CHAPTER 2

ELEMENTARY PROGRAMMING

Objectives

- To write Java programs to perform simple calculations (§2.2).
- To obtain input from the console using the `Scanner` class (§2.3).
- To use identifiers to name variables, constants, methods, and classes (§2.4).
- To use variables to store data (§§2.5–2.6).
- To program with assignment statements and assignment expressions (§2.6).
- To use constants to store permanent data (§2.7).
- To declare Java primitive data types: `byte`, `short`, `int`, `long`, `float`, `double`, and `char` (§2.8.1).
- To use Java operators to write numeric expressions (§§2.8.2–2.8.3).
- To display the current time (§2.9).
- To use shorthand operators (§2.10).
- To cast the value of one type to another type (§2.11).
- To compute loan payments (§2.12).
- To represent characters using the `char` type (§2.13).
- To compute monetary changes (§2.14).
- To represent a string using the `String` type (§2.15).
- To become familiar with Java documentation, programming style, and naming conventions (§2.16).
- To distinguish syntax errors, runtime errors, and logic errors and debug errors (§2.17).
- (GUI) To obtain input using the `JOptionPane` input dialog boxes (§2.18).

2.1 Introduction

In Chapter 1 you learned how to create, compile, and run a Java program. Now you will learn how to solve practical problems programmatically. Through these problems, you will learn elementary programming using primitive data types, variables, constants, operators, expressions, and input and output.

2.2 Writing Simple Programs

problem

To begin, let's look at a simple problem for computing the area of a circle. How do we write a program for solving this problem?

algorithm

Writing a program involves designing algorithms and translating algorithms into code. An *algorithm* describes how a problem is solved in terms of the actions to be executed 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

pseudocode (i.e., natural language mixed with programming code). The algorithm for this program can be described as follows:

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

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

3. Display the area.

Many of the problems you will encounter when taking an introductory course in programming can be described with simple, straightforward algorithms.

When you *code*, you translate an algorithm into a program. You already know that every Java program begins with a class declaration 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. So the program is 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.

variable

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 designates a location in memory for storing data and computational results in the program. A variable has a name that can be used to access the memory location.

descriptive names

floating-point number

primitive data types

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. Java provides simple data types for representing integers, floating-point numbers (i.e., numbers with a decimal point), characters, and Boolean types. These types are known as *primitive data types* or *fundamental types*.

Declare `radius` and `area` as double-precision floating-point numbers. The program can be expanded as follows:

```
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 double-precision floating-point values stored in the computer.

The first step is to read in `radius`. Reading a number from the keyboard is not a simple matter. For the time being, let us assign a fixed value to `radius` in the program.

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

In the final step, display `area` on the console by using the `System.out.println` method.

The complete program is shown in Listing 2.1. A sample run of the program is shown in Figure 2.1.

FIGURE 2.1 The program displays the area of a circle.

LISTING 2.1 ComputeArea.java

```
1 public class ComputeArea {
2   public static void main(String[] args) {
3     double radius; // Declare radius
4     double area; // Declare area
5
6     // Assign a radius
7     radius = 20; // New value is radius
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 }
```

declaring variable

assign value

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 `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. Hand trace is helpful to understand how a program works, and it is also a useful tool for finding errors in the program.

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

concatenating strings

concatenating strings with numbers

The plus sign (+) has two meanings: one for addition and the other for concatenating strings. The plus sign (+) in lines 13–14 is called a *string concatenation operator*. It combines two strings if two operands are strings. If one of the operands is a nonstring (e.g., a number), the nonstring value is converted into a string and concatenated with the other string. So 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 §2.15, "The `String` Type."

Caution

breaking a long string

A string constant 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");
```

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");
```

Tip

incremental development and testing

This example consists of three steps. It is a good approach to develop and test these steps incrementally by adding them one at a time.

2.3 Reading Input from the Console

Video Note
Obtain input

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. You can use the `Scanner` class for console 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 methods in Table 2.1 to read various types of input.

TABLE 2.1 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.
`next()`	reads a string that ends before a whitespace character.
`nextLine()`	reads a line of text (i.e., a string ending with the *Enter* key pressed).

For now, we will see how to read a number that includes a decimal point by invoking the `nextDouble()` method. Other methods will be covered when they are used. Listing 2.2 rewrites Listing 2.1 to prompt the user to enter a radius.

LISTING 2.2 ComputeAreaWithConsoleInput.java

```
1  import java.util.Scanner; // Scanner is in the java.util package
2
3  public class ComputeAreaWithConsoleInput {
4    public static void main(String[] args) {
5      // Create a Scanner object
6      Scanner input = new Scanner(System.in);
7
8      // Prompt the user to enter a radius
9      System.out.print("Enter a number for radius: ");
10     double radius = input.nextDouble() ;
11
12     // Compute area
13     double area = radius * radius * 3.14159;
14
15     // Display result
16     System.out.println("The area for the circle of radius " +
17       radius + " is " + area);
18   }
19 }
```

import class

create a **Scanner**

read a **double**

```
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
```

The `Scanner` class is in the `java.util` package. It is imported in line 1. Line 6 creates a `Scanner` object.

The statement in line 9 displays a message to prompt the user for input.

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

`print` vs. `println`

The `print` method is identical to the `println` method except that `println` moves the cursor to the next line after displaying the string, but `print` does not advance the cursor to the next line when completed.

The statement in line 10 reads an input from the keyboard.

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

After the user enters a number and presses the *Enter* key, the number is read and assigned to `radius`.

More details on objects will be introduced in Chapter 8, "Objects and Classes." For the time being, simply accept that this is how to obtain input from the console.

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

LISTING 2.3 ComputeAverage.java

import class

create a **`Scanner`**

read a **`double`**

```
1  import java.util.Scanner; // Scanner is in the java.util package
2
3  public class ComputeAverage {
4    public static void main(String[] args) {
5      // Create a Scanner object
6      Scanner input = new Scanner(System.in);
7
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 result
18     System.out.println("The average of " + number1 + " " + number2
19       + " " + number3 + " is " + average);
20   }
21 }
```

enter input in one line

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

enter input in multiple lines

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

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.

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.

2.4 Identifiers

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. Such names are called *identifiers*. All identifiers must obey the following rules:

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, "Java Keywords," 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 `showMessageDialog` 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.

Note

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

case sensitive

Tip

Identifiers are for naming variables, constants, methods, classes, and packages. Descriptive identifiers make programs easy to read.

descriptive names

Tip

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

the $ character

2.5 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 double-precision, floating-point type. You can assign any numerical value to `radius` and `area`, and the values of `radius` and `area` can be reassigned. For example, you can write the code shown below to compute the area for different radii:

why called variables?

```
// Compute the first area
radius = 1.0;
area = radius * radius * 3.14159;
System.out.println("The area is " + area + " for radius " + radius);

// Compute the second area
radius = 2.0;
area = radius * radius * 3.14159;
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;
```

declaring variables

Here are some examples of variable declarations:

```
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;
```

The examples use the data types `int`, `double`, and `char`. 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
```

Note

naming variables

By convention, variable names are in lowercase. If a name consists of several words, concatenate all of them and capitalize the first letter of each word except the first. Examples of variables are `radius` and `interestRate`.

initializing 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;
x = 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.

2.6 Assignment Statements and Assignment Expressions

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;
```

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

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

A variable can also be used in an expression. 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, the variable name must be on the left of the assignment operator. Thus, `1 = x` would be 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`.

In Java, an assignment statement is essentially an expression that evaluates to the value to be assigned to the variable on the left-hand 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: assignment expression

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

which is equivalent to

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

The following statement is also correct:

```
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 §2.11 "Numeric Type Conversions."

2.7 Named Constants

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: constant

```
final datatype CONSTANTNAME = VALUE;
```

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 follows:

```
// ComputeArea.java: Compute the area of a circle
public class ComputeArea {
  public static void main(String[] args) {
    final double PI = 3.14159; // Declare a constant

    // Assign a radius
    double radius = 20;

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

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

naming constants

Caution

By convention, constants are named in uppercase: `PI`, not `pi` or `Pi`.

benefits of constants

Note

There are three benefits of using constants: (1) you don't have to repeatedly type the same value; (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; (3) a descriptive name for a constant makes the program easy to read.

2.8 Numeric Data Types and Operations

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.

Table 2.2 lists the six numeric data types, their ranges, and their storage sizes.

TABLE 2.2 Numeric Data Types

Name	*Range*	*Storage Size*
`byte`	-2^7 (-128) to 2^7-1 (127)	8-bit signed
`short`	-2^{15} (-32768) to $2^{15}-1$ (32767)	16-bit signed
`int`	-2^{31} (-2147483648) to $2^{31}-1$ (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

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 has adopted 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 values as given in Appendix E, "Special Floating-Point Values."

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 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

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*, `float` as *single precision*. Normally you should use the `double` type, because it is more accurate than the `float` type.

Caution

what is overflow?

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` is too large.

```
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 should not be concerned with underflow.

2.8.1 Numeric Operators

operators +, -, *, /, %

The operators for numeric data types include the standard arithmetic operators: addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and remainder (`%`), as shown in Table 2.3.

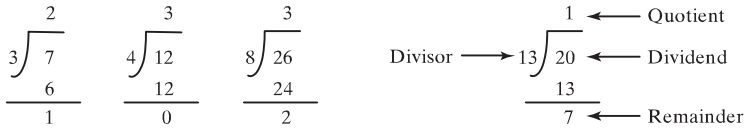
integer division

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

The `%` operator yields the remainder after division. The left-hand operand is the dividend and the right-hand operand the divisor. Therefore, `7 % 3` yields `1`, `12 % 4` yields `0`, `26 % 8` yields `2`, and `20 % 13` yields `7`.

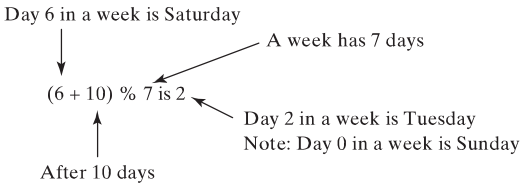
TABLE 2.3 Numeric Operators

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



The `%` operator is often used for positive integers but can be used also 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`. So 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:



Listing 2.4 gives a program that obtains minutes and remaining seconds from an amount of time in seconds. For example, `500` seconds contains `8` minutes and `20` seconds.

LISTING 2.4 DisplayTime.java

import **Scanner**
create a **Scanner**
read an integer
divide
remainder

```
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 }
```

```
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 4 obtains the minutes using `seconds / 60`. Line 5 (`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

Note

Calculations involving floating-point numbers are approximated because these numbers are not stored with complete accuracy. For example,

floating-point approximation

```
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.

2.8.2 Numeric Literals

A *literal* is a constant value that appears directly in a program. For example, `34` and `0.305` are literals in the following statements:

literal

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

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 (e.g., `2147483648L`). `L` is preferred because `l` (lowercase L) can easily be confused with 1 (the digit one). To write integer `2147483648` in a Java program, you have to write it as `2147483648L`, because `2147483648` exceeds the range for the `int` value.

long literal

Note

By default, an integer literal is a decimal integer number. 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, the following code displays the decimal value `65535` for hexadecimal number FFFF.

octal and hex literals

```
System.out.println(0xFFFF);
```

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

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 d or D
suffix f or F

Note

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

double vs. **float**

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

displays `1.0 / 3.0 is 0.3333333333333333`.

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

displays `1.0F / 3.0F is 0.33333334`.

Scientific Notation

Floating-point literals can also be specified in scientific notation; for example, `1.23456e+2`, the same as `1.23456e2`, is equivalent to $1.23456 \times 10^2 = 123.456$, and `1.23456e-2` is equivalent to $1.23456 \times 10^{-2} = 0.0123456$. `E` (or `e`) represents an exponent and can be in either lowercase or uppercase.

Note

why called floating-point?

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. 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.

2.8.3 Evaluating Java 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

Though Java has its own way to evaluate an expression behind the scene, the result of a Java expression and its corresponding arithmetic expression are 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. Multiplication, division, and remainder operators are applied next. 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 2.5 gives a program that converts a Fahrenheit degree to Celsius using the formula $celsius = \left(\frac{5}{9}\right)(fahrenheit - 32)$.

LISTING 2.5 FahrenheitToCelsius.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. decimal division

2.9 Problem: Displaying the Current Time

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.

Video Note
Use operators / and %

The `currentTimeMillis` method in the `System` class returns the current time in milliseconds elapsed since the time `00:00:00` on January 1, 1970 GMT, as shown in Figure 2.2. This time is known as the *Unix epoch,* because `1970` was the year when the Unix operating system was formally introduced.

currentTimeMillis

Unix epoch

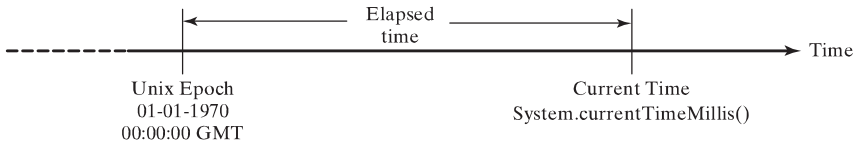


FIGURE 2.2 The `System.currentTimeMillis()` returns the number of milliseconds since the Unix epoch.

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, Jan 1, 1970, in `totalMilliseconds` by invoking `System.currentTimeMillis()` (e.g., `1203183086328` milliseconds).
2. Obtain the total seconds `totalSeconds` by dividing `totalMilliseconds` by `1000` (e.g., `1203183086328` milliseconds / `1000` = `1203183086` seconds).
3. Compute the current second from `totalSeconds % 60` (e.g., `1203183086` seconds % `60` = `26`, which is the current second).
4. Obtain the total minutes `totalMinutes` by dividing `totalSeconds` by `60` (e.g., `1203183086` 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 2.6 gives the complete program.

LISTING 2.6 ShowCurrentTime.java

```
1  public class ShowCurrentTime {
2    public static void main(String[] args) {
3      // Obtain the total milliseconds since midnight, Jan 1, 1970
4      long totalMilliseconds = System.currentTimeMillis();
5
6      // Obtain the total seconds since midnight, Jan 1, 1970
7      long totalSeconds = totalMilliseconds / 1000;
8
9      // Compute the current second in the minute in the hour
10     long currentSecond = (int)(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 }
```

`totalMilliseconds` `totalSeconds` `currentSecond` `totalMinutes` `currentMinute` `totalHours` `currentHour`

```
Current time is 17:31:26 GMT
```

variables \ line#	4	7	10	13	16	19	22
totalMilliseconds	1203183086328						
totalSeconds		1203183086					
currentSecond			26				
totalMinutes				20053051			
currentMinute					31		
totalHours						334217	
currentHour							17

When `System.currentTimeMillis()` (line 4) is invoked, it returns the difference, measured in milliseconds, between the current GMT and midnight, January 1, 1970 GMT. This method returns the milliseconds as a `long` value. So, all the variables are declared as the `long` type in this program.

2.10 Shorthand 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 adds the current value of `i` with `8` and assigns the result back to `i`:

```
i = i + 8;
```

Java allows you to combine assignment and addition operators using a shorthand operator. For example, the preceding statement can be written as:

```
i += 8;
```

The `+=` is called the *addition assignment operator*. Other shorthand operators are shown in Table 2.4.

addition assignment operator

TABLE 2.4 Shorthand 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`

Caution

There are no spaces in the shorthand 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
```

There are two more 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. The two operators are `++` and `--`. For example, the following code increments `i` by `1` and decrements `j` by `1`.

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

The `++` and `--` operators can be used in prefix or suffix mode, as shown in Table 2.5.

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	`int j = ++i; // j is 2, // i is 2`
`var++`	postincrement	Increment `var` by `1`, but use the original `var` value	`int j = i++; // j is 1, // i is 2`
`--var`	predecrement	Decrement `var` by `1` and use the new `var` value	`int j = --i; // j is 0, // i is 0`
`var--`	postdecrement	Decrement `var` by `1` and use the original `var` value	`int j = ++i; // j is 1, // i is 0`

preincrement, predecrement
postincrement, postdecrement

If the operator is *before* (prefixed to) the variable, the variable is incremented or decremented by `1`, then the *new* value of the variable is returned. If the operator is *after* (suffixed to) the variable, then the variable is incremented or decremented by `1`, but the original *old* value of the variable is returned. Therefore, the prefixes `++x` and `--x` are referred to, respectively, as the *preincrement operator* and the *predecrement operator*; and the suffixes `x++` and `x--` are referred to, respectively, as the *postincrement operator* and the *postdecrement operator*. The prefix form of `++` (or `--`) and the suffix form of `++` (or `--`) are the same if they are used in isolation, but they cause different effects when used in an expression. The following code illustrates this:

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

Same effect as

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

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

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

Same effect as

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

`i` is incremented by `1`, and the new value of `i` is returned and 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`.

The increment operator `++` and the decrement operator `--` can be applied to all integer and floating-point types. These operators are often used in loop statements. A *loop statement* is a construct that controls how many times an operation or a sequence of operations is performed in succession. This construct, and the topic of loop statements, are introduced in Chapter 4, "Loops."

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`.

2.11 Numeric Type Conversions

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 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 variable of a type with a small range to a variable of a type with a larger range is known as *widening a type*. Casting a variable of a type with a large range to a variable of a type with a smaller range is known as *narrowing a type*. Widening a type can be performed automatically without explicit casting. Narrowing a type must be performed explicitly.

widening a type

narrowing a type

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

type casting

```
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. Be careful when using casting. 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 not changed, still 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. Please refer to §2.8.2, "Numeric Literals."

Listing 2.7 gives a program that displays the sales tax with two digits after the decimal point.

LISTING 2.7 SalesTax.java

casting

```
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 }
```

```
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

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.

2.12 Problem: Computing Loan Payments

Video Note
Program computations

The problem is to write a program that computes loan payments. The loan can be a car loan, a student loan, or a home mortgage loan. The program lets the user enter the interest rate, number of years, and loan amount, and displays the monthly and total payments.

The formula to compute the monthly payment is as follows:

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

You don't have to know how this formula is derived. Nonetheless, given the monthly interest rate, number of years, and loan amount, you can use it to compute the monthly payment.

In the formula, you have to compute $(1 + monthlyInterestRate)^{numberOfYears \times 12}$. The `pow(a, b)` method in the `Math` class can be used to compute a^b. The `Math` class, which comes with the Java API, is available to all Java programs. For example,

`pow(a, b)` method

```
System.out.println(Math.pow(2, 3)); // Display 8
System.out.println(Math.pow(4, 0.5)); // Display 4
```

$(1 + monthlyInterestRate)^{numberOfYears \times 12}$ can be computed using `Math.pow(1 + monthlyInterestRate, numberOfYears * 12)`.

Here are the steps in developing the program:

1. Prompt the user to enter the annual interest rate, number of years, and loan amount.
2. Obtain the monthly interest rate from the annual interest rate.
3. Compute the monthly payment using the preceding formula.
4. Compute the total payment, which is the monthly payment multiplied by `12` and multiplied by the number of years.
5. Display the monthly payment and total payment.

Listing 2.8 gives the complete program.

LISTING 2.8 ComputeLoan.java

```
1  import java.util.Scanner;                                          import class
2
3  public class ComputeLoan {
4    public static void main(String[] args) {
5      // Create a Scanner
6      Scanner input = new Scanner(System.in);                        create a Scanner
7
8      // Enter yearly interest rate
9      System.out.print("Enter yearly interest rate, for example 8.25: ");
10     double annualInterestRate = input.nextDouble();                enter interest rate
11
12     // Obtain monthly interest rate
13     double monthlyInterestRate = annualInterestRate / 1200;
14
15     // Enter number of years
16     System.out.print(
17       "Enter number of years as an integer, for example 5: ");
18     int numberOfYears = input.nextInt();                           enter years
19
20     // Enter loan amount
21     System.out.print("Enter loan amount, for example 120000.95: ");
```

enter loan amount

```
22      double loanAmount = input.nextDouble();
23
24      // Calculate payment
25      double monthlyPayment  = loanAmount * monthlyInterestRate / (1
26        - 1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12));
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 }
```

monthlyPayment

totalPayment

casting

casting

```
Enter yearly interest rate, for example 8.25: 5.75 Enter
Enter number of years as an integer, for example 5: 15 Enter
Enter loan amount, for example 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 yearly interest rate, which is converted into monthly interest rate in line 13. If you entered an input other than a numeric value, a runtime error would occur.

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 point.

java.lang package

The program uses the **Scanner** class, imported in line 1. The program also uses the **Math** class; why isn't it imported? The **Math** class is in the **java.lang** package. All classes in the **java.lang** package are implicitly imported. So, there is no need to explicitly import the **Math** class.

2.13 Character Data Type and Operations

char type

The character data type, **char**, is used to represent a single character. A character literal is enclosed in single quotation marks. Consider the following code:

```
char letter = 'A';
char numChar = '4';
```

The first statement assigns character `A` to the `char` variable `letter`. The second statement assigns digit character `4` to the `char` variable `numChar`.

Caution

A string literal must be enclosed in quotation marks. A character literal is a single character enclosed in single quotation marks. So `"A"` is a string, and `'A'` is a character.

char literal

2.13.1 Unicode and ASCII code

character encoding

Computers use binary numbers internally. A character is stored in a computer as a sequence of 0s and 1s. Mapping a character to its binary representation is called *encoding*. There are different ways to encode a character. How characters are encoded is defined by an *encoding scheme*.

Unicode

Java supports *Unicode*, an encoding scheme established by the Unicode Consortium to support the interchange, processing, and display of written texts in the world's diverse languages. Unicode was originally designed as a 16-bit character encoding. The primitive data type `char` was intended to take advantage of this design by providing a simple data type that could hold any character. However, it turned out that the `65,536` characters possible in a 16-bit encoding are not sufficient to represent all the characters in the world. The Unicode standard therefore has been extended to allow up to `1,112,064` characters. Those characters that go beyond the original 16-bit limit are called *supplementary characters*. Java supports supplementary characters. The processing and representing of supplementary characters are beyond the scope of this book. For simplicity, this book considers only the original 16-bit Unicode characters. These characters can be stored in a `char` type variable.

original Unicode

supplementary Unicode

A 16-bit Unicode takes two bytes, preceded by `\u`, expressed in four hexadecimal digits that run from `'\u0000'` to `'\uFFFF'`. For example, the word "welcome" is translated into Chinese using two characters, 欢迎 . The Unicodes of these two characters are "`\u6B22\u8FCE`".

Listing 2.9 gives a program that displays two Chinese characters and three Greek letters.

LISTING 2.9 DisplayUnicode.java

```
1 import javax.swing.JOptionPane;
2
3 public class DisplayUnicode {
4   public static void main(String[] args) {
5     JOptionPane.showMessageDialog(null,
6       "\u6B22\u8FCE \u03b1 \u03b2 \u03b3",
7       "\u6B22\u8FCE Welcome",
8       JOptionPane.INFORMATION_MESSAGE);
9   }
10 }
```

If no Chinese font is installed on your system, you will not be able to see the Chinese characters. The Unicodes for the Greek letters α β γ are `\u03b1` `\u03b2` `\u03b3`.

ASCII

Most computers use *ASCII* (*American Standard Code for Information Interchange*), a 7-bit encoding scheme for representing all uppercase and lowercase letters, digits, punctuation marks, and control characters. Unicode includes ASCII code, with `'\u0000'` to `'\u007F'` corresponding to the 128 ASCII characters. (See Appendix B, "The ASCII Character Set," for a list of ASCII characters and their decimal and hexadecimal codes.) You can use ASCII characters such as `'X'`, `'1'`, and `'$'` in a Java program as well as Unicodes. Thus, for example, the following statements are equivalent:

```
char letter = 'A';
char letter = '\u0041'; // Character A's Unicode is 0041
```

Both statements assign character `A` to `char` variable `letter`.

char increment and decrement

Note

The increment and decrement operators can also be used on **char** variables to get the next or preceding Unicode character. For example, the following statements display character **b**.

```
char ch = 'a';
System.out.println(++ch);
```

2.13.2 Escape Sequences for Special Characters

Suppose you want to print a message with quotation marks in the output. Can you write a statement like this?

```
System.out.println("He said "Java is fun"");
```

No, this statement has a syntax error. The compiler thinks the second quotation character is the end of the string and does not know what to do with the rest of characters.

backslash

To overcome this problem, Java defines escape sequences to represent special characters, as shown in Table 2.6. An escape sequence begins with the backslash character (\) followed by a character that has a special meaning to the compiler.

TABLE 2.6 Java Escape Sequences

Character Escape Sequence	*Name*	*Unicode Code*
\b	Backspace	\u0008
\t	Tab	\u0009
\n	Linefeed	\u000A
\f	Formfeed	\u000C
\r	Carriage Return	\u000D
\\	Backslash	\u005C
\'	Single Quote	\u0027
\"	Double Quote	\u0022

So, now you can print the quoted message using the following statement:

```
System.out.println("He said \"Java is fun\"");
```

The output is

```
He said "Java is fun"
```

2.13.3 Casting between char and Numeric Types

A **char** can be cast into any numeric type, and vice versa. When an integer is cast into a **char**, only its lower 16 bits of data are used; the other part is ignored. For example:

```
char ch = (char)0XAB0041; // the lower 16 bits hex code 0041 is
                          // assigned to ch
System.out.println(ch);   // ch is character A
```

When a floating-point value is cast into a **char**, the floating-point value is first cast into an **int**, which is then cast into a **char**.

```
char ch = (char)65.25;    // decimal 65 is assigned to ch
System.out.println(ch);   // ch is character A
```

When a **char** is cast into a numeric type, the character's Unicode is cast into the specified numeric type.

```
int i = (int)'A'; // the Unicode of character A is assigned to i
System.out.println(i);  // i is 65
```

Implicit casting can be used if the result of a casting fits into the target variable. Otherwise, explicit casting must be used. For example, since the Unicode of **'a'** is **97**, which is within the range of a byte, these implicit castings are fine:

```
byte b = 'a';
int i = 'a';
```

But the following casting is incorrect, because the Unicode **\uFFF4** cannot fit into a byte:

```
byte b = '\uFFF4';
```

To force assignment, use explicit casting, as follows:

```
byte b = (byte)'\uFFF4';
```

Any positive integer between **0** and **FFFF** in hexadecimal can be cast into a character implicitly. Any number not in this range must be cast into a **char** explicitly.

Note

All numeric operators can be applied to **char** operands. A **char** operand is automatically cast into a number if the other operand is a number or a character. If the other operand is a string, the character is concatenated with the string. For example, the following statements

numeric operators on characters

```
int i = '2' + '3'; // (int)'2' is 50 and (int)'3' is 51
System.out.println("i is " + i); // i is 101

int j = 2 + 'a'; // (int)'a' is 97
System.out.println("j is " + j); // j is 99
System.out.println(j + " is the Unicode for character "
  + (char)j);

System.out.println("Chapter " + '2');
```

display

```
i is 101
j is 99
99 is the Unicode for character c
Chapter 2
```

Note

The Unicodes for lowercase letters are consecutive integers starting from the Unicode for **'a'**, then for **'b'**, **'c'**, ..., and **'z'**. The same is true for the uppercase letters. Furthermore, the Unicode for **'a'** is greater than the Unicode for **'A'**. So **'a' - 'A'** is the same as **'b' - 'B'**. For a lowercase letter *ch*, its corresponding uppercase letter is **(char)('A' + (ch - 'a'))**.

2.14 Problem: Counting Monetary Units

Suppose you want to develop a program that classifies 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 dollars, quarters, dimes, nickels, and pennies, as shown in the sample run.

Your program should report the maximum number of dollars, then the maximum number of quarters, and so on, in this order.

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

import class

```
1  import java.util.Scanner;
2
3  public class ComputeChange {
4    public static void main(String[] args) {
5      // Create a Scanner
6      Scanner input = new Scanner(System.in);
7
8      // Receive the amount
9      System.out.print(
10       "Enter an amount in double, for example 11.56: ");
```

enter input

```
11     double amount = input.nextDouble();
12
13     int remainingAmount = (int)(amount * 100);
14
15     // Find the number of one dollars
```

dollars

```
16     int numberOfOneDollars = remainingAmount / 100;
17     remainingAmount = remainingAmount % 100;
18
19     // Find the number of quarters in the remaining amount
```

quarters

```
20     int numberOfQuarters = remainingAmount / 25;
21     remainingAmount = remainingAmount % 25;
22
23     // Find the number of dimes in the remaining amount
```

dimes

```
24     int numberOfDimes = remainingAmount / 10;
25     remainingAmount = remainingAmount % 10;
26
27     // Find the number of nickels in the remaining amount
```

nickels

```
28     int numberOfNickels = remainingAmount / 5;
29     remainingAmount = remainingAmount % 5;
30
31     // Find the number of pennies in the remaining amount
```

pennies

```
32     int numberOfPennies = remainingAmount;
33
34     // Display results
```

prepare output

```
35     System.out.println("Your amount " + amount + " consists of \n" +
```

```
36        "\t" + numberOfOneDollars + " dollars\n" +
37        "\t" + numberOfQuarters + " quarters\n" +
38        "\t" + numberOfDimes + " dimes\n" +
39        "\t" + numberOfNickels + " nickels\n" +
40        "\t" + numberOfPennies + " pennies");
41    }
42 }
```

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

variables \ line#	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				
numberOfNickles									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 **remainingAmount**.

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 total amount and obtains the 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 Exercise 2.9).

As shown in the sample run, 0 dimes, 1 nickels, and 1 pennies are displayed in the result. It would be better not to display 0 dimes, and to display 1 nickel and 1 penny using the singular forms of the words. You will learn how to use selection statements to modify this program in the next chapter (see Exercise 3.7).

2.15 The `String` Type

The `char` type represents only one character. To represent a string of characters, use the data type called `String`. For example, the following code declares the message to be a string with value "Welcome to Java".

```
String message = "Welcome to Java";
```

`String` is actually a predefined class in the Java library just like the classes `System`, `JOptionPane`, and `Scanner`. The `String` type is not a primitive type. It is known as a *reference type*. Any Java class can be used as a reference type for a variable. Reference data types will be thoroughly discussed in Chapter 8, "Objects and Classes." For the time being, you need to know only how to declare a `String` variable, how to assign a string to the variable, and how to concatenate strings.

concatenating strings and numbers

As first shown in Listing 2.1, two strings can be concatenated. The plus sign (+) is the concatenation operator if one of the operands is a string. If one of the operands is a nonstring (e.g., a number), the nonstring value is converted into a string and concatenated with the other string. Here are some examples:

```
// Three strings are concatenated
String message = "Welcome " + "to " + "Java";

// String Chapter is concatenated with number 2
String s = "Chapter" + 2; // s becomes Chapter2

// String Supplement is concatenated with character B
String s1 = "Supplement" + 'B'; // s1 becomes SupplementB
```

If neither of the operands is a string, the plus sign (+) is the addition operator that adds two numbers.

The shorthand += operator can also be used for string concatenation. For example, the following code appends the string "and Java is fun" with the string "Welcome to Java" in `message`.

```
message += " and Java is fun";
```

So the new `message` is "Welcome to Java and Java is fun".

Suppose that `i = 1` and `j = 2`, what is the output of the following statement?

```
System.out.println("i + j is " + i + j);
```

The output is "i + j is 12" because `"i + j is "` is concatenated with the value of `i` first. To force `i + j` to be executed first, enclose `i + j` in the parentheses, as follows:

```
System.out.println("i + j is " + (i + j));
```

reading strings

To read a string from the console, invoke the `next()` method on a `Scanner` object. For example, the following code reads three strings from the keyboard:

```
Scanner input = new Scanner(System.in);
System.out.println("Enter three strings: ");
String s1 = input.next();
```

```
String s2 = input.next();
String s3 = input.next();
System.out.println("s1 is " + s1);
System.out.println("s2 is " + s2);
System.out.println("s3 is " + s3);
```

```
Enter a string: Welcome to Java Enter
s1 is Welcome
s2 is to
s3 is Java
```

The `next()` method reads a string that ends with a whitespace character (i.e., `' '`, `'\t'`, `'\f'`, `'\r'`, or `'\n'`).

You can use the `nextLine()` method to read an entire line of text. The `nextLine()` method reads a string that ends with the *Enter* key pressed. For example, the following statements read a line of text.

```
Scanner input = new Scanner(System.in);
System.out.println("Enter a string: ");
String s = input.nextLine();
System.out.println("The string entered is " + s);
```

```
Enter a string: Welcome to Java Enter
The string entered is "Welcome to Java"
```

Important Caution

To *avoid input errors*, do not use `nextLine()` after `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()`, and `next()`. The reasons will be explained in §9.7.3, "How Does `Scanner` Work?"

avoiding input errors

2.16 Programming Style and Documentation

Programming style deals with what programs look like. A program can compile and run properly even if written on only one line, but writing it all on one line would be bad programming style because it would be hard to read. *Documentation* is the body of explanatory remarks and comments pertaining to a program. Programming style and documentation are as important as coding. Good programming style and appropriate documentation reduce the chance of errors and make programs easy to read. So far you have learned some good programming styles. This section summarizes them and gives several guidelines. More detailed guidelines can be found in Supplement I.D, "Java Coding Style Guidelines," on the Companion Website.

programming style

documentation

2.16.1 Appropriate Comments and Comment Styles

Include a summary at the beginning of the program to explain what the program does, its key features, and any unique techniques it uses. In a long program, you should also include comments that introduce each major step and explain anything that is difficult to read. It is important to make comments concise so that they do not crowd the program or make it difficult to read.

In addition to line comment `//` and block comment `/*`, Java supports comments of a special type, referred to as *javadoc comments*. javadoc comments begin with `/**` and end with `*/`. They can be extracted into an HTML file using JDK's `javadoc` command. For more information, see java.sun.com/j2se/javadoc.

javadoc comment

Use javadoc comments (`/** ... */`) for commenting on an entire class or an entire method. These comments must precede the class or the method header in order to be extracted in a javadoc HTML file. For commenting on steps inside a method, use line comments (`//`).

2.16.2 Naming Conventions

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

naming 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 `showInputDialog`.

naming classes

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

naming 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 programs easy to read.

Caution

naming classes

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

Tip

using full descriptive names

Avoid using abbreviations for identifiers. Using complete words is more descriptive. For example, `numberOfStudents` is better than `numStuds`, `numOfStuds`, or `numOfStudents`.

2.16.3 Proper Indentation and Spacing

indent code

A consistent indentation style makes programs clear and easy to read, debug, and maintain. *Indentation* is used to illustrate the structural relationships between a program's components or statements. Java can read the program even if all of the statements are in a straight line, but humans find it easier to read and maintain code that is aligned properly. Indent each subcomponent or statement at least *two* spaces more than the construct within which it is nested.

A single space should be added on both sides of a binary operator, as shown in the following statement:

```
int i= 3+4 * 4;     ← Bad style
int i = 3 + 4 * 4;  ← Good style
```

A single space line should be used to separate segments of the code to make the program easier to read.

2.16.4 Block Styles

A block is a group of statements surrounded by braces. There are two popular styles, *next-line* style and *end-of-line* style, as shown below.

```
public class Test
{
  public static void main(String[] args)
  {
    System.out.println("Block Styles");
  }
}
```

Next-line style

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Block Styles");
  }
}
```

End-of-line style

The next-line style aligns braces vertically and makes programs easy to read, whereas the end-of-line style saves space and may help avoid some subtle programming errors. Both are acceptable block styles. The choice depends on personal or organizational preference. You should use a block style consistently. Mixing styles is not recommended. This book uses the *end-of-line* style to be consistent with the Java API source code.

2.17 Programming Errors

Programming errors are unavoidable, even for experienced programmers. Errors can be categorized into three types: syntax errors, runtime errors, and logic errors.

2.17.1 Syntax Errors

Errors that occur during compilation are called *syntax errors* or *compile errors*. Syntax errors result from errors in code construction, such as mistyping a keyword, omitting some necessary punctuation, or using an opening brace without a corresponding closing brace. These errors are usually easy to detect, because the compiler tells you where they are and what caused them. For example, the following program has a syntax error, as shown in Figure 2.3.

syntax errors

Compile →

FIGURE 2.3 The compiler reports syntax errors.

```
1 // ShowSyntaxErrors.java: The program contains syntax errors
2 public class ShowSyntaxErrors {
3   public static void main(String[] args) {
4     i = 30;
5     System.out.println(i + 4);
6   }
7 }
```

syntax error

Two errors are detected. Both are the result of not declaring variable `i`. Since a single error will often display many lines of compile errors, it is a good practice to start debugging from the top line and work downward. Fixing errors that occur earlier in the program may also fix additional errors that occur later.

2.17.2 Runtime Errors

runtime errors

Runtime errors are errors that cause a program to terminate abnormally. They occur while a program is running if the environment detects an operation that is impossible to carry out. Input errors typically cause runtime errors.

An *input error* occurs when the user enters an unexpected input value that the program cannot handle. For instance, if the program expects to read in a number, but instead the user enters a string, this causes data-type errors to occur in the program. To prevent input errors, the program should prompt the user to enter values of the correct type. It may display a message such as "Please enter an integer" before reading an integer from the keyboard.

Another common source of runtime errors is division by zero. This happens when the divisor is zero for integer divisions. For instance, the following program would cause a runtime error, as shown in Figure 2.4.

FIGURE 2.4 The runtime error causes the program to terminate abnormally.

```
1 // ShowRuntimeErrors.java: Program contains runtime errors
2 public class ShowRuntimeErrors {
3   public static void main(String[] args) {
4     int i = 1 / 0;
5   }
6 }
```

runtime error

2.17.3 Logic Errors

Logic errors occur when a program does not perform the way it was intended to. Errors of this kind occur for many different reasons. For example, suppose you wrote the following program to add `number1` to `number2`.

```
// ShowLogicErrors.java: The program contains a logic error
public class ShowLogicErrors {
  public static void main(String[] args) {
    // Add number1 to number2
    int number1 = 3;
    int number2 = 3;
    number2 += number1 + number2;
    System.out.println("number2 is " + number2);
  }
}
```

The program does not have syntax errors or runtime errors, but it does not print the correct result for `number2`. See if you can find the error.

2.17.4 Debugging

In general, syntax errors are easy to find and easy to correct, because the compiler gives indications as to where the errors came from and why they are wrong. Runtime errors are not difficult to find, either, since the reasons and locations of the errors are displayed on the console when the program aborts. Finding logic errors, on the other hand, can be very challenging.

Logic errors are called *bugs*. The process of finding and correcting errors is called *debugging*. A common approach is to use a combination of methods to narrow down to the part of the program where the bug is located. You can *hand-trace* the program (i.e., catch errors by reading the program), or you can insert print statements in order to show the values of the variables or the execution flow of the program. This approach might work for a short, simple program. But for a large, complex program, the most effective approach is to use a debugger utility.

bugs
debugging
hand traces

Pedagogical NOTE

An IDE not only helps debug errors but also is an effective pedagogical tool. Supplement II shows you how to use a debugger to trace programs and how debugging can help you to learn Java effectively.

learning tool

2.18 (GUI) Getting Input from Input Dialogs

You can obtain input from the console. Alternatively, you may obtain input from an input dialog box by invoking the `JOptionPane.showInputDialog` method, as shown in Figure 2.5.

`JOptionPane` class

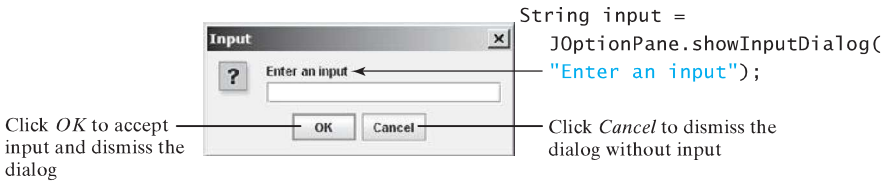


FIGURE 2.5 The input dialog box enables the user to enter a string.

When this method is executed, a dialog is displayed to enable you to enter an input value. After entering a string, click *OK* to accept the input and dismiss the dialog box. The input is returned from the method as a string.

There are several ways to use the `showInputDialog` method. For the time being, you need to know only two ways to invoke it.

`showInputDialog` method

One is to use a statement like this one:

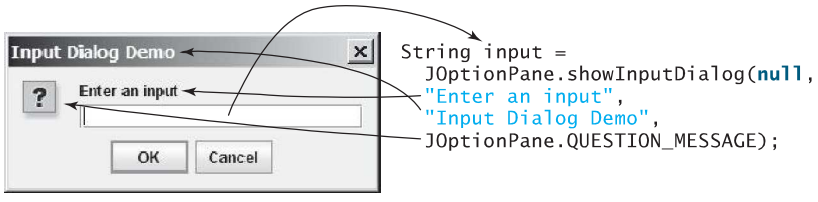
```
JOptionPane.showInputDialog(x);
```

where `x` is a string for the prompting message.

The other is to use a statement such as the following:

```
String string = JOptionPane.showInputDialog(null, x,
  y, JOptionPane.QUESTION_MESSAGE);
```

where `x` is a string for the prompting message and `y` is a string for the title of the input dialog box, as shown in the example below.



2.18.1 Converting Strings to Numbers

The input returned from the input dialog box is a string. If you enter a numeric value such as `123`, it returns `"123"`. You have to convert a string into a number to obtain the input as a number.

`Integer.parseInt` method

To convert a string into an `int` value, use the `parseInt` method in the `Integer` class, as follows:

```
int intValue = Integer.parseInt(intString);
```

where `intString` is a numeric string such as `"123"`.

`Double.parseDouble` method

To convert a string into a `double` value, use the `parseDouble` method in the `Double` class, as follows:

```
double doubleValue = Double.parseDouble(doubleString);
```

where `doubleString` is a numeric string such as `"123.45"`.

The `Integer` and `Double` classes are both included in the `java.lang` package, and thus they are automatically imported.

2.18.2 Using Input Dialog Boxes

Listing 2.8, ComputeLoan.java, reads input from the console. Alternatively, you can use input dialog boxes.

Listing 2.11 gives the complete program. Figure 2.6 shows a sample run of the program.

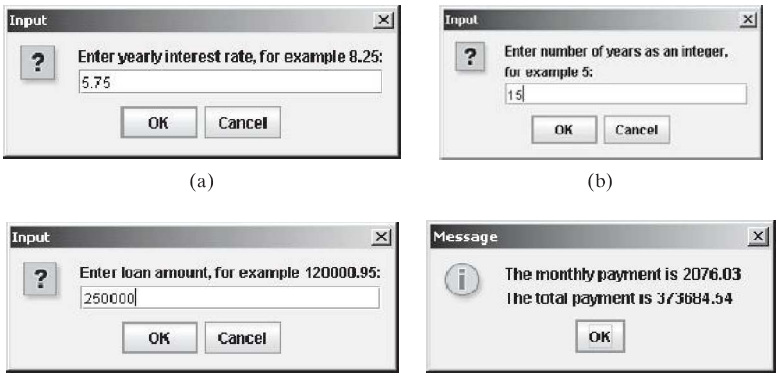


FIGURE 2.6 The program accepts the annual interest rate (a), number of years (b), and loan amount (c), then displays the monthly payment and total payment (d).

LISTING 2.11 ComputeLoanUsingInputDialog.java

```
1  import javax.swing.JOptionPane;
2
3  public class ComputeLoanUsingInputDialog {
4    public static void main(String[] args) {
5      // Enter yearly interest rate
6      String annualInterestRateString = JOptionPane.showInputDialog(
7        "Enter yearly interest rate, for example 8.25:");
8
9      // Convert string to double
10     double annualInterestRate =
11       Double.parseDouble(annualInterestRateString);
12
```

enter interest rate

convert string to double

```
13      // Obtain monthly interest rate
14      double monthlyInterestRate = annualInterestRate / 1200;
15
16      // Enter number of years
17      String numberOfYearsString = JOptionPane.showInputDialog(
18        "Enter number of years as an integer, \nfor example 5:");
19
20      // Convert string to int
21      int numberOfYears = Integer.parseInt(numberOfYearsString);
22
23      // Enter loan amount
24      String loanString = JOptionPane.showInputDialog(
25        "Enter loan amount, for example 120000.95:");
26
27      // Convert string to double
28      double loanAmount = Double.parseDouble(loanString);
29
30      // Calculate payment
31      double monthlyPayment = loanAmount * monthlyInterestRate / (1
32        - 1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12));
33      double totalPayment = monthlyPayment * numberOfYears * 12;
34
35      // Format to keep two digits after the decimal point
36      monthlyPayment = (int)(monthlyPayment * 100) / 100.0;
37      totalPayment = (int)(totalPayment * 100) / 100.0;
38
39      // Display results
40      String output = "The monthly payment is " + monthlyPayment +
41        "\nThe total payment is " + totalPayment;
42      JOptionPane.showMessageDialog(null, output);
43    }
44  }
```

monthlyPayment

totalPayment

preparing output

The **showInputDialog** method in lines 6–7 displays an input dialog. Enter the interest rate as a double value and click *OK* to accept the input. The value is returned as a string that is assigned to the **String** variable **annualInterestRateString**. The **Double.parseDouble(annualInterestRateString)** (line 11) is used to convert the string into a **double** value. If you entered an input other than a numeric value or clicked *Cancel* in the input dialog box, a runtime error would occur. In Chapter 13, "Exception Handling," you will learn how to handle the exception so that the program can continue to run.

Pedagogical Note

For obtaining input you can use **JOptionPane** or **Scanner**, whichever is convenient. For consistency most examples in this book use **Scanner** for getting input. You can easily revise the examples using **JOptionPane** for getting input.

JOptionPane or **Scanner**?

Key Terms

algorithm 24
assignment operator (=) 30
assignment statement 30
backslash (\) 46
byte type 27
casting 41
char type 44
constant 31
data type 25
debugger 55
debugging 55
declaration 30
decrement operator (--) 41
double type 33
encoding 45
final 31

Chapter Summary

1. Identifiers are names for things in a program.
2. An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar signs ($).
3. An identifier must start with a letter or an underscore. It cannot start with a digit.
4. An identifier cannot be a reserved word.
5. An identifier can be of any length.
6. Choosing descriptive identifiers can make programs easy to read.
7. Variables are used to store data in a program
8. To declare a variable is to tell the compiler what type of data a variable can hold.
9. By convention, variable names are in lowercase.
10. In Java, the equal sign (=) is used as the *assignment operator*.
11. A variable declared in a method must be assigned a value before it can be used.
12. A *named constant* (or simply a *constant*) represents permanent data that never changes.
13. A named constant is declared by using the keyword `final`.
14. By convention, constants are named in uppercase.
15. Java provides four integer types (`byte`, `short`, `int`, `long`) that represent integers of four different sizes.

16. Java provides two floating-point types (`float`, `double`) that represent floating-point numbers of two different precisions.

17. Java provides operators that perform numeric operations: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `%` (remainder).

18. Integer arithmetic (`/`) yields an integer result.

19. The numeric operators in a Java expression are applied the same way as in an arithmetic expression.

20. Java provides shorthand operators `+=` (addition assignment), `-=` (subtraction assignment), `*=` (multiplication assignment), `/=` (division assignment), and `%=` (remainder assignment).

21. The increment operator (`++`) and the decrement operator (`--`) increment or decrement a variable by `1`.

22. When evaluating an expression with values of mixed types, Java automatically converts the operands to appropriate types.

23. You can explicitly convert a value from one type to another using the `(type)exp` notation.

24. Casting a variable of a type with a small range to a variable of a type with a larger range is known as *widening a type*.

25. Casting a variable of a type with a large range to a variable of a type with a smaller range is known as *narrowing a type*.

26. Widening a type can be performed automatically without explicit casting. Narrowing a type must be performed explicitly.

27. Character type (`char`) represents a single character.

28. The character `\` is called the escape character.

29. Java allows you to use escape sequences to represent special characters such as `'\t'` and `'\n'`.

30. The characters `' '`, `'\t'`, `'\f'`, `'\r'`, and `'\n'` are known as the whitespace characters.

31. In computer science, midnight of January 1, 1970, is known as the *Unix epoch*.

32. Programming errors can be categorized into three types: syntax errors, runtime errors, and logic errors.

33. Errors that occur during compilation are called *syntax errors* or *compile errors*.

34. *Runtime errors* are errors that cause a program to terminate abnormally.

35. *Logic errors* occur when a program does not perform the way it was intended to.

REVIEW QUESTIONS

Sections 2.2–2.7

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

```
applet, Applet, a++, --a, 4#R, $4, #44, apps
class, public, int, x, y, radius
```

2.2 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 `MILES_PER_KILOMETER` with value `1.609`;
- Step 3: Declare a `double` variable named `kilometers`, multiply miles and `MILES_PER_KILOMETER`, and assign the result to `kilometers`.
- Step 4: Display `kilometers` to the console.

What is `kilometers` after Step 4?

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

Sections 2.8–2.10

2.4 Assume that `int a = 1` and `double d = 1.0`, and that each expression is independent. What are the results of the following expressions?

```
a = 46 / 9;
a = 46 % 9 + 4 * 4 - 2;
a = 45 + 43 % 5 * (23 * 3 % 2);
a %= 3 / a + 3;
d = 4 + d * d + 4;
d += 1.5 * 3 + (++a);
d -= 1.5 * 3 + a++;
```

2.5 Show the result of the following remainders.

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

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

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

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

2.9 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);
```

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

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

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

2.12 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.

2.13 Which of the following are correct literals for floating-point numbers?

```
12.3, 12.3e+2, 23.4e-2, -334.4, 20, 39F, 40D
```

2.14 Identify and fix the errors in the following code:

```
1 public class Test {
2   public void main(string[] args) {
3     int i;
4     int k = 100.0;
5     int j = i + 1;
6
7     System.out.println("j is " + j + " and
8       k is " + k);
9   }
10 }
```

2.15 How do you obtain the current minute using the `System.currentTimeMillis()` method?

Section 2.11

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

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

2.18 Show the following output.

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

Section 2.13

2.19 Use print statements to find out the ASCII code for `'1'`, `'A'`, `'B'`, `'a'`, `'b'`. Use print statements to find out the character for the decimal code `40`, `59`, `79`, `85`, `90`. Use print statements to find out the character for the hexadecimal code `40`, `5A`, `71`, `72`, `7A`.

2.20 Which of the following are correct literals for characters?

```
'1', '\u345dE', '\u3fFa', '\b', \t
```

2.21 How do you display characters `\` and `"`?

2.22 Evaluate the following:

```
int i = '1';
int j = '1' + '2';
int k = 'a';
char c = 90;
```

2.23 Can the following conversions involving casting be allowed? If so, find the converted result.

```
char c = 'A';
i = (int)c;
float f = 1000.34f;
int i = (int)f;

double d = 1000.34;
int i = (int)d;

int i = 97;
char c = (char)i;
```

2.24 Show the output of the following program:

```
public class Test {
  public static void main(String[] args) {
    char x = 'a';
    char y = 'c';

    System.out.println(++x);
    System.out.println(y++);
    System.out.println(x - y);
  }
}
```

Section 2.15

2.25 Show the output of the following statements (write a program to verify your result):

```
System.out.println("1" + 1);
System.out.println('1' + 1);
System.out.println("1" + 1 + 1);
System.out.println("1" + (1 + 1));
System.out.println('1' + 1 + 1);
```

2.26 Evaluate the following expressions (write a program to verify your result):

```
1 + "Welcome " + 1 + 1
1 + "Welcome " + (1 + 1)
1 + "Welcome " + ('\u0001'  + 1)
1 + "Welcome " + 'a'  + 1
```

Sections 2.16–2.17

2.27 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, readInt
```

2.28 Reformat the following program according to the programming style and documentation guidelines. Use the next-line brace style.

```
public class Test
{
  // Main method
  public static void main(String[] args) {
```

```
    /** Print a line */
    System.out.println("2 % 3 = "+2%3);
    }
}
```

2.29 Describe syntax errors, runtime errors, and logic errors.

Section 2.18

2.30 Why do you have to import `JOptionPane` but not the `Math` class?

2.31 How do you prompt the user to enter an input using a dialog box?

2.32 How do you convert a string to an integer? How do you convert a string to a double?

Programming Exercises

Note

Students can run all exercises by downloading `exercise8e.zip` from `www.cs.armstrong.edu/liang/intro8e/exercise8e.zip` and use the command `java -cp exercise8e.zip Exercise`*i_j* to run Exercise*i_j*. For example, to run Exercise2_1, use

sample runs

```
java -cp exercise8e.zip Exercise2_1
```

This will give you an idea how the program runs.

Debugging TIP

The compiler usually gives a reason for a syntax error. If you don't know how to correct it, compare your program closely, character by character, with similar examples in the text.

learn from examples

Sections 2.2–2.9

2.1 (*Converting Celsius to Fahrenheit*) Write a program that reads a Celsius degree in double from the console, then converts it to Fahrenheit and displays the result. The formula for the conversion is as follows:

```
fahrenheit = (9 / 5) * celsius + 32
```

Hint: In Java, `9 / 5` is `1`, but `9.0 / 5` is `1.8`.

Here is a sample run:

```
Enter a degree in Celsius: 43 Enter
43 Celsius is 109.4 Fahrenheit
```

2.2 (*Computing the volume of a cylinder*) Write a program that reads in the radius and length of a cylinder and computes volume using the following formulas:

```
area = radius * radius * π
volume = area * length
```

Here is a sample run:

```
Enter the radius and length of a cylinder: 5.5 12 Enter
The area is 95.0331
The volume is 1140.4
```

2.3 (*Converting feet into meters*) Write a program that reads a number in feet, converts it to meters, and displays the result. One foot is **0.305** meter. Here is a sample run:

```
Enter a value for feet: 16 Enter
16 feet is 4.88 meters
```

2.4 (*Converting pounds into kilograms*) Write a program that converts pounds into kilograms. The program prompts the user to enter a number in pounds, converts it to kilograms, and displays the result. One pound is **0.454** kilograms. Here is a sample run:

```
Enter a number in pounds: 55.5 Enter
55.5 pounds is 25.197 kilograms
```

2.5* (*Financial application: calculating tips*) Write a program that reads the subtotal and the gratuity rate, then computes the gratuity and total. For example, if the user enters **10** for subtotal and **15%** for gratuity rate, the program displays **$1.5** as gratuity and **$11.5** as total. Here is a sample run:

```
Enter the subtotal and a gratuity rate: 15.69 15 Enter
The gratuity is 2.35 and total is 18.04
```

2.6** (*Summing the digits in an integer*) Write a program that reads an integer between **0** and **1000** and adds all the digits in the integer. For example, if an integer is **932**, the sum of all its digits is **14**.

Hint: Use the **%** operator to extract digits, and use the **/** operator to remove the extracted digit. For instance, **932 % 10 = 2** and **932 / 10 = 93**.

Here is a sample run:

```
Enter a number between 0 and 1000: 999 Enter
The sum of the digits is 27
```

2.7* (*Finding the number of years*) Write a program that prompts the user to enter the minutes (e.g., 1 billion) and displays the number of years and days for the minutes. For simplicity, assume a year has **365** days. Here is a sample run:

```
Enter the number of minutes: 1000000000 Enter
1000000000 minutes is approximately 1902 years and 214 days.
```

Section 2.13

2.8* (*Finding the character of an ASCII code*) Write a program that receives an ASCII code (an integer between `0` and `128`) and displays its character. For example, if the user enters `97`, the program displays character `a`. Here is a sample run:

```
Enter an ASCII code: 69 Enter
The character for ASCII code 69 is E
```

2.9* (*Financial application: monetary units*) Rewrite Listing 2.10, ComputeChange.java, to fix the possible loss of accuracy when converting a `double` value to an `int` value. Enter the input as an integer whose last two digits represent the cents. For example, the input `1156` represents `11` dollars and `56` cents.

Section 2.18

2.10* (*Using the GUI input*) Rewrite Listing 2.10, ComputeChange.java, using the GUI input and output.

Comprehensive

2.11* (*Financial application: payroll*) Write a program that reads the following information and prints a payroll statement:

Employee's name (e.g., Smith)

Number of hours worked in a week (e.g., 10)

Hourly pay rate (e.g., 6.75)

Federal tax withholding rate (e.g., 20%)

State tax withholding rate (e.g., 9%)

Write this program in two versions: (a) Use dialog boxes to obtain input and display output; (b) Use console input and output. A sample run of the console input and output is shown below:

```
Enter employee's name: Smith Enter
Enter number of hours worked in a week: 10 Enter
Enter hourly pay rate: 6.75 Enter
Enter federal tax withholding rate: 0.20 Enter
Enter state tax withholding rate: 0.09 Enter
Employee Name: Smith
Hours Worked:  10.0
Pay Rate:  $6.75
Gross Pay:  $67.5
Deductions:
  Federal Withholding (20.0%):  $13.5
  State Withholding (9.0%):  $6.07
  Total Deduction:  $19.57
Net Pay:  $47.92
```

2.12* (*Financial application: calculating interest*) If you know the balance and the annual percentage interest rate, you can compute the interest on the next monthly payment using the following formula:

$$\text{interest} = \text{balance} \times (\text{annualInterestRate} / 1200)$$

Write a program that reads the balance and the annual percentage interest rate and displays the interest for the next month in two versions: (a) Use dialog boxes to obtain input and display output; (b) Use console input and output. Here is a sample run:

```
Enter balance and interest rate (e.g., 3 for 3%): 1000 3.5 Enter
The interest is 2.91667
```

2.13* (*Financial application: calculating the future investment value*) Write a program that reads in investment amount, annual interest rate, and number of years, and displays the future investment value using the following formula:

$$futureInvestmentValue = investmentAmount \times (1 + monthlyInterestRate)^{numberOfYears*12}$$

For example, if you enter amount `1000`, annual interest rate `3.25%`, and number of years `1`, the future investment value is `1032.98`.

Hint: Use the `Math.pow(a, b)` method to compute `a` raised to the power of `b`.

Here is a sample run:

```
Enter investment amount: 1000 Enter
Enter monthly interest rate: 4.25 Enter
Enter number of years: 1 Enter
Accumulated value is 1043.34
```

Video Note
Compute BMI

2.14* (*Health application: computing BMI*) Body Mass Index (BMI) is a measure of health on weight. It can be calculated by taking your weight in kilograms and dividing by the square of your height in meters. Write a program that prompts the user to enter a weight in pounds and height in inches and display the BMI. Note that one pound is `0.45359237` kilograms and one inch is `0.0254` meters. Here is a sample run:

```
Enter weight in pounds: 95.5 Enter
Enter height in inches: 50 Enter
BMI is 26.8573
```

2.15** (*Financial application: compound value*) Suppose you save `$100` *each* month into a savings account with the annual interest rate 5%. So, the monthly interest rate is 0.05 / 12 = 0.00417. After the first month, the value in the account becomes

```
100 * (1 + 0.00417) = 100.417
```

After the second month, the value in the account becomes

```
(100 + 100.417) * (1 + 0.00417) = 201.252
```

After the third month, the value in the account becomes

```
(100 + 201.252) * (1 + 0.00417) = 302.507
```

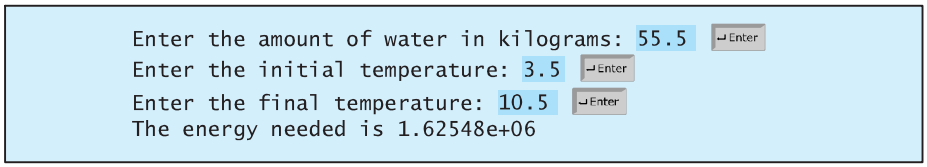
and so on.

Write a program to display the account value after the sixth month. (In Exercise 4.30, you will use a loop to simplify the code and display the account value for any month.)

2.16 (*Science: calculating energy*) Write a program that calculates the energy needed to heat water from an initial temperature to a final temperature. Your program should prompt the user to enter the amount of water in kilograms and the initial and final temperatures of the water. The formula to compute the energy is

```
Q = M * (final temperature - initial temperature) * 4184
```

where `M` is the weight of water in kilograms, temperatures are in degrees Celsius, and energy `Q` is measured in joules. Here is a sample run:

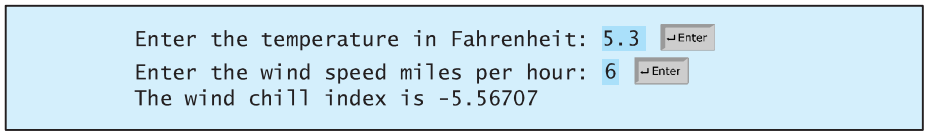


2.17* (*Science: wind-chill temperature*) How cold is it outside? The temperature alone is not enough to provide the answer. Other factors including wind speed, relative humidity, and sunshine play important roles in determining coldness outside. In 2001, the National Weather Service (NWS) implemented the new wind-chill temperature to measure the coldness using temperature and wind speed. The formula is given as follows:

$$t_{wc} = 35.74 + 0.6215t_a - 35.75v^{0.16} + 0.4275t_a v^{0.16}$$

where t_a is the outside temperature measured in degrees Fahrenheit and v is the speed measured in miles per hour. t_{wc} is the wind-chill temperature. The formula cannot be used for wind speeds below 2 mph or temperatures below $-58°$F or above 41°F.

Write a program that prompts the user to enter a temperature between $-58°$F and 41°F and a wind speed greater than or equal to `2` and displays the wind-chill temperature. Use `Math.pow(a, b)` to compute $v^{0.16}$. Here is a sample run:



2.18 (*Printing a table*) Write a program that displays the following table:

```
a     b     pow(a, b)
1     2     1
2     3     8
3     4     81
4     5     1024
5     6     15625
```

2.19 (*Random character*) Write a program that displays a random uppercase letter using the `System.CurrentTimeMillis()` method.

2.20 (*Geometry: distance of two points*) Write a program that prompts the user to enter two points `(x1, y1)` and `(x2, y2)` and displays their distances. The formula for computing the distance is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Note you can use the `Math.pow(a, 0.5)` to compute $\sqrt{a}$. Here is a sample run:

```
Enter x1 and y1: 1.5 -3.4 Enter
Enter x2 and y2: 4 5 Enter
The distance of the two points is 8.764131445842194
```

2.21* (*Geometry: area of a triangle*) Write a program that prompts the user to enter three points `(x1, y1)`, `(x2, y2)`, (`x3`, `y3`) of a triangle and displays its area. The formula for computing the area of a triangle is

$$s = (side1 + side2 + side3)/2;$$

$$area = \sqrt{s(s - side1)(s - side2)(s - side3)}$$

Here is a sample run.

```
Enter three points for a triangle: 1.5 -3.4 4.6 5 9.5 -3.4 Enter
The area of the triangle is 33.6
```

2.22 (*Geometry: area of a hexagon*) Write a program that prompts the user to enter the side of a hexagon and displays its area. The formula for computing the area of a hexagon is

$$Area = \frac{3\sqrt{3}}{2}s^2,$$

where s is the length of a side. Here is a sample run:

```
Enter the side: 5.5 Enter
The area of the hexagon is 78.5895
```

2.23 (*Physics: acceleration*) Average acceleration is defined as the change of velocity divided by the time taken to make the change, as shown in the following formula:

$$a = \frac{v_1 - v_0}{t}$$

Write a program that prompts the user to enter the starting velocity v_0 in meters/second, the ending velocity v_1 in meters/second, and the time span t in seconds, and displays the average acceleration. Here is a sample run:

```
Enter v0, v1, and t: 5.5 50.9 4.5 Enter
The average acceleration is 10.0889
```

2.24 (*Physics: finding runway length*) Given an airplane's acceleration a and take-off speed v, you can compute the minimum runway length needed for an airplane to take off using the following formula:

$$length = \frac{v^2}{2a}$$

Write a program that prompts the user to enter v in meters/second (m/s) and the acceleration a in meters/second squared (m/s^2), and displays the minimum runway length. Here is a sample run:

```
Enter v and a: 60 3.5 Enter
The minimum runway length for this airplane is 514.286
```

2.25* (*Current time*) Listing 2.6, ShowCurrentTime.java, gives a program that displays the current time in GMT. Revise the program so that it prompts the user to enter the time zone offset to GMT and displays the time in the specified time zone. Here is a sample run:

```
Enter the time zone offset to GMT: -5 Enter
The current time is 4:50:34
```