methods. If a name consists of several words, concatenate them into one, making the first word lowercase and capitalizing the first letter of each subsequent word—for example, the variables `radius` and `area` and the method `print`.

name classes

- Capitalize the first letter of each word in a class name—for example, the class names `ComputeArea` and `System`.

name constants

- Capitalize every letter in a constant, and use underscores between words—for example, the constants `PI` and `MAX_VALUE`.

It is important to follow the naming conventions to make your programs easy to read.

name classes

Caution

Do not choose class names that are already used in the Java library. For example, since the `System` class is defined in Java, you should not name your class `System`.

Check Point

2.7 What are the benefits of using constants? Declare an `int` constant `SIZE` with value `20`.

2.8 What are the naming conventions for class names, method names, constants, and variables? Which of the following items can be a constant, a method, a variable, or a class according to the Java naming conventions?

`MAX_VALUE`, `Test`, `read`, `readDouble`

2.9 Translate the following algorithm into Java code:

Step 1: Declare a `double` variable named `miles` with initial value `100`.

Step 2: Declare a `double` constant named `KILOMETERS_PER_MILE` with value `1.609`.

Step 3: Declare a `double` variable named `kilometers`, multiply `miles` and `KILOMETERS_PER_MILE`, and assign the result to `kilometers`.

Step 4: Display `kilometers` to the console.

What is `kilometers` after Step 4?

2.9 Numeric Data Types and Operations

Key Point

*Java has six numeric types for integers and floating-point numbers with operators +, -, *, /, and %.*

2.9.1 Numeric Types

Every data type has a range of values. The compiler allocates memory space for each variable or constant according to its data type. Java provides eight primitive data types for numeric values, characters, and Boolean values. This section introduces numeric data types and operators.

Table 2.1 lists the six numeric data types, their ranges, and their storage sizes.

TABLE 2.1 Numeric Data Types

Name	*Range*	*Storage Size*
`byte`	-2^7 to $2^7 - 1$ (-128 to 127)	8-bit signed
`short`	-2^{15} to $2^{15} - 1$ (-32768 to 32767)	16-bit signed
`int`	-2^{31} to $2^{31} - 1$ (-2147483648 to 2147483647)	32-bit signed
`long`	-2^{63} to $2^{63} - 1$ (i.e., -9223372036854775808 to 9223372036854775807)	64-bit signed
`float`	Negative range: $-3.4028235E + 38$ to $-1.4E - 45$ Positive range: $1.4E - 45$ to $3.4028235E + 38$	32-bit IEEE 754
`double`	Negative range: $-1.7976931348623157E + 308$ to $-4.9E - 324$ Positive range: $4.9E - 324$ to $1.7976931348623157E + 308$	64-bit IEEE 754

`byte` type

`short` type

`int` type

`long` type

`float` type

`double` type

Note

IEEE 754 is a standard approved by the Institute of Electrical and Electronics Engineers for representing floating-point numbers on computers. The standard has been widely adopted. Java uses the 32-bit **IEEE 754** for the `float` type and the 64-bit **IEEE 754** for the `double` type. The **IEEE 754** standard also defines special floating-point values, which are listed in Appendix E.

integer types

Java uses four types for integers: `byte`, `short`, `int`, and `long`. Choose the type that is most appropriate for your variable. For example, if you know an integer stored in a variable is within a range of a byte, declare the variable as a `byte`. For simplicity and consistency, we will use `int` for integers most of the time in this book.

floating-point types

Java uses two types for floating-point numbers: `float` and `double`. The `double` type is twice as big as `float`, so the `double` is known as *double precision* and `float` as *single precision*. Normally, you should use the `double` type, because it is more accurate than the `float` type.

2.9.2 Reading Numbers from the Keyboard

You know how to use the `nextDouble()` method in the `Scanner` class to read a double value from the keyboard. You can also use the methods listed in Table 2.2 to read a number of the `byte`, `short`, `int`, `long`, and `float` type.

TABLE 2.2 Methods for `Scanner` Objects

Method	*Description*
`nextByte()`	reads an integer of the `byte` type.
`nextShort()`	reads an integer of the `short` type.
`nextInt()`	reads an integer of the `int` type.
`nextLong()`	reads an integer of the `long` type.
`nextFloat()`	reads a number of the `float` type.
`nextDouble()`	reads a number of the `double` type.

Here are examples for reading values of various types from the keyboard:

```
1  Scanner input = new Scanner(System.in);
2  System.out.print("Enter a byte value: ");
3  byte byteValue = input.nextByte();
4
5  System.out.print("Enter a short value: ");
6  short shortValue = input.nextShort();
7
8  System.out.print("Enter an int value: ");
9  int intValue = input.nextInt();
10
11 System.out.print("Enter a long value: ");
12 long longValue = input.nextLong();
13
14 System.out.print("Enter a float value: ");
15 float floatValue = input.nextFloat();
```

If you enter a value with an incorrect range or format, a runtime error would occur. For example, you enter a value `128` for line 3, an error would occur because `128` is out of range for a `byte` type integer.

2.9.3 Numeric Operators

operators +, -, *, /, %

operands

The operators for numeric data types include the standard arithmetic operators: addition (`+`), subtraction (`–`), multiplication (`*`), division (`/`), and remainder (`%`), as shown in Table 2.3. The *operands* are the values operated by an operator.

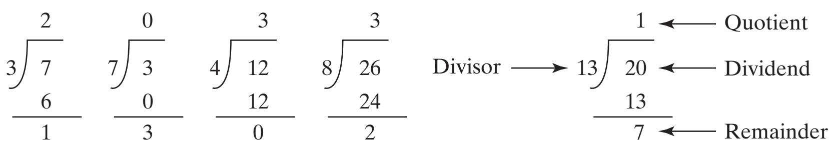
TABLE 2.3 Numeric Operators

Name	*Meaning*	*Example*	*Result*
`+`	Addition	34 + 1	35
`-`	Subtraction	34.0 – 0.1	33.9
`*`	Multiplication	300 * 30	9000
`/`	Division	1.0 / 2.0	0.5
`%`	Remainder	20 % 3	2

integer division

When both operands of a division are integers, the result of the division is the quotient and the fractional part is truncated. For example, `5 / 2` yields `2`, not `2.5`, and `-5 / 2` yields `-2`, not `-2.5`. To perform a float-point division, one of the operands must be a floating-point number. For example, `5.0 / 2` yields `2.5`.

The `%` operator, known as *remainder* or *modulo* operator, yields the remainder after division. The operand on the left is the dividend and the operand on the right is the divisor. Therefore, `7 % 3` yields `1`, `3 % 7` yields `3`, `12 % 4` yields `0`, `26 % 8` yields `2`, and `20 % 13` yields `7`.



The `%` operator is often used for positive integers, but it can also be used with negative integers and floating-point values. The remainder is negative only if the dividend is negative. For example, `-7 % 3` yields `-1`, `-12 % 4` yields `0`, `-26 % -8` yields `-2`, and `20 % -13` yields `7`.

Remainder is very useful in programming. For example, an even number `% 2` is always `0` and an odd number `% 2` is always `1`. Thus, you can use this property to determine whether a number is even or odd. If today is Saturday, it will be Saturday again in 7 days. Suppose you and your friends are going to meet in 10 days. What day is in 10 days? You can find that the day is Tuesday using the following expression:

Day 6 in a week is Saturday

A week has 7 days

(6 + 10) % 7 is 2

After 10 days

Day 2 in a week is Tuesday
Note: Day 0 in a week is Sunday

The program in Listing 2.5 obtains minutes and remaining seconds from an amount of time in seconds. For example, `500` seconds contains `8` minutes and `20` seconds.

LISTING 2.5 DisplayTime.java

```
1  import java.util.Scanner;
2
3  public class DisplayTime {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6      // Prompt the user for input
7      System.out.print("Enter an integer for seconds: ");
8      int seconds = input.nextInt();
9
10     int minutes = seconds / 60; // Find minutes in seconds
11     int remainingSeconds = seconds % 60; // Seconds remaining
12     System.out.println(seconds + " seconds is " + minutes +
13       " minutes and " + remainingSeconds + " seconds");
14   }
15 }
```

import Scanner

create a Scanner

read an integer

divide

remainder

```
Enter an integer for seconds: 500 ↵Enter
500 seconds is 8 minutes and 20 seconds
```

line#	seconds	minutes	remainingSeconds
8	500		
10		8	
11			20

The `nextInt()` method (line 8) reads an integer for `seconds`. Line 10 obtains the minutes using `seconds / 60`. Line 11 (`seconds % 60`) obtains the remaining seconds after taking away the minutes.

The `+` and `-` operators can be both unary and binary. A *unary* operator has only one operand; a *binary* operator has two. For example, the `-` operator in `-5` is a unary operator to negate number `5`, whereas the `-` operator in `4 - 5` is a binary operator for subtracting `5` from `4`.

unary operator

binary operator

2.9.4 Exponent Operations

Math.pow(a, b) method

The `Math.pow(a, b)` method can be used to compute a^b. The `pow` method is defined in the `Math` class in the Java API. You invoke the method using the syntax `Math.pow(a, b)` (e.g., `Math.pow(2, 3)`), which returns the result of a^b (2^3). Here, `a` and `b` are parameters for the `pow` method and the numbers `2` and `3` are actual values used to invoke the method. For example,

```
System.out.println(Math.pow(2, 3)); // Displays 8.0
System.out.println(Math.pow(4, 0.5)); // Displays 2.0
System.out.println(Math.pow(2.5, 2)); // Displays 6.25
System.out.println(Math.pow(2.5, -2)); // Displays 0.16
```

Chapter 5 introduces more details on methods. For now, all you need to know is how to invoke the `pow` method to perform the exponent operation.

Check Point

2.10 Find the largest and smallest `byte`, `short`, `int`, `long`, `float`, and `double`. Which of these data types requires the least amount of memory?

2.11 Show the result of the following remainders.

```
 56 %  6
 78 % -4
-34 %  5
-34 % -5
  5 %  1
  1 %  5
```

2.12 If today is Tuesday, what will be the day in 100 days?

2.13 What is the result of `25 / 4`? How would you rewrite the expression if you wished the result to be a floating-point number?

2.14 Show the result of the following code:

```
System.out.println(2 * (5 / 2 + 5 / 2));
System.out.println(2 * 5 / 2 + 2 * 5 / 2);
System.out.println(2 * (5 / 2));
System.out.println(2 * 5 / 2);
```

2.15 Are the following statements correct? If so, show the output.

```
System.out.println("25 / 4 is " + 25 / 4);
System.out.println("25 / 4.0 is " + 25 / 4.0);
System.out.println("3 * 2 / 4 is " + 3 * 2 / 4);
System.out.println("3.0 * 2 / 4 is " + 3.0 * 2 / 4);
```

2.16 Write a statement to display the result of $2^{3.5}$.

2.17 Suppose `m` and `r` are integers. Write a Java expression for mr^2 to obtain a floating-point result.

2.10 Numeric Literals

Key Point

A literal *is a constant value that appears directly in a program.*

literal

For example, `34` and `0.305` are literals in the following statements:

```
int numberOfYears = 34;
double weight = 0.305;
```

2.10.1 Integer Literals

An integer literal can be assigned to an integer variable as long as it can fit into the variable. A compile error will occur if the literal is too large for the variable to hold. The statement `byte b = 128`, for example, will cause a compile error, because `128` cannot be stored in a variable of the `byte` type. (Note that the range for a byte value is from `-128` to `127`.)

An integer literal is assumed to be of the `int` type, whose value is between -2^{31} (-2147483648) and $2^{31} - 1$ (2147483647). To denote an integer literal of the `long` type, append the letter `L` or `l` to it. For example, to write integer `2147483648` in a Java program, you have to write it as `2147483648L` or `2147483648l`, because `2147483648` exceeds the range for the `int` value. `L` is preferred because `l` (lowercase `L`) can easily be confused with 1 (the digit one).

binary, octal, and hex literals

Note

By default, an integer literal is a decimal integer number. To denote a binary integer literal, use a leading ***0b*** or ***0B*** (zero B), to denote an octal integer literal, use a leading ***0*** (zero), and to denote a hexadecimal integer literal, use a leading ***0x*** or ***0X*** (zero X). For example,

```
System.out.println(0B1111); // Displays 15
System.out.println(07777); // Displays 4095
System.out.println(0XFFFF); // Displays 65535
```

Hexadecimal numbers, binary numbers, and octal numbers are introduced in Appendix F.

2.10.2 Floating-Point Literals

Floating-point literals are written with a decimal point. By default, a floating-point literal is treated as a `double` type value. For example, `5.0` is considered a `double` value, not a `float` value. You can make a number a `float` by appending the letter `f` or `F`, and you can make a number a `double` by appending the letter `d` or `D`. For example, you can use `100.2f` or `100.2F` for a `float` number, and `100.2d` or `100.2D` for a `double` number.

suffix f or F

suffix d or D

`double` vs. `float`

Note

The `double` type values are more accurate than the `float` type values. For example,

```
System.out.println("1.0 / 3.0 is " + 1.0 / 3.0);
```

displays `1.0 / 3.0 is 0.3333333333333333`

$\underbrace{3333333333333333}_{\text{16 digits}}$

```
System.out.println("1.0F / 3.0F is " + 1.0F / 3.0F);
```

displays `1.0F / 3.0F is 0.33333334`

$\underbrace{33333334}_{\text{8 digits}}$

A float value has `7` to `8` number of significant digits and a double value has `15` to `17` number of significant digits.

2.10.3 Scientific Notation

Floating-point literals can be written in scientific notation in the form of $a \times 10^b$. For example, the scientific notation for 123.456 is 1.23456×10^2 and for 0.0123456 is 1.23456×10^{-2}. A special syntax is used to write scientific notation numbers. For example, 1.23456×10^2 is written as `1.23456E2` or `1.23456E+2` and 1.23456×10^{-2} as `1.23456E-2`. `E` (or `e`) represents an exponent and can be in either lowercase or uppercase.

why called floating-point?

Note

The `float` and `double` types are used to represent numbers with a decimal point. Why are they called *floating-point numbers*? These numbers are stored in scientific notation internally. When a number such as `50.534` is converted into scientific notation, such as `5.0534E+1`, its decimal point is moved (i.e., floated) to a new position.

Note

To improve readability, Java allows you to use underscores between two digits in a number literal. For example, the following literals are correct.

```
long ssn = 232_45_4519;
long creditCardNumber = 2324_4545_4519_3415L;
```

underscores in numbers

However, `45_` or `_45` is incorrect. The underscore must be placed between two digits.

Check Point

2.18 How many accurate digits are stored in a `float` or `double` type variable?

2.19 Which of the following are correct literals for floating-point numbers?
`12.3, 12.3e+2, 23.4e-2, -334.4, 20.5, 39F, 40D`

2.20 Which of the following are the same as `52.534`?
`5.2534e+1, 0.52534e+2, 525.34e-1, 5.2534e+0`

2.21 Which of the following are correct literals?
`5_2534e+1, _2534, 5_2, 5_`

2.11 Evaluating Expressions and Operator Precedence

Key Point

Java expressions are evaluated in the same way as arithmetic expressions.

Writing a numeric expression in Java involves a straightforward translation of an arithmetic expression using Java operators. For example, the arithmetic expression

$$\frac{3 + 4x}{5} - \frac{10(y - 5)(a + b + c)}{x} + 9\left(\frac{4}{x} + \frac{9 + x}{y}\right)$$

can be translated into a Java expression as:

```
(3 + 4 * x) / 5 – 10 * (y - 5) * (a + b + c) / x +
9 * (4 / x + (9 + x) / y)
```

evaluating an expression

operator precedence rule

Though Java has its own way to evaluate an expression behind the scene, the result of a Java expression and its corresponding arithmetic expression is the same. Therefore, you can safely apply the arithmetic rule for evaluating a Java expression. Operators contained within pairs of parentheses are evaluated first. Parentheses can be nested, in which case the expression in the inner parentheses is evaluated first. When more than one operator is used in an expression, the following operator precedence rule is used to determine the order of evaluation.

- Multiplication, division, and remainder operators are applied first. If an expression contains several multiplication, division, and remainder operators, they are applied from left to right.
- Addition and subtraction operators are applied last. If an expression contains several addition and subtraction operators, they are applied from left to right.

Here is an example of how an expression is evaluated:

```
3 + 4 * 4 + 5 * (4 + 3) - 1
                    (1) inside parentheses first
3 + 4 * 4 + 5 * 7 – 1
                    (2) multiplication
3 + 16 + 5 * 7 – 1
                    (3) multiplication
3 + 16 + 35 – 1
                    (4) addition
19 + 35 – 1
                    (5) addition
    54 – 1
                    (6) subtraction
    53
```

Listing 2.6 gives a program that converts a Fahrenheit degree to Celsius using the formula celsius $= (\frac{5}{9})(\text{fahrenheit} - 32)$.

LISTING 2.6 FahrenheitToCelsius.java

```java
1  import java.util.Scanner;
2
3  public class FahrenheitToCelsius {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6
7      System.out.print("Enter a degree in Fahrenheit: ");
8      double fahrenheit = input.nextDouble();
9
10     // Convert Fahrenheit to Celsius
11     double celsius = (5.0 / 9) * (fahrenheit - 32);
12     System.out.println("Fahrenheit " + fahrenheit + " is " +
13       celsius + " in Celsius");
14   }
15 }
```

divide

```
Enter a degree in Fahrenheit: 100 ↵Enter
Fahrenheit 100.0 is 37.77777777777778 in Celsius
```

line#	fahrenheit	celsius
8	100	
11		37.77777777777778

Be careful when applying division. Division of two integers yields an integer in Java. $\frac{5}{9}$ is translated to `5.0 / 9` instead of `5 / 9` in line 11, because `5 / 9` yields `0` in Java.

integer vs. floating-point division

2.22 How would you write the following arithmetic expression in Java?

Check Point

a. $\dfrac{4}{3(r + 34)} - 9(a + bc) + \dfrac{3 + d(2 + a)}{a + bd}$

b. $5.5 \times (r + 2.5)^{2.5+t}$

2.12 Case Study: Displaying the Current Time

Key Point

You can invoke `System.currentTimeMillis()` *to return the current time.*

VideoNote
Use operators / and %

The problem is to develop a program that displays the current time in GMT (Greenwich Mean Time) in the format hour:minute:second, such as 13:19:8.

currentTimeMillis
UNIX epoch

The `currentTimeMillis` method in the `System` class returns the current time in milliseconds elapsed since midnight, January 1, 1970 GMT, as shown in Figure 2.2. This time is known as the *UNIX epoch*. The epoch is the point when time starts, and `1970` was the year when the UNIX operating system was formally introduced.

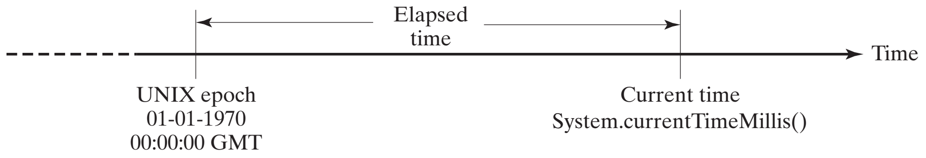


Figure 2.2 The `System.currentTimeMillis()` returns the number of milliseconds since the UNIX epoch.

You can use this method to obtain the current time, and then compute the current second, minute, and hour as follows.

1. Obtain the total milliseconds since midnight, January 1, 1970, in `totalMilliseconds` by invoking `System.currentTimeMillis()` (e.g., `1203183068328` milliseconds).
2. Obtain the total seconds `totalSeconds` by dividing `totalMilliseconds` by `1000` (e.g., `1203183068328` milliseconds / `1000` = `1203183068` seconds).
3. Compute the current second from `totalSeconds % 60` (e.g., `1203183068` seconds % `60` = `8`, which is the current second).
4. Obtain the total minutes `totalMinutes` by dividing `totalSeconds` by `60` (e.g., `1203183068` seconds / `60` = `20053051` minutes).
5. Compute the current minute from `totalMinutes % 60` (e.g., `20053051` minutes % `60` = `31`, which is the current minute).
6. Obtain the total hours `totalHours` by dividing `totalMinutes` by `60` (e.g., `20053051` minutes / `60` = `334217` hours).
7. Compute the current hour from `totalHours % 24` (e.g., `334217` hours % `24` = `17`, which is the current hour).

Listing 2.7 gives the complete program.

Listing 2.7 ShowCurrentTime.java

totalMilliseconds
totalSeconds
currentSecond

```java
public class ShowCurrentTime {
  public static void main(String[] args) {
    // Obtain the total milliseconds since midnight, Jan 1, 1970
    long totalMilliseconds = System.currentTimeMillis();

    // Obtain the total seconds since midnight, Jan 1, 1970
    long totalSeconds = totalMilliseconds / 1000;

    // Compute the current second in the minute in the hour
    long currentSecond = totalSeconds % 60;
```

```
11
12      // Obtain the total minutes
13      long totalMinutes = totalSeconds / 60;
14
15      // Compute the current minute in the hour
16      long currentMinute = totalMinutes % 60;
17
18      // Obtain the total hours
19      long totalHours = totalMinutes / 60;
20
21      // Compute the current hour
22      long currentHour = totalHours % 24;
23
24      // Display results
25      System.out.println("Current time is " + currentHour + ":"
26        + currentMinute + ":" + currentSecond + " GMT");
27    }
28  }
```

totalMinutes

currentMinute

totalHours

currentHour

preparing output

```
Current time is 17:31:8 GMT
```

Line 4 invokes **System.currentTimeMillis()** to obtain the current time in milliseconds as a **long** value. Thus, all the variables are declared as the long type in this program. The seconds, minutes, and hours are extracted from the current time using the **/** and **%** operators (lines 6–22).

variables \ line#	4	7	10	13	16	19	22
totalMilliseconds	1203183068328						
totalSeconds		1203183068					
currentSecond			8				
totalMinutes				20053051			
currentMinute					31		
totalHours						334217	
currentHour							17

In the sample run, a single digit **8** is displayed for the second. The desirable output would be **08**. This can be fixed by using a method that formats a single digit with a prefix **0** (see Exercise 6.37).

2.23 How do you obtain the current second, minute, and hour?

Check Point

2.13 Augmented Assignment Operators

Key Point

The operators `+`, `-`, `*`, `/`, and `%` can be combined with the assignment operator to form augmented operators.

Very often the current value of a variable is used, modified, and then reassigned back to the same variable. For example, the following statement increases the variable **count** by **1**:

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

Java allows you to combine assignment and addition operators using an augmented (or compound) assignment operator. For example, the preceding statement can be written as

```
count += 1;
```

addition assignment operator

The `+=` is called the *addition assignment operator*. Table 2.4 shows other augmented assignment operators.

TABLE 2.4 Augmented Assignment Operators

Operator	*Name*	*Example*	*Equivalent*
`+=`	Addition assignment	`i += 8`	`i = i + 8`
`-=`	Subtraction assignment	`i -= 8`	`i = i - 8`
`*=`	Multiplication assignment	`i *= 8`	`i = i * 8`
`/=`	Division assignment	`i /= 8`	`i = i / 8`
`%=`	Remainder assignment	`i %= 8`	`i = i % 8`

The augmented assignment operator is performed last after all the other operators in the expression are evaluated. For example,

```
x /= 4 + 5.5 * 1.5;
```

is same as

```
x = x / (4 + 5.5 * 1.5);
```

Caution

There are no spaces in the augmented assignment operators. For example, `+ =` should be `+=`.

Note

Like the assignment operator (`=`), the operators (`+=`, `-=`, `*=`, `/=`, `%=`) can be used to form an assignment statement as well as an expression. For example, in the following code, `x += 2` is a statement in the first line and an expression in the second line.

```
x += 2; // Statement
System.out.println(x += 2); // Expression
```

Check Point

2.24 Show the output of the following code:

```
double a = 6.5;
a += a + 1;
```

```
System.out.println(a);
a = 6;
a /= 2;
System.out.println(a);
```

2.14 Increment and Decrement Operators

The increment operator (++) and decrement operator (−−) are for incrementing and decrementing a variable by 1.

Key Point

The ++ and -- are two shorthand operators for incrementing and decrementing a variable by 1. These are handy because that's often how much the value needs to be changed in many programming tasks. For example, the following code increments i by 1 and decrements j by 1.

increment operator (++)
decrement operator (--)

```
int i = 3, j = 3;
i++; // i becomes 4
j--; // j becomes 2
```

i++ is pronounced as i plus plus and i-- as i minus minus. These operators are known as *postfix increment* (or postincrement) and *postfix decrement* (or postdecrement), because the operators ++ and -- are placed after the variable. These operators can also be placed before the variable. For example,

postincrement
postdecrement

```
int i = 3, j = 3;
++i;  // i becomes 4
--j; // j becomes 2
```

++i increments i by 1 and --j decrements j by 1. These operators are known as *prefix increment* (or preincrement) and *prefix decrement* (or predecrement).

preincrement
predecrement

As you see, the effect of i++ and ++i or i-- and --i are the same in the preceding examples. However, their effects are different when they are used in statements that do more than just increment and decrement. Table 2.5 describes their differences and gives examples.

Table 2.5 Increment and Decrement Operators

Operator	*Name*	*Description*	*Example (assume i = 1)*
++var	preincrement	Increment var by 1, and use the new var value in the statement	int j = ++i; // j is 2, i is 2
var++	postincrement	Increment var by 1, but use the original var value in the statement	int j = i++; // j is 1, i is 2
--var	predecrement	Decrement var by 1, and use the new var value in the statement	int j = --i; // j is 0, i is 0
var--	postdecrement	Decrement var by 1, and use the original var value in the statement	int j = i--; // j is 1, i is 0

Here are additional examples to illustrate the differences between the prefix form of ++ (or --) and the postfix form of ++ (or --). Consider the following code:

```
int i = 10;
int newNum = 10 * i++;

System.out.print("i is " + i
  + ", newNum is " + newNum);
```

Same effect as

```
int newNum = 10 * i;
i = i + 1;
```

```
i is 11, newNum is 100
```

In this case, `i` is incremented by `1`, then the *old* value of `i` is used in the multiplication. So `newNum` becomes `100`. If `i++` is replaced by `++i` as follows,

```
int i = 10;
int newNum = 10 * (++i);
System.out.print("i is " + i
  + ", newNum is " + newNum);
```

Same effect as

```
i = i + 1;
int newNum = 10 * i;
```

```
i is 11, newNum is 110
```

`i` is incremented by `1`, and the new value of `i` is used in the multiplication. Thus `newNum` becomes `110`.

Here is another example:

```
double x = 1.0;
double y = 5.0;
double z = x-- + (++y);
```

After all three lines are executed, `y` becomes `6.0`, `z` becomes `7.0`, and `x` becomes `0.0`.

Tip

Using increment and decrement operators makes expressions short, but it also makes them complex and difficult to read. Avoid using these operators in expressions that modify multiple variables or the same variable multiple times, such as this one: `int k = ++i + i`.

Check Point

2.25 Which of these statements are true?

a. Any expression can be used as a statement.

b. The expression `x++` can be used as a statement.

c. The statement `x = x + 5` is also an expression.

d. The statement `x = y = x = 0` is illegal.

2.26 Show the output of the following code:

```
int a = 6;
int b = a++;
System.out.println(a);
System.out.println(b);
a = 6;
b = ++a;
System.out.println(a);
System.out.println(b);
```

2.15 Numeric Type Conversions

Key Point

Floating-point numbers can be converted into integers using explicit casting.

Can you perform binary operations with two operands of different types? Yes. If an integer and a floating-point number are involved in a binary operation, Java automatically converts the integer to a floating-point value. So, `3 * 4.5` is same as `3.0 * 4.5`.

You can always assign a value to a numeric variable whose type supports a larger range of values; thus, for instance, you can assign a long value to a float variable. You cannot, however, assign a value to a variable of a type with a smaller range unless you use *type casting*. *Casting* is an operation that converts a value of one data type into a value of another data type. Casting a type with a small range to a type with a larger range is known as *widening a type*. Casting a type with a large range to a type with a smaller range is known as *narrowing a type*. Java will automatically widen a type, but you must narrow a type explicitly.

casting

widening a type

narrowing a type

The syntax for casting a type is to specify the target type in parentheses, followed by the variable's name or the value to be cast. For example, the following statement

```
System.out.println((int)1.7);
```

displays 1. When a double value is cast into an int value, the fractional part is truncated.

The following statement

```
System.out.println((double)1 / 2);
```

displays 0.5, because 1 is cast to 1.0 first, then 1.0 is divided by 2. However, the statement

```
System.out.println(1 / 2);
```

displays 0, because 1 and 2 are both integers and the resulting value should also be an integer.

Caution

Casting is necessary if you are assigning a value to a variable of a smaller type range, such as assigning a double value to an int variable. A compile error will occur if casting is not used in situations of this kind. However, be careful when using casting, as loss of information might lead to inaccurate results.

possible loss of precision

Note

Casting does not change the variable being cast. For example, d is not changed after casting in the following code:

```
double d = 4.5;
int i = (int)d;  // i becomes 4, but d is still 4.5
```

Note

In Java, an augmented expression of the form x1 op= x2 is implemented as x1 = (T)(x1 op x2), where T is the type for x1. Therefore, the following code is correct.

casting in an augmented expression

```
int sum = 0;
sum += 4.5; // sum becomes 4 after this statement
```

sum += 4.5 is equivalent to sum = (int)(sum + 4.5).

Note

To assign a variable of the int type to a variable of the short or byte type, explicit casting must be used. For example, the following statements have a compile error:

```
int i = 1;
byte b = i; // Error because explicit casting is required
```

However, so long as the integer literal is within the permissible range of the target variable, explicit casting is not needed to assign an integer literal to a variable of the short or byte type (see Section 2.10, Numeric Literals).

The program in Listing 2.8 displays the sales tax with two digits after the decimal point.

LISTING 2.8 SalesTax.java

```
1  import java.util.Scanner;
2
3  public class SalesTax {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6
7      System.out.print("Enter purchase amount: ");
8      double purchaseAmount = input.nextDouble();
9
10     double tax = purchaseAmount * 0.06;
11     System.out.println("Sales tax is $" + (int)(tax * 100) / 100.0);
12   }
13 }
```

casting

```
Enter purchase amount: 197.55 ↵Enter
Sales tax is $11.85
```

line#	purchaseAmount	tax	output
8	197.55		
10		11.853	
11			11.85

formatting numbers

The variable **purchaseAmount** is **197.55** (line 8). The sales tax is **6%** of the purchase, so the **tax** is evaluated as **11.853** (line 10). Note that

tax * 100 is **1185.3**
(int)(tax * 100) is **1185**
(int)(tax * 100) / 100.0 is **11.85**

So, the statement in line 11 displays the tax **11.85** with two digits after the decimal point.

Check Point

2.27 Can different types of numeric values be used together in a computation?

2.28 What does an explicit casting from a **double** to an **int** do with the fractional part of the **double** value? Does casting change the variable being cast?

2.29 Show the following output:

```
float f = 12.5F;
int i = (int)f;
System.out.println("f is " + f);
System.out.println("i is " + i);
```

2.30 If you change **(int)(tax * 100) / 100.0** to **(int)(tax * 100) / 100** in line 11 in Listing 2.8, what will be the output for the input purchase amount of **197.55**?

2.31 Show the output of the following code:

```
double amount = 5;
System.out.println(amount / 2);
System.out.println(5 / 2);
```

2.16 Software Development Process

The software development life cycle is a multistage process that includes requirements specification, analysis, design, implementation, testing, deployment, and maintenance.

Key Point

Developing a software product is an engineering process. Software products, no matter how large or how small, have the same life cycle: requirements specification, analysis, design, implementation, testing, deployment, and maintenance, as shown in Figure 2.3.

VideoNote
Software development process

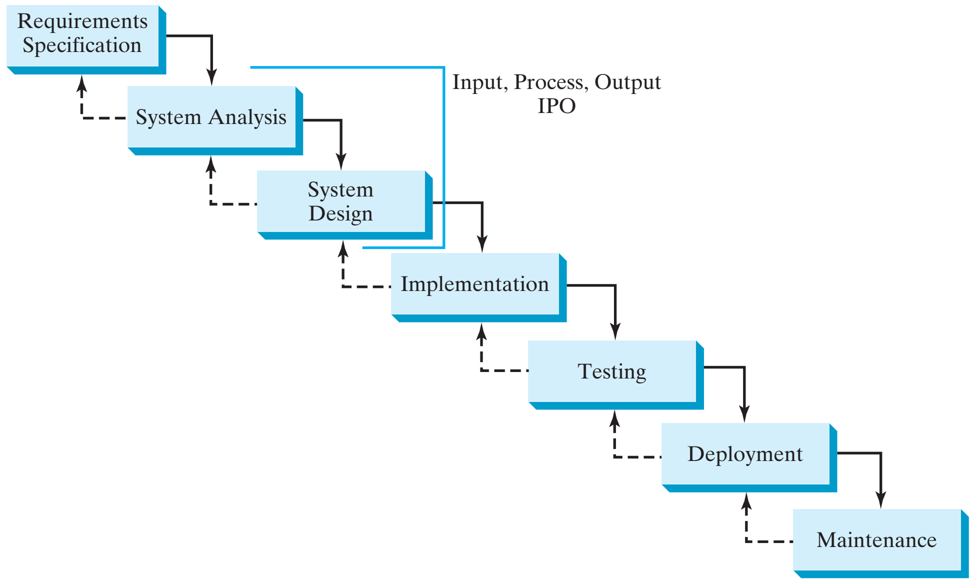


FIGURE 2.3 At any stage of the software development life cycle, it may be necessary to go back to a previous stage to correct errors or deal with other issues that might prevent the software from functioning as expected.

Requirements specification is a formal process that seeks to understand the problem that the software will address and to document in detail what the software system needs to do. This phase involves close interaction between users and developers. Most of the examples in this book are simple, and their requirements are clearly stated. In the real world, however, problems are not always well defined. Developers need to work closely with their customers (the individuals or organizations that will use the software) and study the problem carefully to identify what the software needs to do.

requirements specification

System analysis seeks to analyze the data flow and to identify the system's input and output. When you do analysis, it helps to identify what the output is first, and then figure out what input data you need in order to produce the output.

system analysis

System design is to design a process for obtaining the output from the input. This phase involves the use of many levels of abstraction to break down the problem into manageable components and design strategies for implementing each component. You can view each component as a subsystem that performs a specific function of the system. The essence of system analysis and design is input, process, and output (IPO).

system design

IPO

Implementation involves translating the system design into programs. Separate programs are written for each component and then integrated to work together. This phase requires the use of a programming language such as Java. The implementation involves coding, self-testing, and debugging (that is, finding errors, called *bugs,* in the code).

implementation

testing

Testing ensures that the code meets the requirements specification and weeds out bugs. An independent team of software engineers not involved in the design and implementation of the product usually conducts such testing.

deployment

Deployment makes the software available for use. Depending on the type of software, it may be installed on each user's machine or installed on a server accessible on the Internet.

maintenance

Maintenance is concerned with updating and improving the product. A software product must continue to perform and improve in an ever-evolving environment. This requires periodic upgrades of the product to fix newly discovered bugs and incorporate changes.

VideoNote
Compute loan payments

To see the software development process in action, we will now create a program that computes loan payments. The loan can be a car loan, a student loan, or a home mortgage loan. For an introductory programming course, we focus on requirements specification, analysis, design, implementation, and testing.

Stage 1: Requirements Specification

The program must satisfy the following requirements:

- It must let the user enter the interest rate, the loan amount, and the number of years for which payments will be made.
- It must compute and display the monthly payment and total payment amounts.

Stage 2: System Analysis

The output is the monthly payment and total payment, which can be obtained using the following formulas:

$$monthlyPayment = \frac{loanAmount \times monthlyInterestRate}{1 - \dfrac{1}{(1 + monthlyInterestRate)^{numberOfYears \times 12}}}$$

$$totalPayment = monthlyPayment \times numberOfYears \times 12$$

So, the input needed for the program is the monthly interest rate, the length of the loan in years, and the loan amount.

Note

The requirements specification says that the user must enter the annual interest rate, the loan amount, and the number of years for which payments will be made. During analysis, however, it is possible that you may discover that input is not sufficient or that some values are unnecessary for the output. If this happens, you can go back and modify the requirements specification.

Note

In the real world, you will work with customers from all walks of life. You may develop software for chemists, physicists, engineers, economists, and psychologists, and of course you will not have (or need) complete knowledge of all these fields. Therefore, you don't have to know how formulas are derived, but given the monthly interest rate, the number of years, and the loan amount, you can compute the monthly payment in this program. You will, however, need to communicate with customers and understand how a mathematical model works for the system.

Stage 3: System Design

During system design, you identify the steps in the program.

Step 1. Prompt the user to enter the annual interest rate, the number of years, and the loan amount.

(The interest rate is commonly expressed as a percentage of the principal for a period of one year. This is known as the *annual interest rate*.)

Step 2. The input for the annual interest rate is a number in percent format, such as 4.5%. The program needs to convert it into a decimal by dividing it by `100`. To obtain the monthly interest rate from the annual interest rate, divide it by `12`, since a year has 12 months. So, to obtain the monthly interest rate in decimal format, you need to divide the annual interest rate in percentage by `1200`. For example, if the annual interest rate is 4.5%, then the monthly interest rate is $4.5/1200 = 0.00375$.

Step 3. Compute the monthly payment using the preceding formula.

Step 4. Compute the total payment, which is the monthly payment multiplied by `12` and multiplied by the number of years.

Step 5. Display the monthly payment and total payment.

Stage 4: Implementation

Implementation is also known as *coding* (writing the code). In the formula, you have to compute $(1 + monthlyInterestRate)^{numberOfYears \times 12}$, which can be obtained using `Math.pow(1 + monthlyInterestRate, numberOfYears * 12)`.

`Math.pow(a, b)` method

Listing 2.9 gives the complete program.

LISTING 2.9 ComputeLoan.java

```
import java.util.Scanner;

public class ComputeLoan {
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Enter annual interest rate in percentage, e.g., 7.25%
    System.out.print("Enter annual interest rate, e.g., 7.25%: ");
    double annualInterestRate = input.nextDouble();

    // Obtain monthly interest rate
    double monthlyInterestRate = annualInterestRate / 1200;

    // Enter number of years
    System.out.print(
      "Enter number of years as an integer, e.g., 5: ");
    int numberOfYears = input.nextInt();

    // Enter loan amount
    System.out.print("Enter loan amount, e.g., 120000.95: ");
    double loanAmount = input.nextDouble();

    // Calculate payment
    double monthlyPayment = loanAmount * monthlyInterestRate / (1
      - 1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12));
```

import class

create a `Scanner`

enter interest rate

enter years

enter loan amount

`monthlyPayment`

totalPayment

casting

casting

```
27      double totalPayment = monthlyPayment * numberOfYears * 12;
28
29      // Display results
30      System.out.println("The monthly payment is $" +
31        (int)(monthlyPayment * 100) / 100.0);
32      System.out.println("The total payment is $" +
33        (int)(totalPayment * 100) / 100.0);
34    }
35  }
```

```
Enter annual interest rate, e.g., 5.75%: 5.75 ↵Enter
Enter number of years as an integer, e.g., 5: 15 ↵Enter
Enter loan amount, e.g., 120000.95: 250000 ↵Enter
The monthly payment is $2076.02
The total payment is $373684.53
```

variables \ line#	10	13	18	22	25	27
annualInterestRate	5.75					
monthlyInterestRate		0.0047916666666				
numberOfYears			15			
loanAmount				250000		
monthlyPayment					2076.0252175	
totalPayment						373684.539

Line 10 reads the annual interest rate, which is converted into the monthly interest rate in line 13.

Choose the most appropriate data type for the variable. For example, **numberOfYears** is best declared as an **int** (line 18), although it could be declared as a **long**, **float**, or **double**. Note that **byte** might be the most appropriate for **numberOfYears**. For simplicity, however, the examples in this book will use **int** for integer and **double** for floating-point values.

The formula for computing the monthly payment is translated into Java code in lines 25–27.

Casting is used in lines 31 and 33 to obtain a new **monthlyPayment** and **totalPayment** with two digits after the decimal points.

java.lang package

The program uses the **Scanner** class, imported in line 1. The program also uses the **Math** class, and you might be wondering why that class isn't imported into the program. The **Math** class is in the **java.lang** package, and all classes in the **java.lang** package are implicitly imported. Therefore, you don't need to explicitly import the **Math** class.

Stage 5: Testing

After the program is implemented, test it with some sample input data and verify whether the output is correct. Some of the problems may involve many cases, as you will see in later chapters. For these types of problems, you need to design test data that cover all cases.

incremental code and test

Tip

The system design phase in this example identified several steps. It is a good approach to code and test these steps incrementally by adding them one at a time. This approach makes it much easier to pinpoint problems and debug the program.

2.32 How would you write the following arithmetic expression?

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Check Point

2.17 Case Study: Counting Monetary Units

This section presents a program that breaks a large amount of money into smaller units.

Key Point

Suppose you want to develop a program that changes a given amount of money into smaller monetary units. The program lets the user enter an amount as a `double` value representing a total in dollars and cents, and outputs a report listing the monetary equivalent in the maximum number of dollars, quarters, dimes, nickels, and pennies, in this order, to result in the minimum number of coins.

Here are the steps in developing the program:

1. Prompt the user to enter the amount as a decimal number, such as `11.56`.
2. Convert the amount (e.g., `11.56`) into cents (`1156`).
3. Divide the cents by `100` to find the number of dollars. Obtain the remaining cents using the cents remainder `100`.
4. Divide the remaining cents by `25` to find the number of quarters. Obtain the remaining cents using the remaining cents remainder `25`.
5. Divide the remaining cents by `10` to find the number of dimes. Obtain the remaining cents using the remaining cents remainder `10`.
6. Divide the remaining cents by `5` to find the number of nickels. Obtain the remaining cents using the remaining cents remainder `5`.
7. The remaining cents are the pennies.
8. Display the result.

The complete program is given in Listing 2.10.

LISTING 2.10 ComputeChange.java

```java
import java.util.Scanner;

public class ComputeChange {
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Receive the amount
    System.out.print(
      "Enter an amount in double, for example 11.56: ");
    double amount = input.nextDouble();

    int remainingAmount = (int)(amount * 100);

    // Find the number of one dollars
    int numberOfOneDollars = remainingAmount / 100;
    remainingAmount = remainingAmount % 100;

    // Find the number of quarters in the remaining amount
    int numberOfQuarters = remainingAmount / 25;
```

import class

enter input

dollars

quarters

dimes
nickels
pennies
output

```
21         remainingAmount = remainingAmount % 25;
22
23         // Find the number of dimes in the remaining amount
24         int numberOfDimes = remainingAmount / 10;
25         remainingAmount = remainingAmount % 10;
26
27         // Find the number of nickels in the remaining amount
28         int numberOfNickels = remainingAmount / 5;
29         remainingAmount = remainingAmount % 5;
30
31         // Find the number of pennies in the remaining amount
32         int numberOfPennies = remainingAmount;
33
34         // Display results
35         System.out.println("Your amount " + amount + " consists of");
36         System.out.println("    " + numberOfOneDollars + " dollars");
37         System.out.println("    " + numberOfQuarters + " quarters ");
38         System.out.println("    " + numberOfDimes + " dimes");
39         System.out.println("    " + numberOfNickels + " nickels");
40         System.out.println("    " + numberOfPennies + " pennies");
41     }
42   }
```

```
Enter an amount, for example, 11.56: 11.56 ↵Enter
Your amount 11.56 consists of
      11 dollars
      2 quarters
      0 dimes
      1 nickels
      1 pennies
```

line# / variables	11	13	16	17	20	21	24	25	28	29	32
amount	11.56										
remainingAmount		1156		56		6		6		1	
numberOfOneDollars			11								
numberOfQuarters					2						
numberOfDimes							0				
numberOfNickels									1		
numberOfPennies											1

The variable **amount** stores the amount entered from the console (line 11). This variable is not changed, because the amount has to be used at the end of the program to display the results. The program introduces the variable **remainingAmount** (line 13) to store the changing remaining amount.

The variable **amount** is a **double** decimal representing dollars and cents. It is converted to an **int** variable **remainingAmount**, which represents all the cents. For instance, if **amount** is 11.56, then the initial remainingAmount is 1156. The division operator yields the integer part of the division, so 1156 / 100 is 11. The remainder operator obtains the remainder of the division, so 1156 % 100 is 56.

The program extracts the maximum number of singles from the remaining amount and obtains a new remaining amount in the variable remainingAmount (lines 16–17). It then extracts the maximum number of quarters from remainingAmount and obtains a new remainingAmount (lines 20–21). Continuing the same process, the program finds the maximum number of dimes, nickels, and pennies in the remaining amount.

loss of precision

One serious problem with this example is the possible loss of precision when casting a double amount to an int remainingAmount. This could lead to an inaccurate result. If you try to enter the amount 10.03, 10.03 * 100 becomes 1002.9999999999999. You will find that the program displays 10 dollars and 2 pennies. To fix the problem, enter the amount as an integer value representing cents (see Programming Exercise 2.22).

Check Point

2.33 Show the output with the input value 1.99.

2.18 Common Errors and Pitfalls

Key Point

Common elementary programming errors often involve undeclared variables, uninitialized variables, integer overflow, unintended integer division, and round-off errors.

Common Error 1: Undeclared/Uninitialized Variables and Unused Variables

A variable must be declared with a type and assigned a value before using it. A common error is not declaring a variable or initializing a variable. Consider the following code:

```
double interestRate = 0.05;
double interest = interestrate * 45;
```

This code is wrong, because interestRate is assigned a value 0.05; but interestrate has not been declared and initialized. Java is case sensitive, so it considers interestRate and interestrate to be two different variables.

If a variable is declared, but not used in the program, it might be a potential programming error. So, you should remove the unused variable from your program. For example, in the following code, taxRate is never used. It should be removed from the code.

```
double interestRate = 0.05;
double taxRate = 0.05;
double interest = interestRate * 45;
System.out.println("Interest is " + interest);
```

If you use an IDE such as Eclipse and NetBeans, you will receive a warning on unused variables.

Common Error 2: Integer Overflow

what is overflow?

Numbers are stored with a limited numbers of digits. When a variable is assigned a value that is too large (*in size*) to be stored, it causes *overflow*. For example, executing the following statement causes overflow, because the largest value that can be stored in a variable of the int type is 2147483647. 2147483648 will be too large for an int value.

```
int value = 2147483647 + 1;
// value will actually be -2147483648
```

Likewise, executing the following statement causes overflow, because the smallest value that can be stored in a variable of the int type is -2147483648. -2147483649 is too large in size to be stored in an int variable.

```
int value = -2147483648 - 1;
// value will actually be 2147483647
```

Java does not report warnings or errors on overflow, so be careful when working with numbers close to the maximum or minimum range of a given type.

what is underflow?

When a floating-point number is too small (i.e., too close to zero) to be stored, it causes *underflow*. Java approximates it to zero, so normally you don't need to be concerned about underflow.

Common Error 3: Round-off Errors

floating-point approximation

A *round-off error*, also called a *rounding error*, is the difference between the calculated approximation of a number and its exact mathematical value. For example, 1/3 is approximately 0.333 if you keep three decimal places, and is 0.3333333 if you keep seven decimal places. Since the number of digits that can be stored in a variable is limited, round-off errors are inevitable. Calculations involving floating-point numbers are approximated because these numbers are not stored with complete accuracy. For example,

```
System.out.println(1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1);
```

displays `0.5000000000000001`, not `0.5`, and

```
System.out.println(1.0 - 0.9);
```

displays `0.09999999999999998`, not `0.1`. Integers are stored precisely. Therefore, calculations with integers yield a precise integer result.

Common Error 4: Unintended Integer Division

Java uses the same divide operator, namely `/`, to perform both integer and floating-point division. When two operands are integers, the `/` operator performs an integer division. The result of the operation is an integer. The fractional part is truncated. To force two integers to perform a floating-point division, make one of the integers into a floating-point number. For example, the code in (a) displays that average is `1` and the code in (b) displays that average is `1.5`.

```
int number1 = 1;
int number2 = 2;
double average = (number1 + number2) / 2;
System.out.println(average);
```

(a)

```
int number1 = 1;
int number2 = 2;
double average = (number1 + number2) / 2.0;
System.out.println(average);
```

(b)

Common Pitfall 1: Redundant Input Objects

New programmers often write the code to create multiple input objects for each input. For example, the following code reads an integer and a double value.

```
Scanner input = new Scanner(System.in);
System.out.print("Enter an integer: ");
int v1 = input.nextInt();

Scanner input1 = new Scanner(System.in);
System.out.print("Enter a double value: ");
double v2 = input1.nextDouble();
```

BAD CODE